



SymPy Documentation

Release 1.11.1

SymPy Development Team

August 27, 2022

CONTENTS

1	Installation	1
1.1	Anaconda	1
1.2	Git	1
1.3	Other Methods	2
1.4	Run SymPy	2
1.5	mpmath	2
1.6	Questions	3
2	Tutorials	5
2.1	Introductory Tutorial	5
3	How-to Guides	71
3.1	Assumptions	71
3.2	Symbolic and fuzzy booleans	95
3.3	Writing Custom Functions	102
3.4	Solve Equations	131
3.5	Citing SymPy	139
4	Explanations	141
4.1	Gotchas and Pitfalls	141
4.2	SymPy Special Topics	157
4.3	List of active deprecations	163
5	API Reference	185
5.1	Basics	185
5.2	Code Generation	185
5.3	Logic	185
5.4	Matrices	185
5.5	Number Theory	186
5.6	Physics	186
5.7	Utilities	186
5.8	Topics	186
6	Contributing	2983
6.1	Development Environment Setup	2983
6.2	Dependencies	2985
6.3	Build the Documentation	2992
6.4	Debugging	2996
6.5	SymPy Docstrings Style Guide	2997
6.6	Documentation Style Guide	3010
6.7	Making a Contribution	3017

6.8 Deprecation Policy	3017
Bibliography	3027
Python Module Index	3065
Index	3069

INSTALLATION

The SymPy CAS can be installed on virtually any computer with Python. SymPy does require `mpmath` Python library to be installed first. The recommended method of installation is through Anaconda, which includes `mpmath`, as well as several other useful libraries. Alternatively, some Linux distributions have SymPy packages available.

SymPy officially supports Python 3.8, 3.9, 3.10, and PyPy.

1.1 Anaconda

Anaconda is a free Python distribution from Continuum Analytics that includes SymPy, Matplotlib, IPython, NumPy, and many more useful packages for scientific computing. This is recommended because many nice features of SymPy are only enabled when certain libraries are installed. For example, without Matplotlib, only simple text-based plotting is enabled. With the IPython notebook or qtconsole, you can get nicer $\text{\LaTeX}$ printing by running `init_printing()`.

If you already have Anaconda and want to update SymPy to the latest version, use:

```
conda update sympy
```

1.2 Git

If you wish to contribute to SymPy or like to get the latest updates as they come, install SymPy from git. To download the repository, execute the following from the command line:

```
git clone https://github.com/sympy/sympy.git
```

To update to the latest version, go into your repository and execute:

```
git pull origin master
```

If you want to install SymPy, but still want to use the git version, you can run from your repository:

```
python setup.py develop
```

This will cause the installed version to always point to the version in the git directory.

1.3 Other Methods

You may also install SymPy using pip or from source. In addition, most Linux and Python distributions have some SymPy version available to install using their package manager. Here is a list of several such Python distributions:

- [Anaconda](#)
- [Enthought Canopy](#)
- [ActivePython](#)
- [Spack](#)

1.4 Run SymPy

After installation, it is best to verify that your freshly-installed SymPy works. To do this, start up Python and import the SymPy libraries:

```
$ python
>>> from sympy import *
```

From here, execute some simple SymPy statements like the ones below:

```
>>> x = Symbol('x')
>>> limit(sin(x)/x, x, 0)
1
>>> integrate(1/x, x)
log(x)
```

For a starter guide on using SymPy effectively, refer to the [Introductory Tutorial](#) (page 5).

1.5 mpmath

Versions of SymPy prior to 1.0 included [mpmath](#), but it now depends on it as an external dependency. If you installed SymPy with Anaconda, it will already include mpmath. Use:

```
conda install mpmath
```

to ensure that it is installed.

If you do not wish to use Anaconda, you can use `pip install mpmath`.

If you use mpmath via `sympy.mpmath` in your code, you will need to change this to use just mpmath. If you depend on code that does this that you cannot easily change, you can work around it by doing:

```
import sys
import mpmath
sys.modules['sympy.mpmath'] = mpmath
```

before the code that imports `sympy.mpmath`. It is recommended to change code that uses `sympy.mpmath` to use mpmath directly wherever possible.

1.6 Questions

If you have a question about installation or SymPy in general, feel free to visit our chat on [Gitter](#). In addition, our [mailing list](#) is an excellent source of community support.

If you think there's a bug or you would like to request a feature, please open an [issue ticket](#).

TUTORIALS

Tutorials are the best place to start for anyone new to SymPy or one of SymPy's features.

2.1 Introductory Tutorial

If you are new to SymPy, start here.

2.1.1 Introductory Tutorial

This tutorial aims to give an introduction to SymPy for someone who has not used the library before. Many features of SymPy will be introduced in this tutorial, but they will not be exhaustive. In fact, virtually every functionality shown in this tutorial will have more options or capabilities than what will be shown. The rest of the SymPy documentation serves as API documentation, which extensively lists every feature and option of each function.

These are the goals of this tutorial:

- To give a guide, suitable for someone who has never used SymPy (but who has used Python and knows the necessary mathematics).
- To be written in a narrative format, which is both easy and fun to follow. It should read like a book.
- To give insightful examples and exercises, to help the reader learn and to make it entertaining to work through.
- To introduce concepts in a logical order.
- To use good practices and idioms, and avoid antipatterns. Functions or methodologies that tend to lead to antipatterns are avoided. Features that are only useful to advanced users are not shown.
- To be consistent. If there are multiple ways to do it, only the best way is shown.
- To avoid unnecessary duplication, it is assumed that previous sections of the tutorial have already been read.

Feedback on this tutorial, or on SymPy in general is always welcome. Just write to our [mailing list](#).

Content

Preliminaries

This tutorial assumes that the reader already knows the basics of the Python programming language. If you do not, the [official Python tutorial](#) is excellent.

This tutorial assumes a decent mathematical background. Most examples require knowledge lower than a calculus level, and some require knowledge at a calculus level. Some of the advanced features require more than this. If you come across a section that uses some mathematical function you are not familiar with, you can probably skip over it, or replace it with a similar one that you are more familiar with. Or look up the function on Wikipedia and learn something new. Some important mathematical concepts that are not common knowledge will be introduced as necessary.

Installation

You will need to install SymPy first. See the [installation guide](#) (page 1).

Exercises

This tutorial was the basis for a tutorial given at the 2013 SciPy conference in Austin, TX. The website for that tutorial is [here](#). It has links to videos, materials, and IPython notebook exercises. The IPython notebook exercises in particular are recommended to anyone going through this tutorial.

Introduction

What is Symbolic Computation?

Symbolic computation deals with the computation of mathematical objects symbolically. This means that the mathematical objects are represented exactly, not approximately, and mathematical expressions with unevaluated variables are left in symbolic form.

Let's take an example. Say we wanted to use the built-in Python functions to compute square roots. We might do something like this

```
>>> import math
>>> math.sqrt(9)
3.0
```

9 is a perfect square, so we got the exact answer, 3. But suppose we computed the square root of a number that isn't a perfect square

```
>>> math.sqrt(8)
2.82842712475
```

Here we got an approximate result. 2.82842712475 is not the exact square root of 8 (indeed, the actual square root of 8 cannot be represented by a finite decimal, since it is an irrational number). If all we cared about was the decimal form of the square root of 8, we would be done.

But suppose we want to go further. Recall that $\sqrt{8} = \sqrt{4 \cdot 2} = 2\sqrt{2}$. We would have a hard time deducing this from the above result. This is where symbolic computation comes in. With

a symbolic computation system like SymPy, square roots of numbers that are not perfect squares are left unevaluated by default

```
>>> import sympy
>>> sympy.sqrt(3)
sqrt(3)
```

Furthermore—and this is where we start to see the real power of symbolic computation—symbolic results can be symbolically simplified.

```
>>> sympy.sqrt(8)
2*sqrt(2)
```

A More Interesting Example

The above example starts to show how we can manipulate irrational numbers exactly using SymPy. But it is much more powerful than that. Symbolic computation systems (which by the way, are also often called computer algebra systems, or just CASs) such as SymPy are capable of computing symbolic expressions with variables.

As we will see later, in SymPy, variables are defined using symbols. Unlike many symbolic manipulation systems, variables in SymPy must be defined before they are used (the reason for this will be discussed in the [next section](#) (page 10)).

Let us define a symbolic expression, representing the mathematical expression $x + 2y$.

```
>>> from sympy import symbols
>>> x, y = symbols('x y')
>>> expr = x + 2*y
>>> expr
x + 2*y
```

Note that we wrote $x + 2y$ just as we would if x and y were ordinary Python variables. But in this case, instead of evaluating to something, the expression remains as just $x + 2y$. Now let us play around with it:

```
>>> expr + 1
x + 2*y + 1
>>> expr - x
2*y
```

Notice something in the above example. When we typed $\text{expr} - x$, we did not get $x + 2y - x$, but rather just $2y$. The x and the $-x$ automatically canceled one another. This is similar to how $\text{sqrt}(8)$ automatically turned into $2\sqrt{2}$ above. This isn't always the case in SymPy, however:

```
>>> x*expr
x*(x + 2*y)
```

Here, we might have expected $x(x + 2y)$ to transform into $x^2 + 2xy$, but instead we see that the expression was left alone. This is a common theme in SymPy. Aside from obvious simplifications like $x - x = 0$ and $\sqrt{8} = 2\sqrt{2}$, most simplifications are not performed automatically. This is because we might prefer the factored form $x(x + 2y)$, or we might prefer the expanded form $x^2 + 2xy$. Both forms are useful in different circumstances. In SymPy, there are functions to go from one form to the other

```
>>> from sympy import expand, factor
>>> expanded_expr = expand(x*expr)
>>> expanded_expr
x**2 + 2*x*y
>>> factor(expanded_expr)
x*(x + 2*y)
```

The Power of Symbolic Computation

The real power of a symbolic computation system such as SymPy is the ability to do all sorts of computations symbolically. SymPy can simplify expressions, compute derivatives, integrals, and limits, solve equations, work with matrices, and much, much more, and do it all symbolically. It includes modules for plotting, printing (like 2D pretty printed output of math formulas, or $\text{\LaTeX}$), code generation, physics, statistics, combinatorics, number theory, geometry, logic, and more. Here is a small sampling of the sort of symbolic power SymPy is capable of, to whet your appetite.

```
>>> from sympy import *
>>> x, t, z, nu = symbols('x t z nu')
```

This will make all further examples pretty print with unicode characters.

```
>>> init_printing(use_unicode=True)
```

Take the derivative of $\sin(x)e^x$.

```
>>> diff(sin(x)*exp(x), x)
      x      x
e  ·sin(x) + e  ·cos(x)
```

Compute $\int (e^x \sin(x) + e^x \cos(x)) dx$.

```
>>> integrate(exp(x)*sin(x) + exp(x)*cos(x), x)
      x
e  ·sin(x)
```

Compute $\int_{-\infty}^{\infty} \sin(x^2) dx$.

```
>>> integrate(sin(x**2), (x, -oo, oo))
√2·√π
-----
2
```

Find $\lim_{x \rightarrow 0} \frac{\sin(x)}{x}$.

```
>>> limit(sin(x)/x, x, 0)
1
```

Solve $x^2 - 2 = 0$.

```
>>> solve(x**2 - 2, x)
[-√2, √2]
```


Solve the differential equation $y'' - y = e^t$.

```
>>> y = Function('y')
>>> dsolve(Eq(y(t).diff(t, t) - y(t), exp(t)), y(t))
```

$$y(t) = C_2 \cdot e^{-t} + \left(C_1 + \frac{t}{2} \right) \cdot e^t$$

Find the eigenvalues of $\begin{bmatrix} 1 & 2 \\ 2 & 2 \end{bmatrix}$.

```
>>> Matrix([[1, 2], [2, 2]]).eigenvals()
```

$$\left\{ -\frac{3}{2} - \frac{\sqrt{17}}{2}: 1, -\frac{3}{2} + \frac{\sqrt{17}}{2}: 1 \right\}$$

Rewrite the Bessel function $J_\nu(z)$ in terms of the spherical Bessel function $j_\nu(z)$.

```
>>> besselj(nu, z).rewrite(jn)
```

$$\frac{\sqrt{2} \cdot \sqrt{z} \cdot j_n\left(\sqrt{z} - \frac{1}{2}\right)}{\sqrt{\pi}}$$

Print $\int_0^\pi \cos^2(x) dx$ using L^AT_EX.

```
>>> latex(Integral(cos(x)**2, (x, 0, pi)))
```

$$\int\limits_0^{\pi} \cos^2\left(x\right) dx$$

Why SymPy?

There are many computer algebra systems out there. [This](#) Wikipedia article lists many of them. What makes SymPy a better choice than the alternatives?

First off, SymPy is completely free. It is open source, and licensed under the liberal BSD license, so you can modify the source code and even sell it if you want to. This contrasts with popular commercial systems like Maple or Mathematica that cost hundreds of dollars in licenses.

Second, SymPy uses Python. Most computer algebra systems invent their own language. Not SymPy. SymPy is written entirely in Python, and is executed entirely in Python. This means that if you already know Python, it is much easier to get started with SymPy, because you already know the syntax (and if you don't know Python, it is really easy to learn). We already know that Python is a well-designed, battle-tested language. The SymPy developers are confident in their abilities in writing mathematical software, but programming language design is a completely different thing. By reusing an existing language, we are able to focus on those things that matter: the mathematics.

Another computer algebra system, Sage also uses Python as its language. But Sage is large, with a download of over a gigabyte. An advantage of SymPy is that it is lightweight. In addition to being relatively small, it has no dependencies other than Python, so it can be used almost anywhere easily. Furthermore, the goals of Sage and the goals of SymPy are different. Sage aims to be a full featured system for mathematics, and aims to do so by compiling all the major open source mathematical systems together into one. When you call some function in Sage, such as `integrate`, it calls out to one of the open source packages that it includes. In

fact, SymPy is included in Sage. SymPy on the other hand aims to be an independent system, with all the features implemented in SymPy itself.

A final important feature of SymPy is that it can be used as a library. Many computer algebra systems focus on being usable in interactive environments, but if you wish to automate or extend them, it is difficult to do. With SymPy, you can just as easily use it in an interactive Python environment or import it in your own Python application. SymPy also provides APIs to make it easy to extend it with your own custom functions.

Gotchas

To begin, we should make something about SymPy clear. SymPy is nothing more than a Python library, like NumPy, Django, or even modules in the Python standard library `sys` or `re`. What this means is that SymPy does not add anything to the Python language. Limitations that are inherent in the Python language are also inherent in SymPy. It also means that SymPy tries to use Python idioms whenever possible, making programming with SymPy easy for those already familiar with programming with Python. As a simple example, SymPy uses Python syntax to build expressions. Implicit multiplication (like $3x$ or $3 \cdot x$) is not allowed in Python, and thus not allowed in SymPy. To multiply 3 and x , you must type $3*x$ with the `*`.

Symbols

One consequence of this fact is that SymPy can be used in any environment where Python is available. We just import it, like we would any other library:

```
>>> from sympy import *
```

This imports all the functions and classes from SymPy into our interactive Python session. Now, suppose we start to do a computation.

```
>>> x + 1
Traceback (most recent call last):
...
NameError: name 'x' is not defined
```

Oops! What happened here? We tried to use the variable x , but it tells us that x is not defined. In Python, variables have no meaning until they are defined. SymPy is no different. Unlike many symbolic manipulation systems you may have used, in SymPy, variables are not defined automatically. To define variables, we must use `symbols`.

```
>>> x = symbols('x')
>>> x + 1
x + 1
```

`symbols` takes a string of variable names separated by spaces or commas, and creates Symbols out of them. We can then assign these to variable names. Later, we will investigate some convenient ways we can work around this issue. For now, let us just define the most common variable names, x , y , and z , for use through the rest of this section

```
>>> x, y, z = symbols('x y z')
```

As a final note, we note that the name of a Symbol and the name of the variable it is assigned to need not have anything to do with one another.

```
>>> a, b = symbols('b a')
>>> a
b
>>> b
a
```

Here we have done the very confusing thing of assigning a Symbol with the name `a` to the variable `b`, and a Symbol of the name `b` to the variable `a`. Now the Python variable named `a` points to the SymPy Symbol named `b`, and vice versa. How confusing. We could have also done something like

```
>>> crazy = symbols('unrelated')
>>> crazy + 1
unrelated + 1
```

This also shows that Symbols can have names longer than one character if we want.

Usually, the best practice is to assign Symbols to Python variables of the same name, although there are exceptions: Symbol names can contain characters that are not allowed in Python variable names, or may just want to avoid typing long names by assigning Symbols with long names to single letter Python variables.

To avoid confusion, throughout this tutorial, Symbol names and Python variable names will always coincide. Furthermore, the word “Symbol” will refer to a SymPy Symbol and the word “variable” will refer to a Python variable.

Finally, let us be sure we understand the difference between SymPy Symbols and Python variables. Consider the following:

```
x = symbols('x')
expr = x + 1
x = 2
print(expr)
```

What do you think the output of this code will be? If you thought 3, you’re wrong. Let’s see what really happens

```
>>> x = symbols('x')
>>> expr = x + 1
>>> x = 2
>>> print(expr)
x + 1
```

Changing `x` to 2 had no effect on `expr`. This is because `x = 2` changes the Python variable `x` to 2, but has no effect on the SymPy Symbol `x`, which was what we used in creating `expr`. When we created `expr`, the Python variable `x` was a Symbol. After we created it, we changed the Python variable `x` to 2. But `expr` remains the same. This behavior is not unique to SymPy. All Python programs work this way: if a variable is changed, expressions that were already created with that variable do not change automatically. For example

```
>>> x = 'abc'
>>> expr = x + 'def'
>>> expr
'abcdef'
>>> x = 'ABC'
```

(continues on next page)

(continued from previous page)

```
>>> expr
'abcdef'
```

Quick Tip

To change the value of a Symbol in an expression, use `subs`

```
>>> x = symbols('x')
>>> expr = x + 1
>>> expr.subs(x, 2)
3
```

In this example, if we want to know what `expr` is with the new value of `x`, we need to reevaluate the code that created `expr`, namely, `expr = x + 1`. This can be complicated if several lines created `expr`. One advantage of using a symbolic computation system like SymPy is that we can build a symbolic representation for `expr`, and then substitute `x` with values. The correct way to do this in SymPy is to use `subs`, which will be discussed in more detail later.

```
>>> x = symbols('x')
>>> expr = x + 1
>>> expr.subs(x, 2)
3
```

Equals signs

Another very important consequence of the fact that SymPy does not extend Python syntax is that `=` does not represent equality in SymPy. Rather it is Python variable assignment. This is hard-coded into the Python language, and SymPy makes no attempts to change that.

You may think, however, that `==`, which is used for equality testing in Python, is used for SymPy as equality. This is not quite correct either. Let us see what happens when we use `==`.

```
>>> x + 1 == 4
False
```

Instead of treating `x + 1 == 4` symbolically, we just got `False`. In SymPy, `==` represents exact structural equality testing. This means that `a == b` means that we are *asking* if $a = b$. We always get a `bool` as the result of `==`. There is a separate object, called `Eq`, which can be used to create symbolic equalities

```
>>> Eq(x + 1, 4)
Eq(x + 1, 4)
```

There is one additional caveat about `==` as well. Suppose we want to know if $(x+1)^2 = x^2+2x+1$. We might try something like this:

```
>>> (x + 1)**2 == x**2 + 2*x + 1
False
```

We got `False` again. However, $(x+1)^2$ *does* equal x^2+2x+1 . What is going on here? Did we find a bug in SymPy, or is it just not powerful enough to recognize this basic algebraic fact?

Recall from above that `==` represents *exact* structural equality testing. “Exact” here means that two expressions will compare equal with `==` only if they are exactly equal structurally. Here, $(x + 1)^2$ and $x^2 + 2x + 1$ are not the same structurally. One is the power of an addition of two terms, and the other is the addition of three terms.

It turns out that when using SymPy as a library, having `==` test for exact structural equality is far more useful than having it represent symbolic equality, or having it test for mathematical equality. However, as a new user, you will probably care more about the latter two. We have already seen an alternative to representing equalities symbolically, `Eq`. To test if two things are equal, it is best to recall the basic fact that if $a = b$, then $a - b = 0$. Thus, the best way to check if $a = b$ is to take $a - b$ and simplify it, and see if it goes to 0. We will learn [later](#) (page 25) that the function to do this is called `simplify`. This method is not infallible—in fact, it can be [theoretically proven](#) that it is impossible to determine if two symbolic expressions are identically equal in general—but for most common expressions, it works quite well.

```
>>> a = (x + 1)**2
>>> b = x**2 + 2*x + 1
>>> simplify(a - b)
0
>>> c = x**2 - 2*x + 1
>>> simplify(a - c)
4*x
```

There is also a method called `equals` that tests if two expressions are equal by evaluating them numerically at random points.

```
>>> a = cos(x)**2 - sin(x)**2
>>> b = cos(2*x)
>>> a.equals(b)
True
```

Two Final Notes: `^` and `/`

You may have noticed that we have been using `**` for exponentiation instead of the standard `^`. That’s because SymPy follows Python’s conventions. In Python, `^` represents logical exclusive or. SymPy follows this convention:

```
>>> True ^ False
True
>>> True ^ True
False
>>> Xor(x, y)
x ^ y
```

Finally, a small technical discussion on how SymPy works is in order. When you type something like `x + 1`, the SymPy Symbol `x` is added to the Python int `1`. Python’s operator rules then allow SymPy to tell Python that SymPy objects know how to be added to Python ints, and so `1` is automatically converted to the SymPy Integer object.

This sort of operator magic happens automatically behind the scenes, and you rarely need to even know that it is happening. However, there is one exception. Whenever you combine a SymPy object and a SymPy object, or a SymPy object and a Python object, you get a SymPy object, but whenever you combine two Python objects, SymPy never comes into play, and so you get a Python object.

```
>>> type(Integer(1) + 1)
<class 'sympy.core.numbers.Integer'>
>>> type(1 + 1)
<... 'int'>
```

This is usually not a big deal. Python ints work much the same as SymPy Integers, but there is one important exception: division. In SymPy, the division of two Integers gives a Rational:

```
>>> Integer(1)/Integer(3)
1/3
>>> type(Integer(1)/Integer(3))
<class 'sympy.core.numbers.Rational'>
```

But in Python `/` represents either integer division or floating point division, depending on whether you are in Python 2 or Python 3, and depending on whether or not you have run `from __future__ import division` in Python 2 which is no longer supported from versions above SymPy 1.5.1:

```
>>> from __future__ import division
>>> 1/2
0.5
```

To avoid this, we can construct the rational object explicitly

```
>>> Rational(1, 2)
1/2
```

This problem also comes up whenever we have a larger symbolic expression with `int/int` in it. For example:

```
>>> x + 1/2
x + 0.5
```

This happens because Python first evaluates `1/2` into `0.5`, and then that is cast into a SymPy type when it is added to `x`. Again, we can get around this by explicitly creating a Rational:

```
>>> x + Rational(1, 2)
x + 1/2
```

There are several tips on avoiding this situation in the [Gotchas and Pitfalls](#) (page 141) document.

Further Reading

For more discussion on the topics covered in this section, see [Gotchas and Pitfalls](#) (page 141).

SymPy Features

This section discusses the common and advanced SymPy operations and features.

Content

Basic Operations

Here we discuss some of the most basic operations needed for expression manipulation in SymPy. Some more advanced operations will be discussed later in the [advanced expression manipulation](#) (page 61) section.

```
>>> from sympy import *
>>> x, y, z = symbols("x y z")
```

Substitution

One of the most common things you might want to do with a mathematical expression is substitution. Substitution replaces all instances of something in an expression with something else. It is done using the `subs` method. For example

```
>>> expr = cos(x) + 1
>>> expr.subs(x, y)
cos(y) + 1
```

Substitution is usually done for one of two reasons:

1. Evaluating an expression at a point. For example, if our expression is $\cos(x) + 1$ and we want to evaluate it at the point $x = 0$, so that we get $\cos(0) + 1$, which is 2.

```
>>> expr.subs(x, 0)
2
```

2. Replacing a subexpression with another subexpression. There are two reasons we might want to do this. The first is if we are trying to build an expression that has some symmetry, such as x^{x^x} . To build this, we might start with $x^{**}y$, and replace y with $x^{**}y$. We would then get $x^{**}(x^{**}y)$. If we replaced y in this new expression with $x^{**}x$, we would get $x^{**}(x^{**}(x^{**}x))$, the desired expression.

```
>>> expr = x**y
>>> expr
x**y
>>> expr = expr.subs(y, x**y)
>>> expr
x**(x**y)
>>> expr = expr.subs(y, x**x)
>>> expr
x**(x**(x**x))
```

The second is if we want to perform a very controlled simplification, or perhaps a simplification that SymPy is otherwise unable to do. For example, say we have $\sin(2x) + \cos(2x)$, and we want to replace $\sin(2x)$ with $2\sin(x)\cos(x)$. As we will learn later, the function

`expand_trig` does this. However, this function will also expand $\cos(2x)$, which we may not want. While there are ways to perform such precise simplification, and we will learn some of them in the [advanced expression manipulation](#) (page 61) section, an easy way is to just replace $\sin(2x)$ with $2\sin(x)\cos(x)$.

```
>>> expr = sin(2*x) + cos(2*x)
>>> expand_trig(expr)
2*sin(x)*cos(x) + 2*cos(x)**2 - 1
>>> expr.subs(sin(2*x), 2*sin(x)*cos(x))
2*sin(x)*cos(x) + cos(2*x)
```

There are two important things to note about `subs`. First, it returns a new expression. SymPy objects are immutable. That means that `subs` does not modify it in-place. For example

```
>>> expr = cos(x)
>>> expr.subs(x, 0)
1
>>> expr
cos(x)
>>> x
x
```

Quick Tip

SymPy expressions are immutable. No function will change them in-place.

Here, we see that performing `expr.subs(x, 0)` leaves `expr` unchanged. In fact, since SymPy expressions are immutable, no function will change them in-place. All functions will return new expressions.

To perform multiple substitutions at once, pass a list of (old, new) pairs to `subs`.

```
>>> expr = x**3 + 4*x*y - z
>>> expr.subs([(x, 2), (y, 4), (z, 0)])
40
```

It is often useful to combine this with a list comprehension to do a large set of similar replacements all at once. For example, say we had $x^4 - 4x^3 + 4x^2 - 2x + 3$ and we wanted to replace all instances of x that have an even power with y , to get $y^4 - 4x^3 + 4y^2 - 2x + 3$.

```
>>> expr = x**4 - 4*x**3 + 4*x**2 - 2*x + 3
>>> replacements = [(x**i, y**i) for i in range(5) if i % 2 == 0]
>>> expr.subs(replacements)
-4*x**3 - 2*x + y**4 + 4*y**2 + 3
```


Converting Strings to SymPy Expressions

The `sympify` function (that's `sympify`, not to be confused with `simplify`) can be used to convert strings into SymPy expressions.

For example

```
>>> str_expr = "x**2 + 3*x - 1/2"
>>> expr = sympify(str_expr)
>>> expr
x**2 + 3*x - 1/2
>>> expr.subs(x, 2)
19/2
```

Warning: `sympify` uses `eval`. Don't use it on unsanitized input.

evalf

To evaluate a numerical expression into a floating point number, use `evalf`.

```
>>> expr = sqrt(8)
>>> expr.evalf()
2.82842712474619
```

SymPy can evaluate floating point expressions to arbitrary precision. By default, 15 digits of precision are used, but you can pass any number as the argument to `evalf`. Let's compute the first 100 digits of π .

```
>>> pi.evalf(100)
3.
↪ 1415926535897932384626433832795028841971693993751058209749445923078164062862089986280348
```

To numerically evaluate an expression with a Symbol at a point, we might use `subs` followed by `evalf`, but it is more efficient and numerically stable to pass the substitution to `evalf` using the `subs` flag, which takes a dictionary of Symbol: point pairs.

```
>>> expr = cos(2*x)
>>> expr.evalf(subs={x: 2.4})
0.0874989834394464
```

Sometimes there are roundoff errors smaller than the desired precision that remain after an expression is evaluated. Such numbers can be removed at the user's discretion by setting the `chop` flag to `True`.

```
>>> one = cos(1)**2 + sin(1)**2
>>> (one - 1).evalf()
-0.e-124
>>> (one - 1).evalf(chop=True)
0
```

lambdify

`subs` and `evalf` are good if you want to do simple evaluation, but if you intend to evaluate an expression at many points, there are more efficient ways. For example, if you wanted to evaluate an expression at a thousand points, using SymPy would be far slower than it needs to be, especially if you only care about machine precision. Instead, you should use libraries like [NumPy](#) and [SciPy](#).

The easiest way to convert a SymPy expression to an expression that can be numerically evaluated is to use the `lambdify` function. `lambdify` acts like a `lambda` function, except it converts the SymPy names to the names of the given numerical library, usually NumPy. For example

```
>>> import numpy
>>> a = numpy.arange(10)
>>> expr = sin(x)
>>> f = lambdify(x, expr, "numpy")
>>> f(a)
[ 0.          0.84147098  0.90929743  0.14112001 -0.7568025  -0.95892427
 -0.2794155   0.6569866   0.98935825  0.41211849]
```

Warning: `lambdify` uses `eval`. Don't use it on unsanitized input.

You can use other libraries than NumPy. For example, to use the standard library `math` module, use `"math"`.

```
>>> f = lambdify(x, expr, "math")
>>> f(0.1)
0.0998334166468
```

To use `lambdify` with numerical libraries that it does not know about, pass a dictionary of `sympy_name:numerical_function` pairs. For example

```
>>> def mysin(x):
...     """
...     My sine. Note that this is only accurate for small x.
...     """
...     return x
>>> f = lambdify(x, expr, {"sin":mysin})
>>> f(0.1)
0.1
```

Printing

As we have already seen, SymPy can pretty print its output using Unicode characters. This is a short introduction to the most common printing options available in SymPy.

Printers

There are several printers available in SymPy. The most common ones are

- str
- srepr
- ASCII pretty printer
- Unicode pretty printer
- LaTeX
- MathML
- Dot

In addition to these, there are also “printers” that can output SymPy objects to code, such as C, Fortran, Javascript, Theano, and Python. These are not discussed in this tutorial.

Setting up Pretty Printing

If all you want is the best pretty printing, use the `init_printing()` function. This will automatically enable the best printer available in your environment.

```
>>> from sympy import init_printing
>>> init_printing()
```

If you plan to work in an interactive calculator-type session, the `init_session()` function will automatically import everything in SymPy, create some common Symbols, setup plotting, and run `init_printing()`.

```
>>> from sympy import init_session
>>> init_session()
```

```
Python console for SymPy 0.7.3 (Python 2.7.5-64-bit) (ground types: u
→ gmpy)
```

These commands were executed:

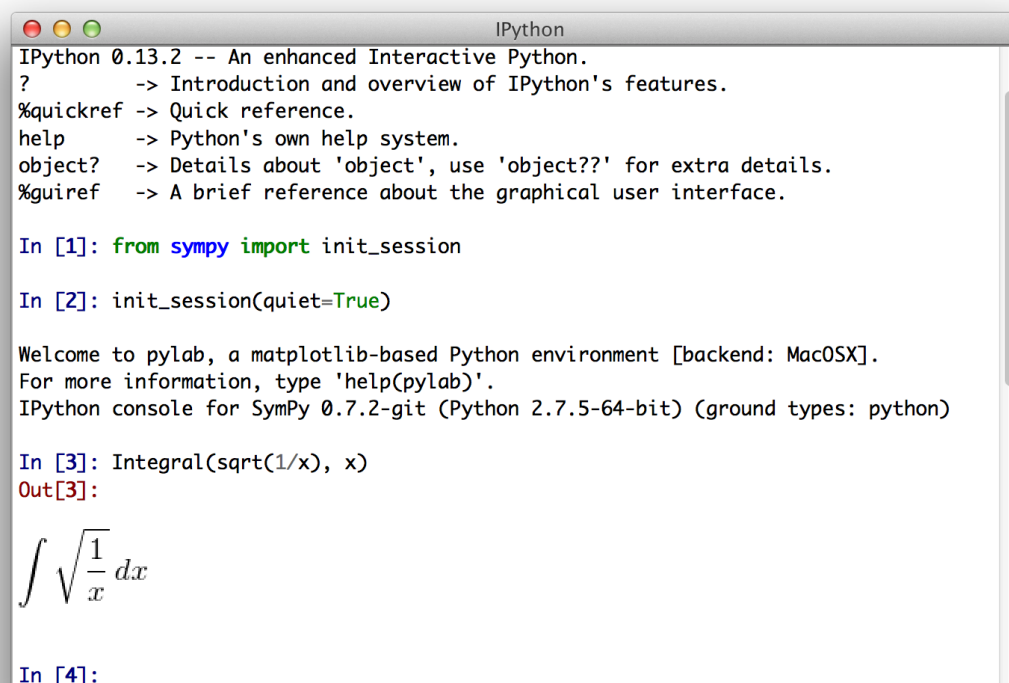
```
>>> from __future__ import division
>>> from sympy import *
>>> x, y, z, t = symbols('x y z t')
>>> k, m, n = symbols('k m n', integer=True)
>>> f, g, h = symbols('f g h', cls=Function)
>>> init_printing() # doctest: +SKIP
```

Documentation can be found at <http://www.sympy.org>

```
>>>
```

In any case, this is what will happen:

- In the IPython QTConsole, if $\text{\LaTeX}$ is installed, it will enable a printer that uses $\text{\LaTeX}$.



```
IPython 0.13.2 -- An enhanced Interactive Python.
?      -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help    -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.
%guieref -> A brief reference about the graphical user interface.

In [1]: from sympy import init_session

In [2]: init_session(quiet=True)

Welcome to pylab, a matplotlib-based Python environment [backend: MacOSX].
For more information, type 'help(pylab)'.
IPython console for SymPy 0.7.2-git (Python 2.7.5-64-bit) (ground types: python)

In [3]: Integral(sqrt(1/x), x)
Out[3]:
```

$$\int \sqrt{\frac{1}{x}} dx$$

```
In [4]:
```

If $\text{\LaTeX}$ is not installed, but Matplotlib is installed, it will use the Matplotlib rendering engine. If Matplotlib is not installed, it uses the Unicode pretty printer.

- In the IPython notebook, it will use MathJax to render $\text{\LaTeX}$.

```
In [1]: from sympy import *
x, y, z = symbols('x y z')
init_printing()
```

```
In [2]: Integral(sqrt(1/x), x)
```

```
Out[2]:
```

$$\int \sqrt{\frac{1}{x}} dx$$

- In an IPython console session, or a regular Python session, it will use the Unicode pretty printer if the terminal supports Unicode.

```
Python 2.7.5 (default, May 16 2013, 18:48:51)
[GCC 4.2.1 Compatible Apple LLVM 4.2 (clang-425.0.28)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from sympy import init_session
>>> init_session()
Python console for SymPy 0.7.2-git (Python 2.7.5-64-bit) (ground types: gmpy)

These commands were executed:
>>> from __future__ import division
>>> from sympy import *
>>> x, y, z, t = symbols('x y z t')
>>> k, m, n = symbols('k m n', integer=True)
>>> f, g, h = symbols('f g h', cls=Function)

Documentation can be found at http://www.sympy.org

>>> Integral(sqrt(1/x), x)

$$\int \sqrt{\frac{1}{x}} dx$$

>>>
```

- In a terminal that does not support Unicode, the ASCII pretty printer is used.

```
Python 2.7.5 (default, May 16 2013, 18:48:51)
[GCC 4.2.1 Compatible Apple LLVM 4.2 (clang-425.0.28)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from sympy import init_session
>>> init_session()
Python console for SymPy 0.7.2-git (Python 2.7.5-64-bit) (ground types: gmpy)

These commands were executed:
>>> from __future__ import division
>>> from sympy import *
>>> x, y, z, t = symbols('x y z t')
>>> k, m, n = symbols('k m n', integer=True)
>>> f, g, h = symbols('f g h', cls=Function)

Documentation can be found at http://www.sympy.org

>>> Integral(sqrt(1/x), x)

$$\int \sqrt{\frac{1}{x}} dx$$

>>> █
```

To explicitly not use $\LaTeX$, pass `use_latex=False` to `init_printing()` or `init_session()`.

To explicitly not use Unicode, pass `use_unicode=False`.

Printing Functions

In addition to automatic printing, you can explicitly use any one of the printers by calling the appropriate function.

str

To get a string form of an expression, use `str(expr)`. This is also the form that is produced by `print(expr)`. String forms are designed to be easy to read, but in a form that is correct Python syntax so that it can be copied and pasted. The `str()` form of an expression will usually look exactly the same as the expression as you would enter it.

```
>>> from sympy import *
>>> x, y, z = symbols('x y z')
>>> str(Integral(sqrt(1/x), x))
'Integral(sqrt(1/x), x)'
>>> print(Integral(sqrt(1/x), x))
Integral(sqrt(1/x), x)
```

srepr

The `srepr` form of an expression is designed to show the exact form of an expression. It will be discussed more in the *Advanced Expression Manipulation* (page 61) section. To get it, use `srepr()`¹.

```
>>> srepr(Integral(sqrt(1/x), x))
"Integral(Pow(Pow(Symbol('x'), Integer(-1)), Rational(1, 2)), Tuple(Symbol('x'
↪ ')))"
```

The srepr form is mostly useful for understanding how an expression is built internally.

ASCII Pretty Printer

The ASCII pretty printer is accessed from `pprint()`. If the terminal does not support Unicode, the ASCII printer is used by default. Otherwise, you must pass `use_unicode=False`.

```
>>> pprint(Integral(sqrt(1/x), x), use_unicode=False)
```

$$\int \frac{1}{x} dx$$

(continues on next page)

¹ SymPy does not use the Python builtin `repr()` function for repr printing, because in Python `str(list)` calls `repr()` on the elements of the list, and some SymPy functions return lists (such as `solve()`). Since `srepr()` is so verbose, it is unlikely that anyone would want it called by default on the output of `solve()`.

$$\int \sqrt{\frac{1}{x}} \, dx$$

MathML

There is also a printer to MathML, called `print_mathml()`. It must be imported from `sympy.printing.mathml`.

```
>>> from sympy.printing.mathml import print_mathml
>>> print_mathml(Integral(sqrt(1/x), x))
<apply>
  <int/>
    <bvar>
      <ci>x</ci>
    </bvar>
    <apply>
      <root/>
        <apply>
          <power/>
            <ci>x</ci>
            <cn>-1</cn>
          </apply>
        </apply>
      </apply>
    </int/>
  </apply>
```

`print_mathml()` prints the output. If you want the string, use the function `mathml()`.

Dot

The `dotprint()` function in `sympy.printing.dot` prints output to dot format, which can be rendered with Graphviz. See the [Advanced Expression Manipulation](#) (page 61) section for some examples of the output of this printer.

Here is an example of the raw output of the `dotprint()` function

```
>>> from sympy.printing.dot import dotprint
>>> from sympy.abc import x
>>> print(dotprint(x+2))
digraph{

# Graph style
"ordering"="out"
"rankdir"="TD"

#####
# Nodes #
#####

"Add(Integer(2), Symbol('x'))_()" ["color"="black", "label"="Add", "shape"=
->"ellipse"];
"Integer(2)_()" ["color"="black", "label"="2", "shape"="ellipse"];
"Symbol('x')_()" ["color"="black", "label"="x", "shape"="ellipse"];

#####
# Edges #
```

(continues on next page)

(continued from previous page)

```
#####
"Add(Integer(2), Symbol('x'))_()" -> "Integer(2)_(0,)"
"Add(Integer(2), Symbol('x'))_()" -> "Symbol('x')_(1,)"
}
```

Simplification

To make this document easier to read, we are going to enable pretty printing.

```
>>> from sympy import *
>>> x, y, z = symbols('x y z')
>>> init_printing(use_unicode=True)
```

simplify

Now let's jump in and do some interesting mathematics. One of the most useful features of a symbolic manipulation system is the ability to simplify mathematical expressions. SymPy has dozens of functions to perform various kinds of simplification. There is also one general function called `simplify()` that attempts to apply all of these functions in an intelligent way to arrive at the simplest form of an expression. Here are some examples

```
>>> simplify(sin(x)**2 + cos(x)**2)
1
>>> simplify((x**3 + x**2 - x - 1)/(x**2 + 2*x + 1))
x - 1
>>> simplify(gamma(x)/gamma(x - 2))
(x - 2)*(x - 1)
```

Here, `gamma(x)` is $\Gamma(x)$, the [gamma function](#). We see that `simplify()` is capable of handling a large class of expressions.

But `simplify()` has a pitfall. It just applies all the major simplification operations in SymPy, and uses heuristics to determine the simplest result. But “simplest” is not a well-defined term. For example, say we wanted to “simplify” $x^2 + 2x + 1$ into $(x + 1)^2$:

```
>>> simplify(x**2 + 2*x + 1)
2
x + 2*x + 1
```

We did not get what we want. There is a function to perform this simplification, called `factor()`, which will be discussed below.

Another pitfall to `simplify()` is that it can be unnecessarily slow, since it tries many kinds of simplifications before picking the best one. If you already know exactly what kind of simplification you are after, it is better to apply the specific simplification function(s) that apply those simplifications.

Applying specific simplification functions instead of `simplify()` also has the advantage that specific functions have certain guarantees about the form of their output. These will be discussed with each function below. For example, `factor()`, when called on a polynomial

with rational coefficients, is guaranteed to factor the polynomial into irreducible factors. `simplify()` has no guarantees. It is entirely heuristical, and, as we saw above, it may even miss a possible type of simplification that SymPy is capable of doing.

`simplify()` is best when used interactively, when you just want to whittle down an expression to a simpler form. You may then choose to apply specific functions once you see what `simplify()` returns, to get a more precise result. It is also useful when you have no idea what form an expression will take, and you need a catchall function to simplify it.

Polynomial/Rational Function Simplification

expand

`expand()` is one of the most common simplification functions in SymPy. Although it has a lot of scopes, for now, we will consider its function in expanding polynomial expressions. For example:

```
>>> expand((x + 1)**2)
      2
x  + 2·x + 1
>>> expand((x + 2)*(x - 3))
      2
x  - x - 6
```

Given a polynomial, `expand()` will put it into a canonical form of a sum of monomials.

`expand()` may not sound like a simplification function. After all, by its very name, it makes expressions bigger, not smaller. Usually this is the case, but often an expression will become smaller upon calling `expand()` on it due to cancellation.

```
>>> expand((x + 1)*(x - 2) - (x - 1)*x)
-2
```

factor

`factor()` takes a polynomial and factors it into irreducible factors over the rational numbers. For example:

```
>>> factor(x**3 - x**2 + x - 1)
      ( 2
(x - 1)·(x  + 1)
>>> factor(x**2*z + 4*x*y*z + 4*y**2*z)
      2
z·(x + 2·y)
```

For polynomials, `factor()` is the opposite of `expand()`. `factor()` uses a complete multivariate factorization algorithm over the rational numbers, which means that each of the factors returned by `factor()` is guaranteed to be irreducible.

If you are interested in the factors themselves, `factor_list` returns a more structured output.

```
>>> factor_list(x**2*z + 4*x*y*z + 4*y**2*z)
(1, [(z, 1), (x + 2·y, 2)])
```

Note that the input to `factor` and `expand` need not be polynomials in the strict sense. They will intelligently factor or expand any kind of expression (though note that the factors may not be irreducible if the input is no longer a polynomial over the rationals).

```
>>> expand((cos(x) + sin(x))**2)
      2      2
sin (x) + 2·sin(x)·cos(x) + cos (x)
>>> factor(cos(x)**2 + 2*cos(x)*sin(x) + sin(x)**2)
      2
(sin(x) + cos(x))
```

collect

`collect()` collects common powers of a term in an expression. For example

```
>>> expr = x*y + x - 3 + 2*x**2 - z*x**2 + x**3
>>> expr
      3      2      2
x  - x  ·z + 2·x  + x·y + x - 3
>>> collected_expr = collect(expr, x)
>>> collected_expr
      3      2
x  + x  ·(2 - z) + x·(y + 1) - 3
```

`collect()` is particularly useful in conjunction with the `.coeff()` method. `expr.coeff(x, n)` gives the coefficient of x^n in `expr`:

```
>>> collected_expr.coeff(x, 2)
2 - z
```

cancel

`cancel()` will take any rational function and put it into the standard canonical form, $\frac{p}{q}$, where p and q are expanded polynomials with no common factors, and the leading coefficients of p and q do not have denominators (i.e., are integers).

```
>>> cancel((x**2 + 2*x + 1)/(x**2 + x))
x + 1
-----
x
```

```
>>> expr = 1/x + (3*x/2 - 2)/(x - 4)
>>> expr
      3·x
----- - 2
      2      1
----- + -
      x - 4  x
>>> cancel(expr)
      2
```

(continues on next page)

(continued from previous page)

$$\frac{3 \cdot x^2 - 2 \cdot x - 8}{2 \cdot x^2 - 8 \cdot x}$$

```
>>> expr = (x*y**2 - 2*x*y*z + x*z**2 + y**2 - 2*y*z + z**2)/(x**2 - 1)
>>> expr
```

$$\frac{x^2 \cdot y^2 - 2 \cdot x \cdot y \cdot z + x \cdot z^2 + y^2 - 2 \cdot y \cdot z + z^2}{x^2 - 1}$$

```
>>> cancel(expr)
```

$$\frac{y^2 - 2 \cdot y \cdot z + z^2}{x - 1}$$

Note that since `factor()` will completely factorize both the numerator and the denominator of an expression, it can also be used to do the same thing:

```
>>> factor(expr)
```

$$\frac{(y - z)^2}{x - 1}$$

However, if you are only interested in making sure that the expression is in canceled form, `cancel()` is more efficient than `factor()`.

apart

`apart()` performs a [partial fraction decomposition](#) on a rational function.

```
>>> expr = (4*x**3 + 21*x**2 + 10*x + 12)/(x**4 + 5*x**3 + 5*x**2 + 4*x)
>>> expr
```

$$\frac{4 \cdot x^3 + 21 \cdot x^2 + 10 \cdot x + 12}{x^4 + 5 \cdot x^3 + 5 \cdot x^2 + 4 \cdot x}$$

```
>>> apart(expr)
```

$$\frac{2 \cdot x - 1}{x^2 + x + 1} - \frac{1}{x + 4} + \frac{3}{x}$$

Trigonometric Simplification

Note: SymPy follows Python’s naming conventions for inverse trigonometric functions, which is to append an `a` to the front of the function’s name. For example, the inverse cosine, or arc cosine, is called `acos()`.

```
>>> acos(x)
acos(x)
>>> cos(acos(x))
x
>>> asin(1)
π
—
2
```

trigsimp

To simplify expressions using trigonometric identities, use `trigsimp()`.

```
>>> trigsimp(sin(x)**2 + cos(x)**2)
1
>>> trigsimp(sin(x)**4 - 2*cos(x)**2*sin(x)**2 + cos(x)**4)
cos(4·x)  1
——— + —
  2      2
>>> trigsimp(sin(x)*tan(x)/sec(x))
  2
sin (x)
```

`trigsimp()` also works with hyperbolic trig functions.

```
>>> trigsimp(cosh(x)**2 + sinh(x)**2)
cosh(2·x)
>>> trigsimp(sinh(x)/tanh(x))
cosh(x)
```

Much like `simplify()`, `trigsimp()` applies various trigonometric identities to the input expression, and then uses a heuristic to return the “best” one.

expand_trig

To expand trigonometric functions, that is, apply the sum or double angle identities, use `expand_trig()`.

```
>>> expand_trig(sin(x + y))
sin(x)·cos(y) + sin(y)·cos(x)
>>> expand_trig(tan(2·x))
  2·tan(x)
———
```

(continues on next page)

(continued from previous page)

$$\frac{2}{1 - \tan^2(x)}$$

Because `expand_trig()` tends to make trigonometric expressions larger, and `trigsimp()` tends to make them smaller, these identities can be applied in reverse using `trigsimp()`

```
>>> trigsimp(sin(x)*cos(y) + sin(y)*cos(x))
sin(x + y)
```

Powers

Before we introduce the power simplification functions, a mathematical discussion on the identities held by powers is in order. There are three kinds of identities satisfied by exponents

1. $x^a x^b = x^{a+b}$
2. $x^a y^a = (xy)^a$
3. $(x^a)^b = x^{ab}$

Identity 1 is always true.

Identity 2 is not always true. For example, if $x = y = -1$ and $a = \frac{1}{2}$, then $x^a y^a = \sqrt{-1}\sqrt{-1} = i \cdot i = -1$, whereas $(xy)^a = \sqrt{-1 \cdot -1} = \sqrt{1} = 1$. However, identity 2 is true at least if x and y are nonnegative and a is real (it may also be true under other conditions as well). A common consequence of the failure of identity 2 is that $\sqrt{x}\sqrt{y} \neq \sqrt{xy}$.

Identity 3 is not always true. For example, if $x = -1$, $a = 2$, and $b = \frac{1}{2}$, then $(x^a)^b = ((-1)^2)^{1/2} = \sqrt{1} = 1$ and $x^{ab} = (-1)^{2 \cdot 1/2} = (-1)^1 = -1$. However, identity 3 is true when b is an integer (again, it may also hold in other cases as well). Two common consequences of the failure of identity 3 are that $\sqrt{x^2} \neq x$ and that $\sqrt{\frac{1}{x}} \neq \frac{1}{\sqrt{x}}$.

To summarize

Identity	Sufficient conditions to hold	Counterexample when conditions are not met	Important consequences
1. $x^a x^b = x^{a+b}$	Always true	None	None
2. $x^a y^a = (xy)^a$	$x, y \geq 0$ and $a \in \mathbb{R}$	$(-1)^{1/2}(-1)^{1/2} \neq (-1 \cdot -1)^{1/2}$	$\sqrt{x}\sqrt{y} \neq \sqrt{xy}$ in general
3. $(x^a)^b = x^{ab}$	$b \in \mathbb{Z}$	$((-1)^2)^{1/2} \neq (-1)^{2 \cdot 1/2}$	$\sqrt{x^2} \neq x$ and $\sqrt{\frac{1}{x}} \neq \frac{1}{\sqrt{x}}$ in general

This is important to remember, because by default, SymPy will not perform simplifications if they are not true in general.

In order to make SymPy perform simplifications involving identities that are only true under certain assumptions, we need to put assumptions on our Symbols. We will undertake a full discussion of the assumptions system later, but for now, all we need to know are the following.

- By default, SymPy Symbols are assumed to be complex (elements of $\mathbb{C}$). That is, a simplification will not be applied to an expression with a given Symbol unless it holds for all complex numbers.
- Symbols can be given different assumptions by passing the assumption to `symbols()`. For the rest of this section, we will be assuming that x and y are positive, and that a and b are real. We will leave z , t , and c as arbitrary complex Symbols to demonstrate what happens in that case.

```
>>> x, y = symbols('x y', positive=True)
>>> a, b = symbols('a b', real=True)
>>> z, t, c = symbols('z t c')
```

Note: In SymPy, `sqrt(x)` is just a shortcut to `x**Rational(1, 2)`. They are exactly the same object.

```
>>> sqrt(x) == x**Rational(1, 2)
True
```

powsimp

`powsimp()` applies identities 1 and 2 from above, from left to right.

```
>>> powsimp(x**a*x**b)
a + b
x
>>> powsimp(x**a*y**a)
a
(x·y)
```

Notice that `powsimp()` refuses to do the simplification if it is not valid.

```
>>> powsimp(t**c*z**c)
c c
t · z
```

If you know that you want to apply this simplification, but you don't want to mess with assumptions, you can pass the `force=True` flag. This will force the simplification to take place, regardless of assumptions.

```
>>> powsimp(t**c*z**c, force=True)
c
(t·z)
```

Note that in some instances, in particular, when the exponents are integers or rational numbers, and identity 2 holds, it will be applied automatically.

```
>>> (z*t)**2
2 2
t · z
>>> sqrt(x*y)
√x·√y
```

This means that it will be impossible to undo this identity with `powsimp()`, because even if `powsimp()` were to put the bases together, they would be automatically split apart again.

```
>>> powsimp(z**2*t**2)
  2 2
  t · z
>>> powsimp(sqrt(x)*sqrt(y))
√x·√y
```

expand_power_exp / expand_power_base

`expand_power_exp()` and `expand_power_base()` apply identities 1 and 2 from right to left, respectively.

```
>>> expand_power_exp(x**(a + b))
  a  b
  x · x
```

```
>>> expand_power_base((x*y)**a)
  a  a
  x · y
```

As with `powsimp()`, identity 2 is not applied if it is not valid.

```
>>> expand_power_base((z*t)**c)
      c
  (t·z)
```

And as with `powsimp()`, you can force the expansion to happen without fiddling with assumptions by using `force=True`.

```
>>> expand_power_base((z*t)**c, force=True)
  c  c
  t · z
```

As with identity 2, identity 1 is applied automatically if the power is a number, and hence cannot be undone with `expand_power_exp()`.

```
>>> x**2*x**3
  5
  x
>>> expand_power_exp(x**5)
  5
  x
```


powdenest

`powdenest()` applies identity 3, from left to right.

```
>>> powdenest((x**a)**b)
a·b
x
```

As before, the identity is not applied if it is not true under the given assumptions.

```
>>> powdenest((z**a)**b)
b
( a )
( z )
```

And as before, this can be manually overridden with `force=True`.

```
>>> powdenest((z**a)**b, force=True)
a·b
z
```

Exponentials and logarithms

Note: In SymPy, as in Python and most programming languages, `log` is the natural logarithm, also known as `ln`. SymPy automatically provides an alias `ln = log` in case you forget this.

```
>>> ln(x)
log(x)
```

Logarithms have similar issues as powers. There are two main identities

1. $\log(xy) = \log(x) + \log(y)$
2. $\log(x^n) = n \log(x)$

Neither identity is true for arbitrary complex x and y , due to the branch cut in the complex plane for the complex logarithm. However, sufficient conditions for the identities to hold are if x and y are positive and n is real.

```
>>> x, y = symbols('x y', positive=True)
>>> n = symbols('n', real=True)
```

As before, z and t will be Symbols with no additional assumptions.

Note that the identity $\log\left(\frac{x}{y}\right) = \log(x) - \log(y)$ is a special case of identities 1 and 2 by $\log\left(\frac{x}{y}\right) = \log\left(x \cdot \frac{1}{y}\right) = \log(x) + \log(y^{-1}) = \log(x) - \log(y)$, and thus it also holds if x and y are positive, but may not hold in general.

We also see that $\log(e^x) = x$ comes from $\log(e^x) = x \log(e) = x$, and thus holds when x is real (and it can be verified that it does not hold in general for arbitrary complex x , for example, $\log(e^{x+2\pi i}) = \log(e^x) = x \neq x + 2\pi i$).

expand_log

To apply identities 1 and 2 from left to right, use `expand_log()`. As always, the identities will not be applied unless they are valid.

```
>>> expand_log(log(x*y))
log(x) + log(y)
>>> expand_log(log(x/y))
log(x) - log(y)
>>> expand_log(log(x**2))
2*log(x)
>>> expand_log(log(x**n))
n*log(x)
>>> expand_log(log(z*t))
log(t*z)
```

As with `powsimp()` and `powdenest()`, `expand_log()` has a `force` option that can be used to ignore assumptions.

```
>>> expand_log(log(z**2))
log(z2)
>>> expand_log(log(z**2), force=True)
2*log(z)
```

logcombine

To apply identities 1 and 2 from right to left, use `logcombine()`.

```
>>> logcombine(log(x) + log(y))
log(x*y)
>>> logcombine(n*log(x))
log(xn)
>>> logcombine(n*log(z))
n*log(z)
```

`logcombine()` also has a `force` option that can be used to ignore assumptions.

```
>>> logcombine(n*log(z), force=True)
log(zn)
```

Special Functions

SymPy implements dozens of special functions, ranging from functions in combinatorics to mathematical physics.

An extensive list of the special functions included with SymPy and their documentation is at the [Functions Module](#) (page 382) page.

For the purposes of this tutorial, let's introduce a few special functions in SymPy.

Let's define x , y , and z as regular, complex Symbols, removing any assumptions we put on them in the previous section. We will also define k , m , and n .

```
>>> x, y, z = symbols('x y z')
>>> k, m, n = symbols('k m n')
```

The [factorial](#) function is `factorial`. `factorial(n)` represents $n! = 1 \cdot 2 \cdots (n-1) \cdot n$. $n!$ represents the number of permutations of n distinct items.

```
>>> factorial(n)
n!
```

The [binomial coefficient](#) function is `binomial`. `binomial(n, k)` represents $\binom{n}{k}$, the number of ways to choose k items from a set of n distinct items. It is also often written as nCk , and is pronounced “ n choose k ”.

```
>>> binomial(n, k)

$$\binom{n}{k}$$

```

The factorial function is closely related to the [gamma function](#), `gamma`. `gamma(z)` represents $\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt$, which for positive integer z is the same as $(z-1)!$.

```
>>> gamma(z)
 $\Gamma(z)$ 
```

The [generalized hypergeometric function](#) is `hyper`. `hyper([a_1, ..., a_p], [b_1, ..., b_q], z)` represents ${}_pF_q \left(\begin{matrix} a_1, \dots, a_p \\ b_1, \dots, b_q \end{matrix} \middle| z \right)$. The most common case is ${}_2F_1$, which is often referred to as the [ordinary hypergeometric function](#).

```
>>> hyper([1, 2], [3], z)

$${}_2F_1 \left( \begin{matrix} 1, 2 \\ 3 \end{matrix} \middle| z \right)$$

```

rewrite

A common way to deal with special functions is to rewrite them in terms of one another. This works for any function in SymPy, not just special functions. To rewrite an expression in terms of a function, use `expr.rewrite(function)`. For example,

```
>>> tan(x).rewrite(cos)
cos( $x - \frac{\pi}{2}$ )
cos(x)
>>> factorial(x).rewrite(gamma)
 $\Gamma(x + 1)$ 
```

For some tips on applying more targeted rewriting, see the [Advanced Expression Manipulation](#) (page 61) section.

expand_func

To expand special functions in terms of some identities, use `expand_func()`. For example

```
>>> expand_func(gamma(x + 3))
x*(x + 1)*(x + 2)* $\Gamma(x)$ 
```

hyperexpand

To rewrite hyper in terms of more standard functions, use `hyperexpand()`.

```
>>> hyperexpand(hyper([1, 1], [2], z))
 $-\frac{\log(1 - z)}{z}$ 
```

`hyperexpand()` also works on the more general Meijer G-function (see [its documentation](#) (page 522) for more information).

```
>>> expr = meijerg([[1],[1]], [[1],[ ]], -z)
>>> expr
 $G_{2,1}^{1,1}\left(\begin{matrix} 1 & 1 \\ 1 \end{matrix} \middle| -z\right)$ 
>>> hyperexpand(expr)
 $-\frac{1}{z}e^z$ 
```

combsimp

To simplify combinatorial expressions, use `combsimp()`.

```
>>> n, k = symbols('n k', integer = True)
>>> combsimp(factorial(n)/factorial(n - 3))
n*(n - 2)*(n - 1)
>>> combsimp(binomial(n+1, k+1)/binomial(n, k))
n + 1
-----
k + 1
```

gammasimp

To simplify expressions with gamma functions or combinatorial functions with non-integer argument, use `gammasimp()`.

```
>>> gammasimp(gamma(x)*gamma(1 - x))
π
-----
sin(π*x)
```

Example: Continued Fractions

Let's use SymPy to explore continued fractions. A [continued fraction](#) is an expression of the form

$$a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{\ddots + \frac{1}{a_n}}}}$$

where $a_0, \dots, a_n$ are integers, and $a_1, \dots, a_n$ are positive. A continued fraction can also be infinite, but infinite objects are more difficult to represent in computers, so we will only examine the finite case here.

A continued fraction of the above form is often represented as a list $[a_0; a_1, \dots, a_n]$. Let's write a simple function that converts such a list to its continued fraction form. The easiest way to construct a continued fraction from a list is to work backwards. Note that despite the apparent symmetry of the definition, the first element, a_0 , must usually be handled differently from the rest.

```
>>> def list_to_frac(l):
...     expr = Integer(0)
...     for i in reversed(l[1:]):
...         expr += i
...         expr = 1/expr
...     return l[0] + expr
>>> list_to_frac([x, y, z])
```

(continues on next page)

(continued from previous page)

$$x + \frac{1}{y + \frac{1}{z}}$$

We use `Integer(0)` in `list_to_frac` so that the result will always be a SymPy object, even if we only pass in Python ints.

```
>>> list_to_frac([1, 2, 3, 4])
43
—
30
```

Every finite continued fraction is a rational number, but we are interested in symbolics here, so let's create a symbolic continued fraction. The `symbols()` function that we have been using has a shortcut to create numbered symbols. `symbols('a0:5')` will create the symbols `a0`, `a1`, ..., `a4`.

```
>>> syms = symbols('a0:5')
>>> syms
(a0, a1, a2, a3, a4)
>>> a0, a1, a2, a3, a4 = syms
>>> frac = list_to_frac(syms)
>>> frac
```

$$a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{a_3 + \frac{1}{a_4}}}}$$

This form is useful for understanding continued fractions, but let's put it into standard rational function form using `cancel()`.

```
>>> frac = cancel(frac)
>>> frac
a0*a1*a2*a3*a4 + a0*a1*a2 + a0*a1*a4 + a0*a3*a4 + a0 + a2*a3*a4 + a2 + a4
—
a1*a2*a3*a4 + a1*a2 + a1*a4 + a3*a4 + 1
```

Now suppose we were given `frac` in the above canceled form. In fact, we might be given the fraction in any form, but we can always put it into the above canonical form with `cancel()`. Suppose that we knew that it could be rewritten as a continued fraction. How could we do this with SymPy? A continued fraction is recursively $c + \frac{1}{f}$, where c is an integer and f is a (smaller) continued fraction. If we could write the expression in this form, we could pull out each c recursively and add it to a list. We could then get a continued fraction with our `list_to_frac()` function.

The key observation here is that we can convert an expression to the form $c + \frac{1}{f}$ by doing a partial fraction decomposition with respect to c . This is because f does not contain c . This

means we need to use the `apart()` function. We use `apart()` to pull the term out, then subtract it from the expression, and take the reciprocal to get the f part.

```
>>> l = []
>>> frac = apart(frac, a0)
>>> frac
```

$$a_0 + \frac{a_2 \cdot a_3 \cdot a_4 + a_2 + a_4}{a_1 \cdot a_2 \cdot a_3 \cdot a_4 + a_1 \cdot a_2 + a_1 \cdot a_4 + a_3 \cdot a_4 + 1}$$

```
>>> l.append(a0)
>>> frac = 1/(frac - a0)
>>> frac
```

$$\frac{a_1 \cdot a_2 \cdot a_3 \cdot a_4 + a_1 \cdot a_2 + a_1 \cdot a_4 + a_3 \cdot a_4 + 1}{a_2 \cdot a_3 \cdot a_4 + a_2 + a_4}$$

Now we repeat this process

```
>>> frac = apart(frac, a1)
>>> frac
```

$$a_1 + \frac{a_3 \cdot a_4 + 1}{a_2 \cdot a_3 \cdot a_4 + a_2 + a_4}$$

```
>>> l.append(a1)
>>> frac = 1/(frac - a1)
>>> frac = apart(frac, a2)
>>> frac
```

$$a_2 + \frac{a_4}{a_3 \cdot a_4 + 1}$$

```
>>> l.append(a2)
>>> frac = 1/(frac - a2)
>>> frac = apart(frac, a3)
>>> frac
```

$$a_3 + \frac{1}{a_4}$$

```
>>> l.append(a3)
>>> frac = 1/(frac - a3)
>>> frac = apart(frac, a4)
>>> frac
```

$$a_4$$

```
>>> l.append(a4)
>>> list_to_frac(l)
```

$$a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{a_3 + \frac{1}{a_4}}}}$$

Of course, this exercise seems pointless, because we already know that our `frac` is `list_to_frac([a0, a1, a2, a3, a4])`. So try the following exercise. Take a list of symbols and randomize them, and create the canceled continued fraction, and see if you can reproduce the original list. For example

```
>>> import random
>>> l = list(symbols('a0:5'))
>>> random.shuffle(l)
>>> orig_frac = frac = cancel(list_to_frac(l))
>>> del l
```

In SymPy, on the above example, try to reproduce `l` from `frac`. I have deleted `l` at the end to remove the temptation for peeking (you can check your answer at the end by calling `cancel(list_to_frac(l))` on the list that you generate at the end, and comparing it to `orig_frac`).

See if you can think of a way to figure out what symbol to pass to `apart()` at each stage (hint: think of what happens to a_0 in the formula $a_0 + \frac{1}{a_1 + \dots}$ when it is canceled).

Calculus

This section covers how to do basic calculus tasks such as derivatives, integrals, limits, and series expansions in SymPy. If you are not familiar with the math of any part of this section, you may safely skip it.

```
>>> from sympy import *
>>> x, y, z = symbols('x y z')
>>> init_printing(use_unicode=True)
```

Derivatives

To take derivatives, use the `diff` function.

```
>>> diff(cos(x), x)
-sin(x)
>>> diff(exp(x**2), x)
      ( 2 )
      (x )
2·x·e
```

`diff` can take multiple derivatives at once. To take multiple derivatives, pass the variable as many times as you wish to differentiate, or pass a number after the variable. For example, both of the following find the third derivative of x^4 .

```
>>> diff(x**4, x, x, x)
24·x
>>> diff(x**4, x, 3)
24·x
```

You can also take derivatives with respect to many variables at once. Just pass each derivative in order, using the same syntax as for single variable derivatives. For example, each of the following will compute $\frac{\partial^7}{\partial x \partial y^2 \partial z^4} e^{xyz}$.


```

>>> expr = exp(x*y*z)
>>> diff(expr, x, y, y, z, z, z, z)
      3 2 ( 3 3 3      2 2 2 ) x·y·z
x·y·(x·y·z + 14·x·y·z + 52·x·y·z + 48)·e
>>> diff(expr, x, y, 2, z, 4)
      3 2 ( 3 3 3      2 2 2 ) x·y·z
x·y·(x·y·z + 14·x·y·z + 52·x·y·z + 48)·e
>>> diff(expr, x, y, y, z, 4)
      3 2 ( 3 3 3      2 2 2 ) x·y·z
x·y·(x·y·z + 14·x·y·z + 52·x·y·z + 48)·e

```

`diff` can also be called as a method. The two ways of calling `diff` are exactly the same, and are provided only for convenience.

```

>>> expr.diff(x, y, y, z, 4)
      3 2 ( 3 3 3      2 2 2 ) x·y·z
x·y·(x·y·z + 14·x·y·z + 52·x·y·z + 48)·e

```

To create an unevaluated derivative, use the `Derivative` class. It has the same syntax as `diff`.

```

>>> deriv = Derivative(expr, x, y, y, z, 4)
>>> deriv
      7
      ∂
      ( x·y·z )
----- (e )
      4 2
      ∂z ∂y ∂x

```

To evaluate an unevaluated derivative, use the `doit` method.

```

>>> deriv.doit()
      3 2 ( 3 3 3      2 2 2 ) x·y·z
x·y·(x·y·z + 14·x·y·z + 52·x·y·z + 48)·e

```

These unevaluated objects are useful for delaying the evaluation of the derivative, or for printing purposes. They are also used when SymPy does not know how to compute the derivative of an expression (for example, if it contains an undefined function, which are described in the [Solving Differential Equations](#) (page 50) section).

Derivatives of unspecified order can be created using tuple (x, n) where n is the order of the derivative with respect to x .

```

>>> m, n, a, b = symbols('m n a b')
>>> expr = (a*x + b)**m
>>> expr.diff((x, n))
      n
      ∂
      (      m )
----- (a·x + b)
      n
      ∂x

```

Integrals

To compute an integral, use the `integrate` function. There are two kinds of integrals, definite and indefinite. To compute an indefinite integral, that is, an antiderivative, or primitive, just pass the variable after the expression.

```
>>> integrate(cos(x), x)
sin(x)
```

Note that SymPy does not include the constant of integration. If you want it, you can add one yourself, or rephrase your problem as a differential equation and use `dsolve` to solve it, which does add the constant (see [Solving Differential Equations](#) (page 50)).

Quick Tip

∞ in SymPy is `oo` (that's the lowercase letter "oh" twice). This is because `oo` looks like ∞ , and is easy to type.

To compute a definite integral, pass the argument (`integration_variable`, `lower_limit`, `upper_limit`). For example, to compute

$$\int_0^{\infty} e^{-x} dx,$$

we would do

```
>>> integrate(exp(-x), (x, 0, oo))
1
```

As with indefinite integrals, you can pass multiple limit tuples to perform a multiple integral. For example, to compute

$$\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} e^{-x^2-y^2} dx dy,$$

do

```
>>> integrate(exp(-x**2 - y**2), (x, -oo, oo), (y, -oo, oo))
pi
```

If `integrate` is unable to compute an integral, it returns an unevaluated `Integral` object.

```
>>> expr = integrate(x**x, x)
>>> print(expr)
Integral(x**x, x)
>>> expr

$$\int x^x dx$$

```

As with `Derivative`, you can create an unevaluated integral using `Integral`. To later evaluate this integral, call `doit`.

```
>>> expr = Integral(log(x)**2, x)
>>> expr

$$\int \log^2(x) dx$$

>>> expr.doit()

$$x \cdot \log^2(x) - 2 \cdot x \cdot \log(x) + 2 \cdot x$$

```

`integrate` uses powerful algorithms that are always improving to compute both definite and indefinite integrals, including heuristic pattern matching type algorithms, a partial implementation of the [Risch algorithm](#), and an algorithm using [Meijer G-functions](#) that is useful for computing integrals in terms of special functions, especially definite integrals. Here is a sampling of some of the power of `integrate`.

```
>>> integ = Integral((x**4 + x**2*exp(x) - x**2 - 2*x*exp(x) - 2*x -
... exp(x))*exp(x)/((x - 1)**2*(x + 1)**2*(exp(x) + 1)), x)
>>> integ

$$\int \frac{\left( x^4 + x^2 \cdot e^x - x^2 - 2 \cdot x \cdot e^x - 2 \cdot x - e^x \right) \cdot e^x}{(x - 1)^2 \cdot (x + 1)^2 \cdot (e^x + 1)} dx$$

>>> integ.doit()

$$\log(e^x + 1) + \frac{e^x}{x^2 - 1}$$

```

```
>>> integ = Integral(sin(x**2), x)
>>> integ

$$\int \sin(x^2) dx$$

>>> integ.doit()

$$\frac{3 \cdot \sqrt{2} \cdot \sqrt{\pi} \cdot S\left(\frac{\sqrt{2} \cdot x}{\sqrt{\pi}}\right) \cdot \Gamma(3/4)}{8 \cdot \Gamma(7/4)}$$

```

```
>>> integ = Integral(x**y*exp(-x), (x, 0, oo))
>>> integ

$$\int_0^{\infty} x^y e^{-x} dx$$

```

(continues on next page)

(continued from previous page)

```

|  y  -x
|  x  ·e   dx
|
|  0
|
|  >>> integ.doit()
|  { Γ(y + 1)      for re(y) > -1
|
|  ∞
|  {
|  {  y  -x
|  {  x  ·e   dx      otherwise
|  {
|  {  0
|  {

```

This last example returned a Piecewise expression because the integral does not converge unless $\Re(y) > 1$.

Limits

SymPy can compute symbolic limits with the `limit` function. The syntax to compute

$$\lim_{x \rightarrow x_0} f(x)$$

is `limit(f(x), x, x0)`.

```

>>> limit(sin(x)/x, x, 0)
1

```

`limit` should be used instead of `subs` whenever the point of evaluation is a singularity. Even though SymPy has objects to represent ∞ , using them for evaluation is not reliable because they do not keep track of things like rate of growth. Also, things like $\infty - \infty$ and $\frac{\infty}{\infty}$ return `nan` (not-a-number). For example

```

>>> expr = x**2/exp(x)
>>> expr.subs(x, oo)
nan
>>> limit(expr, x, oo)
0

```

Like `Derivative` and `Integral`, `limit` has an unevaluated counterpart, `Limit`. To evaluate it, use `doit`.

```

>>> expr = Limit((cos(x) - 1)/x, x, 0)
>>> expr
      (cos(x) - 1)
lim |—————|
x→0+ | x      |
>>> expr.doit()
0

```

To evaluate a limit at one side only, pass '+' or '-' as a fourth argument to `limit`. For example, to compute

$$\lim_{x \rightarrow 0^+} \frac{1}{x},$$

do

```
>>> limit(1/x, x, 0, '+')
 $\infty$ 
```

As opposed to

```
>>> limit(1/x, x, 0, '-')
 $-\infty$ 
```

Series Expansion

SymPy can compute asymptotic series expansions of functions around a point. To compute the expansion of $f(x)$ around the point $x = x_0$ terms of order x^n , use `f(x).series(x, x0, n)`. `x0` and `n` can be omitted, in which case the defaults `x0=0` and `n=6` will be used.

```
>>> expr = exp(sin(x))
>>> expr.series(x, 0, 4)

$$1 + x + \frac{x^2}{2} + \frac{x^3}{6} + O(x^4)$$

```

The $O(x^4)$ term at the end represents the Landau order term at $x = 0$ (not to be confused with big O notation used in computer science, which generally represents the Landau order term at x where $x \rightarrow \infty$). It means that all x terms with power greater than or equal to x^4 are omitted. Order terms can be created and manipulated outside of series. They automatically absorb higher order terms.

```
>>> x + x**3 + x**6 + O(x**4)

$$x + x^3 + O(x^4)$$

>>> x*O(1)
 $O(x)$ 
```

If you do not want the order term, use the `removeO` method.

```
>>> expr.series(x, 0, 4).removeO()

$$\frac{x^2}{2} + x + 1$$

```

The `O` notation supports arbitrary limit points (other than 0):

```
>>> exp(x - 6).series(x, x0=6)

$$\frac{x^2}{2} - \frac{x^3}{3} + \frac{x^4}{4} - \frac{x^5}{5} + O(x^6)$$

```

(continues on next page)

(continued from previous page)

$$-5 + \frac{(x-6)}{2} + \frac{(x-6)^2}{6} + \frac{(x-6)^3}{24} + \frac{(x-6)^4}{120} + x + O\left(\frac{(x-6)^5}{(x-6)}; x \rightarrow 6\right)$$

Finite differences

So far we have looked at expressions with analytic derivatives and primitive functions respectively. But what if we want to have an expression to estimate a derivative of a curve for which we lack a closed form representation, or for which we don't know the functional values for yet. One approach would be to use a finite difference approach.

The simplest way the differentiate using finite differences is to use the `differentiate_finite` function:

```
>>> f, g = symbols('f g', cls=Function)
>>> differentiate_finite(f(x)*g(x))
-f(x - 1/2)*g(x - 1/2) + f(x + 1/2)*g(x + 1/2)
```

If you already have a `Derivative` instance, you can use the `as_finite_difference` method to generate approximations of the derivative to arbitrary order:

```
>>> f = Function('f')
>>> dfdx = f(x).diff(x)
>>> dfdx.as_finite_difference()
-f(x - 1/2) + f(x + 1/2)
```

here the first order derivative was approximated around x using a minimum number of points (2 for 1st order derivative) evaluated equidistantly using a step-size of 1. We can use arbitrary steps (possibly containing symbolic expressions):

```
>>> f = Function('f')
>>> d2fdx2 = f(x).diff(x, 2)
>>> h = Symbol('h')
>>> d2fdx2.as_finite_difference([-3*h, -h, 2*h])
f(-3*h)  f(-h)  2*f(2*h)
-----  -  -----  +  -----
      2      2      2
    5*h    3*h   15*h
```

If you are just interested in evaluating the weights, you can do so manually:

```
>>> finite_diff_weights(2, [-3, -1, 2], 0)[-1][-1]
[1/5, -1/3, 2/15]
```

note that we only need the last element in the last sublist returned from `finite_diff_weights`. The reason for this is that the function also generates weights for lower derivatives and using fewer points (see the documentation of `finite_diff_weights` for more details).

If using `finite_diff_weights` directly looks complicated, and the `as_finite_difference` method of `Derivative` instances is not flexible enough, you can use `apply_finite_diff` which takes `order`, `x_list`, `y_list` and `x0` as parameters:

```
>>> x_list = [-3, 1, 2]
>>> y_list = symbols('a b c')
>>> apply_finite_diff(1, x_list, y_list, 0)
  3·a   b   2·c
- ---- - - + ----
  20   4   5
```

Solvers

```
>>> from sympy import *
>>> x, y, z = symbols('x y z')
>>> init_printing(use_unicode=True)
```

A Note about Equations

Recall from the [gotchas](#) (page 12) section of this tutorial that symbolic equations in SymPy are not represented by `=` or `==`, but by `Eq`.

```
>>> Eq(x, y)
x = y
```

However, there is an even easier way. In SymPy, any expression not in an `Eq` is automatically assumed to equal 0 by the solving functions. Since $a = b$ if and only if $a - b = 0$, this means that instead of using `x == y`, you can just use `x - y`. For example

```
>>> solveset(Eq(x**2, 1), x)
{-1, 1}
>>> solveset(Eq(x**2 - 1, 0), x)
{-1, 1}
>>> solveset(x**2 - 1, x)
{-1, 1}
```

This is particularly useful if the equation you wish to solve is already equal to 0. Instead of typing `solveset(Eq(expr, 0), x)`, you can just use `solveset(expr, x)`.

Solving Equations Algebraically

The main function for solving algebraic equations is `solveset`. The syntax for `solveset` is `solveset(equation, variable=None, domain=S.Complexes)` Where equations may be in the form of `Eq` instances or expressions that are assumed to be equal to zero.

Please note that there is another function called `solve` which can also be used to solve equations. The syntax is `solve(equations, variables)` However, it is recommended to use `solveset` instead.

When solving a single equation, the output of `solveset` is a `FiniteSet` or an `Interval` or `ImageSet` of the solutions.

```
>>> solveset(x**2 - x, x)
{0, 1}
>>> solveset(x - x, x, domain=S.Reals)
ℝ
>>> solveset(sin(x) - 1, x, domain=S.Reals)

$$\left\{ 2 \cdot n \cdot \pi + \frac{\pi}{2} \mid n \in \mathbb{Z} \right\}$$

```

If there are no solutions, an EmptySet is returned and if it is not able to find solutions then a ConditionSet is returned.

```
>>> solveset(exp(x), x)      # No solution exists
∅
>>> solveset(cos(x) - x, x)  # Not able to find solution
{x | x ∈ ℂ ∧ (-x + cos(x) = 0)}
```

In the solveset module, the linear system of equations is solved using linsolve. In future we would be able to use linsolve directly from solveset. Following is an example of the syntax of linsolve.

- List of Equations Form:

```
>>> linsolve([x + y + z - 1, x + y + 2*z - 3], (x, y, z))
{(-y - 1, y, 2)}
```

- Augmented Matrix Form:

```
>>> linsolve(Matrix([[1, 1, 1, 1], [1, 1, 2, 3]]), (x, y, z))
{(-y - 1, y, 2)}
```

- A*x = b Form

```
>>> M = Matrix(((1, 1, 1, 1), (1, 1, 2, 3)))
>>> system = A, b = M[:, :-1], M[:, -1]
>>> linsolve(system, x, y, z)
{(-y - 1, y, 2)}
```

Note: The order of solution corresponds the order of given symbols.

In the solveset module, the non linear system of equations is solved using nonlinsolve. Following are examples of nonlinsolve.

1. When only real solution is present:

```
>>> a, b, c, d = symbols('a, b, c, d', real=True)
>>> nonlinsolve([a**2 + a, a - b], [a, b])
{(-1, -1), (0, 0)}
>>> nonlinsolve([x*y - 1, x - 2], x, y)
{(2, 1/2)}
```

2. When only complex solution is present:


```
>>> nonlinsolve([x**2 + 1, y**2 + 1], [x, y])
{(-i, -i), (-i, i), (i, -i), (i, i)}
```

3. When both real and complex solution are present:

```
>>> from sympy import sqrt
>>> system = [x**2 - 2*y**2 - 2, x*y - 2]
>>> vars = [x, y]
>>> nonlinsolve(system, vars)
{(-2, -1), (2, 1), (-sqrt(2)*i, sqrt(2)*i), (sqrt(2)*i, -sqrt(2)*i)}
```

```
>>> system = [exp(x) - sin(y), 1/y - 3]
>>> nonlinsolve(system, vars)
{(2*n*i*pi + log(sin(1/3)) | n in ZZ, 1/3)}
```

4. When the system is positive-dimensional system (has infinitely many solutions):

```
>>> nonlinsolve([x*y, x*y - x], [x, y])
{(0, y)}
```

```
>>> system = [a**2 + a*c, a - b]
>>> nonlinsolve(system, [a, b])
{(0, 0), (-c, -c)}
```

Note:

1. The order of solution corresponds the order of given symbols.
2. Currently nonlinsolve doesn't return solution in form of LambertW (if there is solution present in the form of LambertW).

solve can be used for such cases:

```
>>> solve([x**2 - y**2/exp(x)], [x, y], dict=True)
[[{x: -x*sqrt(x/e)}, {x: x*sqrt(x/e)}],
 [{y: -x*sqrt(x/e)}, {y: x*sqrt(x/e)}]]
>>> solve(x**2 - y**2/exp(x), x, dict=True)
[[{x: 2*W(-y/2)}, {x: 2*W(y/2)}],
 [{x: 2*W(-y/2)}, {x: 2*W(y/2)}]]
```

3. Currently nonlinsolve is not properly capable of solving the system of equations having trigonometric functions.

solve can be used for such cases (but does not give all solution):

```
>>> solve([sin(x + y), cos(x - y)], [x, y])
[[(-3*pi/4, 3*pi/4), (-pi/4, pi/4), (pi/4, 3*pi/4), (3*pi/4, pi/4)],
 [(3*pi/4, -pi/4), (pi/4, -3*pi/4), (-pi/4, -pi/4), (-3*pi/4, -3*pi/4)]]
```

`solveset` reports each solution only once. To get the solutions of a polynomial including multiplicity use `roots`.

```
>>> solveset(x**3 - 6*x**2 + 9*x, x)
{0, 3}
>>> roots(x**3 - 6*x**2 + 9*x, x)
{0: 1, 3: 2}
```

The output `{0: 1, 3: 2}` of `roots` means that 0 is a root of multiplicity 1 and 3 is a root of multiplicity 2.

Note: Currently `solveset` is not capable of solving the following types of equations:

- Equations solvable by `LambertW` (Transcendental equation solver).

`solve` can be used for such cases:

```
>>> solve(x*exp(x) - 1, x)
[W(1)]
```

Solving Differential Equations

To solve differential equations, use `dsolve`. First, create an undefined function by passing `cls=Function` to the `symbols` function.

```
>>> f, g = symbols('f g', cls=Function)
```

`f` and `g` are now undefined functions. We can call `f(x)`, and it will represent an unknown function.

```
>>> f(x)
f(x)
```

Derivatives of `f(x)` are unevaluated.

```
>>> f(x).diff(x)
d
—(f(x))
dx
```

(see the [Derivatives](#) (page 40) section for more on derivatives).

To represent the differential equation $f''(x) - 2f'(x) + f(x) = \sin(x)$, we would thus use

```
>>> diffeq = Eq(f(x).diff(x, x) - 2*f(x).diff(x) + f(x), sin(x))
>>> diffeq
```

$$f(x) - 2 \cdot \frac{d}{dx}(f(x)) + \frac{d^2}{dx^2}(f(x)) = \sin(x)$$

To solve the ODE, pass it and the function to solve for to `dsolve`.

```
>>> dsolve(diffeq, f(x))
f(x) = (C1 + C2·x)·ex +  $\frac{\cos(x)}{2}$ 
```

`dsolve` returns an instance of `Eq`. This is because in general, solutions to differential equations cannot be solved explicitly for the function.

```
>>> dsolve(f(x).diff(x)*(1 - sin(f(x))) - 1, f(x))
x - f(x) - cos(f(x)) = C1
```

The arbitrary constants in the solutions from `dsolve` are symbols of the form `C1`, `C2`, `C3`, and so on.

Matrices

```
>>> from sympy import *
>>> init_printing(use_unicode=True)
```

To make a matrix in SymPy, use the `Matrix` object. A matrix is constructed by providing a list of row vectors that make up the matrix. For example, to construct the matrix

$$\begin{bmatrix} 1 & -1 \\ 3 & 4 \\ 0 & 2 \end{bmatrix}$$

use

```
>>> Matrix([[1, -1], [3, 4], [0, 2]])
 $\begin{bmatrix} 1 & -1 \\ 3 & 4 \\ 0 & 2 \end{bmatrix}$ 
```

To make it easy to make column vectors, a list of elements is considered to be a column vector.

```
>>> Matrix([1, 2, 3])
 $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ 
```

Matrices are manipulated just like any other object in SymPy or Python.

```
>>> M = Matrix([[1, 2, 3], [3, 2, 1]])
>>> N = Matrix([0, 1, 1])
>>> M*N
 $\begin{bmatrix} 5 \\ 3 \end{bmatrix}$ 
```

One important thing to note about SymPy matrices is that, unlike every other object in SymPy, they are mutable. This means that they can be modified in place, as we will see below. The downside to this is that `Matrix` cannot be used in places that require immutability, such as inside other SymPy expressions or as keys to dictionaries. If you need an immutable version of `Matrix`, use `ImmutableMatrix`.

Basic Operations

Here are some basic operations on `Matrix`.

Shape

To get the shape of a matrix, use `shape()` (page 1155) function.

```
>>> from sympy import shape
>>> M = Matrix([[1, 2, 3], [-2, 0, 4]])
>>> M
[1  2  3]
[-2 0  4]
>>> shape(M)
(2, 3)
```

Accessing Rows and Columns

To get an individual row or column of a matrix, use `row` or `col`. For example, `M.row(0)` will get the first row. `M.col(-1)` will get the last column.

```
>>> M.row(0)
[1  2  3]
>>> M.col(-1)
[3]
[4]
```

Deleting and Inserting Rows and Columns

To delete a row or column, use `row_del` or `col_del`. These operations will modify the `Matrix` **in place**.

```
>>> M.col_del(0)
>>> M
[2  3]
[0  4]
>>> M.row_del(1)
>>> M
[2  3]
```

To insert rows or columns, use `row_insert` or `col_insert`. These operations **do not** operate in place.

```
>>> M
[2  3]
>>> M = M.row_insert(1, Matrix([[0, 4]]))
>>> M
[2  3]
[0  4]
>>> M = M.col_insert(0, Matrix([1, -2]))
>>> M
[1  2  3]
[-2 0  4]
```

Unless explicitly stated, the methods mentioned below do not operate in place. In general, a method that does not operate in place will return a new `Matrix` and a method that does operate in place will return `None`.

Basic Methods

As noted above, simple operations like addition and multiplication are done just by using `+`, `*`, and `**`. To find the inverse of a matrix, just raise it to the `-1` power.

```
>>> M = Matrix([[1, 3], [-2, 3]])
>>> N = Matrix([[0, 3], [0, 7]])
>>> M + N
[1  6]
[-2 10]
>>> M*N
[0  24]
[0  15]
>>> 3*M
[3  9]
[-6 9]
>>> M**2
[-5 12]
[-8  3]
>>> M**-1
[1/3 -1/3]
[2/9  1/9]
>>> N**-1
Traceback (most recent call last):
...
NonInvertibleMatrixError: Matrix det == 0; not invertible.
```

To take the transpose of a Matrix, use `T`.

```
>>> M = Matrix([[1, 2, 3], [4, 5, 6]])
>>> M
[1 2 3]
[4 5 6]
>>> M.T
[1 4]
[2 5]
[3 6]
```

Matrix Constructors

Several constructors exist for creating common matrices. To create an identity matrix, use `eye`. `eye(n)` will create an $n \times n$ identity matrix.

```
>>> eye(3)
[1 0 0]
[0 1 0]
[0 0 1]
>>> eye(4)
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

To create a matrix of all zeros, use `zeros`. `zeros(n, m)` creates an $n \times m$ matrix of 0s.

```
>>> zeros(2, 3)
[0 0 0]
[0 0 0]
```

Similarly, `ones` creates a matrix of ones.

```
>>> ones(3, 2)
[1 1]
[1 1]
[1 1]
```

To create diagonal matrices, use `diag`. The arguments to `diag` can be either numbers or matrices. A number is interpreted as a 1×1 matrix. The matrices are stacked diagonally. The remaining elements are filled with 0s.

```
>>> diag(1, 2, 3)
[1  0  0]
[0  2  0]
[0  0  3]
>>> diag(-1, ones(2, 2), Matrix([5, 7, 5]))
[-1  0  0  0]
[ 0  1  1  0]
[ 0  1  1  0]
[ 0  0  0  5]
[ 0  0  0  7]
[ 0  0  0  5]
```

Advanced Methods

Determinant

To compute the determinant of a matrix, use `det`.

```
>>> M = Matrix([[1, 0, 1], [2, -1, 3], [4, 3, 2]])
>>> M
[1  0  1]
[2 -1  3]
[4  3  2]
>>> M.det()
-1
```

RREF

To put a matrix into reduced row echelon form, use `rref`. `rref` returns a tuple of two elements. The first is the reduced row echelon form, and the second is a tuple of indices of the pivot columns.

```
>>> M = Matrix([[1, 0, 1, 3], [2, 3, 4, 7], [-1, -3, -3, -4]])
>>> M
[1  0  1  3 ]
```

(continues on next page)

(continued from previous page)

```

| 2  3  4  7 |
| -1 -3 -3 -4 |
>>> M.rref()
([1 0 1 3], (0, 1))
| 0 1 2/3 1/3 |
| 0 0 0 0 |

```

Note: The first element of the tuple returned by `rref` is of type `Matrix`. The second is of type tuple.

Nullspace

To find the nullspace of a matrix, use `nullspace`. `nullspace` returns a list of column vectors that span the nullspace of the matrix.

```

>>> M = Matrix([[1, 2, 3, 0, 0], [4, 10, 0, 0, 1]])
>>> M
| 1  2  3  0  0 |
| 4 10  0  0  1 |
>>> M.nullspace()
[[-15], [0], [1]]
| 6 | | 0 | | -1/2 |
| 1 | | 0 | | 0 |
| 0 | | 1 | | 0 |
| 0 | | 0 | | 1 |

```

Columnspace

To find the column space of a matrix, use `columnspace`. `columnspace` returns a list of column vectors that span the column space of the matrix.

```

>>> M = Matrix([[1, 1, 2], [2, 1, 3], [3, 1, 4]])
>>> M
| 1  1  2 |
| 2  1  3 |

```

(continues on next page)

(continued from previous page)

```

|
| 3  1  4 |
|>>> M.columnspace()
| [[1]  [1] |
|  [2]  [1] |
|  [3]  [1] |
|

```

Eigenvalues, Eigenvectors, and Diagonalization

To find the eigenvalues of a matrix, use `eigenvals`. `eigenvals` returns a dictionary of eigenvalue: algebraic_multiplicity pairs (similar to the output of [roots](#) (page 49)).

```

>>> M = Matrix([[3, -2, 4, -2], [5, 3, -3, -2], [5, -2, 2, -2], [5, -2, -3,
→ 3]])
>>> M
[3  -2  4  -2]
[5  3  -3  -2]
[5  -2  2  -2]
[5  -2 -3  3]
>>> M.eigenvals()
{-2: 1, 3: 1, 5: 2}

```

This means that M has eigenvalues -2, 3, and 5, and that the eigenvalues -2 and 3 have algebraic multiplicity 1 and that the eigenvalue 5 has algebraic multiplicity 2.

To find the eigenvectors of a matrix, use `eigenvects`. `eigenvects` returns a list of tuples of the form (eigenvalue, algebraic_multiplicity, [eigenvectors]).

```

>>> M.eigenvects()
[( (-2, 1, [[0], [1], [1], [1]]), (3, 1, [[1], [1], [1], [1]]), (5, 2, [[1], [0], [1], [1]], [[0], [-1], [0], [1]])]

```

This shows us that, for example, the eigenvalue 5 also has geometric multiplicity 2, because it has two eigenvectors. Because the algebraic and geometric multiplicities are the same for all the eigenvalues, M is diagonalizable.

To diagonalize a matrix, use `diagonalize`. `diagonalize` returns a tuple (P, D) , where D is diagonal and $M = PDP^{-1}$.

```
>>> P, D = M.diagonalize()
>>> P

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & -1 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix}$$

>>> D

$$\begin{bmatrix} -2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 5 \end{bmatrix}$$

>>> P*D*P**-1

$$\begin{bmatrix} 3 & -2 & 4 & -2 \\ 5 & 3 & -3 & -2 \\ 5 & -2 & 2 & -2 \\ 5 & -2 & -3 & 3 \end{bmatrix}$$

>>> P*D*P**-1 == M
True
```

Quick Tip

`lambda` is a reserved keyword in Python, so to create a Symbol called λ , while using the same names for SymPy Symbols and Python variables, use `lamda` (without the b). It will still pretty print as λ .

Note that since `eigenvects` also includes the eigenvalues, you should use it instead of `eigenvals` if you also want the eigenvectors. However, as computing the eigenvectors may often be costly, `eigenvals` should be preferred if you only wish to find the eigenvalues.

If all you want is the characteristic polynomial, use `charpoly`. This is more efficient than `eigenvals`, because sometimes symbolic roots can be expensive to calculate.

```
>>> lamda = symbols('lamda')
>>> p = M.charpoly(lamda)
>>> factor(p.as_expr())

$$(\lambda^2 - 5) \cdot (\lambda - 3) \cdot (\lambda + 2)$$

```

Possible Issues

Zero Testing

If your matrix operations are failing or returning wrong answers, the common reasons would likely be from zero testing. If there is an expression not properly zero-tested, it can possibly bring issues in finding pivots for gaussian elimination, or deciding whether the matrix is invertible, or any high level functions which relies on the prior procedures.

Currently, the SymPy's default method of zero testing `_iszero` is only guaranteed to be accurate in some limited domain of numerics and symbols, and any complicated expressions beyond its decidability are treated as `None`, which behaves similarly to logical `False`.

The list of methods using zero testing procedures are as follows:

`echelon_form` , `is_echelon` , `rank` , `rref` , `nullspace` , `eigenvecs` , `inverse_ADJ` , `inverse_GE` , `inverse_LU` , `LUdecomposition` , `LUdecomposition_Simple` , `LUsolve`

They have property `iszerofunc` opened up for user to specify zero testing method, which can accept any function with single input and boolean output, while being defaulted with `_iszero`.

Here is an example of solving an issue caused by undertested zero. While the output for this particular matrix has since been improved, the technique below is still of interest.¹²³

```
>>> from sympy import *
>>> q = Symbol("q", positive = True)
>>> m = Matrix([
... [-2*cosh(q/3),      exp(-q),      1],
... [      exp(q), -2*cosh(q/3),      1],
... [      1,      1, -2*cosh(q/3)]]])
>>> m.nullspace()
[]
```

You can trace down which expression is being underevaluated, by injecting a custom zero test with warnings enabled.

```
>>> import warnings
>>>
>>> def my_iszero(x):
...     try:
...         result = x.is_zero
...     except AttributeError:
...         result = None
...
...     # Warnings if evaluated into None
...     if result is None:
...         warnings.warn("Zero testing of {} evaluated into None".format(x))
...     return result
>>> m.nullspace(iszerofunc=my_iszero)
__main__:9: UserWarning: Zero testing of 4*cosh(q/3)**2 - 1 evaluated into
↳ None
```

(continues on next page)

¹ Inspired by <https://gitter.im/sympy/sympy?at=5b7c3e8ee5b40332abdb206c>

² Discovered from <https://github.com/sympy/sympy/issues/15141>

³ Improved by <https://github.com/sympy/sympy/pull/19548>

(continued from previous page)

```

__main__:9: UserWarning: Zero testing of (-exp(q) - 2*cosh(q/3))*(-2*cosh(q/
→3) - exp(-q)) - (4*cosh(q/3)**2 - 1)**2 evaluated into None
__main__:9: UserWarning: Zero testing of 2*exp(q)*cosh(q/3) - 16*cosh(q/3)**4
→+ 12*cosh(q/3)**2 + 2*exp(-q)*cosh(q/3) evaluated into None
__main__:9: UserWarning: Zero testing of -(4*cosh(q/3)**2 - 1)*exp(-q) -
→2*cosh(q/3) - exp(-q) evaluated into None
[]

```

In this case, $(-exp(q) - 2*cosh(q/3))*(-2*cosh(q/3) - exp(-q)) - (4*cosh(q/3)**2 - 1)**2$ should yield zero, but the zero testing had failed to catch. possibly meaning that a stronger zero test should be introduced. For this specific example, rewriting to exponentials and applying simplify would make zero test stronger for hyperbolics, while being harmless to other polynomials or transcendental functions.

```

>>> def my_iszero(x):
...     try:
...         result = x.rewrite(exp).simplify().is_zero
...     except AttributeError:
...         result = None
...
...     # Warnings if evaluated into None
...     if result is None:
...         warnings.warn("Zero testing of {} evaluated into None".format(x))
...     return result
...
>>> m.nullspace(iszerofunc=my_iszero)
__main__:9: UserWarning: Zero testing of -2*cosh(q/3) - exp(-q) evaluated
→into None

```

$$\begin{bmatrix} \begin{pmatrix} q \\ -e^{-q} - 2 \cdot \cosh\left(\frac{q}{3}\right) \end{pmatrix} \cdot e^{-q} + 4 \cdot \cosh\left(\frac{2q}{3}\right) - 1 \\ \hline 2 \cdot \left(4 \cdot \cosh\left(\frac{2q}{3}\right) - 1 \right) \cdot \cosh\left(\frac{q}{3}\right) \\ - \begin{pmatrix} q \\ -e^{-q} - 2 \cdot \cosh\left(\frac{q}{3}\right) \end{pmatrix} \\ \hline 4 \cdot \cosh\left(\frac{2q}{3}\right) - 1 \\ \hline 1 \end{pmatrix}$$

You can clearly see nullspace returning proper result, after injecting an alternative zero test.

Note that this approach is only valid for some limited cases of matrices containing only numerics, hyperbolics, and exponentials. For other matrices, you should use different method opted for their domains.

Possible suggestions would be either taking advantage of rewriting and simplifying, with tradeoff of speed⁴, or using random numeric testing, with tradeoff of accuracy⁵.

If you wonder why there is no generic algorithm for zero testing that can work with any symbolic entities, it's because of the constant problem stating that zero testing is undecidable⁶, and not only the SymPy, but also other computer algebra systems⁷⁸ would face the same fundamental issue.

However, discovery of any zero test failings can provide some good examples to improve SymPy, so if you have encountered one, you can report the issue to SymPy issue tracker⁹ to get detailed help from the community.

Advanced Expression Manipulation

In this section, we discuss some ways that we can perform advanced manipulation of expressions.

Understanding Expression Trees

Before we can do this, we need to understand how expressions are represented in SymPy. A mathematical expression is represented as a tree. Let us take the expression $x^2 + xy$, i.e., `x**2 + x*y`. We can see what this expression looks like internally by using `srepr`

```
>>> from sympy import *
>>> x, y, z = symbols('x y z')
```

```
>>> expr = x**2 + x*y
>>> srepr(expr)
"Add(Pow(Symbol('x'), Integer(2)), Mul(Symbol('x'), Symbol('y')))"
```

The easiest way to tear this apart is to look at a diagram of the expression tree:

⁴ Suggested from <https://github.com/sympy/sympy/issues/10120>

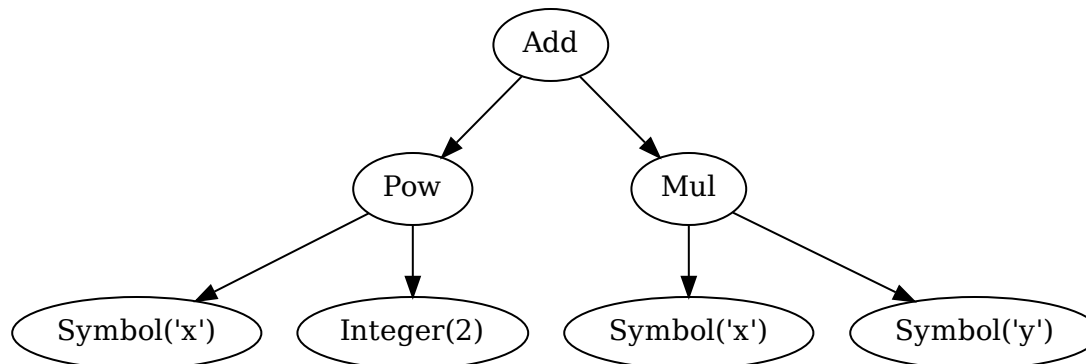
⁵ Suggested from <https://github.com/sympy/sympy/issues/10279>

⁶ https://en.wikipedia.org/wiki/Constant_problem

⁷ How mathematica tests zero <https://reference.wolfram.com/language/ref/PossibleZeroQ.html>

⁸ How matlab tests zero https://www.mathworks.com/help/symbolic/mupad_ref/iszero.html

⁹ <https://github.com/sympy/sympy/issues>



Note: The above diagram was made using [Graphviz](#) and the [dotprint](#) (page 2188) function.

First, let’s look at the leaves of this tree. Symbols are instances of the class `Symbol`. While we have been doing

```
>>> x = symbols('x')
```

we could have also done

```
>>> x = Symbol('x')
```

Either way, we get a `Symbol` with the name “x”¹. For the number in the expression, 2, we got `Integer(2)`. `Integer` is the SymPy class for integers. It is similar to the Python built-in type `int`, except that `Integer` plays nicely with other SymPy types.

When we write `x**2`, this creates a `Pow` object. `Pow` is short for “power”.

```
>>> srepr(x**2)
"Pow(Symbol('x'), Integer(2))"
```

We could have created the same object by calling `Pow(x, 2)`

```
>>> Pow(x, 2)
x**2
```

Note that in the `srepr` output, we see `Integer(2)`, the SymPy version of integers, even though technically, we input 2, a Python `int`. In general, whenever you combine a SymPy object with a non-SymPy object via some function or operation, the non-SymPy object will be converted into a SymPy object. The function that does this is `sympify`².

```
>>> type(2)
<... 'int'>
```

(continues on next page)

¹ We have been using `symbols` instead of `Symbol` because it automatically splits apart strings into multiple `Symbols`. `symbols('x y z')` returns a tuple of three `Symbols`. `Symbol('x y z')` returns a single `Symbol` called `x y z`.

² Technically, it is an internal function called `_sympify`, which differs from `sympify` in that it does not convert strings. `x + '2'` is not allowed.

(continued from previous page)

```
>>> type(sympify(2))
<class 'sympy.core.numbers.Integer'>
```

We have seen that x^{**2} is represented as `Pow(x, 2)`. What about $x*y$? As we might expect, this is the multiplication of x and y . The SymPy class for multiplication is `Mul`.

```
>>> srepr(x*y)
"Mul(Symbol('x'), Symbol('y'))"
```

Thus, we could have created the same object by writing `Mul(x, y)`.

```
>>> Mul(x, y)
x*y
```

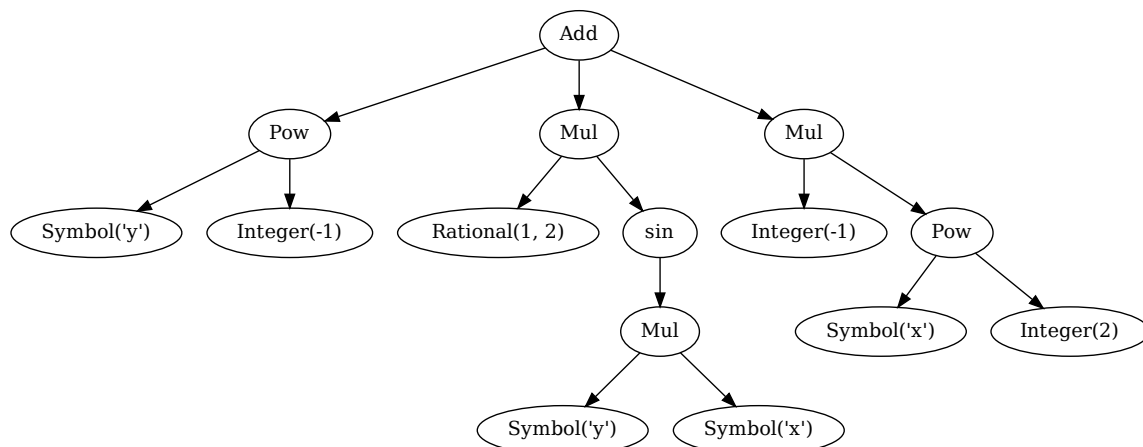
Now we get to our final expression, $x^{**2} + x*y$. This is the addition of our last two objects, `Pow(x, 2)`, and `Mul(x, y)`. The SymPy class for addition is `Add`, so, as you might expect, to create this object, we use `Add(Pow(x, 2), Mul(x, y))`.

```
>>> Add(Pow(x, 2), Mul(x, y))
x**2 + x*y
```

SymPy expression trees can have many branches, and can be quite deep or quite broad. Here is a more complicated example

```
>>> expr = sin(x*y)/2 - x**2 + 1/y
>>> srepr(expr)
"Add(Mul(Integer(-1), Pow(Symbol('x'), Integer(2))), Mul(Rational(1, 2),
sin(Mul(Symbol('x'), Symbol('y')))), Pow(Symbol('y'), Integer(-1)))"
```

Here is a diagram

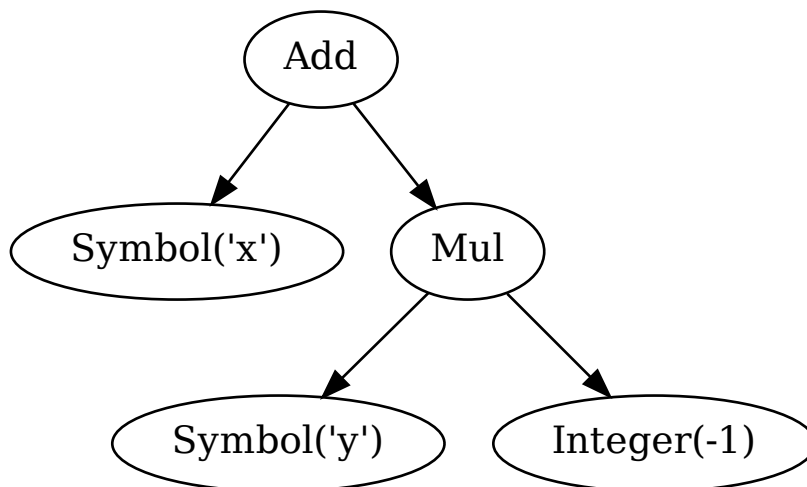


This expression reveals some interesting things about SymPy expression trees. Let's go through them one by one.

Let's first look at the term x^{**2} . As we expected, we see `Pow(x, 2)`. One level up, we see we have `Mul(-1, Pow(x, 2))`. There is no subtraction class in SymPy. $x - y$ is represented as

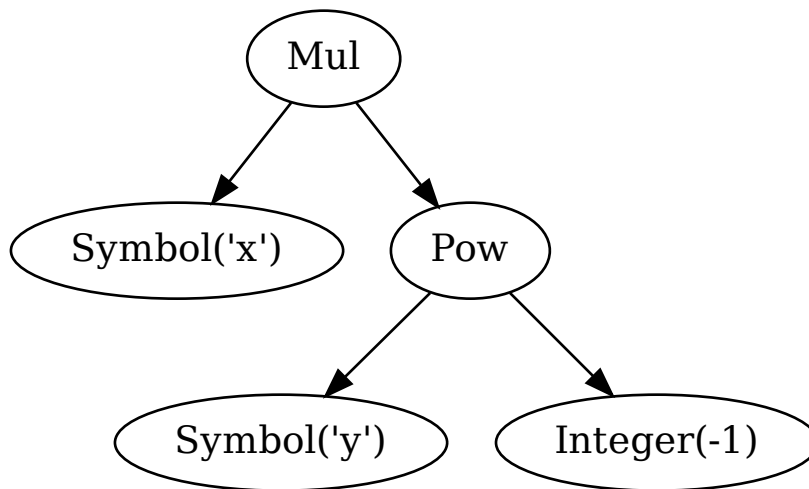
$x + -y$, or, more completely, $x + -1*y$, i.e., `Add(x, Mul(-1, y))`.

```
>>> srepr(x - y)
"Add(Symbol('x'), Mul(Integer(-1), Symbol('y')))"
```



Next, look at $1/y$. We might expect to see something like `Div(1, y)`, but similar to subtraction, there is no class in SymPy for division. Rather, division is represented by a power of -1. Hence, we have `Pow(y, -1)`. What if we had divided something other than 1 by y , like x/y ? Let's see.

```
>>> expr = x/y
>>> srepr(expr)
"Mul(Symbol('x'), Pow(Symbol('y'), Integer(-1)))"
```

We see that x/y is represented as $x*y**-1$, i.e., `Mul(x, Pow(y, -1))`.

Finally, let's look at the $\sin(x*y)/2$ term. Following the pattern of the previous example, we might expect to see `Mul(sin(x*y), Pow(Integer(2), -1))`. But instead, we have `Mul(Rational(1, 2), sin(x*y))`. Rational numbers are always combined into a single term in a multiplication, so that when we divide by 2, it is represented as multiplying by $1/2$.

Finally, one last note. You may have noticed that the order we entered our expression and the order that it came out from `srepr` or in the graph were different. You may have also noticed this phenomenon earlier in the tutorial. For example

```
>>> 1 + x
x + 1
```

This because in SymPy, the arguments of the commutative operations `Add` and `Mul` are stored in an arbitrary (but consistent!) order, which is independent of the order inputted (if you're worried about noncommutative multiplication, don't be. In SymPy, you can create noncommutative Symbols using `Symbol('A', commutative=False)`, and the order of multiplication for noncommutative Symbols is kept the same as the input). Furthermore, as we shall see in the next section, the printing order and the order in which things are stored internally need not be the same either.

Quick Tip

The way an expression is represented internally and the way it is printed are often not the same.

In general, an important thing to keep in mind when working with SymPy expression trees is this: the internal representation of an expression and the way it is printed need not be the same. The same is true for the input form. If some expression manipulation algorithm is not working in the way you expected it to, chances are, the internal representation of the object is different from what you thought it was.

Recurring through an Expression Tree

Now that you know how expression trees work in SymPy, let's look at how to dig our way through an expression tree. Every object in SymPy has two very important attributes, `func`, and `args`.

`func`

`func` is the head of the object. For example, `(x*y).func` is `Mul`. Usually it is the same as the class of the object (though there are exceptions to this rule).

Two notes about `func`. First, the class of an object need not be the same as the one used to create it. For example

```
>>> expr = Add(x, x)
>>> expr.func
<class 'sympy.core.mul.Mul'>
```

We created `Add(x, x)`, so we might expect `expr.func` to be `Add`, but instead we got `Mul`. Why is that? Let's take a closer look at `expr`.

```
>>> expr
2*x
```

`Add(x, x)`, i.e., $x + x$, was automatically converted into `Mul(2, x)`, i.e., $2*x$, which is a `Mul`. SymPy classes make heavy use of the `__new__` class constructor, which, unlike `__init__`, allows a different class to be returned from the constructor.

Second, some classes are special-cased, usually for efficiency reasons³.

```
>>> Integer(2).func
<class 'sympy.core.numbers.Integer'>
>>> Integer(0).func
<class 'sympy.core.numbers.Zero'>
>>> Integer(-1).func
<class 'sympy.core.numbers.NegativeOne'>
```

For the most part, these issues will not bother us. The special classes `Zero`, `One`, `NegativeOne`, and so on are subclasses of `Integer`, so as long as you use `isinstance`, it will not be an issue.

³ Classes like `One` and `Zero` are singletonized, meaning that only one object is ever created, no matter how many times the class is called. This is done for space efficiency, as these classes are very common. For example, `Zero` might occur very often in a sparse matrix represented densely. As we have seen, `NegativeOne` occurs any time we have $-x$ or $1/x$. It is also done for speed efficiency because singletonized objects can be compared by `is`. The unique objects for each singletonized class can be accessed from the `S` object.

args

args are the top-level arguments of the object. $(x*y).args$ would be (x, y) . Let's look at some examples

```
>>> expr = 3*y**2*x
>>> expr.func
<class 'sympy.core.mul.Mul'>
>>> expr.args
(3, x, y**2)
```

From this, we can see that $expr == Mul(3, y**2, x)$. In fact, we can see that we can completely reconstruct $expr$ from its `func` and its `args`.

```
>>> expr.func(*expr.args)
3*x*y**2
>>> expr == expr.func(*expr.args)
True
```

Note that although we entered $3*y**2*x$, the args are $(3, x, y**2)$. In a `Mul`, the Rational coefficient will come first in the args, but other than that, the order of everything else follows no special pattern. To be sure, though, there is an order.

```
>>> expr = y**2*3*x
>>> expr.args
(3, x, y**2)
```

`Mul`'s args are sorted, so that the same `Mul` will have the same args. But the sorting is based on some criteria designed to make the sorting unique and efficient that has no mathematical significance.

The `srepr` form of our `expr` is `Mul(3, x, Pow(y, 2))`. What if we want to get at the args of `Pow(y, 2)`. Notice that the $y**2$ is in the third slot of `expr.args`, i.e., `expr.args[2]`.

```
>>> expr.args[2]
y**2
```

So to get the args of this, we call `expr.args[2].args`.

```
>>> expr.args[2].args
(y, 2)
```

Now what if we try to go deeper. What are the args of `y`. Or `2`. Let's see.

```
>>> y.args
()
>>> Integer(2).args
()
```

They both have empty args. In SymPy, empty args signal that we have hit a leaf of the expression tree.

So there are two possibilities for a SymPy expression. Either it has empty args, in which case it is a leaf node in any expression tree, or it has args, in which case, it is a branch node of any expression tree. When it has args, it can be completely rebuilt from its `func` and its `args`. This is expressed in the key invariant.

Key Invariant

Every well-formed SymPy expression must either have empty args or satisfy `expr == expr.func(*expr.args)`.

(Recall that in Python if `a` is a tuple, then `f(*a)` means to call `f` with arguments from the elements of `a`, e.g., `f(*(1, 2, 3))` is the same as `f(1, 2, 3)`.)

This key invariant allows us to write simple algorithms that walk expression trees, change them, and rebuild them into new expressions.

Walking the Tree

With this knowledge, let's look at how we can recurse through an expression tree. The nested nature of `args` is a perfect fit for recursive functions. The base case will be empty `args`. Let's write a simple function that goes through an expression and prints all the args at each level.

```
>>> def pre(expr):  
...     print(expr)  
...     for arg in expr.args:  
...         pre(arg)
```

See how nice it is that `()` signals leaves in the expression tree. We don't even have to write a base case for our recursion; it is handled automatically by the for loop.

Let's test our function.

```
>>> expr = x*y + 1  
>>> pre(expr)  
x*y + 1  
1  
x*y  
x  
y
```

Can you guess why we called our function `pre`? We just wrote a pre-order traversal function for our expression tree. See if you can write a post-order traversal function.

Such traversals are so common in SymPy that the generator functions `preorder_traversal` and `postorder_traversal` are provided to make such traversals easy. We could have also written our algorithm as

```
>>> for arg in preorder_traversal(expr):  
...     print(arg)  
x*y + 1  
1  
x*y  
x  
y
```

Prevent expression evaluation

There are generally two ways to prevent the evaluation, either pass an `evaluate=False` parameter while constructing the expression, or create an evaluation stopper by wrapping the expression with `UnevaluatedExpr`.

For example:

```
>>> from sympy import Add
>>> from sympy.abc import x, y, z
>>> x + x
2*x
>>> Add(x, x)
2*x
>>> Add(x, x, evaluate=False)
x + x
```

If you don't remember the class corresponding to the expression you want to build (operator overloading usually assumes `evaluate=True`), just use `sympify` and pass a string:

```
>>> from sympy import sympify
>>> sympify("x + x", evaluate=False)
x + x
```

Note that `evaluate=False` won't prevent future evaluation in later usages of the expression:

```
>>> expr = Add(x, x, evaluate=False)
>>> expr
x + x
>>> expr + x
3*x
```

That's why the class `UnevaluatedExpr` comes handy. `UnevaluatedExpr` is a method provided by SymPy which lets the user keep an expression unevaluated. By *unevaluated* it is meant that the value inside of it will not interact with the expressions outside of it to give simplified outputs. For example:

```
>>> from sympy import UnevaluatedExpr
>>> expr = x + UnevaluatedExpr(x)
>>> expr
x + x
>>> x + expr
2*x + x
```

The x remaining alone is the x wrapped by `UnevaluatedExpr`. To release it:

```
>>> (x + expr).doit()
3*x
```

Other examples:

```
>>> from sympy import *
>>> from sympy.abc import x, y, z
>>> uexpr = UnevaluatedExpr(S.One*5/7)*UnevaluatedExpr(S.One*3/4)
```

(continues on next page)

(continued from previous page)

```
>>> uexpr
(5/7)*(3/4)
>>> x*UnevaluatedExpr(1/x)
x*1/x
```

A point to be noted is that `UnevaluatedExpr` cannot prevent the evaluation of an expression which is given as argument. For example:

```
>>> expr1 = UnevaluatedExpr(x + x)
>>> expr1
2*x
>>> expr2 = sympify('x + x', evaluate=False)
>>> expr2
x + x
```

Remember that `expr2` will be evaluated if included into another expression. Combine both of the methods to prevent both inside and outside evaluations:

```
>>> UnevaluatedExpr(sympify("x + x", evaluate=False)) + y
y + (x + x)
```

`UnevaluatedExpr` is supported by SymPy printers and can be used to print the result in different output forms. For example

```
>>> from sympy import latex
>>> uexpr = UnevaluatedExpr(S.One*5/7)*UnevaluatedExpr(S.One*3/4)
>>> print(latex(uexpr))
\frac{5}{7} \cdot \frac{3}{4}
```

In order to release the expression and get the evaluated LaTeX form, just use `.doit()`:

```
>>> print(latex(uexpr.doit()))
\frac{15}{28}
```

What's Next

Congratulations on finishing the SymPy tutorial!

If you are a developer interested in using SymPy in your code, please visit the [How-to Guides](#) (page 71) which discuss key developer tasks.

Intermediate SymPy users and developers might want to visit the [Explanations](#) (page 141) section for common pitfalls and advanced topics.

The [SymPy API Reference](#) (page 185) has a detailed description of the SymPy API.

If you are interested in contributing to SymPy, visit the [contribution guides](#) (page 2983) and the full [Development Workflow](#) guide on the SymPy wiki.

HOW-TO GUIDES

How-to guides are step-by-step instructions on how to do specific tasks.

For a deeper and elaborate exploration of other SymPy topics, see the [Explanations](#) (page 141) and [API reference](#) (page 185) sections.

3.1 Assumptions

This page outlines the core assumptions system in SymPy. It explains what the core assumptions system is, how the assumptions system is used and what the different assumptions predicates mean.

Note: This page describes the core assumptions system also often referred to as the “old assumptions” system. There is also a “new assumptions” system which is described elsewhere. Note that the system described here is actually the system that is widely used in SymPy. The “new assumptions” system is not really used anywhere in SymPy yet and the “old assumptions” system will not be removed. At the time of writing (SymPy 1.7) it is still recommended for users to use the old assumption system.

Firstly we consider what happens when taking the square root of the square of a concrete integer such as 2 or -2 :

```
>>> from sympy import sqrt
>>> sqrt(2**2)
2
>>> sqrt((-2)**2)
2
>>> x = 2
>>> sqrt(x**2)
2
>>> sqrt(x**2) == x
True
>>> y = -2
>>> sqrt(y**2) == y
False
>>> sqrt(y**2) == -y
True
```

What these examples demonstrate is that for a positive number x we have $\sqrt{x^2} = x$ whereas for a negative number we would instead have $\sqrt{x^2} = -x$. That may seem obvious but the

situation can be more surprising when working with a symbol rather than an explicit number. For example

```
>>> from sympy import Symbol, simplify
>>> x = Symbol('x')
>>> sqrt(x**2)
sqrt(x**2)
```

It might look as if that should simplify to x but it does not even if `simplify()` (page 661) is used:

```
>>> simplify(sqrt(x**2))
sqrt(x**2)
```

This is because SymPy will refuse to simplify this expression if the simplification is not valid for *every* possible value of x . By default the symbol x is considered only to represent something roughly like an arbitrary complex number and the obvious simplification here is only valid for positive real numbers. Since x is not known to be positive or even real no simplification of this expression is possible.

We can tell SymPy that a symbol represents a positive real number when creating the symbol and then the simplification will happen automatically:

```
>>> y = Symbol('y', positive=True)
>>> sqrt(y**2)
y
```

This is what is meant by “assumptions” in SymPy. If the symbol y is created with `positive=True` then SymPy will *assume* that it represents a positive real number rather than an arbitrary complex or possibly infinite number. That *assumption* can make it possible to simplify expressions or might allow other manipulations to work. It is usually a good idea to be as precise as possible about the assumptions on a symbol when creating it.

3.1.1 The (old) assumptions system

There are two sides to the assumptions system. The first side is that we can declare assumptions on a symbol when creating the symbol. The other side is that we can query the assumptions on any expression using the corresponding `is_*` attribute. For example:

```
>>> x = Symbol('x', positive=True)
>>> x.is_positive
True
```

We can query assumptions on any expression not just a symbol:

```
>>> x = Symbol('x', positive=True)
>>> expr = 1 + x**2
>>> expr
x**2 + 1
>>> expr.is_positive
True
>>> expr.is_negative
False
```


The values given in an assumptions query use three-valued “fuzzy” logic. Any query can return True, False, or None where None should be interpreted as meaning that the result is *unknown*.

```
>>> x = Symbol('x')
>>> y = Symbol('y', positive=True)
>>> z = Symbol('z', negative=True)
>>> print(x.is_positive)
None
>>> print(y.is_positive)
True
>>> print(z.is_positive)
False
```

Note: We need to use print in the above examples because the special value None does not display by default in the Python interpreter.

There are several reasons why an assumptions query might give None. It is possible that the query is *unknowable* as in the case of x above. Since x does not have any assumptions declared it roughly represents an arbitrary complex number. An arbitrary complex number *might* be a positive real number but it also might *not* be. Without further information there is no way to resolve the query x.is_positive.

Another reason why an assumptions query might give None is that there does in many cases the problem of determining whether an expression is e.g. positive is *undecidable*. That means that there does not exist an algorithm for answering the query in general. For some cases an algorithm or at least a simple check would be possible but has not yet been implemented although it could be added to SymPy.

The final reason that an assumptions query might give None is just that the assumptions system does not try very hard to answer complicated queries. The system is intended to be fast and uses simple heuristic methods to conclude a True or False answer in common cases. For example any sum of positive terms is positive so:

```
>>> from sympy import symbols
>>> x, y = symbols('x, y', positive=True)
>>> expr = x + y
>>> expr
x + y
>>> expr.is_positive
True
```

The last example is particularly simple so the assumptions system is able to give a definite answer. If the sum involved a mix of positive or negative terms it would be a harder query:

```
>>> x = Symbol('x', real=True)
>>> expr = 1 + (x - 2)**2
>>> expr
(x - 2)**2 + 1
>>> expr.is_positive
True
>>> expr2 = expr.expand()
>>> expr2
```

(continues on next page)

(continued from previous page)

```
x**2 - 4*x + 5
>>> print(expr2.is_positive)
None
```

Ideally that last example would give `True` rather than `None` because the expression is always positive for any real value of x (and x has been assumed real). The assumptions system is intended to be efficient though: it is expected many more complex queries will not be fully resolved. This is because assumptions queries are primarily used internally by SymPy as part of low-level calculations. Making the system more comprehensive would slow SymPy down.

Note that in fuzzy logic giving an indeterminate result `None` is never a contradiction. If it is possible to infer a definite `True` or `False` result when resolving a query then that is better than returning `None`. However a result of `None` is not a *bug*. Any code that uses the assumptions system needs to be prepared to handle all three cases for any query and should not presume that a definite answer will always be given.

The assumptions system is not just for symbols or for complex expressions. It can also be used for plain SymPy integers and other objects. The assumptions predicates are available on any instance of *Basic* (page 927) which is the superclass for most classes of SymPy objects. A plain Python `int` is not a *Basic* (page 927) instance and can not be used to query assumptions predicates. We can “sympify” regular Python objects to become SymPy objects with *sympify()* (page 918) or *S* (*SingletonRegistry* (page 945)) and then the assumptions system can be used:

```
>>> from sympy import S
>>> x = 2
>>> x.is_positive
Traceback (most recent call last):
...
AttributeError: 'int' object has no attribute 'is_positive'
>>> x = S(2)
>>> type(x)
<class 'sympy.core.numbers.Integer'>
>>> x.is_positive
True
```

3.1.2 Gotcha: symbols with different assumptions

In SymPy it is possible to declare two symbols with different names and they will implicitly be considered equal under *structural equality*:

```
>>> x1 = Symbol('x')
>>> x2 = Symbol('x')
>>> x1
x
>>> x2
x
>>> x1 == x2
True
```

However if the symbols have different assumptions then they will be considered to represent distinct symbols:

```
>>> x1 = Symbol('x', positive=True)
>>> x2 = Symbol('x')
>>> x1
x
>>> x2
x
>>> x1 == x2
False
```

One way to simplify an expression is to use the `posify()` (page 668) function which will replace all symbols in an expression with symbols that have the assumption `positive=True` (unless that contradicts any existing assumptions for the symbol):

```
>>> from sympy import jsonify, exp
>>> x = Symbol('x')
>>> expr = exp(sqrt(x**2))
>>> expr
exp(sqrt(x**2))
>>> jsonify(expr)
(exp(_x), {_x: x})
>>> expr2, rep = jsonify(expr)
>>> expr2
exp(_x)
```

The `posify()` (page 668) function returns the expression with all symbols replaced (which might lead to simplifications) and also a dict which maps the new symbols to the old that can be used with `subs()` (page 941). This is useful because otherwise the new expression with the new symbols having the `positive=True` assumption will not compare equal to the old:

```
>>> expr2
exp(_x)
>>> expr2 == exp(x)
False
>>> expr2.subs(rep)
exp(x)
>>> expr2.subs(rep) == exp(x)
True
```

3.1.3 Applying assumptions to string inputs

We have seen how to set assumptions when `Symbol` (page 976) or `symbols()` (page 978) explicitly. A natural question to ask is in what other situations can we assign assumptions to an object?

It is common for users to use strings as input to SymPy functions (although the general feeling among SymPy developers is that this should be discouraged) e.g.:

```
>>> from sympy import solve
>>> solve('x**2 - 1')
[-1, 1]
```

When creating symbols explicitly it would be possible to assign assumptions that would affect the behaviour of `solve()` (page 836):

```
>>> x = Symbol('x', positive=True)
>>> solve(x**2 - 1)
[1]
```

When using string input SymPy will create the expression and create all of the symbols implicitly so the question arises how can the assumptions be specified? The answer is that rather than depending on implicit string conversion it is better to use the `parse_expr()` (page 2122) function explicitly and then it is possible to provide assumptions for the symbols e.g.:

```
>>> from sympy import parse_expr
>>> parse_expr('x**2 - 1')
x**2 - 1
>>> eq = parse_expr('x**2 - 1', {'x':Symbol('x', positive=True)})
>>> solve(eq)
[1]
```

Note: The `solve()` (page 836) function is unusual as a high level API in that it actually checks the assumptions on any input symbols (the unknowns) and uses that to tailor its output. The assumptions system otherwise affects low-level evaluation but is not necessarily handled explicitly by high-level APIs.

3.1.4 Predicates

There are many different predicates that can be assumed for a symbol or can be queried for an expression. It is possible to combine multiple predicates when creating a symbol. Predicates are logically combined using *and* so if a symbol is declared with `positive=True` and also with `integer=True` then it is both positive *and* integer:

```
>>> x = Symbol('x', positive=True, integer=True)
>>> x.is_positive
True
>>> x.is_integer
True
```

The full set of known predicates for a symbol can be accessed using the `assumptions0` (page 929) attribute:

```
>>> x.assumptions0
{'algebraic': True,
 'commutative': True,
 'complex': True,
 'extended_negative': False,
 'extended_nonnegative': True,
 'extended_nonpositive': False,
 'extended_nonzero': True,
 'extended_positive': True,
 'extended_real': True,
 'finite': True,
 'hermitian': True,
 'imaginary': False,
```

(continues on next page)

(continued from previous page)

```
'infinite': False,
'integer': True,
'irrational': False,
'negative': False,
'noninteger': False,
'nonnegative': True,
'nonpositive': False,
'nonzero': True,
'positive': True,
'rational': True,
'real': True,
'transcendental': False,
'zero': False}
```

We can see that there are many more predicates listed than the two that were used to create `x`. This is because the assumptions system can infer some predicates from combinations of other predicates. For example if a symbol is declared with `positive=True` then it is possible to infer that it should have `negative=False` because a positive number can never be negative. Similarly if a symbol is created with `integer=True` then it is possible to infer that it should have `rational=True` because every integer is a rational number.

A full table of the possible predicates and their definitions is given below.

Table1: Assumptions predicates for the (old) assumptions

Predicate	Definition	Implications
commutative	A commutative expression. A commutative expression commutes with all other expressions under multiplication. If an expression <code>a</code> has <code>commutative=True</code> then <code>a * b == b * a</code> for any other expression <code>b</code> (even if <code>b</code> is not commutative). Unlike all other assumptions predicates <code>commutative</code> must always be <code>True</code> or <code>False</code> and can never be <code>None</code> . Also unlike all other predicates <code>commutative</code> defaults to <code>True</code> in e.g. <code>Symbol('x')</code> . [commutative]	
infinite	An infinite expression such as <code>oo</code> , <code>-oo</code> or <code>zoo</code> . [infinite]	<code>== !finite</code>
finite	A finite expression. Any expression that is not infinite is considered finite. [infinite]	<code>== !infinite</code>
hermitian	An element of the field of Hermitian operators. [antihermitian]	
antihermitian	An element of the field of antihermitian operators. [antihermitian]	

continues on next page

Table 1 – continued from previous page

Predicate	Definition	Implications
complex	A complex number, $z \in \mathbb{C}$. Any number of the form $x + iy$ where x and y are real and $i = \sqrt{-1}$. All complex numbers are finite. Includes all real numbers. [complex]	-> commutative -> finite
algebraic	An algebraic number, $z \in \overline{\mathbb{Q}}$. Any number that is a root of a non-zero polynomial $p(z) \in \mathbb{Q}[z]$ having rational coefficients. All algebraic numbers are complex. An algebraic number may or may not be real. Includes all rational numbers. [algebraic]	-> complex
transcendental	A complex number that is not algebraic, $z \in \mathbb{C} - \overline{\mathbb{Q}}$. All transcendental numbers are complex. A transcendental number may or may not be real but can never be rational. [transcendental]	== (complex & !algebraic)
extended_real	An element of the extended real number line, $x \in \mathbb{R}$ where $\mathbb{R} = \mathbb{R} \cup \{-\infty, +\infty\}$. An extended_real number is either real or $\pm\infty$. The relational operators $<$, $<=$, $>=$ and $>$ are defined only for expressions that are extended_real. [extended_real]	-> commutative
real	A real number, $x \in \mathbb{R}$. All real numbers are finite and complex (the set of reals is a subset of the set of complex numbers). Includes all rational numbers. A real number is either negative, zero or positive. [real]	-> complex == (extended_real & finite) == (negative zero positive) -> hermitian
imaginary	An imaginary number, $z \in \mathbb{I} - \{0\}$. A number of the form $z = yi$ where y is real, $y \neq 0$ and $i = \sqrt{-1}$. All imaginary numbers are complex and not real. Note in particular that zero is <i>not</i> considered imaginary in SymPy. [imaginary]	-> complex -> antihermitian -> !extended_real
rational	A rational number, $q \in \mathbb{Q}$. Any number of the form $\frac{a}{b}$ where a and b are integers and $b \neq 0$. All rational numbers are real and algebraic. Includes all integer numbers. [rational]	-> real -> algebraic
irrational	A real number that is not rational, $x \in \mathbb{R} - \mathbb{Q}$. [irrational]	== (real & !rational)

continues on next page

Table 1 – continued from previous page

Predicate	Definition	Implications
integer	An integer, $a \in \mathbb{Z}$. All integers are rational. Includes zero and all prime, composite, even and odd numbers. [integer]	-> rational
noninteger	An extended real number that is not an integer, $x \in \mathbb{R} - \mathbb{Z}$.	== (extended_real & !integer)
even	An even number, $e \in \{2k : k \in \mathbb{Z}\}$. All even numbers are integer numbers. Includes zero. [parity]	-> integer -> !odd
odd	An odd number, $o \in \{2k + 1 : k \in \mathbb{Z}\}$. All odd numbers are integer numbers. [parity]	-> integer -> !even
prime	A prime number, $p \in \mathbb{P}$. All prime numbers are positive and integer. [prime]	-> integer -> positive
composite	A composite number, $c \in \mathbb{N} - (\mathbb{P} \cup \{1\})$. A positive integer that is the product of two or more primes. A composite number is always a positive integer and is not prime. [composite]	-> (integer & positive & !prime) !composite -> (!positive !even prime)
zero	The number 0. An expression with zero=True represents the number 0 which is an integer. [zero]	-> even & finite == (extended_nonnegative & extended_nonpositive) == (nonnegative & nonpositive)
nonzero	A nonzero real number, $x \in \mathbb{R} - \{0\}$. A nonzero number is always real and can not be zero.	-> real == (extended_nonzero & finite)
extended_nonzero	A member of the extended reals that is not zero, $x \in \mathbb{R} - \{0\}$.	== (extended_real & !zero)
positive	A positive real number, $x \in \mathbb{R}, x > 0$. All positive numbers are finite so oo is not positive. [positive]	== (nonnegative & nonzero) == (extended_positive & finite)

continues on next page

Table 1 – continued from previous page

Predicate	Definition	Implications
nonnegative	A nonnegative real number, $x \in \mathbb{R}, x \geq 0$. All nonnegative numbers are finite so ∞ is not nonnegative. [positive]	<code>== (real & !negative)</code> <code>== (extended_nonnegative & finite)</code>
negative	A negative real number, $x \in \mathbb{R}, x < 0$. All negative numbers are finite so $-\infty$ is not negative. [negative]	<code>== (nonpositive & nonzero)</code> <code>== (extended_negative & finite)</code>
nonpositive	A nonpositive real number, $x \in \mathbb{R}, x \leq 0$. All nonpositive numbers are finite so $-\infty$ is not nonpositive. [negative]	<code>== (real & !positive)</code> <code>== (extended_nonpositive & finite)</code>
extended_positive	A positive extended real number, $x \in \mathbb{R}, x > 0$. An extended_positive number is either positive or ∞ . [extended_real]	<code>== (extended_nonnegative & extended_nonzero)</code>
extended_nonnegative	A nonnegative extended real number, $x \in \mathbb{R}, x \geq 0$. An extended_nonnegative number is either nonnegative or ∞ . [extended_real]	<code>== (extended_real & !extended_negative)</code>
extended_negative	A negative extended real number, $x \in \mathbb{R}, x < 0$. An extended_negative number is either negative or $-\infty$. [extended_real]	<code>== (extended_nonpositive & extended_nonzero)</code>
extended_nonpositive	A nonpositive extended real number, $x \in \mathbb{R}, x \leq 0$. An extended_nonpositive number is either nonpositive or $-\infty$. [extended_real]	<code>== (extended_real & !extended_positive)</code>

References for the above definitions

3.1.5 Implications

The assumptions system uses the inference rules to infer new predicates beyond those immediately specified when creating a symbol:

```
>>> x = Symbol('x', real=True, negative=False, zero=False)
>>> x.is_positive
True
```

Although `x` was not explicitly declared positive it can be inferred from the predicates that were given explicitly. Specifically one of the inference rules is `real == negative | zero | positive` so if `real` is `True` and both `negative` and `zero` are `False` then `positive` must be `True`.

In practice the assumption inference rules mean that it is not necessary to include redundant

predicates for example a positive real number can be simply be declared as positive:

```
>>> x1 = Symbol('x1', positive=True, real=True)
>>> x2 = Symbol('x2', positive=True)
>>> x1.is_real
True
>>> x2.is_real
True
>>> x1.assumptions0 == x2.assumptions0
True
```

Combining predicates that are inconsistent will give an error:

```
>>> x = Symbol('x', commutative=False, real=True)
Traceback (most recent call last):
...
InconsistentAssumptions: {
    algebraic: False,
    commutative: False,
    complex: False,
    composite: False,
    even: False,
    extended_negative: False,
    extended_nonnegative: False,
    extended_nonpositive: False,
    extended_nonzero: False,
    extended_positive: False,
    extended_real: False,
    imaginary: False,
    integer: False,
    irrational: False,
    negative: False,
    noninteger: False,
    nonnegative: False,
    nonpositive: False,
    nonzero: False,
    odd: False,
    positive: False,
    prime: False,
    rational: False,
    real: False,
    transcendental: False,
    zero: False}, real=True
```

3.1.6 Interpretation of the predicates

Although the predicates are defined in the table above it is worth taking some time to think about how to interpret them. Firstly many of the concepts referred to by the predicate names like “zero”, “prime”, “rational” etc have a basic meaning in mathematics but can also have more general meanings. For example when dealing with matrices a matrix of all zeros might be referred to as “zero”. The predicates in the assumptions system do not allow any generalizations such as this. The predicate zero is strictly reserved for the plain number 0. Instead matrices have an `is_zero_matrix()` (page 1344) property for this purpose (although that property is not strictly part of the assumptions system):

```
>>> from sympy import Matrix
>>> M = Matrix([[0, 0], [0, 0]])
>>> M.is_zero
False
>>> M.is_zero_matrix
True
```

Similarly there are generalisations of the integers such as the Gaussian integers which have a different notion of prime number. The prime predicate in the assumptions system does not include those and strictly refers only to the standard prime numbers $\mathbb{P} = \{2, 3, 5, 7, 11, \dots\}$. Likewise integer only means the standard concept of the integers $\mathbb{Z} = \{0, \pm 1, \pm 2, \dots\}$, rational only means the standard concept of the rational numbers $\mathbb{Q}$ and so on.

The predicates set up schemes of subsets such as the chain beginning with the complex numbers which are considered as a superset of the reals which are in turn a superset of the rationals and so on. The chain of subsets

$$\mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$$

corresponds to the chain of implications in the assumptions system

```
integer -> rational -> real -> complex
```

A “vanilla” symbol with no assumptions explicitly attached is not known to belong to any of these sets and is not even known to be finite:

```
>>> x = Symbol('x')
>>> x.assumptions0
{'commutative': True}
>>> print(x.is_commutative)
True
>>> print(x.is_rational)
None
>>> print(x.is_complex)
None
>>> print(x.is_real)
None
>>> print(x.is_integer)
None
>>> print(x.is_finite)
None
```

It is hard for SymPy to know what it can do with such a symbol that is not even known to be finite or complex so it is generally better to give some assumptions to the symbol explicitly.

Many parts of SymPy will implicitly treat such a symbol as complex and in some cases SymPy will permit manipulations that would not strictly be valid given that x is not known to be finite. In a formal sense though very little is known about a vanilla symbol which makes manipulations involving it difficult.

Defining *something* about a symbol can make a big difference. For example if we declare the symbol to be an integer then this implies a suite of other predicates that will help in further manipulations:

```
>>> n = Symbol('n', integer=True)
>>> n.assumptions0
{'algebraic': True,
 'commutative': True,
 'complex': True,
 'extended_real': True,
 'finite': True,
 'hermitian': True,
 'imaginary': False,
 'infinite': False,
 'integer': True,
 'irrational': False,
 'noninteger': False,
 'rational': True,
 'real': True,
 'transcendental': False}
```

These assumptions can lead to very significant simplifications e.g. `integer=True` gives:

```
>>> from sympy import sin, pi
>>> n1 = Symbol('n1')
>>> n2 = Symbol('n2', integer=True)
>>> sin(n1 * pi)
sin(pi*n1)
>>> sin(n2 * pi)
0
```

Replacing a whole expression with 0 is about as good as simplification can get!

It is normally advisable to set as many assumptions as possible on any symbols so that expressions can be simplified as much as possible. A common misunderstanding leads to defining a symbol with a False predicate e.g.:

```
>>> x = Symbol('x', negative=False)
>>> print(x.is_negative)
False
>>> print(x.is_nonnegative)
None
>>> print(x.is_real)
None
>>> print(x.is_complex)
None
>>> print(x.is_finite)
None
```

If the intention is to say that x is a real number that is not positive then that needs to be explicitly stated. In the context that the symbol is known to be real, the predicate `positive=False`

becomes much more meaningful:

```
>>> x = Symbol('x', real=True, negative=False)
>>> print(x.is_negative)
False
>>> print(x.is_nonnegative)
True
>>> print(x.is_real)
True
>>> print(x.is_complex)
True
>>> print(x.is_finite)
True
```

A symbol declared as `Symbol('x', real=True, negative=False)` is equivalent to a symbol declared as `Symbol('x', nonnegative=True)`. Simply declaring a symbol as `Symbol('x', positive=False)` does not allow the assumptions system to conclude much about it because a vanilla symbol is not known to be finite or even complex.

A related confusion arises with `Symbol('x', complex=True)` and `Symbol('x', real=False)`. Often when either of these is used neither is what is actually wanted. The first thing to understand is that all real numbers are complex so a symbol created with `real=True` will also have `complex=True` and a symbol created with `complex=True` will not have `real=False`. If the intention was to create a complex number that is not a real number then it should be `Symbol('x', complex=True, real=False)`. On the other hand declaring `real=False` alone is not sufficient to conclude that `complex=True` because knowing that it is not a real number does not tell us whether it is finite or whether or not it is some completely different kind of object from a complex number.

A vanilla symbol is defined by not knowing whether it is finite etc but there is no clear definition of what it *should* actually represent. It is tempting to think of it as an “arbitrary complex number or possibly one of the infinities” but there is no way to query an arbitrary (non-symbol) expression in order to determine if it meets those criteria. It is important to bear in mind that within the SymPy codebase and potentially in downstream libraries many other kinds of mathematical objects can be found that might also have `commutative=True` while being something very different from an ordinary number (in this context even SymPy’s standard infinities are considered “ordinary”).

The only predicate that is applied by default for a symbol is `commutative`. We can also declare a symbol to be *noncommutative* e.g.:

```
>>> x, y = symbols('x, y', commutative=False)
>>> z = Symbol('z') # defaults to commutative=True
>>> x*y + y*x
x*y + y*x
>>> x*z + z*x
2*z*x
```

Note here that since `x` and `y` are both noncommutative `x` and `y` do not commute so `x*y != y*x`. On the other hand since `z` is commutative `x` and `z` commute and `x*z == z*x` even though `x` is noncommutative.

The interpretation of what a vanilla symbol represents is unclear but the interpretation of an expression with `commutative=False` is entirely obscure. Such an expression is necessarily not a complex number or an extended real or any of the standard infinities (even `zoo` is commutative). We are left with very little that we can say about what such an expression *does*

represent.

3.1.7 Other `is_*` properties

There are many properties and attributes in SymPy that have names beginning with `is_` that look similar to the properties used in the (old) assumptions system but are not in fact part of the assumptions system. Some of these have a similar meaning and usage as those of the assumptions system such as the `is_zero_matrix()` (page 1344) property shown above. Another example is the `is_empty` property of sets:

```
>>> from sympy import FiniteSet, Intersection
>>> S1 = FiniteSet(1, 2)
>>> S1
{1, 2}
>>> print(S1.is_empty)
False
>>> S2 = Intersection(FiniteSet(1), FiniteSet(Symbol('x')))
>>> S2
Intersection({1}, {x})
>>> print(S2.is_empty)
None
```

The `is_empty` property gives a fuzzy-bool indicating whether or not a [Set](#) (page 1185) is the empty set. In the example of `S2` it is not possible to know whether or not the set is empty without knowing whether or not `x` is equal to 1 so `S2.is_empty` gives `None`. The `is_empty` property for sets plays a similar role to the `is_zero` property for numbers in the assumptions system: `is_empty` is normally only `True` for the [EmptySet](#) (page 1202) object but it is still useful to be able to distinguish between the cases where `is_empty=False` and `is_empty=None`.

Although `is_zero_matrix` and `is_empty` are used for similar purposes to the assumptions properties such as `is_zero` they are not part of the (old) assumptions system. There are no associated inference rules connecting e.g. `Set.is_empty` and `Set.is_finite_set` because the inference rules are part of the (old) assumptions system which only deals with the predicates listed in the table above. It is not possible to declare a [MatrixSymbol](#) (page 1372) with e.g. `zero_matrix=False` and there is no `SetSymbol` class but if there was it would not have a system for understanding predicates like `empty=False`.

The properties `is_zero_matrix()` (page 1344) and `is_empty` are similar to those of the assumptions system because they concern *semantic* aspects of an expression. There are a large number of other properties that focus on *structural* aspects such as `is_Number`, `is_number()` (page 967), `is_comparable()` (page 935). Since these properties refer to structural aspects of an expression they will always give `True` or `False` rather than a fuzzy bool that also has the possibility of being `None`. Capitalised properties such as `is_Number` are usually shorthand for `isinstance` checks e.g.:

```
>>> from sympy import Number, Rational
>>> x = Rational(1, 2)
>>> isinstance(x, Number)
True
>>> x.is_Number
True
>>> y = Symbol('y', rational=True)
>>> isinstance(y, Number)
```

(continues on next page)

(continued from previous page)

```
False
>>> y.is_Number
False
```

The *Number* (page 981) class is the superclass for *Integer* (page 987), *Rational* (page 985) and *Float* (page 982) so any instance of *Number* (page 981) represents a concrete number with a known value. A symbol such as *y* that is declared with `rational=True` might represent the same value as *x* but it is not a concrete number with a known value so this is a structural rather than a semantic distinction. Properties like `is_Number` are sometimes used in SymPy in place of e.g. `isinstance(obj, Number)` because they do not have problems with circular imports and checking `x.is_Number` can be faster than a call to `isinstance`.

The *is_number* (page 967) (lower-case) property is very different from `is_Number`. The *is_number* (page 967) property is True for any expression that can be numerically evaluated to a floating point complex number with *evalf()* (page 1065):

```
>>> from sympy import I
>>> expr1 = I + sqrt(2)
>>> expr1
sqrt(2) + I
>>> expr1.is_number
True
>>> expr1.evalf()
1.4142135623731 + 1.0*I
>>> x = Symbol('x')
>>> expr2 = 1 + x
>>> expr2
x + 1
>>> expr2.is_number
False
>>> expr2.evalf()
x + 1.0
```

The primary reason for checking `expr.is_number` is to predict whether a call to *evalf()* (page 1065) will fully evaluate. The *is_comparable()* (page 935) property is similar to *is_number()* (page 967) except that if `is_comparable` gives True then the expression is guaranteed to numerically evaluate to a *real Float* (page 982). When `a.is_comparable` and `b.is_comparable` the inequality `a < b` should be resolvable as something like `a.evalf() < b.evalf()`.

The full set of `is_*` properties, attributes and methods in SymPy is large. It is important to be clear though that only those that are listed in the table of predicates above are actually part of the assumptions system. It is only those properties that are involved in the *mechanism* that implements the assumptions system which is explained below.

3.1.8 Implementing assumptions handlers

We will now work through an example of how to implement a SymPy symbolic function so that we can see how the old assumptions are used internally. SymPy already has an `exp` function which is defined for all complex numbers but we will define an `expreal` function which is restricted to real arguments.

```
>>> from sympy import Function
>>> from sympy.core.logic import fuzzy_and, fuzzy_or
>>>
>>> class expreal(Function):
...     """exponential function  $E^{*x}$  restricted to the extended reals"""
...
...     is_extended_nonnegative = True
...
...     @classmethod
...     def eval(cls, x):
...         # Validate the argument
...         if x.is_extended_real is False:
...             raise ValueError("non-real argument to expreal")
...         # Evaluate for special values
...         if x.is_zero:
...             return S.One
...         elif x.is_infinite:
...             if x.is_extended_negative:
...                 return S.Zero
...             elif x.is_extended_positive:
...                 return S.Infinity
...
...     @property
...     def x(self):
...         return self.args[0]
...
...     def _eval_is_finite(self):
...         return fuzzy_or([self.x.is_real, self.x.is_extended_nonpositive])
...
...     def _eval_is_algebraic(self):
...         if fuzzy_and([self.x.is_rational, self.x.is_nonzero]):
...             return False
...
...     def _eval_is_integer(self):
...         if self.x.is_zero:
...             return True
...
...     def _eval_is_zero(self):
...         return fuzzy_and([self.x.is_infinite, self.x.is_extended_
... ↪negative])
```

The `Function.eval` method is used to pick up on special values of the function so that we can return a different object if it would be a simplification. When `expreal(x)` is called the `expreal.__new__` class method (defined in the superclass `Function`) will call `expreal.eval(x)`. If `expreal.eval` returns something other than `None` then that will be returned instead of an unevaluated `expreal(x)`:

```
>>> from sympy import oo
>>> expreal(1)
expreal(1)
>>> expreal(0)
1
>>> expreal(-oo)
0
>>> expreal(oo)
oo
```

Note that the `expreal.eval` method does not compare the argument using `==`. The special values are verified using the assumptions system to query the properties of the argument. That means that the `expreal` method can also evaluate for different forms of expression that have matching properties e.g.

```
>>> x = Symbol('x', extended_negative=True, infinite=True)
>>> x
x
>>> expreal(x)
0
```

Of course the assumptions system can only resolve a limited number of special values so most `eval` methods will also check against some special values with `==` but it is preferable to check e.g. `x.is_zero` rather than `x==0`.

Note also that the `expreal.eval` method validates that the argument is real. We want to allow $\pm\infty$ as arguments to `expreal` so we check for `extended_real` rather than `real`. If the argument is not extended real then we raise an error:

```
>>> expreal(I)
Traceback (most recent call last):
...
ValueError: non-real argument to expreal
```

Importantly we check `x.is_extended_real` is `False` rather than not `x.is_extended_real` which means that we only reject the argument if it is *definitely* not extended real: if `x.is_extended_real` gives `None` then the argument will not be rejected. The first reason for allowing `x.is_extended_real=None` is so that a vanilla symbol can be used with `expreal`. The second reason is that an assumptions query can always give `None` even in cases where an argument is definitely real e.g.:

```
>>> x = Symbol('x')
>>> print(x.is_extended_real)
None
>>> expreal(x)
expreal(x)
>>> expr = (1 + I)/sqrt(2) + (1 - I)/sqrt(2)
>>> print(expr.is_extended_real)
None
>>> expr.expand()
sqrt(2)
>>> expr.expand().is_extended_real
True
```

(continues on next page)

(continued from previous page)

```
>>> expreal(expr)
expreal(sqrt(2)*(1 - I)/2 + sqrt(2)*(1 + I)/2)
```

Validating the argument in `expreal.eval` does mean that it will not be validated when `evaluate=False` is passed but there is not really a better place to perform the validation:

```
>>> expreal(I, evaluate=False)
expreal(I)
```

The extended `_nonnegative` class attribute and the `_eval_is_*` methods on the `expreal` class implement queries in the assumptions system for instances of `expreal`:

```
>>> expreal(2)
expreal(2)
>>> expreal(2).is_finite
True
>>> expreal(2).is_integer
False
>>> expreal(2).is_rational
False
>>> expreal(2).is_algebraic
False
>>> z = expreal(-oo, evaluate=False)
>>> z
expreal(-oo)
>>> z.is_integer
True
>>> x = Symbol('x', real=True)
>>> expreal(x)
expreal(x)
>>> expreal(x).is_nonnegative
True
```

The assumptions system resolves queries like `expreal(2).is_finite` using the corresponding handler `expreal._eval_is_finite` and *also* the implication rules. For example it is known that `expreal(2).is_rational` is `False` because `expreal(2)._eval_is_algebraic` returns `False` and there is an implication rule `rational -> algebraic`. This means that an `is_rational` query can be resolved in this case by the `_eval_is_algebraic` handler. It is actually better not to implement assumptions handlers for every possible predicate but rather to try and identify a minimal set of handlers that can resolve as many queries as possible with as few checks as possible.

Another point to note is that the `_eval_is_*` methods only make assumptions queries on the argument `x` and do not make any assumptions queries on `self`. Recursive assumptions queries on the same object will interfere with the assumptions implications resolver potentially leading to non-deterministic behaviour so they should not be used (there are examples of this in the SymPy codebase but they should be removed).

Many of the `expreal` methods implicitly return `None`. This is a common pattern in the assumptions system. The `eval` method and the `_eval_is_*` methods can all return `None` and often will. A Python function that ends without reaching a return statement will implicitly return `None`. We take advantage of this by leaving out many of the `else` clauses from the `if` statements and allowing `None` to be returned implicitly. When following the control flow of these methods it is important to bear in mind firstly that any queried property can give `True`,

False or None and also that any function will implicitly return None if all of the conditionals fail.

3.1.9 Mechanism of the assumptions system

Note: This section describes internal details that could change in a future SymPy version.

This section will explain the inner workings of the assumptions system. It is important to understand that these inner workings are implementation details and could change from one SymPy version to another. This explanation is written as of SymPy 1.7. Although the (old) assumptions system has many limitations (discussed in the next section) it is a mature system that is used extensively in SymPy and has been well optimised for its current usage. The assumptions system is used implicitly in most SymPy operations to control evaluation of elementary expressions.

There are several stages in the implementation of the assumptions system within a SymPy process that lead up to the evaluation of a single query in the assumptions system. Briefly these are:

1. At import time the assumptions rules defined in `sympy/core/assumptions.py` are processed into a canonical form ready for efficiently applying the implication rules. This happens once when SymPy is imported before even the *Basic* (page 927) class is defined.
2. The `ManagedProperties` metaclass is defined which is the metaclass for all *Basic* (page 927) subclasses. This class will post-process every *Basic* (page 927) subclass to add the relevant properties needed for assumptions queries. This also adds the `default_assumptions` attribute to the class. This happens each time a *Basic* (page 927) subclass is defined.
3. Every *Basic* (page 927) instance initially uses the `default_assumptions` class attribute. When an assumptions query is made on a *Basic* (page 927) instance in the first instance the query will be answered from the `default_assumptions` for the class.
4. If there is no cached value for the assumptions query in the `default_assumptions` for the class then the default assumptions will be copied to make an assumptions cache for the instance. Then the `_ask()` function is called to resolve the query which will firstly call the relevant instance handler `_eval_is` method. If the handler returns non-None then the result will be cached and returned.
5. If the handler does not exist or gives None then the implications resolver is tried. This will enumerate (in a randomised order) all possible combinations of predicates that could potentially be used to resolve the query under the implication rules. In each case the handler `_eval_is` method will be called to see if it gives non-None. If any combination of handlers and implication rules leads to a definitive result for the query then that result is cached in the instance cache and returned.
6. Finally if the implications resolver failed to resolve the query then the query is considered unresolvable. The value of None for the query is cached in the instance cache and returned.

The assumptions rules defined in `sympy/core/assumptions.py` are given in forms like `real == negative | zero | positive`. When this module is imported these are converted into a `FactRules` instance called `_assume_rules`. This preprocesses the implication rules into the form of “A” and “B” rules that can be used for the implications resolver. This is explained in

the code in `sympy/core/facts.py`. We can access this internal object directly like (full output omitted):

```
>>> from sympy.core.assumptions import _assume_rules
>>> _assume_rules.defined_facts
{'algebraic',
 'antihermitian',
 'commutative',
 'complex',
 'composite',
 'even',
 ...
>>> _assume_rules.full_implications
defaultdict(set,
              {('extended_positive', False): {('composite', False),
 ('positive', False),
 ('prime', False)},
 ('finite', False): {('algebraic', False),
 ('complex', False),
 ('composite', False),
 ...
```

The `ManagedProperties` metaclass will inspect the attributes of each `Basic` class to see if any assumptions related attributes are defined. An example of these is the `is_extended_nonnegative = True` attribute defined in the `Expr` class. The implications of any such attributes will be used to precompute any statically knowable assumptions. For example `is_extended_nonnegative=True` implies `real=True` etc. A `StdFactKB` instance is created for the class which stores those assumptions whose values are known at this stage. The `StdFactKB` instance is assigned as the class attribute `default_assumptions`. We can see this with

```
>>> from sympy import Expr
...
>>> class A(Expr):
...     is_positive = True
...
...     def _eval_is_rational(self):
...         # Let's print something to see when this method is called...
...         print('!!! calling _eval_is_rational')
...         return True
...
>>> A.is_positive
True
>>> A.is_real # inferred from is_positive
True
```

Although only `is_positive` was defined in the class `A` it also has attributes such as `is_real` which are inferred from `is_positive`. The set of all such assumptions for class `A` can be seen in `default_assumptions` which looks like a dict but is in fact a `StdFactKB` instance:

```
>>> type(A.default_assumptions)
<class 'sympy.core.assumptions.StdFactKB'>
>>> A.default_assumptions
{'commutative': True,
```

(continues on next page)

(continued from previous page)

```
'complex': True,
'extended_negative': False,
'extended_nonnegative': True,
'extended_nonpositive': False,
'extended_nonzero': True,
'extended_positive': True,
'extended_real': True,
'finite': True,
'hermitian': True,
'imaginary': False,
'infinite': False,
'negative': False,
'nonnegative': True,
'nonpositive': False,
'nonzero': True,
'positive': True,
'real': True,
'zero': False}
```

When an instance of any [Basic](#) (page 927) subclass is created `Basic.__new__` will assign its `_assumptions` attribute which will initially be a reference to `cls.default_assumptions` shared amongst all instances of the same class. The instance will use this to resolve any assumptions queries until that fails to give a definitive result at which point a copy of `cls.default_assumptions` will be created and assigned to the instance's `_assumptions` attribute. The copy will be used as a cache to store any results computed for the instance by its `_eval_is` handlers.

When the `_assumptions` attribute fails to give the relevant result it is time to call the `_eval_is` handlers. At this point the `_ask()` function is called. The `_ask()` function will initially try to resolve a query such as `is_rational` by calling the corresponding method i.e. `_eval_is_rational`. If that gives non-None then the result is stored in `_assumptions` and any implications of that result are computed and stored as well. At that point the query is resolved and the value returned.

```
>>> a = A()
>>> a._assumptions is A.default_assumptions
True
>>> a.is_rational
!!! calling _eval_is_rational
True
>>> a._assumptions is A.default_assumptions
False
>>> a._assumptions # rational now shows as True
{'algebraic': True,
 'commutative': True,
 'complex': True,
 'extended_negative': False,
 'extended_nonnegative': True,
 'extended_nonpositive': False,
 'extended_nonzero': True,
 'extended_positive': True,
 'extended_real': True,
```

(continues on next page)

(continued from previous page)

```

'finite': True,
'hermitian': True,
'imaginary': False,
'infinite': False,
'irrational': False,
'negative': False,
'nonnegative': True,
'nonpositive': False,
'nonzero': True,
'positive': True,
'rational': True,
'real': True,
'transcendental': False,
'zero': False}

```

If e.g. `_eval_is_rational` does not exist or gives `None` then `_ask()` will try all possibilities to use the implication rules and any other handler methods such as `_eval_is_integer`, `_eval_is_algebraic` etc that might possibly be able to give an answer to the original query. If any method leads to a definite result being known for the original query then that is returned. Otherwise once all possibilities for using a handler and the implication rules to resolve the query are exhausted `None` will be cached and returned.

```

>>> b = A()
>>> b.is_algebraic      # called _eval_is_rational indirectly
!!! calling _eval_is_rational
True
>>> c = A()
>>> print(c.is_prime)   # called _eval_is_rational indirectly
!!! calling _eval_is_rational
None
>>> c._assumptions      # prime now shows as None
{'algebraic': True,
 'commutative': True,
 'complex': True,
 'extended_negative': False,
 'extended_nonnegative': True,
 'extended_nonpositive': False,
 'extended_nonzero': True,
 'extended_positive': True,
 'extended_real': True,
 'finite': True,
 'hermitian': True,
 'imaginary': False,
 'infinite': False,
 'irrational': False,
 'negative': False,
 'nonnegative': True,
 'nonpositive': False,
 'nonzero': True,
 'positive': True,
 'prime': None,
 'rational': True,

```

(continues on next page)

(continued from previous page)

```
'real': True,  
'transcendental': False,  
'zero': False}
```

Note: In the `_ask()` function the handlers are called in a randomised order which can mean that execution at this point is non-deterministic. Provided all of the different handler methods are consistent (i.e. there are no bugs) then the end result will still be deterministic. However a bug where two handlers are inconsistent can manifest in non-deterministic behaviour because this randomisation might lead to the handlers being called in different orders when the same program is run multiple times.

3.1.10 Limitations

Combining predicates with *or*

In the old assumptions we can easily combine predicates with *and* when creating a `Symbol` e.g.:

```
>>> x = Symbol('x', integer=True, positive=True)  
>>> x.is_positive  
True  
>>> x.is_integer  
True
```

We can also easily query whether two conditions are jointly satisfied with

```
>>> fuzzy_and([x.is_positive, x.is_integer])  
True  
>>> x.is_positive and x.is_integer  
True
```

However there is no way in the old assumptions to create a `Symbol` (page 976) with assumptions predicates combined with *or*. For example if we wanted to say that “x is positive or x is an integer” then it is not possible to create a `Symbol` (page 976) with those assumptions.

It is also not possible to ask an assumptions query based on *or* e.g. “is expr an expression that is positive or an integer”. We can use e.g.

```
>>> fuzzy_or([x.is_positive, x.is_integer])  
True
```

However if all that is known about x is that it is possibly positive or otherwise a negative integer then both queries `x.is_positive` and `x.is_integer` will resolve to `None`. That means that the query becomes

```
>>> fuzzy_or([None, None])
```

which then also gives `None`.

Relations between different symbols

A fundamental limitation of the old assumptions system is that all explicit assumptions are properties of an individual symbol. There is no way in this system to make an assumption about the *relationship* between two symbols. One of the most common requests is the ability to assume something like $x < y$ but there is no way to even specify that in the old assumptions.

The new assumptions have the theoretical capability that relational assumptions can be specified. However the algorithms to make use of that information are not yet implemented and the exact API for specifying relational assumptions has not been decided upon.

3.2 Symbolic and fuzzy booleans

This page describes what a symbolic *Boolean* (page 1163) in SymPy is and also how that relates to three-valued fuzzy-booleans that are used in many parts of SymPy. It also discusses some common problems that arise when writing code that uses three-valued logic and how to handle them correctly.

3.2.1 Symbolic Boolean vs three valued bool

Assumptions queries like `x.ispositive` give fuzzy-bool `True`, `False` or `None` results¹. These are low-level Python objects rather than SymPy's symbolic *Boolean* (page 1163) expressions.

```
>>> from sympy import Symbol, symbols
>>> xpos = Symbol('xpos', positive=True)
>>> xneg = Symbol('xneg', negative=True)
>>> x = Symbol('x')
>>> print(xpos.is_positive)
True
>>> print(xneg.is_positive)
False
>>> print(x.is_positive)
None
```

A `None` result as a fuzzy-bool should be interpreted as meaning “maybe” or “unknown”.

An example of a symbolic *Boolean* (page 1163) class in SymPy can be found when using inequalities. When an inequality is not known to be true or false a *Boolean* (page 1163) can represent indeterminate results symbolically:

```
>>> xpos > 0
True
>>> xneg > 0
False
>>> x > 0
x > 0
>>> type(x > 0)
<class 'sympy.core.relational.StrictGreaterThan'>
```

¹ Note that what is referred to in SymPy as a “fuzzy bool” is really about using three-valued logic. In normal usage “fuzzy logic” refers to a system where logical values are continuous in between zero and one which is something different from three-valued logic.

The last example shows what happens when an inequality is indeterminate: we get an instance of *StrictGreaterThan* (page 1032) which represents the inequality as a symbolic expression. Internally when attempting to evaluate an inequality like $a > b$ SymPy will compute $(a - b).is\_extended\_positive$. If the result is `True` or `False` then SymPy's symbolic `S.true` or `S.false` will be returned. If the result is `None` then an unevaluated *StrictGreaterThan* (page 1032) is returned as shown for $x > 0$ above.

It is not obvious that queries like $xpos > 0$ return `S.true` rather than `True` because both objects display in the same way but we can check this using the Python `is` operator:

```
>>> from sympy import S
>>> xpos.is_positive is True
True
>>> xpos.is_positive is S.true
False
>>> (xpos > 0) is True
False
>>> (xpos > 0) is S.true
True
```

There is no general symbolic analogue of `None` in SymPy. In the cases where a low-level assumptions query gives `None` the symbolic query will result in an unevaluated symbolic *Boolean* (page 1163) (e.g. $x > 0$). We can use a symbolic *Boolean* (page 1163) as part of a symbolic expression such as a *Piecewise* (page 415):

```
>>> from sympy import Piecewise
>>> p = Piecewise((1, x > 0), (2, True))
>>> p
Piecewise((1, x > 0), (2, True))
>>> p.subs(x, 3)
1
```

Here `p` represents an expression that will be equal to 1 if $x > 0$ or otherwise it will be equal to 2. The unevaluated *Boolean* (page 1163) inequality $x > 0$ represents the condition for deciding the value of the expression symbolically. When we substitute a value for x the inequality will resolve to `S.true` and then the *Piecewise* (page 415) can evaluate to 1 or 2.

The same will not work when using a fuzzy-bool instead of a symbolic *Boolean* (page 1163):

```
>>> p2 = Piecewise((1, x.is_positive), (2, True))
Traceback (most recent call last):
...
TypeError: Second argument must be a Boolean, not `NoneType`
```

The *Piecewise* (page 415) can not use `None` as the condition because unlike the inequality $x > 0$ it gives no information. With the inequality it is possible to decide in future if the condition might `True` or `False` once a value for x is known. A value of `None` can not be used in that way so it is rejected.

Note: We can use `True` in the *Piecewise* (page 415) because `True` sympifies to `S.true`. Sympifying `None` just gives `None` again which is not a valid symbolic SymPy object.

There are many other symbolic *Boolean* (page 1163) types in SymPy. The same considerations about the differences between fuzzy bool and symbolic *Boolean* (page 1163) apply to all other

SymPy [Boolean](#) (page 1163) types. To give a different example there is [Contains](#) (page 1216) which represents the statement that an object is contained in a set:

```
>>> from sympy import Reals, Contains
>>> x = Symbol('x', real=True)
>>> y = Symbol('y')
>>> Contains(x, Reals)
True
>>> Contains(y, Reals)
Contains(y, Reals)
>>> Contains(y, Reals).subs(y, 1)
True
```

The Python operator corresponding to [Contains](#) (page 1216) is `in`. A quirk of `in` is that it can only evaluate to a bool (True or False) so if the result is indeterminate then an exception will be raised:

```
>>> from sympy import I
>>> 2 in Reals
True
>>> I in Reals
False
>>> x in Reals
True
>>> y in Reals
Traceback (most recent call last):
...
TypeError: did not evaluate to a bool: (-oo < y) & (y < oo)
```

The exception can be avoided by using `Contains(x, Reals)` or `Reals.contains(x)` rather than `x in Reals`.

3.2.2 Three-valued logic with fuzzy bools

Whether we use the fuzzy-bool or symbolic [Boolean](#) (page 1163) we always need to be aware of the possibility that a query might be indeterminate. How to write code that handles this is different in the two cases though. We will look at fuzzy-bools first.

Consider the following function:

```
>>> def both_positive(a, b):
...     """ask whether a and b are both positive"""
...     if a.is_positive and b.is_positive:
...         return True
...     else:
...         return False
```

The `both_positive` function is supposed to tell us whether or not `a` and `b` are both positive. However the `both_positive` function will fail if either of the `is_positive` queries gives `None`:

```
>>> print(both_positive(S(1), S(1)))
True
>>> print(both_positive(S(1), S(-1)))
```

(continues on next page)

(continued from previous page)

```
False
>>> print(both_positive(S(-1), S(-1)))
False
>>> x = Symbol('x') # may or may not be positive
>>> print(both_positive(S(1), x))
False
```

Note: We need to sympify the arguments to this function using `S` because the assumptions are only defined on SymPy objects and not regular Python `int` objects.

Here `False` is incorrect because it is *possible* that `x` is positive in which case both arguments would be positive. We get `False` here because `x.is_positive` gives `None` and Python will treat `None` as “falsey”.

In order to handle all possible cases correctly we need to separate the logic for identifying the `True` and `False` cases. An improved function might be:

```
>>> def both_positive_better(a, b):
...     """ask whether a and b are both positive"""
...     if a.is_positive is False or b.is_positive is False:
...         return False
...     elif a.is_positive is True and b.is_positive is True:
...         return True
...     else:
...         return None
```

This function now can handle all cases of `True`, `False` or `None` for both `a` and `b` and will always return a fuzzy bool representing whether the statement “`a` and `b` are both positive” is true, false or unknown:

```
>>> print(both_positive_better(S(1), S(1)))
True
>>> print(both_positive_better(S(1), S(-1)))
False
>>> x = Symbol('x')
>>> y = Symbol('y', positive=True)
>>> print(both_positive_better(S(1), x))
None
>>> print(both_positive_better(S(-1), x))
False
>>> print(both_positive_better(S(1), y))
True
```

Another case that we need to be careful of when using fuzzy-booleans is negation with Python’s `not` operator e.g.:

```
>>> x = Symbol('x')
>>> print(x.is_positive)
None
>>> not x.is_positive
True
```

The correct negation of a fuzzy bool `None` is `None` again. If we do not know whether the statement “`x` is positive” is `True` or `False` then we also do not know whether its negation “`x` is not positive” is `True` or `False`. The reason we get `True` instead is again because `None` is considered “falsey”. When `None` is used with a logical operator such as `not` it will first be converted to a bool and then negated:

```
>>> bool(None)
False
>>> not bool(None)
True
>>> not None
True
```

The fact that `None` is treated as falsey can be useful if used correctly. For example we may want to do something only if `x` is known to positive in which case we can do

```
>>> x = Symbol('x', positive=True)
>>> if x.is_positive:
...     print("x is definitely positive")
... else:
...     print("x may or may not be positive")
x is definitely positive
```

Provided we understand that an alternate condition branch refers to two cases (`False` and `None`) then this can be a useful way of writing conditionals. When we really do need to distinguish all cases then we need to use things like `x.is_positive is False`. What we need to be careful of though is using Python’s binary logic operators like `not` or `and` with fuzzy bools as they will not handle the indeterminate case correctly.

In fact SymPy has internal functions that are designed to handle fuzzy-bools correctly:

```
>>> from sympy.core.logic import fuzzy_not, fuzzy_and
>>> print(fuzzy_not(True))
False
>>> print(fuzzy_not(False))
True
>>> print(fuzzy_not(None))
None
>>> print(fuzzy_and([True, True]))
True
>>> print(fuzzy_and([True, None]))
None
>>> print(fuzzy_and([False, None]))
False
```

Using the `fuzzy_and` function we can write the `both_positive` function more simply:

```
>>> def both_positive_best(a, b):
...     """ask whether a and b are both positive"""
...     return fuzzy_and([a.is_positive, b.is_positive])
```

Making use of `fuzzy_and`, `fuzzy_or` and `fuzzy_not` leads to simpler code and can also reduce the chance of introducing a logic error because the code can look more like it would in the case of ordinary binary logic.

3.2.3 Three-valued logic with symbolic Booleans

When working with symbolic *Boolean* (page 1163) rather than fuzzy-bool the issue of None silently being treated as falsey does not arise so it is easier not to end up with a logic error. However instead the indeterminate case will often lead to an exception being raised if not handled carefully.

We will try to implement the `both_positive` function this time using symbolic *Boolean* (page 1163):

```
>>> def both_positive(a, b):
...     """ask whether a and b are both positive"""
...     if a > 0 and b > 0:
...         return S.true
...     else:
...         return S.false
```

The first difference is that we return the symbolic *Boolean* (page 1163) objects `S.true` and `S.false` rather than `True` and `False`. The second difference is that we test e.g. `a > 0` rather than `a.is_positive`. Trying this out we get

```
>>> both_positive(1, 2)
True
>>> both_positive(-1, 1)
False
>>> x = Symbol('x') # may or may not be positive
>>> both_positive(x, 1)
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
```

What happens now is that testing `x > 0` gives an exception when `x` is not known to be positive or not positive. More precisely `x > 0` does not give an exception but `if x > 0` does and that is because the `if` statement implicitly calls `bool(x > 0)` which raises.

```
>>> x > 0
x > 0
>>> bool(x > 0)
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
>>> if x > 0:
...     print("x is positive")
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
```

The Python expression `x > 0` creates a SymPy *Boolean* (page 1163). Since in this case the *Boolean* (page 1163) can not evaluate to `True` or `False` we get an unevaluated *StrictGreaterThan* (page 1032). Attempting to force that into a `bool` with `bool(x > 0)` raises an exception. That is because a regular Python `bool` must be either `True` or `False` and neither of those are known to be correct in this case.

The same kind of issue arises when using `and`, `or` or `not` with symbolic *Boolean* (page 1163). The solution is to use SymPy's symbolic *And* (page 1166), *Or* (page 1167) and *Not* (page 1167) or equivalently Python's bitwise logical operators `&`, `|` and `~`:

```

>>> from sympy import And, Or, Not
>>> x > 0
x > 0
>>> x > 0 and x < 1
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
>>> And(x > 0, x < 1)
(x > 0) & (x < 1)
>>> (x > 0) & (x < 1)
(x > 0) & (x < 1)
>>> Or(x < 0, x > 1)
(x > 1) | (x < 0)
>>> Not(x < 0)
x >= 0
>>> ~(x < 0)
x >= 0

```

As before we can make a better version of `both_positive` if we avoid directly using a SymPy `Boolean` (page 1163) in an `if`, `and`, `or`, or `not`. Instead we can test whether or not the `Boolean` (page 1163) has evaluated to `S.true` or `S.false`:

```

>>> def both_positive_better(a, b):
...     """ask whether a and b are both positive"""
...     if (a > 0) is S.false or (b > 0) is S.false:
...         return S.false
...     elif (a > 0) is S.true and (b > 0) is S.true:
...         return S.true
...     else:
...         return And(a > 0, b > 0)

```

Now with this version we don't get any exceptions and if the result is indeterminate we will get a symbolic `Boolean` (page 1163) representing the conditions under which the statement "a and b are both positive" would be true:

```

>>> both_positive_better(S(1), S(2))
True
>>> both_positive_better(S(1), S(-1))
False
>>> x, y = symbols("x, y")
>>> both_positive_better(x, y + 1)
(x > 0) & (y + 1 > 0)
>>> both_positive_better(x, S(3))
x > 0

```

The last case shows that actually using the `And` (page 1166) with a condition that is known to be true simplifies the `And` (page 1166). In fact we have

```

>>> And(x > 0, 3 > 0)
x > 0
>>> And(4 > 0, 3 > 0)
True
>>> And(-1 > 0, 3 > 0)
False

```

What this means is that we can improve `both_positive_better`. The different cases are not needed at all. Instead we can simply return the `And` (page 1166) and let it simplify if possible:

```
>>> def both_positive_best(a, b):  
...     """ask whether a and b are both positive"""  
...     return And(a > 0, b > 0)
```

Now this will work with any symbolic real objects and produce a symbolic result. We can also substitute into the result to see how it would work for particular values:

```
>>> both_positive_best(2, 1)  
True  
>>> both_positive_best(-1, 2)  
False  
>>> both_positive_best(x, 3)  
x > 0  
>>> condition = both_positive_best(x/y, x + y)  
>>> condition  
(x + y > 0) & (x/y > 0)  
>>> condition.subs(x, 1)  
(1/y > 0) & (y + 1 > 0)  
>>> condition.subs(x, 1).subs(y, 2)  
True
```

The idea when working with symbolic *Boolean* (page 1163) objects is as much as possible to avoid trying to branch on them with `if/else` and other logical operators like `and` etc. Instead think of computing a condition and passing it around as a variable. The elementary symbolic operations like *And* (page 1166), *Or* (page 1167) and *Not* (page 1167) can then take care of the logic for you.

3.3 Writing Custom Functions

This guide will describe how to create custom function classes in SymPy. Custom user defined functions use the same mechanisms as the *functions* (page 381) that are included with SymPy such as the common *elementary functions* (page 382) like *exp()* (page 411) or *sin()* (page 389), *special functions* (page 450) like *gamma()* (page 459) or *Si()* (page 491), and *combinatorial functions* (page 427) and *number theory functions* (page 1476) like *factorial()* (page 434) or *primepi()* (page 1479). Consequently, this guide serves both as a guide to end users who want to define their own custom functions and to SymPy developers wishing to extend the functions included with SymPy.

This guide describes how to define complex valued functions, that is functions that map a subset of $\mathbb{C}^n$ to $\mathbb{C}$. Functions that accept or return other kinds of objects than complex numbers should subclass another class, such as *Boolean* (page 1163), *MatrixExpr* (page 1370), *Expr* (page 947), or *Basic* (page 927). Some of what is written here will apply to general *Basic* (page 927) or *Expr* (page 947) subclasses, but much of it only applies to *Function* (page 1050) subclasses.

3.3.1 Easy Cases: Fully Symbolic or Fully Evaluated

Before digging into the more advanced functionality for custom functions, we should mention two common cases, the case where the function is fully symbolic, and the case where the function is fully evaluated. Both of these cases have much simpler alternatives than the full mechanisms described in this guide.

The Fully Symbolic Case

If your function `f` has no mathematical properties you want to define on it, and should never evaluate on any arguments, you can create an undefined function using `Function('f')`

```
>>> from sympy import symbols, Function
>>> x = symbols('x')
>>> f = Function('f')
```

```
>>> f(x)
f(x)
>>> f(0)
f(0)
```

This is useful, for instance, when solving *ODEs* (page 755).

This is also useful if you only wish to create a symbol that depends on another symbol for the purposes of differentiation. By default, SymPy assumes all symbols are independent of one another:

```
>>> from sympy.abc import x, y
>>> y.diff(x)
0
```

To make a symbol that depends on another symbol, you can use a function that explicitly depends on that symbol.

```
>>> y = Function('y')
>>> y(x).diff(x)
Derivative(y(x), x)
```

If you want your function to have additional behavior, for example, to have a custom derivative, or to evaluate on certain arguments, you should create a custom `Function` subclass as *described below* (page 105). However, undefined functions do support one additional feature, which is that assumptions can be defined on them, using the same syntax as used by symbols. This defines the assumptions of the output of the function, not the input (that is, it defines the function's range, not its domain).

```
>>> g = Function('g', real=True)
```

```
>>> g(x)
g(x)
>>> g(x).is_real
True
```

To make a function's assumptions depend on its input in some way, you should create a custom `Function` subclass and define assumptions handlers as *described below* (page 111).

The Fully Evaluated Case

At the other end of the spectrum are functions that always evaluate to something no matter what their inputs are. These functions are never left in an unevaluated, symbolic form like $f(x)$.

In this case, you should use a normal Python function using the `def` keyword:

```
>>> def f(x):
...     if x == 0:
...         return 0
...     else:
...         return x + 1
```

```
>>> f(0)
0
>>> f(1)
2
>>> f(x)
x + 1
```

If you find yourself defining an `eval()` (page 106) method on a Function subclass where you always return a value and never return `None`, you should consider just using a normal Python function instead, as there is no benefit to using a symbolic Function subclass in that case (see the [Best Practices for eval\(\)](#) (page 107) section below)

Note that in many cases, functions like these can be represented directly using SymPy classes. For example, the above function can be represented symbolically using [Piecewise](#) (page 415). The Piecewise expression can be evaluated for specific values of x using [subs\(\)](#) (page 941).

```
>>> from sympy import Piecewise, Eq, pprint
>>> f = Piecewise((0, Eq(x, 0)), (x + 1, True))
```

```
>>> pprint(f, use_unicode=True)
{
  0    for x = 0
}
{x + 1 otherwise
>>> f.subs(x, 0)
0
>>> f.subs(x, 1)
2
```

Fully symbolic representations like `Piecewise` have the advantage that they accurately represent symbolic values. For example, in the above Python `def` definition of `f`, `f(x)` implicitly assumes that x is nonzero. The `Piecewise` version handles this case correctly and won't evaluate to the $x \neq 0$ case unless x is known to not be zero.

Another option, if you want a function that not only evaluates, but always evaluates to a numerical value, is to use [lambdify\(\)](#) (page 2100). This will convert a SymPy expression into a function that can be evaluated using NumPy.

```
>>> from sympy import lambdify
>>> func = lambdify(x, Piecewise((0, Eq(x, 0)), (x + 1, True)))
>>> import numpy as np
```

(continues on next page)

(continued from previous page)

```
>>> func(np.arange(5))
array([0., 2., 3., 4., 5.] )
```

Ultimately, the correct tool for the job depends on what you are doing and what exact behavior you want.

3.3.2 Creating a Custom Function

The first step to creating a custom function is to subclass *Function* (page 1050). The name of the subclass will be the name of the function. Different methods should then be defined on this subclass, depending on what functionality you want to provide.

As a motivating example for this document, let's create a custom function class representing the *versine function*. Versine is a trigonometric function which was used historically alongside some of the more familiar trigonometric functions like sine and cosine. It is rarely used today. Versine can be defined by the identity

$$\text{versin}(x) = 1 - \cos(x).$$

SymPy does not already include versine because it is used so rarely in modern mathematics and because it is so easily defined in terms of the more familiar cosine.

Let us start by subclassing Function.

```
>>> class versin(Function):
...     pass
```

At this point, *versin* has no behaviors defined on it. It is very similar to the *undefined functions* (page 103) we discussed above. Note that *versin* is a class, and *versin(x)* is an instance of this class.

```
>>> versin(x)
versin(x)
>>> isinstance(versin(x), versin)
True
```

Note: All the methods described below are optional. They can be included if you want to define the given behavior, but if they are omitted, SymPy will default to leaving things unevaluated. For example, if you do not define *differentiation* (page 118), *diff()* (page 1048) will just return an unevaluated *Derivative* (page 1042).

Defining Automatic Evaluation with `eval()`

Reminder

Remember that `eval()` should be defined with the `@classmethod` decorator.

The first and most common thing we might want to define on our custom function is automatic evaluation, that is, the cases where it will return an actual value instead of just remaining unevaluated as-is.

This is done by defining the class method `eval()`. `eval()` should take the arguments of the function and return either a value or `None`. If it returns `None`, the function will remain unevaluated in that case. This also serves to define the signature of the function (by default, without an `eval()` method, a `Function` subclass will accept any number of arguments).

For our function `versin`, we might recall that $\cos(n\pi) = (-1)^n$ for integer n , so $\text{versin}(n\pi) = 1 - (-1)^n$. We can make `versin` automatically evaluate to this value when passed an integer multiple of π :

```
>>> from sympy import pi, Integer
>>> class versin(Function):
...     @classmethod
...     def eval(cls, x):
...         # If x is an integer multiple of pi, x/pi will cancel and be an
↪Integer
...         n = x/pi
...         if isinstance(n, Integer):
...             return 1 - (-1)**n
```

```
>>> versin(pi)
2
>>> versin(2*pi)
0
```

Here we make use of the fact that if a Python function does not explicitly return a value, it automatically returns `None`. So in the cases where the `if isinstance(n, Integer)` statement is not triggered, `eval()` returns `None` and `versin` remains unevaluated.

```
>>> versin(x*pi)
versin(pi*x)
```

Note: Function subclasses should not redefine `__new__` or `__init__`. If you want to implement behavior that isn't possible with `eval()`, it might make more sense to subclass *Expr* (page 947) rather than `Function`.

`eval()` can take any number of arguments, including an arbitrary number with `*args` and optional keyword arguments. The `.args` of the function will always be the arguments that were passed in by the user. For example

```
>>> class f(Function):
...     @classmethod
```

(continues on next page)

(continued from previous page)

```
...     def eval(cls, x, y=1, *args):
...         return None
```

```
>>> f(1).args
(1,)
>>> f(1, 2).args
(1, 2)
>>> f(1, 2, 3).args
(1, 2, 3)
```

Finally, note that automatic evaluation on floating-point inputs happens automatically once `evalf()` is *defined* (page 114), so you do not need to handle it explicitly in `eval()`.

Best Practices for `eval()`

Certain antipatterns are common when defining `eval()` methods and should be avoided.

- **Don't just return an expression.**

In the above example, we might have been tempted to write

```
>>> from sympy import cos
>>> class versin(Function):
...     @classmethod
...     def eval(cls, x):
...         # !! Not actually a good eval() method !!
...         return 1 - cos(x)
```

However, this would make it so that `versin(x)` would *always* return `1 - cos(x)`, regardless of what `x` is. If all you want is a quick shorthand to `1 - cos(x)`, that is fine, but would be much simpler and more explicit to just *use a Python function as described above* (page 104). If we defined `versin` like this, it would never actually be represented as `versin(x)`, and none of the other behavior we define below would matter, because the other behaviors we are going to define on the `versin` class only apply when the returned object is actually a `versin` instance. So for example, `versin(x).diff(x)` would actually just be `(1 - cos(x)).diff(x)`, instead of calling *the `fdiff()` method we define below* (page 118).

Key Point

The purpose of `eval()` is not to define what the function *is*, mathematically, but rather to specify on what inputs it should automatically evaluate. The mathematical definition of a function is determined through the specification of various mathematical properties with the methods outlined below, like *numerical evaluation* (page 114), *differentiation* (page 118), and so on.

If you find yourself doing this, you should think about what you actually want to achieve. If you just want a shorthand function for an expression, it will be simpler to just *define a Python function* (page 104). If you really do want a symbolic function, think about when you want it to evaluate to something else and when you want it to stay unevaluated. One option is to make your function unevaluated in `eval()` and define a *`doit()` method* (page 116) to evaluate it.

- **Avoid too much automatic evaluation.**

It is recommended to minimize what is evaluated automatically by `eval()`. It is typically better to put more advanced simplifications in [other methods](#) (page 115), like `doit()` (page 116). Remember that whatever you define for automatic evaluation will *always* evaluate.¹ As in the previous point, if you evaluate every value, there is little point to even having a symbolic function in the first place. For example, we might be tempted to evaluate some trig identities on `versin` in `eval()`, but then these identities would always evaluate, and it wouldn't be possible to represent one half of the identity.

One should also avoid doing anything in `eval()` that is slow to compute. SymPy generally assumes that it is cheap to create expressions, and if this is not true, it can lead to performance issues.

Finally, it is recommended to avoid performing automatic evaluation in `eval()` based on assumptions. Instead, `eval()` should typically only evaluate explicit numerical special values and return `None` for everything else. You might have noticed in [the example above](#) (page 106) that we used `isinstance(n, Integer)` instead of checking `n.is_integer` using the assumptions system. We could have done that instead, which would make `versin(n*pi)` evaluate even if `n = Symbol('n', integer=True)`. But this is a case where we might not always want evaluation to happen, and if `n` is a more complicated expression, `n.is_integer` might be more expensive to compute.

Let's consider an example. Using the identity $\cos(x + y) = \cos(x)\cos(y) - \sin(x)\sin(y)$, we can derive the identity

$$\text{versin}(x + y) = \text{versin}(x)\text{versin}(y) - \text{versin}(x) - \text{versin}(y) - \sin(x)\sin(y) + 1.$$

Suppose we decided to automatically expand this in `eval()`:

```
>>> from sympy import Add, sin
>>> class versin(Function):
...     @classmethod
...     def eval(cls, x):
...         # !! Not actually a good eval() method !!
...         if isinstance(x, Add):
...             a, b = x.as_two_terms()
...             return (versin(a)*versin(b) - versin(a) - versin(b)
...                     - sin(a)*sin(b) + 1)
```

This method recursively splits `Add` terms into two parts and applies the above identity.

```
>>> x, y, z = symbols('x y z')
>>> versin(x + y)
-sin(x)*sin(y) + versin(x)*versin(y) - versin(x) - versin(y) + 1
```

But now it's impossible to represent `versin(x + y)` without it expanding. This will affect other methods too. For example, suppose we define [differentiation](#) (see below) (page 118):

¹ While it is technically possible to bypass automatic evaluation by using `evaluate=False`, this is recommended against for two reasons. Firstly, `evaluate=False` is fragile because any function that rebuilds the expression from its `.args` will not keep the `evaluate=False` flag, causing it to evaluate. Secondly, `evaluate=False` tends to be bug prone, because other code may be written expecting the invariants from the automatic evaluation to hold. It is much better to not evaluate such cases at all in `eval()`, and move such simplifications to `doit()` (page 116) instead.

```
>>> class versin(Function):
...     @classmethod
...     def eval(cls, x):
...         # !! Not actually a good eval() method !!
...         if isinstance(x, Add):
...             a, b = x.as_two_terms()
...             return (versin(a)*versin(b) - versin(a) - versin(b)
...                     - sin(a)*sin(b) + 1)
...
...     def fdiff(self, argindex=1):
...         return sin(self.args[0])
```

We would expect `versin(x + y).diff(x)` to return `sin(x + y)`, and indeed, if we hadn't expanded this identity in `eval()`, *it would* (page 119). But with this version, `versin(x + y)` gets automatically expanded before `diff()` gets called, instead we get a more complicated expression:

```
>>> versin(x + y).diff(x)
sin(x)*versin(y) - sin(x) - sin(y)*cos(x)
```

And things are even worse than that. Let's try an `Add` with three terms:

```
>>> versin(x + y + z)
(-sin(y)*sin(z) + versin(y)*versin(z) - versin(y) - versin(z) +
1)*versin(x) - sin(x)*sin(y + z) + sin(y)*sin(z) - versin(x) -
versin(y)*versin(z) + versin(y) + versin(z)
```

We can see that things are getting out of control quite quickly. In fact, `versin(Add(*symbols('x:100')))` (`versin()` on an `Add` with 100 terms) takes over a second to evaluate, and that's just to *create* the expression, without even doing anything with it yet.

Identities like this are better left out of `eval` and implemented in other methods instead (in the case of this identity, `expand_trig()` (page 118)).

- **When restricting the input domain: allow None input assumptions.**

Our example function `versin(x)` is a function from $\mathbb{C}$ to $\mathbb{C}$, so it can accept any input. But suppose we had a function that only made sense with certain inputs. As a second example, let's define a function `divides` as

$$\text{divides}(m, n) = \begin{cases} 1 & \text{for } m \mid n \\ 0 & \text{for } m \nmid n \end{cases}.$$

That is, `divides(m, n)` will be 1 if m divides n and 0 otherwise. `divides` clearly only makes sense if m and n are integers.

We might be tempted to define the `eval()` method for `divides` like this:

```
>>> class divides(Function):
...     @classmethod
...     def eval(cls, m, n):
...         # !! Not actually a good eval() method !!
...         ...
```

(continues on next page)

(continued from previous page)

```

...     # Evaluate for explicit integer m and n. This part is fine.
...     if isinstance(m, Integer) and isinstance(n, Integer):
...         return int(n % m == 0)
...
...     # For symbolic arguments, require m and n to be integer.
...     # If we write the logic this way, we will run into trouble.
...     if not m.is_integer or not n.is_integer:
...         raise TypeError("m and n should be integers")

```

The problem here is that by using `if not m.is_integer`, we are requiring `m.is_integer` to be `True`. If it is `None`, it will fail (see the [guide on booleans and three-valued logic](#) (page 95) for details on what it means for an assumption to be `None`). This is problematic for two reasons. Firstly, it forces the user to define assumptions on any input variable. If the user omits them, it will fail:

```

>>> n, m = symbols('n m')
>>> print(n.is_integer)
None
>>> divides(m, n)
Traceback (most recent call last):
...
TypeError: m and n should be integers

```

Instead they have to write

```

>>> n, m = symbols('n m', integer=True)
>>> divides(m, n)
divides(m, n)

```

This may seem like an acceptable restriction, but there is a bigger problem. Sometimes, SymPy's assumptions system cannot deduce an assumption, even though it is mathematically true. In this case, it will give `None` (`None` means both "undefined" and "cannot compute" in SymPy's assumptions). For example

```

>>> # n and m are still defined as integer=True as above
>>> divides(2, (m**2 + m)/2)
Traceback (most recent call last):
...
TypeError: m and n should be integers

```

Here the expression $(m^2 + m)/2$ is always an integer, but SymPy's assumptions system is not able to deduce this:

```

>>> print(((m**2 + m)/2).is_integer)
None

```

SymPy's assumptions system is always improving, but there will always be cases like this that it cannot deduce, due to the fundamental computational complexity of the problem, and the fact that the general problem is [often undecidable](#).

Consequently, one should always test *negated* assumptions for input variables, that is, fail if the assumption is `False` but allow the assumption to be `None`.

```
>>> class divides(Function):
...     @classmethod
...     def eval(cls, m, n):
...         # Evaluate for explicit integer m and n. This part is fine.
...         if isinstance(m, Integer) and isinstance(n, Integer):
...             return int(n % m == 0)
...
...         # For symbolic arguments, require m and n to be integer.
...         # This is the better way to write this logic.
...         if m.is_integer is False or n.is_integer is False:
...             raise TypeError("m and n should be integers")
```

This still disallows non-integer inputs as desired:

```
>>> divides(1.5, 1)
Traceback (most recent call last):
...
TypeError: m and n should be integers
```

But it does not fail in cases where the assumption is None:

```
>>> divides(2, (m**2 + m)/2)
divides(2, m**2/2 + m/2)
>>> _.subs(m, 2)
0
>>> n, m = symbols('n m') # Redefine n and m without the integer_
↪assumption
>>> divides(m, n)
divides(m, n)
```

Note: This rule of allowing None assumptions only applies to instances where an exception would be raised, such as type checking an input domain. In cases where simplifications or other operations are done, one should treat a None assumption as meaning “can be either True or False” and not perform an operation that might not be mathematically valid.

Assumptions

The next thing you might want to define are the assumptions on our function. The assumptions system allows defining what mathematical properties your function has given its inputs, for example, “ $f(x)$ is positive when x is real.”

The [guide on the assumptions system](#) (page 71) goes into the assumptions system in great detail. It is recommended to read through that guide first to understand what the different assumptions mean and how the assumptions system works.

The simplest case is a function that always has a given assumption regardless of its input. In this case, you can define `is_assumption` directly on the class.

For example, our [example divides function](#) (page 109) is always an integer, because its value is always either 0 or 1:

Note

From here on out in this guide, in the interest of space, we will omit the previous method definitions in the examples unless they are needed for the given example to work. There are *complete examples* (page 122) at the end of this guide with all the methods.

```
>>> class divides(Function):
...     is_integer = True
...     is_negative = False
```

```
>>> divides(m, n).is_integer
True
>>> divides(m, n).is_nonnegative
True
```

In general, however, the assumptions of a function depend on the assumptions of its inputs. In this case, you should define an `_eval_assumption` method.

For our `versin(x)` *example* (page 105), the function is always in $[0, 2]$ when x is real, and it is 0 exactly when x is an even multiple of π . So `versin(x)` should be *nonnegative* whenever x is *real* and *positive* whenever x is *real* and not an *even* multiple of π . Remember that by default, a function's domain is all of $\mathbb{C}$, and indeed `versin(x)` makes perfect sense with non-real x .

To see if x is an even multiple of π , we can use `as_independent()` (page 952) to match x structurally as `coeff*pi`. Pulling apart subexpressions structurally like this in assumptions handlers is preferable to using something like `(x/pi).is_even`, because that will create a new expression x/π . The creation of a new expression is much slower. Furthermore, whenever an expression is created, the constructors that are called when creating the expression will often themselves cause assumptions to be queried. If you are not careful, this can lead to infinite recursion. So a good general rule for assumptions handlers is, **never create a new expression in an assumptions handler**. Always pull apart the args of the function using structural methods like `as_independent`.

Note that `versin(x)` can be nonnegative for nonreal x , for example:

```
>>> from sympy import I
>>> 1 - cos(pi + I*pi)
1 + cosh(pi)
>>> (1 - cos(pi + I*pi)).evalf()
12.5919532755215
```

So for the `_eval_is_nonnegative` handler, we want to return `True` if `x.is_real` is `True` but `None` if `x.is_real` is either `False` or `None`. It is left as an exercise to the reader to handle the cases for nonreal x that make `versin(x)` nonnegative, using similar logic from the `_eval_is_positive` handler.

In the assumptions handler methods, as in all methods, we can access the arguments of the function using `self.args`.

```
>>> from sympy.core.logic import fuzzy_and, fuzzy_not
>>> class versin(Function):
...     def _eval_is_nonnegative(self):
...         # versin(x) is nonnegative if x is real
...         x = self.args[0]
```

(continues on next page)

(continued from previous page)

```

...     if x.is_real is True:
...         return True
...
...     def _eval_is_positive(self):
...         # versin(x) is positive iff x is real and not an even multiple of  $\pi$ 
...         x = self.args[0]
...
...         # x.as_independent(pi, as_Add=False) will split x as a Mul of the
...         # form coeff*pi
...         coeff, pi_ = x.as_independent(pi, as_Add=False)
...         # If pi_ = pi, x = coeff*pi. Otherwise x is not (structurally) of
...         # the form coeff*pi.
...         if pi_ == pi:
...             return fuzzy_and([x.is_real, fuzzy_not(coeff.is_even)])
...         elif x.is_real is False:
...             return False
...         # else: return None. We do not know for sure whether x is an even
...         # multiple of pi

```

```

>>> versin(1).is_nonnegative
True
>>> versin(2*pi).is_positive
False
>>> versin(3*pi).is_positive
True

```

Note the use of fuzzy_ functions in the more complicated `_eval_is_positive()` handler, and the careful handling of the if/elif. It is important when working with assumptions to always be careful about [handling three-valued logic correctly](#) (page 95). This ensures that the method returns the correct answer when `x.is_real` or `coeff.is_even` are `None`.

Warning: Never define `is_assumption` as a `@property` method. Doing so will break the automatic deduction of other assumptions. `is_assumption` should only ever be defined as a class variable equal to `True` or `False`. If the assumption depends on the `.args` of the function somehow, define the `_eval_assumption` method.

In this example, it is not necessary to define `_eval_is_real()` because it is deduced automatically from the other assumptions, since `nonnegative -> real`. In general, you should avoid defining assumptions that the assumptions system can deduce automatically given its [known facts](#) (page 77).

```

>>> versin(1).is_real
True

```

The assumptions system is often able to deduce more than you might think. For example, from the above, it can deduce that `versin(2*n*pi)` is zero when `n` is an integer.

```

>>> n = symbols('n', integer=True)
>>> versin(2*n*pi).is_zero
True

```

It's always worth checking if the assumptions system can deduce something automatically before manually coding it.

Finally, a word of warning: be very careful about correctness when coding assumptions. Make sure to use the exact *definitions* (page 77) of the various assumptions, and always check that you're handling None cases correctly with the fuzzy three-valued logic functions. Incorrect or inconsistent assumptions can lead to subtle bugs. It's recommended to use unit tests to check all the various cases whenever your function has a nontrivial assumption handler. All functions defined in SymPy itself are required to be extensively tested.

Numerical Evaluation with `evalf()`

Here we show how to define how a function should numerically evaluate to a floating point *Float* (page 982) value, for instance, via `evalf()`. Implementing numerical evaluation enables several behaviors in SymPy. For example, once `evalf()` is defined, you can plot your function, and things like inequalities can evaluate to explicit values.

If your function has the same name as a function in *mpmath*, which is the case for most functions included with SymPy, numerical evaluation will happen automatically and you do not need to do anything.

If this is not the case, numerical evaluation can be specified by defining the method `_eval_evalf(self, prec)`, where `prec` is the binary precision of the input. The method should return the expression evaluated to the given precision, or None if this is not possible.

Note: The `prec` argument to `_eval_evalf()` is the *binary* precision, that is, the number of bits in the floating-point representation. This differs from the first argument to the `evalf()` method, which is the *decimal* precision, or `dps`. For example, the default binary precision of *Float* is 53, corresponding to a decimal precision of 15. Therefore, if your `_eval_evalf()` method recursively calls `evalf` on another expression, it should call `expr._eval_evalf(prec)` rather than `expr.evalf(prec)`, as the latter will incorrectly use `prec` as the decimal precision.

We can define numerical evaluation for *our example versin(x) function* (page 105) by recursively evaluating $2\sin^2(\frac{x}{2})$, which is a more numerically stable way of writing $1 - \cos(x)$.

```
>>> from sympy import sin
>>> class versin(Function):
...     def _eval_evalf(self, prec):
...         return (2*sin(self.args[0]/2)**2)._eval_evalf(prec)
```

```
>>> versin(1).evalf()
0.459697694131860
```

Once `_eval_evalf()` is defined, this enables the automatic evaluation of floating-point inputs. It is not required to implement this manually in `eval()` (page 106).

```
>>> versin(1.)
0.459697694131860
```

Note that `evalf()` may be passed any expression, not just one that can be evaluated numerically. In this case, it is expected that the numerical parts of an expression will be evaluated. A general pattern to follow is to recursively call `_eval_evalf(prec)` on the arguments of the function.

Whenever possible, it's best to reuse the evalf functionality defined in existing SymPy functions. However, in some cases it will be necessary to use mpmath directly.

Rewriting and Simplification

Various simplification functions and methods allow specifying their behavior on custom subclasses. Not every function in SymPy has such hooks. See the documentation of each individual function for details.

rewrite()

The `rewrite()` (page 940) method allows rewriting an expression in terms of a specific function or rule. For example,

```
>>> sin(x).rewrite(cos)
cos(x - pi/2)
```

To implement rewriting, define a method `_eval_rewrite(self, rule, args, **hints)`, where

- `rule` is the *rule* passed to the `rewrite()` method. Typically `rule` will be the class of the object to be rewritten to, although for more complex rewrites, it can be anything. Each object that defines `_eval_rewrite()` defines what rule(s) it supports. Many SymPy functions rewrite to common classes, like `expr.rewrite(Add)`, to perform simplifications or other computations.
- `args` are the arguments of the function to be used for rewriting. This should be used instead of `self.args` because any recursive expressions in the `args` will be rewritten in `args` (assuming the caller used `rewrite(deep=True)`, which is the default).
- `**hints` are additional keyword arguments which may be used to specify the behavior of the rewrite. Unknown hints should be ignored as they may be passed to other `_eval_rewrite()` methods.

The method should return a rewritten expression, using `args` as the arguments to the function, or `None` if the expression should be unchanged.

For our *versin example* (page 105), an obvious rewrite we can implement is rewriting `versin(x)` as `1 - cos(x)`:

```
>>> class versin(Function):
...     def _eval_rewrite(self, rule, args, **hints):
...         if rule == cos:
...             return 1 - cos(*args)
>>> versin(x).rewrite(cos)
1 - cos(x)
```

Once we've defined this, `simplify()` (page 661) is now able to simplify some expressions containing `versin`:

```
>>> from sympy import simplify
>>> simplify(versin(x) + cos(x))
1
```

doit()

The `doit()` (page 932) method is used to evaluate “unevaluated” functions. To define `doit()` implement `doit(self, deep=True, **hints)`. If `deep=True`, `doit()` should recursively call `doit()` on the arguments. `**hints` will be any other keyword arguments passed to the user, which should be passed to any recursive calls to `doit()`. You can use `hints` to allow the user to specify specific behavior for `doit()`.

The typical usage of `doit()` in custom Function subclasses is to perform more advanced evaluation which is not performed in `eval()` (page 106).

For example, for our [divides example](#) (page 109), there are several instances that could be simplified using some identities. For example, we defined `eval()` to evaluate on explicit integers, but we might also want to evaluate examples like `divides(k, k*n)` where the divisibility is symbolically true. One of the [best practices for eval\(\)](#) (page 107) is to avoid too much automatic evaluation. Automatically evaluating in this case might be considered too much, as it would make use of the assumptions system, which could be expensive. Furthermore, we might want to be able to represent `divides(k, k*n)` without it always evaluating.

The solution is to implement these more advanced evaluations in `doit()`. That way, we can explicitly perform them by calling `expr.doit()`, but they won’t happen by default. An example `doit()` for `divides` that performs this simplification (along with the [above definition of eval\(\)](#) (page 111)) might look like this:

Note: If `doit()` returns a Python `int` literal, convert it to an `Integer` so that the returned object is a SymPy type.

```
>>> from sympy import Integer
>>> class divides(Function):
...     # Define evaluation on basic inputs, as well as type checking that the
...     # inputs are not nonintegral.
...     @classmethod
...     def eval(cls, m, n):
...         # Evaluate for explicit integer m and n.
...         if isinstance(m, Integer) and isinstance(n, Integer):
...             return int(n % m == 0)
...
...         # For symbolic arguments, require m and n to be integer.
...         if m.is_integer is False or n.is_integer is False:
...             raise TypeError("m and n should be integers")
...
...     # Define doit() as further evaluation on symbolic arguments using
...     # assumptions.
...     def doit(self, deep=False, **hints):
...         m, n = self.args
...         # Recursively call doit() on the args whenever deep=True.
...         # Be sure to pass deep=True and **hints through here.
...         if deep:
...             m, n = m.doit(deep=deep, **hints), n.doit(deep=deep, **hints)
...
...         # divides(m, n) is 1 iff n/m is an integer. Note that m and n are
...         # already assumed to be integers because of the logic in eval().
...         isint = (n/m).is_integer
```

(continues on next page)

(continued from previous page)

```

...     if isint is True:
...         return Integer(1)
...     elif isint is False:
...         return Integer(0)
...     else:
...         return divides(m, n)

```

(Note that this uses the [convention](#) that $k \mid 0$ for all k so that we do not need to check if m or n are nonzero. If we used a different convention we would need to check if `m.is_zero` and `n.is_zero` before performing the simplification.)

```

>>> n, m, k = symbols('n m k', integer=True)
>>> divides(k, k*n)
divides(k, k*n)
>>> divides(k, k*n).doit()
1

```

Another common way to implement `doit()` is for it to always return another expression. This effectively treats the function as an “unevaluated” form of another expression.

For example, let’s define a function for [fused multiply-add](#): $\text{FMA}(x, y, z) = xy + z$. It may be useful to express this function as a distinct function, e.g., for the purposes of code generation, but it would also be useful in some contexts to “evaluate” $\text{FMA}(x, y, z)$ to $x*y + z$ so that it can properly simplify with other expressions.

```

>>> from sympy import Number
>>> class FMA(Function):
...     """
...     FMA(x, y, z) = x*y + z
...     """
...     @classmethod
...     def eval(cls, x, y, z):
...         # Number is the base class of Integer, Rational, and Float
...         if all(isinstance(i, Number) for i in [x, y, z]):
...             return x*y + z
...
...     def doit(self, deep=True, **hints):
...         x, y, z = self.args
...         # Recursively call doit() on the args whenever deep=True.
...         # Be sure to pass deep=True and **hints through here.
...         if deep:
...             x = x.doit(deep=deep, **hints)
...             y = y.doit(deep=deep, **hints)
...             z = z.doit(deep=deep, **hints)
...         return x*y + z

```

```

>>> x, y, z = symbols('x y z')
>>> FMA(x, y, z)
FMA(x, y, z)
>>> FMA(x, y, z).doit()
x*y + z

```

Most custom functions will not want to define `doit()` in this way. However, this can provide

a happy medium between having a function that always evaluates and a function that never evaluates, producing a function that doesn't evaluate by default but can be evaluated on demand (see the [discussion above](#) (page 107)).

`expand()`

The `expand()` (page 1053) function “expands” an expression in various ways. It is actually a wrapper around several sub-expansion hints. Each function corresponds to a hint to the `expand()` function/method. A specific `expand hint` can be defined in a custom function by defining `_eval_expand_hint(self, **hints)`. See the documentation of `expand()` (page 1053) for details on which hints are defined and the documentation for each specific `expand_hint()` function (e.g., `expand_trig()` (page 1062)) for details on what each hint is designed to do.

The `**hints` keyword arguments are additional hints that may be passed to the `expand` function to specify additional behavior (these are separate from the predefined `hints` described in the previous paragraph). Unknown hints should be ignored as they may apply to other functions' custom `expand()` methods. A common hint to define is `force`, where `force=True` would force an expansion that might not be mathematically valid for all the given input assumptions. For example, `expand_log(log(x*y), force=True)` produces `log(x) + log(y)` even though this identity is not true for all complex `x` and `y` (typically `force=False` is the default).

Note that `expand()` automatically takes care of recursively expanding expressions using its own deep flag, so `_eval_expand_*` methods should not recursively call `expand` on the arguments of the function.

For our [versin example](#) (page 105), we can define rudimentary trig expansion by defining an `_eval_expand_trig` method, which recursively calls `expand_trig()` on `1 - cos(x)`:

```
>>> from sympy import expand_trig
>>> y = symbols('y')
>>> class versin(Function):
...     def _eval_expand_trig(self, **hints):
...         x = self.args[0]
...         return expand_trig(1 - cos(x))
>>> versin(x + y).expand(trig=True)
sin(x)*sin(y) - cos(x)*cos(y) + 1
```

A more sophisticated implementation might attempt to rewrite the result of `expand_trig(1 - cos(x))` back into `versin` functions. This is left as an exercise for the reader.

Differentiation

To define differentiation via `diff()` (page 1048), define a method `fdiff(self, argindex)`. `fdiff()` should return the derivative of the function, without considering the chain rule, with respect to the `argindex`-th variable. `argindex` is indexed starting at 1.

That is, `f(x1, ..., xi, ..., xn).fdiff(i)` should return $\frac{d}{dx_i} f(x_1, \dots, x_i, \dots, x_n)$, where x_k are independent of one another. `diff()` will automatically apply the chain rule using the result of `fdiff()`. User code should use `diff()` and not call `fdiff()` directly.

Note: Function subclasses should define differentiation using `fdiff()`. Subclasses of [Expr](#) (page 947) that aren't Function subclasses will need to define `_eval_derivative()` instead.

It is not recommended to redefine `_eval_derivative()` on a Function subclass.

For our [versin example function](#) (page 105), the derivative is $\sin(x)$.

```
>>> class versin(Function):
...     def fdiff(self, argindex=1):
...         # argindex indexes the args, starting at 1
...         return sin(self.args[0])
```

```
>>> versin(x).diff(x)
sin(x)
>>> versin(x**2).diff(x)
2*x*sin(x**2)
>>> versin(x + y).diff(x)
sin(x + y)
```

As an example of a function that has multiple arguments, consider the [fused multiply-add \(FMA\) example](#) (page 117) defined above ($\text{FMA}(x, y, z) = xy + z$).

We have

$$\frac{d}{dx} \text{FMA}(x, y, z) = y,$$

$$\frac{d}{dy} \text{FMA}(x, y, z) = x,$$

$$\frac{d}{dz} \text{FMA}(x, y, z) = 1.$$

So the `fdiff()` method for FMA would look like this:

```
>>> from sympy import Number, symbols
>>> x, y, z = symbols('x y z')
>>> class FMA(Function):
...     """
...     FMA(x, y, z) = x*y + z
...     """
...     def fdiff(self, argindex):
...         # argindex indexes the args, starting at 1
...         x, y, z = self.args
...         if argindex == 1:
...             return y
...         elif argindex == 2:
...             return x
...         elif argindex == 3:
...             return 1
```

```
>>> FMA(x, y, z).diff(x)
y
>>> FMA(x, y, z).diff(y)
x
>>> FMA(x, y, z).diff(z)
1
>>> FMA(x**2, x + 1, y).diff(x)
x**2 + 2*x*(x + 1)
```

To leave a derivative unevaluated, raise `sympy.core.function.ArgumentIndexError(self, argindex)`. This is the default behavior if `fdiff()` is not defined. Here is an example function $f(x, y)$ that is linear in the first argument and has an unevaluated derivative on the second argument.

```
>>> from sympy.core.function import ArgumentIndexError
>>> class f(Function):
...     @classmethod
...     def eval(cls, x, y):
...         pass
...
...     def fdiff(self, argindex):
...         if argindex == 1:
...             return 1
...         raise ArgumentIndexError(self, argindex)
```

```
>>> f(x, y).diff(x)
1
>>> f(x, y).diff(y)
Derivative(f(x, y), y)
```

Printing

You can define how a function prints itself with the various [printers](#) (page 2134) such as the [string printer](#) (page 2178), [pretty printers](#) (page 2139), and [LaTeX printer](#) (page 2170), as well as code printers for various languages like [C](#) (page 2141) and [Fortran](#) (page 2149).

In most cases, you will not need to define any printing methods. The default behavior is to print functions using their name. However, in some cases we may want to define special printing for a function.

For example, for our [divides example above](#) (page 109), we may want the LaTeX printer to print a more mathematical expression. Let's make the LaTeX printer represent `divides(m, n)` as $\left[m \middle| n \right]$, which looks like $[m|n]$ (here $[P]$ is the [Iverson bracket](#), which is 1 if P is true and 0 if P is false).

There are two primary ways to define printing for SymPy objects. One is to define a printer on the printer class. Most classes that are part of the SymPy library should use this method, by defining the printers on the respective classes in `sympy.printing`. For user code, this may be preferable if you are defining a custom printer, or if you have many custom functions that you want to define printing for. See [Example of Custom Printer](#) (page 2136) for an example of how to define a printer in this way.

The other way is to define the printing as a method on the function class. To do this, first look up the `printmethod` attribute on the printer you want to define the printing for. This is the name of the method you should define for that printer. For the LaTeX printer, [LatexPrinter.printmethod](#) (page 2170) is `'_latex'`. The print method always takes one argument, `printer`. `printer._print` should be used to recursively print any other expressions, including the arguments of the function.

So to define our `divides` LaTeX printer, we will define the function `_latex(self, printer)` on the class, like this:


```
>>> from sympy import latex
>>> class divides(Function):
...     def _latex(self, printer):
...         m, n = self.args
...         _m, _n = printer._print(m), printer._print(n)
...         return r'\left [ %s \middle | %s \right ]' % (_m, _n)
```

```
>>> print(latex(divides(m, n)))
\left [ m \middle | n \right ]
```

See [Example of Custom Printing Method](#) (page 2137) for more details on how to define printer methods and some pitfalls to avoid. Most importantly, you should always use `printer._print()` to recursively print the arguments of the function inside of a custom printer.

Other Methods

Several other methods can be defined on custom functions to specify various behaviors.

`inverse()`

The `inverse(self, argindex=1)` method can be defined to specify the inverse of the function. This is used by [solve\(\)](#) (page 836) and [solveset\(\)](#) (page 866). The `argindex` argument is the argument of the function, starting at 1 (similar to the same argument name for the [fdiff\(\) method](#) (page 118)).

`inverse()` should return a function (not an expression) for the inverse. If the inverse is a larger expression than a single function, it can return a lambda function.

`inverse()` should only be defined for functions that are one-to-one. In other words, $f(x)$. `inverse()` is the [left inverse](#) of $f(x)$. Defining `inverse()` on a function that is not one-to-one may result in `solve()` not giving all possible solutions to an expression containing the function.

Our [example versine function](#) (page 105) is not one-to-one (because cosine is not), but its inverse `arcversin` is. We may define it as follows (using the same naming convention as other inverse trig functions in SymPy):

```
>>> class aversin(Function):
...     def inverse(self, argindex=1):
...         return versin
```

This makes `solve()` work on `aversin(x)`:

```
>>> from sympy import solve
>>> solve(aversin(x) - y, x)
[versin(y)]
```

as_real_imag()

The method `as_real_imag()` (page 956) method defines how to split a function into its real and imaginary parts. It is used by various SymPy functions that operate on the real and imaginary parts of an expression separately.

`as_real_imag(self, deep=True, **hints)` should return a 2-tuple containing the real part and imaginary part of the function. That is `expr.as_real_imag()` returns `(re(expr), im(expr))`, where `expr == re(expr) + im(expr)*I`, and `re(expr)` and `im(expr)` are real.

If `deep=True`, it should recursively call `as_real_imag(deep=True, **hints)` on its arguments. As with `doit()` (page 116) and `the _eval_expand_*() methods` (page 118), `**hints` may be any hints to allow the user to specify the behavior of the method. Unknown hints should be ignored and passed through on any recursive calls in case they are meant for other `as_real_imag()` methods.

For our *versin example* (page 105), we can recursively use the `as_real_imag()` that is already defined for `1 - cos(x)`.

```
>>> class versin(Function):
...     def as_real_imag(self, deep=True, **hints):
...         return (1 - cos(self.args[0])).as_real_imag(deep=deep, **hints)
>>> versin(x).as_real_imag()
(-cos(re(x))*cosh(im(x)) + 1, sin(re(x))*sinh(im(x)))
```

Defining `as_real_imag()` also automatically makes `expand_complex()` (page 1062) work.

```
>>> versin(x).expand(complex=True)
I*sin(re(x))*sinh(im(x)) - cos(re(x))*cosh(im(x)) + 1
```

Miscellaneous _eval_* methods

There are many other functions in SymPy whose behavior can be defined on custom functions via a custom `_eval_*` method, analogous to the ones described above. See the documentation of the specific function for details on how to define each method.

3.3.3 Complete Examples

Here are complete examples for the example functions defined in this guide. See the above sections for details on each method.

Versine

The versine (versed sine) function is defined as

$$\text{versin}(x) = 1 - \cos(x).$$

Versine is an example of a simple function defined for all complex numbers. The mathematical definition is simple, which makes it straightforward to define all the above methods on it (in most cases we can just reuse the existing SymPy logic defined on $1 - \cos(x)$).

Definition

```
>>> from sympy import Function, cos, expand_trig, Integer, pi, sin
>>> from sympy.core.logic import fuzzy_and, fuzzy_not
>>> class versin(Function):
...     r"""
...     The versine function.
...
...      $\operatorname{versin}(x) = 1 - \cos(x) = 2\sin(x/2)^2$ .
...
...     Geometrically, given a standard right triangle with angle  $x$  in the
...     unit circle, the versine of  $x$  is the positive horizontal distance from
...     the right angle of the triangle to the rightmost point on the unit
...     circle. It was historically used as a more numerically accurate way to
...     compute  $1 - \cos(x)$ , but it is rarely used today.
...
...     References
...     =====
...
...     .. [1] https://en.wikipedia.org/wiki/Versine
...     .. [2] https://blogs.scientificamerican.com/roots-of-unity/10-secret-
    trig-functions-your-math-teachers-never-taught-you/
...     """
...     # Define evaluation on basic inputs.
...     @classmethod
...     def eval(cls, x):
...         # If  $x$  is an explicit integer multiple of  $\pi$ ,  $x/\pi$  will cancel and
...         # be an Integer.
...         n = x/pi
...         if isinstance(n, Integer):
...             return 1 - (-1)**n
...
...     # Define numerical evaluation with evalf().
...     def _eval_evalf(self, prec):
...         return (2*sin(self.args[0]/2)**2)._eval_evalf(prec)
...
...     # Define basic assumptions.
...     def _eval_is_nonnegative(self):
...         #  $\text{versin}(x)$  is nonnegative if  $x$  is real
...         x = self.args[0]
...         if x.is_real is True:
...             return True
```

(continues on next page)

(continued from previous page)

```

...
def _eval_is_positive(self):
    # versin(x) is positive iff x is real and not an even multiple of  $\pi$ 
    x = self.args[0]

    # x.as_independent(pi, as_Add=False) will split x as a Mul of the
    # form n*pi
    coeff, pi_ = x.as_independent(pi, as_Add=False)
    # If pi_ = pi, x = coeff*pi. Otherwise pi_ = 1 and x is not
    # (structurally) of the form n*pi.
    if pi_ == pi:
        return fuzzy_and([x.is_real, fuzzy_not(coeff.is_even)])
    elif x.is_real is False:
        return False
    # else: return None. We do not know for sure whether x is an even
    # multiple of pi

    # Define the behavior for various simplification and rewriting
    # functions.
    def _eval_rewrite(self, rule, args, **hints):
        if rule == cos:
            return 1 - cos(*args)
        elif rule == sin:
            return 2*sin(x/2)**2

    def _eval_expand_trig(self, **hints):
        x = self.args[0]
        return expand_trig(1 - cos(x))

    def as_real_imag(self, deep=True, **hints):
        # reuse _eval_rewrite(cos) defined above
        return self.rewrite(cos).as_real_imag(deep=deep, **hints)

    # Define differentiation.
    def fdiff(self, argindex=1):
        return sin(self.args[0])

```

Examples

Evaluation:

```

>>> x, y = symbols('x y')
>>> versin(x)
versin(x)
>>> versin(2*pi)
0
>>> versin(1.0)
0.459697694131860

```

Assumptions:

```

>>> n = symbols('n', integer=True)
>>> versin(n).is_real
True
>>> versin((2*n + 1)*pi).is_positive
True
>>> versin(2*n*pi).is_zero
True
>>> print(versin(n*pi).is_positive)
None
>>> r = symbols('r', real=True)
>>> print(versin(r).is_positive)
None
>>> nr = symbols('nr', real=False)
>>> print(versin(nr).is_nonnegative)
None

```

Simplification:

```

>>> a, b = symbols('a b', real=True)
>>> from sympy import I
>>> versin(x).rewrite(cos)
1 - cos(x)
>>> versin(x).rewrite(sin)
2*sin(x/2)**2
>>> versin(2*x).expand(trig=True)
2 - 2*cos(x)**2
>>> versin(a + b*I).expand(complex=True)
I*sin(a)*sinh(b) - cos(a)*cosh(b) + 1

```

Differentiation:

```

>>> versin(x).diff(x)
sin(x)

```

Solving:

(a more general version of `aversin` would have all the above methods defined as well)

```

>>> class aversin(Function):
...     def inverse(self, argindex=1):
...         return versin
>>> from sympy import solve
>>> solve(aversin(x**2) - y, x)
[-sqrt(versin(y)), sqrt(versin(y))]

```

divides

`divides` is a function defined by

$$\text{divides}(m, n) = \begin{cases} 1 & \text{for } m \mid n \\ 0 & \text{for } m \nmid n \end{cases},$$

that is, `divides(m, n)` is 1 if m divides n and 0 if m does not divide n . It is only defined for integer m and n . For the sake of simplicity, we use the convention that $m \mid 0$ for all integer m .

`divides` is an example of a function that is only defined for certain input values (integers). `divides` also gives an example of defining a custom printer (`_latex()`).

Definition

```
>>> from sympy import Function, Integer
>>> from sympy.core.logic import fuzzy_not
>>> class divides(Function):
...     r"""
...     $$\operatorname{divides}(m, n) = \begin{cases} 1 & \text{for } m \mid n \\ 0 & \text{for } m \nmid n \end{cases}.$
...     \mid n \mid 0 & \text{for } m \mid n \end{cases}.$
...
...     That is, ``divides(m, n)`` is ``1`` if ``m`` divides ``n`` and ``0``
...     if ``m`` does not divide ``n``. It is undefined if ``m`` or ``n`` are
...     not integers. For simplicity, the convention is used that
...     ``divides(m, 0) = 1`` for all integers ``m``.
...
...     References
...     =====
...
...     .. [1] https://en.wikipedia.org/wiki/Divisor#Definition
...
...     # Define evaluation on basic inputs, as well as type checking that the
...     # inputs are not nonintegral.
...     @classmethod
...     def eval(cls, m, n):
...         # Evaluate for explicit integer m and n.
...         if isinstance(m, Integer) and isinstance(n, Integer):
...             return int(n % m == 0)
...
...         # For symbolic arguments, require m and n to be integer.
...         if m.is_integer is False or n.is_integer is False:
...             raise TypeError("m and n should be integers")
...
...     # Define basic assumptions.
...
...     # divides is always either 0 or 1.
...     is_integer = True
...     is_negative = False
...
...     # Whether divides(m, n) is 0 or 1 depends on m and n. Note that this
...     # method only makes sense because we don't automatically evaluate on
```

(continues on next page)

(continued from previous page)

```

...     # such cases, but instead simplify these cases in doit() below.
...     def _eval_is_zero(self):
...         m, n = self.args
...         if m.is_integer and n.is_integer:
...             return fuzzy_not((n/m).is_integer)
...
...     # Define doit() as further evaluation on symbolic arguments using
...     # assumptions.
...     def doit(self, deep=False, **hints):
...         m, n = self.args
...         # Recursively call doit() on the args whenever deep=True.
...         # Be sure to pass deep=True and **hints through here.
...         if deep:
...             m, n = m.doit(deep=deep, **hints), n.doit(deep=deep, **hints)
...
...         # divides(m, n) is 1 iff n/m is an integer. Note that m and n are
...         # already assumed to be integers because of the logic in eval().
...         isint = (n/m).is_integer
...         if isint is True:
...             return Integer(1)
...         elif isint is False:
...             return Integer(0)
...         else:
...             return divides(m, n)
...
...     # Define LaTeX printing for use with the latex() function and the
...     # Jupyter notebook.
...     def _latex(self, printer):
...         m, n = self.args
...         _m, _n = printer._print(m), printer._print(n)
...         return r'\left [ %s \middle | %s \right ]' % (_m, _n)
...

```

Examples

Evaluation

```

>>> from sympy import symbols
>>> n, m, k = symbols('n m k', integer=True)
>>> divides(3, 10)
0
>>> divides(3, 12)
1
>>> divides(m, n).is_integer
True
>>> divides(k, 2*k)
divides(k, 2*k)
>>> divides(k, 2*k).is_zero
False
>>> divides(k, 2*k).doit()
1

```

Printing:

```
>>> str(divides(m, n)) # This is using the default str printer
'divides(m, n)'
>>> print(latex(divides(m, n)))
\left [ m \middle | n \right ]
```

Fused Multiply-Add (FMA)

Fused Multiply-Add (FMA) is a multiplication followed by an addition:

$$\text{FMA}(x, y, z) = xy + z.$$

It is often implemented in hardware as a single floating-point operation that has better rounding and performance than the equivalent combination of multiplication and addition operations.

FMA is an example of a custom function that is defined as an unevaluated “shorthand” to another function. This is because the `doit()` (page 116) method is defined to return $x*y + z$, meaning the FMA function can easily be evaluated to the expression it represents, but the `eval()` (page 106) method does *not* return anything (except when x , y , and z are all explicit numeric values), meaning that it stays unevaluated by default.

Contrast this with the `versine` (page 123) example, which treats `versin` as a first-class function in its own regard. Even though `versin(x)` can be expressed in terms of other functions ($1 - \cos(x)$) it does not evaluate on general symbolic inputs in `versin.eval()`, and `versin.doit()` is not defined at all.

FMA also represents an example of a continuous function defined on multiple variables, which demonstrates how `argindex` works in the `fdiff` (page 118) example.

Finally, FMA shows an example of defining some code printers for C and C++ (using the method names from `C99CodePrinter.printmethod` (page 2141) and `CXX11CodePrinter.printmethod` (page 2144)), since that is a typical use-case for this function.

The mathematical definition of FMA is very simple and it would be easy to define every method on it, but only a handful are shown here. The `versine` (page 123) and `divides` (page 126) examples show how to define the other important methods discussed in this guide.

Note that if you want to actually use fused-multiply add for code generation, there is already a version in SymPy `sympy.codegen.cfunctions.fma()` which is supported by the existing code printers. The version here is only designed to serve as an example.

Definition

```
>>> from sympy import Number, symbols, Add, Mul
>>> x, y, z = symbols('x y z')
>>> class FMA(Function):
...     """
...     FMA(x, y, z) = x*y + z
...
...     FMA is often defined as a single operation in hardware for better
...     rounding and performance.
```

(continues on next page)

(continued from previous page)

```

...     FMA can be evaluated by using the doit() method.
...
...     References
...     =====
...
...     .. [1] https://en.wikipedia.org/wiki/Multiply%E2%80%93accumulate\_
...     ↪ operation#Fused\_multiply%E2%80%93add
...     """
...     # Define automatic evaluation on explicit numbers
...     @classmethod
...     def eval(cls, x, y, z):
...         # Number is the base class of Integer, Rational, and Float
...         if all(isinstance(i, Number) for i in [x, y, z]):
...             return x*y + z
...
...     # Define numerical evaluation with evalf().
...     def _eval_evalf(self, prec):
...         return self.doit(deep=False)._eval_evalf(prec)
...
...     # Define full evaluation to Add and Mul in doit(). This effectively
...     # treats FMA(x, y, z) as just a shorthand for x*y + z that is useful
...     # to have as a separate expression in some contexts and which can be
...     # evaluated to its expanded form in other contexts.
...     def doit(self, deep=True, **hints):
...         x, y, z = self.args
...         # Recursively call doit() on the args whenever deep=True.
...         # Be sure to pass deep=True and **hints through here.
...         if deep:
...             x = x.doit(deep=deep, **hints)
...             y = y.doit(deep=deep, **hints)
...             z = z.doit(deep=deep, **hints)
...         return x*y + z
...
...     # Define FMA.rewrite(Add) and FMA.rewrite(Mul).
...     def _eval_rewrite(self, rule, args, **hints):
...         x, y, z = self.args
...         if rule in [Add, Mul]:
...             return self.doit()
...
...     # Define differentiation.
...     def fdiff(self, argindex):
...         # argindex indexes the args, starting at 1
...         x, y, z = self.args
...         if argindex == 1:
...             return y
...         elif argindex == 2:
...             return x
...         elif argindex == 3:
...             return 1
...
...     # Define code printers for ccode() and cxxcode()

```

(continues on next page)

(continued from previous page)

```
...     def _ccode(self, printer):
...         x, y, z = self.args
...         _x, _y, _z = printer._print(x), printer._print(y), printer._
→print(z)
...         return "fma(%s, %s, %s)" % (_x, _y, _z)
...
...     def _cxxcode(self, printer):
...         x, y, z = self.args
...         _x, _y, _z = printer._print(x), printer._print(y), printer._
→print(z)
...         return "std::fma(%s, %s, %s)" % (_x, _y, _z)
```

Examples

Evaluation:

```
>>> x, y, z = symbols('x y z')
>>> FMA(2, 3, 4)
10
>>> FMA(x, y, z)
FMA(x, y, z)
>>> FMA(x, y, z).doit()
x*y + z
>>> FMA(x, y, z).rewrite(Add)
x*y + z
>>> FMA(2, pi, 1).evalf()
7.28318530717959
```

Differentiation

```
>>> FMA(x, x, y).diff(x)
2*x
>>> FMA(x, y, x).diff(x)
y + 1
```

Code Printers

```
>>> from sympy import ccode, cxxcode
>>> ccode(FMA(x, y, z))
'fma(x, y, z)'
>>> cxxcode(FMA(x, y, z))
'std::fma(x, y, z)'
```

3.3.4 Additional Tips

- SymPy includes dozens of functions. These can serve as useful examples for how to write a custom function, especially if the function is similar to one that is already implemented. Remember that everything in this guide applies equally well to functions that are included with SymPy and user-defined functions. Indeed, this guide is designed to serve as both a developer guide for contributors to SymPy and a guide for end-users of SymPy.
 - If you have many custom functions that share common logic, you can use a common base class to contain this shared logic. For an example of this, see the [source code for the trigonometric functions in SymPy](#), which use `TrigonometricFunction`, `InverseTrigonometricFunction`, and `ReciprocalTrigonometricFunction` base classes with some shared logic.
 - As with any code, it is a good idea to write extensive tests for your function. The [SymPy test suite](#) is a good resource for examples of how to write tests for such functions. All code included in SymPy itself is required to be tested. Functions included in SymPy should also always contain a docstring with references, a mathematical definition, and doctest examples.
-

3.4 Solve Equations

The Python package SymPy can symbolically solve equations, differential equations, linear equations, nonlinear equations, matrix problems, inequalities, Diophantine equations, and evaluate integrals. SymPy can also solve numerically.

Learn how to use SymPy computer algebra system to:

Description	Example	Solution
<i>Solve an equation algebraically</i> (page 132)	$x^2 = y$	$x \in \{-\sqrt{y}, \sqrt{y}\}$
<i>Solve a system of equations algebraically</i> (page 836)	$x^2 + y = 2, x - y = 4$	$\{(x = -3, y = -7), (x = 2, y = 2)\}$
<i>Solve an equation numerically</i> (page 848)	$\cos(x) = x$	$x \approx 0.739085133215161$
<i>Solve an ordinary differential equation algebraically</i> (page 755)	$y''(x) + 9y(x) = 0$	$y(x) = C_1 \sin(3x) + C_2 \cos(3x)$
<i>Solve a system of linear equations algebraically</i> (page 836)	$x + y = 2, x - y = 0$	$x = 1, y = 1$
<i>Solve a system of nonlinear equations algebraically</i> (page 836)	$x^2 + y^3 = 1, x^3 - y^2 = 0$	$x = 1, y = 0$
<i>Solve a matrix problem algebraically</i> (page 1312)	$\begin{bmatrix} 1 & 1 \\ 1 & -1 \\ 2 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \\ 1 \end{bmatrix}$	
<i>Solve an inequality algebraically</i> (page 750)	$x^2 < 4$	$-2 < x < 2$
<i>Solve a system of inequalities algebraically</i> (page 750)	$x^2 < 4, x > 0$	$0 < x < 2$
<i>Solve (find the roots of) a polynomial algebraically</i> (page 2342)	$x^2 - x = 0$	$x \in \{0, 1\}$
<i>Solve a Diophantine equation (find integer solutions to a polynomial equation) algebraically</i> (page 714)	$x^2 - 4xy + 8y^2 - 3x + 7y - 5 = 0$	$\{(x = 2, y = 1), (x = 5, y = 1)\}$

Note: SymPy has a function called `solve()` (page 836) which is designed to find the roots of an equation or system of equations. SymPy `solve()` (page 836) may or may not be what you need for a particular problem, so we recommend you use the links on this page to learn how to “solve” your problem. And while a common, colloquial expression is, for example, “*solve an integral*” (page 575), in SymPy’s terminology it would be “*evaluate an integral*.” (page 575)

3.4.1 Solve an equation algebraically

Use SymPy to solve an equation algebraically (symbolically). For example, solving $x^2 = y$ for x yields $x \in \{-\sqrt{y}, \sqrt{y}\}$.

Alternatives to consider:

- SymPy can also *solve many other types of problems including sets of equations* (page 131).
- Some equations cannot be solved algebraically (either at all or by SymPy), so you may have to *solve your equation numerically* (page 848) instead.

There are two high-level functions to solve equations, `solve()` (page 836) and `solveset()` (page 866). Here is a simple example of each:

`solve()` (page 836)

```
>>> from sympy.abc import x, y
>>> from sympy import solve
>>> solve(x**2 - y, x, dict=True)
[{x: -sqrt(y)}, {x: sqrt(y)}]
```

`solveset()` (page 866)

```
>>> from sympy import solveset
>>> from sympy.abc import x, y
>>> solveset(x**2 - y, x)
{-sqrt(y), sqrt(y)}
```

Here are recommendations on when to use:

- `solve()` (page 836)
 - You want to get explicit symbolic representations of the different values a variable could take that would satisfy the equation.
 - You want to substitute those explicit solution values into other equations or expressions involving the same variable using `subs()` (page 941)
- `solveset()` (page 866)
 - You want to represent the solutions in a mathematically precise way, using *mathematical sets* (page 1185).
 - You want a representation of all the solutions, including if there are infinitely many.
 - You want a consistent input interface.
 - You want to limit the domain of the solutions to any arbitrary set.
 - You do not need to programmatically extract solutions from the solution set: solution sets cannot necessarily be interrogated programmatically.

Guidance

Include the variable to be solved for in the function call

We recommend you include the variable to be solved for as the second argument for either function. While this is optional for equations with a single symbol, it is a good practice because it ensures SymPy will solve for the desired symbol. For example, you may expect the following to solve for x , and SymPy will solve for y :

```
>>> from sympy.abc import x, y
>>> from sympy import solve
>>> solve(x**2 - y, dict=True)
[{y: x**2}]
```

Specifying the variable to solve for ensures that SymPy solves for it:

```
>>> from sympy.abc import x, y
>>> from sympy import solve
>>> solve(x**2 - y, x, dict=True)
[{x: -sqrt(y)}, {x: sqrt(y)}]
```

Ensure consistent formatting from solve() by using dict=True

`solve()` (page 836) produces various output formats depending on the answer, unless you use `dict=True` to ensure the result will be formatted as a dictionary. We recommend using `dict=True`, especially if you want to extract information from the result programmatically.

Solve an equation using solve() or solveset()

You can solve an equation in several ways. The examples below demonstrate using both `solve()` (page 836) and `solveset()` (page 866) where applicable. You can choose the function best suited to your equation.

Make your equation into an expression that equals zero

Use the fact that any expression not in an Eq (equation) is automatically assumed to equal zero (0) by the solving functions. You can rearrange the equation $x^2 = y$ to $x^2 - y = 0$, and solve that expression. This approach is convenient if you are interactively solving an expression which already equals zero, or an equation that you do not mind rearranging to $expression = 0$.

```
>>> from sympy import solve, solveset
>>> from sympy.abc import x, y
>>> solve(x**2 - y, x, dict=True)
[{x: -sqrt(y)}, {x: sqrt(y)}]
>>> solveset(x**2 - y, x)
{-sqrt(y), sqrt(y)}
```

Put your equation into Eq form

Put your equation into Eq form, then solve the Eq. This approach is convenient if you are interactively solving an equation which you already have in the form of an equation, or which you think of as an equality.

```
>>> from sympy import Eq, solve, solveset
>>> from sympy.abc import x, y
>>> eqn = Eq(x**2, y)
>>> eqn
Eq(x**2, y)
>>> solutions = solve(eqn, x, dict=True)
>>> print(solutions)
[{x: -sqrt(y)}, {x: sqrt(y)}]
>>> solutions_set = solveset(eqn, x)
>>> print(solutions_set)
{-sqrt(y), sqrt(y)}
>>> for solution_set in solutions_set:
...     print(solution_set)
sqrt(y)
-sqrt(y)
```

Restrict the domain of solutions

By default, SymPy will return solutions in the complex domain, which also includes purely real and imaginary values. Here, the first two solutions are real, and the last two are imaginary:

```
>>> from sympy import Symbol, solve, solveset
>>> x = Symbol('x')
>>> solve(x**4 - 256, x, dict=True)
[{x: -4}, {x: 4}, {x: -4*I}, {x: 4*I}]
>>> solveset(x**4 - 256, x)
{-4, 4, -4*I, 4*I}
```

To restrict returned solutions to real numbers, or another domain or range, the different solving functions use different methods.

For `solve()` (page 836), place an assumption on the symbol to be solved for, x

```
>>> from sympy import Symbol, solve
>>> x = Symbol('x', real=True)
>>> solve(x**4 - 256, x, dict=True)
[{x: -4}, {x: 4}]
```

or restrict the solutions with standard Python techniques for filtering a list such as a list comprehension:

```
>>> from sympy import Or, Symbol, solve
>>> x = Symbol('x', real=True)
>>> expr = (x-4)*(x-3)*(x-2)*(x-1)
>>> solution = solve(expr, x)
>>> print(solution)
[1, 2, 3, 4]
>>> solution_outside_2_3 = [v for v in solution if (v.is_real and Or(v<2,v>
→ 3))]
>>> print(solution_outside_2_3)
[1, 4]
```

For `solveset()` (page 866), limit the output domain in the function call by setting a domain

```
>>> from sympy import S, solveset
>>> from sympy.abc import x
>>> solveset(x**4 - 256, x, domain=S.Reals)
{-4, 4}
```

or by restricting returned solutions to any arbitrary set, including an interval:

```
>>> from sympy import Interval, pi, sin, solveset
>>> from sympy.abc import x
>>> solveset(sin(x), x, Interval(-pi, pi))
{0, -pi, pi}
```

and if you restrict the solutions to a domain in which there are no solutions, `solveset()` (page 866) will return the empty set, `EmptySet` (page 1185):

```
>>> from sympy import solveset, S
>>> from sympy.abc import x
```

(continues on next page)

(continued from previous page)

```
>>> solveset(x**2 + 1, x, domain=S.Reals)
EmptySet
```

Explicitly represent infinite sets of possible solutions using solveset()

`solveset()` (page 866) *can represent infinite sets of possible solutions* (page 859) and express them in standard mathematical notation, for example $\sin(x) = 0$ for $x = n * \pi$ for every integer value of n :

```
>>> from sympy import pprint, sin, solveset
>>> from sympy.abc import x
>>> solution = solveset(sin(x), x)
>>> pprint(solution)
{2*n*pi | n in Integers} U {2*n*pi + pi | n in Integers}
```

However, `solve()` (page 836) will return only a finite number of solutions:

```
>>> from sympy import sin, solve
>>> from sympy.calculus.util import periodicity
>>> from sympy.abc import x
>>> f = sin(x)
>>> solve(f, x)
[0, pi]
>>> periodicity(f, x)
2*pi
```

`solve()` (page 836) tries to return just enough solutions so that all (infinitely many) solutions can be generated from the returned solutions by adding integer multiples of the *periodicity()* (page 249) of the equation, here 2π .

Use the solution result

Substitute solutions from solve() into an expression

You can substitute solutions from `solve()` (page 836) into an expression.

A common use case is finding the critical points and values for a function f . At the critical points, the *Derivative* (page 1042) equals zero (or is undefined). You can then obtain the function values at those critical points by substituting the critical points back into the function using `subs()` (page 941). You can also tell if the critical point is a maxima or minima by substituting the values into the expression for the second derivative: a negative value indicates a maximum, and a positive value indicates a minimum.

```
>>> from sympy.abc import x
>>> from sympy import solve, diff
>>> f = x**3 + x**2 - x
>>> derivative = diff(f, x)
>>> critical_points = solve(derivative, x, dict=True)
>>> print(critical_points)
[{x: -1}, {x: 1/3}]
```

(continues on next page)

(continued from previous page)

```

>>> point1, point2 = critical_points
>>> print(f.subs(point1))
1
>>> print(f.subs(point2))
-5/27
>>> curvature = diff(f, x, 2)
>>> print(curvature.subs(point1))
-4
>>> print(curvature.subs(point2))
4

```

solveset() solution sets cannot necessarily be interrogated programmatically

If `solveset()` (page 866) returns a finite set (class `FiniteSet` (page 1197)), you can iterate through the solutions:

```

>>> from sympy import solveset
>>> from sympy.abc import x, y
>>> solution_set = solveset(x**2 - y, x)
>>> print(solution_set)
{-sqrt(y), sqrt(y)}
>>> solution_list = list(solution_set)
>>> print(solution_list)
[sqrt(y), -sqrt(y)]

```

However, for more complex results, it may not be possible to list the solutions:

```

>>> from sympy import S, solveset, symbols
>>> x, y = symbols('x, y')
>>> solution_set = solveset(x**2 - y, x, domain=S.Reals)
>>> print(solution_set)
Intersection({-sqrt(y), sqrt(y)}, Reals)
>>> list(solution_set)
Traceback (most recent call last):
...
TypeError: The computation had not completed because of the undecidable set
membership is found in every candidates.

```

In this case, it is because, if y is negative, its square root would be imaginary rather than real and therefore outside the declared domain of the solution set. By declaring y to be real and positive, SymPy can determine that its square root is real, and thus resolve the intersection between the solutions and the set of real numbers:

```

>>> from sympy import S, Symbol, solveset
>>> x = Symbol('x')
>>> y = Symbol('y', real=True, positive=True)
>>> solution_set = solveset(x**2 - y, x, domain=S.Reals)
>>> print(solution_set)
{-sqrt(y), sqrt(y)}
>>> list(solution_set)
[sqrt(y), -sqrt(y)]

```

Alternatively, you can extract the sets from the solution set using [args](#) (page 928), then create a list from the set containing the symbolic solutions:

```
>>> from sympy import S, solveset, symbols
>>> x, y = symbols('x, y')
>>> solution_set = solveset(x**2 - y, x, domain=S.Reals)
>>> print(solution_set)
Intersection({-sqrt(y), sqrt(y)}, Reals)
>>> solution_set_args = solution_set.args
>>> print(solution_set_args)
(Reals, {-sqrt(y), sqrt(y)})
>>> list(solution_set_args[1])
[sqrt(y), -sqrt(y)]
```

Options that can speed up solve()

Include solutions making any denominator zero by using check=False

Normally, [solve\(\)](#) (page 836) checks whether any solutions make any denominator zero, and automatically excludes them. If you want to include those solutions, and speed up [solve\(\)](#) (page 836) (at the risk of obtaining invalid solutions), set `check=False`:

```
>>> from sympy import Symbol, sin, solve
>>> x = Symbol("x")
>>> solve(sin(x)/x) # 0 is excluded
[pi]
>>> solve(sin(x)/x, check=False) # 0 is not excluded
[0, pi]
```

Do not simplify solutions by using simplify=False

Normally, [solve\(\)](#) (page 836) simplifies all but polynomials of order 3 or greater before returning them and (if `check` is not `False`) uses the general [simplify](#) (page 661) function on the solutions and the expression obtained when they are substituted into the function which should be zero. If you do not want the solutions simplified, and want to speed up [solve\(\)](#) (page 836), use `simplify=False`.

```
>>> from sympy import solve
>>> from sympy.abc import x, y
>>> expr = x**2 - (y**5 - 3*y**3 + y**2 - 3)
>>> solve(expr, x, dict=True)
[{x: -sqrt(y**5 - 3*y**3 + y**2 - 3)}, {x: sqrt(y**5 - 3*y**3 + y**2 - 3)}]
>>> solve(expr, x, dict=True, simplify=False)
[{x: -sqrt((y + 1)*(y**2 - 3)*(y**2 - y + 1))}, {x: sqrt((y + 1)*(y**2 -
↪ 3)*(y**2 - y + 1))}]
```

Not all equations can be solved

Equations with no solution

Some equations have no solution, in which case SymPy may return an empty set. For example, the equation $x - 7 = x + 2$ reduces to $-7 = 2$, which has no solution because no value of x will make it true:

```
>>> from sympy import solve, Eq
>>> from sympy.abc import x
>>> eqn = Eq(x - 7, x + 2)
>>> solve(eqn, x)
[]
```

So if SymPy returns an empty list, you may want to check whether there is a mistake in the equation.

Equations which have an analytical solution, and SymPy cannot solve

It is also possible that there is an algebraic solution to your equation, and SymPy has not implemented an appropriate algorithm. If that happens, or SymPy returns an empty set or list when there is a mathematical solution (indicating a bug in SymPy), please post it on the [mailing list](#), or open an issue on [SymPy's GitHub page](#). Until the issue is resolved, you can *solve your equation numerically* (page 848) instead.

3.5 Citing SymPy

To cite SymPy in publications use

```
Meurer A, Smith CP, Paprocki M, Čertík O, Kirpichev SB, Rocklin M, Kumar A,
Ivanov S, Moore JK, Singh S, Rathnayake T, Vig S, Granger BE, Muller RP,
Bonazzi F, Gupta H, Vats S, Johansson F, Pedregosa F, Curry MJ, Terrel AR,
Roučka Š, Saboo A, Fernando I, Kulal S, Cimrman R, Scopatz A. (2017) SymPy:
symbolic computing in Python. *PeerJ Computer Science* 3:e103
https://doi.org/10.7717/peerj-cs.103
```

A BibTeX entry for LaTeX users is

```
@article{10.7717/peerj-cs.103,
  title = {SymPy: symbolic computing in Python},
  author = {Meurer, Aaron and Smith, Christopher P. and Paprocki, Mateusz
    and Čertík, Ondřej and Kirpichev, Sergey B. and Rocklin,
    Matthew and Kumar, AMIT and Ivanov, Sergiu and Moore, Jason K. and Singh,
    Sartaj and Rathnayake, Thilina and Vig, Sean and Granger, Brian E. and
    Muller, Richard P. and Bonazzi, Francesco and Gupta, Harsh and Vats, Shivam
    and Johansson, Fredrik and Pedregosa, Fabian and Curry, Matthew J. and
    Terrel, Andy R. and Roučka, Štěpán and Saboo, Ashutosh and
    Fernando, Isuru and Kulal, Sumith and Cimrman, Robert and Scopatz, Anthony},
  year = 2017,
  month = jan,
```

(continues on next page)

(continued from previous page)

```
keywords = {Python, Computer algebra system, Symbolics},
abstract = {
    SymPy is an open source computer algebra system written in
    ↪ pure Python. It is built with a focus on extensibility and ease of use,
    ↪ through both interactive and programmatic applications. These
    ↪ characteristics have led SymPy to become a popular symbolic library for the
    ↪ scientific Python ecosystem. This paper presents the architecture of SymPy,
    ↪ a description of its features, and a discussion of select submodules. The
    ↪ supplementary material provide additional examples and further outline
    ↪ details of the architecture and features of SymPy.
    },
    volume = 3,
    pages = {e103},
    journal = {PeerJ Computer Science},
    issn = {2376-5992},
    url = {https://doi.org/10.7717/peerj-cs.103},
    doi = {10.7717/peerj-cs.103}
}
```

SymPy is [BSD licensed](#), so you are free to use it whatever you like, be it academic, commercial, creating forks or derivatives, as long as you copy the BSD statement if you redistribute it (see the LICENSE file for details). That said, although not required by the SymPy license, if it is convenient for you, please cite SymPy when using it in your work and also consider contributing all your changes back, so that we can incorporate it and all of us will benefit in the end.

The SymPy development team members are listed in the [AUTHORS file on GitHub](#).

A list of [papers citing SymPy](#) can be found on [Zotero](#).

EXPLANATIONS

Explanations provide in-depth discussions about select SymPy features. These topic guides talk about things like the motivation behind design decisions, technical implementation details, and opinionated recommendations.

Content

4.1 Gotchas and Pitfalls

4.1.1 Introduction

SymPy runs under the [Python Programming Language](#), so there are some things that may behave differently than they do in other, independent computer algebra systems like Maple or Mathematica. These are some of the gotchas and pitfalls that you may encounter when using SymPy. See also the [FAQ](#), the [introductory tutorial](#) (page 5), the remainder of the SymPy Docs, and the [official Python Tutorial](#).

If you are already familiar with C or Java, you might also want to look at this [4 minute Python tutorial](#).

Ignore `#doctest: +SKIP` in the examples. That has to do with internal testing of the examples.

4.1.2 Equals Signs (=)

Single Equals Sign

The equals sign (=) is the assignment operator, not equality. If you want to do $x = y$, use `Eq(x, y)` for equality. Alternatively, all expressions are assumed to equal zero, so you can just subtract one side and use `x - y`.

The proper use of the equals sign is to assign expressions to variables.

For example:

```
>>> from sympy.abc import x, y
>>> a = x - y
>>> print(a)
x - y
```

Double Equals Signs

Double equals signs (==) are used to test equality. However, this tests expressions exactly, not symbolically. For example:

```
>>> (x + 1)**2 == x**2 + 2*x + 1
False
>>> (x + 1)**2 == (x + 1)**2
True
```

If you want to test for symbolic equality, one way is to subtract one expression from the other and run it through functions like [expand\(\)](#) (page 1053), [simplify\(\)](#) (page 661), and [trigsimp\(\)](#) (page 679) and see if the equation reduces to 0.

```
>>> from sympy import simplify, cos, sin, expand
>>> simplify((x + 1)**2 - (x**2 + 2*x + 1))
0
>>> eq = sin(2*x) - 2*sin(x)*cos(x)
>>> simplify(eq)
0
>>> expand(eq, trig=True)
0
```

Note: See also [Why does SymPy say that two equal expressions are unequal?](#) in the FAQ.

4.1.3 Variables

Variables Assignment does not Create a Relation Between Expressions

When you use = to do assignment, remember that in Python, as in most programming languages, the variable does not change if you change the value you assigned to it. The equations you are typing use the values present at the time of creation to “fill in” values, just like regular Python definitions. They are not altered by changes made afterwards. Consider the following:

```
>>> from sympy import Symbol
>>> a = Symbol('a') # Symbol, 'a', stored as variable "a"
>>> b = a + 1       # an expression involving 'a' stored as variable "b"
>>> print(b)
a + 1
>>> a = 4           # "a" now points to literal integer 4, not Symbol('a')
>>> print(a)
4
>>> print(b)       # "b" is still pointing at the expression involving 'a'
a + 1
```

Changing quantity a does not change b; you are not working with a set of simultaneous equations. It might be helpful to remember that the string that gets printed when you print a variable referring to a SymPy object is the string that was given to it when it was created; that string does not have to be the same as the variable that you assign it to.

```

>>> from sympy import var
>>> r, t, d = var('rate time short_life')
>>> d = r*t
>>> print(d)
rate*time
>>> r = 80
>>> t = 2
>>> print(d)           # We haven't changed d, only r and t
rate*time
>>> d = r*t
>>> print(d)           # Now d is using the current values of r and t
160

```

If you need variables that have dependence on each other, you can define functions. Use the `def` operator. Indent the body of the function. See the Python docs for more information on defining functions.

```

>>> c, d = var('c d')
>>> print(c)
c
>>> print(d)
d
>>> def ctimesd():
...     """
...     This function returns whatever c is times whatever d is.
...     """
...     return c*d
...
>>> ctimesd()
c*d
>>> c = 2
>>> print(c)
2
>>> ctimesd()
2*d

```

If you define a circular relationship, you will get a `RuntimeError`.

```

>>> def a():
...     return b()
...
>>> def b():
...     return a()
...
>>> a()
Traceback (most recent call last):
  File "...", line ..., in ...
    compileflags, 1) in test.globs
  File "<...>", line 1, in <module>
    a()
  File "<...>", line 2, in a
    return b()
  File "<...>", line 2, in b

```

(continues on next page)

(continued from previous page)

```
    return a()
File "<...>", line 2, in a
    return b()
...
RuntimeError: maximum recursion depth exceeded
```

Note: See also [Why doesn't changing one variable change another that depends on it?](#) in the FAQ.

Symbols

Symbols are variables, and like all other variables, they need to be assigned before you can use them. For example:

```
>>> import sympy
>>> z**2 # z is not defined yet
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'z' is not defined
>>> sympy.var('z') # This is the easiest way to define z as a standard symbol
z
>>> z**2
z**2
```

If you use **isympy**, it runs the following commands for you, giving you some default Symbols and Functions.

```
>>> from __future__ import division
>>> from sympy import *
>>> x, y, z, t = symbols('x y z t')
>>> k, m, n = symbols('k m n', integer=True)
>>> f, g, h = symbols('f g h', cls=Function)
```

You can also import common symbol names from [sympy.abc](#) (page 884).

```
>>> from sympy.abc import w
>>> w
w
>>> import sympy
>>> dir(sympy.abc)
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
 'P', 'Q', 'R', 'S', 'Symbol', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z',
 '_builtins_', '_doc_', '_file_', '_name_', '_package_', '_greek',
 '_latin_', 'a', 'alpha', 'b', 'beta', 'c', 'chi', 'd', 'delta', 'e',
 'epsilon', 'eta', 'f', 'g', 'gamma', 'h', 'i', 'iota', 'j', 'k', 'kappa',
 'l', 'm', 'mu', 'n', 'nu', 'o', 'omega', 'omicron', 'p', 'phi', 'pi',
 'psi', 'q', 'r', 'rho', 's', 'sigma', 't', 'tau', 'theta', 'u', 'upsilon',
 'v', 'w', 'x', 'xi', 'y', 'z', 'zeta']
```


If you want control over the assumptions of the variables, use [Symbol](#) (page 976) and [symbols\(\)](#) (page 978). See [Keyword Arguments](#) (page 153) below.

Lastly, it is recommended that you not use *I* (page 1000), *E* (page 2961), *S* (page 1845), *N* (page 1068), *C*, *O* (page 627), or *Q* (page 191) for variable or symbol names, as those are used for the imaginary unit (*i*), the base of the natural logarithm (*e*), the [sympify\(\)](#) (page 918) function (see [Symbolic Expressions](#) (page 146) below), numeric evaluation ([N\(\)](#) (page 1068) is equivalent to [evalf\(\)](#) (page 1095)), the big *O* order symbol (as in $O(n \log n)$), and the assumptions object that holds a list of supported ask keys (such as `Q.real`), respectively. You can use the mnemonic OSINEQ to remember what Symbols are defined by default in SymPy. Or better yet, always use lowercase letters for Symbol names. Python will not prevent you from overriding default SymPy names or functions, so be careful.

```
>>> cos(pi) # cos and pi are a built-in sympy names.
-1
>>> pi = 3 # Notice that there is no warning for overriding pi.
>>> cos(pi)
cos(3)
>>> def cos(x): # No warning for overriding built-in functions either.
...     return 5*x
...
>>> cos(pi)
15
>>> from sympy import cos # reimport to restore normal behavior
```

To get a full list of all default names in SymPy do:

```
>>> import sympy
>>> dir(sympy)
# A big list of all default sympy names and functions follows.
# Ignore everything that starts and ends with __.
```

If you have [IPython](#) installed and use **isympy**, you can also press the TAB key to get a list of all built-in names and to autocomplete. Also, see [this page](#) for a trick for getting tab completion in the regular Python console.

Note: See also [What is the best way to create symbols?](#) in the FAQ.

Functions

A function like $f(x)$ can be created by defining the Function and the variable:

```
>>> from sympy import Function
>>> f = Function('f')
>>> x = Symbol('x')
>>> f(x)
f(x)
```

If you assign $f(x)$ to a Python variable *f* you will lose your ability to copy and paste that function or to create a function with a different argument: `Function('f')` is callable, but `Function('f')(x)` is not:

```
>>> f1 = Function('f1')
>>> f2 = Function('f2')('x')
>>> f1
f1
>>> f2
f2(x)
>>> f1(1)
f1(1)
>>> f2(1)
Traceback (most recent call last):
...
TypeError: 'f2' object is not callable
>>> f2.subs(x, 1)
f2(1)
```

4.1.4 Symbolic Expressions

Python numbers vs. SymPy Numbers

SymPy uses its own classes for integers, rational numbers, and floating point numbers instead of the default Python `int` and `float` types because it allows for more control. But you have to be careful. If you type an expression that just has numbers in it, it will default to a Python expression. Use the `sympify()` (page 918) function, or just `S` (page 1845), to ensure that something is a SymPy expression.

```
>>> 6.2 # Python float. Notice the floating point accuracy problems.
6.2000000000000002
>>> type(6.2) # <type 'float'> in Python 2.x, <class 'float'> in Py3k
<... 'float'>
>>> S(6.2) # SymPy Float has no such problems because of arbitrary precision.
6.2000000000000000
>>> type(S(6.2))
<class 'sympy.core.numbers.Float'>
```

If you include numbers in a SymPy expression, they will be sympified automatically, but there is one gotcha you should be aware of. If you do `<number>/<number>` inside of a SymPy expression, Python will evaluate the two numbers before SymPy has a chance to get to them. The solution is to `sympify()` (page 918) one of the numbers, or use `Rational` (page 985).

```
>>> x**(1/2) # evaluates to x**0 or x**0.5
x**0.5
>>> x**(S(1)/2) # sympify one of the ints
sqrt(x)
>>> x**Rational(1, 2) # use the Rational class
sqrt(x)
```

With a power of `1/2` you can also use `sqrt` shorthand:

```
>>> sqrt(x) == x**Rational(1, 2)
True
```

If the two integers are not directly separated by a division sign then you don't have to worry about this problem:

```
>>> x**(2*x/3)
x**(2*x/3)
```

Note: A common mistake is copying an expression that is printed and reusing it. If the expression has a *Rational* (page 985) (i.e., <number>/<number>) in it, you will not get the same result, obtaining the Python result for the division rather than a SymPy Rational.

```
>>> x = Symbol('x')
>>> print(solve(7*x - 22, x))
[22/7]
>>> 22/7 # If we just copy and paste we get int 3 or a float
3.142857142857143
>>> # One solution is to just assign the expression to a variable
>>> # if we need to use it again.
>>> a = solve(7*x - 22, x)[0]
>>> a
22/7
```

The other solution is to put quotes around the expression and run it through S() (i.e., sympify it):

```
>>> S("22/7")
22/7
```

Also, if you do not use `isympy`, you could use `from __future__ import division` to prevent the / sign from performing integer division.

```
>>> from __future__ import division
>>> 1/2 # With division imported it evaluates to a python float
0.5
>>> 1//2 # You can still achieve integer division with //
0
```

But be careful: you will now receive floats where you might have desired a Rational:

```
>>> x**(1/2)
x**0.5
```

Rational (page 985) only works for number/number and is only meant for rational numbers. If you want a fraction with symbols or expressions in it, just use /. If you do number/expression or expression/number, then the number will automatically be converted into a SymPy Number. You only need to be careful with number/number.

```
>>> Rational(2, x)
Traceback (most recent call last):
...
TypeError: invalid input: x
>>> 2/x
2/x
```

Evaluating Expressions with Floats and Rationals

SymPy keeps track of the precision of Float objects. The default precision is 15 digits. When an expression involving a Float is evaluated, the result will be expressed to 15 digits of precision but those digits (depending on the numbers involved with the calculation) may not all be significant.

The first issue to keep in mind is how the Float is created: it is created with a value and a precision. The precision indicates how precise of a value to use when that Float (or an expression it appears in) is evaluated.

The values can be given as strings, integers, floats, or rationals.

- strings and integers are interpreted as exact

```
>>> Float(100)
100.00000000000000
>>> Float('100', 5)
100.00
```

- to have the precision match the number of digits, the null string can be used for the precision

```
>>> Float(100, '')
100.
>>> Float('12.34')
12.340000000000000
>>> Float('12.34', '')
12.34
```

```
>>> s, r = [Float(j, 3) for j in ('0.25', Rational(1, 7))]
>>> for f in [s, r]:
...     print(f)
0.250
0.143
```

Next, notice that each of those values looks correct to 3 digits. But if we try to evaluate them to 20 digits, a difference will become apparent:

The 0.25 (with precision of 3) represents a number that has a non-repeating binary decimal; 1/7 is repeating in binary and decimal – it cannot be represented accurately too far past those first 3 digits (the correct decimal is a repeating 142857):

```
>>> s.n(20)
0.25000000000000000000
>>> r.n(20)
0.14285278320312500000
```

It is important to realize that although a Float is being displayed in decimal at arbitrary precision, it is actually stored in binary. Once the Float is created, its binary information is set at the given precision. The accuracy of that value cannot be subsequently changed; so 1/7, at a precision of 3 digits, can be padded with binary zeros, but these will not make it a more accurate value of 1/7.

If inexact, low-precision numbers are involved in a calculation with higher precision values, the evalf engine will increase the precision of the low precision values and inexact results will be obtained. This is feature of calculations with limited precision:

```
>>> Float('0.1', 10) + Float('0.1', 3)
0.2000061035
```

Although the evalf engine tried to maintain 10 digits of precision (since that was the highest precision represented) the 3-digit precision used limits the accuracy to about 4 digits - not all the digits you see are significant. evalf doesn't try to keep track of the number of significant digits.

That very simple expression involving the addition of two numbers with different precisions will hopefully be instructive in helping you understand why more complicated expressions (like trig expressions that may not be simplified) will not evaluate to an exact zero even though, with the right simplification, they should be zero. Consider this unsimplified trig identity, multiplied by a big number:

```
>>> big = 12345678901234567890
>>> big_trig_identity = big*cos(x)**2 + big*sin(x)**2 - big*1
>>> abs(big_trig_identity.subs(x, .1).n(2)) > 1000
True
```

When the cos and sin terms were evaluated to 15 digits of precision and multiplied by the big number, they gave a large number that was only precise to 15 digits (approximately) and when the 20 digit big number was subtracted the result was not zero.

There are three things that will help you obtain more precise numerical values for expressions:

- 1) Pass the desired substitutions with the call to evaluate. By doing the subs first, the Float values cannot be updated as necessary. By passing the desired substitutions with the call to evalf the ability to re-evaluate as necessary is gained and the results are impressively better:

```
>>> big_trig_identity.n(2, {x: 0.1})
-0.e-91
```

- 2) Use Rationals, not Floats. During the evaluation process, the Rational can be computed to an arbitrary precision while the Float, once created - at a default of 15 digits - cannot. Compare the value of -1.4e+3 above with the nearly zero value obtained when replacing x with a Rational representing 1/10 - before the call to evaluate:

```
>>> big_trig_identity.subs(x, S('1/10')).n(2)
0.e-91
```

- 3) Try to simplify the expression. In this case, SymPy will recognize the trig identity and simplify it to zero so you don't even have to evaluate it numerically:

```
>>> big_trig_identity.simplify()
0
```

Immutability of Expressions

Expressions in SymPy are immutable, and cannot be modified by an in-place operation. This means that a function will always return an object, and the original expression will not be modified. The following example snippet demonstrates how this works:

```
def main():
    var('x y a b')
    expr = 3*x + 4*y
    print('original =', expr)
    expr_modified = expr.subs({x: a, y: b})
    print('modified =', expr_modified)

if __name__ == "__main__":
    main()
```

The output shows that the `subs()` (page 941) function has replaced variable `x` with variable `a`, and variable `y` with variable `b`:

```
original = 3*x + 4*y
modified = 3*a + 4*b
```

The `subs()` (page 941) function does not modify the original expression `expr`. Rather, a modified copy of the expression is returned. This returned object is stored in the variable `expr_modified`. Note that unlike C/C++ and other high-level languages, Python does not require you to declare a variable before it is used.

Mathematical Operators

SymPy uses the same default operators as Python. Most of these, like `*/+-`, are standard. Aside from integer division discussed in *Python numbers vs. SymPy Numbers* (page 146) above, you should also be aware that implied multiplication is not allowed. You need to use `*` whenever you wish to multiply something. Also, to raise something to a power, use `**`, not `^` as many computer algebra systems use. Parentheses `()` change operator precedence as you would normally expect.

In **isympy**, with the **ipython** shell:

```
>>> 2x
Traceback (most recent call last):
...
SyntaxError: invalid syntax
>>> 2*x
2*x
>>> (x + 1)^2 # This is not power. Use ** instead.
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for ^: 'Add' and 'int'
>>> (x + 1)**2
(x + 1)**2
>>> pprint(3 - x**(2*x)/(x + 1))
      2*x
      x
```

(continues on next page)

(continued from previous page)

```
- ---- + 3
  x + 1
```

Inverse Trig Functions

SymPy uses different names for some functions than most computer algebra systems. In particular, the inverse trig functions use the python names of [asin](#) (page 395), [acos](#) (page 396) and so on instead of the usual `arcsin` and `arccos`. Use the methods described in [Symbols](#) (page 144) above to see the names of all SymPy functions.

Sqrt is not a Function

There is no `sqrt` function in the same way that there is an exponential function (`exp`). `sqrt(x)` is used to represent $\text{Pow}(x, S(1)/2)$ so if you want to know if an expression has any square roots in it, `expr.has(sqrt)` will not work. You must look for `Pow` with an exponent of one half (or negative one half if it is in a denominator, e.g.

```
>>> (y + sqrt(x)).find(Wild('w')**S.Half)
{sqrt(x)}
>>> (y + 1/sqrt(x)).find(Wild('w')**-S.Half)
{1/sqrt(x)}
```

If you are interested in any power of the `sqrt` then the following pattern would be appropriate

```
>>> sq = lambda s: s.is_Pow and s.exp.is_Rational and s.exp.q == 2
>>> (y + sqrt(x)**3).find(sq)
{x**(3/2)}
```

4.1.5 Special Symbols

The symbols `[]`, `{}`, `=`, and `()` have special meanings in Python, and thus in SymPy. See the Python docs linked to above for additional information.

Lists

Square brackets `[]` denote a list. A list is a container that holds any number of different objects. A list can contain anything, including items of different types. Lists are mutable, which means that you can change the elements of a list after it has been created. You access the items of a list also using square brackets, placing them after the list or list variable. Items are numbered using the space before the item.

Note: List indexes begin at 0.

Example:

```
>>> a = [x, 1]  # A simple list of two items
>>> a
[x, 1]
>>> a[0]  # This is the first item
x
>>> a[0] = 2  # You can change values of lists after they have been created
>>> print(a)
[2, 1]
>>> print(solve(x**2 + 2*x - 1, x)) # Some functions return lists
[-1 + sqrt(2), -sqrt(2) - 1]
```

Note: See the Python docs for more information on lists and the square bracket notation for accessing elements of a list.

Dictionaries

Curly brackets {} denote a dictionary, or a dict for short. A dictionary is an unordered list of non-duplicate keys and values. The syntax is {key: value}. You can access values of keys using square bracket notation.

```
>>> d = {'a': 1, 'b': 2}  # A dictionary.
>>> d
{'a': 1, 'b': 2}
>>> d['a']  # How to access items in a dict
1
>>> roots((x - 1)**2*(x - 2), x)  # Some functions return dicts
{1: 2, 2: 1}
>>> # Some SymPy functions return dictionaries. For example,
>>> # roots returns a dictionary of root:multiplicity items.
>>> roots((x - 5)**2*(x + 3), x)
{-3: 1, 5: 2}
>>> # This means that the root -3 occurs once and the root 5 occurs twice.
```

Note: See the Python docs for more information on dictionaries.

Tuples

Parentheses (), aside from changing operator precedence and their use in function calls, (like `cos(x)`), are also used for tuples. A tuple is identical to a [list](#) (page 151), except that it is not mutable. That means that you cannot change their values after they have been created. In general, you will not need tuples in SymPy, but sometimes it can be more convenient to type parentheses instead of square brackets.

```
>>> t = (1, 2, x)  # Tuples are like lists
>>> t
(1, 2, x)
>>> t[0]
```

(continues on next page)

(continued from previous page)

```

1
>>> t[0] = 4 # Except you cannot change them after they have been
↳ created
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment

```

Single element tuples, unlike lists, must have a comma in them:

```

>>> (x,)
(x,)

```

Without the comma, a single expression without a comma is not a tuple:

```

>>> (x)
x

```

integrate takes a sequence as the second argument if you want to integrate with limits (and a tuple or list will work):

```

>>> integrate(x**2, (x, 0, 1))
1/3
>>> integrate(x**2, [x, 0, 1])
1/3

```

Note: See the Python docs for more information on tuples.

Keyword Arguments

Aside from the usage described [above](#) (page 141), equals signs (=) are also used to give named arguments to functions. Any function that has key=value in its parameters list (see below on how to find this out), then key is set to value by default. You can change the value of the key by supplying your own value using the equals sign in the function call. Also, functions that have ** followed by a name in the parameters list (usually ****kwargs** or ****assumptions**) allow you to add any number of key=value pairs that you want, and they will all be evaluated according to the function.

`sqrt(x**2)` doesn't auto simplify to `x` because `x` is assumed to be complex by default, and, for example, `sqrt((-1)**2) == sqrt(1) == 1 != -1`:

```

>>> sqrt(x**2)
sqrt(x**2)

```

Giving assumptions to Symbols is an example of using the keyword argument:

```

>>> x = Symbol('x', positive=True)

```

The square root will now simplify since it knows that `x >= 0`:

```

>>> sqrt(x**2)
x

```

`powsimp` has a default argument of `combine='all'`:

```
>>> pprint(powsimp(x**n*x**m*y**n*y**m))
      m + n
(x*y)
```

Setting `combine` to the default value is the same as not setting it.

```
>>> pprint(powsimp(x**n*x**m*y**n*y**m, combine='all'))
      m + n
(x*y)
```

The non-default options are `'exp'`, which combines exponents...

```
>>> pprint(powsimp(x**n*x**m*y**n*y**m, combine='exp'))
      m + n   m + n
x          *y
```

...and `'base'`, which combines bases.

```
>>> pprint(powsimp(x**n*x**m*y**n*y**m, combine='base'))
      m       n
(x*y) * (x*y)
```

Note: See the Python docs for more information on function parameters.

4.1.6 Getting help from within SymPy

`help()`

Although all docs are available at docs.sympy.org or on the [SymPy Wiki](#), you can also get info on functions from within the Python interpreter that runs SymPy. The easiest way to do this is to do `help(function)`, or `function?` if you are using **ipython**:

```
In [1]: help(powsimp)  # help() works everywhere

In [2]: # But in ipython, you can also use ?, which is better because it
In [3]: # it gives you more information
In [4]: powsimp?
```

These will give you the function parameters and docstring for `powsimp()` (page 680). The output will look something like this:

```
sympy.simplify.simplify.powsimp(expr, deep=False, combine='all', force=False,
                                measure=<function count_ops>)
    reduces expression by combining powers with similar bases and exponents.
```

Explanation

If `deep` is `True` then `powsimp()` will also simplify arguments of functions. By default `deep` is set to `False`.

If `force` is `True` then bases will be combined without checking for assumptions, e.g. `sqrt(x)*sqrt(y) -> sqrt(x*y)` which is not true if `x` and `y` are both negative.

You can make `powsimp()` only combine bases or only combine exponents by changing `combine='base'` or `combine='exp'`. By default, `combine='all'`, which does both. `combine='base'` will only combine:

$$x^a * y^a \Rightarrow (x*y)^a \quad \text{as well as things like } 2^{2x} \Rightarrow 4^x$$

and `combine='exp'` will only combine

$$x^a * x^b \Rightarrow x^{(a+b)}$$

`combine='exp'` will strictly only combine exponents in the way that used to be automatic. Also use `deep=True` if you need the old behavior.

When `combine='all'`, `'exp'` is evaluated first. Consider the first example below for when there could be an ambiguity relating to this. This is done so things like the second example can be completely combined. If you want `'base'` combined first, do something like `powsimp(powsimp(expr, combine='base'), combine='exp')`.

Examples

```
>>> from sympy import powsimp, exp, log, symbols
>>> from sympy.abc import x, y, z, n
>>> powsimp(x**y*x**z*y**z, combine='all')
x**(y + z)*y**z
>>> powsimp(x**y*x**z*y**z, combine='exp')
x**(y + z)*y**z
>>> powsimp(x**y*x**z*y**z, combine='base', force=True)
x**y*(x*y)**z
```

```
>>> powsimp(x**z*x**y*n**z*n**y, combine='all', force=True)
(n*x)**(y + z)
>>> powsimp(x**z*x**y*n**z*n**y, combine='exp')
n**(y + z)*x**(y + z)
>>> powsimp(x**z*x**y*n**z*n**y, combine='base', force=True)
(n*x)**y*(n*x)**z
```

```
>>> x, y = symbols('x y', positive=True)
>>> powsimp(log(exp(x)*exp(y)))
log(exp(x)*exp(y))
>>> powsimp(log(exp(x)*exp(y)), deep=True)
x + y
```

Radicals with `Mul` bases will be combined if `combine='exp'`

```
>>> from sympy import sqrt
>>> x, y = symbols('x y')
```

Two radicals are automatically joined through Mul:

```
>>> a=sqrt(x*sqrt(y))
>>> a*a**3 == a**4
True
```

But if an integer power of that radical has been autoexpanded then Mul does not join the resulting factors:

```
>>> a**4 # auto expands to a Mul, no longer a Pow
x**2*y
>>> _a # so Mul doesn't combine them
x**2*y*sqrt(x*sqrt(y))
>>> powsimp(_) # but powsimp will
(x*sqrt(y))**(5/2)
>>> powsimp(x*y*a) # but won't when doing so would violate assumptions
x*y*sqrt(x*sqrt(y))
```

source()

Another useful option is the `source()` (page 2116) function. This will print the source code of a function, including any docstring that it may have. You can also do `function??` in **ipython**. For example, from SymPy 0.6.5:

```
>>> source(simplify) # simplify() is actually only 2 lines of code.
In file: ./sympy/simplify/simplify.py
def simplify(expr):
    """Naively simplifies the given expression.
    ...
    Simplification is not a well defined term and the exact strategies
    this function tries can change in the future versions of SymPy. If
    your algorithm relies on "simplification" (whatever it is), try to
    determine what you need exactly - is it powsimp()? radsimp()?
    together()?, logcombine()?, or something else? And use this particular
    function directly, because those are well defined and thus your
    ↪algorithm
    will be robust.
    ...
    """
    expr = Poly.cancel(powsimp(expr))
    return powsimp(together(expr.expand()), combine='exp', deep=True)
```

4.2 SymPy Special Topics

The purpose of this collection of documents is to provide users of SymPy with topics which are not strictly tutorial or are longer than tutorials and tests. The documents will hopefully fill a gap as SymPy matures and users find more ways to show how SymPy can be used in more advanced topics.

4.2.1 Finite Difference Approximations to Derivatives

Introduction

Finite difference approximations to derivatives is quite important in numerical analysis and in computational physics. In this tutorial we show how to use SymPy to compute approximations of varying accuracy. The hope is that these notes could be useful for the practicing researcher who is developing code in some language and needs to be able to efficiently generate finite difference formulae for various approximations.

In order to establish notation, we first state that we envision that there exists a continuous function F of a single variable x , with F having as many derivatives as desired. We sample x values uniformly at points along the real line separated by h . In most cases we want h to be small in some sense. $F(x)$ may be expanded about some point x_0 via the usual Taylor series expansion. Let $x = x_0 + h$. Then the Taylor expansion is

$$F(x_0 + h) = F(x_0) + \left(\frac{dF}{dx}\right)_{x_0} * h + \frac{1}{2!} \left(\frac{d^2F}{dx^2}\right)_{x_0} * h^2 + \frac{1}{3!} \left(\frac{d^3F}{dx^3}\right)_{x_0} * h^3 + \dots$$

In order to simplify the notation, we now define a set of coefficients c_n , where

$$c_n := \frac{1}{n!} \left(\frac{d^n F}{dx^n}\right)_{x_0}.$$

So now our series has the form:

$$F(x_0 + h) = F(x_0) + c_1 * h + c_2 * h^2 + c_3 * h^3 + \dots$$

In the following we will only use a finite grid of values x_i with i running from $1, \dots, N$ and the corresponding values of our function F at these grid points denoted by F_i . So the problem is how to generate approximate values for the derivatives of F with the constraint that we use a subset of the finite set of pairs (x_i, F_i) of size N .

What follows are manipulations using SymPy to formulate approximations for derivatives of a given order and to assess its accuracy. First, we use SymPy to derive the approximations by using a rather brute force method frequently covered in introductory treatments. Later we shall make use of other SymPy functions which get the job done with more efficiency.

A Direct Method Using SymPy Matrices

If we let $x_0 = x_i$, evaluate the series at $x_{i+1} = x_i + h$ and truncate all terms above $O(h^1)$ we can solve for the single coefficient c_1 and obtain an approximation to the first derivative:

$$\left(\frac{dF}{dx}\right)_{x_0} \approx \frac{F_{i+1} - F_i}{h} + O(h)$$

where the $O(h)$ refers to the lowest order term in the series in h . This establishes that the derivative approximation is of first order accuracy. Put another way, if we decide that we can only use the two pairs (x_i, F_i) and (x_{i+1}, F_{i+1}) we obtain a “first order accurate” derivative.

In addition to (x_i, F_i) we next use the two points (x_{i+1}, F_{i+1}) and (x_{i+2}, F_{i+2}) . Then we have two equations:

$$F_{i+1} = F_i + c_1 * h + \frac{1}{2} * c_2 * h^2 + \frac{1}{3!} * c_3 * h^3 + \dots$$

$$F_{i+2} = F_i + c_1 * (2h) + \frac{1}{2} * c_2 * (2h)^2 + \frac{1}{3!} * c_3 * (2h)^3 + \dots$$

If we again want to find the first derivative (c_1), we can do that by eliminating the term involving c_2 from the two equations. We show how to do it using SymPy.

```
>>> from __future__ import print_function
>>> from sympy import *
>>> x, x0, h = symbols('x, x_0, h')
>>> Fi, Fip1, Fip2 = symbols('F_{i}, F_{i+1}, F_{i+2}')
>>> n = 3 # there are the coefficients c_0=Fi, c_1=dF/dx, c_2=d**2F/dx**2
>>> c = symbols('c:3')
>>> def P(x, x0, c, n):
...     return sum( ((1/factorial(i))*c[i] * (x-x0)**i for i in range(n)) )
```

Vector of right hand sides:

```
>>> R = Matrix([[Fi], [Fip1], [Fip2]])
```

Now we make a matrix consisting of the coefficients of the c_i in the n th degree polynomial P . Coefficients of c_i evaluated at x_i :

```
>>> m11 = P(x0, x0, c, n).diff(c[0])
>>> m12 = P(x0, x0, c, n).diff(c[1])
>>> m13 = P(x0, x0, c, n).diff(c[2])
```

Coefficients of c_i evaluated at $x_i + h$:

```
>>> m21 = P(x0+h, x0, c, n).diff(c[0])
>>> m22 = P(x0+h, x0, c, n).diff(c[1])
>>> m23 = P(x0+h, x0, c, n).diff(c[2])
```

Coefficients of c_i evaluated at $x_i + 2 * h$:

```
>>> m31 = P(x0+2*h, x0, c, n).diff(c[0])
>>> m32 = P(x0+2*h, x0, c, n).diff(c[1])
>>> m33 = P(x0+2*h, x0, c, n).diff(c[2])
```

Matrix of the coefficients is 3x3 in this case:

```
>>> M = Matrix([[m11, m12, m13], [m21, m22, m23], [m31, m32, m33]])
```

Matrix form of the three equations for the c_i is $M * X = R$:

The solution is obtained by directly inverting the 3x3 matrix M :

```
>>> X = M.inv() * R
```

Note that all three coefficients make up the solution. The desired first derivative is coefficient c_1 which is $X[1]$.

```
>>> print(together(X[1]))
(4*F_{i+1} - F_{i+2} - 3*F_{i-1})/(2*h)
```

It is instructive to compute another three-point approximation to the first derivative, except centering the approximation at x_i and thus using points at x_{i-1} , x_i , and x_{i+1} . So here is how this can be done using the 'brute force' method:

```
>>> from __future__ import print_function
>>> from sympy import *
>>> x, x0, h = symbols('x, x_i, h')
>>> Fi, Fim1, Fip1 = symbols('F_{i}, F_{i-1}, F_{i+1}')
>>> n = 3 # there are the coefficients c_0=Fi, c_1=dF/h, c_2=d**2F/h**2
>>> c = symbols('c:3')
>>> # define a polynomial of degree n
>>> def P(x, x0, c, n):
...     return sum( ((1/factorial(i))*c[i] * (x-x0)**i for i in range(n)) )
>>> # now we make a matrix consisting of the coefficients
>>> # of the c_i in the nth degree polynomial P
>>> # coefficients of c_i evaluated at x_i
>>> m11 = P(x0, x0, c, n).diff(c[0])
>>> m12 = P(x0, x0, c, n).diff(c[1])
>>> m13 = P(x0, x0, c, n).diff(c[2])
>>> # coefficients of c_i evaluated at x_i - h
>>> m21 = P(x0-h, x0, c, n).diff(c[0])
>>> m22 = P(x0-h, x0, c, n).diff(c[1])
>>> m23 = P(x0-h, x0, c, n).diff(c[2])
>>> # coefficients of c_i evaluated at x_i + h
>>> m31 = P(x0+h, x0, c, n).diff(c[0])
>>> m32 = P(x0+h, x0, c, n).diff(c[1])
>>> m33 = P(x0+h, x0, c, n).diff(c[2])
>>> # matrix of the coefficients is 3x3 in this case
>>> M = Matrix([[m11, m12, m13], [m21, m22, m23], [m31, m32, m33]])
```

Now that we have the matrix of coefficients we next form the right-hand-side and solve by inverting M :

```
>>> # matrix of the function values...actually a vector of right hand sides
>>> R = Matrix([[Fi], [Fim1], [Fip1]])
>>> # matrix form of the three equations for the c_i is M*X = R
>>> # solution directly inverting the 3x3 matrix M:
>>> X = M.inv() * R
>>> # note that all three coefficients make up the solution
>>> # the first derivative is coefficient c_1 which is X[1].
>>> print("The second-order accurate approximation for the first derivative_
↪is: ")
The second-order accurate approximation for the first derivative is:
>>> print(together(X[1]))
(F_{i+1} - F_{i-1})/(2*h)
```

These two examples serve to show how one can directly find second order accurate first derivatives using SymPy. The first example uses values of x and F at all three points x_i , x_{i+1} , and x_{i+2} whereas the second example only uses values of x at the two points x_{i-1} and x_{i+1} and thus is a bit more efficient.

From these two simple examples a general rule is that if one wants a first derivative to be accurate to $O(h^n)$ then one needs $n+1$ function values in the approximating polynomial (here provided via the function $P(x, x_0, c, n)$).

Now let's assess the question of the accuracy of the centered difference result to see how we determine that it is really second order. To do this we take the result for dF/dx and substitute in the polynomial expansion for a higher order polynomial and see what we get. To this end, we make a set of eight coefficients d and use them to perform the check:

```
>>> d = symbols('c:8')
>>> dfdxcheck = (P(x0+h, x0, d, 8) - P(x0-h, x0, d, 8))/(2*h)
>>> print(simplify(dfdxcheck)) # so the appropriate cancellation of terms
    involving `h` happens
c1 + c3*h**2/6 + c5*h**4/120 + c7*h**6/5040
```

Thus we see that indeed the derivative is c_1 with the next term in the series of order h^2 .

However, it can quickly become rather tedious to generalize the direct method as presented above when attempting to generate a derivative approximation to high order, such as 6 or 8 although the method certainly works and using the present method is certainly less tedious than performing the calculations by hand.

As we have seen in the discussion above, the simple centered approximation for the first derivative only uses two point values of the (x_i, F_i) pairs. This works fine until one encounters the last point in the domain, say at $i = N$. Since our centered derivative approximation would use data at the point (x_{N+1}, F_{N+1}) we see that the derivative formula will not work. So, what to do? Well, a simple way to handle this is to devise a different formula for this last point which uses points for which we do have values. This is the so-called backward difference formula. To obtain it, we can use the same direct approach, except now use the three points (x_N, F_N) , (x_{N-1}, F_{N-1}) , and (x_{N-2}, F_{N-2}) and center the approximation at (x_N, F_N) . Here is how it can be done using SymPy:

```
>>> from __future__ import print_function
>>> from sympy import *
>>> x, xN, h = symbols('x, x_N, h')
>>> FN, FNm1, FNm2 = symbols('F_{N}, F_{N-1}, F_{N-2}')
>>> n = 8 # there are the coefficients c_0=Fi, c_1=dF/h, c_2=d**2F/h**2
>>> c = symbols('c:8')
>>> # define a polynomial of degree d
>>> def P(x, x0, c, n):
...     return sum( ((1/factorial(i))*c[i] * (x-x0)**i for i in range(n)) )
```

Now we make a matrix consisting of the coefficients of the c_i in the d th degree polynomial P coefficients of c_i evaluated at x_i , x_{i-1} , and x_{i+1} :

```
>>> m11 = P(xN, xN, c, n).diff(c[0])
>>> m12 = P(xN, xN, c, n).diff(c[1])
>>> m13 = P(xN, xN, c, n).diff(c[2])
>>> # coefficients of c_i evaluated at x_i - h
>>> m21 = P(xN-h, xN, c, n).diff(c[0])
>>> m22 = P(xN-h, xN, c, n).diff(c[1])
```

(continues on next page)

(continued from previous page)

```
>>> m23 = P(xN-h, xN, c, n).diff(c[2])
>>> # coefficients of c_i evaluated at x_i + h
>>> m31 = P(xN-2*h, xN, c, n).diff(c[0])
>>> m32 = P(xN-2*h, xN, c, n).diff(c[1])
>>> m33 = P(xN-2*h, xN, c, n).diff(c[2])
```

Next we construct the 3×3 matrix of the coefficients:

```
>>> M = Matrix([[m11, m12, m13], [m21, m22, m23], [m31, m32, m33]])
>>> # matrix of the function values...actually a vector of right hand sides
>>> R = Matrix([[FN], [FNm1], [FNm2]])
```

Then we invert M and write the solution to the 3×3 system.

The matrix form of the three equations for the c_i is $M * C = R$. The solution is obtained by directly inverting M :

```
>>> X = M.inv() * R
```

The first derivative is coefficient c_1 which is $X[1]$. Thus the second order accurate approximation for the first derivative is:

```
>>> print("The first derivative centered at the last point on the right is:")
The first derivative centered at the last point on the right is:
>>> print(together(X[1]))
(-4*F_{N-1} + F_{N-2} + 3*F_{N})/(2*h)
```

Of course, we can devise a similar formula for the value of the derivative at the left end of the set of points at (x_1, F_1) in terms of values at (x_2, F_2) and (x_3, F_3) .

Also, we note that output of formats appropriate to Fortran, C, etc. may be done in the examples given above.

Next we show how to perform these and many other discretizations of derivatives, but using a much more efficient approach originally due to Bengt Fornberg and now incorporated into SymPy.

Finite differences (page 46)

Finite difference weights (page 240)

4.2.2 Classification of SymPy objects

There are several ways of how SymPy object is classified.

class

Like any other object in Python, SymPy expression is an instance of class. You can get the class of the object with built-in `type()` function, and check it with `isinstance()` function.

```
>>> from sympy import Add
>>> from sympy.abc import x,y
>>> type(x + y)
<class 'sympy.core.add.Add'>
>>> isinstance(x + y, Add)
True
```

Classes represent only the programmatic structures of the objects, and does not distinguish the mathematical difference between them. For example, the integral of number and the integral of matrix both have the class *Integral*, although the former is number and the latter is matrix.

```
>>> from sympy import MatrixSymbol, Integral
>>> A = MatrixSymbol('A', 2, 2)
>>> type(Integral(1, x))
<class 'sympy.integrals.integrals.Integral'>
>>> type(Integral(A, x))
<class 'sympy.integrals.integrals.Integral'>
```

kind

Kind indicates what mathematical object does the expression represent. You can retrieve the kind of expression with `.kind` property.

```
>>> Integral(1, x).kind
NumberKind
>>> Integral(A, x).kind
MatrixKind(NumberKind)
```

This result shows that $\text{Integral}(1, x)$ is number, and $\text{Integral}(A, x)$ is matrix with number element.

Since the class cannot guarantee to catch this difference, kind of the object is very important. For example, if you are building a function or class that is designed to work only for numbers, you should consider filtering the arguments with *NumberKind* so that the user does not naively pass unsupported objects such as $\text{Integral}(A, x)$.

For the performance, set theory is not implemented in kind system. For example,

NumberKind does not distinguish the real number and complex number.

```
>>> from sympy import pi, I
>>> pi.kind
NumberKind
>>> I.kind
NumberKind
```

SymPy's *Set* and *kind* are not compatible.

```
>>> from sympy import S
>>> from sympy.core.kind import NumberKind
>>> S.Reals.is_subset(S.Complexes)
True
>>> S.Reals.is_subset(NumberKind)
Traceback (most recent call last):
...
ValueError: Unknown argument 'NumberKind'
```

sets and assumptions

If you want to classify the object in strictly mathematical way, you may need SymPy's sets and assumptions.

```
>>> from sympy import ask, Q
>>> S.One in S.Reals
True
>>> ask(Q.even(2*x), Q.odd(x))
True
```

See *assumptions* module and *sets* module for more information.

func

func is the head of the object, and it is used to recurse over the expression tree.

```
>>> Add(x + y).func
<class 'sympy.core.add.Add'>
>>> Add(x + x).func
<class 'sympy.core.mul.Mul'>
>>> Q.even(x).func
<class 'sympy.assumptions.assume.AppliedPredicate'>
```

As you can see, resulting head may be a class or another SymPy object. Keep this in mind when you classify the object with this attribute. See [Advanced Expression Manipulation](#) (page 61) for detailed information.

4.3 List of active deprecations

This page lists all active deprecations in the SymPy codebase. See the [Deprecation Policy](#) (page 3017) page for a description of SymPy's deprecation policy, as well as instructions for contributors on how to deprecate things.

In particular, the deprecation policy for SymPy is for deprecations to last at least **1 year** after the first major release that includes the deprecation. After that period, the deprecated functionality may be removed from SymPy, and code will need to be updated to use the replacement feature to continue working.

During the deprecation period, a `SymPyDeprecationWarning` message will be printed whenever the deprecated functionality is used. It is recommended for users to update their code so that it does not use deprecated functionality, as described below for each given deprecation.

4.3.1 Silencing SymPy Deprecation Warnings

To silence SymPy deprecation warnings, add a filter using the `warnings` module. For example:

```
import warnings
from sympy.utilities.exceptions import SymPyDeprecationWarning

warnings.filterwarnings(
    # replace "ignore" with "error" to make the warning raise an exception.
    # This is useful if you want to test you aren't using deprecated code.
    "ignore",

    # message may be omitted to filter all SymPyDeprecationWarnings
    message=r"(?s).*<regex matching the warning message>",

    category=SymPyDeprecationWarning,
    module=r"<regex matching your module>"
)
```

Here `(?s).*<regex matching the warning message>` is a regular expression matching the warning message. For example, to filter a warning about `sympy.printing`, you might use `message=r"(?s).*sympy\.printing"`. The leading `(?s).*` is there because the `warnings` module matches `message` against the start of the warning message, and because typical warning messages span multiple lines.

`<regex matching your module>` should be a regular expression matching your module that uses the deprecated code. It is recommended to include this so that you don't also silence the same warning for unrelated modules.

This same pattern may be used to instead turn `SymPyDeprecationWarning` into an error so that you can test that you aren't using deprecated code. To do this, replace `"ignore"` with `"error"` in the above example. You may also omit `message` to make this apply to all `SymPyDeprecationWarning` warnings.

If you are using `pytest`, you can use the [pytest warnings filtering capabilities](#) to either ignore `SymPyDeprecationWarning` or turn them into errors.

Note: The Python `-W` flag and `PYTHONWARNINGS` environment variable will NOT work to filter SymPy deprecation warnings (see [this blog post](#) by Ned Batchelder and [this SymPy issue](#) for details on why). You will need to either add a `warnings` filter as above or use `pytest` to filter SymPy deprecation warnings.

4.3.2 Version 1.11

New Mathematica code parser

The old mathematica code parser defined in the module `sympy.parsing.mathematica` in the function `mathematica` is deprecated. The function `parse_mathematica` with a new and more comprehensive parser should be used instead.

The `additional_translations` parameter for the Mathematica parser is not available in `parse_mathematica`. Additional translation rules to convert Mathematica expressions into SymPy ones should be specified after the conversion using SymPy's `.replace()` or `.subs()` methods on the output expression. If the translator fails to recognize the logical meaning of a Mathematica expression, a form similar to Mathematica's full form will be returned, using SymPy's `Function` object to encode the nodes of the syntax tree.

For example, suppose you want `F` to be a function that returns the maximum value multiplied by the minimum value, the previous way to specify this conversion was:

```
>>> from sympy.parsing.mathematica import mathematica
>>> mathematica('F[7,5,3]', {'F[*x]': 'Max(*x)*Min(*x)'})
21
```

Now you can do the same with

```
>>> from sympy.parsing.mathematica import parse_mathematica
>>> from sympy import Function, Max, Min
>>> parse_mathematica("F[7,5,3]").replace(Function("F"), lambda *x:
↳ Max(*x)*Min(*x))
21
```

Redundant static methods in `carmichael`

A number of static methods in `~.carmichael` are just wrappers around other functions. Instead of `carmichael.is_perfect_square` use `sympy.ntheory.primetest.is_square` and instead of `carmichael.is_prime` use `~.isprime`. Finally, `carmichael.divides` can be replaced by `instead checking`

```
n % p == 0
```

The `check` argument to `HadamardProduct`, `MatAdd` and `MatMul`

This argument can be used to pass incorrect values to `~.HadamardProduct`, `~.MatAdd`, and `~.MatMul` leading to later problems. The `check` argument will be removed and the arguments will always be checked for correctness, i.e., the arguments are matrices or matrix symbols.

4.3.3 Version 1.10

Some traversal functions have been moved

Some traversal functions have moved. Specifically, the functions

- `bottom_up`
- `interactive_traversal`
- `postorder_traversal`
- `preorder_traversal`
- `use`

have moved to different SymPy submodules.

These functions should be used from the top-level `sympy` namespace, like

```
sympy.preorder_traversal
```

or

```
from sympy import preorder_traversal
```

In general, end-users should use the top-level `sympy` namespace for any functions present there. If a name is in the top-level namespace, its specific SymPy submodule should not be relied on, as functions may move around due to internal refactorings.

`sympy.core.trace`

The `trace` object `sympy.core.trace.Tr()` was moved to `sympy.physics.quantum.trace.Tr()`. This was because it was only used in the `sympy.physics.quantum` submodule, so it was better to have it there than in the core.

The `sympy.core.compatibility` submodule

The `sympy.core.compatibility` submodule is deprecated.

This submodule was only ever intended for internal use. Now that SymPy no longer supports Python 2, this module is no longer necessary, and the remaining helper functions have been moved to more convenient places in the SymPy codebase.

Some of the functions that were in this module are available from the top-level SymPy namespace, i.e.,

```
sympy.ordered  
sympy.default_sort_key
```

or

```
from sympy import ordered, default_sort_key
```

In general, end-users should use the top-level `sympy` namespace for any functions present there. If a name is in the top-level namespace, its specific SymPy submodule should not be relied on, as functions may move around due to internal refactorings.

The remaining functions in `sympy.core.compatibility` were only intended for internal SymPy use and should not be used by user code.

Additionally, these two functions, `ordered` and `default_sort_key`, also used to be in `sympy.utilities.iterables` but have been moved from there as well.

4.3.4 Version 1.9

`expr_free_symbols`

The `expr_free_symbols` attribute of various SymPy objects is deprecated.

`expr_free_symbols` was meant to represent indexed objects such as `MatrixElement` and `Indexed` (page 1402) as free symbols. This was intended to make derivatives of free symbols work. However, this now works without making use of the method:

```
>>> from sympy import Indexed, MatrixSymbol, diff
>>> a = Indexed("A", 0)
>>> diff(a**2, a)
2*A[0]
>>> X = MatrixSymbol("X", 3, 3)
>>> diff(X[0, 0]**2, X[0, 0])
2*X[0, 0]
```

This was a general property that was added to solve a very specific problem but it added a layer of abstraction that is not necessary in general.

1. objects that have structural “non-expression” nodes already allow one to focus on the expression node if desired, e.g.

```
>>> from sympy import Derivative, symbols, Function
>>> x = symbols('x')
>>> f = Function('f')
>>> Derivative(f(x), x).expr
f(x)
```

introduction of this property encourages imprecise thinking when requesting `free_symbols` since it allows one to get symbols from a specific node of an object without specifying the node

2. the property was incorrectly added to `AtomicExpr` so numbers are returned as `expr_free_symbols`:

```
>>> S(2).expr_free_symbols
2
```

3. the application of the concept was misapplied to define `Subs.expr_free_symbols`: it added in `expr_free_symbols` of the point but the point is a `Tuple` so nothing was added
4. it was not used anywhere else in the codebase except in the context of differentiating a `Subs` object, which suggested that it was not something of general use, this is also confirmed by the fact that,

5. it was added without specific tests except for test of the derivatives of the Subs object for which it was introduced

See issue [#21494](#) for more discussion.

`sympy.stats.sample(numsamples=n)`

The `numsamples` parameter to `sympy.stats.sample()` (page 2971) is deprecated.

`numsamples` makes `sample()` return a list of size `numsamples`, like

```
>>> from sympy.stats import Die, sample
>>> X = Die('X', 6)
>>> sample(X, numsamples=3)
[3, 2, 3]
```

However, this functionality can be easily implemented by the user with a list comprehension

```
>>> [sample(X) for i in range(3)]
[5, 4, 3]
```

Additionally, it is redundant with the `size` parameter, which makes `sample` return a NumPy array with the given shape.

```
>>> sample(X, size=(3,))
array([6, 6, 1])
```

Historically, `sample` was changed in SymPy 1.7 so it returned an iterator instead of sample value. Since an iterator was returned, a `numsamples` parameter was added to specify the length of the iterator.

However, this new behavior was considered confusing, as discussed in issue [#21563](#), so it was reverted. Now, `sample_iter` should be used if a iterator is needed. Consequently, the `numsamples` parameter is no longer needed for `sample()`.

`sympy.polys.solvers.RawMatrix`

The `RawMatrix` class is deprecated. The `RawMatrix` class was a subclass of `Matrix` that used domain elements instead of `Expr` as the elements of the matrix. This breaks a key internal invariant of `Matrix` and this kind of subclassing limits improvements to the `Matrix` class.

The only part of SymPy that documented the use of the `RawMatrix` class was the Smith normal form code, and that has now been changed to use `DomainMatrix` instead. It is recommended that anyone using `RawMatrix` with the previous Smith Normal Form code should switch to using `DomainMatrix` as shown in issue [#21402](#). A better API for the Smith normal form will be added later.

Non-Expr objects in a Matrix

In SymPy 1.8 and earlier versions it was possible to put non-*Expr* (page 947) elements in a *Matrix* (page 1361) and the matrix elements could be any arbitrary Python object:

```
>>> M = Matrix([[(1, 2), {}]])
```

This is not useful and does not really work, e.g.:

```
>>> M + M
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for +: 'Dict' and 'Dict'
```

The main reason for making this possible was that there were a number of *Matrix* subclasses in the SymPy codebase that wanted to work with objects from the polys module, e.g.

1. *RawMatrix* (see *above* (page 168)) was used in `solve_lin_sys` which was part of `heurisch` and was also used by `smith_normal_form`. The *NewMatrix* class used domain elements as the elements of the *Matrix* rather than *Expr*.
2. *NewMatrix* was used in the `holonomic` module and also used domain elements as matrix elements
3. *PolyMatrix* used a mix of *Poly* and *Expr* as the matrix elements and was used by `risch`.

All of these matrix subclasses were broken in different ways and the introduction of *DomainMatrix* (page 2671) ([#20780](#), [#20759](#), [#20621](#), [#19882](#), [#18844](#)) provides a better solution for all cases. Previous PRs have removed the dependence of these other use cases on *Matrix* ([#21441](#), [#21427](#), [#21402](#)) and now [#21496](#) has deprecated having non-*Expr* in a *Matrix*.

This change makes it possible to improve the internals of the *Matrix* class but it potentially impacts on some downstream use cases that might be similar to the uses of *Matrix* with non-*Expr* elements that were in the SymPy codebase. A potential replacement for code that used *Matrix* with non-*Expr* elements is *DomainMatrix* (page 2671) if the elements are something like domain elements and a domain object can be provided for them. Alternatively if the goal is just printing support then perhaps *TableForm* can be used.

It isn't clear what to advise as a replacement here without knowing more about the usecase. If you are unclear how to update your code, please [open an issue](#) or [write to our mailing list](#) so we can discuss it.

The `get_segments` attribute of plotting objects

The `get_segments` method implemented in *Line2DBaseSeries* (page 2865) is used to convert two list of coordinates, `x` and `y`, into a list of segments used by Matplotlib's *LineCollection* to plot a line.

Since the list of segments is only required by Matplotlib (for example, Bokeh, Plotly, Mayavi, K3D only require lists of coordinates), this has been moved inside the *MatplotlibBackend* class.

Note that previously, the method `get_points()` (page 2866) always returned uniformly sampled points, which meant that some functions were not plotted correctly when using `get_points()` to plot with Matplotlib.

To avoid this problem, the method `get_segments()` could be used, which used adaptive sampling and which could be used with Matplotlib's `LineCollection`. However, this has been changed, and now `get_points()` can also use adaptive sampling. The `get_data()` (page 2866) method can also be used.

The `mdft` function in `sympy.physics.matrices`

The `sympy.physics.matrices.mdft()` function is deprecated. It can be replaced with the `DFT` class in `sympy.matrices.expressions.fourier`.

In particular, replace `mdft(n)` with `DFT(n).as_explicit()`. For example:

```
>>> from sympy.physics.matrices import mdft
>>> mdft(3) # DEPRECATED
Matrix([
[sqrt(3)/3,          sqrt(3)/3,          sqrt(3)/3],
[sqrt(3)/3, sqrt(3)*exp(-2*I*pi/3)/3, sqrt(3)*exp(2*I*pi/3)/3],
[sqrt(3)/3, sqrt(3)*exp(2*I*pi/3)/3, sqrt(3)*exp(-2*I*pi/3)/3]])
```

```
>>> from sympy.matrices.expressions.fourier import DFT
>>> DFT(3)
DFT(3)
>>> DFT(3).as_explicit()
Matrix([
[sqrt(3)/3,          sqrt(3)/3,          sqrt(3)/3],
[sqrt(3)/3, sqrt(3)*exp(-2*I*pi/3)/3, sqrt(3)*exp(2*I*pi/3)/3],
[sqrt(3)/3, sqrt(3)*exp(2*I*pi/3)/3, sqrt(3)*exp(-2*I*pi/3)/3]])
```

This was changed because the `sympy.physics` submodule is supposed to only contain things that are specific to physics, but the discrete Fourier transform matrix is a more general mathematical concept, so it is better located in the `sympy.matrices` module. Furthermore, the `DFT` class is a *matrix expression* (page 1369), meaning it can be unevaluated and support symbolic shape.

The private `SparseMatrix._smat` and `DenseMatrix._mat` attributes

The `._mat` attribute of *Matrix* (page 1361) and the `._smat` attribute of *SparseMatrix* (page 1364) are deprecated.

The internal representation of *Matrix* and *SparseMatrix* was changed to be a *DomainMatrix* (page 2671) in #21626 so that it is no longer possible to expose a mutable list/dict as a way of mutating a *Matrix*. Instead of `._mat` the new `.flat()` method can be used, which returns a new list that cannot be used to mutate the *Matrix* itself. Instead of `._smat` the `.todok()` method can be used which returns a new dict.

Note that these attributes are already changed in SymPy 1.9 to return read-only copies, so that any code that relied on mutating them will be broken. Also these attributes were technically always private (they started with an underscore), so user code should not really have been using them in the first place.

laplace_transform of a Matrix with noconds=False

Prior to version 1.9, calling `laplace_transform()` (page 577) on a *Matrix* (page 1361) with `noconds=False` (which is the default), resulted in a Matrix of tuples:

```
>>> from sympy import laplace_transform, symbols, eye
>>> t, z = symbols('t z')
>>> laplace_transform(eye(2), t, z)
Matrix([
[(1/z, 0, True), (0, 0, True)],
[ (0, 0, True), (1/z, 0, True)]])
```

However, Matrix is only designed to work with Expr objects (see *Non-Expr objects in a Matrix* (page 169) above).

To avoid this, either use `noconds=True` to remove the convergence conditions

```
>>> laplace_transform(eye(2), t, z, noconds=True)
Matrix([
[1/z, 0],
[ 0, 1/z]])
```

or use `legacy_matrix=False` to return the new behavior, which will be to return a single tuple with the Matrix in the first argument and the convergence conditions combined into a single condition for the whole matrix.

```
>>> laplace_transform(eye(2), t, z, legacy_matrix=False)
(Matrix([
[1/z, 0],
[ 0, 1/z]]), 0, True)
```

When this deprecation is removed the `legacy_matrix=False` behavior will become the default, but the flag will be left intact for compatibility.

4.3.5 Version 1.8

sympy.printing.theanocode

Theano has been discontinued, and forked into a new project called *Aesara*. The `sympy.printing.theanocode` module has been renamed to `sympy.printing.aesaracode` (page 2166), and all the corresponding functions have been renamed (e.g., `theano_code` has been renamed to `aesara_code()` (page 2167), `TheanoPrinter` has been renamed to `AesaraPrinter` (page 2166), and so on).

`sympy.assumptions.handlers.AskHandler` and related methods

`Predicate` has experienced a big design change. Previously, its handler was a list of `AskHandler` classes and registration was done by `add_handler()` and `remove_handler()` functions. Now, its handler is a `MultiDispatch` instance and registration is done by `register()` or `register_many()` methods. Users must define a predicate class to introduce a new one.

Previously, handlers were defined and registered this way:

```
class AskPrimeHandler(AskHandler):
    @staticmethod
    def Integer(expr, assumptions):
        return expr.is_prime

register_handler('prime', AskPrimeHandler)
```

It should be changed to this:

```
# Predicate definition.
# Not needed if you are registering the handler to existing predicate.
class PrimePredicate(Predicate):
    name = 'prime'
Q.prime = PrimePredicate()

# Handler registration
@Q.prime.register(Integer)
def _(expr, assumptions):
    return expr.is_prime
```

See GitHub issue [#20209](#).

4.3.6 Version 1.7.1

Calling `sympy.stats.StochasticProcess.distribution` with `RandomIndexedSymbol`

The `distribution` method of `sympy.stats.stochastic processes` (page 2944) used to accept a `RandomIndexedSymbol` (that is, a stochastic process indexed with a timestamp), but should now only be called with the timestamp.

For example, if you have

```
>>> from sympy import symbols
>>> from sympy.stats import WienerProcess
>>> W = WienerProcess('W')
>>> t = symbols('t', positive=True)
```

Previously this would work

```
W.distribution(W(t)) # DEPRECATED
```

It should now be called like

```
>>> W.distribution(t)
NormalDistribution(0, sqrt(t))
```

This was change was made as part of a change to store only Basic objects in `sympy.stats`.args. See issue [#20078](#) for details.

4.3.7 Version 1.7

`sympy.stats.DiscreteMarkovChain.absorbing_probabilites()`

The `absorbing_probabilites` method name was misspelled. The correct spelling `absorbing_probabilities()` (page 2946) (“absorbing probabilities”) should be used instead.

`sympy.utilities.misc.find_executable()`

The function `sympy.utilities.misc.find_executable()` is deprecated. Instead use the standard library `shutil.which()` function, which has been in the standard library since Python 3.3 and is more powerful.

Mutable attributes in `sympy.diffgeom`

Several parts of `sympy.diffgeom` (page 2799) have been updated to no longer be mutable, which better matches the immutable design used in the rest of SymPy.

- Passing strings for symbol names in `CoordSystem` (page 2801) is deprecated. Instead you should be explicit and pass symbols with the appropriate assumptions, for instance, instead of

```
CoordSystem(name, patch, ['x', 'y']) # DEPRECATED
```

use

```
CoordSystem(name, patch, symbols('x y', real=True))
```

- Similarly, the `names` keyword argument has been renamed to `symbols`, which should be a list of symbols.
- The `Manifold.patches` attribute is deprecated. Patches should be tracked separately.
- The `Patch.coord_systems` attribute is deprecated. Coordinate systems should be tracked separately.
- The `CoordSystem.transforms` attribute, `CoordSystem.connect_to()` method, and `CoordSystem.coord_tuple_transform_to()` method are deprecated. Instead, use the `relations` keyword to the `CoordSystem` class constructor and the `CoordSystem.transformation()` (page 2805) and `CoordSystem.transform()` (page 2805) methods (see the docstring of `CoordSystem` (page 2801) for examples).

The unicode argument and attribute to `sympy.printing.pretty.stringpict.prettyForm` and the `sympy.printing.pretty.pretty_symbology.xstr` function

The `sympy.printing.pretty.pretty_symbology.xstr` function, and the `unicode` argument and attribute to `sympy.printing.pretty.stringpict.prettyForm` (page 2188) were both present to support the Unicode behavior of Python 2. Since Unicode strings are the default in Python 3, these are not needed any more. `xstr()` should be replaced with just `str()`, the `unicode` argument to `prettyForm` should be omitted, and the `prettyForm.unicode` attribute should be replaced with the `prettyForm.s` attribute.

Passing the arguments to `lambdify` as a set

Passing the function arguments to `lambdify` as a set is deprecated. Instead pass them as a list or tuple. For example, instead of

```
lambdify({x, y}, x + 2*y) # WRONG
```

use

```
lambdify((x, y), x + 2*y) # RIGHT
```

This is because sets are unordered. For instance, in the above example it would be impossible for `lambdify` to know if it was called with `{x, y}` or `{y, x}`. Thus, when passed the arguments as a set `lambdify` would have to guess their order, which would lead to an incorrect function if it guessed incorrectly.

Core operators no longer accept non-Expr args

The core operator classes `Add` (page 1013), `Mul` (page 1009), and `Pow` (page 1005) can no longer be constructed directly with objects that are not subclasses of `Expr` (page 947).

`Expr` (page 947) is the superclass of all SymPy classes that represent scalar numeric quantities. For example, `sin` (page 389), `Symbol` (page 976), and `Add` (page 1013) are all subclasses of `Expr` (page 947). However, many objects in SymPy are not `Expr` (page 947) because they represent some other type of mathematical object. For example, `Set` (page 1185), `Poly` (page 2378), and `Boolean` (page 1163) are all non-Expr. These do not make mathematical sense inside of `Add`, `Mul`, and `Pow`, which are designed specifically to represent the addition, multiplication, and exponentiation of scalar complex numbers.

Manually constructing one of these classes with such an object is possible, but it will generally create something that will then break. For example

```
Mul(1, Tuple(2)) # This is deprecated
```

works and creates `Tuple(2)`, but only because `Mul` is “tricked” by always treating $1 \cdot x = x$. If instead you try

```
Mul(2, Tuple(2)) # This is deprecated
```

it fails with an exception

```
AttributeError: 'Tuple' object has no attribute 'as_coeff_Mul'
```

because it tries to call a method of `Expr` on the `Tuple` object, which does not have all the `Expr` methods (because it is not a subclass of `Expr`).

If you want to use the `+`, `*`, or `**` operation on a non-`Expr` object, use the operator directly rather than using `Mul`, `Add` or `Pow`. If functional versions of these are desired, you can use a `lambda` or the `operator` module.

4.3.8 Version 1.6

Various `sympy.utilities` submodules have moved

The following submodules have been renamed.

- `sympy.utilities.benchmarking` → `sympy.testing.benchmarking`
- `sympy.utilities.pytest` → `sympy.testing.pytest`
- `sympy.utilities.randtests` → `sympy.core.random`
- `sympy.utilities.runtests` → `sympy.testing.runtests`
- `sympy.utilities.tmpfiles` → `sympy.testing.tmpfiles`

`sympy.testing.randtest`

`sympy.testing.randtest` is deprecated. The functions in it have been moved to `sympy.core.random`. The following functions have been moved.

- `sympy.testing.randtest.random_complex_number` → `sympy.core.random.random_complex_number`
- `sympy.testing.randtest.verify_numerically` → `sympy.core.random.verify_numerically`
- `sympy.testing.randtest.test_derivative_numerically` → `sympy.core.random.test_derivative_numerically`
- `sympy.testing.randtest._randrange` → `sympy.core.random._randrange`
- `sympy.testing.randtest._randint` → `sympy.core.random._randint`

Mixing `Poly` and non-polynomial expressions in binary operations

In previous versions of SymPy, `Poly` (page 2378) was a subclass of `Expr` (page 947), but it has been changed to only be a subclass of `Basic` (page 927). This means that some things that used to work with `Poly` are now deprecated because they are only designed to work with `Expr` (page 947) objects.

This includes combining `Poly` with `Expr` objects using binary operations, for example

```
Poly(x)*sin(x) # DEPRECATED
```

To do this, either explicitly convert the non-`Poly` operand to a `Poly` using `Expr.as_poly()` (page 956) or convert the `Poly` operand to an `Expr` (page 947) using `Poly.as_expr()` (page 2383), depending on which type you want the result to be.

The `print_cyclic` flag of `sympy.combinatorics.Permutation`

The `print_cyclic` attribute of `sympy.combinatorics.Permutation` (page 257) controls whether permutations print as cycles or arrays. This would be done by setting `Permutation.print_cyclic = True` or `Permutation.print_cyclic = False`. However, this method of controlling printing is bad because it is a global flag, but printing should not depend on global behavior.

Instead, users should use the `perm_cyclic` flag of the corresponding printer. The easiest way to configure this is to set the flag when calling `init_printing()` (page 2119), like

```
>>> from sympy import init_printing
>>> init_printing(perm_cyclic=False) # Makes Permutation print in array form
>>> from sympy.combinatorics import Permutation
>>> Permutation(1, 2)(3, 4)
(0 1 2 3 4)
(0 2 1 4 3)
```

The `Permutation` (page 257) docstring contains more details on the `perm_cyclic` flag.

Using `integrate` with `Poly`

In previous versions of SymPy, `Poly` (page 2378) was a subclass of `Expr` (page 947), but it has been changed to only be a subclass of `Basic` (page 927). This means that some things that used to work with `Poly` are now deprecated because they are only designed to work with `Expr` (page 947) objects.

This includes calling `integrate()` (page 598) or `Integral` (page 601) with `Poly`.

To integrate a `Poly`, use the `Poly.integrate()` (page 2398) method. To compute the integral as an `Expr` (page 947) object, call the `Poly.as_expr()` (page 2383) method first.

See also *Mixing Poly and non-polynomial expressions in binary operations* (page 175) above.

The string fallback in `sympify()`

The current behavior of `sympify()` (page 918) is that `sympify(expr)` tries various methods to try to convert `expr` into a SymPy object. If all these methods fail, it takes `str(expr)` and tries to parse it using `parse_expr()` (page 2122). This string fallback feature is deprecated. It is problematic for a few reasons:

- It can affect performance in major ways. See for instance issues [#18056](#) and [#15416](#) where it caused up to 100x slowdowns. The issue is that SymPy functions automatically call `sympify` on their arguments. Whenever a function is passed something that `sympify` doesn't know how to convert to a SymPy object, for instance, a Python function type, it passes the string to `parse_expr()` (page 2122). This is significantly slower than the direct conversions that happen by default. This occurs specifically whenever `sympify()` is used in library code instead of `_sympify()` (or equivalently `sympify(strict=True)`), but presently this is done a lot. Using `strict=True` will at some point be the default for all library code, but this is a [harder change to make](#).
- It can cause security issues, since strings are eval'd, and objects can return whatever string they want in their `__repr__`. See also <https://github.com/sympy/sympy/pull/12524>.

- It really isn't very useful to begin with. Just because an object's string form can be parsed into a SymPy expression doesn't mean it should be parsed that way. This is usually correct for custom numeric types, but an object's repr could be anything. For instance, if the string form of an object looks like a valid Python identifier, it will parse as a Symbol.

There are plenty of ways to make custom objects work inside of `sympify()` (page 918).

- Firstly, if an object is intended to work alongside other SymPy expressions, it should subclass from `Basic` (page 927) (or `Expr` (page 947)). If it does, `sympify()` (page 918) will just return it unchanged because it will already be a valid SymPy object.
- For objects that you control, you can add the `_sympy_` method. The `sympify docstring` (page 918) has an example of this.
- For objects that you don't control, you can add a custom converter to the `sympy.core.sympify.converter` dictionary. The `sympify()` (page 918) docstring also has an example of this.

To silence this deprecation warning in all cases, you can pass `strict=True` to `sympify()`. However, note that this will also disable some other conversions such as conversion of strings (for converting strings to SymPy types, you can explicitly use `parse_expr()` (page 2122)).

Creating an indefinite Integral with an Eq argument

Passing an `Eq()` (page 1023) object to `integrate()` (page 598) is deprecated in the case where the integral is indefinite. This is because if $f(x) = g(x)$, then $\int f(x) dx = \int g(x) dx$ is not true in general, due to the arbitrary constants (which `integrate` does not include).

If you want to make an equality of indefinite integrals, use `Eq(integrate(f(x), x), integrate(g(x), x))` explicitly.

If you already have an equality object `eq`, you can use `Eq(integrate(eq.lhs, x), integrate(eq.rhs, x))`.

4.3.9 Version 1.5

Tensor.fun_eval and Tensor.__call__

`TensExpr.fun_eval` and `Tensor.__call__` (i.e., calling a tensor to evaluate it) are deprecated. The `Tensor.substitute_indices()` method should be used. This was changed because `fun_eval` was considered a confusing name and using function evaluation was considered both confusing and dangerous.

TensorType

The `TensorType` class is deprecated. Use `tensor_heads()` (page 1413) instead. The `TensorType` class had no purpose except shorter creation of `TensorHead` (page 1411) objects.

See also *The tensorhead() function* (page 178) below.

The `dummy_fmt` argument to `TensorIndexType`

The `dummy_fmt` keyword argument to *TensorIndexType* (page 1409) is deprecated. Setting `dummy_fmt='L'` leads to `_dummy_fmt='L_%d'`, which is confusing and uses obsolete string formatting. `dummy_name` should be used instead. This change was made because `dummy_name` is a clearer name.

The `metric` argument to `TensorIndexType`

The `metric` keyword argument to *TensorIndexType* (page 1409) is deprecated. The name “metric” was ambiguous because it meant “metric symmetry” in some places and “metric tensor” in others.

Either the `metric_symmetry` keyword or the `TensorIndexType.set_metric()` method should be used instead.

The `get_kronecker_delta()` and `get_epsilon()` methods of `TensorIndexType`

The `get_kronecker_delta()` and `get_epsilon()` methods of *TensorIndexType* (page 1409) are deprecated. Use the `TensorIndexType.delta` and `TensorIndexType.epsilon` properties instead, respectively.

The `tensorsymmetry()` function

The `tensorsymmetry()` function in `sympy.tensor` is deprecated. Use the *TensorSymmetry* (page 1420) class constructor instead.

`TensorSymmetry` is preferred over `tensorsymmetry()` because the latter

1. Does not have any extra functionality
2. Involves obscure Young tableau
3. Is not a member of the `TensorSymmetry` class

The `tensorhead()` function

The `tensorhead()` function is deprecated in favor of *tensor_heads()* (page 1413). `tensor_heads()` is more consistent with other SymPy names (i.e., `Symbol` and `symbols()` or `TensorIndex` and `tensor_indices()`). It also does not use Young tableau to denote symmetries.

Methods to `sympy.physics.units.Quantity`

The following methods of `sympy.physics.units.quantities.Quantity` (page 1587) are deprecated.

- `Quantity.set_dimension()`. This should be replaced with `unit_system.set_quantity_dimension` or `Quantity.set_global_dimension()`.
- `Quantity.set_scale_factor()`. This should be replaced with `unit_system.set_quantity_scale_factor` or `Quantity.set_global_relative_scale_factor()` (page 1588)
- `Quantity.get_dimensional_expr()`. This is now associated with `UnitSystem` (page 1587) objects. The dimensional relations depend on the unit system used. Use `unit_system.get_dimensional_expr()` instead.
- `Quantity._collect_factor_and_dimension`. This has been moved to the `UnitSystem` (page 1587) class. Use `unit_system._collect_factor_and_dimension(expr)` instead.

See *The dimension and scale factor arguments to `sympy.physics.units.Quantity`* (page 182) below for the motivation for this change.

The `is_EmptySet` attribute of sets

The `is_EmptySet` attribute of `Set` (page 1185) objects is deprecated. Instead either use

```
from sympy import S
s is S.EmptySet
```

or

```
s.is_empty
```

The difference is that `s.is_empty` may return `None` if it is unknown if the set is empty.

`ProductSet(iterable)`

Passing a single iterable as the first argument to `ProductSet` (page 1199) is deprecated. Creating a product set from an iterable should be done using `ProductSet(*iterable)`, or as each individual argument. For example

```
>>> from sympy import ProductSet
>>> sets = [{i} for i in range(3)]
>>> ProductSet(*sets)
ProductSet({0}, {1}, {2})
>>> ProductSet({1, 2}, {1})
ProductSet({1, 2}, {1})
```

This is done because sets themselves can be iterables, and sets of sets are allowed. But the product set of a single set should mathematically be that set itself (or more exactly, the set of 1-tuples of elements of that set). Automatically denesting a single iterable makes it impossible to represent this object and makes `ProductSet` not generalize correctly when passed 1 argument. On the other hand, treating the first argument differently if it is a set than if it is another type of iterable (which is what is currently done in the deprecated code path) is confusing behavior.

The `set_potential_energy` method in `sympy.physics.mechanics`

The `set_potential_energy()` methods of `sympy.physics.mechanics.particle.Particle` (page 1743) and `sympy.physics.mechanics.rigidbody.RigidBody` (page 1746) are deprecated.

Instead one should set the `Particle.potential_energy` (page 1746) and `RigidBody.potential_energy` (page 1749) attributes to set the potential energy, like

```
P.potential_energy = scalar
```

This change was made to be more Pythonic, by using setters and getters of a `@property` method rather than an explicit `set_` method.

Using a set for the condition in `ConditionSet`

Using a set for the condition in `ConditionSet` is deprecated. A boolean should be used instead. This is because the condition is mathematically a boolean, and it is ambiguous what a set should mean in this context.

To fix this deprecation, replace

```
ConditionSet(symbol, set_condition)
```

with

```
ConditionSet(symbol, And(*[Eq(lhs, 0) for lhs in set_condition]))
```

For example,

```
ConditionSet((x, y), {x + 1, x + y}, S.Reals) # DEPRECATED
```

would become

```
ConditionSet((x, y), Eq(x + 1, 0) & Eq(x + y, 0), S.Reals)
```

The `max_degree` and `get_upper_degree` properties of `sympy.polys.multivariate_resultants.DixonResultant`

The `max_degree` property and `get_upper_degree()` methods of `DixonResultant` are deprecated. See issue [#17749](#) for details.

`Eq(expr)` with the `rhs` defaulting to 0

Calling `Eq` (page 1023) with a single argument is deprecated. This caused the right-hand side to default to 0, but this behavior was confusing. You should explicitly use `Eq(expr, 0)` instead.

Non-tuple iterable for the first argument to Lambda

Using a non-tuple as the first argument to *Lambda* (page 1039) is deprecated. If you have a non-tuple, convert it to a tuple first, like `Lambda(tuple(args), expr)`.

This was done so that `Lambda` could support general tuple unpacking, like

```
>>> from sympy import Lambda, symbols
>>> x, y, z = symbols('x y z')
>>> f = Lambda((x, (y, z)), x + y + z)
>>> f(1, (2, 3))
6
```

The evaluate flag to `differentiate_finite`

The evaluate flag to `differentiate_finite()` (page 241) is deprecated.

`differentiate_finite(expr, x, evaluate=True)` expands the intermediate derivatives before computing differences. But this usually not what you want, as it does not satisfy the product rule.

If you really do want this behavior, you can emulate it with

```
diff(expr, x).replace(
    lambda arg: arg.is_Derivative,
    lambda arg: arg.as_finite_difference())
```

See the discussion on issue [#17881](#).

4.3.10 Version 1.4

TensorIndexType.data and related methods

The `TensorIndexType.data` property is deprecated, as well as several methods which made use of it including the `get_matrix()`, the `__getitem__()` (indexing), `__iter__()` (iteration), `__components_data_full_destroy()`, and `__pow__()` (`**`) methods. Storing data on tensor objects was a design flaw and not consistent with how the rest of SymPy works.

Instead, the `TensExpr.replace_with_arrays()` (page 1414) method should be used.

The `clear_cache` and `clear_subproducts` keywords to `Matrix.is_diagonalizable`

The `clear_cache` and `clear_subproducts` keywords to `Matrix.is_diagonalizable()` (page 1241) are deprecated. These used to clear cached entries, but this cache was removed because it was not actually safe given that `Matrix` is mutable. The keywords now do nothing.

The rows and cols keyword arguments to `Matrix.jordan_block`

The rows and cols keywords to `Matrix.jordan_block` (page 1345) are deprecated. The size parameter should be used to specify the (square) number of rows and columns.

The non-square matrices created by setting rows and cols are not mathematically Jordan block matrices, which only make sense as square matrices.

To emulate the deprecated `jordan_block(rows=n, cols=m)` behavior, use a general banded matrix constructor, like

```
>>> from sympy import Matrix, symbols
>>> eigenvalue = symbols('x')
>>> def entry(i, j):
...     if i == j:
...         return eigenvalue
...     elif i + 1 == j: # use j + 1 == i for band='lower'
...         return 1
...     return 0
>>> # the same as the deprecated Matrix.jordan_block(rows=3, cols=5,
↳eigenvalue=x)
>>> Matrix(3, 5, entry)
Matrix([
[x, 1, 0, 0, 0],
[0, x, 1, 0, 0],
[0, 0, x, 1, 0]])
```

4.3.11 Version 1.3

The `source()` function

The `source()` (page 2116) function is deprecated. Use `inspect.getsource(obj)` instead, or if you are in IPython or Jupyter, use `obj??`.

The dimension and scale_factor arguments to `sympy.physics.units.Quantity`

The dimension and scale_factor arguments to `sympy.physics.units.quantities.Quantity` (page 1587) are deprecated.

The problem with these arguments is that **dimensions** are **not** an **absolute** association to a quantity. For example:

- in natural units length and time are the same dimension (so you can sum meters and seconds).
- SI and cgs units have different dimensions for the same quantities.

At this point a problem arises for scale factor as well: while it is always true that `kilometer / meter == 1000`, some other quantities may have a relative scale factor or not depending on which unit system is currently being used.

Instead, things should be managed on the `DimensionSystem` (page 1584) class. The `DimensionSystem.set_quantity_dimension()` method should be used instead of the

dimension argument, and the `DimensionSystem.set_quantity_scale_factor()` method should be used instead of the `scale_factor` argument.

See issue [#14318](#) for more details. See also [Methods to sympy.physics.units.Quantity](#) (page 179) above.

Importing `classof` and `a2idx` from `sympy.matrices.matrices`

The functions `sympy.matrices.matrices.classof` and `sympy.matrices.matrices.a2idx` were duplicates of the same functions in `sympy.matrices.common`. The two functions should be used from the `sympy.matrices.common` module instead.

4.3.12 Version 1.2

Dot product of non-row/column vectors

The `Matrix.dot()` (page 1296) method has confusing behavior where `A.dot(B)` returns a list corresponding to `flatten(A.T*B.T)` when `A` and `B` are matrices that are not vectors (i.e., neither dimension is size 1). This is confusing. The purpose of `Matrix.dot()` is to perform a mathematical dot product, which should only be defined for vectors (i.e., either a $n \times 1$ or $1 \times n$ matrix), but in a way that works regardless of whether each argument is a row or column vector. Furthermore, returning a list here was much less useful than a matrix would be, and resulted in a polymorphic return type depending on the shapes of the inputs.

This behavior is deprecated. `Matrix.dot` should only be used to do a mathematical dot product, which operates on row or column vectors. Use the `*` or `@` operators to do matrix multiplication.

```
>>> from sympy import Matrix
>>> A = Matrix([[1, 2], [3, 4]])
>>> B = Matrix([[2, 3], [1, 2]])
>>> A*B
Matrix([
 [ 4,  7],
[10, 17]])
>>> A@B
Matrix([
 [ 4,  7],
[10, 17]])
```

`sympy.geometry.Line3D.equation` no longer needs the `k` argument

The `k` argument to `sympy.geometry.line.Line3D.equation()` (page 2244) method is deprecated.

Previously, the function `Line3D.equation` returned `(X, Y, Z, k)` which was changed to `(Y-X, Z-X)` (here `X`, `Y` and `Z` are expressions of `x`, `y` and `z` respectively). As in 2D an equation is returned relating `x` and `y` just like that in 3D two equations will be returned relating `x`, `y` and `z`.

So in the new `Line3D.equation` the `k` argument is not needed anymore. Now the `k` argument is effectively ignored. A `k` variable is temporarily formed inside `equation()` and then gets substituted using `subs()` in terms of `x` and then $(Y-X, Z-X)$ is returned.

Previously:

```
>>> from sympy import Point3D, Line3D
>>> p1, p2 = Point3D(1, 2, 3), Point3D(5, 6, 7)
>>> l = Line3D(p1, p2)
>>> l.equation()
(x/4 - 1/4, y/4 - 1/2, z/4 - 3/4, k)
```

Now:

```
>>> from sympy import Point3D, Line3D, solve
>>> p1, p2 = Point3D(1, 2, 3), Point3D(5, 6, 7)
>>> l = Line3D(p1, p2)
>>> l.equation()
(-x + y - 1, -x + z - 2)
```


API REFERENCE

This section contains a summary of SymPy modules, functions, classes, and methods. All functions and objects implemented in the `sympy` core subpackage are documented below.

5.1 Basics

Contains a description of operations for the basic modules. Subcategories include: *absolute basics*, *manipulation*, *assumptions*, *functions*, *simplification*, *calculus*, *solvers*, and some other subcategories.

5.2 Code Generation

Contains a description of methods for the generation of compilable and executable code.

5.3 Logic

Contains method details for the *logic* and *sets* modules.

5.4 Matrices

Discusses methods for the matrices, tensor and vector modules.

5.5 Number Theory

Documents methods for the Number theory module.

5.6 Physics

Contains documentation for Physics methods.

5.7 Utilities

Contains docstrings for methods of several utility modules. Subcategories include: *Interactive, Parsing, Printing, Testing, Utilities*.

5.8 Topics

Contains method docstrings for several modules. Subcategories include : *Plotting, Polynomials, Geometry, Category Theory, Cryptography, Differential, Holonomic, Lie Algebra*, and *Stats*.

5.8.1 Basics

Contents

Assumptions

A module to implement logical predicates and assumption system.

Predicate

class sympy.assumptions.assume.**Predicate**(*args, **kwargs)

Base class for mathematical predicates. It also serves as a constructor for undefined predicate objects.

Explanation

Predicate is a function that returns a boolean value [1].

Predicate function is object, and it is instance of predicate class. When a predicate is applied to arguments, AppliedPredicate instance is returned. This merely wraps the argument and remain unevaluated. To obtain the truth value of applied predicate, use the function ask.

Evaluation of predicate is done by multiple dispatching. You can register new handler to the predicate to support new types.

Every predicate in SymPy can be accessed via the property of Q. For example, Q.even returns the predicate which checks if the argument is even number.

To define a predicate which can be evaluated, you must subclass this class, make an instance of it, and register it to Q. After then, dispatch the handler by argument types.

If you directly construct predicate using this class, you will get UndefinedPredicate which cannot be dispatched. This is useful when you are building boolean expressions which do not need to be evaluated.

Examples

Applying and evaluating to boolean value:

```
>>> from sympy import Q, ask
>>> ask(Q.prime(7))
True
```

You can define a new predicate by subclassing and dispatching. Here, we define a predicate for sexy primes [2] as an example.

```
>>> from sympy import Predicate, Integer
>>> class SexyPrimePredicate(Predicate):
...     name = "sexyprime"
>>> Q.sexyprime = SexyPrimePredicate()
>>> @Q.sexyprime.register(Integer, Integer)
... def _(int1, int2, assumptions):
...     args = sorted([int1, int2])
...     if not all(ask(Q.prime(a), assumptions) for a in args):
...         return False
...     return args[1] - args[0] == 6
>>> ask(Q.sexyprime(5, 11))
True
```

Direct constructing returns UndefinedPredicate, which can be applied but cannot be dispatched.

```
>>> from sympy import Predicate, Integer
>>> Q.P = Predicate("P")
>>> type(Q.P)
<class 'sympy.assumptions.assume.UndefinedPredicate'>
>>> Q.P(1)
Q.P(1)
```

(continues on next page)

(continued from previous page)

```
>>> Q.P.register(Integer)(lambda expr, assump: True)
Traceback (most recent call last):
...
TypeError: <class 'sympy.assumptions.assume.UndefinedPredicate'> cannot
↳ be dispatched.
```

References

[R5], [R6]

eval(args, assumptions=True)

Evaluate self(*args) under the given assumptions.

This uses only direct resolution methods, not logical inference.

handler = <dispatched AskPredicateHandler>

classmethod register(*types, **kwargs)

Register the signature to the handler.

classmethod register_many(*types, **kwargs)

Register multiple signatures to same handler.

class sympy.assumptions.assume.**AppliedPredicate**(predicate, *args)

The class of expressions resulting from applying Predicate to the arguments. AppliedPredicate merely wraps its argument and remain unevaluated. To evaluate it, use the ask() function.

Examples

```
>>> from sympy import Q, ask
>>> Q.integer(1)
Q.integer(1)
```

The function attribute returns the predicate, and the arguments attribute returns the tuple of arguments.

```
>>> type(Q.integer(1))
<class 'sympy.assumptions.assume.AppliedPredicate'>
>>> Q.integer(1).function
Q.integer
>>> Q.integer(1).arguments
(1,)
```

Applied predicates can be evaluated to a boolean value with ask:

```
>>> ask(Q.integer(1))
True
```

property arg

Return the expression used by this assumption.

Examples

```
>>> from sympy import Q, Symbol
>>> x = Symbol('x')
>>> a = Q.integer(x + 1)
>>> a.arg
x + 1
```

property arguments

Return the arguments which are applied to the predicate.

property function

Return the predicate.

Querying

Queries are used to ask information about expressions. Main method for this is `ask()`:

```
sympy.assumptions.ask(proposition, assumptions=True, context={})
```

Function to evaluate the proposition with assumptions.

Parameters

proposition : Any boolean expression.

Proposition which will be evaluated to boolean value. If this is not `AppliedPredicate`, it will be wrapped by `Q.is_true`.

assumptions : Any boolean expression, optional.

Local assumptions to evaluate the *proposition*.

context : `AssumptionsContext`, optional.

Default assumptions to evaluate the *proposition*. By default, this is `sympy.assumptions.global_assumptions` variable.

Returns

True, False, or None

Raises

TypeError : *proposition* or *assumptions* is not valid logical expression.

ValueError : assumptions are inconsistent.

Explanation

This function evaluates the proposition to True or False if the truth value can be determined. If not, it returns None.

It should be discerned from `refine()` (page 197) which, when applied to a proposition, simplifies the argument to symbolic Boolean instead of Python built-in True, False or None.

Syntax

- **ask(*proposition*)**

Evaluate the *proposition* in global assumption context.

- **ask(proposition, assumptions)**

Evaluate the *proposition* with respect to *assumptions* in global assumption context.

Examples

```
>>> from sympy import ask, Q, pi
>>> from sympy.abc import x, y
>>> ask(Q.rational(pi))
False
>>> ask(Q.even(x*y), Q.even(x) & Q.integer(y))
True
>>> ask(Q.prime(4*x), Q.integer(x))
False
```

If the truth value cannot be determined, None will be returned.

```
>>> print(ask(Q.odd(3*x))) # cannot determine unless we know x
None
```

ValueError is raised if assumptions are inconsistent.

```
>>> ask(Q.integer(x), Q.even(x) & Q.odd(x))
Traceback (most recent call last):
...
ValueError: inconsistent assumptions Q.even(x) & Q.odd(x)
```

Notes

Relations in assumptions are not implemented (yet), so the following will not give a meaningful result.

```
>>> ask(Q.positive(x), x > 0)
```

It is however a work in progress.

See also:

[*sympy.assumptions.refine.refine*](#) (page 197)

Simplification using assumptions. Proposition is not reduced to None if the truth value cannot be determined.

ask's optional second argument should be a boolean expression involving assumptions about objects in *expr*. Valid values include:

- `Q.integer(x)`
- `Q.positive(x)`
- `Q.integer(x) & Q.positive(x)`
- etc.

`Q` is an object holding known predicates.

See documentation for the logic module for a complete list of valid boolean expressions.

You can also define a context so you don't have to pass that argument each time to function `ask()`. This is done by using the assuming context manager from module `sympy.assumptions`.

```
>>> from sympy import *
>>> x = Symbol('x')
>>> y = Symbol('y')
>>> facts = Q.positive(x), Q.positive(y)
>>> with assuming(*facts):
...     print(ask(Q.positive(2*x + y)))
True
```

Contents

Ask

Module for querying SymPy objects about assumptions.

class `sympy.assumptions.ask.AssumptionKeys`

This class contains all the supported keys by `ask`. It should be accessed via the instance `sympy.Q`.

`sympy.assumptions.ask.ask(proposition, assumptions=True, context={})`

Function to evaluate the proposition with assumptions.

Parameters

proposition : Any boolean expression.

Proposition which will be evaluated to boolean value. If this is not `AppliedPredicate`, it will be wrapped by `Q.is_true`.

assumptions : Any boolean expression, optional.

Local assumptions to evaluate the *proposition*.

context : `AssumptionsContext`, optional.

Default assumptions to evaluate the *proposition*. By default, this is `sympy.assumptions.global_assumptions` variable.

Returns

True, False, or None

Raises

TypeError : *proposition* or *assumptions* is not valid logical expression.

ValueError : assumptions are inconsistent.

Explanation

This function evaluates the proposition to True or False if the truth value can be determined. If not, it returns None.

It should be discerned from *refine()* (page 197) which, when applied to a proposition, simplifies the argument to symbolic Boolean instead of Python built-in True, False or None.

Syntax

- **ask(*proposition*)**
Evaluate the *proposition* in global assumption context.
- **ask(*proposition*, *assumptions*)**
Evaluate the *proposition* with respect to *assumptions* in global assumption context.

Examples

```
>>> from sympy import ask, Q, pi
>>> from sympy.abc import x, y
>>> ask(Q.rational(pi))
False
>>> ask(Q.even(x*y), Q.even(x) & Q.integer(y))
True
>>> ask(Q.prime(4*x), Q.integer(x))
False
```

If the truth value cannot be determined, None will be returned.

```
>>> print(ask(Q.odd(3*x))) # cannot determine unless we know x
None
```

ValueError is raised if assumptions are inconsistent.

```
>>> ask(Q.integer(x), Q.even(x) & Q.odd(x))
Traceback (most recent call last):
...
ValueError: inconsistent assumptions Q.even(x) & Q.odd(x)
```

Notes

Relations in assumptions are not implemented (yet), so the following will not give a meaningful result.

```
>>> ask(Q.positive(x), x > 0)
```

It is however a work in progress.

See also:

***sympy.assumptions.refine.refine* (page 197)**

Simplification using assumptions. Proposition is not reduced to None if the truth value cannot be determined.

`sympy.assumptions.ask.register_handler(key, handler)`

Register a handler in the ask system. key must be a string and handler a class inheriting from AskHandler.

Deprecated since version 1.8.: Use `multidispatch` handler instead. See [Predicate](#) (page 195).

`sympy.assumptions.ask.remove_handler(key, handler)`

Removes a handler from the ask system. Same syntax as `register_handler`

Deprecated since version 1.8.: Use `multidispatch` handler instead. See [Predicate](#) (page 195).

Assume

A module which implements predicates and assumption context.

class `sympy.assumptions.assume.AppliedPredicate(predicate, *args)`

The class of expressions resulting from applying `Predicate` to the arguments. `AppliedPredicate` merely wraps its argument and remain unevaluated. To evaluate it, use the `ask()` function.

Examples

```
>>> from sympy import Q, ask
>>> Q.integer(1)
Q.integer(1)
```

The function attribute returns the predicate, and the arguments attribute returns the tuple of arguments.

```
>>> type(Q.integer(1))
<class 'sympy.assumptions.assume.AppliedPredicate'>
>>> Q.integer(1).function
Q.integer
>>> Q.integer(1).arguments
(1,)
```

Applied predicates can be evaluated to a boolean value with `ask`:

```
>>> ask(Q.integer(1))
True
```

property arg

Return the expression used by this assumption.

Examples

```
>>> from sympy import Q, Symbol
>>> x = Symbol('x')
>>> a = Q.integer(x + 1)
>>> a.arg
x + 1
```

property arguments

Return the arguments which are applied to the predicate.

property function

Return the predicate.

class sympy.assumptions.assume.AssumptionsContext

Set containing default assumptions which are applied to the `ask()` function.

Explanation

This is used to represent global assumptions, but you can also use this class to create your own local assumptions contexts. It is basically a thin wrapper to Python's set, so see its documentation for advanced usage.

Examples

The default assumption context is `global_assumptions`, which is initially empty:

```
>>> from sympy import ask, Q
>>> from sympy.assumptions import global_assumptions
>>> global_assumptions
AssumptionsContext()
```

You can add default assumptions:

```
>>> from sympy.abc import x
>>> global_assumptions.add(Q.real(x))
>>> global_assumptions
AssumptionsContext({Q.real(x)})
>>> ask(Q.real(x))
True
```

And remove them:

```
>>> global_assumptions.remove(Q.real(x))
>>> print(ask(Q.real(x)))
None
```

The `clear()` method removes every assumption:

```
>>> global_assumptions.add(Q.positive(x))
>>> global_assumptions
AssumptionsContext({Q.positive(x)})
```

(continues on next page)

(continued from previous page)

```
>>> global_assumptions.clear()
>>> global_assumptions
AssumptionsContext()
```

See also:*assuming* (page 197)

add(*assumptions)
Add assumptions.

class sympy.assumptions.assume.**Predicate**(*args, **kwargs)

Base class for mathematical predicates. It also serves as a constructor for undefined predicate objects.

Explanation

Predicate is a function that returns a boolean value [1].

Predicate function is object, and it is instance of predicate class. When a predicate is applied to arguments, AppliedPredicate instance is returned. This merely wraps the argument and remain unevaluated. To obtain the truth value of applied predicate, use the function ask.

Evaluation of predicate is done by multiple dispatching. You can register new handler to the predicate to support new types.

Every predicate in SymPy can be accessed via the property of Q. For example, Q.even returns the predicate which checks if the argument is even number.

To define a predicate which can be evaluated, you must subclass this class, make an instance of it, and register it to Q. After then, dispatch the handler by argument types.

If you directly construct predicate using this class, you will get UndefinedPredicate which cannot be dispatched. This is useful when you are building boolean expressions which do not need to be evaluated.

Examples

Applying and evaluating to boolean value:

```
>>> from sympy import Q, ask
>>> ask(Q.prime(7))
True
```

You can define a new predicate by subclassing and dispatching. Here, we define a predicate for sexy primes [2] as an example.

```
>>> from sympy import Predicate, Integer
>>> class SexyPrimePredicate(Predicate):
...     name = "sexyprime"
>>> Q.sexyprime = SexyPrimePredicate()
>>> @Q.sexyprime.register(Integer, Integer)
... def _(int1, int2, assumptions):
```

(continues on next page)

(continued from previous page)

```
...     args = sorted([int1, int2])
...     if not all(ask(Q.prime(a), assumptions) for a in args):
...         return False
...     return args[1] - args[0] == 6
>>> ask(Q.sexyprime(5, 11))
True
```

Direct constructing returns UndefinedPredicate, which can be applied but cannot be dispatched.

```
>>> from sympy import Predicate, Integer
>>> Q.P = Predicate("P")
>>> type(Q.P)
<class 'sympy.assumptions.assume.UndefinedPredicate'>
>>> Q.P(1)
Q.P(1)
>>> Q.P.register(Integer)(lambda expr, assumpt: True)
Traceback (most recent call last):
...
TypeError: <class 'sympy.assumptions.assume.UndefinedPredicate'> cannot
→ be dispatched.
```

References

[R3], [R4]

eval(args, assumptions=True)

Evaluate self(*args) under the given assumptions.

This uses only direct resolution methods, not logical inference.

handler = <dispatched AskPredicateHandler>

classmethod register(*types, **kwargs)

Register the signature to the handler.

classmethod register_many(*types, **kwargs)

Register multiple signatures to same handler.

class sympy.assumptions.assume.UndefinedPredicate(name, handlers=None)

Predicate without handler.

Explanation

This predicate is generated by using Predicate directly for construction. It does not have a handler, and evaluating this with arguments is done by SAT solver.

Examples

```
>>> from sympy import Predicate, Q
>>> Q.P = Predicate('P')
>>> Q.P.func
<class 'sympy.assumptions.assume.UndefinedPredicate'>
>>> Q.P.name
Str('P')
```

`sympy.assumptions.assume.assuming(*assumptions)`
Context manager for assumptions.

Examples

```
>>> from sympy import assuming, Q, ask
>>> from sympy.abc import x, y
>>> print(ask(Q.integer(x + y)))
None
>>> with assuming(Q.integer(x), Q.integer(y)):
...     print(ask(Q.integer(x + y)))
True
```

Refine

`sympy.assumptions.refine.refine(expr, assumptions=True)`
Simplify an expression using assumptions.

Explanation

Unlike `simplify()` (page 661) which performs structural simplification without any assumption, this function transforms the expression into the form which is only valid under certain assumptions. Note that `simplify()` is generally not done in refining process.

Refining boolean expression involves reducing it to `S.true` or `S.false`. Unlike `ask()` (page 191), the expression will not be reduced if the truth value cannot be determined.

Examples

```
>>> from sympy import refine, sqrt, Q
>>> from sympy.abc import x
>>> refine(sqrt(x**2), Q.real(x))
Abs(x)
>>> refine(sqrt(x**2), Q.positive(x))
x
```

```
>>> refine(Q.real(x), Q.positive(x))
True
>>> refine(Q.positive(x), Q.real(x))
Q.positive(x)
```

See also:

[*sympy.simplify.simplify.simplify*](#) (page 661)

Structural simplification without assumptions.

[*sympy.assumptions.ask.ask*](#) (page 191)

Query for boolean expressions using assumptions.

`sympy.assumptions.refine.refine_Pow(expr, assumptions)`

Handler for instances of Pow.

Examples

```
>>> from sympy import Q
>>> from sympy.assumptions.refine import refine_Pow
>>> from sympy.abc import x,y,z
>>> refine_Pow((-1)**x, Q.real(x))
>>> refine_Pow((-1)**x, Q.even(x))
1
>>> refine_Pow((-1)**x, Q.odd(x))
-1
```

For powers of -1, even parts of the exponent can be simplified:

```
>>> refine_Pow((-1)**(x+y), Q.even(x))
(-1)**y
>>> refine_Pow((-1)**(x+y+z), Q.odd(x) & Q.odd(z))
(-1)**y
>>> refine_Pow((-1)**(x+y+2), Q.odd(x))
(-1)**(y + 1)
>>> refine_Pow((-1)**(x+3), True)
(-1)**(x + 1)
```

`sympy.assumptions.refine.refine_abs(expr, assumptions)`

Handler for the absolute value.

Examples

```
>>> from sympy import Q, Abs
>>> from sympy.assumptions.refine import refine_abs
>>> from sympy.abc import x
>>> refine_abs(Abs(x), Q.real(x))
>>> refine_abs(Abs(x), Q.positive(x))
x
>>> refine_abs(Abs(x), Q.negative(x))
-x
```

`sympy.assumptions.refine.refine_arg(expr, assumptions)`

Handler for complex argument

Explanation

```
>>> from sympy.assumptions.refine import refine_arg
>>> from sympy import Q, arg
>>> from sympy.abc import x
>>> refine_arg(arg(x), Q.positive(x))
0
>>> refine_arg(arg(x), Q.negative(x))
pi
```

`sympy.assumptions.refine.refine_atan2(expr, assumptions)`

Handler for the atan2 function.

Examples

```
>>> from sympy import Q, atan2
>>> from sympy.assumptions.refine import refine_atan2
>>> from sympy.abc import x, y
>>> refine_atan2(atan2(y,x), Q.real(y) & Q.positive(x))
atan(y/x)
>>> refine_atan2(atan2(y,x), Q.negative(y) & Q.negative(x))
atan(y/x) - pi
>>> refine_atan2(atan2(y,x), Q.positive(y) & Q.negative(x))
atan(y/x) + pi
>>> refine_atan2(atan2(y,x), Q.zero(y) & Q.negative(x))
pi
>>> refine_atan2(atan2(y,x), Q.positive(y) & Q.zero(x))
pi/2
>>> refine_atan2(atan2(y,x), Q.negative(y) & Q.zero(x))
-pi/2
>>> refine_atan2(atan2(y,x), Q.zero(y) & Q.zero(x))
nan
```

`sympy.assumptions.refine.refine_im(expr, assumptions)`

Handler for imaginary part.

Explanation

```
>>> from sympy.assumptions.refine import refine_im
>>> from sympy import Q, im
>>> from sympy.abc import x
>>> refine_im(im(x), Q.real(x))
0
>>> refine_im(im(x), Q.imaginary(x))
-I*x
```

`sympy.assumptions.refine.refine_matrixelement(expr, assumptions)`

Handler for symmetric part.

Examples

```
>>> from sympy.assumptions.refine import refine_matrixelement
>>> from sympy import MatrixSymbol, Q
>>> X = MatrixSymbol('X', 3, 3)
>>> refine_matrixelement(X[0, 1], Q.symmetric(X))
X[0, 1]
>>> refine_matrixelement(X[1, 0], Q.symmetric(X))
X[0, 1]
```

`sympy.assumptions.refine.refine_re(expr, assumptions)`

Handler for real part.

Examples

```
>>> from sympy.assumptions.refine import refine_re
>>> from sympy import Q, re
>>> from sympy.abc import x
>>> refine_re(re(x), Q.real(x))
x
>>> refine_re(re(x), Q.imaginary(x))
0
```

`sympy.assumptions.refine.refine_sign(expr, assumptions)`

Handler for sign.

Examples

```
>>> from sympy.assumptions.refine import refine_sign
>>> from sympy import Symbol, Q, sign, im
>>> x = Symbol('x', real = True)
>>> expr = sign(x)
>>> refine_sign(expr, Q.positive(x) & Q.nonzero(x))
1
>>> refine_sign(expr, Q.negative(x) & Q.nonzero(x))
-1
>>> refine_sign(expr, Q.zero(x))
0
>>> y = Symbol('y', imaginary = True)
>>> expr = sign(y)
>>> refine_sign(expr, Q.positive(im(y)))
I
>>> refine_sign(expr, Q.negative(im(y)))
-I
```


Predicates

Common

Tautological

class sympy.assumptions.predicates.common.**IsTruePredicate**(*args, **kwargs)
Generic predicate.

Explanation

`ask(Q.is_true(x))` is true iff `x` is true. This only makes sense if `x` is a boolean object.

Examples

```
>>> from sympy import ask, Q
>>> from sympy.abc import x, y
>>> ask(Q.is_true(True))
True
```

Wrapping another applied predicate just returns the applied predicate.

```
>>> Q.is_true(Q.even(x))
Q.even(x)
```

Wrapping binary relation classes in SymPy core returns applied binary relational predicates.

```
>>> from sympy import Eq, Gt
>>> Q.is_true(Eq(x, y))
Q.eq(x, y)
>>> Q.is_true(Gt(x, y))
Q.gt(x, y)
```

Notes

This class is designed to wrap the boolean objects so that they can behave as if they are applied predicates. Consequently, wrapping another applied predicate is unnecessary and thus it just returns the argument. Also, binary relation classes in SymPy core have binary predicates to represent themselves and thus wrapping them with `Q.is_true` converts them to these applied predicates.

Handler

Multiply dispatched method: `IsTrueHandler`

Wrapper allowing to query the truth value of a boolean expression.

handler = <dispatched `IsTrueHandler`>

Commutative

class `sympy.assumptions.predicates.common.CommutativePredicate(*args, **kwargs)`
Commutative predicate.

Explanation

`ask(Q.commutative(x))` is true iff `x` commutes with any other object with respect to multiplication operation.

Handler

Multiply dispatched method: `CommutativeHandler`

Handler for key 'commutative'.

handler = <dispatched `CommutativeHandler`>

Calculus

Finite

class `sympy.assumptions.predicates.calculus.FinitePredicate(*args, **kwargs)`
Finite number predicate.

Explanation

`Q.finite(x)` is true if `x` is a number but neither an infinity nor a NaN. In other words, `ask(Q.finite(x))` is true for all numerical `x` having a bounded absolute value.

Examples

```
>>> from sympy import Q, ask, S, oo, I, zoo
>>> from sympy.abc import x
>>> ask(Q.finite(oo))
False
>>> ask(Q.finite(-oo))
False
>>> ask(Q.finite(zoo))
```

(continues on next page)

(continued from previous page)

```
False
>>> ask(Q.finite(1))
True
>>> ask(Q.finite(2 + 3*I))
True
>>> ask(Q.finite(x), Q.positive(x))
True
>>> print(ask(Q.finite(S.NaN)))
None
```

Handler

Multiply dispatched method: FiniteHandler

Handler for Q.finite. Test that an expression is bounded respect to all its variables.

References

[R7]

handler = <dispatched FiniteHandler>

Infinite

class sympy.assumptions.predicates.calculus.**InfinitePredicate**(*args, **kwargs)

Infinite number predicate.

Q.infinite(x) is true iff the absolute value of x is infinity.

Handler

Multiply dispatched method: InfiniteHandler

Handler for Q.infinite key.

handler = <dispatched InfiniteHandler>

Matrix**Symmetric**

class sympy.assumptions.predicates.matrices.**SymmetricPredicate**(*args, **kwargs)

Symmetric matrix predicate.

Explanation

`Q.symmetric(x)` is true iff `x` is a square matrix and is equal to its transpose. Every square diagonal matrix is a symmetric matrix.

Examples

```
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 2, 2)
>>> Y = MatrixSymbol('Y', 2, 3)
>>> Z = MatrixSymbol('Z', 2, 2)
>>> ask(Q.symmetric(X*Z), Q.symmetric(X) & Q.symmetric(Z))
True
>>> ask(Q.symmetric(X + Z), Q.symmetric(X) & Q.symmetric(Z))
True
>>> ask(Q.symmetric(Y))
False
```

Handler

Multiply dispatched method: `SymmetricHandler`

Handler for `Q.symmetric`.

References

[R8]

`handler = <dispatched SymmetricHandler>`

Invertible

```
class sympy.assumptions.predicates.matrices.InvertiblePredicate(*args,
                                                                    **kwargs)
```

Invertible matrix predicate.

Explanation

`Q.invertible(x)` is true iff `x` is an invertible matrix. A square matrix is called invertible only if its determinant is 0.

Examples

```
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 2, 2)
>>> Y = MatrixSymbol('Y', 2, 3)
>>> Z = MatrixSymbol('Z', 2, 2)
>>> ask(Q.invertible(X*Y), Q.invertible(X))
False
>>> ask(Q.invertible(X*Z), Q.invertible(X) & Q.invertible(Z))
True
>>> ask(Q.invertible(X), Q.fullrank(X) & Q.square(X))
True
```

Handler

Multiply dispatched method: `InvertibleHandler`

Handler for `Q.invertible`.

References

[R9]

`handler = <dispatched InvertibleHandler>`

Orthogonal

```
class sympy.assumptions.predicates.matrices.OrthogonalPredicate(*args,
                                                                **kwargs)
```

Orthogonal matrix predicate.

Explanation

`Q.orthogonal(x)` is true iff `x` is an orthogonal matrix. A square matrix `M` is an orthogonal matrix if it satisfies $M^T M = M M^T = I$ where M^T is the transpose matrix of `M` and `I` is an identity matrix. Note that an orthogonal matrix is necessarily invertible.

Examples

```
>>> from sympy import Q, ask, MatrixSymbol, Identity
>>> X = MatrixSymbol('X', 2, 2)
>>> Y = MatrixSymbol('Y', 2, 3)
>>> Z = MatrixSymbol('Z', 2, 2)
>>> ask(Q.orthogonal(Y))
False
>>> ask(Q.orthogonal(X*Z*X), Q.orthogonal(X) & Q.orthogonal(Z))
True
```

(continues on next page)

(continued from previous page)

```
>>> ask(Q.orthogonal(Identity(3)))
True
>>> ask(Q.invertible(X), Q.orthogonal(X))
True
```

Handler

Multiply dispatched method: `OrthogonalHandler`

Handler for key 'orthogonal'.

References

[R10]

`handler = <dispatched OrthogonalHandler>`

Unitary

class `sympy.assumptions.predicates.matrices.UnitaryPredicate(*args, **kwargs)`
Unitary matrix predicate.

Explanation

`Q.unitary(x)` is true iff `x` is a unitary matrix. Unitary matrix is an analogue to orthogonal matrix. A square matrix `M` with complex elements is unitary if $M^T M = M M^T = I$ where M^T is the conjugate transpose matrix of `M`.

Examples

```
>>> from sympy import Q, ask, MatrixSymbol, Identity
>>> X = MatrixSymbol('X', 2, 2)
>>> Y = MatrixSymbol('Y', 2, 3)
>>> Z = MatrixSymbol('Z', 2, 2)
>>> ask(Q.unitary(Y))
False
>>> ask(Q.unitary(X*Z*X), Q.unitary(X) & Q.unitary(Z))
True
>>> ask(Q.unitary(Identity(3)))
True
```

Handler

Multiply dispatched method: UnitaryHandler

Handler for key 'unitary'.

References

[R11]

`handler = <dispatched UnitaryHandler>`

Positive Definite

`class sympy.assumptions.predicates.matrices.PositiveDefinitePredicate(*args, **kwargs)`

Positive definite matrix predicate.

Explanation

If M is a $n \times n$ symmetric real matrix, it is said to be positive definite if $Z^T M Z$ is positive for every non-zero column vector Z of n real numbers.

Examples

```

>>> from sympy import Q, ask, MatrixSymbol, Identity
>>> X = MatrixSymbol('X', 2, 2)
>>> Y = MatrixSymbol('Y', 2, 3)
>>> Z = MatrixSymbol('Z', 2, 2)
>>> ask(Q.positive_definite(Y))
False
>>> ask(Q.positive_definite(Identity(3)))
True
>>> ask(Q.positive_definite(X + Z), Q.positive_definite(X) &
...     Q.positive_definite(Z))
True

```

Handler

Multiply dispatched method: PositiveDefiniteHandler

Handler for key 'positive_definite'.

References

[R12]

handler = <dispatched PositiveDefiniteHandler>

Upper triangular

class sympy.assumptions.predicates.matrices.UpperTriangularPredicate(*args,
**kwargs)

Upper triangular matrix predicate.

Explanation

A matrix M is called upper triangular matrix if $M_{ij} = 0$ for $i < j$.

Examples

```
>>> from sympy import Q, ask, ZeroMatrix, Identity
>>> ask(Q.upper_triangular(Identity(3)))
True
>>> ask(Q.upper_triangular(ZeroMatrix(3, 3)))
True
```

Handler

Multiply dispatched method: UpperTriangularHandler

Handler for key 'upper_triangular'.

References

[R13]

handler = <dispatched UpperTriangularHandler>

Lower triangular

class sympy.assumptions.predicates.matrices.LowerTriangularPredicate(*args,
**kwargs)

Lower triangular matrix predicate.

Explanation

A matrix M is called lower triangular matrix if $M_{ij} = 0$ for $i > j$.

Examples

```
>>> from sympy import Q, ask, ZeroMatrix, Identity
>>> ask(Q.lower_triangular(Identity(3)))
True
>>> ask(Q.lower_triangular(ZeroMatrix(3, 3)))
True
```

Handler

Multiply dispatched method: LowerTriangularHandler

Handler for key 'lower_triangular'.

References

[R14]

handler = <dispatched LowerTriangularHandler>

Diagonal

class sympy.assumptions.predicates.matrices.**DiagonalPredicate**(*args, **kwargs)
 Diagonal matrix predicate.

Explanation

$Q.diagonal(x)$ is true iff x is a diagonal matrix. A diagonal matrix is a matrix in which the entries outside the main diagonal are all zero.

Examples

```
>>> from sympy import Q, ask, MatrixSymbol, ZeroMatrix
>>> X = MatrixSymbol('X', 2, 2)
>>> ask(Q.diagonal(ZeroMatrix(3, 3)))
True
>>> ask(Q.diagonal(X), Q.lower_triangular(X) &
...     Q.upper_triangular(X))
True
```

Handler

Multiply dispatched method: `DiagonalHandler`

Handler for key 'diagonal'.

References

[R15]

`handler = <dispatched DiagonalHandler>`

Full rank

`class sympy.assumptions.predicates.matrices.FullRankPredicate(*args, **kwargs)`

Fullrank matrix predicate.

Explanation

`Q.fullrank(x)` is true iff `x` is a full rank matrix. A matrix is full rank if all rows and columns of the matrix are linearly independent. A square matrix is full rank iff its determinant is nonzero.

Examples

```
>>> from sympy import Q, ask, MatrixSymbol, ZeroMatrix, Identity
>>> X = MatrixSymbol('X', 2, 2)
>>> ask(Q.fullrank(X.T), Q.fullrank(X))
True
>>> ask(Q.fullrank(ZeroMatrix(3, 3)))
False
>>> ask(Q.fullrank(Identity(3)))
True
```

Handler

Multiply dispatched method: `FullRankHandler`

Handler for key 'fullrank'.

`handler = <dispatched FullRankHandler>`

Square

class sympy.assumptions.predicates.matrices.**SquarePredicate**(*args, **kwargs)
 Square matrix predicate.

Explanation

$Q.\text{square}(x)$ is true iff x is a square matrix. A square matrix is a matrix with the same number of rows and columns.

Examples

```
>>> from sympy import Q, ask, MatrixSymbol, ZeroMatrix, Identity
>>> X = MatrixSymbol('X', 2, 2)
>>> Y = MatrixSymbol('X', 2, 3)
>>> ask(Q.square(X))
True
>>> ask(Q.square(Y))
False
>>> ask(Q.square(ZeroMatrix(3, 3)))
True
>>> ask(Q.square(Identity(3)))
True
```

Handler

Multiply dispatched method: SquareHandler

Handler for $Q.\text{square}$.

References

[R16]

handler = <dispatched SquareHandler>

Integer elements

class sympy.assumptions.predicates.matrices.**IntegerElementsPredicate**(*args, **kwargs)
 Integer elements matrix predicate.

Explanation

`Q.integer_elements(x)` is true iff all the elements of `x` are integers.

Examples

```
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 4, 4)
>>> ask(Q.integer(X[1, 2]), Q.integer_elements(X))
True
```

Handler

Multiply dispatched method: `IntegerElementsHandler`

Handler for key `'integer_elements'`.

handler = <dispatched IntegerElementsHandler>

Real elements

```
class sympy.assumptions.predicates.matrices.RealElementsPredicate(*args,
                                                                    **kwargs)
```

Real elements matrix predicate.

Explanation

`Q.real_elements(x)` is true iff all the elements of `x` are real numbers.

Examples

```
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 4, 4)
>>> ask(Q.real(X[1, 2]), Q.real_elements(X))
True
```

Handler

Multiply dispatched method: `RealElementsHandler`

Handler for key `'real_elements'`.

handler = <dispatched RealElementsHandler>

Complex elements

class sympy.assumptions.predicates.matrices.**ComplexElementsPredicate**(*args, **kwargs)

Complex elements matrix predicate.

Explanation

$Q.\text{complex\_elements}(x)$ is true iff all the elements of x are complex numbers.

Examples

```
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 4, 4)
>>> ask(Q.complex(X[1, 2]), Q.complex_elements(X))
True
>>> ask(Q.complex_elements(X), Q.integer_elements(X))
True
```

Handler

Multiply dispatched method: ComplexElementsHandler

Handler for key 'complex_elements'.

handler = <dispatched ComplexElementsHandler>

Singular

class sympy.assumptions.predicates.matrices.**SingularPredicate**(*args, **kwargs)

Singular matrix predicate.

A matrix is singular iff the value of its determinant is 0.

Examples

```
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 4, 4)
>>> ask(Q.singular(X), Q.invertible(X))
False
>>> ask(Q.singular(X), ~Q.invertible(X))
True
```

Handler

Multiply dispatched method: SingularHandler

Predicate fore key 'singular'.

References

[R17]

handler = <dispatched SingularHandler>

Normal

class sympy.assumptions.predicates.matrices.**NormalPredicate**(*args, **kwargs)

Normal matrix predicate.

A matrix is normal if it commutes with its conjugate transpose.

Examples

```
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 4, 4)
>>> ask(Q.normal(X), Q.unitary(X))
True
```

Handler

Multiply dispatched method: NormalHandler

Predicate fore key 'normal'.

References

[R18]

handler = <dispatched NormalHandler>

Triangular

class sympy.assumptions.predicates.matrices.**TriangularPredicate**(*args,
**kwargs)

Triangular matrix predicate.

Explanation

`Q.triangular(X)` is true if `X` is one that is either lower triangular or upper triangular.

Examples

```
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 4, 4)
>>> ask(Q.triangular(X), Q.upper_triangular(X))
True
>>> ask(Q.triangular(X), Q.lower_triangular(X))
True
```

Handler

Multiply dispatched method: `TriangularHandler`

Predicate fore key 'triangular'.

References

[R19]

`handler = <dispatched TriangularHandler>`

Unit triangular

```
class sympy.assumptions.predicates.matrices.UnitTriangularPredicate(*args,
                                                                    **kwargs)
```

Unit triangular matrix predicate.

Explanation

A unit triangular matrix is a triangular matrix with 1s on the diagonal.

Examples

```
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 4, 4)
>>> ask(Q.triangular(X), Q.unit_triangular(X))
True
```

Handler

Multiply dispatched method: UnitTriangularHandler

Predicate for key 'unit_triangular'.

handler = <dispatched UnitTriangularHandler>

Number Theory

Even

class sympy.assumptions.predicates.ntheory.**EvenPredicate**(*args, **kwargs)
Even number predicate.

Explanation

`ask(Q.even(x))` is true iff x belongs to the set of even integers.

Examples

```
>>> from sympy import Q, ask, pi
>>> ask(Q.even(0))
True
>>> ask(Q.even(2))
True
>>> ask(Q.even(3))
False
>>> ask(Q.even(pi))
False
```

Handler

Multiply dispatched method: EvenHandler

Handler for key 'even'.

handler = <dispatched EvenHandler>

Odd

class sympy.assumptions.predicates.ntheory.**OddPredicate**(*args, **kwargs)
Odd number predicate.

Explanation

`ask(Q.odd(x))` is true iff x belongs to the set of odd numbers.

Examples

```

>>> from sympy import Q, ask, pi
>>> ask(Q.odd(0))
False
>>> ask(Q.odd(2))
False
>>> ask(Q.odd(3))
True
>>> ask(Q.odd(pi))
False

```

Handler

Multiply dispatched method: `OddHandler`

Handler for key 'odd'. Test that an expression represents an odd number.

handler = <dispatched `OddHandler`>

Prime

class `sympy.assumptions.predicates.ntheory.PrimePredicate(*args, **kwargs)`
 Prime number predicate.

Explanation

`ask(Q.prime(x))` is true iff x is a natural number greater than 1 that has no positive divisors other than 1 and the number itself.

Examples

```

>>> from sympy import Q, ask
>>> ask(Q.prime(0))
False
>>> ask(Q.prime(1))
False
>>> ask(Q.prime(2))
True
>>> ask(Q.prime(20))
False
>>> ask(Q.prime(-3))
False

```

Handler

Multiply dispatched method: PrimeHandler

Handler for key 'prime'. Test that an expression represents a prime number. When the expression is an exact number, the result (when True) is subject to the limitations of `isprime()` which is used to return the result.

handler = <dispatched PrimeHandler>

Composite

class sympy.assumptions.predicates.ntheory.**CompositePredicate**(*args, **kwargs)
Composite number predicate.

Explanation

`ask(Q.composite(x))` is true iff `x` is a positive integer and has at least one positive divisor other than 1 and the number itself.

Examples

```
>>> from sympy import Q, ask
>>> ask(Q.composite(0))
False
>>> ask(Q.composite(1))
False
>>> ask(Q.composite(2))
False
>>> ask(Q.composite(20))
True
```

Handler

Multiply dispatched method: CompositeHandler

Handler for key 'composite'.

handler = <dispatched CompositeHandler>

Order

Positive

class sympy.assumptions.predicates.order.**PositivePredicate**(*args, **kwargs)
 Positive real number predicate.

Explanation

`Q.positive(x)` is true iff x is real and $x > 0$, that is if x is in the interval $(0, \infty)$. In particular, infinity is not positive.

A few important facts about positive numbers:

- **Note that `Q.nonpositive` and `~Q.positive` are *not* the same** thing. `~Q.positive(x)` simply means that x is not positive, whereas `Q.nonpositive(x)` means that x is real and not positive, i.e., `Q.nonpositive(x)` is logically equivalent to `Q.negative(x)|Q.zero(x)`. So for example, `~Q.positive(I)` is true, whereas `Q.nonpositive(I)` is false.
- **See the documentation of `Q.real` for more information about** related facts.

Examples

```
>>> from sympy import Q, ask, symbols, I
>>> x = symbols('x')
>>> ask(Q.positive(x), Q.real(x) & ~Q.negative(x) & ~Q.zero(x))
True
>>> ask(Q.positive(1))
True
>>> ask(Q.nonpositive(I))
False
>>> ask(~Q.positive(I))
True
```

Handler

Multiply dispatched method: `PositiveHandler`

Handler for key 'positive'. Test that an expression is strictly greater than zero.

handler = <dispatched `PositiveHandler`>

Negative

class sympy.assumptions.predicates.order.**NegativePredicate**(*args, **kwargs)
Negative number predicate.

Explanation

`Q.negative(x)` is true iff x is a real number and $x < 0$, that is, it is in the interval $(-\infty, 0)$. Note in particular that negative infinity is not negative.

A few important facts about negative numbers:

- **Note that `Q.nonnegative` and `~Q.negative` are *not* the same** thing. `~Q.negative(x)` simply means that x is not negative, whereas `Q.nonnegative(x)` means that x is real and not negative, i.e., `Q.nonnegative(x)` is logically equivalent to `Q.zero(x) | Q.positive(x)`. So for example, `~Q.negative(I)` is true, whereas `Q.nonnegative(I)` is false.
- **See the documentation of `Q.real` for more information about** related facts.

Examples

```
>>> from sympy import Q, ask, symbols, I
>>> x = symbols('x')
>>> ask(Q.negative(x), Q.real(x) & ~Q.positive(x) & ~Q.zero(x))
True
>>> ask(Q.negative(-1))
True
>>> ask(Q.nonnegative(I))
False
>>> ask(~Q.negative(I))
True
```

Handler

Multiply dispatched method: `NegativeHandler`

Handler for `Q.negative`. Test that an expression is strictly less than zero.

handler = <dispatched `NegativeHandler`>

Zero

class sympy.assumptions.predicates.order.**ZeroPredicate**(*args, **kwargs)
Zero number predicate.

Explanation

`ask(Q.zero(x))` is true iff the value of `x` is zero.

Examples

```
>>> from sympy import ask, Q, oo, symbols
>>> x, y = symbols('x, y')
>>> ask(Q.zero(0))
True
>>> ask(Q.zero(1/oo))
True
>>> print(ask(Q.zero(0*oo)))
None
>>> ask(Q.zero(1))
False
>>> ask(Q.zero(x*y), Q.zero(x) | Q.zero(y))
True
```

Handler

Multiply dispatched method: `ZeroHandler`

Handler for key 'zero'.

handler = <dispatched ZeroHandler>

Nonzero

class sympy.assumptions.predicates.order.**NonZeroPredicate**(*args, **kwargs)
Nonzero real number predicate.

Explanation

`ask(Q.nonzero(x))` is true iff `x` is real and `x` is not zero. Note in particular that `Q.nonzero(x)` is false if `x` is not real. Use `~Q.zero(x)` if you want the negation of being zero without any real assumptions.

A few important facts about nonzero numbers:

- `Q.nonzero` is logically equivalent to `Q.positive | Q.negative`.
- **See the documentation of `Q.real` for more information about related facts.**

Examples

```
>>> from sympy import Q, ask, symbols, I, oo
>>> x = symbols('x')
>>> print(ask(Q.nonzero(x), ~Q.zero(x)))
None
>>> ask(Q.nonzero(x), Q.positive(x))
True
>>> ask(Q.nonzero(x), Q.zero(x))
False
>>> ask(Q.nonzero(0))
False
>>> ask(Q.nonzero(I))
False
>>> ask(~Q.zero(I))
True
>>> ask(Q.nonzero(oo))
False
```

Handler

Multiply dispatched method: NonZeroHandler

Handler for key 'zero'. Test that an expression is not identically zero.

handler = <dispatched NonZeroHandler>

Nonpositive

class sympy.assumptions.predicates.order.**NonPositivePredicate**(*args, **kwargs)
Nonpositive real number predicate.

Explanation

`ask(Q.nonpositive(x))` is true iff x belongs to the set of negative numbers including zero.

- **Note that `Q.nonpositive` and `~Q.positive` are *not* the same** thing. `~Q.positive(x)` simply means that x is not positive, whereas `Q.nonpositive(x)` means that x is real and not positive, i.e., `Q.nonpositive(x)` is logically equivalent to `Q.negative(x)|Q.zero(x)`. So for example, `~Q.positive(I)` is true, whereas `Q.nonpositive(I)` is false.

Examples

```
>>> from sympy import Q, ask, I
```

```
>>> ask(Q.nonpositive(-1))
True
>>> ask(Q.nonpositive(0))
True
>>> ask(Q.nonpositive(1))
False
>>> ask(Q.nonpositive(I))
False
>>> ask(Q.nonpositive(-I))
False
```

Handler

Multiply dispatched method: NonPositiveHandler

Handler for key 'nonpositive'.

handler = <dispatched NonPositiveHandler>

Nonnegative

class sympy.assumptions.predicates.order.NonNegativePredicate(*args, **kwargs)
Nonnegative real number predicate.

Explanation

`ask(Q.nonnegative(x))` is true iff x belongs to the set of positive numbers including zero.

- **Note that `Q.nonnegative` and `~Q.negative` are *not* the same** thing. `~Q.negative(x)` simply means that x is not negative, whereas `Q.nonnegative(x)` means that x is real and not negative, i.e., `Q.nonnegative(x)` is logically equivalent to `Q.zero(x) | Q.positive(x)`. So for example, `~Q.negative(I)` is true, whereas `Q.nonnegative(I)` is false.

Examples

```
>>> from sympy import Q, ask, I
>>> ask(Q.nonnegative(1))
True
>>> ask(Q.nonnegative(0))
True
>>> ask(Q.nonnegative(-1))
False
```

(continues on next page)

(continued from previous page)

```
>>> ask(Q.nonnegative(I))
False
>>> ask(Q.nonnegative(-I))
False
```

Handler

Multiply dispatched method: NonNegativeHandler

Handler for Q.nonnegative.

handler = <dispatched NonNegativeHandler>

Sets

Integer

class sympy.assumptions.predicates.sets.**IntegerPredicate**(*args, **kwargs)
Integer predicate.

Explanation

Q.integer(x) is true iff x belongs to the set of integer numbers.

Examples

```
>>> from sympy import Q, ask, S
>>> ask(Q.integer(5))
True
>>> ask(Q.integer(S(1)/2))
False
```

Handler

Multiply dispatched method: IntegerHandler

Handler for Q.integer.

Test that an expression belongs to the field of integer numbers.

References

[R20]

`handler = <dispatched IntegerHandler>`

Rational

`class sympy.assumptions.predicates.sets.RationalPredicate(*args, **kwargs)`
Rational number predicate.

Explanation

`Q.rational(x)` is true iff x belongs to the set of rational numbers.

Examples

```
>>> from sympy import ask, Q, pi, S
>>> ask(Q.rational(0))
True
>>> ask(Q.rational(S(1)/2))
True
>>> ask(Q.rational(pi))
False
```

Handler

Multiply dispatched method: `RationalHandler`

Handler for `Q.rational`.

Test that an expression belongs to the field of rational numbers.

References

[R21]

`handler = <dispatched RationalHandler>`

Irrational

class sympy.assumptions.predicates.sets.**IrrationalPredicate**(*args, **kwargs)
Irrational number predicate.

Explanation

`Q.irrational(x)` is true iff x is any real number that cannot be expressed as a ratio of integers.

Examples

```
>>> from sympy import ask, Q, pi, S, I
>>> ask(Q.irrational(0))
False
>>> ask(Q.irrational(S(1)/2))
False
>>> ask(Q.irrational(pi))
True
>>> ask(Q.irrational(I))
False
```

Handler

Multiply dispatched method: `IrrationalHandler`

Handler for `Q.irrational`.

Test that an expression is irrational numbers.

References

[R22]

`handler = <dispatched IrrationalHandler>`

Real

class sympy.assumptions.predicates.sets.**RealPredicate**(*args, **kwargs)
Real number predicate.

Explanation

`Q.real(x)` is true iff x is a real number, i.e., it is in the interval $(-\infty, \infty)$. Note that, in particular the infinities are not real. Use `Q.extended_real` if you want to consider those as well.

A few important facts about reals:

- **Every real number is positive, negative, or zero. Furthermore,** because these sets are pairwise disjoint, each real number is exactly one of those three.
- Every real number is also complex.
- Every real number is finite.
- Every real number is either rational or irrational.
- Every real number is either algebraic or transcendental.
- **The facts `Q.negative`, `Q.zero`, `Q.positive`,** `Q.nonnegative`, `Q.nonpositive`, `Q.nonzero`, `Q.integer`, `Q.rational`, and `Q.irrational` all imply `Q.real`, as do all facts that imply those facts.
- **The facts `Q.algebraic`, and `Q.transcendental` do not imply** `Q.real`; they imply `Q.complex`. An algebraic or transcendental number may or may not be real.
- **The “non” facts (i.e., `Q.nonnegative`, `Q.nonzero`,** `Q.nonpositive` and `Q.noninteger`) are not equivalent to not the fact, but rather, not the fact *and* `Q.real`. For example, `Q.nonnegative` means $\sim Q.negative \ \& \ Q.real$. So for example, 1 is not nonnegative, nonzero, or nonpositive.

Examples

```
>>> from sympy import Q, ask, symbols
>>> x = symbols('x')
>>> ask(Q.real(x), Q.positive(x))
True
>>> ask(Q.real(0))
True
```

Handler

Multiply dispatched method: `RealHandler`

Handler for `Q.real`.

Test that an expression belongs to the field of real numbers.

References

[R23]

handler = <dispatched RealHandler>

Extended real

class sympy.assumptions.predicates.sets.**ExtendedRealPredicate**(*args, **kwargs)
Extended real predicate.

Explanation

`Q.extended_real(x)` is true iff x is a real number or $\{-\infty, \infty\}$.

See documentation of `Q.real` for more information about related facts.

Examples

```
>>> from sympy import ask, Q, oo, I
>>> ask(Q.extended_real(1))
True
>>> ask(Q.extended_real(I))
False
>>> ask(Q.extended_real(oo))
True
```

Handler

Multiply dispatched method: `ExtendedRealHandler`

Handler for `Q.extended_real`.

Test that an expression belongs to the field of extended real numbers, that is real numbers union $\{\text{Infinity}, -\text{Infinity}\}$.

handler = <dispatched ExtendedRealHandler>

Hermitian

class sympy.assumptions.predicates.sets.**HermitianPredicate**(*args, **kwargs)
Hermitian predicate.

Explanation

`ask(Q.hermitian(x))` is true iff x belongs to the set of Hermitian operators.

Handler

Multiply dispatched method: `HermitianHandler`

Handler for `Q.hermitian`.

Test that an expression belongs to the field of Hermitian operators.

References

[R24]

`handler = <dispatched HermitianHandler>`

Complex

`class sympy.assumptions.predicates.sets.ComplexPredicate(*args, **kwargs)`
Complex number predicate.

Explanation

`Q.complex(x)` is true iff x belongs to the set of complex numbers. Note that every complex number is finite.

Examples

```
>>> from sympy import Q, Symbol, ask, I, oo
>>> x = Symbol('x')
>>> ask(Q.complex(0))
True
>>> ask(Q.complex(2 + 3*I))
True
>>> ask(Q.complex(oo))
False
```

Handler

Multiply dispatched method: `ComplexHandler`

Handler for `Q.complex`.

Test that an expression belongs to the field of complex numbers.

References

[R25]

`handler = <dispatched ComplexHandler>`

Imaginary

`class sympy.assumptions.predicates.sets.ImaginaryPredicate(*args, **kwargs)`
Imaginary number predicate.

Explanation

`Q.imaginary(x)` is true iff x can be written as a real number multiplied by the imaginary unit I . Please note that 0 is not considered to be an imaginary number.

Examples

```
>>> from sympy import Q, ask, I
>>> ask(Q.imaginary(3*I))
True
>>> ask(Q.imaginary(2 + 3*I))
False
>>> ask(Q.imaginary(0))
False
```

Handler

Multiply dispatched method: `ImaginaryHandler`

Handler for `Q.imaginary`.

Test that an expression belongs to the field of imaginary numbers, that is, numbers in the form $x*I$, where x is real.

References

[R26]

`handler = <dispatched ImaginaryHandler>`

Antihermitian

class sympy.assumptions.predicates.sets.**AntihermitianPredicate**(*args, **kwargs)
Antihermitian predicate.

Explanation

`Q.antihermitian(x)` is true iff x belongs to the field of antihermitian operators, i.e., operators in the form $x*I$, where x is Hermitian.

Handler

Multiply dispatched method: `AntiHermitianHandler`

Handler for `Q.antihermitian`.

Test that an expression belongs to the field of anti-Hermitian operators, that is, operators in the form $x*I$, where x is Hermitian.

References

[R27]

`handler = <dispatched AntiHermitianHandler>`

Algebraic

class sympy.assumptions.predicates.sets.**AlgebraicPredicate**(*args, **kwargs)
Algebraic number predicate.

Explanation

`Q.algebraic(x)` is true iff x belongs to the set of algebraic numbers. x is algebraic if there is some polynomial in $p(x) \in \mathbb{Q}[x]$ such that $p(x) = 0$.

Examples

```
>>> from sympy import ask, Q, sqrt, I, pi
>>> ask(Q.algebraic(sqrt(2)))
True
>>> ask(Q.algebraic(I))
True
>>> ask(Q.algebraic(pi))
False
```

Handler

Multiply dispatched method: AskAlgebraicpredicateHandler

Handler for key AskAlgebraicpredicateHandler

References

[R28]

AlgebraicHandler = <dispatched AlgebraicHandler>

handler = <dispatched AskAlgebraicpredicateHandler>

Transcendental

```
class sympy.assumptions.predicates.sets.TranscendentalPredicate(*args,
                                                                **kwargs)
```

Transcendental number predicate.

Explanation

`Q.transcendental(x)` is true iff x belongs to the set of transcendental numbers. A transcendental number is a real or complex number that is not algebraic.

Handler

Multiply dispatched method: Transcendental

Handler for `Q.transcendental` key.

handler = <dispatched Transcendental>

Performance improvements

On queries that involve symbolic coefficients, logical inference is used. Work on improving satisfiable function (`sympy.logic.inference.satisfiable`) should result in notable speed improvements.

Logic inference used in one ask could be used to speed up further queries, and current system does not take advantage of this. For example, a truth maintenance system (https://en.wikipedia.org/wiki/Truth_maintenance_system) could be implemented.

Misc

You can find more examples in the form of tests in the directory `sympy/assumptions/tests/`

Calculus

Calculus-related methods.

This module implements a method to find Euler-Lagrange Equations for given Lagrangian.

`sympy.calculus.euler.euler_equations(L, func=(), vars=())`

Find the Euler-Lagrange equations [R29] for a given Lagrangian.

Parameters

L : Expr

The Lagrangian that should be a function of the functions listed in the second argument and their derivatives.

For example, in the case of two functions $f(x, y)$, $g(x, y)$ and two independent variables x , y the Lagrangian has the form:

$$L\left(f(x, y), g(x, y), \frac{\partial f(x, y)}{\partial x}, \frac{\partial f(x, y)}{\partial y}, \frac{\partial g(x, y)}{\partial x}, \frac{\partial g(x, y)}{\partial y}, x, y\right)$$

In many cases it is not necessary to provide anything, except the Lagrangian, it will be auto-detected (and an error raised if this cannot be done).

func : Function or an iterable of Functions

The functions that the Lagrangian depends on. The Euler equations are differential equations for each of these functions.

vars : Symbol or an iterable of Symbols

The Symbols that are the independent variables of the functions.

Returns

eqns : list of Eq

The list of differential equations, one for each function.

Examples

```
>>> from sympy import euler_equations, Symbol, Function
>>> x = Function('x')
>>> t = Symbol('t')
>>> L = (x(t).diff(t))**2/2 - x(t)**2/2
>>> euler_equations(L, x(t), t)
[Eq(-x(t) - Derivative(x(t), (t, 2)), 0)]
>>> u = Function('u')
>>> x = Symbol('x')
>>> L = (u(t, x).diff(t))**2/2 - (u(t, x).diff(x))**2/2
>>> euler_equations(L, u(t, x), [t, x])
[Eq(-Derivative(u(t, x), (t, 2)) + Derivative(u(t, x), (x, 2)), 0)]
```

References

[R29]

Singularities

This module implements algorithms for finding singularities for a function and identifying types of functions.

The differential calculus methods in this module include methods to identify the following function types in the given Interval: - Increasing - Strictly Increasing - Decreasing - Strictly Decreasing - Monotonic

`sympy.calculus.singularities.is_decreasing(expression, interval=Reals,
 symbol=None)`

Return whether the function is decreasing in the given interval.

Parameters

expression : Expr

The target function which is being checked.

interval : Set, optional

The range of values in which we are testing (defaults to set of all real numbers).

symbol : Symbol, optional

The symbol present in expression which gets varied over the given range.

Returns

Boolean

True if expression is decreasing (either strictly decreasing or constant) in the given interval, False otherwise.

Examples

```
>>> from sympy import is_decreasing
>>> from sympy.abc import x, y
>>> from sympy import S, Interval, oo
>>> is_decreasing(1/(x**2 - 3*x), Interval.open(S(3)/2, 3))
True
>>> is_decreasing(1/(x**2 - 3*x), Interval.open(1.5, 3))
True
>>> is_decreasing(1/(x**2 - 3*x), Interval.Lopen(3, oo))
True
>>> is_decreasing(1/(x**2 - 3*x), Interval.Ropen(-oo, S(3)/2))
False
>>> is_decreasing(1/(x**2 - 3*x), Interval.Ropen(-oo, 1.5))
False
>>> is_decreasing(-x**2, Interval(-oo, 0))
False
>>> is_decreasing(-x**2 + y, Interval(-oo, 0), x)
False
```

`sympy.calculus.singularities.is_increasing(expression, interval=Reals, symbol=None)`

Return whether the function is increasing in the given interval.

Parameters

expression : Expr

The target function which is being checked.

interval : Set, optional

The range of values in which we are testing (defaults to set of all real numbers).

symbol : Symbol, optional

The symbol present in expression which gets varied over the given range.

Returns

Boolean

True if expression is increasing (either strictly increasing or constant) in the given interval, False otherwise.

Examples

```
>>> from sympy import is_increasing
>>> from sympy.abc import x, y
>>> from sympy import S, Interval, oo
>>> is_increasing(x**3 - 3*x**2 + 4*x, S.Reals)
True
>>> is_increasing(-x**2, Interval(-oo, 0))
True
>>> is_increasing(-x**2, Interval(0, oo))
```

(continues on next page)

(continued from previous page)

```
False
>>> is_increasing(4*x**3 - 6*x**2 - 72*x + 30, Interval(-2, 3))
False
>>> is_increasing(x**2 + y, Interval(1, 2), x)
True
```

`sympy.calculus.singularities.is_monotonic(expression, interval=Reals, symbol=None)`

Return whether the function is monotonic in the given interval.

Parameters

expression : Expr

The target function which is being checked.

interval : Set, optional

The range of values in which we are testing (defaults to set of all real numbers).

symbol : Symbol, optional

The symbol present in expression which gets varied over the given range.

Returns

Boolean

True if expression is monotonic in the given interval, False otherwise.

Raises

NotImplementedError

Monotonicity check has not been implemented for the queried function.

Examples

```
>>> from sympy import is_monotonic
>>> from sympy.abc import x, y
>>> from sympy import S, Interval, oo
>>> is_monotonic(1/(x**2 - 3*x), Interval.open(S(3)/2, 3))
True
>>> is_monotonic(1/(x**2 - 3*x), Interval.open(1.5, 3))
True
>>> is_monotonic(1/(x**2 - 3*x), Interval.Lopen(3, oo))
True
>>> is_monotonic(x**3 - 3*x**2 + 4*x, S.Reals)
True
>>> is_monotonic(-x**2, S.Reals)
False
>>> is_monotonic(x**2 + y + 1, Interval(1, 2), x)
True
```

`sympy.calculus.singularities.is_strictly_decreasing(expression, interval=Reals, symbol=None)`

Return whether the function is strictly decreasing in the given interval.

Parameters

expression : Expr

The target function which is being checked.

interval : Set, optional

The range of values in which we are testing (defaults to set of all real numbers).

symbol : Symbol, optional

The symbol present in expression which gets varied over the given range.

Returns

Boolean

True if expression is strictly decreasing in the given interval, False otherwise.

Examples

```
>>> from sympy import is_strictly_decreasing
>>> from sympy.abc import x, y
>>> from sympy import S, Interval, oo
>>> is_strictly_decreasing(1/(x**2 - 3*x), Interval.Lopen(3, oo))
True
>>> is_strictly_decreasing(1/(x**2 - 3*x), Interval.Ropen(-oo, S(3)/2))
False
>>> is_strictly_decreasing(1/(x**2 - 3*x), Interval.Ropen(-oo, 1.5))
False
>>> is_strictly_decreasing(-x**2, Interval(-oo, 0))
False
>>> is_strictly_decreasing(-x**2 + y, Interval(-oo, 0), x)
False
```

`sympy.calculus.singularities.is_strictly_increasing(expression, interval=Reals, symbol=None)`

Return whether the function is strictly increasing in the given interval.

Parameters

expression : Expr

The target function which is being checked.

interval : Set, optional

The range of values in which we are testing (defaults to set of all real numbers).

symbol : Symbol, optional

The symbol present in expression which gets varied over the given range.

Returns

Boolean

True if expression is strictly increasing in the given interval, False otherwise.

Examples

```
>>> from sympy import is_strictly_increasing
>>> from sympy.abc import x, y
>>> from sympy import Interval, oo
>>> is_strictly_increasing(4*x**3 - 6*x**2 - 72*x + 30, Interval.Ropen(-
→oo, -2))
True
>>> is_strictly_increasing(4*x**3 - 6*x**2 - 72*x + 30, Interval.Lopen(3,
→oo))
True
>>> is_strictly_increasing(4*x**3 - 6*x**2 - 72*x + 30, Interval.open(-2,
→3))
False
>>> is_strictly_increasing(-x**2, Interval(0, oo))
False
>>> is_strictly_increasing(-x**2 + y, Interval(-oo, 0), x)
False
```

`sympy.calculus.singularities.monotonicity_helper(expression, predicate,
interval=Reals, symbol=None)`

Helper function for functions checking function monotonicity.

Parameters

expression : Expr

The target function which is being checked

predicate : function

The property being tested for. The function takes in an integer and returns a boolean. The integer input is the derivative and the boolean result should be true if the property is being held, and false otherwise.

interval : Set, optional

The range of values in which we are testing, defaults to all reals.

symbol : Symbol, optional

The symbol present in expression which gets varied over the given range.

It returns a boolean indicating whether the interval in which the function's derivative satisfies given predicate is a superset of the given interval.

Returns

Boolean

True if predicate is true for all the derivatives when symbol is varied in range, False otherwise.

`sympy.calculus.singularities.singularities(expression, symbol, domain=None)`

Find singularities of a given function.

Parameters

expression : Expr

The target function in which singularities need to be found.

symbol : Symbol

The symbol over the values of which the singularity in expression is being searched for.

Returns

Set

A set of values for symbol for which expression has a singularity. An EmptySet is returned if expression has no singularities for any given value of Symbol.

Raises

NotImplementedError

Methods for determining the singularities of this function have not been developed.

Notes

This function does not find non-isolated singularities nor does it find branch points of the expression.

Currently supported functions are:

- univariate continuous (real or complex) functions

Examples

```
>>> from sympy import singularities, Symbol, log
>>> x = Symbol('x', real=True)
>>> y = Symbol('y', real=False)
>>> singularities(x**2 + x + 1, x)
EmptySet
>>> singularities(1/(x + 1), x)
{-1}
>>> singularities(1/(y**2 + 1), y)
{-I, I}
>>> singularities(1/(y**3 + 1), y)
{-1, 1/2 - sqrt(3)*I/2, 1/2 + sqrt(3)*I/2}
>>> singularities(log(x), x)
{0}
```

References

[R30]

Finite difference weights

This module implements an algorithm for efficient generation of finite difference weights for ordinary differentials of functions for derivatives from 0 (interpolation) up to arbitrary order.

The core algorithm is provided in the finite difference weight generating function (`finite_diff_weights`), and two convenience functions are provided for:

- **estimating a derivative (or interpolate) directly from a series of points**
is also provided (`apply_finite_diff`).
- **differentiating by using finite difference approximations**
(`differentiate_finite`).

`sympy.calculus.finite_diff.apply_finite_diff(order, x_list, y_list, x0=0)`

Calculates the finite difference approximation of the derivative of requested order at `x0` from points provided in `x_list` and `y_list`.

Parameters

order: int

order of derivative to approximate. 0 corresponds to interpolation.

x_list: sequence

Sequence of (unique) values for the independent variable.

y_list: sequence

The function value at corresponding values for the independent variable in `x_list`.

x0: Number or Symbol

At what value of the independent variable the derivative should be evaluated. Defaults to 0.

Returns

`sympy.core.add.Add` or `sympy.core.numbers.Number`

The finite difference expression approximating the requested derivative order at `x0`.

Examples

```
>>> from sympy import apply_finite_diff
>>> cube = lambda arg: (1.0*arg)**3
>>> xlist = range(-3,3+1)
>>> apply_finite_diff(2, xlist, map(cube, xlist), 2) - 12
-3.55271367880050e-15
```

we see that the example above only contain rounding errors. `apply_finite_diff` can also be used on more abstract objects:


```
>>> from sympy import IndexedBase, Idx
>>> x, y = map(IndexedBase, 'xy')
>>> i = Idx('i')
>>> x_list, y_list = zip(*[(x[i+j], y[i+j]) for j in range(-1,2)])
>>> apply_finite_diff(1, x_list, y_list, x[i])
((x[i + 1] - x[i])/(-x[i - 1] + x[i]) - 1)*y[i]/(x[i + 1] - x[i]) -
(x[i + 1] - x[i])*y[i - 1]/((x[i + 1] - x[i - 1])*(-x[i - 1] + x[i])) +
(-x[i - 1] + x[i])*y[i + 1]/((x[i + 1] - x[i - 1])*(x[i + 1] - x[i]))
```

Notes

Order = 0 corresponds to interpolation. Only supply so many points you think makes sense to around x_0 when extracting the derivative (the function need to be well behaved within that region). Also beware of Runge's phenomenon.

See also:

[`sympy.calculus.finite\_diff.finite\_diff\_weights`](#) (page 242)

References

Fortran 90 implementation with Python interface for numerics: [finitediff](#)

`sympy.calculus.finite_diff.differentiate_finite(expr, *symbols, points=1, x0=None, wrt=None, evaluate=False)`

Differentiate `expr` and replace Derivatives with finite differences.

Parameters

expr : expression

***symbols** : differentiate with respect to symbols

points: sequence, coefficient or undefined function, optional

see `Derivative.as_finite_difference`

x0: number or Symbol, optional

see `Derivative.as_finite_difference`

wrt: Symbol, optional

see `Derivative.as_finite_difference`

Examples

```
>>> from sympy import sin, Function, differentiate_finite
>>> from sympy.abc import x, y, h
>>> f, g = Function('f'), Function('g')
>>> differentiate_finite(f(x)*g(x), x, points=[x-h, x+h])
-f(-h + x)*g(-h + x)/(2*h) + f(h + x)*g(h + x)/(2*h)
```

`differentiate_finite` works on any expression, including the expressions with embedded derivatives:

```
>>> differentiate_finite(f(x) + sin(x), x, 2)
-2*f(x) + f(x - 1) + f(x + 1) - 2*sin(x) + sin(x - 1) + sin(x + 1)
>>> differentiate_finite(f(x, y), x, y)
f(x - 1/2, y - 1/2) - f(x - 1/2, y + 1/2) - f(x + 1/2, y - 1/2) + f(x +
→ 1/2, y + 1/2)
>>> differentiate_finite(f(x)*g(x).diff(x), x)
(-g(x) + g(x + 1))*f(x + 1/2) - (g(x) - g(x - 1))*f(x - 1/2)
```

To make finite difference with non-constant discretization step use undefined functions:

```
>>> dx = Function('dx')
>>> differentiate_finite(f(x)*g(x).diff(x), points=dx(x))
-(-g(x - dx(x)/2 - dx(x - dx(x)/2)/2)/dx(x - dx(x)/2) +
g(x - dx(x)/2 + dx(x - dx(x)/2)/2)/dx(x - dx(x)/2))*f(x - dx(x)/2)/dx(x)
→ +
(-g(x + dx(x)/2 - dx(x + dx(x)/2)/2)/dx(x + dx(x)/2) +
g(x + dx(x)/2 + dx(x + dx(x)/2)/2)/dx(x + dx(x)/2))*f(x + dx(x)/2)/dx(x)
```

`sympy.calculus.finite_diff.finite_diff_weights(order, x_list, x0=1)`

Calculates the finite difference weights for an arbitrarily spaced one-dimensional grid (`x_list`) for derivatives at `x0` of order 0, 1, ..., up to order using a recursive formula. Order of accuracy is at least `len(x_list) - order`, if `x_list` is defined correctly.

Parameters

order: int

Up to what derivative order weights should be calculated. 0 corresponds to interpolation.

x_list: sequence

Sequence of (unique) values for the independent variable. It is useful (but not necessary) to order `x_list` from nearest to furthest from `x0`; see examples below.

x0: Number or Symbol

Root or value of the independent variable for which the finite difference weights should be generated. Default is `S.One`.

Returns

list

A list of sublists, each corresponding to coefficients for increasing derivative order, and each containing lists of coefficients for increasing subsets of `x_list`.

Examples

```
>>> from sympy import finite_diff_weights, S
>>> res = finite_diff_weights(1, [-S(1)/2, S(1)/2, S(3)/2, S(5)/2], 0)
>>> res
[[[1, 0, 0, 0],
  [1/2, 1/2, 0, 0],
  [3/8, 3/4, -1/8, 0],
  [5/16, 15/16, -5/16, 1/16]],
 [[0, 0, 0, 0],
  [-1, 1, 0, 0],
  [-1, 1, 0, 0],
  [-23/24, 7/8, 1/8, -1/24]]]
>>> res[0][-1] # FD weights for 0th derivative, using full x_list
[5/16, 15/16, -5/16, 1/16]
>>> res[1][-1] # FD weights for 1st derivative
[-23/24, 7/8, 1/8, -1/24]
>>> res[1][-2] # FD weights for 1st derivative, using x_list[:-1]
[-1, 1, 0, 0]
>>> res[1][-1][0] # FD weight for 1st deriv. for x_list[0]
-23/24
>>> res[1][-1][1] # FD weight for 1st deriv. for x_list[1], etc.
7/8
```

Each sublist contains the most accurate formula at the end. Note, that in the above example `res[1][1]` is the same as `res[1][2]`. Since `res[1][2]` has an order of accuracy of `len(x_list[:3]) - order = 3 - 1 = 2`, the same is true for `res[1][1]`!

```
>>> res = finite_diff_weights(1, [S(0), S(1), -S(1), S(2), -S(2)], 0)[1]
>>> res
[[0, 0, 0, 0, 0],
 [-1, 1, 0, 0, 0],
 [0, 1/2, -1/2, 0, 0],
 [-1/2, 1, -1/3, -1/6, 0],
 [0, 2/3, -2/3, -1/12, 1/12]]
>>> res[0] # no approximation possible, using x_list[0] only
[0, 0, 0, 0, 0]
>>> res[1] # classic forward step approximation
[-1, 1, 0, 0, 0]
>>> res[2] # classic centered approximation
[0, 1/2, -1/2, 0, 0]
>>> res[3:] # higher order approximations
[[-1/2, 1, -1/3, -1/6, 0], [0, 2/3, -2/3, -1/12, 1/12]]
```

Let us compare this to a differently defined `x_list`. Pay attention to `foo[i][k]` corresponding to the gridpoint defined by `x_list[k]`.

```
>>> foo = finite_diff_weights(1, [-S(2), -S(1), S(0), S(1), S(2)], 0)[1]
>>> foo
[[0, 0, 0, 0, 0],
 [-1, 1, 0, 0, 0],
 [1/2, -2, 3/2, 0, 0],
 [1/6, -1, 1/2, 1/3, 0],
```

(continues on next page)

(continued from previous page)

```
[1/12, -2/3, 0, 2/3, -1/12]]
>>> foo[1] # not the same and of lower accuracy as res[1]!
[-1, 1, 0, 0, 0]
>>> foo[2] # classic double backward step approximation
[1/2, -2, 3/2, 0, 0]
>>> foo[4] # the same as res[4]
[1/12, -2/3, 0, 2/3, -1/12]
```

Note that, unless you plan on using approximations based on subsets of `x_list`, the order of gridpoints does not matter.

The capability to generate weights at arbitrary points can be used e.g. to minimize Runge's phenomenon by using Chebyshev nodes:

```
>>> from sympy import cos, symbols, pi, simplify
>>> N, (h, x) = 4, symbols('h x')
>>> x_list = [x+h*cos(i*pi/(N)) for i in range(N,-1,-1)] # chebyshev
↳ nodes
>>> print(x_list)
[-h + x, -sqrt(2)*h/2 + x, x, sqrt(2)*h/2 + x, h + x]
>>> mycoeffs = finite_diff_weights(1, x_list, 0)[1][4]
>>> [simplify(c) for c in mycoeffs]
[(h**3/2 + h**2*x - 3*h*x**2 - 4*x**3)/h**4,
 (-sqrt(2)*h**3 - 4*h**2*x + 3*sqrt(2)*h*x**2 + 8*x**3)/h**4,
 (6*h**2*x - 8*x**3)/h**4,
 (sqrt(2)*h**3 - 4*h**2*x - 3*sqrt(2)*h*x**2 + 8*x**3)/h**4,
 (-h**3/2 + h**2*x + 3*h*x**2 - 4*x**3)/h**4]
```

Notes

If weights for a finite difference approximation of 3rd order derivative is wanted, weights for 0th, 1st and 2nd order are calculated “for free”, so are formulae using subsets of `x_list`. This is something one can take advantage of to save computational cost. Be aware that one should define `x_list` from nearest to furthest from `x0`. If not, subsets of `x_list` will yield poorer approximations, which might not grand an order of accuracy of `len(x_list) - order`.

See also:

[`sympy.calculus.finite\_diff.apply\_finite\_diff`](#) (page 240)

References

[R31]

`sympy.calculus.util.continuous_domain(f, symbol, domain)`

Returns the intervals in the given domain for which the function is continuous. This method is limited by the ability to determine the various singularities and discontinuities of the given function.

Parameters

f : [*Expr*](#) (page 947)

The concerned function.

symbol : *Symbol* (page 976)

The variable for which the intervals are to be determined.

domain : *Interval* (page 1194)

The domain over which the continuity of the symbol has to be checked.

Returns

Interval (page 1194)

Union of all intervals where the function is continuous.

Raises

NotImplementedError

If the method to determine continuity of such a function has not yet been developed.

Examples

```
>>> from sympy import Interval, Symbol, S, tan, log, pi, sqrt
>>> from sympy.calculus.util import continuous_domain
>>> x = Symbol('x')
>>> continuous_domain(1/x, x, S.Reals)
Union(Interval.open(-oo, 0), Interval.open(0, oo))
>>> continuous_domain(tan(x), x, Interval(0, pi))
Union(Interval.Ropen(0, pi/2), Interval.Lopen(pi/2, pi))
>>> continuous_domain(sqrt(x - 2), x, Interval(-5, 5))
Interval(2, 5)
>>> continuous_domain(log(2*x - 1), x, S.Reals)
Interval.open(1/2, oo)
```

`sympy.calculus.util.function_range(f, symbol, domain)`

Finds the range of a function in a given domain. This method is limited by the ability to determine the singularities and determine limits.

Parameters

f : *Expr* (page 947)

The concerned function.

symbol : *Symbol* (page 976)

The variable for which the range of function is to be determined.

domain : *Interval* (page 1194)

The domain under which the range of the function has to be found.

Returns

Interval (page 1194)

Union of all ranges for all intervals under domain where function is continuous.

Raises

NotImplementedError

If any of the intervals, in the given domain, for which function is continuous are not finite or real, OR if the critical points of the function on the domain cannot be found.

Examples

```
>>> from sympy import Interval, Symbol, S, exp, log, pi, sqrt, sin, tan
>>> from sympy.calculus.util import function_range
>>> x = Symbol('x')
>>> function_range(sin(x), x, Interval(0, 2*pi))
Interval(-1, 1)
>>> function_range(tan(x), x, Interval(-pi/2, pi/2))
Interval(-oo, oo)
>>> function_range(1/x, x, S.Reals)
Union(Interval.open(-oo, 0), Interval.open(0, oo))
>>> function_range(exp(x), x, S.Reals)
Interval.open(0, oo)
>>> function_range(log(x), x, S.Reals)
Interval(-oo, oo)
>>> function_range(sqrt(x), x, Interval(-5, 9))
Interval(0, 3)
```

`sympy.calculus.util.is_convex(f, *syms, domain=Reals)`

Determines the convexity of the function passed in the argument.

Parameters

f : *Expr* (page 947)

The concerned function.

syms : Tuple of *Symbol* (page 976)

The variables with respect to which the convexity is to be determined.

domain : *Interval* (page 1194), optional

The domain over which the convexity of the function has to be checked. If unspecified, *S.Reals* will be the default domain.

Returns

bool

The method returns True if the function is convex otherwise it returns False.

Raises

NotImplementedError

The check for the convexity of multivariate functions is not implemented yet.

Notes

To determine concavity of a function pass $-f$ as the concerned function. To determine logarithmic convexity of a function pass $\log(f)$ as concerned function. To determine logarithmic concavity of a function pass $-\log(f)$ as concerned function.

Currently, convexity check of multivariate functions is not handled.

Examples

```
>>> from sympy import is_convex, symbols, exp, oo, Interval
>>> x = symbols('x')
>>> is_convex(exp(x), x)
True
>>> is_convex(x**3, x, domain = Interval(-1, oo))
False
>>> is_convex(1/x**2, x, domain=Interval.open(0, oo))
True
```

References

[R32], [R33], [R34], [R35], [R36]

`sympy.calculus.util.lcim(numbers)`

Returns the least common integral multiple of a list of numbers.

The numbers can be rational or irrational or a mixture of both. *None* is returned for incommensurable numbers.

Parameters

numbers : list

Numbers (rational and/or irrational) for which lcim is to be found.

Returns

number

lcim if it exists, otherwise None for incommensurable numbers.

Examples

```
>>> from sympy.calculus.util import lcim
>>> from sympy import S, pi
>>> lcim([S(1)/2, S(3)/4, S(5)/6])
15/2
>>> lcim([2*pi, 3*pi, pi, pi/2])
6*pi
>>> lcim([S(1), 2*pi])
```

`sympy.calculus.util.maximum(f, symbol, domain=Reals)`

Returns the maximum value of a function in the given domain.

Parameters

f : *Expr* (page 947)

The concerned function.

symbol : *Symbol* (page 976)

The variable for maximum value needs to be determined.

domain : *Interval* (page 1194)

The domain over which the maximum have to be checked. If unspecified, then the global maximum is returned.

Returns

number

Maximum value of the function in given domain.

Examples

```
>>> from sympy import Interval, Symbol, S, sin, cos, pi, maximum
>>> x = Symbol('x')
```

```
>>> f = -x**2 + 2*x + 5
>>> maximum(f, x, S.Reals)
6
```

```
>>> maximum(sin(x), x, Interval(-pi, pi/4))
sqrt(2)/2
```

```
>>> maximum(sin(x)*cos(x), x)
1/2
```

`sympy.calculus.util.minimum(f, symbol, domain=Reals)`

Returns the minimum value of a function in the given domain.

Parameters

f : *Expr* (page 947)

The concerned function.

symbol : *Symbol* (page 976)

The variable for minimum value needs to be determined.

domain : *Interval* (page 1194)

The domain over which the minimum have to be checked. If unspecified, then the global minimum is returned.

Returns

number

Minimum value of the function in the given domain.

Examples

```
>>> from sympy import Interval, Symbol, S, sin, cos, minimum
>>> x = Symbol('x')
```

```
>>> f = x**2 + 2*x + 5
>>> minimum(f, x, S.Reals)
4
```

```
>>> minimum(sin(x), x, Interval(2, 3))
sin(3)
```

```
>>> minimum(sin(x)*cos(x), x)
-1/2
```

`sympy.calculus.util.not_empty_in(finset_intersection, *syms)`

Finds the domain of the functions in `finset_intersection` in which the `finite_set` is not-empty

Parameters

finset_intersection : Intersection of FiniteSet

The unevaluated intersection of FiniteSet containing real-valued functions with Union of Sets

syms : Tuple of symbols

Symbol for which domain is to be found

Raises

NotImplementedError

The algorithms to find the non-emptiness of the given FiniteSet are not yet implemented.

ValueError

The input is not valid.

RuntimeError

It is a bug, please report it to the github issue tracker (<https://github.com/sympy/sympy/issues>).

Examples

```
>>> from sympy import FiniteSet, Interval, not_empty_in, oo
>>> from sympy.abc import x
>>> not_empty_in(FiniteSet(x/2).intersect(Interval(0, 1)), x)
Interval(0, 2)
>>> not_empty_in(FiniteSet(x, x**2).intersect(Interval(1, 2)), x)
Union(Interval(1, 2), Interval(-sqrt(2), -1))
>>> not_empty_in(FiniteSet(x**2/(x + 2)).intersect(Interval(1, oo)), x)
Union(Interval.Lopen(-2, -1), Interval(2, oo))
```

`sympy.calculus.util.periodicity(f, symbol, check=False)`

Tests the given function for periodicity in the given symbol.

Parameters

f : *Expr* (page 947).

The concerned function.

symbol : *Symbol* (page 976)

The variable for which the period is to be determined.

check : bool, optional

The flag to verify whether the value being returned is a period or not.

Returns

period

The period of the function is returned. None is returned when the function is aperiodic or has a complex period. The value of 0 is returned as the period of a constant function.

Raises

NotImplementedError

The value of the period computed cannot be verified.

Notes

Currently, we do not support functions with a complex period. The period of functions having complex periodic values such as `exp`, `sinh` is evaluated to `None`.

The value returned might not be the “fundamental” period of the given function i.e. it may not be the smallest periodic value of the function.

The verification of the period through the `check` flag is not reliable due to internal simplification of the given expression. Hence, it is set to `False` by default.

Examples

```
>>> from sympy import periodicity, Symbol, sin, cos, tan, exp
>>> x = Symbol('x')
>>> f = sin(x) + sin(2*x) + sin(3*x)
>>> periodicity(f, x)
2*pi
>>> periodicity(sin(x)*cos(x), x)
pi
>>> periodicity(exp(tan(2*x) - 1), x)
pi/2
>>> periodicity(sin(4*x)**cos(2*x), x)
pi
>>> periodicity(exp(x), x)
```

`sympy.calculus.util.stationary_points(f, symbol, domain=Reals)`

Returns the stationary points of a function (where derivative of the function is 0) in the given domain.

Parameters**f** : [Expr](#) (page 947)

The concerned function.

symbol : [Symbol](#) (page 976)

The variable for which the stationary points are to be determined.

domain : [Interval](#) (page 1194)The domain over which the stationary points have to be checked. If unspecified, `S.Reals` will be the default domain.**Returns**

Set

A set of stationary points for the function. If there are no stationary point, an [EmptySet](#) (page 1202) is returned.**Examples**

```
>>> from sympy import Interval, Symbol, S, sin, pi, pprint, stationary_
      points
>>> x = Symbol('x')
```

```
>>> stationary_points(1/x, x, S.Reals)
EmptySet
```

```
>>> pprint(stationary_points(sin(x), x), use_unicode=False)
      pi              3*pi
{2*n*pi + -- | n in Integers} U {2*n*pi + ---- | n in Integers}
      2              2
```

```
>>> stationary_points(sin(x), x, Interval(0, 4*pi))
{pi/2, 3*pi/2, 5*pi/2, 7*pi/2}
```

Combinatorics**Contents****Partitions****class** `sympy.combinatorics.partitions.Partition(*partition)`

This class represents an abstract partition.

A partition is a set of disjoint sets whose union equals a given set.

See also:[sympy.utilities.iterables.partitions](#) (page 2089), [sympy.utilities.iterables.multiset_partitions](#) (page 2084)**property** `RGS`

Returns the “restricted growth string” of the partition.

Explanation

The RGS is returned as a list of indices, L , where $L[i]$ indicates the block in which element i appears. For example, in a partition of 3 elements (a, b, c) into 2 blocks ($\{c\}, \{a, b\}$) the RGS is $[1, 1, 0]$: “ a ” is in block 1, “ b ” is in block 1 and “ c ” is in block 0.

Examples

```
>>> from sympy.combinatorics import Partition
>>> a = Partition([1, 2], [3], [4, 5])
>>> a.members
(1, 2, 3, 4, 5)
>>> a.RGS
(0, 0, 1, 2, 2)
>>> a + 1
Partition({3}, {4}, {5}, {1, 2})
>>> _.RGS
(0, 0, 1, 2, 3)
```

classmethod `from_rgs(rgs, elements)`

Creates a set partition from a restricted growth string.

Explanation

The indices given in *rgs* are assumed to be the index of the element as given in *elements* *as provided* (the elements are not sorted by this routine). Block numbering starts from 0. If any block was not referenced in *rgs* an error will be raised.

Examples

```
>>> from sympy.combinatorics import Partition
>>> Partition.from_rgs([0, 1, 2, 0, 1], list('abcde'))
Partition({c}, {a, d}, {b, e})
>>> Partition.from_rgs([0, 1, 2, 0, 1], list('cbead'))
Partition({e}, {a, c}, {b, d})
>>> a = Partition([1, 4], [2], [3, 5])
>>> Partition.from_rgs(a.RGS, a.members)
Partition({2}, {1, 4}, {3, 5})
```

property `partition`

Return partition as a sorted list of lists.

Examples

```
>>> from sympy.combinatorics import Partition
>>> Partition([1], [2, 3]).partition
[[1], [2, 3]]
```

property rank

Gets the rank of a partition.

Examples

```
>>> from sympy.combinatorics import Partition
>>> a = Partition([1, 2], [3], [4, 5])
>>> a.rank
13
```

sort_key(*order=None*)

Return a canonical key that can be used for sorting.

Ordering is based on the size and sorted elements of the partition and ties are broken with the rank.

Examples

```
>>> from sympy import default_sort_key
>>> from sympy.combinatorics import Partition
>>> from sympy.abc import x
>>> a = Partition([1, 2])
>>> b = Partition([3, 4])
>>> c = Partition([1, x])
>>> d = Partition(list(range(4)))
>>> l = [d, b, a + 1, a, c]
>>> l.sort(key=default_sort_key); l
[Partition({1, 2}), Partition({1}, {2}), Partition({1, x}), Partition(
↪ {3, 4}), Partition({0, 1, 2, 3})]
```

class sympy.combinatorics.partitions.IntegerPartition(*partition, integer=None*)

This class represents an integer partition.

Explanation

In number theory and combinatorics, a partition of a positive integer, n , also called an integer partition, is a way of writing n as a list of positive integers that sum to n . Two partitions that differ only in the order of summands are considered to be the same partition; if order matters then the partitions are referred to as compositions. For example, 4 has five partitions: [4], [3, 1], [2, 2], [2, 1, 1], and [1, 1, 1, 1]; the compositions [1, 2, 1] and [1, 1, 2] are the same as partition [2, 1, 1].

See also:

sympy.utilities.iterables.partitions (page 2089), *sympy.utilities.iterables.multiset_partitions* (page 2084)

References

[R50]

as_dict()

Return the partition as a dictionary whose keys are the partition integers and the values are the multiplicity of that integer.

Examples

```
>>> from sympy.combinatorics.partitions import IntegerPartition
>>> IntegerPartition([1]*3 + [2] + [3]*4).as_dict()
{1: 3, 2: 1, 3: 4}
```

as_ferrers(char='#')

Prints the ferrer diagram of a partition.

Examples

```
>>> from sympy.combinatorics.partitions import IntegerPartition
>>> print(IntegerPartition([1, 1, 5]).as_ferrers())
#####
#
#
```

property conjugate

Computes the conjugate partition of itself.

Examples

```
>>> from sympy.combinatorics.partitions import IntegerPartition
>>> a = IntegerPartition([6, 3, 3, 2, 1])
>>> a.conjugate
[5, 4, 3, 1, 1, 1]
```

next_lex()

Return the next partition of the integer, n , in lexical order, wrapping around to $[n]$ if the partition is $[1, \dots, 1]$.

Examples

```
>>> from sympy.combinatorics.partitions import IntegerPartition
>>> p = IntegerPartition([3, 1])
>>> print(p.next_lex())
[4]
>>> p.partition < p.next_lex().partition
True
```

`prev_lex()`

Return the previous partition of the integer, n , in lexical order, wrapping around to $[1, \dots, 1]$ if the partition is $[n]$.

Examples

```
>>> from sympy.combinatorics.partitions import IntegerPartition
>>> p = IntegerPartition([4])
>>> print(p.prev_lex())
[3, 1]
>>> p.partition > p.prev_lex().partition
True
```

`sympy.combinatorics.partitions.random_integer_partition( $n$ ,  $seed=None$ )`

Generates a random integer partition summing to n as a list of reverse-sorted integers.

Examples

```
>>> from sympy.combinatorics.partitions import random_integer_partition
```

For the following, a seed is given so a known value can be shown; in practice, the seed would not be given.

```
>>> random_integer_partition(100, seed=[1, 1, 12, 1, 2, 1, 85, 1])
[85, 12, 2, 1]
>>> random_integer_partition(10, seed=[1, 2, 3, 1, 5, 1])
[5, 3, 1, 1]
>>> random_integer_partition(1)
[1]
```

`sympy.combinatorics.partitions.RGS_generalized( $m$ )`

Computes the $m + 1$ generalized unrestricted growth strings and returns them as rows in matrix.

Examples

```
>>> from sympy.combinatorics.partitions import RGS_generalized
>>> RGS_generalized(6)
Matrix([
[ 1, 1, 1, 1, 1, 1, 1],
[ 1, 2, 3, 4, 5, 6, 0],
[ 2, 5, 10, 17, 26, 0, 0],
[ 5, 15, 37, 77, 0, 0, 0],
[ 15, 52, 151, 0, 0, 0, 0],
[ 52, 203, 0, 0, 0, 0, 0],
[203, 0, 0, 0, 0, 0, 0]])
```

`sympy.combinatorics.partitions.RGS_enum(m)`

`RGS_enum` computes the total number of restricted growth strings possible for a superset of size *m*.

Examples

```
>>> from sympy.combinatorics.partitions import RGS_enum
>>> from sympy.combinatorics import Partition
>>> RGS_enum(4)
15
>>> RGS_enum(5)
52
>>> RGS_enum(6)
203
```

We can check that the enumeration is correct by actually generating the partitions. Here, the 15 partitions of 4 items are generated:

```
>>> a = Partition(list(range(4)))
>>> s = set()
>>> for i in range(20):
...     s.add(a)
...     a += 1
...
>>> assert len(s) == 15
```

`sympy.combinatorics.partitions.RGS_unrank(rank, m)`

Gives the unranked restricted growth string for a given superset size.

Examples

```
>>> from sympy.combinatorics.partitions import RGS_unrank
>>> RGS_unrank(14, 4)
[0, 1, 2, 3]
>>> RGS_unrank(0, 4)
[0, 0, 0, 0]
```

`sympy.combinatorics.partitions.RGS_rank(rgs)`
 Computes the rank of a restricted growth string.

Examples

```
>>> from sympy.combinatorics.partitions import RGS_rank, RGS_unrank
>>> RGS_rank([0, 1, 2, 1, 3])
42
>>> RGS_rank(RGS_unrank(4, 7))
4
```

Permutations

class `sympy.combinatorics.permutations.Permutation(*args, size=None, **kwargs)`

A permutation, alternatively known as an ‘arrangement number’ or ‘ordering’ is an arrangement of the elements of an ordered list into a one-to-one mapping with itself. The permutation of a given arrangement is given by indicating the positions of the elements after re-arrangement [R69]. For example, if one started with elements `[x, y, a, b]` (in that order) and they were reordered as `[x, y, b, a]` then the permutation would be `[0, 1, 3, 2]`. Notice that (in SymPy) the first element is always referred to as 0 and the permutation uses the indices of the elements in the original ordering, not the elements (`a, b, ...`) themselves.

```
>>> from sympy.combinatorics import Permutation
>>> from sympy import init_printing
>>> init_printing(perm_cyclic=False, pretty_print=False)
```

Permutations Notation

Permutations are commonly represented in disjoint cycle or array forms.

Array Notation And 2-line Form

In the 2-line form, the elements and their final positions are shown as a matrix with 2 rows:

```
[0 1 2 ... n-1] [p(0) p(1) p(2) ... p(n-1)]
```

Since the first line is always $\text{range}(n)$, where n is the size of p , it is sufficient to represent the permutation by the second line, referred to as the “array form” of the permutation. This is entered in brackets as the argument to the `Permutation` class:

```
>>> p = Permutation([0, 2, 1]); p
Permutation([0, 2, 1])
```

Given i in $\text{range}(p.\text{size})$, the permutation maps i to i^p

```
>>> [i^p for i in range(p.size)]
[0, 2, 1]
```

The composite of two permutations $p*q$ means first apply p , then q , so $i^{(p*q)} = (i^p)^q$ which is $i^p{}^q$ according to Python precedence rules:

```
>>> q = Permutation([2, 1, 0])
>>> [i^p^q for i in range(3)]
[2, 0, 1]
>>> [i^(p*q) for i in range(3)]
[2, 0, 1]
```

One can use also the notation $p(i) = i^p$, but then the composition rule is $(p*q)(i) = q(p(i))$, not $p(q(i))$:

```
>>> [(p*q)(i) for i in range(p.size)]
[2, 0, 1]
>>> [q(p(i)) for i in range(p.size)]
[2, 0, 1]
>>> [p(q(i)) for i in range(p.size)]
[1, 2, 0]
```

Disjoint Cycle Notation

In disjoint cycle notation, only the elements that have shifted are indicated.

For example, $[1, 3, 2, 0]$ can be represented as $(0, 1, 3)(2)$. This can be understood from the 2 line format of the given permutation. In the 2-line form, $[0\ 1\ 2\ 3]\ [1\ 3\ 2\ 0]$

The element in the 0th position is 1, so $0 \rightarrow 1$. The element in the 1st position is three, so $1 \rightarrow 3$. And the element in the third position is again 0, so $3 \rightarrow 0$. Thus, $0 \rightarrow 1 \rightarrow 3 \rightarrow 0$, and $2 \rightarrow 2$. Thus, this can be represented as 2 cycles: $(0, 1, 3)(2)$. In common notation, singular cycles are not explicitly written as they can be inferred implicitly.

Only the relative ordering of elements in a cycle matter:

```
>>> Permutation(1,2,3) == Permutation(2,3,1) == Permutation(3,1,2)
True
```

The disjoint cycle notation is convenient when representing permutations that have several cycles in them:

```
>>> Permutation(1, 2)(3, 5) == Permutation([[1, 2], [3, 5]])
True
```

It also provides some economy in entry when computing products of permutations that are written in disjoint cycle notation:

```
>>> Permutation(1, 2)(1, 3)(2, 3)
Permutation([0, 3, 2, 1])
>>> == Permutation([[1, 2]])*Permutation([[1, 3]])*Permutation([[2, 3]])
True
```

Caution: when the cycles have common elements between them then the order in which the permutations are applied matters. This module applies the permutations from *left to right*.

```
>>> Permutation(1, 2)(2, 3) == Permutation([(1, 2), (2, 3)])
True
>>> Permutation(1, 2)(2, 3).list()
[0, 3, 1, 2]
```

In the above case, (1,2) is computed before (2,3). As 0 -> 0, 0 -> 0, element in position 0 is 0. As 1 -> 2, 2 -> 3, element in position 1 is 3. As 2 -> 1, 1 -> 1, element in position 2 is 1. As 3 -> 3, 3 -> 2, element in position 3 is 2.

If the first and second elements had been swapped first, followed by the swapping of the second and third, the result would have been [0, 2, 3, 1]. If, you want to apply the cycles in the conventional right to left order, call the function with arguments in reverse order as demonstrated below:

```
>>> Permutation([(1, 2), (2, 3)][::-1]).list()
[0, 2, 3, 1]
```

Entering a singleton in a permutation is a way to indicate the size of the permutation. The size keyword can also be used.

Array-form entry:

```
>>> Permutation([[1, 2], [9]])
Permutation([0, 2, 1], size=10)
>>> Permutation([[1, 2]], size=10)
Permutation([0, 2, 1], size=10)
```

Cyclic-form entry:

```
>>> Permutation(1, 2, size=10)
Permutation([0, 2, 1], size=10)
>>> Permutation(9)(1, 2)
Permutation([0, 2, 1], size=10)
```

Caution: no singleton containing an element larger than the largest in any previous cycle can be entered. This is an important difference in how Permutation and Cycle

handle the `__call__` syntax. A singleton argument at the start of a `Permutation` performs instantiation of the `Permutation` and is permitted:

```
>>> Permutation(5)
Permutation([], size=6)
```

A singleton entered after instantiation is a call to the permutation - a function call - and if the argument is out of range it will trigger an error. For this reason, it is better to start the cycle with the singleton:

The following fails because there is no element 3:

```
>>> Permutation(1, 2)(3)
Traceback (most recent call last):
...
IndexError: list index out of range
```

This is ok: only the call to an out of range singleton is prohibited; otherwise the permutation autosizes:

```
>>> Permutation(3)(1, 2)
Permutation([0, 2, 1, 3])
>>> Permutation(1, 2)(3, 4) == Permutation(3, 4)(1, 2)
True
```

Equality Testing

The array forms must be the same in order for permutations to be equal:

```
>>> Permutation([1, 0, 2, 3]) == Permutation([1, 0])
False
```

Identity Permutation

The identity permutation is a permutation in which no element is out of place. It can be entered in a variety of ways. All the following create an identity permutation of size 4:

```
>>> I = Permutation([0, 1, 2, 3])
>>> all(p == I for p in [
...     Permutation(3),
...     Permutation(range(4)),
...     Permutation([], size=4),
...     Permutation(size=4)])
True
```

Watch out for entering the range *inside* a set of brackets (which is cycle notation):

```
>>> I == Permutation([range(4)])
False
```

Permutation Printing

There are a few things to note about how Permutations are printed.

Deprecated since version 1.6: Configuring Permutation printing by setting `Permutation.print_cyclic` is deprecated. Users should use the `perm_cyclic` flag to the printers, as described below.

1) If you prefer one form (array or cycle) over another, you can set `init_printing` with the `perm_cyclic` flag.

```
>>> from sympy import init_printing
>>> p = Permutation(1, 2)(4, 5)(3, 4)
>>> p
Permutation([0, 2, 1, 4, 5, 3])
```

```
>>> init_printing(perm_cyclic=True, pretty_print=False)
>>> p
(1 2)(3 4 5)
```

2) Regardless of the setting, a list of elements in the array for cyclic form can be obtained and either of those can be copied and supplied as the argument to `Permutation`:

```
>>> p.array_form
[0, 2, 1, 4, 5, 3]
>>> p.cyclic_form
[[1, 2], [3, 4, 5]]
>>> Permutation(_) == p
True
```

3) Printing is economical in that as little as possible is printed while retaining all information about the size of the permutation:

```
>>> init_printing(perm_cyclic=False, pretty_print=False)
>>> Permutation([1, 0, 2, 3])
Permutation([1, 0, 2, 3])
>>> Permutation([1, 0, 2, 3], size=20)
Permutation([1, 0], size=20)
>>> Permutation([1, 0, 2, 4, 3, 5, 6], size=20)
Permutation([1, 0, 2, 4, 3], size=20)
```

```
>>> p = Permutation([1, 0, 2, 3])
>>> init_printing(perm_cyclic=True, pretty_print=False)
>>> p
(3)(0 1)
>>> init_printing(perm_cyclic=False, pretty_print=False)
```

The 2 was not printed but it is still there as can be seen with the `array_form` and `size` methods:

```
>>> p.array_form
[1, 0, 2, 3]
>>> p.size
4
```

Short Introduction To Other Methods

The permutation can act as a bijective function, telling what element is located at a given position

```
>>> q = Permutation([5, 2, 3, 4, 1, 0])
>>> q.array_form[1] # the hard way
2
>>> q(1) # the easy way
2
>>> {i: q(i) for i in range(q.size)} # showing the bijection
{0: 5, 1: 2, 2: 3, 3: 4, 4: 1, 5: 0}
```

The full cyclic form (including singletons) can be obtained:

```
>>> p.full_cyclic_form
[[0, 1], [2], [3]]
```

Any permutation can be factored into transpositions of pairs of elements:

```
>>> Permutation([[1, 2], [3, 4, 5]]).transpositions()
[(1, 2), (3, 5), (3, 4)]
>>> Permutation.rmul(*[Permutation([ti], size=6) for ti in _]).cyclic_
→ form
[[1, 2], [3, 4, 5]]
```

The number of permutations on a set of n elements is given by $n!$ and is called the cardinality.

```
>>> p.size
4
>>> p.cardinality
24
```

A given permutation has a rank among all the possible permutations of the same elements, but what that rank is depends on how the permutations are enumerated. (There are a number of different methods of doing so.) The lexicographic rank is given by the rank method and this rank is used to increment a permutation with addition/subtraction:

```
>>> p.rank()
6
>>> p + 1
Permutation([1, 0, 3, 2])
>>> p.next_lex()
Permutation([1, 0, 3, 2])
>>> _.rank()
7
>>> p.unrank_lex(p.size, rank=7)
Permutation([1, 0, 3, 2])
```

The product of two permutations p and q is defined as their composition as functions, $(p*q)(i) = q(p(i))$ [R73].

```
>>> p = Permutation([1, 0, 2, 3])
>>> q = Permutation([2, 3, 1, 0])
>>> list(q*p)
[2, 3, 0, 1]
>>> list(p*q)
[3, 2, 1, 0]
>>> [q(p(i)) for i in range(p.size)]
[3, 2, 1, 0]
```

The permutation can be ‘applied’ to any list-like object, not only Permutations:

```
>>> p(['zero', 'one', 'four', 'two'])
['one', 'zero', 'four', 'two']
>>> p('zo42')
['o', 'z', '4', '2']
```

If you have a list of arbitrary elements, the corresponding permutation can be found with the `from_sequence` method:

```
>>> Permutation.from_sequence('SymPy')
Permutation([1, 3, 2, 0, 4])
```

Checking If A Permutation Is Contained In A Group

Generally if you have a group of permutations G on n symbols, and you’re checking if a permutation on less than n symbols is part of that group, the check will fail.

Here is an example for $n=5$ and we check if the cycle $(1,2,3)$ is in G :

```
>>> from sympy import init_printing
>>> init_printing(perm_cyclic=True, pretty_print=False)
>>> from sympy.combinatorics import Cycle, Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> G = PermutationGroup(Cycle(2, 3)(4, 5), Cycle(1, 2, 3, 4, 5))
>>> p1 = Permutation(Cycle(2, 5, 3))
>>> p2 = Permutation(Cycle(1, 2, 3))
>>> a1 = Permutation(Cycle(1, 2, 3).list(6))
>>> a2 = Permutation(Cycle(1, 2, 3)(5))
>>> a3 = Permutation(Cycle(1, 2, 3), size=6)
>>> for p in [p1, p2, a1, a2, a3]: p, G.contains(p)
((2 5 3), True)
((1 2 3), False)
((5)(1 2 3), True)
((5)(1 2 3), True)
((5)(1 2 3), True)
```

The check for $p2$ above will fail.

Checking if $p1$ is in G works because SymPy knows G is a group on 5 symbols, and $p1$ is also on 5 symbols (its largest element is 5).

For $a1$, the `.list(6)` call will extend the permutation to 5 symbols, so the test will work as well. In the case of $a2$ the permutation is being extended to 5 symbols by using a singleton, and in the case of $a3$ it’s extended through the constructor argument `size=6`.

There is another way to do this, which is to tell the `contains` method that the number of symbols the group is on does not need to match perfectly the number of symbols for the permutation:

```
>>> G.contains(p2,strict=False)
True
```

This can be via the `strict` argument to the `contains` method, and SymPy will try to extend the permutation on its own and then perform the containment check.

See also:

[Cycle](#) (page 284)

References

[R68], [R69], [R70], [R71], [R72], [R73], [R74]

`apply(i)`

Apply the permutation to an expression.

Parameters

i : Expr

It should be an integer between 0 and $n - 1$ where n is the size of the permutation.

If it is a symbol or a symbolic expression that can have integer values, an `AppliedPermutation` object will be returned which can represent an unevaluated function.

Notes

Any permutation can be defined as a bijective function $\sigma : \{0, 1, \dots, n-1\} \rightarrow \{0, 1, \dots, n-1\}$ where n denotes the size of the permutation.

The definition may even be extended for any set with distinctive elements, such that the permutation can even be applied for real numbers or such, however, it is not implemented for now for computational reasons and the integrity with the group theory module.

This function is similar to the `__call__` magic, however, `__call__` magic already has some other applications like permuting an array or attaching new cycles, which would not always be mathematically consistent.

This also guarantees that the return type is a SymPy integer, which guarantees the safety to use assumptions.

property `array_form`

Return a copy of the attribute `_array_form` Examples =====

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([[2, 0], [3, 1]])
>>> p.array_form
[2, 3, 0, 1]
>>> Permutation([[2, 0, 3, 1]]).array_form
```

(continues on next page)

(continued from previous page)

```
[3, 2, 0, 1]
>>> Permutation([2, 0, 3, 1]).array_form
[2, 0, 3, 1]
>>> Permutation([[1, 2], [4, 5]]).array_form
[0, 2, 1, 3, 5, 4]
```

ascents()

Returns the positions of ascents in a permutation, ie, the location where $p[i] < p[i+1]$

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([4, 0, 1, 3, 2])
>>> p.ascents()
[1, 2]
```

See also:

[descents](#) (page 267), [inversions](#) (page 272), [min](#) (page 276), [max](#) (page 275)

atoms()

Returns all the elements of a permutation

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation([0, 1, 2, 3, 4, 5]).atoms()
{0, 1, 2, 3, 4, 5}
>>> Permutation([[0, 1], [2, 3], [4, 5]]).atoms()
{0, 1, 2, 3, 4, 5}
```

property cardinality

Returns the number of all possible permutations.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.cardinality
24
```

See also:

[length](#) (page 275), [order](#) (page 277), [rank](#) (page 279), [size](#) (page 282)

commutator(x)

Return the commutator of `self` and `x`: $\sim x \sim \text{self} * x * \text{self}$

If f and g are part of a group, G , then the commutator of f and g is the group identity iff f and g commute, i.e. $fg == gf$.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> from sympy import init_printing
>>> init_printing(perm_cyclic=False, pretty_print=False)
>>> p = Permutation([0, 2, 3, 1])
>>> x = Permutation([2, 0, 3, 1])
>>> c = p.commutator(x); c
Permutation([2, 1, 3, 0])
>>> c == ~x*~p*x*p
True
```

```
>>> I = Permutation(3)
>>> p = [I + i for i in range(6)]
>>> for i in range(len(p)):
...     for j in range(len(p)):
...         c = p[i].commutator(p[j])
...         if p[i]*p[j] == p[j]*p[i]:
...             assert c == I
...         else:
...             assert c != I
... 
```

References

[R75]

commutes_with(*other*)

Checks if the elements are commuting.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> a = Permutation([1, 4, 3, 0, 2, 5])
>>> b = Permutation([0, 1, 2, 3, 4, 5])
>>> a.commutates_with(b)
True
>>> b = Permutation([2, 3, 5, 4, 1, 0])
>>> a.commutates_with(b)
False
```

property cycle_structure

Return the cycle structure of the permutation as a dictionary indicating the multiplicity of each cycle length.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation(3).cycle_structure
{1: 4}
>>> Permutation(0, 4, 3)(1, 2)(5, 6).cycle_structure
{2: 2, 3: 1}
```

property cycles

Returns the number of cycles contained in the permutation (including singletons).

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation([0, 1, 2]).cycles
3
>>> Permutation([0, 1, 2]).full_cyclic_form
[[0], [1], [2]]
>>> Permutation(0, 1)(2, 3).cycles
2
```

See also:

[*sympy.functions.combinatorial.numbers.stirling*](#) (page 444)

property cyclic_form

This is used to convert to the cyclic notation from the canonical notation. Singletons are omitted.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([0, 3, 1, 2])
>>> p.cyclic_form
[[1, 3, 2]]
>>> Permutation([1, 0, 2, 4, 3, 5]).cyclic_form
[[0, 1], [3, 4]]
```

See also:

[*array_form*](#) (page 264), [*full_cyclic_form*](#) (page 268)

descents()

Returns the positions of descents in a permutation, ie, the location where $p[i] > p[i+1]$

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([4, 0, 1, 3, 2])
>>> p.descents()
[0, 3]
```

See also:

[*ascents*](#) (page 265), [*inversions*](#) (page 272), [*min*](#) (page 276), [*max*](#) (page 275)

classmethod `from_inversion_vector(inversion)`

Calculates the permutation from the inversion vector.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> from sympy import init_printing
>>> init_printing(perm_cyclic=False, pretty_print=False)
>>> Permutation.from_inversion_vector([3, 2, 1, 0, 0])
Permutation([3, 2, 1, 0, 4, 5])
```

classmethod `from_sequence(i, key=None)`

Return the permutation needed to obtain *i* from the sorted elements of *i*. If custom sorting is desired, a key can be given.

Examples

```
>>> from sympy.combinatorics import Permutation
```

```
>>> Permutation.from_sequence('SymPy')
(4)(0 1 3)
>>> _(sorted("SymPy"))
['S', 'y', 'm', 'P', 'y']
>>> Permutation.from_sequence('SymPy', key=lambda x: x.lower())
(4)(0 2)(1 3)
```

property `full_cyclic_form`

Return permutation in cyclic form including singletons.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation([0, 2, 1]).full_cyclic_form
[[0], [1, 2]]
```

get_adjacency_distance(*other*)

Computes the adjacency distance between two permutations.

Explanation

This metric counts the number of times a pair i, j of jobs is adjacent in both p and p' . If n_{adj} is this quantity then the adjacency distance is $n - n_{\text{adj}} - 1$ [1]

[1] Reeves, Colin R. Landscapes, Operators and Heuristic search, Annals of Operational Research, 86, pp 473-490. (1999)

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([0, 3, 1, 2, 4])
>>> q = Permutation.josephus(4, 5, 2)
>>> p.get_adjacency_distance(q)
3
>>> r = Permutation([0, 2, 1, 4, 3])
>>> p.get_adjacency_distance(r)
4
```

See also:

[`get\_precedence\_matrix`](#) (page 270), [`get\_precedence\_distance`](#) (page 270), [`get\_adjacency\_matrix`](#) (page 269)

`get_adjacency_matrix()`

Computes the adjacency matrix of a permutation.

Explanation

If job i is adjacent to job j in a permutation p then we set $m[i, j] = 1$ where m is the adjacency matrix of p .

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation.josephus(3, 6, 1)
>>> p.get_adjacency_matrix()
Matrix([
[0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 1, 0],
[0, 0, 0, 0, 0, 1],
[0, 1, 0, 0, 0, 0],
[1, 0, 0, 0, 0, 0],
[0, 0, 0, 1, 0, 0]])
>>> q = Permutation([0, 1, 2, 3])
>>> q.get_adjacency_matrix()
Matrix([
[0, 1, 0, 0],
[0, 0, 1, 0],
[0, 0, 0, 1],
[0, 0, 0, 0]])
```

See also:

[`get\_precedence\_matrix`](#) (page 270), [`get\_precedence\_distance`](#) (page 270),
[`get\_adjacency\_distance`](#) (page 268)

get_positional_distance(*other*)

Computes the positional distance between two permutations.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([0, 3, 1, 2, 4])
>>> q = Permutation.josephus(4, 5, 2)
>>> r = Permutation([3, 1, 4, 0, 2])
>>> p.get_positional_distance(q)
12
>>> p.get_positional_distance(r)
12
```

See also:

[`get\_precedence\_distance`](#) (page 270), [`get\_adjacency\_distance`](#) (page 268)

get_precedence_distance(*other*)

Computes the precedence distance between two permutations.

Explanation

Suppose p and p' represent n jobs. The precedence metric counts the number of times a job j is preceded by job i in both p and p' . This metric is commutative.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([2, 0, 4, 3, 1])
>>> q = Permutation([3, 1, 2, 4, 0])
>>> p.get_precedence_distance(q)
7
>>> q.get_precedence_distance(p)
7
```

See also:

[`get\_precedence\_matrix`](#) (page 270), [`get\_adjacency\_matrix`](#) (page 269),
[`get\_adjacency\_distance`](#) (page 268)

get_precedence_matrix()

Gets the precedence matrix. This is used for computing the distance between two permutations.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> from sympy import init_printing
>>> init_printing(perm_cyclic=False, pretty_print=False)
>>> p = Permutation.josephus(3, 6, 1)
>>> p
Permutation([2, 5, 3, 1, 4, 0])
>>> p.get_precedence_matrix()
Matrix([
[0, 0, 0, 0, 0, 0],
[1, 0, 0, 0, 1, 0],
[1, 1, 0, 1, 1, 1],
[1, 1, 0, 0, 1, 0],
[1, 0, 0, 0, 0, 0],
[1, 1, 0, 1, 1, 0]])
```

See also:

[*get_precedence_distance*](#) (page 270), [*get_adjacency_matrix*](#) (page 269), [*get_adjacency_distance*](#) (page 268)

index()

Returns the index of a permutation.

The index of a permutation is the sum of all subscripts j such that $p[j]$ is greater than $p[j+1]$.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([3, 0, 2, 1, 4])
>>> p.index()
2
```

inversion_vector()

Return the inversion vector of the permutation.

The inversion vector consists of elements whose value indicates the number of elements in the permutation that are lesser than it and lie on its right hand side.

The inversion vector is the same as the Lehmer encoding of a permutation.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([4, 8, 0, 7, 1, 5, 3, 6, 2])
>>> p.inversion_vector()
[4, 7, 0, 5, 0, 2, 1, 1]
>>> p = Permutation([3, 2, 1, 0])
>>> p.inversion_vector()
[3, 2, 1]
```

The inversion vector increases lexicographically with the rank of the permutation, the i -th element cycling through $0..i$.

```
>>> p = Permutation(2)
>>> while p:
...     print('%s %s %s' % (p, p.inversion_vector(), p.rank()))
...     p = p.next_lex()
(2) [0, 0] 0
(1 2) [0, 1] 1
(2)(0 1) [1, 0] 2
(0 1 2) [1, 1] 3
(0 2 1) [2, 0] 4
(0 2) [2, 1] 5
```

See also:

[*from_inversion_vector*](#) (page 268)

inversions()

Computes the number of inversions of a permutation.

Explanation

An inversion is where $i > j$ but $p[i] < p[j]$.

For small length of p , it iterates over all i and j values and calculates the number of inversions. For large length of p , it uses a variation of merge sort to calculate the number of inversions.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([0, 1, 2, 3, 4, 5])
>>> p.inversions()
0
>>> Permutation([3, 2, 1, 0]).inversions()
6
```

See also:

[*descents*](#) (page 267), [*ascents*](#) (page 265), [*min*](#) (page 276), [*max*](#) (page 275)

References

[R76]

property is_Empty

Checks to see if the permutation is a set with zero elements

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation([]).is_Empty
True
>>> Permutation([0]).is_Empty
False
```

See also:

[*is_Singleton*](#) (page 273)

property `is_Identity`

Returns True if the Permutation is an identity permutation.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([])
>>> p.is_Identity
True
>>> p = Permutation([[0], [1], [2]])
>>> p.is_Identity
True
>>> p = Permutation([0, 1, 2])
>>> p.is_Identity
True
>>> p = Permutation([0, 2, 1])
>>> p.is_Identity
False
```

See also:

[*order*](#) (page 277)

property `is_Singleton`

Checks to see if the permutation contains only one number and is thus the only possible permutation of this set of numbers

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation([0]).is_Singleton
True
>>> Permutation([0, 1]).is_Singleton
False
```

See also:

[*is_Empty*](#) (page 272)

property `is_even`

Checks if a permutation is even.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.is_even
True
>>> p = Permutation([3, 2, 1, 0])
>>> p.is_even
True
```

See also:

[*is_odd*](#) (page 274)

property `is_odd`

Checks if a permutation is odd.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.is_odd
False
>>> p = Permutation([3, 2, 0, 1])
>>> p.is_odd
True
```

See also:

[*is_even*](#) (page 273)

classmethod `josephus(m, n, s=1)`

Return as a permutation the shuffling of `range(n)` using the Josephus scheme in which every `m`-th item is selected until all have been chosen. The returned permutation has elements listed by the order in which they were selected.

The parameter `s` stops the selection process when there are `s` items remaining and these are selected by continuing the selection, counting by 1 rather than by `m`.

Consider selecting every 3rd item from 6 until only 2 remain:

choices	chosen
=====	=====
012345	
01 345	2
01 34	25
01 4	253
0 4	2531
0	25314
	253140

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.josephus(3, 6, 2).array_form
[2, 5, 3, 1, 4, 0]
```

References

[R77], [R78], [R79]

`length()`

Returns the number of integers moved by a permutation.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation([0, 3, 2, 1]).length()
2
>>> Permutation([[0, 1], [2, 3]]).length()
4
```

See also:

[min](#) (page 276), [max](#) (page 275), [support](#) (page 282), [cardinality](#) (page 265), [order](#) (page 277), [rank](#) (page 279), [size](#) (page 282)

`list(size=None)`

Return the permutation as an explicit list, possibly trimming unmoved elements if size is less than the maximum element in the permutation; if this is desired, setting size=-1 will guarantee such trimming.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation(2, 3)(4, 5)
>>> p.list()
[0, 1, 3, 2, 5, 4]
>>> p.list(10)
[0, 1, 3, 2, 5, 4, 6, 7, 8, 9]
```

Passing a length too small will trim trailing, unchanged elements in the permutation:

```
>>> Permutation(2, 4)(1, 2, 4).list(-1)
[0, 2, 1]
>>> Permutation(3).list(-1)
[]
```

`max()`

The maximum element moved by the permutation.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([1, 0, 2, 3, 4])
>>> p.max()
1
```

See also:

[min](#) (page 276), [descents](#) (page 267), [ascents](#) (page 265), [inversions](#) (page 272)

min()

The minimum element moved by the permutation.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([0, 1, 4, 3, 2])
>>> p.min()
2
```

See also:

[max](#) (page 275), [descents](#) (page 267), [ascents](#) (page 265), [inversions](#) (page 272)

mul_inv(other)

other*~self, self and other have `_array_form`

next_lex()

Returns the next permutation in lexicographical order. If self is the last permutation in lexicographical order it returns None. See [4] section 2.4.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([2, 3, 1, 0])
>>> p = Permutation([2, 3, 1, 0]); p.rank()
17
>>> p = p.next_lex(); p.rank()
18
```

See also:

[rank](#) (page 279), [unrank_lex](#) (page 283)

next_nonlex()

Returns the next permutation in nonlex order [3]. If self is the last permutation in this order it returns None.

Examples

```

>>> from sympy.combinatorics import Permutation
>>> from sympy import init_printing
>>> init_printing(perm_cyclic=False, pretty_print=False)
>>> p = Permutation([2, 0, 3, 1]); p.rank_nonlex()
5
>>> p = p.next_nonlex(); p
Permutation([3, 0, 1, 2])
>>> p.rank_nonlex()
6

```

See also:

[*rank_nonlex*](#) (page 279), [*unrank_nonlex*](#) (page 284)

`next_trotterjohnson()`

Returns the next permutation in Trotter-Johnson order. If self is the last permutation it returns None. See [4] section 2.4. If it is desired to generate all such permutations, they can be generated in order more quickly with the `generate_bell` function.

Examples

```

>>> from sympy.combinatorics import Permutation
>>> from sympy import init_printing
>>> init_printing(perm_cyclic=False, pretty_print=False)
>>> p = Permutation([3, 0, 2, 1])
>>> p.rank_trotterjohnson()
4
>>> p = p.next_trotterjohnson(); p
Permutation([0, 3, 2, 1])
>>> p.rank_trotterjohnson()
5

```

See also:

[*rank_trotterjohnson*](#) (page 279), [*unrank_trotterjohnson*](#) (page 284), [*sympy.utilities.iterables.generate_bell*](#) (page 2074)

`order()`

Computes the order of a permutation.

When the permutation is raised to the power of its order it equals the identity permutation.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> from sympy import init_printing
>>> init_printing(perm_cyclic=False, pretty_print=False)
>>> p = Permutation([3, 1, 5, 2, 4, 0])
>>> p.order()
4
>>> (p**(p.order()))
Permutation([], size=6)
```

See also:

[*identity*](#) (page 309), [*cardinality*](#) (page 265), [*length*](#) (page 275), [*rank*](#) (page 279), [*size*](#) (page 282)

parity()

Computes the parity of a permutation.

Explanation

The parity of a permutation reflects the parity of the number of inversions in the permutation, i.e., the number of pairs of x and y such that $x > y$ but $p[x] < p[y]$.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.parity()
0
>>> p = Permutation([3, 2, 0, 1])
>>> p.parity()
1
```

See also:

[*_af_parity*](#) (page 286)

classmethod random(*n*)

Generates a random permutation of length n .

Uses the underlying Python pseudo-random number generator.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.random(2) in (Permutation([1, 0]), Permutation([0,
↪ 1]))
True
```

rank()

Returns the lexicographic rank of the permutation.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.rank()
0
>>> p = Permutation([3, 2, 1, 0])
>>> p.rank()
23
```

See also:

[next_lex](#) (page 276), [unrank_lex](#) (page 283), [cardinality](#) (page 265), [length](#) (page 275), [order](#) (page 277), [size](#) (page 282)

rank_nonlex(inv_perm=None)

This is a linear time ranking algorithm that does not enforce lexicographic order [3].

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.rank_nonlex()
23
```

See also:

[next_nonlex](#) (page 276), [unrank_nonlex](#) (page 284)

rank_trotterjohnson()

Returns the Trotter Johnson rank, which we get from the minimal change algorithm. See [4] section 2.4.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.rank_trotterjohnson()
0
>>> p = Permutation([0, 2, 1, 3])
>>> p.rank_trotterjohnson()
7
```

See also:

[*unrank_trotterjohnson*](#) (page 284), [*next_trotterjohnson*](#) (page 277)

`resize(n)`

Resize the permutation to the new size *n*.

Parameters

n : int

The new size of the permutation.

Raises

ValueError

If the permutation cannot be resized to the given size. This may only happen when resized to a smaller size than the original.

Examples

```
>>> from sympy.combinatorics import Permutation
```

Increasing the size of a permutation:

```
>>> p = Permutation(0, 1, 2)
>>> p = p.resize(5)
>>> p
(4)(0 1 2)
```

Decreasing the size of the permutation:

```
>>> p = p.resize(4)
>>> p
(3)(0 1 2)
```

If resizing to the specific size breaks the cycles:

```
>>> p.resize(2)
Traceback (most recent call last):
...
ValueError: The permutation cannot be resized to 2 because the
cycle (0, 1, 2) may break.
```


static `rmul(*args)`

Return product of Permutations `[a, b, c, ...]` as the Permutation whose `i`th value is `a(b(c(i)))`.

`a, b, c, ...` can be Permutation objects or tuples.

Examples

```
>>> from sympy.combinatorics import Permutation
```

```
>>> a, b = [1, 0, 2], [0, 2, 1]
>>> a = Permutation(a); b = Permutation(b)
>>> list(Permutation.rmul(a, b))
[1, 2, 0]
>>> [a(b(i)) for i in range(3)]
[1, 2, 0]
```

This handles the operands in reverse order compared to the `*` operator:

```
>>> a = Permutation(a); b = Permutation(b)
>>> list(a*b)
[2, 0, 1]
>>> [b(a(i)) for i in range(3)]
[2, 0, 1]
```

Notes

All items in the sequence will be parsed by Permutation as necessary as long as the first item is a Permutation:

```
>>> Permutation.rmul(a, [0, 2, 1]) == Permutation.rmul(a, b)
True
```

The reverse order of arguments will raise a `TypeError`.

classmethod `rmul_with_af(*args)`

same as `rmul`, but the elements of `args` are Permutation objects which have `_array_form`

runs()

Returns the runs of a permutation.

An ascending sequence in a permutation is called a run [5].

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([2, 5, 7, 3, 6, 0, 1, 4, 8])
>>> p.runs()
[[2, 5, 7], [3, 6], [0, 1, 4, 8]]
>>> q = Permutation([1, 3, 2, 0])
>>> q.runs()
[[1, 3], [2], [0]]
```

signature()

Gives the signature of the permutation needed to place the elements of the permutation in canonical order.

The signature is calculated as $(-1)^{\text{number of inversions}}$

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([0, 1, 2])
>>> p.inversions()
0
>>> p.signature()
1
>>> q = Permutation([0, 2, 1])
>>> q.inversions()
1
>>> q.signature()
-1
```

See also:

[*inversions*](#) (page 272)

property size

Returns the number of elements in the permutation.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation([[3, 2], [0, 1]]).size
4
```

See also:

[*cardinality*](#) (page 265), [*length*](#) (page 275), [*order*](#) (page 277), [*rank*](#) (page 279)

support()

Return the elements in permutation, P , for which $P[i] \neq i$.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([[3, 2], [0, 1], [4]])
>>> p.array_form
[1, 0, 3, 2, 4]
>>> p.support()
[0, 1, 2, 3]
```

transpositions()

Return the permutation decomposed into a list of transpositions.

Explanation

It is always possible to express a permutation as the product of transpositions, see [1]

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([[1, 2, 3], [0, 4, 5, 6, 7]])
>>> t = p.transpositions()
>>> t
[(0, 7), (0, 6), (0, 5), (0, 4), (1, 3), (1, 2)]
>>> print(''.join(str(c) for c in t))
(0, 7)(0, 6)(0, 5)(0, 4)(1, 3)(1, 2)
>>> Permutation.rmul(*[Permutation([ti], size=p.size) for ti in t])
↪ == p
True
```

References

[R80]

classmethod unrank_lex(size, rank)

Lexicographic permutation unranking.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> from sympy import init_printing
>>> init_printing(perm_cyclic=False, pretty_print=False)
>>> a = Permutation.unrank_lex(5, 10)
>>> a.rank()
10
>>> a
Permutation([0, 2, 4, 1, 3])
```

See also:

[rank](#) (page 279), [next_lex](#) (page 276)

classmethod `unrank_nonlex(n, r)`

This is a linear time unranking algorithm that does not respect lexicographic order [3].

Examples

```
>>> from sympy.combinatorics import Permutation
>>> from sympy import init_printing
>>> init_printing(perm_cyclic=False, pretty_print=False)
>>> Permutation.unrank_nonlex(4, 5)
Permutation([2, 0, 3, 1])
>>> Permutation.unrank_nonlex(4, -1)
Permutation([0, 1, 2, 3])
```

See also:

[next_nonlex](#) (page 276), [rank_nonlex](#) (page 279)

classmethod `unrank_trotterjohnson(size, rank)`

Trotter Johnson permutation unranking. See [4] section 2.4.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> from sympy import init_printing
>>> init_printing(perm_cyclic=False, pretty_print=False)
>>> Permutation.unrank_trotterjohnson(5, 10)
Permutation([0, 3, 1, 2, 4])
```

See also:

[rank_trotterjohnson](#) (page 279), [next_trotterjohnson](#) (page 277)

class `sympy.combinatorics.permutations.Cycle(*args)`

Wrapper around dict which provides the functionality of a disjoint cycle.

Explanation

A cycle shows the rule to use to move subsets of elements to obtain a permutation. The Cycle class is more flexible than Permutation in that 1) all elements need not be present in order to investigate how multiple cycles act in sequence and 2) it can contain singletons:

```
>>> from sympy.combinatorics.permutations import Perm, Cycle
```

A Cycle will automatically parse a cycle given as a tuple on the rhs:

```
>>> Cycle(1, 2)(2, 3)
(1 3 2)
```

The identity cycle, `Cycle()`, can be used to start a product:

```
>>> Cycle()(1, 2)(2, 3)
(1 3 2)
```

The array form of a `Cycle` can be obtained by calling the `list` method (or passing it to the `list` function) and all elements from 0 will be shown:

```
>>> a = Cycle(1, 2)
>>> a.list()
[0, 2, 1]
>>> list(a)
[0, 2, 1]
```

If a larger (or smaller) range is desired use the `list` method and provide the desired size - but the `Cycle` cannot be truncated to a size smaller than the largest element that is out of place:

```
>>> b = Cycle(2, 4)(1, 2)(3, 1, 4)(1, 3)
>>> b.list()
[0, 2, 1, 3, 4]
>>> b.list(b.size + 1)
[0, 2, 1, 3, 4, 5]
>>> b.list(-1)
[0, 2, 1]
```

Singletons are not shown when printing with one exception: the largest element is always shown - as a singleton if necessary:

```
>>> Cycle(1, 4, 10)(4, 5)
(1 5 4 10)
>>> Cycle(1, 2)(4)(5)(10)
(1 2)(10)
```

The array form can be used to instantiate a `Permutation` so other properties of the permutation can be investigated:

```
>>> Perm(Cycle(1, 2)(3, 4).list()).transpositions()
[(1, 2), (3, 4)]
```

Notes

The underlying structure of the `Cycle` is a dictionary and although the `__iter__` method has been redefined to give the array form of the cycle, the underlying dictionary items are still available with the such methods as `items()`:

```
>>> list(Cycle(1, 2).items())
[(1, 2), (2, 1)]
```

See also:

[Permutation](#) (page 257)

list(size=None)

Return the cycles as an explicit list starting from 0 up to the greater of the largest value in the cycles and size.

Truncation of trailing unmoved items will occur when size is less than the maximum element in the cycle; if this is desired, setting size=-1 will guarantee such trimming.

Examples

```
>>> from sympy.combinatorics import Cycle
>>> p = Cycle(2, 3)(4, 5)
>>> p.list()
[0, 1, 3, 2, 5, 4]
>>> p.list(10)
[0, 1, 3, 2, 5, 4, 6, 7, 8, 9]
```

Passing a length too small will trim trailing, unchanged elements in the permutation:

```
>>> Cycle(2, 4)(1, 2, 4).list(-1)
[0, 2, 1]
```

`sympy.combinatorics.permutations._af_parity(pi)`

Computes the parity of a permutation in array form.

Explanation

The parity of a permutation reflects the parity of the number of inversions in the permutation, i.e., the number of pairs of x and y such that $x > y$ but $p[x] < p[y]$.

Examples

```
>>> from sympy.combinatorics.permutations import _af_parity
>>> _af_parity([0, 1, 2, 3])
0
>>> _af_parity([3, 2, 0, 1])
1
```

See also:

[Permutation](#) (page 257)

Generators

`generators.symmetric()`

Generates the symmetric group of order n , S_n .

Examples

```
>>> from sympy.combinatorics.generators import symmetric
>>> list(symmetric(3))
[(2), (1 2), (2)(0 1), (0 1 2), (0 2 1), (0 2)]
```

`generators.cyclic()`

Generates the cyclic group of order n , C_n .

Examples

```
>>> from sympy.combinatorics.generators import cyclic
>>> list(cyclic(5))
[(4), (0 1 2 3 4), (0 2 4 1 3),
 (0 3 1 4 2), (0 4 3 2 1)]
```

See also:

[*dihedral*](#) (page 287)

`generators.alternating()`

Generates the alternating group of order n , A_n .

Examples

```
>>> from sympy.combinatorics.generators import alternating
>>> list(alternating(3))
[(2), (0 1 2), (0 2 1)]
```

`generators.dihedral()`

Generates the dihedral group of order $2n$, D_n .

The result is given as a subgroup of S_n , except for the special cases $n=1$ (the group S_2) and $n=2$ (the Klein 4-group) where that's not possible and embeddings in S_2 and S_4 respectively are given.

Examples

```
>>> from sympy.combinatorics.generators import dihedral
>>> list(dihedral(3))
[(2), (0 2), (0 1 2), (1 2), (0 2 1), (2)(0 1)]
```

See also:

[*cyclic*](#) (page 287)

Permutation Groups

```
class sympy.combinatorics.perm_groups.PermutationGroup(*args, dups=True,
                                                         **kwargs)
```

The class defining a Permutation group.

Explanation

`PermutationGroup([p1, p2, ..., pn])` returns the permutation group generated by the list of permutations. This group can be supplied to `Polyhedron` if one desires to decorate the elements to which the indices of the permutation refer.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> from sympy.combinatorics import Polyhedron
```

The permutations corresponding to motion of the front, right and bottom face of a 2×2 Rubik's cube are defined:

```
>>> F = Permutation(2, 19, 21, 8)(3, 17, 20, 10)(4, 6, 7, 5)
>>> R = Permutation(1, 5, 21, 14)(3, 7, 23, 12)(8, 10, 11, 9)
>>> D = Permutation(6, 18, 14, 10)(7, 19, 15, 11)(20, 22, 23, 21)
```

These are passed as permutations to `PermutationGroup`:

```
>>> G = PermutationGroup(F, R, D)
>>> G.order()
3674160
```

The group can be supplied to a `Polyhedron` in order to track the objects being moved. An example involving the 2×2 Rubik's cube is given there, but here is a simple demonstration:

```
>>> a = Permutation(2, 1)
>>> b = Permutation(1, 0)
>>> G = PermutationGroup(a, b)
>>> P = Polyhedron(list('ABC'), pgroup=G)
>>> P.corners
(A, B, C)
```

(continues on next page)

(continued from previous page)

```
>>> P.rotate(0) # apply permutation 0
>>> P.corners
(A, C, B)
>>> P.reset()
>>> P.corners
(A, B, C)
```

Or one can make a permutation as a product of selected permutations and apply them to an iterable directly:

```
>>> P10 = G.make_perm([0, 1])
>>> P10('ABC')
['C', 'A', 'B']
```

See also:

sympy.combinatorics.polyhedron.Polyhedron (page 332), *sympy.combinatorics.permutations.Permutation* (page 257)

References

[R51], [R52], [R53], [R54], [R55], [R56], [R57], [R58], [R59],^{10, 11, 12, 13, 14}

__contains__(i)

Return True if *i* is contained in PermutationGroup.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> p = Permutation(1, 2, 3)
>>> Permutation(3) in PermutationGroup(p)
True
```

__mul__(other)

Return the direct product of two permutation groups as a permutation group.

¹⁰ https://en.wikipedia.org/wiki/Centralizer_and_normalizer

¹¹ http://groupprops.subwiki.org/wiki/Derived_subgroup

¹² https://en.wikipedia.org/wiki/Nilpotent_group

¹³ <http://www.math.colostate.edu/~hulpke/CGT/cgtnotes.pdf>

¹⁴ <https://www.gap-system.org/Manuals/doc/ref/manual.pdf>

Explanation

This implementation realizes the direct product by shifting the index set for the generators of the second group: so if we have G acting on n_1 points and H acting on n_2 points, $G*H$ acts on $n_1 + n_2$ points.

Examples

```
>>> from sympy.combinatorics.named_groups import CyclicGroup
>>> G = CyclicGroup(5)
>>> H = G*G
>>> H
PermutationGroup([
    (9)(0 1 2 3 4),
    (5 6 7 8 9)])
>>> H.order()
25
```

static `__new__(cls, *args, dups=True, **kwargs)`

The default constructor. Accepts Cycle and Permutation forms. Removes duplicates unless `dups` keyword is False.

__weakref__

list of weak references to the object (if defined)

__coset_representative(g, H)

Return the representative of Hg from the transversal that would be computed by `self.coset_transversal(H)`.

classmethod `_distinct_primes_lemma(primes)`

Subroutine to test if there is only one cyclic group for the order.

property `_elements`

Returns all the elements of the permutation group as a list

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> p = PermutationGroup(Permutation(1, 3), Permutation(1, 2))
>>> p._elements
[(3), (3)(1 2), (1 3), (2 3), (1 2 3), (1 3 2)]
```

__eval_is_alt_sym_monte_carlo($eps=0.05, perms=None$)

A test using monte-carlo algorithm.

Parameters

eps : float, optional

The criterion for the incorrect False return.

perms : list[Permutation], optional

If explicitly given, it tests over the given candidates for testing.

If None, it randomly computes `N_eps` and chooses `N_eps` sample of the permutation from the group.

See also:

[`\_check\_cycles\_alt\_sym`](#) (page 357)

`_eval_is_alt_sym_naive(only_sym=False, only_alt=False)`

A naive test using the group order.

`_p_elements_group(p)`

For an abelian p -group, return the subgroup consisting of all elements of order p (and the identity)

`_random_pr_init(r, n, _random_prec_n=None)`

Initialize random generators for the product replacement algorithm.

Explanation

The implementation uses a modification of the original product replacement algorithm due to Leedham-Green, as described in [1], pp. 69-71; also, see [2], pp. 27-29 for a detailed theoretical analysis of the original product replacement algorithm, and [4].

The product replacement algorithm is used for producing random, uniformly distributed elements of a group G with a set of generators S . For the initialization `_random_pr_init`, a list R of $\max\{r, |S|\}$ group generators is created as the attribute `G._random_gens`, repeating elements of S if necessary, and the identity element of G is appended to R - we shall refer to this last element as the accumulator. Then the function `random_pr()` is called n times, randomizing the list R while preserving the generation of G by R . The function `random_pr()` itself takes two random elements g, h among all elements of R but the accumulator and replaces g with a randomly chosen element from $\{gh, g(h), hg, (h)g\}$. Then the accumulator is multiplied by whatever g was replaced by. The new value of the accumulator is then returned by `random_pr()`.

The elements returned will eventually (for n large enough) become uniformly distributed across G ([5]). For practical purposes however, the values $n = 50$, $r = 11$ are suggested in [1].

Notes

THIS FUNCTION HAS SIDE EFFECTS: it changes the attribute `self._random_gens`

See also:

[`random\_pr`](#) (page 324)

`_sylow_alt_sym(p)`

Return a p -Sylow subgroup of a symmetric or an alternating group.

Explanation

The algorithm for this is hinted at in [1], Chapter 4, Exercise 4.

For $\text{Sym}(n)$ with $n = p^i$, the idea is as follows. Partition the interval $[0..n-1]$ into p equal parts, each of length p^{i-1} : $[0..p^{i-1}-1]$, $[p^{i-1}..2*p^{i-1}-1]$... $[(p-1)*p^{i-1}..p^i-1]$. Find a p -Sylow subgroup of $\text{Sym}(p^{i-1})$ (treated as a subgroup of self) acting on each of the parts. Call the subgroups $P_1, P_2...P_p$. The generators for the subgroups $P_2...P_p$ can be obtained from those of P_1 by applying a “shifting” permutation to them, that is, a permutation mapping $[0..p^{i-1}-1]$ to the second part (the other parts are obtained by using the shift multiple times). The union of this permutation and the generators of P_1 is a p -Sylow subgroup of self .

For n not equal to a power of p , partition $[0..n-1]$ in accordance with how n would be written in base p . E.g. for $p=2$ and $n=11$, $11 = 2^3 + 2^2 + 1$ so the partition is $[[0..7], [8..9], \{10\}]$. To generate a p -Sylow subgroup, take the union of the generators for each of the parts. For the above example, $\{(0\ 1), (0\ 2)(1\ 3), (0\ 4), (1\ 5)(2\ 7)\}$ from the first part, $\{(8\ 9)\}$ from the second part and nothing from the third. This gives 4 generators in total, and the subgroup they generate is p -Sylow.

Alternating groups are treated the same except when $p=2$. In this case, $(0\ 1)(s\ s+1)$ should be added for an appropriate s (the start of a part) for each part in the partitions.

See also:

[`syLOW\_subgroup`](#) (page 330), [`is\_Alt\_sym`](#) (page 310)

[`\_union\_find\_merge`](#)(*first*, *second*, *ranks*, *parents*, *not_rep*)

Merges two classes in a union-find data structure.

Explanation

Used in the implementation of Atkinson’s algorithm as suggested in [1], pp. 83-87. The class merging process uses union by rank as an optimization. ([7])

Notes

THIS FUNCTION HAS SIDE EFFECTS: the list of class representatives, *parents*, the list of class sizes, *ranks*, and the list of elements that are not representatives, *not_rep*, are changed due to class merging.

See also:

[`minimal\_block`](#) (page 319), [`\_union\_find\_rep`](#) (page 293)

References

[R60], [R66]

`_union_find_rep(num, parents)`

Find representative of a class in a union-find data structure.

Explanation

Used in the implementation of Atkinson's algorithm as suggested in [1], pp. 83-87. After the representative of the class to which `num` belongs is found, path compression is performed as an optimization ([7]).

Notes

THIS FUNCTION HAS SIDE EFFECTS: the list of class representatives, `parents`, is altered due to path compression.

See also:

[`minimal\_block`](#) (page 319), [`\_union\_find\_merge`](#) (page 292)

References

[R62], [R68]

`_verify(K, phi, z, alpha)`

Return a list of relators `rels` in generators `gens_h` that are mapped to `H`. generators by `phi` so that given a finite presentation $\langle \text{gens}_k \mid \text{rels}_k \rangle$ of K on a subset of `gens_h` $\langle \text{gens}_h \mid \text{rels}_k + \text{rels} \rangle$ is a finite presentation of H .

Explanation

H should be generated by the union of K .generators and z (a single generator), and $H.\text{stabilizer}(\alpha) == K$; `phi` is a canonical injection from a free group into a permutation group containing H .

The algorithm is described in [1], Chapter 6.

Examples

```
>>> from sympy.combinatorics import free_group, Permutation,
↳ PermutationGroup
>>> from sympy.combinatorics.homomorphisms import homomorphism
>>> from sympy.combinatorics.fp_groups import FpGroup
```

```
>>> H = PermutationGroup(Permutation(0, 2), Permutation(1, 5))
>>> K = PermutationGroup(Permutation(5)(0, 2))
>>> F = free_group("x_0 x_1")[0]
>>> gens = F.generators
>>> phi = homomorphism(F, H, F.generators, H.generators)
>>> rels_k = [gens[0]**2] # relators for presentation of K
>>> z = Permutation(1, 5)
>>> check, rels_h = H._verify(K, phi, z, 1)
>>> check
True
>>> rels = rels_k + rels_h
>>> G = FpGroup(F, rels) # presentation of H
>>> G.order() == H.order()
True
```

See also:

[*strong_presentation*](#) (page 329), [*presentation*](#) (page 324), [*stabilizer*](#) (page 328)

abelian_invariants()

Returns the abelian invariants for the given group. Let G be a nontrivial finite abelian group. Then G is isomorphic to the direct product of finitely many nontrivial cyclic groups of prime-power order.

Explanation

The prime-powers that occur as the orders of the factors are uniquely determined by G . More precisely, the primes that occur in the orders of the factors in any such decomposition of G are exactly the primes that divide $|G|$ and for any such prime p , if the orders of the factors that are p -groups in one such decomposition of G are $p^{t_1} \geq p^{t_2} \geq \dots p^{t_r}$, then the orders of the factors that are p -groups in any such decomposition of G are $p^{t_1} \geq p^{t_2} \geq \dots p^{t_r}$.

The uniquely determined integers $p^{t_1} \geq p^{t_2} \geq \dots p^{t_r}$, taken for all primes that divide $|G|$ are called the invariants of the nontrivial group G as suggested in ([14], p. 542).

Notes

We adopt the convention that the invariants of a trivial group are [].

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation([0, 2, 1])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.abelian_invariants()
[2]
>>> from sympy.combinatorics import CyclicGroup
>>> G = CyclicGroup(7)
>>> G.abelian_invariants()
[7]
```

property base

Return a base from the Schreier-Sims algorithm.

Explanation

For a permutation group G , a base is a sequence of points $B = (b_1, b_2, \dots, b_k)$ such that no element of G apart from the identity fixes all the points in B . The concepts of a base and strong generating set and their applications are discussed in depth in [1], pp. 87-89 and [2], pp. 55-57.

An alternative way to think of B is that it gives the indices of the stabilizer cosets that contain more than the identity permutation.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> G = PermutationGroup([Permutation(0, 1, 3)(2, 4)])
>>> G.base
[0, 2]
```

See also:

[*strong_gens*](#) (page 328), [*basic_transversals*](#) (page 298), [*basic_orbits*](#) (page 297), [*basic_stabilizers*](#) (page 297)

baseswap(*base*, *strong_gens*, *pos*, *randomized=False*, *transversals=None*, *basic_orbits=None*, *strong_gens_distr=None*)

Swap two consecutive base points in base and strong generating set.

Parameters

base, strong_gens

The base and strong generating set.

pos

The position at which swapping is performed.

randomized

A switch between randomized and deterministic version.

transversals

The transversals for the basic orbits, if known.

basic_orbits

The basic orbits, if known.

strong_gens_distr

The strong generators distributed by basic stabilizers, if known.

Returns

(base, strong_gens)

base is the new base, and strong_gens is a generating set relative to it.

Explanation

If a base for a group G is given by $(b_1, b_2, \dots, b_k)$, this function returns a base $(b_1, b_2, \dots, b_{i+1}, b_i, \dots, b_k)$, where i is given by `pos`, and a strong generating set relative to that base. The original base and strong generating set are not modified.

The randomized version (default) is of Las Vegas type.

Examples

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.testutil import _verify_bsgs
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> S = SymmetricGroup(4)
>>> S.schreier_sims()
>>> S.base
[0, 1, 2]
>>> base, gens = S.baseswap(S.base, S.strong_gens, 1,
↳ randomized=False)
>>> base, gens
([0, 2, 1],
[(0 1 2 3), (3)(0 1), (1 3 2),
(2 3), (1 3)])
```

check that base, gens is a BSGS

```
>>> S1 = PermutationGroup(gens)
>>> _verify_bsgs(S1, base, gens)
True
```


Notes

The deterministic version of the algorithm is discussed in [1], pp. 102-103; the randomized version is discussed in [1], p.103, and [2], p.98. It is of Las Vegas type. Notice that [1] contains a mistake in the pseudocode and discussion of BASESWAP: on line 3 of the pseudocode, $|\beta_{i+1}^{(T)}|$ should be replaced by $|\beta_i^{(T)}|$, and the same for the discussion of the algorithm.

See also:

[*schreier_sims*](#) (page 324)

property `basic_orbits`

Return the basic orbits relative to a base and strong generating set.

Explanation

If $(b_1, b_2, \dots, b_k)$ is a base for a group G , and $G^{(i)} = G_{b_1, b_2, \dots, b_{i-1}}$ is the i -th basic stabilizer (so that $G^{(1)} = G$), the i -th basic orbit relative to this base is the orbit of b_i under $G^{(i)}$. See [1], pp. 87-89 for more information.

Examples

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> S = SymmetricGroup(4)
>>> S.basic_orbits
[[0, 1, 2, 3], [1, 2, 3], [2, 3]]
```

See also:

[*base*](#) (page 295), [*strong_gens*](#) (page 328), [*basic_transversals*](#) (page 298), [*basic_stabilizers*](#) (page 297)

property `basic_stabilizers`

Return a chain of stabilizers relative to a base and strong generating set.

Explanation

The i -th basic stabilizer $G^{(i)}$ relative to a base $(b_1, b_2, \dots, b_k)$ is $G_{b_1, b_2, \dots, b_{i-1}}$. For more information, see [1], pp. 87-89.

Examples

```
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> A = AlternatingGroup(4)
>>> A.schreier_sims()
>>> A.base
[0, 1]
>>> for g in A.basic_stabilizers:
...     print(g)
```

(continues on next page)

(continued from previous page)

```
...
PermutationGroup([
    (3)(0 1 2),
    (1 2 3)])
PermutationGroup([
    (1 2 3)])
```

See also:

[base](#) (page 295), [strong_gens](#) (page 328), [basic_orbits](#) (page 297), [basic_transversals](#) (page 298)

property `basic_transversals`

Return basic transversals relative to a base and strong generating set.

Explanation

The basic transversals are transversals of the basic orbits. They are provided as a list of dictionaries, each dictionary having keys - the elements of one of the basic orbits, and values - the corresponding transversal elements. See [1], pp. 87-89 for more information.

Examples

```
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> A = AlternatingGroup(4)
>>> A.basic_transversals
[{0: (3), 1: (3)(0 1 2), 2: (3)(0 2 1), 3: (0 3 1)}, {1: (3), 2: (1 2 3),
  ↪ 3: (1 3 2)}]
```

See also:

[strong_gens](#) (page 328), [base](#) (page 295), [basic_orbits](#) (page 297), [basic_stabilizers](#) (page 297)

center()

Return the center of a permutation group.

Explanation

The center for a group G is defined as $Z(G) = \{z \in G | \forall g \in G, zg = gz\}$, the set of elements of G that commute with all elements of G . It is equal to the centralizer of G inside G , and is naturally a subgroup of G ([9]).

Examples

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> D = DihedralGroup(4)
>>> G = D.center()
>>> G.order()
2
```

Notes

This is a naive implementation that is a straightforward application of `.centralizer()`.

See also:

[`centralizer`](#) (page 299)

`centralizer(other)`

Return the centralizer of a group/set/element.

Parameters

`other`

a permutation group/list of permutations/single permutation

Explanation

The centralizer of a set of permutations S inside a group G is the set of elements of G that commute with all elements of S :

$$C_G(S) = \{ g \in G \mid gs = sg \text{ for all } s \in S \}$$

Usually, S is a subset of G , but if G is a proper subgroup of the full symmetric group, we allow for S to have elements outside G .

It is naturally a subgroup of G ; the centralizer of a permutation group is equal to the centralizer of any set of generators for that group, since any element commuting with the generators commutes with any product of the generators.

Examples

```
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... CyclicGroup)
>>> S = SymmetricGroup(6)
>>> C = CyclicGroup(6)
>>> H = S.centralizer(C)
>>> H.is_subgroup(C)
True
```

Notes

The implementation is an application of `.subgroup_search()` with tests using a specific base for the group G .

See also:

[`subgroup\_search`](#) (page 329)

`commutator(G, H)`

Return the commutator of two subgroups.

Explanation

For a permutation group K and subgroups G, H , the commutator of G and H is defined as the group generated by all the commutators $[g, h] = hgh^{-1}g^{-1}$ for g in G and h in H . It is naturally a subgroup of K ([1], p.27).

Examples

```
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... AlternatingGroup)
>>> S = SymmetricGroup(5)
>>> A = AlternatingGroup(5)
>>> G = S.commutator(S, A)
>>> G.is_subgroup(A)
True
```

Notes

The commutator of two subgroups H, G is equal to the normal closure of the commutators of all the generators, i.e. $hgh^{-1}g^{-1}$ for h a generator of H and g a generator of G ([1], p.28)

See also:

[`derived\_subgroup`](#) (page 306)

`composition_series()`

Return the composition series for a group as a list of permutation groups.

Explanation

The composition series for a group G is defined as a subnormal series $G = H_0 > H_1 > H_2 \dots$. A composition series is a subnormal series such that each factor group $H(i+1)/H(i)$ is simple. A subnormal series is a composition series only if it is of maximum length.

The algorithm works as follows: Starting with the derived series the idea is to fill the gap between $G = der[i]$ and $H = der[i+1]$ for each i independently. Since, all subgroups of the abelian group G/H are normal so, first step is to take the generators g of G and add them to generators of H one by one.

The factor groups formed are not simple in general. Each group is obtained from the previous one by adding one generator g , if the previous group is denoted by H then the next group K is generated by g and H . The factor group K/H is cyclic and its order is $K.order()/H.order()$. The series is then extended between K and H by groups generated by powers of g and H . The series formed is then prepended to the already existing series.

Examples

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.named_groups import CyclicGroup
>>> S = SymmetricGroup(12)
>>> G = S.sylow_subgroup(2)
>>> C = G.composition_series()
>>> [H.order() for H in C]
[1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1]
>>> G = S.sylow_subgroup(3)
>>> C = G.composition_series()
>>> [H.order() for H in C]
[243, 81, 27, 9, 3, 1]
>>> G = CyclicGroup(12)
>>> C = G.composition_series()
>>> [H.order() for H in C]
[12, 6, 3, 1]
```

`conjugacy_class(x)`

Return the conjugacy class of an element in the group.

Explanation

The conjugacy class of an element g in a group G is the set of elements x in G that are conjugate with g , i.e. for which

$$g = xax^{-1}$$

for some a in G .

Note that conjugacy is an equivalence relation, and therefore that conjugacy classes are partitions of G . For a list of all the conjugacy classes of the group, use the `conjugacy_classes()` method.

In a permutation group, each conjugacy class corresponds to a particular *cyclestructure*: *forexample, in S_3* , the conjugacy classes are:

- the identity class, $\{()\}$
- all transpositions, $\{(1\ 2), (1\ 3), (2\ 3)\}$
- all 3-cycles, $\{(1\ 2\ 3), (1\ 3\ 2)\}$

Examples

```
>>> from sympy.combinatorics import Permutation, SymmetricGroup
>>> S3 = SymmetricGroup(3)
>>> S3.conjugacy_class(Permutation(0, 1, 2))
{(0 1 2), (0 2 1)}
```

Notes

This procedure computes the conjugacy class directly by finding the orbit of the element under conjugation in G . This algorithm is only feasible for permutation groups of relatively small order, but is like the `orbit()` function itself in that respect.

`conjugacy_classes()`

Return the conjugacy classes of the group.

Explanation

As described in the documentation for the `.conjugacy_class()` function, conjugacy is an equivalence relation on a group G which partitions the set of elements. This method returns a list of all these conjugacy classes of G .

Examples

```
>>> from sympy.combinatorics import SymmetricGroup
>>> SymmetricGroup(3).conjugacy_classes()
[{(2)}, {(0 1 2), (0 2 1)}, {(0 2), (1 2), (2)(0 1)}]
```

`contains(g, strict=True)`

Test if permutation g belong to self, G .

Explanation

If g is an element of G it can be written as a product of factors drawn from the cosets of G 's stabilizers. To see if g is one of the actual generators defining the group use `G.has(g)`.

If `strict` is not `True`, g will be resized, if necessary, to match the size of permutations in self.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
```

```
>>> a = Permutation(1, 2)
>>> b = Permutation(2, 3, 1)
>>> G = PermutationGroup(a, b, degree=5)
>>> G.contains(G[0]) # trivial check
True
>>> elem = Permutation([[2, 3]], size=5)
>>> G.contains(elem)
True
>>> G.contains(Permutation(4)(0, 1, 2, 3))
False
```

If strict is False, a permutation will be resized, if necessary:

```
>>> H = PermutationGroup(Permutation(5))
>>> H.contains(Permutation(3))
False
>>> H.contains(Permutation(3), strict=False)
True
```

To test if a given permutation is present in the group:

```
>>> elem in G.generators
False
>>> G.has(elem)
False
```

See also:

[coset_factor](#) (page 303), [sympy.core.basic.Basic.has](#) (page 933), [__contains__](#) (page 289)

coset_factor(*g*, *factor_index=False*)

Return *G*'s (self's) coset factorization of *g*

Explanation

If *g* is an element of *G* then it can be written as the product of permutations drawn from the Schreier-Sims coset decomposition,

The permutations returned in *f* are those for which the product gives *g*: $g = f[n] * \dots * f[1] * f[0]$ where $n = \text{len}(B)$ and $B = G.\text{base}$. *f*[*i*] is one of the permutations in *self._basic_orbits*[*i*].

If *factor_index*==True, returns a tuple [*b*[0],...,*b*[*n*]], where *b*[*i*] belongs to *self._basic_orbits*[*i*]

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation(0, 1, 3, 7, 6, 4)(2, 5)
>>> b = Permutation(0, 1, 3, 2)(4, 5, 7, 6)
>>> G = PermutationGroup([a, b])
```

Define g:

```
>>> g = Permutation(7)(1, 2, 4)(3, 6, 5)
```

Confirm that it is an element of G:

```
>>> G.contains(g)
True
```

Thus, it can be written as a product of factors (up to 3) drawn from u. See below that a factor from u1 and u2 and the Identity permutation have been used:

```
>>> f = G.coset_factor(g)
>>> f[2]*f[1]*f[0] == g
True
>>> f1 = G.coset_factor(g, True); f1
[0, 4, 4]
>>> tr = G.basic_transversals
>>> f[0] == tr[0][f1[0]]
True
```

If g is not an element of G then [] is returned:

```
>>> c = Permutation(5, 6, 7)
>>> G.coset_factor(c)
[]
```

See also:

[*sympy.combinatorics.util._strip*](#) (page 361)

coset_rank(g)

rank using Schreier-Sims representation.

Explanation

The coset rank of g is the ordering number in which it appears in the lexicographic listing according to the coset decomposition

The ordering is the same as in `G.generate(method='coset')`. If g does not belong to the group it returns None.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation(0, 1, 3, 7, 6, 4)(2, 5)
>>> b = Permutation(0, 1, 3, 2)(4, 5, 7, 6)
>>> G = PermutationGroup([a, b])
>>> c = Permutation(7)(2, 4)(3, 5)
>>> G.coset_rank(c)
16
>>> G.coset_unrank(16)
(7)(2 4)(3 5)
```

See also:

[`coset\_factor`](#) (page 303)

`coset_table(H)`

Return the standardised (right) coset table of self in H as a list of lists.

`coset_transversal(H)`

Return a transversal of the right cosets of self by its subgroup H using the second method described in [1], Subsection 4.6.7

`coset_unrank(rank, af=False)`

unrank using Schreier-Sims representation

`coset_unrank` is the inverse operation of `coset_rank` if $0 \leq \text{rank} < \text{order}$; otherwise it returns `None`.

property `degree`

Returns the size of the permutations in the group.

Explanation

The number of permutations comprising the group is given by `len(group)`; the number of permutations that can be generated by the group is given by `group.order()`.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation([1, 0, 2])
>>> G = PermutationGroup([a])
>>> G.degree
3
>>> len(G)
1
>>> G.order()
2
>>> list(G.generate())
[(2), (2)(0 1)]
```

See also:

[`order`](#) (page 322)

derived_series()

Return the derived series for the group.

Returns

A list of permutation groups containing the members of the derived series in the order $G = G_0, G_1, G_2, \dots$

Explanation

The derived series for a group G is defined as $G = G_0 > G_1 > G_2 > \dots$ where $G_i = [G_{i-1}, G_{i-1}]$, i.e. G_i is the derived subgroup of G_{i-1} , for $i \in \mathbb{N}$. When we have $G_k = G_{k-1}$ for some $k \in \mathbb{N}$, the series terminates.

Examples

```
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... AlternatingGroup, DihedralGroup)
>>> A = AlternatingGroup(5)
>>> len(A.derived_series())
1
>>> S = SymmetricGroup(4)
>>> len(S.derived_series())
4
>>> S.derived_series()[1].is_subgroup(AlternatingGroup(4))
True
>>> S.derived_series()[2].is_subgroup(DihedralGroup(2))
True
```

See also:

[*derived_subgroup*](#) (page 306)

derived_subgroup()

Compute the derived subgroup.

Explanation

The derived subgroup, or commutator subgroup is the subgroup generated by all commutators $[g, h] = hgh^{-1}g^{-1}$ for $g, h \in G$; it is equal to the normal closure of the set of commutators of the generators ([1], p.28, [11]).

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation([1, 0, 2, 4, 3])
>>> b = Permutation([0, 1, 3, 2, 4])
>>> G = PermutationGroup([a, b])
>>> C = G.derived_subgroup()
>>> list(C.generate(af=True))
[[0, 1, 2, 3, 4], [0, 1, 3, 4, 2], [0, 1, 4, 2, 3]]
```

See also:

[*derived_series*](#) (page 306)

property elements

Returns all the elements of the permutation group as a set

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> p = PermutationGroup(Permutation(1, 3), Permutation(1, 2))
>>> p.elements
{(1 2 3), (1 3 2), (1 3), (2 3), (3), (3)(1 2)}
```

equals (*other*)

Return True if PermutationGroup generated by elements in the group are same i.e they represent the same PermutationGroup.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> p = Permutation(0, 1, 2, 3, 4, 5)
>>> G = PermutationGroup([p, p**2])
>>> H = PermutationGroup([p**2, p])
>>> G.generators == H.generators
False
>>> G.equals(H)
True
```

generate (*method='coset', af=False*)

Return iterator to generate the elements of the group.

Explanation

Iteration is done with one of these methods:

```
method='coset' using the Schreier-Sims coset representation
method='dimino' using the Dimino method
```

If `af = True` it yields the array form of the permutations

Examples

```
>>> from sympy.combinatorics import PermutationGroup
>>> from sympy.combinatorics.polyhedron import tetrahedron
```

The permutation group given in the tetrahedron object is also true groups:

```
>>> G = tetrahedron.pgroup
>>> G.is_group
True
```

Also the group generated by the permutations in the tetrahedron pgroup – even the first two – is a proper group:

```
>>> H = PermutationGroup(G[0], G[1])
>>> J = PermutationGroup(list(H.generate())); J
PermutationGroup([
  (0 1)(2 3),
  (1 2 3),
  (1 3 2),
  (0 3 1),
  (0 2 3),
  (0 3)(1 2),
  (0 1 3),
  (3)(0 2 1),
  (0 3 2),
  (3)(0 1 2),
  (0 2)(1 3)])
>>> J.is_group
True
```

generate_dimino(*af=False*)

Yield group elements using Dimino's algorithm.

If `af == True` it yields the array form of the permutations.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation([0, 2, 1, 3])
>>> b = Permutation([0, 2, 3, 1])
>>> g = PermutationGroup([a, b])
>>> list(g.generate_dimino(af=True))
[[0, 1, 2, 3], [0, 2, 1, 3], [0, 2, 3, 1],
 [0, 1, 3, 2], [0, 3, 2, 1], [0, 3, 1, 2]]
```

References

[R64]

generate_schreier_sims(*af=False*)

Yield group elements using the Schreier-Sims representation in coset_rank order

If *af* = True it yields the array form of the permutations

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation([0, 2, 1, 3])
>>> b = Permutation([0, 2, 3, 1])
>>> g = PermutationGroup([a, b])
>>> list(g.generate_schreier_sims(af=True))
[[0, 1, 2, 3], [0, 2, 1, 3], [0, 3, 2, 1],
 [0, 1, 3, 2], [0, 2, 3, 1], [0, 3, 1, 2]]
```

generator_product(*g, original=False*)

Return a list of strong generators $[s_1, \dots, s_n]$ s.t $g = s_n \times \dots \times s_1$. If *original*=True, make the list contain only the original group generators

property generators

Returns the generators of the group.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation([0, 2, 1])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.generators
[(1 2), (2)(0 1)]
```

property identity

Return the identity element of the permutation group.

index(*H*)

Returns the index of a permutation group.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation(1,2,3)
>>> b = Permutation(3)
>>> G = PermutationGroup([a])
>>> H = PermutationGroup([b])
>>> G.index(H)
3
```

property `is_abelian`

Test if the group is Abelian.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation([0, 2, 1])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.is_abelian
False
>>> a = Permutation([0, 2, 1])
>>> G = PermutationGroup([a])
>>> G.is_abelian
True
```

`is_alt_sym(eps=0.05, _random_prec=None)`

Monte Carlo test for the symmetric/alternating group for degrees ≥ 8 .

Explanation

More specifically, it is one-sided Monte Carlo with the answer True (i.e., G is symmetric/alternating) guaranteed to be correct, and the answer False being incorrect with probability ϵ .

For degree < 8 , the order of the group is checked so the test is deterministic.

Notes

The algorithm itself uses some nontrivial results from group theory and number theory: 1) If a transitive group G of degree n contains an element with a cycle of length $n/2 < p < n-2$ for p a prime, G is the symmetric or alternating group ([1], pp. 81-82) 2) The proportion of elements in the symmetric/alternating group having the property described in 1) is approximately $\log(2)/\log(n)$ ([1], p.82; [2], pp. 226-227). The helper function `_check_cycles_alt_sym` is used to go over the cycles in a permutation and look for ones satisfying 1).

Examples

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> D = DihedralGroup(10)
>>> D.is_alt_sym()
False
```

See also:

[`\_check\_cycles\_alt\_sym`](#) (page 357)

property `is_alternating`

Return True if the group is alternating.

Examples

```
>>> from sympy.combinatorics import AlternatingGroup
>>> g = AlternatingGroup(5)
>>> g.is_alternating
True
```

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> g = PermutationGroup(
...     Permutation(0, 1, 2, 3, 4),
...     Permutation(2, 3, 4))
>>> g.is_alternating
True
```

Notes

This uses a naive test involving the computation of the full group order. If you need more quicker taxonomy for large groups, you can use [`PermutationGroup.is\_alt\_sym\(\)`](#) (page 310). However, [`PermutationGroup.is\_alt\_sym\(\)`](#) (page 310) may not be accurate and is not able to distinguish between an alternating group and a symmetric group.

See also:

[`is\_alt\_sym`](#) (page 310)

property `is_cyclic`

Return True if the group is Cyclic.

Examples

```
>>> from sympy.combinatorics.named_groups import AbelianGroup
>>> G = AbelianGroup(3, 4)
>>> G.is_cyclic
True
>>> G = AbelianGroup(4, 4)
>>> G.is_cyclic
False
```

Notes

If the order of a group n can be factored into the distinct primes $p_1, p_2, \dots, p_s$ and if

$$\forall i, j \in \{1, 2, \dots, s\} : p_i \not\equiv 1 \pmod{p_j}$$

holds true, there is only one group of the order n which is a cyclic group [R65]. This is a generalization of the lemma that the group of order 15, 35, ... are cyclic.

And also, these additional lemmas can be used to test if a group is cyclic if the order of the group is already found.

- If the group is abelian and the order of the group is square-free, the group is cyclic.
- If the order of the group is less than 6 and is not 4, the group is cyclic.
- If the order of the group is prime, the group is cyclic.

References

[R65]

`is_elementary(p)`

Return True if the group is elementary abelian. An elementary abelian group is a finite abelian group, where every nontrivial element has order p , where p is a prime.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation([0, 2, 1])
>>> G = PermutationGroup([a])
>>> G.is_elementary(2)
True
>>> a = Permutation([0, 2, 1, 3])
>>> b = Permutation([3, 1, 2, 0])
>>> G = PermutationGroup([a, b])
>>> G.is_elementary(2)
True
>>> G.is_elementary(3)
False
```


property is_nilpotent

Test if the group is nilpotent.

Explanation

A group G is nilpotent if it has a central series of finite length. Alternatively, G is nilpotent if its lower central series terminates with the trivial group. Every nilpotent group is also solvable ([1], p.29, [12]).

Examples

```
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... CyclicGroup)
>>> C = CyclicGroup(6)
>>> C.is_nilpotent
True
>>> S = SymmetricGroup(5)
>>> S.is_nilpotent
False
```

See also:

[*lower_central_series*](#) (page 317), [*is_solvable*](#) (page 315)

is_normal(*gr*, *strict=True*)

Test if $G=\text{self}$ is a normal subgroup of gr .

Explanation

G is normal in gr if for each g_2 in G , g_1 in gr , $g = g_1 * g_2 * g_1^{-1}$ belongs to G . It is sufficient to check this for each g_1 in $gr.\text{generators}$ and g_2 in $G.\text{generators}$.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation([1, 2, 0])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G1 = PermutationGroup([a, Permutation([2, 0, 1])])
>>> G1.is_normal(G)
True
```

property is_perfect

Return True if the group is perfect. A group is perfect if it equals to its derived subgroup.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation(1,2,3)(4,5)
>>> b = Permutation(1,2,3,4,5)
>>> G = PermutationGroup([a, b])
>>> G.is_perfect
False
```

property `is_polycyclic`

Return True if a group is polycyclic. A group is polycyclic if it has a subnormal series with cyclic factors. For finite groups, this is the same as if the group is solvable.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation([0, 2, 1, 3])
>>> b = Permutation([2, 0, 1, 3])
>>> G = PermutationGroup([a, b])
>>> G.is_polycyclic
True
```

`is_primitive(randomized=True)`

Test if a group is primitive.

Explanation

A permutation group G acting on a set S is called primitive if S contains no nontrivial block under the action of G (a block is nontrivial if its cardinality is more than 1).

Notes

The algorithm is described in [1], p.83, and uses the function `minimal_block` to search for blocks of the form $\{0, k\}$ for k ranging over representatives for the orbits of G_0 , the stabilizer of 0 . This algorithm has complexity $O(n^2)$ where n is the degree of the group, and will perform badly if G_0 is small.

There are two implementations offered: one finds G_0 deterministically using the function `stabilizer`, and the other (default) produces random elements of G_0 using `random_stab`, hoping that they generate a subgroup of G_0 with not too many more orbits than G_0 (this is suggested in [1], p.83). Behavior is changed by the `randomized` flag.

Examples

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> D = DihedralGroup(10)
>>> D.is_primitive()
False
```

See also:

[*minimal_block*](#) (page 319), [*random_stab*](#) (page 324)

property `is_solvable`

Test if the group is solvable.

G is solvable if its derived series terminates with the trivial group ([1], p.29).

Examples

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> S = SymmetricGroup(3)
>>> S.is_solvable
True
```

See also:

[*is_nilpotent*](#) (page 312), [*derived_series*](#) (page 306)

method `is_subgroup(G, strict=True)`

Return True if all elements of self belong to G.

If strict is False then if self's degree is smaller than G's, the elements will be resized to have the same degree.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> from sympy.combinatorics import SymmetricGroup, CyclicGroup
```

Testing is strict by default: the degree of each group must be the same:

```
>>> p = Permutation(0, 1, 2, 3, 4, 5)
>>> G1 = PermutationGroup([Permutation(0, 1, 2), Permutation(0, 1)])
>>> G2 = PermutationGroup([Permutation(0, 2), Permutation(0, 1, 2)])
>>> G3 = PermutationGroup([p, p**2])
>>> assert G1.order() == G2.order() == G3.order() == 6
>>> G1.is_subgroup(G2)
True
>>> G1.is_subgroup(G3)
False
>>> G3.is_subgroup(PermutationGroup(G3[1]))
False
>>> G3.is_subgroup(PermutationGroup(G3[0]))
True
```

To ignore the size, set `strict` to `False`:

```
>>> S3 = SymmetricGroup(3)
>>> S5 = SymmetricGroup(5)
>>> S3.is_subgroup(S5, strict=False)
True
>>> C7 = CyclicGroup(7)
>>> G = S5*C7
>>> S5.is_subgroup(G, False)
True
>>> C7.is_subgroup(G, 0)
False
```

property `is_symmetric`

Return `True` if the group is symmetric.

Examples

```
>>> from sympy.combinatorics import SymmetricGroup
>>> g = SymmetricGroup(5)
>>> g.is_symmetric
True
```

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> g = PermutationGroup(
...     Permutation(0, 1, 2, 3, 4),
...     Permutation(2, 3))
>>> g.is_symmetric
True
```

Notes

This uses a naive test involving the computation of the full group order. If you need more quicker taxonomy for large groups, you can use [*PermutationGroup.is_alt_sym\(\)*](#) (page 310). However, [*PermutationGroup.is_alt_sym\(\)*](#) (page 310) may not be accurate and is not able to distinguish between an alternating group and a symmetric group.

See also:

[*is_alt_sym*](#) (page 310)

`is_transitive(strict=True)`

Test if the group is transitive.

Explanation

A group is transitive if it has a single orbit.

If `strict` is `False` the group is transitive if it has a single orbit of length different from 1.

Examples

```

>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation([0, 2, 1, 3])
>>> b = Permutation([2, 0, 1, 3])
>>> G1 = PermutationGroup([a, b])
>>> G1.is_transitive()
False
>>> G1.is_transitive(strict=False)
True
>>> c = Permutation([2, 3, 0, 1])
>>> G2 = PermutationGroup([a, c])
>>> G2.is_transitive()
True
>>> d = Permutation([1, 0, 2, 3])
>>> e = Permutation([0, 1, 3, 2])
>>> G3 = PermutationGroup([d, e])
>>> G3.is_transitive() or G3.is_transitive(strict=False)
False

```

property `is_trivial`

Test if the group is the trivial group.

This is true if the group contains only the identity permutation.

Examples

```

>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> G = PermutationGroup([Permutation([0, 1, 2])])
>>> G.is_trivial
True

```

`lower_central_series()`

Return the lower central series for the group.

The lower central series for a group G is the series $G = G_0 > G_1 > G_2 > \dots$ where $G_k = [G, G_{k-1}]$, i.e. every term after the first is equal to the commutator of G and the previous term in G_1 ([1], p.29).

Returns

A list of permutation groups in the order $G = G_0, G_1, G_2, \dots$

Examples

```
>>> from sympy.combinatorics.named_groups import (AlternatingGroup,
... DihedralGroup)
>>> A = AlternatingGroup(4)
>>> len(A.lower_central_series())
2
>>> A.lower_central_series()[1].is_subgroup(DihedralGroup(2))
True
```

See also:

[*commutator*](#) (page 300), [*derived_series*](#) (page 306)

make_perm(*n*, *seed*=None)

Multiply *n* randomly selected permutations from *pgroup* together, starting with the identity permutation. If *n* is a list of integers, those integers will be used to select the permutations and they will be applied in L to R order: `make_perm((A, B, C))` will give `CBA(I)` where *I* is the identity permutation.

seed is used to set the seed for the random selection of permutations from *pgroup*. If this is a list of integers, the corresponding permutations from *pgroup* will be selected in the order give. This is mainly used for testing purposes.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a, b = [Permutation([1, 0, 3, 2]), Permutation([1, 3, 0, 2])]
>>> G = PermutationGroup([a, b])
>>> G.make_perm(1, [0])
(0 1)(2 3)
>>> G.make_perm(3, [0, 1, 0])
(0 2 3 1)
>>> G.make_perm([0, 1, 0])
(0 2 3 1)
```

See also:

[*random*](#) (page 324)

property max_div

Maximum proper divisor of the degree of a permutation group.

Explanation

Obviously, this is the degree divided by its minimal proper divisor (larger than 1, if one exists). As it is guaranteed to be prime, the sieve from `sympy.ntheory` is used. This function is also used as an optimization tool for the functions `minimal_block` and `_union_find_merge`.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> G = PermutationGroup([Permutation([0, 2, 1, 3])])
>>> G.max_div
2
```

See also:

[`minimal\_block`](#) (page 319), [`\_union\_find\_merge`](#) (page 292)

`minimal_block(points)`

For a transitive group, finds the block system generated by points.

Explanation

If a group G acts on a set S , a nonempty subset B of S is called a block under the action of G if for all g in G we have $gB = B$ (g fixes B) or gB and B have no common points (g moves B entirely). ([1], p.23; [6]).

The distinct translates gB of a block B for g in G partition the set S and this set of translates is known as a block system. Moreover, we obviously have that all blocks in the partition have the same size, hence the block size divides $|S|$ ([1], p.23). A G -congruence is an equivalence relation $\sim$ on the set S such that $a \sim b$ implies $g(a) \sim g(b)$ for all g in G . For a transitive group, the equivalence classes of a G -congruence and the blocks of a block system are the same thing ([1], p.23).

The algorithm below checks the group for transitivity, and then finds the G -congruence generated by the pairs $(p_0, p_1), (p_0, p_2), \dots, (p_0, p_{k-1})$ which is the same as finding the maximal block system (i.e., the one with minimum block size) such that $p_0, \dots, p_{k-1}$ are in the same block ([1], p.83).

It is an implementation of Atkinson's algorithm, as suggested in [1], and manipulates an equivalence relation on the set S using a union-find data structure. The running time is just above $O(|points||S|)$. ([1], pp. 83-87; [7]).

Examples

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> D = DihedralGroup(10)
>>> D.minimal_block([0, 5])
[0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
>>> D.minimal_block([0, 1])
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

See also:

[`\_union\_find\_rep`](#) (page 293), [`\_union\_find\_merge`](#) (page 292), [`is\_transitive`](#) (page 316), [`is\_primitive`](#) (page 314)

`minimal_blocks(randomized=True)`

For a transitive group, return the list of all minimal block systems. If a group is intransitive, return *False*.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> DihedralGroup(6).minimal_blocks()
[[0, 1, 0, 1, 0, 1], [0, 1, 2, 0, 1, 2]]
>>> G = PermutationGroup(Permutation(1,2,5))
>>> G.minimal_blocks()
False
```

See also:

[*minimal_block*](#) (page 319), [*is_transitive*](#) (page 316), [*is_primitive*](#) (page 314)

normal_closure(*other*, *k=10*)

Return the normal closure of a subgroup/set of permutations.

Parameters

other

a subgroup/list of permutations/single permutation

k

an implementation-specific parameter that determines the number of conjugates that are adjoined to *other* at once

Explanation

If *S* is a subset of a group *G*, the normal closure of *A* in *G* is defined as the intersection of all normal subgroups of *G* that contain *A* ([1], p.14). Alternatively, it is the group generated by the conjugates $x^{-1}yx$ for *x* a generator of *G* and *y* a generator of the subgroup $\langle S \rangle$ generated by *S* (for some chosen generating set for $\langle S \rangle$) ([1], p.73).

Examples

```
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... CyclicGroup, AlternatingGroup)
>>> S = SymmetricGroup(5)
>>> C = CyclicGroup(5)
>>> G = S.normal_closure(C)
>>> G.order()
60
>>> G.is_subgroup(AlternatingGroup(5))
True
```


Notes

The algorithm is described in [1], pp. 73-74; it makes use of the generation of random elements for permutation groups by the product replacement algorithm.

See also:

[commutator](#) (page 300), [derived_subgroup](#) (page 306), [random_pr](#) (page 324)

orbit(*alpha*, *action*='tuples')

Compute the orbit of α $\{g(\alpha) | g \in G\}$ as a set.

Explanation

The time complexity of the algorithm used here is $O(|Orb| * r)$ where $|Orb|$ is the size of the orbit and r is the number of generators of the group. For a more detailed analysis, see [1], p.78, [2], pp. 19-21. Here α can be a single point, or a list of points.

If α is a single point, the ordinary orbit is computed. if α is a list of points, there are three available options:

'union' - computes the union of the orbits of the points in the list 'tuples' - computes the orbit of the list interpreted as an ordered tuple under the group action (i.e., $g((1,2,3)) = (g(1), g(2), g(3))$) 'sets' - computes the orbit of the list interpreted as a sets

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation([1, 2, 0, 4, 5, 6, 3])
>>> G = PermutationGroup([a])
>>> G.orbit(0)
{0, 1, 2}
>>> G.orbit([0, 4], 'union')
{0, 1, 2, 3, 4, 5, 6}
```

See also:

[orbit_transversal](#) (page 322)

orbit_rep(*alpha*, *beta*, *schreier_vector*=None)

Return a group element which sends α to β .

Explanation

If β is not in the orbit of α , the function returns `False`. This implementation makes use of the schreier vector. For a proof of correctness, see [1], p.80

Examples

```
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> G = AlternatingGroup(5)
>>> G.orbit_rep(0, 4)
(0 4 1 2 3)
```

See also:

[`schreier\_vector`](#) (page 327)

`orbit_transversal(alpha, pairs=False)`

Computes a transversal for the orbit of α as a set.

Explanation

For a permutation group G , a transversal for the orbit $Orb = \{g(\alpha) | g \in G\}$ is a set $\{g_\beta | g_\beta(\alpha) = \beta\}$ for $\beta \in Orb$. Note that there may be more than one possible transversal. If `pairs` is set to `True`, it returns the list of pairs (β, g_β) . For a proof of correctness, see [1], p.79

Examples

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> G = DihedralGroup(6)
>>> G.orbit_transversal(0)
[(5), (0 1 2 3 4 5), (0 5)(1 4)(2 3), (0 2 4)(1 3 5), (5)(0 4)(1 3),
 ↪ (0 3)(1 4)(2 5)]
```

See also:

[`orbit`](#) (page 321)

`orbits(rep=False)`

Return the orbits of `self`, ordered according to lowest element in each orbit.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation(1, 5)(2, 3)(4, 0, 6)
>>> b = Permutation(1, 5)(3, 4)(2, 6, 0)
>>> G = PermutationGroup([a, b])
>>> G.orbits()
[{0, 2, 3, 4, 6}, {1, 5}]
```

order()

Return the order of the group: the number of permutations that can be generated from elements of the group.

The number of permutations comprising the group is given by `len(group)`; the length of each permutation in the group is given by `group.size`.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
```

```
>>> a = Permutation([1, 0, 2])
>>> G = PermutationGroup([a])
>>> G.degree
3
>>> len(G)
1
>>> G.order()
2
>>> list(G.generate())
[(2), (2)(0 1)]
```

```
>>> a = Permutation([0, 2, 1])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.order()
6
```

See also:

[*degree*](#) (page 305)

pointwise_stabilizer(points, incremental=True)

Return the pointwise stabilizer for a set of points.

Explanation

For a permutation group G and a set of points $\{p_1, p_2, \dots, p_k\}$, the pointwise stabilizer of $p_1, p_2, \dots, p_k$ is defined as $G_{p_1, \dots, p_k} = \{g \in G \mid g(p_i) = p_i \forall i \in \{1, 2, \dots, k\}\}$ ([1], p20). It is a subgroup of G .

Examples

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> S = SymmetricGroup(7)
>>> Stab = S.pointwise_stabilizer([2, 3, 5])
>>> Stab.is_subgroup(S.stabilizer(2).stabilizer(3).stabilizer(5))
True
```

Notes

When `incremental == True`, rather than the obvious implementation using successive calls to `.stabilizer()`, this uses the incremental Schreier-Sims algorithm to obtain a base with starting segment - the given points.

See also:

[`stabilizer`](#) (page 328), [`schreier\_sims\_incremental`](#) (page 325)

`polycyclic_group()`

Return the `PolycyclicGroup` instance with below parameters:

Explanation

- `pc_sequence` : Polycyclic sequence is formed by collecting all the missing generators between the adjacent groups in the derived series of given permutation group.
- `pc_series` : Polycyclic series is formed by adding all the missing generators of `der[i+1]` in `der[i]`, where `der` represents the derived series.
- `relative_order` : A list, computed by the ratio of adjacent groups in `pc_series`.

`presentation(eliminate_gens=True)`

Return an *FpGroup* presentation of the group.

The algorithm is described in [1], Chapter 6.1.

`random(af=False)`

Return a random group element

`random_pr(gen_count=11, iterations=50, _random_prec=None)`

Return a random group element using product replacement.

Explanation

For the details of the product replacement algorithm, see `_random_pr_init`. In `random_pr` the actual 'product replacement' is performed. Notice that if the attribute `_random_gens` is empty, it needs to be initialized by `_random_pr_init`.

See also:

[`\_random\_pr\_init`](#) (page 291)

`random_stab(alpha, schreier_vector=None, _random_prec=None)`

Random element from the stabilizer of `alpha`.

The schreier vector for `alpha` is an optional argument used for speeding up repeated calls. The algorithm is described in [1], p.81

See also:

[`random\_pr`](#) (page 324), [`orbit\_rep`](#) (page 321)

`schreier_sims()`

Schreier-Sims algorithm.

Explanation

It computes the generators of the chain of stabilizers $G > G_{b_1} > \dots > G_{b_1, \dots, b_r} > 1$ in which $G_{b_1, \dots, b_i}$ stabilizes $b_1, \dots, b_i$, and the corresponding s cosets. An element of the group can be written as the product $h_1 * \dots * h_s$.

We use the incremental Schreier-Sims algorithm.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation([0, 2, 1])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.schreier_sims()
>>> G.basic_transversals
[{0: (2)(0 1), 1: (2), 2: (1 2)},
 {0: (2), 2: (0 2)}]
```

schreier_sims_incremental(*base=None, gens=None, slp_dict=False*)

Extend a sequence of points and generating set to a base and strong generating set.

Parameters

base

The sequence of points to be extended to a base. Optional parameter with default value `[]`.

gens

The generating set to be extended to a strong generating set relative to the base obtained. Optional parameter with default value `self.generators`.

slp_dict

If `True`, return a dictionary $g : gens$ for each strong generator g where $gens$ is a list of strong generators coming before g in $strong\_gens$, such that the product of the elements of $gens$ is equal to g .

Returns

(base, strong_gens)

base is the base obtained, and strong_gens is the strong generating set relative to it. The original parameters base, gens remain unchanged.

Examples

```
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> from sympy.combinatorics.testutil import _verify_bsgs
>>> A = AlternatingGroup(7)
>>> base = [2, 3]
>>> seq = [2, 3]
>>> base, strong_gens = A.schreier_sims_incremental(base=seq)
>>> _verify_bsgs(A, base, strong_gens)
True
>>> base[:2]
[2, 3]
```

Notes

This version of the Schreier-Sims algorithm runs in polynomial time. There are certain assumptions in the implementation - if the trivial group is provided, `base` and `gens` are returned immediately, as any sequence of points is a base for the trivial group. If the identity is present in the generators `gens`, it is removed as it is a redundant generator. The implementation is described in [1], pp. 90-93.

See also:

[*schreier_sims*](#) (page 324), [*schreier_sims_random*](#) (page 326)

`schreier_sims_random`(*base=None*, *gens=None*, *consec_succ=10*,
_random_prec=None)

Randomized Schreier-Sims algorithm.

Parameters

base

The sequence to be extended to a base.

gens

The generating set to be extended to a strong generating set.

consec_succ

The parameter defining the probability of a wrong answer.

_random_prec

An internal parameter used for testing purposes.

Returns

(*base*, *strong_gens*)

base is the base and *strong_gens* is the strong generating set relative to it.

Explanation

The randomized Schreier-Sims algorithm takes the sequence `base` and the generating set `gens`, and extends `base` to a base, and `gens` to a strong generating set relative to that base with probability of a wrong answer at most $2^{-\text{consec\_succ}}$, provided the random generators are sufficiently random.

Examples

```
>>> from sympy.combinatorics.testutil import _verify_bsgs
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> S = SymmetricGroup(5)
>>> base, strong_gens = S.schreier_sims_random(consec_succ=5)
>>> _verify_bsgs(S, base, strong_gens)
True
```

Notes

The algorithm is described in detail in [1], pp. 97-98. It extends the orbits `orbs` and the permutation groups `stabs` to basic orbits and basic stabilizers for the base and strong generating set produced in the end. The idea of the extension process is to “sift” random group elements through the stabilizer chain and amend the stabilizers/orbits along the way when a sift is not successful. The helper function `_strip` is used to attempt to decompose a random group element according to the current state of the stabilizer chain and report whether the element was fully decomposed (successful sift) or not (unsuccessful sift). In the latter case, the level at which the sift failed is reported and used to amend `stabs`, `base`, `gens` and `orbs` accordingly. The halting condition is for `consec_succ` consecutive successful sifts to pass. This makes sure that the current base and gens form a BSGS with probability at least $1 - 1/\text{consec\_succ}$.

See also:

[`schreier\_sims`](#) (page 324)

`schreier_vector(alpha)`

Computes the schreier vector for `alpha`.

Explanation

The Schreier vector efficiently stores information about the orbit of `alpha`. It can later be used to quickly obtain elements of the group that send `alpha` to a particular element in the orbit. Notice that the Schreier vector depends on the order in which the group generators are listed. For a definition, see [3]. Since list indices start from zero, we adopt the convention to use “None” instead of 0 to signify that an element does not belong to the orbit. For the algorithm and its correctness, see [2], pp.78-80.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation([2, 4, 6, 3, 1, 5, 0])
>>> b = Permutation([0, 1, 3, 5, 4, 6, 2])
>>> G = PermutationGroup([a, b])
>>> G.schreier_vector(0)
[-1, None, 0, 1, None, 1, 0]
```

See also:

[`orbit`](#) (page 321)

stabilizer(*alpha*)

Return the stabilizer subgroup of *alpha*.

Explanation

The stabilizer of α is the group $G_\alpha = \{g \in G | g(\alpha) = \alpha\}$. For a proof of correctness, see [1], p.79.

Examples

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> G = DihedralGroup(6)
>>> G.stabilizer(5)
PermutationGroup([
    (5)(0 4)(1 3)])
```

See also:

[`orbit`](#) (page 321)

property strong_gens

Return a strong generating set from the Schreier-Sims algorithm.

Explanation

A generating set $S = \{g_1, g_2, \dots, g_t\}$ for a permutation group G is a strong generating set relative to the sequence of points (referred to as a “base”) $(b_1, b_2, \dots, b_k)$ if, for $1 \leq i \leq k$ we have that the intersection of the pointwise stabilizer $G^{(i+1)} := G_{b_1, b_2, \dots, b_i}$ with S generates the pointwise stabilizer $G^{(i+1)}$. The concepts of a base and strong generating set and their applications are discussed in depth in [1], pp. 87-89 and [2], pp. 55-57.

Examples

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> D = DihedralGroup(4)
>>> D.strong_gens
[(0 1 2 3), (0 3)(1 2), (1 3)]
>>> D.base
[0, 1]
```

See also:

[base](#) (page 295), [basic_transversals](#) (page 298), [basic_orbits](#) (page 297), [basic_stabilizers](#) (page 297)

strong_presentation()

Return a strong finite presentation of group. The generators of the returned group are in the same order as the strong generators of group.

The algorithm is based on Sims' Verify algorithm described in [1], Chapter 6.

Examples

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> P = DihedralGroup(4)
>>> G = P.strong_presentation()
>>> P.order() == G.order()
True
```

See also:

[presentation](#) (page 324), [_verify](#) (page 293)

subgroup(gens)

Return the subgroup generated by *gens* which is a list of elements of the group

subgroup_search(prop, base=None, strong_gens=None, tests=None, init_subgroup=None)

Find the subgroup of all elements satisfying the property *prop*.

Parameters

prop

The property to be used. Has to be callable on group elements and always return True or False. It is assumed that all group elements satisfying *prop* indeed form a subgroup.

base

A base for the supergroup.

strong_gens

A strong generating set for the supergroup.

tests

A list of callables of length equal to the length of *base*. These are used to rule out group elements by partial base images, so that

`tests[l](g)` returns False if the element `g` is known not to satisfy `prop` base on where `g` sends the first `l + 1` base points.

init_subgroup

if a subgroup of the sought group is known in advance, it can be passed to the function as this parameter.

Returns

`res`

The subgroup of all elements satisfying `prop`. The generating set for this group is guaranteed to be a strong generating set relative to the base `base`.

Explanation

This is done by a depth-first search with respect to base images that uses several tests to prune the search tree.

Examples

```
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... AlternatingGroup)
>>> from sympy.combinatorics.testutil import _verify_bsgs
>>> S = SymmetricGroup(7)
>>> prop_even = lambda x: x.is_even
>>> base, strong_gens = S.schreier_sims_incremental()
>>> G = S.subgroup_search(prop_even, base=base, strong_gens=strong_
↳ gens)
>>> G.is_subgroup(AlternatingGroup(7))
True
>>> _verify_bsgs(G, base, G.generators)
True
```

Notes

This function is extremely lengthy and complicated and will require some careful attention. The implementation is described in [1], pp. 114-117, and the comments for the code here follow the lines of the pseudocode in the book for clarity.

The complexity is exponential in general, since the search process by itself visits all members of the supergroup. However, there are a lot of tests which are used to prune the search tree, and users can define their own tests via the `tests` parameter, so in practice, and for some computations, it's not terrible.

A crucial part in the procedure is the frequent base change performed (this is line 11 in the pseudocode) in order to obtain a new basic stabilizer. The book mentions that this can be done by using `.baseswap(...)`, however the current implementation uses a more straightforward way to find the next basic stabilizer - calling the function `.stabilizer(...)` on the previous basic stabilizer.

syLOW_subgroup(*p*)

Return a *p*-Sylow subgroup of the group.

The algorithm is described in [1], Chapter 4, Section 7

Examples

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.named_groups import AlternatingGroup
```

```
>>> D = DihedralGroup(6)
>>> S = D.sylow_subgroup(2)
>>> S.order()
4
>>> G = SymmetricGroup(6)
>>> S = G.sylow_subgroup(5)
>>> S.order()
5
```

```
>>> G1 = AlternatingGroup(3)
>>> G2 = AlternatingGroup(5)
>>> G3 = AlternatingGroup(9)
```

```
>>> S1 = G1.sylow_subgroup(3)
>>> S2 = G2.sylow_subgroup(3)
>>> S3 = G3.sylow_subgroup(3)
```

```
>>> len1 = len(S1.lower_central_series())
>>> len2 = len(S2.lower_central_series())
>>> len3 = len(S3.lower_central_series())
```

```
>>> len1 == len2
True
>>> len1 < len3
True
```

property transitivity_degree

Compute the degree of transitivity of the group.

Explanation

A permutation group G acting on $\Omega = \{0, 1, \dots, n-1\}$ is k -fold transitive, if, for any k points $(a_1, a_2, \dots, a_k) \in \Omega$ and any k points $(b_1, b_2, \dots, b_k) \in \Omega$ there exists $g \in G$ such that $g(a_1) = b_1, g(a_2) = b_2, \dots, g(a_k) = b_k$. The degree of transitivity of G is the maximum k such that G is k -fold transitive. ([8])

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation([1, 2, 0])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.transitivity_degree
3
```

See also:

[*is_transitive*](#) (page 316), [*orbit*](#) (page 321)

Polyhedron

class sympy.combinatorics.polyhedron.**Polyhedron**(*corners*, *faces=()*, *pgroup=()*)
Represents the polyhedral symmetry group (PSG).

Explanation

The PSG is one of the symmetry groups of the Platonic solids. There are three polyhedral groups: the tetrahedral group of order 12, the octahedral group of order 24, and the icosahedral group of order 60.

All doctests have been given in the docstring of the constructor of the object.

References

[R81]

property array_form

Return the indices of the corners.

The indices are given relative to the original position of corners.

Examples

```
>>> from sympy.combinatorics.polyhedron import tetrahedron
>>> tetrahedron = tetrahedron.copy()
>>> tetrahedron.array_form
[0, 1, 2, 3]
```

```
>>> tetrahedron.rotate(0)
>>> tetrahedron.array_form
[0, 2, 3, 1]
>>> tetrahedron.pgroup[0].array_form
[0, 2, 3, 1]
```

See also:

[*corners*](#) (page 333), [*cyclic_form*](#) (page 333)

property corners

Get the corners of the Polyhedron.

The method `vertices` is an alias for `corners`.

Examples

```
>>> from sympy.combinatorics import Polyhedron
>>> from sympy.abc import a, b, c, d
>>> p = Polyhedron(list('abcd'))
>>> p.corners == p.vertices == (a, b, c, d)
True
```

See also:

[`array\_form`](#) (page 332), [`cyclic\_form`](#) (page 333)

property cyclic_form

Return the indices of the corners in cyclic notation.

The indices are given relative to the original position of corners.

See also:

[`corners`](#) (page 333), [`array\_form`](#) (page 332)

property edges

Given the faces of the polyhedra we can get the edges.

Examples

```
>>> from sympy.combinatorics import Polyhedron
>>> from sympy.abc import a, b, c
>>> corners = (a, b, c)
>>> faces = [(0, 1, 2)]
>>> Polyhedron(corners, faces).edges
{(0, 1), (0, 2), (1, 2)}
```

property faces

Get the faces of the Polyhedron.

property pgroup

Get the permutations of the Polyhedron.

reset()

Return corners to their original positions.

Examples

```
>>> from sympy.combinatorics.polyhedron import tetrahedron as T
>>> T = T.copy()
>>> T.corners
(0, 1, 2, 3)
>>> T.rotate(0)
>>> T.corners
(0, 2, 3, 1)
>>> T.reset()
>>> T.corners
(0, 1, 2, 3)
```

`rotate(perm)`

Apply a permutation to the polyhedron *in place*. The permutation may be given as a Permutation instance or an integer indicating which permutation from pgroup of the Polyhedron should be applied.

This is an operation that is analogous to rotation about an axis by a fixed increment.

Notes

When a Permutation is applied, no check is done to see if that is a valid permutation for the Polyhedron. For example, a cube could be given a permutation which effectively swaps only 2 vertices. A valid permutation (that rotates the object in a physical way) will be obtained if one only uses permutations from the pgroup of the Polyhedron. On the other hand, allowing arbitrary rotations (applications of permutations) gives a way to follow named elements rather than indices since Polyhedron allows vertices to be named while Permutation works only with indices.

Examples

```
>>> from sympy.combinatorics import Polyhedron, Permutation
>>> from sympy.combinatorics.polyhedron import cube
>>> cube = cube.copy()
>>> cube.corners
(0, 1, 2, 3, 4, 5, 6, 7)
>>> cube.rotate(0)
>>> cube.corners
(1, 2, 3, 0, 5, 6, 7, 4)
```

A non-physical “rotation” that is not prohibited by this method:

```
>>> cube.reset()
>>> cube.rotate(Permutation([[1, 2]], size=8))
>>> cube.corners
(0, 2, 1, 3, 4, 5, 6, 7)
```

Polyhedron can be used to follow elements of set that are identified by letters instead of integers:

```

>>> shadow = h5 = Polyhedron(list('abcde'))
>>> p = Permutation([3, 0, 1, 2, 4])
>>> h5.rotate(p)
>>> h5.corners
(d, a, b, c, e)
>>> _ == shadow.corners
True
>>> copy = h5.copy()
>>> h5.rotate(p)
>>> h5.corners == copy.corners
False

```

property size

Get the number of corners of the Polyhedron.

property vertices

Get the corners of the Polyhedron.

The method vertices is an alias for corners.

Examples

```

>>> from sympy.combinatorics import Polyhedron
>>> from sympy.abc import a, b, c, d
>>> p = Polyhedron(list('abcd'))
>>> p.corners == p.vertices == (a, b, c, d)
True

```

See also:

[array_form](#) (page 332), [cyclic_form](#) (page 333)

Prufer Sequences

class `sympy.combinatorics.prufer.Prufer(*args, **kw_args)`

The Prufer correspondence is an algorithm that describes the bijection between labeled trees and the Prufer code. A Prufer code of a labeled tree is unique up to isomorphism and has a length of $n - 2$.

Prufer sequences were first used by Heinz Prufer to give a proof of Cayley's formula.

References

[R82]

static edges(*runs)

Return a list of edges and the number of nodes from the given runs that connect nodes in an integer-labelled tree.

All node numbers will be shifted so that the minimum node is 0. It is not a problem if edges are repeated in the runs; only unique edges are returned. There is no

assumption made about what the range of the node labels should be, but all nodes from the smallest through the largest must be present.

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> Prufer.edges([1, 2, 3], [2, 4, 5]) # a T
([[0, 1], [1, 2], [1, 3], [3, 4]], 5)
```

Duplicate edges are removed:

```
>>> Prufer.edges([0, 1, 2, 3], [1, 4, 5], [1, 4, 6]) # a K
([[0, 1], [1, 2], [1, 4], [2, 3], [4, 5], [4, 6]], 7)
```

next(delta=1)

Generates the Prufer sequence that is delta beyond the current one.

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> a = Prufer([[0, 1], [0, 2], [0, 3]])
>>> b = a.next(1) # == a.next()
>>> b.tree_repr
[[0, 2], [0, 1], [1, 3]]
>>> b.rank
1
```

See also:

[*prufer_rank*](#) (page 337), [*rank*](#) (page 337), [*prev*](#) (page 336), [*size*](#) (page 338)

property nodes

Returns the number of nodes in the tree.

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> Prufer([[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]]).nodes
6
>>> Prufer([1, 0, 0]).nodes
5
```

prev(delta=1)

Generates the Prufer sequence that is -delta before the current one.

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> a = Prufer([[0, 1], [1, 2], [2, 3], [1, 4]])
>>> a.rank
36
>>> b = a.prev()
>>> b
Prufer([1, 2, 0])
>>> b.rank
35
```

See also:

[prufer_rank](#) (page 337), [rank](#) (page 337), [next](#) (page 336), [size](#) (page 338)

`prufer_rank()`

Computes the rank of a Prufer sequence.

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> a = Prufer([[0, 1], [0, 2], [0, 3]])
>>> a.prufer_rank()
0
```

See also:

[rank](#) (page 337), [next](#) (page 336), [prev](#) (page 336), [size](#) (page 338)

`property prufer_repr`

Returns Prufer sequence for the Prufer object.

This sequence is found by removing the highest numbered vertex, recording the node it was attached to, and continuing until only two vertices remain. The Prufer sequence is the list of recorded nodes.

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> Prufer([[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]]).prufer_repr
[3, 3, 3, 4]
>>> Prufer([1, 0, 0]).prufer_repr
[1, 0, 0]
```

See also:

[to_prufer](#) (page 338)

`property rank`

Returns the rank of the Prufer sequence.

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> p = Prufer([[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]])
>>> p.rank
778
>>> p.next(1).rank
779
>>> p.prev().rank
777
```

See also:

[prufer_rank](#) (page 337), [next](#) (page 336), [prev](#) (page 336), [size](#) (page 338)

property size

Return the number of possible trees of this Prufer object.

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> Prufer([0]*4).size == Prufer([6]*4).size == 1296
True
```

See also:

[prufer_rank](#) (page 337), [rank](#) (page 337), [next](#) (page 336), [prev](#) (page 336)

static to_prufer(*tree*, *n*)

Return the Prufer sequence for a tree given as a list of edges where *n* is the number of nodes in the tree.

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> a = Prufer([[0, 1], [0, 2], [0, 3]])
>>> a.prufer_repr
[0, 0]
>>> Prufer.to_prufer([[0, 1], [0, 2], [0, 3]], 4)
[0, 0]
```

See also:

[prufer_repr](#) (page 337)

returns Prufer sequence of a Prufer object.

static to_tree(*prufer*)

Return the tree (as a list of edges) of the given Prufer sequence.

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> a = Prufer([0, 2], 4)
>>> a.tree_repr
[[0, 1], [0, 2], [2, 3]]
>>> Prufer.to_tree([0, 2])
[[0, 1], [0, 2], [2, 3]]
```

See also:

[tree_repr](#) (page 339)

returns tree representation of a Prufer object.

References

[R83]

property `tree_repr`

Returns the tree representation of the Prufer object.

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> Prufer([[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]]).tree_repr
[[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]]
>>> Prufer([1, 0, 0]).tree_repr
[[1, 2], [0, 1], [0, 3], [0, 4]]
```

See also:

[to_tree](#) (page 338)

classmethod `unrank(rank, n)`

Finds the unranked Prufer sequence.

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> Prufer.unrank(0, 4)
Prufer([0, 0])
```

Subsets

class `sympy.combinatorics.subsets.Subset`(*subset*, *superset*)

Represents a basic subset object.

Explanation

We generate subsets using essentially two techniques, binary enumeration and lexicographic enumeration. The `Subset` class takes two arguments, the first one describes the initial subset to consider and the second describes the superset.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.next_binary().subset
['b']
>>> a.prev_binary().subset
['c']
```

classmethod `bitlist_from_subset`(*subset*, *superset*)

Gets the bitlist corresponding to a subset.

Examples

```
>>> from sympy.combinatorics import Subset
>>> Subset.bitlist_from_subset(['c', 'd'], ['a', 'b', 'c', 'd'])
'0011'
```

See also:

[`subset\_from\_bitlist`](#) (page 345)

property `cardinality`

Returns the number of all possible subsets.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.cardinality
16
```

See also:

[`subset`](#) (page 344), [`superset`](#) (page 345), [`size`](#) (page 344), [`superset\_size`](#) (page 346)

iterate_binary(*k*)

This is a helper function. It iterates over the binary subsets by *k* steps. This variable can be both positive or negative.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.iterate_binary(-2).subset
['d']
>>> a = Subset(['a', 'b', 'c'], ['a', 'b', 'c', 'd'])
>>> a.iterate_binary(2).subset
[]
```

See also:

[next_binary](#) (page 341), [prev_binary](#) (page 342)

iterate_graycode(*k*)

Helper function used for `prev_gray` and `next_gray`. It performs *k* step overs to get the respective Gray codes.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset([1, 2, 3], [1, 2, 3, 4])
>>> a.iterate_graycode(3).subset
[1, 4]
>>> a.iterate_graycode(-2).subset
[1, 2, 4]
```

See also:

[next_gray](#) (page 341), [prev_gray](#) (page 342)

next_binary()

Generates the next binary ordered subset.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.next_binary().subset
['b']
>>> a = Subset(['a', 'b', 'c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.next_binary().subset
[]
```

See also:

[prev_binary](#) (page 342), [iterate_binary](#) (page 340)

next_gray()

Generates the next Gray code ordered subset.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset([1, 2, 3], [1, 2, 3, 4])
>>> a.next_gray().subset
[1, 3]
```

See also:

[*iterate_graycode*](#) (page 341), [*prev_gray*](#) (page 342)

next_lexicographic()

Generates the next lexicographically ordered subset.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.next_lexicographic().subset
['d']
>>> a = Subset(['d'], ['a', 'b', 'c', 'd'])
>>> a.next_lexicographic().subset
[]
```

See also:

[*prev_lexicographic*](#) (page 343)

prev_binary()

Generates the previous binary ordered subset.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset([], ['a', 'b', 'c', 'd'])
>>> a.prev_binary().subset
['a', 'b', 'c', 'd']
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.prev_binary().subset
['c']
```

See also:

[*next_binary*](#) (page 341), [*iterate_binary*](#) (page 340)

prev_gray()

Generates the previous Gray code ordered subset.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset([2, 3, 4], [1, 2, 3, 4, 5])
>>> a.prev_gray().subset
[2, 3, 4, 5]
```

See also:

[*iterate_graycode*](#) (page 341), [*next_gray*](#) (page 341)

prev_lexicographic()

Generates the previous lexicographically ordered subset.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset([], ['a', 'b', 'c', 'd'])
>>> a.prev_lexicographic().subset
['d']
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.prev_lexicographic().subset
['c']
```

See also:

[*next_lexicographic*](#) (page 342)

property rank_binary

Computes the binary ordered rank.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset([], ['a', 'b', 'c', 'd'])
>>> a.rank_binary
0
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.rank_binary
3
```

See also:

[*iterate_binary*](#) (page 340), [*unrank_binary*](#) (page 346)

property rank_gray

Computes the Gray code ranking of the subset.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.rank_gray
2
>>> a = Subset([2, 4, 5], [1, 2, 3, 4, 5, 6])
>>> a.rank_gray
27
```

See also:

[*iterate_graycode*](#) (page 341), [*unrank_gray*](#) (page 346)

property `rank_lexicographic`

Computes the lexicographic ranking of the subset.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.rank_lexicographic
14
>>> a = Subset([2, 4, 5], [1, 2, 3, 4, 5, 6])
>>> a.rank_lexicographic
43
```

property `size`

Gets the size of the subset.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.size
2
```

See also:

[*subset*](#) (page 344), [*superset*](#) (page 345), [*superset_size*](#) (page 346), [*cardinality*](#) (page 340)

property `subset`

Gets the subset represented by the current instance.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.subset
['c', 'd']
```

See also:

[superset](#) (page 345), [size](#) (page 344), [superset_size](#) (page 346), [cardinality](#) (page 340)

classmethod `subset_from_bitlist(super_set, bitlist)`

Gets the subset defined by the bitlist.

Examples

```
>>> from sympy.combinatorics import Subset
>>> Subset.subset_from_bitlist(['a', 'b', 'c', 'd'], '0011').subset
['c', 'd']
```

See also:

[bitlist_from_subset](#) (page 340)

classmethod `subset_indices(subset, superset)`

Return indices of subset in superset in a list; the list is empty if all elements of subset are not in superset.

Examples

```
>>> from sympy.combinatorics import Subset
>>> superset = [1, 3, 2, 5, 4]
>>> Subset.subset_indices([3, 2, 1], superset)
[1, 2, 0]
>>> Subset.subset_indices([1, 6], superset)
[]
>>> Subset.subset_indices([], superset)
[]
```

property `superset`

Gets the superset of the subset.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.superset
['a', 'b', 'c', 'd']
```

See also:

[subset](#) (page 344), [size](#) (page 344), [superset_size](#) (page 346), [cardinality](#) (page 340)

property `superset_size`

Returns the size of the superset.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.superset_size
4
```

See also:

[subset](#) (page 344), [superset](#) (page 345), [size](#) (page 344), [cardinality](#) (page 340)

classmethod `unrank_binary(rank, superset)`

Gets the binary ordered subset of the specified rank.

Examples

```
>>> from sympy.combinatorics import Subset
>>> Subset.unrank_binary(4, ['a', 'b', 'c', 'd']).subset
['b']
```

See also:

[iterate_binary](#) (page 340), [rank_binary](#) (page 343)

classmethod `unrank_gray(rank, superset)`

Gets the Gray code ordered subset of the specified rank.

Examples

```
>>> from sympy.combinatorics import Subset
>>> Subset.unrank_gray(4, ['a', 'b', 'c']).subset
['a', 'b']
>>> Subset.unrank_gray(0, ['a', 'b', 'c']).subset
[]
```

See also:

[iterate_graycode](#) (page 341), [rank_gray](#) (page 343)

`subsets.ksubsets(k)`

Finds the subsets of size k in lexicographic order.

This uses the `itertools` generator.

Examples

```
>>> from sympy.combinatorics.subsets import ksubsets
>>> list(ksubsets([1, 2, 3], 2))
[(1, 2), (1, 3), (2, 3)]
>>> list(ksubsets([1, 2, 3, 4, 5], 2))
[(1, 2), (1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5)]
```

See also:

[Subset](#) (page 340)

Gray Code

class `sympy.combinatorics.graycode.GrayCode(n, *args, **kw_args)`

A Gray code is essentially a Hamiltonian walk on a n -dimensional cube with edge length of one. The vertices of the cube are represented by vectors whose values are binary. The Hamilton walk visits each vertex exactly once. The Gray code for a 3d cube is ['000', '100', '110', '010', '011', '111', '101', '001'].

A Gray code solves the problem of sequentially generating all possible subsets of n objects in such a way that each subset is obtained from the previous one by either deleting or adding a single object. In the above example, 1 indicates that the object is present, and 0 indicates that its absent.

Gray codes have applications in statistics as well when we want to compute various statistics related to subsets in an efficient manner.

Examples

```
>>> from sympy.combinatorics import GrayCode
>>> a = GrayCode(3)
>>> list(a.generate_gray())
['000', '100', '110', '010', '011', '111', '101', '001']
>>> a = GrayCode(4)
>>> list(a.generate_gray())
['0000', '0001', '0011', '0010', '0110', '0111', '0101', '0100',
 '1100', '1101', '1111', '1110', '1010', '1011', '1001', '1000']
```

References

[R39], [R40]

property `current`

Returns the currently referenced Gray code as a bit string.

Examples

```
>>> from sympy.combinatorics import GrayCode
>>> GrayCode(3, start='100').current
'100'
```

`generate_gray(**hints)`

Generates the sequence of bit vectors of a Gray Code.

Examples

```
>>> from sympy.combinatorics import GrayCode
>>> a = GrayCode(3)
>>> list(a.generate_gray())
['000', '001', '011', '010', '110', '111', '101', '100']
>>> list(a.generate_gray(start='011'))
['011', '010', '110', '111', '101', '100']
>>> list(a.generate_gray(rank=4))
['110', '111', '101', '100']
```

See also:

[*skip*](#) (page 349)

References

[R41]

property `n`

Returns the dimension of the Gray code.

Examples

```
>>> from sympy.combinatorics import GrayCode
>>> a = GrayCode(5)
>>> a.n
5
```

`next(delta=1)`

Returns the Gray code a distance `delta` (default = 1) from the current value in canonical order.

Examples

```
>>> from sympy.combinatorics import GrayCode
>>> a = GrayCode(3, start='110')
>>> a.next().current
'111'
>>> a.next(-1).current
'010'
```

property rank

Ranks the Gray code.

A ranking algorithm determines the position (or rank) of a combinatorial object among all the objects w.r.t. a given order. For example, the 4 bit binary reflected Gray code (BRGC) '0101' has a rank of 6 as it appears in the 6th position in the canonical ordering of the family of 4 bit Gray codes.

Examples

```
>>> from sympy.combinatorics import GrayCode
>>> a = GrayCode(3)
>>> list(a.generate_gray())
['000', '001', '011', '010', '110', '111', '101', '100']
>>> GrayCode(3, start='100').rank
7
>>> GrayCode(3, rank=7).current
'100'
```

See also:

[unrank](#) (page 350)

References

[R42]

property selections

Returns the number of bit vectors in the Gray code.

Examples

```
>>> from sympy.combinatorics import GrayCode
>>> a = GrayCode(3)
>>> a.selections
8
```

skip()

Skips the bit generation.

Examples

```
>>> from sympy.combinatorics import GrayCode
>>> a = GrayCode(3)
>>> for i in a.generate_gray():
...     if i == '010':
...         a.skip()
...     print(i)
...
000
001
011
010
111
101
100
```

See also:

[*generate_gray*](#) (page 348)

classmethod unrank(*n*, *rank*)

Unranks an *n*-bit sized Gray code of rank *k*. This method exists so that a derivative GrayCode class can define its own code of a given rank.

The string here is generated in reverse order to allow for tail-call optimization.

Examples

```
>>> from sympy.combinatorics import GrayCode
>>> GrayCode(5, rank=3).current
'00010'
>>> GrayCode.unrank(5, 3)
'00010'
```

See also:

[*rank*](#) (page 349)

graycode.random_bitstring()

Generates a random bitlist of length *n*.

Examples

```
>>> from sympy.combinatorics.graycode import random_bitstring
>>> random_bitstring(3)
100
```

graycode.gray_to_bin()

Convert from Gray coding to binary coding.

We assume big endian encoding.

Examples

```
>>> from sympy.combinatorics.graycode import gray_to_bin
>>> gray_to_bin('100')
'111'
```

See also:

[*bin_to_gray*](#) (page 351)

`graycode.bin_to_gray()`

Convert from binary coding to gray coding.

We assume big endian encoding.

Examples

```
>>> from sympy.combinatorics.graycode import bin_to_gray
>>> bin_to_gray('111')
'100'
```

See also:

[*gray_to_bin*](#) (page 350)

`graycode.get_subset_from_bitstring(bitstring)`

Gets the subset defined by the bitstring.

Examples

```
>>> from sympy.combinatorics.graycode import get_subset_from_bitstring
>>> get_subset_from_bitstring(['a', 'b', 'c', 'd'], '0011')
['c', 'd']
>>> get_subset_from_bitstring(['c', 'a', 'c', 'c'], '1100')
['c', 'a']
```

See also:

[*graycode_subsets*](#) (page 351)

`graycode.graycode_subsets()`

Generates the subsets as enumerated by a Gray code.

Examples

```
>>> from sympy.combinatorics.graycode import graycode_subsets
>>> list(graycode_subsets(['a', 'b', 'c']))
[[], ['c'], ['b', 'c'], ['b'], ['a', 'b'], ['a', 'b', 'c'], ['a', 'c'],
 → ['a']]
>>> list(graycode_subsets(['a', 'b', 'c', 'c']))
[[[], ['c'], ['c', 'c'], ['c'], ['b', 'c'], ['b', 'c', 'c'], ['b', 'c', 'c', 'c']]]
```

(continues on next page)

(continued from previous page)

```
→ ''], ['b'], ['a', 'b'], ['a', 'b', 'c'], ['a', 'b', 'c', 'c'],      ['a',  
→ 'b', 'c'], ['a', 'c'], ['a', 'c', 'c'], ['a', 'c'], ['a']]
```

See also:[*get_subset_from_bitstring*](#) (page 351)

Named Groups

`sympy.combinatorics.named_groups.SymmetricGroup(n)`Generates the symmetric group on *n* elements as a permutation group.**Explanation**

The generators taken are the *n*-cycle (0 1 2 ... *n*-1) and the transposition (0 1) (in cycle notation). (See [1]). After the group is generated, some of its basic properties are set.

Examples

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup  
>>> G = SymmetricGroup(4)  
>>> G.is_group  
True  
>>> G.order()  
24  
>>> list(G.generate_schreier_sims(af=True))  
[[0, 1, 2, 3], [1, 2, 3, 0], [2, 3, 0, 1], [3, 1, 2, 0], [0, 2, 3, 1],  
[1, 3, 0, 2], [2, 0, 1, 3], [3, 2, 0, 1], [0, 3, 1, 2], [1, 0, 2, 3],  
[2, 1, 3, 0], [3, 0, 1, 2], [0, 1, 3, 2], [1, 2, 0, 3], [2, 3, 1, 0],  
[3, 1, 0, 2], [0, 2, 1, 3], [1, 3, 2, 0], [2, 0, 3, 1], [3, 2, 1, 0],  
[0, 3, 2, 1], [1, 0, 3, 2], [2, 1, 0, 3], [3, 0, 2, 1]]
```

See also:[*CyclicGroup*](#) (page 352), [*DihedralGroup*](#) (page 353), [*AlternatingGroup*](#) (page 354)**References**[\[R46\]](#)`sympy.combinatorics.named_groups.CyclicGroup(n)`Generates the cyclic group of order *n* as a permutation group.

Explanation

The generator taken is the n -cycle $(0\ 1\ 2\ \dots\ n-1)$ (in cycle notation). After the group is generated, some of its basic properties are set.

Examples

```
>>> from sympy.combinatorics.named_groups import CyclicGroup
>>> G = CyclicGroup(6)
>>> G.is_group
True
>>> G.order()
6
>>> list(G.generate_schreier_sims(af=True))
[[0, 1, 2, 3, 4, 5], [1, 2, 3, 4, 5, 0], [2, 3, 4, 5, 0, 1],
 [3, 4, 5, 0, 1, 2], [4, 5, 0, 1, 2, 3], [5, 0, 1, 2, 3, 4]]
```

See also:

[SymmetricGroup](#) (page 352), [DihedralGroup](#) (page 353), [AlternatingGroup](#) (page 354)

`sympy.combinatorics.named_groups.DihedralGroup( $n$ )`

Generates the dihedral group D_n as a permutation group.

Explanation

The dihedral group D_n is the group of symmetries of the regular n -gon. The generators taken are the n -cycle $a = (0\ 1\ 2\ \dots\ n-1)$ (a rotation of the n -gon) and $b = (0\ n-1)(1\ n-2)\dots$ (a reflection of the n -gon) in cycle notation. It is easy to see that these satisfy $a^n = b^2 = 1$ and $bab = a^{-1}$ so they indeed generate D_n (See [1]). After the group is generated, some of its basic properties are set.

Examples

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> G = DihedralGroup(5)
>>> G.is_group
True
>>> a = list(G.generate_dimino())
>>> [perm.cyclic_form for perm in a]
[[], [[0, 1, 2, 3, 4]], [[0, 2, 4, 1, 3]],
 [[0, 3, 1, 4, 2]], [[0, 4, 3, 2, 1]], [[0, 4], [1, 3]],
 [[1, 4], [2, 3]], [[0, 1], [2, 4]], [[0, 2], [3, 4]],
 [[0, 3], [1, 2]]]
```

See also:

[SymmetricGroup](#) (page 352), [CyclicGroup](#) (page 352), [AlternatingGroup](#) (page 354)

References

[R47]

`sympy.combinatorics.named_groups.AlternatingGroup(n)`

Generates the alternating group on n elements as a permutation group.

Explanation

For $n > 2$, the generators taken are $(0\ 1\ 2)$, $(0\ 1\ 2\ \dots\ n-1)$ for n odd and $(0\ 1\ 2)$, $(1\ 2\ \dots\ n-1)$ for n even (See [1], p.31, ex.6.9.). After the group is generated, some of its basic properties are set. The cases $n = 1, 2$ are handled separately.

Examples

```
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> G = AlternatingGroup(4)
>>> G.is_group
True
>>> a = list(G.generate_dimino())
>>> len(a)
12
>>> all(perm.is_even for perm in a)
True
```

See also:

[SymmetricGroup](#) (page 352), [CyclicGroup](#) (page 352), [DihedralGroup](#) (page 353)

References

[R48]

`sympy.combinatorics.named_groups.AbelianGroup(*cyclic_orders)`

Returns the direct product of cyclic groups with the given orders.

Explanation

According to the structure theorem for finite abelian groups ([1]), every finite abelian group can be written as the direct product of finitely many cyclic groups.

Examples

```
>>> from sympy.combinatorics.named_groups import AbelianGroup
>>> AbelianGroup(3, 4)
PermutationGroup([
    (6)(0 1 2),
    (3 4 5 6)])
>>> _.is_group
True
```

See also:

[DirectProduct](#) (page 363)

References

[R49]

Nilpotent, Abelian and Cyclic Numbers

`sympy.combinatorics.group_numbers.is_nilpotent_number(n)`

Check whether n is a nilpotent number. A number n is said to be nilpotent if and only if every finite group of order n is nilpotent. For more information see [R43].

Examples

```
>>> from sympy.combinatorics.group_numbers import is_nilpotent_number
>>> from sympy import randprime
>>> is_nilpotent_number(21)
False
>>> is_nilpotent_number(randprime(1, 30)**12)
True
```

References

[R43]

`sympy.combinatorics.group_numbers.is_abelian_number(n)`

Check whether n is an abelian number. A number n is said to be abelian if and only if every finite group of order n is abelian. For more information see [R44].

Examples

```
>>> from sympy.combinatorics.group_numbers import is_abelian_number
>>> from sympy import randprime
>>> is_abelian_number(4)
True
>>> is_abelian_number(randprime(1, 2000)**2)
True
>>> is_abelian_number(60)
False
```

References

[R44]

`sympy.combinatorics.group_numbers.is_cyclic_number(n)`

Check whether n is a cyclic number. A number n is said to be cyclic if and only if every finite group of order n is cyclic. For more information see [R45].

Examples

```
>>> from sympy.combinatorics.group_numbers import is_cyclic_number
>>> from sympy import randprime
>>> is_cyclic_number(15)
True
>>> is_cyclic_number(randprime(1, 2000)**2)
False
>>> is_cyclic_number(4)
False
```

References

[R45]

Utilities

`sympy.combinatorics.util._base_ordering(base, degree)`

Order $\{0, 1, \dots, n-1\}$ so that base points come first and in order.

Parameters

```base``` : the base

```degree``` : the degree of the associated permutation group

Returns

A list `base_ordering` such that `base_ordering[point]` is the number of point in the ordering.

Examples

```
>>> from sympy.combinatorics import SymmetricGroup
>>> from sympy.combinatorics.util import _base_ordering
>>> S = SymmetricGroup(4)
>>> S.schreier_sims()
>>> _base_ordering(S.base, S.degree)
[0, 1, 2, 3]
```

Notes

This is used in backtrack searches, when we define a relation $\ll$ on the underlying set for a permutation group of degree n , $\{0, 1, \dots, n-1\}$, so that if $(b_1, b_2, \dots, b_k)$ is a base we have $b_i \ll b_j$ whenever $i < j$ and $b_i \ll a$ for all $i \in \{1, 2, \dots, k\}$ and a is not in the base. The idea is developed and applied to backtracking algorithms in [1], pp.108-132. The points that are not in the base are taken in increasing order.

References

[R84]

`sympy.combinatorics.util._check_cycles_alt_sym(perm)`

Checks for cycles of prime length p with $n/2 < p < n-2$.

Explanation

Here n is the degree of the permutation. This is a helper function for the function `is_alt_sym` from `sympy.combinatorics.perm_groups`.

Examples

```
>>> from sympy.combinatorics.util import _check_cycles_alt_sym
>>> from sympy.combinatorics import Permutation
>>> a = Permutation([[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [11, 12]])
>>> _check_cycles_alt_sym(a)
False
>>> b = Permutation([[0, 1, 2, 3, 4, 5, 6], [7, 8, 9, 10]])
>>> _check_cycles_alt_sym(b)
True
```

See also:

`sympy.combinatorics.perm_groups.PermutationGroup.is_alt_sym` (page 310)

`sympy.combinatorics.util._distribute_gens_by_base(base, gens)`

Distribute the group elements *gens* by membership in basic stabilizers.

Parameters

```base``` : a sequence of points in  $\{0, 1, \dots, n-1\}$

```gens``` : a list of elements of a permutation group of degree  $n$ .

Returns

List of length k , where k is the length of base. The i -th entry contains those elements in gens which fix the first i elements of base (so that the 0-th entry is equal to gens itself). If no element fixes the first i elements of base, the i -th element is set to a list containing the identity element.

Explanation

Notice that for a base $(b_1, b_2, \dots, b_k)$, the basic stabilizers are defined as $G^{(i)} = G_{b_1, \dots, b_{i-1}}$ for $i \in \{1, 2, \dots, k\}$.

Examples

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> from sympy.combinatorics.util import _distribute_gens_by_base
>>> D = DihedralGroup(3)
>>> D.schreier_sims()
>>> D.strong_gens
[(0 1 2), (0 2), (1 2)]
>>> D.base
[0, 1]
>>> _distribute_gens_by_base(D.base, D.strong_gens)
[[[(0 1 2), (0 2), (1 2)],
 [(1 2)]]
```

See also:

[`\_strong\_gens\_from\_distr`](#) (page 362), [`\_orbits\_transversals\_from\_bsgs`](#) (page 359), [`\_handle\_precomputed\_bsgs`](#) (page 358)

```
sympy.combinatorics.util._handle_precomputed_bsgs(base, strong_gens,
                                                    transversals=None,
                                                    basic_orbits=None,
                                                    strong_gens_distr=None)
```

Calculate BSGS-related structures from those present.

Parameters

```base``` - the base  
```strong_gens``` - the strong generators  
```transversals``` - basic transversals  
```basic_orbits``` - basic orbits  
```strong_gens_distr``` - strong generators distributed by membership in basic stabilizers

**Returns**

(transversals, basic\_orbits, strong\_gens\_distr) where transversals

are the basic transversals, basic\_orbits are the basic orbits, and strong\_gens\_distr are the strong generators distributed by membership in basic stabilizers.

**Explanation**

The base and strong generating set must be provided; if any of the transversals, basic orbits or distributed strong generators are not provided, they will be calculated from the base and strong generating set.

**Examples**

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> from sympy.combinatorics.util import _handle_precomputed_bsgs
>>> D = DihedralGroup(3)
>>> D.schreier_sims()
>>> _handle_precomputed_bsgs(D.base, D.strong_gens,
... basic_orbits=D.basic_orbits)
([{0: (2), 1: (0 1 2), 2: (0 2)}, {1: (2), 2: (1 2)}], [[0, 1, 2], [1, 2], [2, 0]], [[(0 1 2), (0 2), (1 2)], [(1 2)]])
```

**See also:**

[\\_orbits\\_transversals\\_from\\_bsgs](#) (page 359), [\\_distribute\\_gens\\_by\\_base](#) (page 357)

`sympy.combinatorics.util._orbits_transversals_from_bsgs(base, strong_gens_distr, transversals_only=False, slp=False)`

Compute basic orbits and transversals from a base and strong generating set.

**Parameters**

**``base``** - The base.

**``strong\_gens\_distr``** - Strong generators distributed by membership in basic

stabilizers.

**``transversals\_only``** - bool

A flag switching between returning only the transversals and both orbits and transversals.

**``slp``** -

If True, return a list of dictionaries containing the generator presentations of the elements of the transversals, i.e. the list of indices of generators from `strong_gens_distr[i]` such that their product is the relevant transversal element.

## Explanation

The generators are provided as distributed across the basic stabilizers. If the optional argument `transversals_only` is set to `True`, only the transversals are returned.

## Examples

```
>>> from sympy.combinatorics import SymmetricGroup
>>> from sympy.combinatorics.util import _distribute_gens_by_base
>>> S = SymmetricGroup(3)
>>> S.schreier_sims()
>>> strong_gens_distr = _distribute_gens_by_base(S.base, S.strong_gens)
>>> (S.base, strong_gens_distr)
([0, 1], [[(0 1 2), (2)(0 1), (1 2)], [(1 2)]])
```

### See also:

[`\_distribute\_gens\_by\_base`](#) (page 357), [`\_handle\_precomputed\_bsgs`](#) (page 358)

`sympy.combinatorics.util._remove_gens(base, strong_gens, basic_orbits=None, strong_gens_distr=None)`

Remove redundant generators from a strong generating set.

### Parameters

```base``` - a base

```strong_gens``` - a strong generating set relative to ```base```

```basic_orbits``` - basic orbits

```strong_gens_distr``` - strong generators distributed by membership in basic

stabilizers

### Returns

A strong generating set with respect to `base` which is a subset of `strong_gens`.

## Examples

```
>>> from sympy.combinatorics import SymmetricGroup
>>> from sympy.combinatorics.util import _remove_gens
>>> from sympy.combinatorics.testutil import _verify_bsgs
>>> S = SymmetricGroup(15)
>>> base, strong_gens = S.schreier_sims_incremental()
>>> new_gens = _remove_gens(base, strong_gens)
>>> len(new_gens)
14
>>> _verify_bsgs(S, base, new_gens)
True
```



## Notes

This procedure is outlined in [1],p.95.

## References

[R85]

`sympy.combinatorics.util._strip(g, base, orbits, transversals)`

Attempt to decompose a permutation using a (possibly partial) BSGS structure.

### Parameters

`g` - permutation to be decomposed

`base` - sequence of points

`orbits` - a list in which the `i`-th entry is an orbit of `base[i]` under some subgroup of the pointwise stabilizer of

`base[0], base[1], ..., base[i - 1]`. The groups themselves are implicit

in this function since the only information we need is encoded in the orbits

and transversals

`transversals` - a list of orbit transversals associated with the orbits

`orbits`.

## Explanation

This is done by treating the sequence `base` as an actual base, and the orbits `orbits` and transversals `transversals` as basic orbits and transversals relative to it.

This process is called “sifting”. A sift is unsuccessful when a certain orbit element is not found or when after the sift the decomposition does not end with the identity element.

The argument `transversals` is a list of dictionaries that provides transversal elements for the orbits `orbits`.

## Examples

```
>>> from sympy.combinatorics import Permutation, SymmetricGroup
>>> from sympy.combinatorics.util import _strip
>>> S = SymmetricGroup(5)
>>> S.schreier_sims()
>>> g = Permutation([0, 2, 3, 1, 4])
>>> _strip(g, S.base, S.basic_orbits, S.basic_transversals)
((4), 5)
```

## Notes

The algorithm is described in [1], pp.89-90. The reason for returning both the current state of the element being decomposed and the level at which the sifting ends is that they provide important information for the randomized version of the Schreier-Sims algorithm.

### See also:

[\*sympy.combinatorics.perm\\_groups.PermutationGroup.schreier\\_sims\*](#) (page 324),  
[\*sympy.combinatorics.perm\\_groups.PermutationGroup.schreier\\_sims\\_random\*](#)  
(page 326)

## References

[R86]

`sympy.combinatorics.util._strong_gens_from_distr(strong_gens_distr)`

Retrieve strong generating set from generators of basic stabilizers.

This is just the union of the generators of the first and second basic stabilizers.

### Parameters

```strong_gens_distr``` - strong generators distributed by membership  
in basic
stabilizers

Examples

```
>>> from sympy.combinatorics import SymmetricGroup
>>> from sympy.combinatorics.util import (_strong_gens_from_distr,
... _distribute_gens_by_base)
>>> S = SymmetricGroup(3)
>>> S.schreier_sims()
>>> S.strong_gens
[(0 1 2), (2)(0 1), (1 2)]
>>> strong_gens_distr = _distribute_gens_by_base(S.base, S.strong_gens)
>>> _strong_gens_from_distr(strong_gens_distr)
[(0 1 2), (2)(0 1), (1 2)]
```

See also:

[*_distribute_gens_by_base*](#) (page 357)

Group constructors

`sympy.combinatorics.group_constructs.DirectProduct(*groups)`

Returns the direct product of several groups as a permutation group.

Explanation

This is implemented much like the `__mul__` procedure for taking the direct product of two permutation groups, but the idea of shifting the generators is realized in the case of an arbitrary number of groups. A call to `DirectProduct(G1, G2, ..., Gn)` is generally expected to be faster than a call to `G1*G2*...*Gn` (and thus the need for this algorithm).

Examples

```
>>> from sympy.combinatorics.group_constructs import DirectProduct
>>> from sympy.combinatorics.named_groups import CyclicGroup
>>> C = CyclicGroup(4)
>>> G = DirectProduct(C, C, C)
>>> G.order()
64
```

See also:

`sympy.combinatorics.perm_groups.PermutationGroup.__mul__` (page 289)

Test Utilities

`sympy.combinatorics.testutil._cmp_perm_lists(first, second)`

Compare two lists of permutations as sets.

Explanation

This is used for testing purposes. Since the array form of a permutation is currently a list, `Permutation` is not hashable and cannot be put into a set.

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.testutil import _cmp_perm_lists
>>> a = Permutation([0, 2, 3, 4, 1])
>>> b = Permutation([1, 2, 0, 4, 3])
>>> c = Permutation([3, 4, 0, 1, 2])
>>> ls1 = [a, b, c]
>>> ls2 = [b, c, a]
>>> _cmp_perm_lists(ls1, ls2)
True
```

`sympy.combinatorics.testutil._naive_list_centralizer(self, other, af=False)`

`sympy.combinatorics.testutil._verify_bsgs(group, base, gens)`

Verify the correctness of a base and strong generating set.

Explanation

This is a naive implementation using the definition of a base and a strong generating set relative to it. There are other procedures for verifying a base and strong generating set, but this one will serve for more robust testing.

Examples

```
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> from sympy.combinatorics.testutil import _verify_bsgs
>>> A = AlternatingGroup(4)
>>> A.schreier_sims()
>>> _verify_bsgs(A, A.base, A.strong_gens)
True
```

See also:

[`sympy.combinatorics.perm\_groups.PermutationGroup.schreier\_sims`](#) (page 324)

`sympy.combinatorics.testutil._verify_centralizer(group, arg, centr=None)`

Verify the centralizer of a group/set/element inside another group.

This is used for testing `.centralizer()` from `sympy.combinatorics.perm_groups`

Examples

```
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... AlternatingGroup)
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.testutil import _verify_centralizer
>>> S = SymmetricGroup(5)
>>> A = AlternatingGroup(5)
>>> centr = PermutationGroup([Permutation([0, 1, 2, 3, 4])])
>>> _verify_centralizer(S, A, centr)
True
```

See also:

[`\_naive\_list\_centralizer`](#) (page 363), [`sympy.combinatorics.perm\_groups.PermutationGroup.centralizer`](#) (page 299), [`\_cmp\_perm\_lists`](#) (page 363)

`sympy.combinatorics.testutil._verify_normal_closure(group, arg, closure=None)`

Tensor Canonicalization

`sympy.combinatorics.tensor_can.canonicalize(g, dummies, msym, *v)`

canonicalize tensor formed by tensors

Parameters

g : permutation representing the tensor

dummies : list representing the dummy indices

it can be a list of dummy indices of the same type or a list of lists of dummy indices, one list for each type of index; the dummy indices must come after the free indices, and put in order contravariant, covariant [d0, -d0, d1, -d1, ...]

msym : symmetry of the metric(s)

it can be an integer or a list; in the first case it is the symmetry of the dummy index metric; in the second case it is the list of the symmetries of the index metric for each type

v : list, (base_i, gens_i, n_i, sym_i) for tensors of type *i*

base_i, gens_i : BSGS for tensors of this type.

The BSGS should have minimal base under lexicographic ordering; if not, an attempt is made to get the minimal BSGS; in case of failure, `canonicalize_naive` is used, which is much slower.

n_i : number of tensors of type *i*.

sym_i : symmetry under exchange of component tensors of type *i*.

Both for *msym* and *sym_i* the cases are

- None no symmetry
- 0 commuting
- 1 anticommuting

Returns

0 if the tensor is zero, else return the array form of

the permutation representing the canonical form of the tensor.

Algorithm

First one uses `canonical_free` to get the minimum tensor under lexicographic order, using only the slot symmetries. If the component tensors have not minimal BSGS, it is attempted to find it; if the attempt fails `canonicalize_naive` is used instead.

Compute the residual slot symmetry keeping fixed the free indices using `tensor_gens(base, gens, list_free_indices, sym)`.

Reduce the problem eliminating the free indices.

Then use `double_coset_can_rep` and lift back the result reintroducing the free indices.

Examples

one type of index with commuting metric;

A_{ab} and B_{ab} antisymmetric and commuting

$$T = A_{d0d1} * B^{d0}_{d2} * B^{d2d1}$$

$ord = [d0, -d0, d1, -d1, d2, -d2]$ order of the indices

$g = [1, 3, 0, 5, 4, 2, 6, 7]$

$$T_c = 0$$

```
>>> from sympy.combinatorics.tensor_can import get_symmetric_group_sgs, \
    canonicalize, bsgs_direct_product
>>> from sympy.combinatorics import Permutation
>>> base2a, gens2a = get_symmetric_group_sgs(2, 1)
>>> t0 = (base2a, gens2a, 1, 0)
>>> t1 = (base2a, gens2a, 2, 0)
>>> g = Permutation([1, 3, 0, 5, 4, 2, 6, 7])
>>> canonicalize(g, range(6), 0, t0, t1)
0
```

same as above, but with B_{ab} anticommuting

$$T_c = -A^{d0d1} * B_{d0}^{d2} * B_{d1d2}$$

$can = [0, 2, 1, 4, 3, 5, 7, 6]$

```
>>> t1 = (base2a, gens2a, 2, 1)
>>> canonicalize(g, range(6), 0, t0, t1)
[0, 2, 1, 4, 3, 5, 7, 6]
```

two types of indices $[a, b, c, d, e, f]$ and $[m, n]$, in this order, both with commuting metric

f^{abc} antisymmetric, commuting

A_{ma} no symmetry, commuting

$$T = f^c_{da} * f^f_{eb} * A^d_m * A^{mb} * A_n^a * A^{ne}$$

$ord = [c, f, a, -a, b, -b, d, -d, e, -e, m, -m, n, -n]$

$g = [0, 7, 3, 1, 9, 5, 11, 6, 10, 4, 13, 2, 12, 8, 14, 15]$

The canonical tensor is $T_c = -f^{cab} * f^{fde} * A^m_a * A_{md} * A^n_b * A_{ne}$

$can = [0, 2, 4, 1, 6, 8, 10, 3, 11, 7, 12, 5, 13, 9, 15, 14]$

```
>>> base_f, gens_f = get_symmetric_group_sgs(3, 1)
>>> base_l, gens_l = get_symmetric_group_sgs(1)
>>> base_A, gens_A = bsgs_direct_product(base_l, gens_l, base_l, gens_l)
>>> t0 = (base_f, gens_f, 2, 0)
>>> t1 = (base_A, gens_A, 4, 0)
>>> dummies = [range(2, 10), range(10, 14)]
>>> g = Permutation([0, 7, 3, 1, 9, 5, 11, 6, 10, 4, 13, 2, 12, 8, 14, \
    15])
>>> canonicalize(g, dummies, [0, 0], t0, t1)
[0, 2, 4, 1, 6, 8, 10, 3, 11, 7, 12, 5, 13, 9, 15, 14]
```

`sympy.combinatorics.tensor_can.double_coset_can_rep(dummies, sym, b_S, sgens, S_transversals, g)`

Butler-Portugal algorithm for tensor canonicalization with dummy indices.

Parameters

dummies

list of lists of dummy indices, one list for each type of index; the dummy indices are put in order contravariant, covariant [d0, -d0, d1, -d1, ...].

sym

list of the symmetries of the index metric for each type.

possible symmetries of the metrics

- 0 symmetric
- 1 antisymmetric
- None no symmetry

b_S

base of a minimal slot symmetry BSGS.

sgens

generators of the slot symmetry BSGS.

S_transversals

transversals for the slot BSGS.

g

permutation representing the tensor.

Returns

Return 0 if the tensor is zero, else return the array form of the permutation representing the canonical form of the tensor.

Notes

A tensor with dummy indices can be represented in a number of equivalent ways which typically grows exponentially with the number of indices. To be able to establish if two tensors with many indices are equal becomes computationally very slow in absence of an efficient algorithm.

The Butler-Portugal algorithm [3] is an efficient algorithm to put tensors in canonical form, solving the above problem.

Portugal observed that a tensor can be represented by a permutation, and that the class of tensors equivalent to it under slot and dummy symmetries is equivalent to the double coset $D * g * S$ (Note: in this documentation we use the conventions for multiplication of permutations p, q with $(p*q)(i) = p[q[i]]$ which is opposite to the one used in the Permutation class)

Using the algorithm by Butler to find a representative of the double coset one can find a canonical form for the tensor.

To see this correspondence, let g be a permutation in array form; a tensor with indices ind (the indices including both the contravariant and the covariant ones) can be written as

$$t = T(ind[g[0]], \dots, ind[g[n-1]]),$$

where $n = len(ind)$; g has size $n + 2$, the last two indices for the sign of the tensor (trick introduced in [4]).

A slot symmetry transformation s is a permutation acting on the slots $t \rightarrow T(ind[(g * s)[0]], \dots, ind[(g * s)[n-1]])$

A dummy symmetry transformation acts on ind $t \rightarrow T(ind[(d * g)[0]], \dots, ind[(d * g)[n-1]])$

Being interested only in the transformations of the tensor under these symmetries, one can represent the tensor by g , which transforms as

$g \rightarrow d * g * s$, so it belongs to the coset $D * g * S$, or in other words to the set of all permutations allowed by the slot and dummy symmetries.

Let us explain the conventions by an example.

Given a tensor T^{d3d2d1}_{d1d2d3} with the slot symmetries

$$T^{a0a1a2a3a4a5}_{d1d2d3} = -T^{a2a1a0a3a4a5}_{d1d2d3}$$

$$T^{a0a1a2a3a4a5}_{d1d2d3} = -T^{a4a1a2a3a0a5}_{d1d2d3}$$

and symmetric metric, find the tensor equivalent to it which is the lowest under the ordering of indices: lexicographic ordering $d1, d2, d3$ and then contravariant before covariant index; that is the canonical form of the tensor.

The canonical form is $-T^{d1d2d3}_{d1d2d3}$ obtained using $T^{a0a1a2a3a4a5}_{d1d2d3} = -T^{a2a1a0a3a4a5}_{d1d2d3}$.

To convert this problem in the input for this function, use the following ordering of the index names (- for covariant for short) $d1, -d1, d2, -d2, d3, -d3$

T^{d3d2d1}_{d1d2d3} corresponds to $g = [4, 2, 0, 1, 3, 5, 6, 7]$ where the last two indices are for the sign $sgens = [Permutation(0, 2)(6, 7), Permutation(0, 4)(6, 7)]$

$sgens[0]$ is the slot symmetry $-(0, 2)$ $T^{a0a1a2a3a4a5}_{d1d2d3} = -T^{a2a1a0a3a4a5}_{d1d2d3}$

$sgens[1]$ is the slot symmetry $-(0, 4)$ $T^{a0a1a2a3a4a5}_{d1d2d3} = -T^{a4a1a2a3a0a5}_{d1d2d3}$

The dummy symmetry group D is generated by the strong base generators $[(0, 1), (2, 3), (4, 5), (0, 2)(1, 3), (0, 4)(1, 5)]$ where the first three interchange covariant and contravariant positions of the same index ($d1 \leftrightarrow -d1$) and the last two interchange the dummy indices themselves ($d1 \leftrightarrow d2$).

The dummy symmetry acts from the left $d = [1, 0, 2, 3, 4, 5, 6, 7]$ exchange $d1 \leftrightarrow -d1$ $T^{d3d2d1}_{d1d2d3} == T^{d3d2}_{d1}^{d1}_{d2d3}$

$g = [4, 2, 0, 1, 3, 5, 6, 7] \rightarrow [4, 2, 1, 0, 3, 5, 6, 7] =_a f_rmul(d, g)$ which differs from $_a f_rmul(g, d)$.

The slot symmetry acts from the right $s = [2, 1, 0, 3, 4, 5, 7, 6]$ exchanges slots 0 and 2 and changes sign $T^{d3d2d1}_{d1d2d3} == -T^{d1d2d3}_{d1d2d3}$

$g = [4, 2, 0, 1, 3, 5, 6, 7] \rightarrow [0, 2, 4, 1, 3, 5, 7, 6] =_a f_rmul(g, s)$

Example in which the tensor is zero, same slot symmetries as above: $T^{d2}_{d1d3}^{d1d3}_{d2}$

$= -T^{d3}_{d1d3}^{d1d2}_{d2}$ under slot symmetry $-(0, 4)$;

$= T_{d3d1}^{d3d1d2}_{d2}$ under slot symmetry $-(0, 2)$;

$= T^{d3}_{d1d3}^{d1d2}_{d2}$ symmetric metric;

= 0 since two of these lines have tensors differ only for the sign.

The double coset $D*g*S$ consists of permutations $h = d*g*s$ corresponding to equivalent tensors; if there are two h which are the same apart from the sign, return zero; otherwise choose as representative the tensor with indices ordered lexicographically according to $[d1, -d1, d2, -d2, d3, -d3]$ that is $\text{rep} = \min(D*g*S) = \min([d*g*s \text{ for } d \text{ in } D \text{ for } s \text{ in } S])$

The indices are fixed one by one; first choose the lowest index for slot 0, then the lowest remaining index for slot 1, etc. Doing this one obtains a chain of stabilizers

$$S \rightarrow S_{b_0} \rightarrow S_{b_0, b_1} \rightarrow \dots \text{ and } D \rightarrow D_{p_0} \rightarrow D_{p_0, p_1} \rightarrow \dots$$

where $[b_0, b_1, \dots] = \text{range}(b)$ is a base of the symmetric group; the strong base b_S of S is an ordered sublist of it; therefore it is sufficient to compute once the strong base generators of S using the Schreier-Sims algorithm; the stabilizers of the strong base generators are the strong base generators of the stabilizer subgroup.

$\text{dbase} = [p_0, p_1, \dots]$ is not in general in lexicographic order, so that one must recompute the strong base generators each time; however this is trivial, there is no need to use the Schreier-Sims algorithm for D .

The algorithm keeps a TAB of elements (s_i, d_i, h_i) where $h_i = d_i \times g \times s_i$ satisfying $h_i[j] = p_j$ for $0 \leq j < i$ starting from $s_0 = id, d_0 = id, h_0 = g$.

The equations $h_0[0] = p_0, h_1[1] = p_1, \dots$ are solved in this order, choosing each time the lowest possible value of p_i

For $j < i$ $d_i * g * s_i * S_{b_0, \dots, b_{i-1}} * b_j = D_{p_0, \dots, p_{i-1}} * p_j$ so that for dx in $D_{p_0, \dots, p_{i-1}}$ and sx in $S_{\text{base}[0], \dots, \text{base}[i-1]}$ one has $dx * d_i * g * s_i * sx * b_j = p_j$

Search for dx, sx such that this equation holds for $j = i$; it can be written as $s_i * sx * b_j = J, dx * d_i * g * J = p_j, sx * b_j = s_i ** -1 * J; sx = \text{trace}(s_i ** -1, S_{b_0, \dots, b_{i-1}}) dx ** -1 * p_j = d_i * g * J; dx = \text{trace}(d_i * g * J, D_{p_0, \dots, p_{i-1}})$

$$s_{i+1} = s_i * \text{trace}(s_i ** -1 * J, S_{b_0, \dots, b_{i-1}}) \quad d_{i+1} = \text{trace}(d_i * g * J, D_{p_0, \dots, p_{i-1}}) ** -1 * d_i \quad h_{i+1} * b_i = d_{i+1} * g * s_{i+1} * b_i = p_i$$

$h_n * b_j = p_j$ for all j , so that h_n is the solution.

Add the found (s, d, h) to TAB1.

At the end of the iteration sort TAB1 with respect to the h ; if there are two consecutive h in TAB1 which differ only for the sign, the tensor is zero, so return 0; if there are two consecutive h which are equal, keep only one.

Then stabilize the slot generators under i and the dummy generators under p_i .

Assign $TAB = TAB1$ at the end of the iteration step.

At the end TAB contains a unique (s, d, h) , since all the slots of the tensor h have been fixed to have the minimum value according to the symmetries. The algorithm returns h .

It is important that the slot BSGS has lexicographic minimal base, otherwise there is an i which does not belong to the slot base for which p_i is fixed by the dummy symmetry only, while i is not invariant from the slot stabilizer, so p_i is not in general the minimal value.

This algorithm differs slightly from the original algorithm [3]:

the canonical form is minimal lexicographically, and the BSGS has minimal base under lexicographic order. Equal tensors h are eliminated from TAB.

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.tensor_can import double_coset_can_rep, get_
→ transversals
>>> gens = [Permutation(x) for x in [[2, 1, 0, 3, 4, 5, 7, 6], [4, 1, 2, 3,
→ 0, 5, 7, 6]]]
>>> base = [0, 2]
>>> g = Permutation([4, 2, 0, 1, 3, 5, 6, 7])
>>> transversals = get_transversals(base, gens)
>>> double_coset_can_rep([list(range(6))], [0], base, gens, transversals,
→ g)
[0, 1, 2, 3, 4, 5, 7, 6]
```

```
>>> g = Permutation([4, 1, 3, 0, 5, 2, 6, 7])
>>> double_coset_can_rep([list(range(6))], [0], base, gens, transversals,
→ g)
0
```

`sympy.combinatorics.tensor_can.get_symmetric_group_sgs(n, antisym=False)`

Return base, gens of the minimal BSGS for (anti)symmetric tensor

Parameters

```n```: rank of the tensor

```antisym```: bool

antisym = False symmetric tensor antisym = True antisymmetric tensor

Examples

```
>>> from sympy.combinatorics.tensor_can import get_symmetric_group_sgs
>>> get_symmetric_group_sgs(3)
([0, 1], [(4)(0 1), (4)(1 2)])
```

`sympy.combinatorics.tensor_can.bsgs_direct_product(base1, gens1, base2, gens2, signed=True)`

Direct product of two BSGS.

Parameters

base1: base of the first BSGS.

gens1: strong generating sequence of the first BSGS.

base2, gens2: similarly for the second BSGS.

signed: flag for signed permutations.

Examples

```
>>> from sympy.combinatorics.tensor_can import (get_symmetric_group_sgs,
↳ bsgs_direct_product)
>>> base1, gens1 = get_symmetric_group_sgs(1)
>>> base2, gens2 = get_symmetric_group_sgs(2)
>>> bsgs_direct_product(base1, gens1, base2, gens2)
([1], [(4)(1 2)])
```

Finitely Presented Groups

Introduction

This module presents the functionality designed for computing with finitely- presented groups (fp-groups for short). The name of the corresponding SymPy object is FpGroup. The functions or classes described here are studied under **computational group theory**. All code examples assume:

```
>>> from sympy.combinatorics.free_groups import free_group, vfree_group,
↳ xfree_group
>>> from sympy.combinatorics.fp_groups import FpGroup, CosetTable, coset_
↳ enumeration_r
```

Overview of Facilities

The facilities provided for fp-groups fall into a number of natural groupings

- The construction of fp-groups using a free group and a list of words in generators of that free group.
- Index determination using the famous Todd-Coxeter procedure.
- The construction of all subgroups having index less than some (small) specified positive integer, using the *Low-Index Subgroups* algorithm.
- Algorithms for computing presentations of a subgroup of finite index in a group defined by finite presentation.

For a description of fundamental algorithms of finitely presented groups we often make use of *Handbook of Computational Group Theory*.

The Construction of Finitely Presented Groups

Finitely presented groups are constructed by factoring a free group by a set of relators. The set of relators is taken in as a list of words in generators of free group in SymPy, using a list provides ordering to the relators. If the list of relators is empty, the associated free group is returned.

Example of construction of a finitely-presented group. The symmetric group of degree 4 may be represented as a two generator group with presentation $\langle a, b \mid a^2, b^3, (ab)^4 \rangle$. Giving the relations as a list of relators, group in SymPy would be specified as:

```
>>> F, a, b = free_group("a, b")
>>> G = FpGroup(F, [a**2, b**3, (a*b)**4])
>>> G
<fp group on the generators (a, b)>
```

Currently groups with relators having presentation like $\langle r, s, t \mid r^2, s^2, t^2, rst = str = trs \rangle$ will have to be specified as:

```
>>> F, r, s, t = free_group("r, s, t")
>>> G = FpGroup(F, [r**2, s**2, t**2, r*s*t*r**-1*t**-1*s**-1, s*t*r*s**-
↪ 1*r**-1*t**-1])
```

Obviously this is not a unique way to make that particular group, but the point is that in case of equality with non-identity the user has to manually do that.

Free Groups and Words

Construction of a Free Group

`free_group("gen0, gen1, ..., gen_(n-1)")` constructs a free group F on n generators, where n is a positive integer. The i -th generator of F may be obtained using the method `.generators[i]`, $i = 0, \dots, n-1$.

```
>>> F, x, y = free_group("x, y")
```

creates a free group F of rank 2 and assigns the variables x and y to the two generators.

```
>>> F = vfree_group("x, y")
>>> F
<free group on the generators (x, y)>
```

creates a free group F of rank 2, with tuple of generators `F.generators`, and inserts x and y as generators into the global namespace.

```
>>> F = xfree_group("x, y")
>>> F
(<free group on the generators (x, y)>, (x, y))
>>> x**2
x**2
```

creates a free groups $F[0]$ of rank 2, with tuple of generators $F[1]$.

Construction of words

This section is applicable to words of `FreeGroup` as well as `FpGroup`. When we say *word* in SymPy, it actually means a **reduced word**, since the words are automatically reduced. Given a group G defined on n generators $x_1, x_2, x_3, \dots, x_n$, a word is constructed as $s_1^{r_1} s_2^{r_2} \cdots s_k^{r_k}$ where $s_i \in \{x_1, x_2, \dots, x_n\}$, $r_i \in \mathbb{Z}$ for all k .

Each word can be constructed in a variety of ways, since after reduction they may be equivalent.

Coset Enumeration: The Todd-Coxeter Algorithm

This section describes the use of coset enumeration techniques in SymPy. The algorithm used for coset enumeration procedure is Todd-Coxeter algorithm and is developed in SymPy using [Ho05] and [CDHW73]. The reader should consult [CDHW73] and [Hav91] for a general description of the algorithm.

We have two strategies of coset enumeration *relator-based* and *coset-table based* and the two have been implemented as `coset_enumeration_r`, `coset_enumeration_c` respectively. The two strategies differ in the way they make new definitions for the cosets.

Though from the user point of view it is suggested to rather use the `.coset_enumeration` method of `FpGroup` and specify the `strategy` argument.

strategy:

(default="relator_based") specifies the strategy of coset enumeration to be used, possible values are "relator_based" or "coset_table_based".

CosetTable

Class used to manipulate the information regarding the coset enumeration of the finitely presented group G on the cosets of the subgroup H .

Basically a *coset table* `CosetTable(G,H)`, is the permutation representation of the finitely presented group on the cosets of a subgroup. Most of the set theoretic and group functions use the regular representation of G , i.e., the coset table of G over the trivial subgroup.

The actual mathematical coset table is obtained using `.table` attribute and is a list of lists. For each generator g of G it contains a column and the next column corresponds to g^{-1} and so on for other generators, so in total it has $2 * G.\text{rank}()$ columns. Each column is simply a list of integers. If l is the generator list for the generator g and if $l[i] = j$ then generator g takes the coset i to the coset j by multiplication from the right.

For finitely presented groups, a coset table is computed by a Todd-Coxeter coset enumeration. Note that you may influence the performance of that enumeration by changing the values of the variable `CosetTable.coset_table_max_limit`.

Attributes of CosetTable

For `CosetTable(G, H)` where G is the group and H is the subgroup.

- `n`: A non-negative integer, non-mutable attribute, dependently calculated as the maximum among the live-cosets (i.e. Ω).
- `table`: A list of lists, mutable attribute, mathematically represents the coset table.
- `omega`: A list, dependent on the internal attribute `p`. Ω represents the list of live-cosets. A *standard* coset-table has its $\Omega = \{0, 1, \dots, \text{index} - 1\}$ where *index* is the index of subgroup H in G .

For experienced users we have a number of parameters that can be used to manipulate the algorithm, like

- `coset_table_max_limit` (default value = 4096000): manipulate the maximum number of cosets allowed in coset enumeration, i.e. the number of rows allowed in coset table. A coset enumeration will not finish if the subgroup does not have finite index, and even if it has it may take many more intermediate cosets than the actual index of the subgroup

is. To avoid a coset enumeration “running away” therefore SymPy has a “safety stop” built-in. This is controlled by this variable. To change it, use `max_cosets` keyword. For example:

```
>>> F, a, b = free_group("a, b")
>>> Cox = FpGroup(F, [a**6, b**6, (a*b)**2, (a**2*b**2)**2,
    ↪ (a**3*b**3)**5])
>>> C_r = coset_enumeration_r(Cox, [a], max_cosets=50)
Traceback (most recent call last):
...
ValueError: the coset enumeration has defined more than 50 cosets
```

- `max_stack_size` (default value = 500): manipulate the maximum size of `deduction_stack` above or equal to which the stack is emptied.

Compression and Standardization

For any two entries i, j with $i < j$ in coset table, the first occurrence of i in a coset table precedes the first occurrence of j with respect to the usual row-wise ordering of the table entries. We call such a table a standard coset table. To standardize a `CosetTable` we use the `.standardize` method.

Note the method alters the given table, it does not create a copy.

Subgroups of Finite Index

The functionality in this section are concerned with the construction of subgroups of finite index. We describe a method for computing all subgroups whose index does not exceed some (modest) integer bound.

Low Index Subgroups

`low_index_subgroups(G, N)`: Given a finitely presented group $G = \langle X \mid R \rangle$ (can be a free group), and N a positive integer, determine the conjugacy classes of subgroups of G whose indices is less than or equal to N .

For example to find all subgroups of $G = \langle a, b \mid a^2 = b^3 = (ab)^4 = 1 \rangle$ having index ≤ 4 , can be found as follows:

```
>>> from sympy.combinatorics.fp_groups import low_index_subgroups
>>> F, a, b = free_group("a, b")
>>> G = FpGroup(F, [a**2, b**3, (a*b)**4])
>>> l = low_index_subgroups(G, 4)
>>> for coset_table in l:
...     print(coset_table.table)
...
[[0, 0, 0, 0]]
[[0, 0, 1, 2], [1, 1, 2, 0], [3, 3, 0, 1], [2, 2, 3, 3]]
[[0, 0, 1, 2], [2, 2, 2, 0], [1, 1, 0, 1]]
[[1, 1, 0, 0], [0, 0, 1, 1]]
```

This returns the coset tables of subgroups of satisfying the property that index, *index*, of subgroup in group is $\leq n$.

Constructing a presentation for a subgroup

In this section we discuss finding the presentation of a subgroup in a finitely presentation group. While the *subgroup* is currently allowed as input only in the form of a list of generators for the subgroup, you can expect the functionality of a *coset table* as input for subgroup in the group in near future.

There are two ways to construct a set of defining relations for subgroup from those of G . First is on a set of Schreier generators, known generally as Reidemeister-Schreier algorithm or on the given list of generators of H .

Reidemeister Schreier algorithm

called using `reidemeister_presentation(G, Y)` where G is the group and Y is a list of generators for subgroup H whose presentation we want to find.

```
>>> from sympy.combinatorics.fp_groups import reidemeister_presentation
>>> F, x, y = free_group("x, y")
>>> f = FpGroup(F, [x**3, y**5, (x*y)**2])
>>> H = [x*y, x**-1*y**-1*x*y*x]
>>> p1 = reidemeister_presentation(f, H)
>>> p1
((y_1, y_2), (y_1**2, y_2**3, y_2*y_1*y_2*y_1*y_2*y_1))
```

Bibliography

Polycyclic Groups

Introduction

This module presents the functionality designed for computing with polycyclic groups (PcGroup for short). The name of the corresponding SymPy object is `PolycyclicGroup`. The functions or classes described here are studied under **Computational Group Theory**.

Overview of functionalities

- The construction of `PolycyclicGroup` from a given `PermutationGroup`.
- Computation of polycyclic generating sequence (pcgs for short) and polycyclic series (pc_series).
- Computation of relative order for polycyclic series.
- Implementation of class `Collector` which can be treated as a base for polycyclic groups.
- Implementation of polycyclic group presentation (pc_presentation for short).

- Computation of exponent vector, depth and leading exponent for a given element of a polycyclic group.

For a description of fundamental algorithms of polycyclic groups, we often make use of *Handbook of Computational Group Theory*.

The Construction of Polycyclic Groups

Given a Permutation Group, A Polycyclic Group is constructed by computing the corresponding polycyclic generating sequence, polycyclic series and it's relative order.

Attributes of PolycyclicGroup

- `pc_sequence` : Polycyclic sequence is formed by collecting all the missing generators between the adjacent groups in the derived series of given permutation group.
- `pc_series` : Polycyclic series is formed by adding all the missing generators of `der[i+1]` in `der[i]`, where `der` represents derived series.
- `relative_order` : A list, computed by the ratio of adjacent groups in `pc_series`.
- `collector` : By default, it is None. Collector class provides the polycyclic presentation.

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> G = SymmetricGroup(4)
>>> PcGroup = G.polycyclic_group()
>>> len(PcGroup.pcgs)
4
>>> pc_series = PcGroup.pc_series
>>> pc_series[0].equals(G) # use equals, not literal '=='
True
>>> gen = pc_series[len(pc_series) - 1].generators[0]
>>> gen.is_identity
True
>>> PcGroup.relative_order
[2, 3, 2, 2]
```

The Construction of Collector

Collector is one of the attributes of class PolycyclicGroup.

Attributes of Collector

Collector posses all the attributes of PolycyclicGroup, In addition there are few more attributes which are defined below:

- `free_group` : `free_group` provides the mapping of polycyclic generating sequence with the free group elements.
- `pc_presentation` : Provides the presentation of polycyclic groups with the help of power and conjugate relators.


```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> G = SymmetricGroup(3)
>>> PcGroup = G.polycyclic_group()
>>> Collector = PcGroup.collector
>>> Collector.free_group
<free group on the generators (x0, x1)>
>>> Collector.pc_presentation
{x0**2: (), x1**3: (), x0**-1*x1*x0: x1**2}
```

Computation of Minimal Uncollected Subword

A word V defined on generators in the free_group of pc_group is a minimal uncollected subword of the word W if V is a subword of W and it has one of the following form:

- $v = x_{i+1}^{a_j} x_i$
- $v = x_{i+1}^{a_j} x_i^{-1}$
- $v = x_i^{a_j}$

$a_j \notin \{0, \dots, \text{relative\_order}[j] - 1\}$.

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.free_groups import free_group
>>> G = SymmetricGroup(4)
>>> PcGroup = G.polycyclic_group()
>>> collector = PcGroup.collector
>>> F, x1, x2 = free_group("x1, x2")
>>> word = x2**2*x1**7
>>> collector.minimal_uncollected_subword(word)
((x2, 2),)
```

Computation of Subword Index

For a given word and it's subword, subword_index computes the starting and ending index of the subword in the word.

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.free_groups import free_group
>>> G = SymmetricGroup(4)
>>> PcGroup = G.polycyclic_group()
>>> collector = PcGroup.collector
>>> F, x1, x2 = free_group("x1, x2")
>>> word = x2**2*x1**7
>>> w = x2**2*x1
>>> collector.subword_index(word, w)
(0, 3)
>>> w = x1**7
>>> collector.subword_index(word, w)
(2, 9)
```

Computation of Collected Word

A word W is called collected, if $W = x_{i_1}^{a_1} \dots x_{i_r}^{a_r}$ with $i_1 < i_2 < \dots < i_r$ and a_j is in $\{1 \dots s_{j-1}\}$, where s_j represents the respective relative order.

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.free_groups import free_group
>>> G = SymmetricGroup(4)
>>> PcGroup = G.polycyclic_group()
>>> collector = PcGroup.collector
>>> F, x0, x1, x2, x3 = free_group("x0, x1, x2, x3")
>>> word = x3*x2*x1*x0
>>> collected_word = collector.collected_word(word)
>>> free_to_perm = {}
>>> free_group = collector.free_group
>>> for sym, gen in zip(free_group.symbols, collector.pcgs):
...     free_to_perm[sym] = gen
>>> G1 = PermutationGroup()
>>> for w in word:
...     sym = w[0]
...     perm = free_to_perm[sym]
...     G1 = PermutationGroup([perm] + G1.generators)
>>> G2 = PermutationGroup()
>>> for w in collected_word:
...     sym = w[0]
...     perm = free_to_perm[sym]
...     G2 = PermutationGroup([perm] + G2.generators)
```

The two are not identical but they are equivalent:

```
>>> G1 == G2
False
>>> G1.equals(G2)
True
```

Computation of Polycyclic Presentation

The computation of presentation starts from the bottom of the pcgs and polycyclic series. Storing all the previous generators from pcgs and then taking the last generator as the generator which acts as a conjugator and conjugates all the previous generators in the list.

To get a clear picture, start with an example of SymmetricGroup(4). For $S(4)$ there are 4 generators in pcgs say $[x_0, x_1, x_2, x_3]$ and the relative_order vector is $[2, 3, 2, 2]$. Starting from bottom of this sequence the presentation is computed in order as below.

using only $[x_3]$ from pcgs and pc_series[4] compute:

- x_3^2

using only $[x_3]$ from pcgs and pc_series[3] compute:

- x_2^2
- $x_2^{-1}x_3x_2$

using $[x_3, x_2]$ from pcgs and pc_series[2] compute:

- x_1^3
- $x_1^{-1}x_3x_1$
- $x_1^{-1}x_2x_1$

using $[x_3, x_2, x_1]$ from pcgs and pc_series[1] compute:

- x_0^2
- $x_0^{-1}x_3x_0$
- $x_0^{-1}x_2x_0$
- $x_0^{-1}x_1x_0$

One thing to note is same group can have different pcgs due to varying derived_series which, results in different polycyclic presentations.

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.permutations import Permutation
>>> G = SymmetricGroup(4)
>>> PcGroup = G.polycyclic_group()
>>> collector = PcGroup.collector
>>> pcgs = PcGroup.pcgs
>>> len(pcgs)
4
>>> free_group = collector.free_group
>>> pc_representation = collector.pc_presentation
>>> free_to_perm = {}
>>> for s, g in zip(free_group.symbols, pcgs):
...     free_to_perm[s] = g
>>> for k, v in pc_representation.items():
...     k_array = k.array_form
...     if v != ():
...         v_array = v.array_form
...         lhs = Permutation()
...         for gen in k_array:
...             s = gen[0]
...             e = gen[1]
...             lhs = lhs*free_to_perm[s]**e
...         if v == ():
...             assert lhs.is_identity
...             continue
...         rhs = Permutation()
...         for gen in v_array:
...             s = gen[0]
...             e = gen[1]
...             rhs = rhs*free_to_perm[s]**e
...         assert lhs == rhs
```

Computation of Exponent Vector

Any generator of the polycyclic group can be represented with the help of it's polycyclic generating sequence. Hence, the length of exponent vector is equal to the length of the pcgs.

A given generator g of the polycyclic group, can be represented as $g = x_1^{e_1} \dots x_n^{e_n}$, where x_i represents polycyclic generators and n is the number of generators in the free_group equal to the length of pcgs.

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.permutations import Permutation
>>> G = SymmetricGroup(4)
>>> PcGroup = G.polycyclic_group()
>>> collector = PcGroup.collector
>>> pcgs = PcGroup.pcgs
>>> collector.exponent_vector(G[0])
[1, 0, 0, 0]
>>> exp = collector.exponent_vector(G[1])
>>> g = Permutation()
>>> for i in range(len(exp)):
...     g = g*pcgs[i]**exp[i] if exp[i] else g
>>> assert g == G[1]
```

Depth of Polycyclic generator

Depth of a given polycyclic generator is defined as the index of the first non-zero entry in the exponent vector.

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> G = SymmetricGroup(3)
>>> PcGroup = G.polycyclic_group()
>>> collector = PcGroup.collector
>>> collector.depth(G[0])
2
>>> collector.depth(G[1])
1
```

Computation of Leading Exponent

Leading exponent represents the exponent of polycyclic generator at the above depth.

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> G = SymmetricGroup(3)
>>> PcGroup = G.polycyclic_group()
>>> collector = PcGroup.collector
>>> collector.leading_exponent(G[1])
1
```

Bibliography

Functions

All functions support the methods documented below, inherited from `sympy.core.function.Function` (page 1050).

class `sympy.core.function.Function(*args)`

Base class for applied mathematical functions.

It also serves as a constructor for undefined function classes.

See the [Writing Custom Functions](#) (page 102) guide for details on how to subclass `Function` and what methods can be defined.

Examples

Undefined Functions

To create an undefined function, pass a string of the function name to `Function`.

```
>>> from sympy import Function, Symbol
>>> x = Symbol('x')
>>> f = Function('f')
>>> g = Function('g')(x)
>>> f
f
>>> f(x)
f(x)
>>> g
g(x)
>>> f(x).diff(x)
Derivative(f(x), x)
>>> g.diff(x)
Derivative(g(x), x)
```

Assumptions can be passed to `Function` the same as with a [Symbol](#) (page 976). Alternatively, you can use a `Symbol` with assumptions for the function name and the function will inherit the name and assumptions associated with the `Symbol`:

```
>>> f_real = Function('f', real=True)
>>> f_real(x).is_real
True
>>> f_real_inherit = Function(Symbol('f', real=True))
>>> f_real_inherit(x).is_real
True
```

Note that assumptions on a function are unrelated to the assumptions on the variables it is called on. If you want to add a relationship, subclass `Function` and define custom assumptions handler methods. See the [Assumptions](#) (page 111) section of the [Writing Custom Functions](#) (page 102) guide for more details.

Custom Function Subclasses

The [Writing Custom Functions](#) (page 102) guide has several [Complete Examples](#) (page 122) of how to subclass `Function` to create a custom function.

as_base_exp()

Returns the method as the 2-tuple (base, exponent).

fdiff(*argindex*=1)

Returns the first derivative of the function.

classmethod is_singular(*a*)

Tests whether the argument is an essential singularity or a branch point, or the functions is non-holomorphic.

Contents

Elementary

This module implements elementary functions such as trigonometric, hyperbolic, and sqrt, as well as functions like Abs, Max, Min etc.

sympy.functions.elementary.complexes

re

class sympy.functions.elementary.complexes.**re**(*arg*)

Returns real part of expression. This function performs only elementary analysis and so it will fail to decompose properly more complicated expressions. If completely simplified result is needed then use `Basic.as_real_imag()` or perform complex expansion on instance of this function.

Parameters

arg : Expr

Real or complex expression.

Returns

expr : Expr

Real part of expression.

Examples

```
>>> from sympy import re, im, I, E, symbols
>>> x, y = symbols('x y', real=True)
>>> re(2*E)
2*E
>>> re(2*I + 17)
17
>>> re(2*I)
0
>>> re(im(x) + x*I + 2)
2
>>> re(5 + I + 2)
7
```

See also:[im](#) (page 383)**as_real_imag**(*deep=True, **hints*)

Returns the real number with a zero imaginary part.

im**class** `sympy.functions.elementary.complexes.im(arg)`

Returns imaginary part of expression. This function performs only elementary analysis and so it will fail to decompose properly more complicated expressions. If completely simplified result is needed then use `Basic.as_real_imag()` or perform complex expansion on instance of this function.

Parameters**arg** : Expr

Real or complex expression.

Returns**expr** : Expr

Imaginary part of expression.

Examples

```
>>> from sympy import re, im, E, I
>>> from sympy.abc import x, y
>>> im(2*E)
0
>>> im(2*I + 17)
2
>>> im(x*I)
re(x)
>>> im(re(x) + y)
im(y)
>>> im(2 + 3*I)
3
```

See also:[re](#) (page 382)**as_real_imag**(*deep=True, **hints*)

Return the imaginary part with a zero real part.

sign

class sympy.functions.elementary.complexes.**sign**(arg)

Returns the complex sign of an expression:

Parameters

arg : Expr

Real or imaginary expression.

Returns

expr : Expr

Complex sign of expression.

Explanation

If the expression is real the sign will be:

- 1 if expression is positive
- 0 if expression is equal to zero
- -1 if expression is negative

If the expression is imaginary the sign will be:

- I if $\text{im}(\text{expression})$ is positive
- $-I$ if $\text{im}(\text{expression})$ is negative

Otherwise an unevaluated expression will be returned. When evaluated, the result (in general) will be $\cos(\arg(\text{expr})) + I\sin(\arg(\text{expr}))$.

Examples

```
>>> from sympy import sign, I
```

```
>>> sign(-1)
-1
>>> sign(0)
0
>>> sign(-3*I)
-I
>>> sign(1 + I)
sign(1 + I)
>>> _.evalf()
0.707106781186548 + 0.707106781186548*I
```

See also:

[Abs](#) (page 385), [conjugate](#) (page 386)

Abs

class sympy.functions.elementary.complexes.**Abs**(arg)

Return the absolute value of the argument.

Parameters

arg : Expr

Real or complex expression.

Returns

expr : Expr

Absolute value returned can be an expression or integer depending on input arg.

Explanation

This is an extension of the built-in function `abs()` to accept symbolic values. If you pass a SymPy expression to the built-in `abs()`, it will pass it automatically to `Abs()`.

Examples

```

>>> from sympy import Abs, Symbol, S, I
>>> Abs(-1)
1
>>> x = Symbol('x', real=True)
>>> Abs(-x)
Abs(x)
>>> Abs(x**2)
x**2
>>> abs(-x) # The Python built-in
Abs(x)
>>> Abs(3*x + 2*I)
sqrt(9*x**2 + 4)
>>> Abs(8*I)
8

```

Note that the Python built-in will return either an Expr or int depending on the argument:

```

>>> type(abs(-1))
<... 'int'>
>>> type(abs(S.NegativeOne))
<class 'sympy.core.numbers.One'>

```

Abs will always return a SymPy object.

See also:

[sign](#) (page 384), [conjugate](#) (page 386)

fdiff(argindex=1)

Get the first derivative of the argument to `Abs()`.

arg

class sympy.functions.elementary.complexes.**arg**(arg)

Returns the argument (in radians) of a complex number. The argument is evaluated in consistent convention with `atan2` where the branch-cut is taken along the negative real axis and $\arg(z)$ is in the interval $(-\pi, \pi]$. For a positive number, the argument is always 0; the argument of a negative number is π ; and the argument of 0 is undefined and returns `nan`. So the `arg` function will never nest greater than 3 levels since at the 4th application, the result must be `nan`; for a real number, `nan` is returned on the 3rd application.

Parameters

arg : Expr

Real or complex expression.

Returns

value : Expr

Returns arc tangent of arg measured in radians.

Examples

```
>>> from sympy import arg, I, sqrt, Dummy
>>> from sympy.abc import x
>>> arg(2.0)
0
>>> arg(I)
pi/2
>>> arg(sqrt(2) + I*sqrt(2))
pi/4
>>> arg(sqrt(3)/2 + I/2)
pi/6
>>> arg(4 + 3*I)
atan(3/4)
>>> arg(0.8 + 0.6*I)
0.643501108793284
>>> arg(arg(arg(arg(x))))
nan
>>> real = Dummy(real=True)
>>> arg(arg(arg(real)))
nan
```

conjugate

class sympy.functions.elementary.complexes.**conjugate**(arg)

Returns the *complex conjugate* [R238] of an argument. In mathematics, the complex conjugate of a complex number is given by changing the sign of the imaginary part.

Thus, the conjugate of the complex number $a + ib$ (where a and b are real numbers) is $a - ib$

Parameters

arg : Expr

Real or complex expression.

Returns

arg : Expr

Complex conjugate of arg as real, imaginary or mixed expression.

Examples

```
>>> from sympy import conjugate, I
>>> conjugate(2)
2
>>> conjugate(I)
-I
>>> conjugate(3 + 2*I)
3 - 2*I
>>> conjugate(5 - I)
5 + I
```

See also:

[sign](#) (page 384), [Abs](#) (page 385)

References

[R238]

polar_lift

class sympy.functions.elementary.complexes.**polar_lift**(arg)

Lift argument to the Riemann surface of the logarithm, using the standard branch.

Parameters

arg : Expr

Real or complex expression.

Examples

```
>>> from sympy import Symbol, polar_lift, I
>>> p = Symbol('p', polar=True)
>>> x = Symbol('x')
>>> polar_lift(4)
4*exp_polar(0)
>>> polar_lift(-4)
4*exp_polar(I*pi)
>>> polar_lift(-I)
exp_polar(-I*pi/2)
>>> polar_lift(I + 2)
polar_lift(2 + I)
```

```
>>> polar_lift(4*x)
4*polar_lift(x)
>>> polar_lift(4*p)
4*p
```

See also:

[*sympy.functions.elementary.exponential.exp_polar*](#) (page 414),
[*periodic_argument*](#) (page 388)

periodic_argument

class `sympy.functions.elementary.complexes.periodic_argument(ar, period)`

Represent the argument on a quotient of the Riemann surface of the logarithm. That is, given a period P , always return a value in $(-P/2, P/2]$, by using $\exp(Pi) = 1$.

Parameters

ar : Expr

A polar number.

period : Expr

The period P .

Examples

```
>>> from sympy import exp_polar, periodic_argument
>>> from sympy import I, pi
>>> periodic_argument(exp_polar(10*I*pi), 2*pi)
0
>>> periodic_argument(exp_polar(5*I*pi), 4*pi)
pi
>>> from sympy import exp_polar, periodic_argument
>>> from sympy import I, pi
>>> periodic_argument(exp_polar(5*I*pi), 2*pi)
pi
>>> periodic_argument(exp_polar(5*I*pi), 3*pi)
-pi
>>> periodic_argument(exp_polar(5*I*pi), pi)
0
```

See also:

[*sympy.functions.elementary.exponential.exp_polar*](#) (page 414)

[*polar_lift*](#) (page 387)

Lift argument to the Riemann surface of the logarithm

[*principal_branch*](#) (page 389)

principal_branch

class sympy.functions.elementary.complexes.principal_branch(*x*, *period*)

Represent a polar number reduced to its principal branch on a quotient of the Riemann surface of the logarithm.

Parameters

x : Expr

A polar number.

period : Expr

Positive real number or infinity.

Explanation

This is a function of two arguments. The first argument is a polar number z , and the second one a positive real number or infinity, p . The result is $z \bmod \exp\_polar(I*p)$.

Examples

```

>>> from sympy import exp_polar, principal_branch, oo, I, pi
>>> from sympy.abc import z
>>> principal_branch(z, oo)
z
>>> principal_branch(exp_polar(2*pi*I)*3, 2*pi)
3*exp_polar(0)
>>> principal_branch(exp_polar(2*pi*I)*3*z, 2*pi)
3*principal_branch(z, 2*pi)

```

See also:

[*sympy.functions.elementary.exponential.exp_polar*](#) (page 414)

[*polar_lift*](#) (page 387)

Lift argument to the Riemann surface of the logarithm

[*periodic_argument*](#) (page 388)

sympy.functions.elementary.trigonometric

Trigonometric Functions

sin

class sympy.functions.elementary.trigonometric.sin(*arg*)

The sine function.

Returns the sine of x (measured in radians).

Explanation

This function will evaluate automatically in the case x/π is some rational number [R242]. For example, if x is a multiple of π , $\pi/2$, $\pi/3$, $\pi/4$, and $\pi/6$.

Examples

```
>>> from sympy import sin, pi
>>> from sympy.abc import x
>>> sin(x**2).diff(x)
2*x*cos(x**2)
>>> sin(1).diff(x)
0
>>> sin(pi)
0
>>> sin(pi/2)
1
>>> sin(pi/6)
1/2
>>> sin(pi/12)
-sqrt(2)/4 + sqrt(6)/4
```

See also:

[csc](#) (page 393), [cos](#) (page 390), [sec](#) (page 393), [tan](#) (page 391), [cot](#) (page 392), [asin](#) (page 395), [acsc](#) (page 399), [acos](#) (page 396), [asec](#) (page 398), [atan](#) (page 397), [acot](#) (page 397), [atan2](#) (page 400)

References

[R239], [R240], [R241], [R242]

cos

class sympy.functions.elementary.trigonometric.cos(*arg*)

The cosine function.

Returns the cosine of x (measured in radians).

Explanation

See [sin\(\)](#) (page 389) for notes about automatic evaluation.

Examples

```
>>> from sympy import cos, pi
>>> from sympy.abc import x
>>> cos(x**2).diff(x)
-2*x*sin(x**2)
>>> cos(1).diff(x)
0
>>> cos(pi)
-1
>>> cos(pi/2)
0
>>> cos(2*pi/3)
-1/2
>>> cos(pi/12)
sqrt(2)/4 + sqrt(6)/4
```

See also:

[sin](#) (page 389), [csc](#) (page 393), [sec](#) (page 393), [tan](#) (page 391), [cot](#) (page 392), [asin](#) (page 395), [acsc](#) (page 399), [acos](#) (page 396), [asec](#) (page 398), [atan](#) (page 397), [acot](#) (page 397), [atan2](#) (page 400)

References

[R243], [R244], [R245]

tan

class sympy.functions.elementary.trigonometric.**tan**(arg)

The tangent function.

Returns the tangent of x (measured in radians).

Explanation

See [sin](#) (page 389) for notes about automatic evaluation.

Examples

```
>>> from sympy import tan, pi
>>> from sympy.abc import x
>>> tan(x**2).diff(x)
2*x*(tan(x**2)**2 + 1)
>>> tan(1).diff(x)
0
>>> tan(pi/8).expand()
-1 + sqrt(2)
```

See also:

[sin](#) (page 389), [csc](#) (page 393), [cos](#) (page 390), [sec](#) (page 393), [cot](#) (page 392), [asin](#) (page 395), [acsc](#) (page 399), [acos](#) (page 396), [asec](#) (page 398), [atan](#) (page 397), [acot](#) (page 397), [atan2](#) (page 400)

References

[R246], [R247], [R248]

inverse(*argindex*=1)

Returns the inverse of this function.

cot

class sympy.functions.elementary.trigonometric.**cot**(*arg*)

The cotangent function.

Returns the cotangent of *x* (measured in radians).

Explanation

See [sin](#) (page 389) for notes about automatic evaluation.

Examples

```
>>> from sympy import cot, pi
>>> from sympy.abc import x
>>> cot(x**2).diff(x)
2*x*(-cot(x**2)**2 - 1)
>>> cot(1).diff(x)
0
>>> cot(pi/12)
sqrt(3) + 2
```

See also:

[sin](#) (page 389), [csc](#) (page 393), [cos](#) (page 390), [sec](#) (page 393), [tan](#) (page 391), [asin](#) (page 395), [acsc](#) (page 399), [acos](#) (page 396), [asec](#) (page 398), [atan](#) (page 397), [acot](#) (page 397), [atan2](#) (page 400)

References

[R249], [R250], [R251]

inverse(*argindex*=1)

Returns the inverse of this function.

sec

class sympy.functions.elementary.trigonometric.**sec**(*arg*)

The secant function.

Returns the secant of *x* (measured in radians).

Explanation

See [sin](#) (page 389) for notes about automatic evaluation.

Examples

```
>>> from sympy import sec
>>> from sympy.abc import x
>>> sec(x**2).diff(x)
2*x*tan(x**2)*sec(x**2)
>>> sec(1).diff(x)
0
```

See also:

[sin](#) (page 389), [csc](#) (page 393), [cos](#) (page 390), [tan](#) (page 391), [cot](#) (page 392), [asin](#) (page 395), [acsc](#) (page 399), [acos](#) (page 396), [asec](#) (page 398), [atan](#) (page 397), [acot](#) (page 397), [atan2](#) (page 400)

References

[R252], [R253], [R254]

csc

class sympy.functions.elementary.trigonometric.**csc**(*arg*)

The cosecant function.

Returns the cosecant of *x* (measured in radians).

Explanation

See [`sin\(\)`](#) (page 389) for notes about automatic evaluation.

Examples

```
>>> from sympy import csc
>>> from sympy.abc import x
>>> csc(x**2).diff(x)
-2*x*cot(x**2)*csc(x**2)
>>> csc(1).diff(x)
0
```

See also:

[`sin`](#) (page 389), [`cos`](#) (page 390), [`sec`](#) (page 393), [`tan`](#) (page 391), [`cot`](#) (page 392), [`asin`](#) (page 395), [`acsc`](#) (page 399), [`acos`](#) (page 396), [`asec`](#) (page 398), [`atan`](#) (page 397), [`acot`](#) (page 397), [`atan2`](#) (page 400)

References

[R255], [R256], [R257]

sinc

class `sympy.functions.elementary.trigonometric.sinc(arg)`

Represents an unnormalized sinc function:

$$\text{sinc}(x) = \begin{cases} \frac{\sin x}{x} & x \neq 0 \\ 1 & x = 0 \end{cases}$$

Examples

```
>>> from sympy import sinc, oo, jn
>>> from sympy.abc import x
>>> sinc(x)
sinc(x)
```

- Automated Evaluation

```
>>> sinc(0)
1
>>> sinc(oo)
0
```

- Differentiation

```
>>> sinc(x).diff()
cos(x)/x - sin(x)/x**2
```

- Series Expansion

```
>>> sinc(x).series()
1 - x**2/6 + x**4/120 + O(x**6)
```

- As zero'th order spherical Bessel Function

```
>>> sinc(x).rewrite(jn)
jn(0, x)
```

See also:

[*sin*](#) (page 389)

References

[R258]

Trigonometric Inverses

asin

class sympy.functions.elementary.trigonometric.asin(*arg*)

The inverse sine function.

Returns the arcsine of *x* in radians.

Explanation

asin(*x*) will evaluate automatically in the cases $x \in \{\infty, -\infty, 0, 1, -1\}$ and for some instances when the result is a rational multiple of π (see the `eval` class method).

A purely imaginary argument will lead to an asinh expression.

Examples

```
>>> from sympy import asin, oo
>>> asin(1)
pi/2
>>> asin(-1)
-pi/2
>>> asin(-oo)
oo*I
>>> asin(oo)
-oo*I
```

See also:

[sin](#) (page 389), [csc](#) (page 393), [cos](#) (page 390), [sec](#) (page 393), [tan](#) (page 391), [cot](#) (page 392), [acsc](#) (page 399), [acos](#) (page 396), [asec](#) (page 398), [atan](#) (page 397), [acot](#) (page 397), [atan2](#) (page 400)

References

[R259], [R260], [R261]

inverse(*argindex*=1)

Returns the inverse of this function.

acos

class sympy.functions.elementary.trigonometric.**acos**(*arg*)

The inverse cosine function.

Returns the arc cosine of x (measured in radians).

Examples

```
>>> from sympy import acos, oo
>>> acos(1)
0
>>> acos(0)
pi/2
>>> acos(oo)
oo*I
```

See also:

[sin](#) (page 389), [csc](#) (page 393), [cos](#) (page 390), [sec](#) (page 393), [tan](#) (page 391), [cot](#) (page 392), [asin](#) (page 395), [acsc](#) (page 399), [asec](#) (page 398), [atan](#) (page 397), [acot](#) (page 397), [atan2](#) (page 400)

References

[R262], [R263], [R264]

inverse(*argindex*=1)

Returns the inverse of this function.

atan

class sympy.functions.elementary.trigonometric.**atan**(*arg*)

The inverse tangent function.

Returns the arc tangent of x (measured in radians).

Explanation

`atan(x)` will evaluate automatically in the cases $x \in \{\infty, -\infty, 0, 1, -1\}$ and for some instances when the result is a rational multiple of π (see the `eval` class method).

Examples

```
>>> from sympy import atan, oo
>>> atan(0)
0
>>> atan(1)
pi/4
>>> atan(oo)
pi/2
```

See also:

[sin](#) (page 389), [csc](#) (page 393), [cos](#) (page 390), [sec](#) (page 393), [tan](#) (page 391), [cot](#) (page 392), [asin](#) (page 395), [acsc](#) (page 399), [acos](#) (page 396), [asec](#) (page 398), [acot](#) (page 397), [atan2](#) (page 400)

References

[R265], [R266], [R267]

inverse(*argindex*=1)

Returns the inverse of this function.

acot

class sympy.functions.elementary.trigonometric.**acot**(*arg*)

The inverse cotangent function.

Returns the arc cotangent of x (measured in radians).

Explanation

$\operatorname{acot}(x)$ will evaluate automatically in the cases $x \in \{\infty, -\infty, \tilde{\infty}, 0, 1, -1\}$ and for some instances when the result is a rational multiple of π (see the `eval` class method).

A purely imaginary argument will lead to an `acoth` expression.

$\operatorname{acot}(x)$ has a branch cut along $(-i, i)$, hence it is discontinuous at 0. Its range for real x is $(-\frac{\pi}{2}, \frac{\pi}{2}]$.

Examples

```
>>> from sympy import acot, sqrt
>>> acot(0)
pi/2
>>> acot(1)
pi/4
>>> acot(sqrt(3) - 2)
-5*pi/12
```

See also:

[sin](#) (page 389), [csc](#) (page 393), [cos](#) (page 390), [sec](#) (page 393), [tan](#) (page 391), [cot](#) (page 392), [asin](#) (page 395), [acsc](#) (page 399), [acos](#) (page 396), [asec](#) (page 398), [atan](#) (page 397), [atan2](#) (page 400)

References

[R268], [R269]

`inverse(argindex=1)`

Returns the inverse of this function.

asec

class `sympy.functions.elementary.trigonometric.asec(arg)`

The inverse secant function.

Returns the arc secant of x (measured in radians).

Explanation

$\operatorname{asec}(x)$ will evaluate automatically in the cases $x \in \{\infty, -\infty, 0, 1, -1\}$ and for some instances when the result is a rational multiple of π (see the `eval` class method).

$\operatorname{asec}(x)$ has branch cut in the interval $[-1, 1]$. For complex arguments, it can be defined [R273] as

$$\sec^{-1}(z) = -i \frac{\log(\sqrt{1-z^2} + 1)}{z}$$

At $x = 0$, for positive branch cut, the limit evaluates to ∞ . For negative branch cut, the limit

$$\lim_{z \rightarrow 0} -i \frac{\log(-\sqrt{1-z^2} + 1)}{z}$$

simplifies to $-i \log(z/2 + O(z^3))$ which ultimately evaluates to ∞ .

As $\arccos(x) = \operatorname{asec}(1/x)$, a similar argument can be given for $\arccos(x)$.

Examples

```
>>> from sympy import asec, oo
>>> asec(1)
0
>>> asec(-1)
pi
>>> asec(0)
zoo
>>> asec(-oo)
pi/2
```

See also:

[sin](#) (page 389), [csc](#) (page 393), [cos](#) (page 390), [sec](#) (page 393), [tan](#) (page 391), [cot](#) (page 392), [asin](#) (page 395), [acsc](#) (page 399), [acos](#) (page 396), [atan](#) (page 397), [acot](#) (page 397), [atan2](#) (page 400)

References

[R270], [R271], [R272], [R273]

inverse(*argindex*=1)

Returns the inverse of this function.

acsc

class sympy.functions.elementary.trigonometric.**acsc**(*arg*)

The inverse cosecant function.

Returns the arc cosecant of x (measured in radians).

Explanation

$\operatorname{acsc}(x)$ will evaluate automatically in the cases $x \in \{\infty, -\infty, 0, 1, -1\}$ and for some instances when the result is a rational multiple of π (see the `eval` class method).

Examples

```
>>> from sympy import acsc, oo
>>> acsc(1)
pi/2
>>> acsc(-1)
-pi/2
>>> acsc(oo)
0
>>> acsc(-oo) == acsc(oo)
True
>>> acsc(0)
zoo
```

See also:

[sin](#) (page 389), [csc](#) (page 393), [cos](#) (page 390), [sec](#) (page 393), [tan](#) (page 391), [cot](#) (page 392), [asin](#) (page 395), [acos](#) (page 396), [asec](#) (page 398), [atan](#) (page 397), [acot](#) (page 397), [atan2](#) (page 400)

References

[R274], [R275], [R276]

inverse(*argindex*=1)

Returns the inverse of this function.

atan2

class sympy.functions.elementary.trigonometric.**atan2**(*y*, *x*)

The function `atan2(y, x)` computes $\operatorname{atan}(y/x)$ taking two arguments *y* and *x*. Signs of both *y* and *x* are considered to determine the appropriate quadrant of $\operatorname{atan}(y/x)$. The range is $(-\pi, \pi]$. The complete definition reads as follows:

$$\operatorname{atan2}(y, x) = \begin{cases} \arctan\left(\frac{y}{x}\right) & x > 0 \\ \arctan\left(\frac{y}{x}\right) + \pi & y \geq 0, x < 0 \\ \arctan\left(\frac{y}{x}\right) - \pi & y < 0, x < 0 \\ +\frac{\pi}{2} & y > 0, x = 0 \\ -\frac{\pi}{2} & y < 0, x = 0 \\ \text{undefined} & y = 0, x = 0 \end{cases}$$

Attention: Note the role reversal of both arguments. The *y*-coordinate is the first argument and the *x*-coordinate the second.

If either *x* or *y* is complex:

$$\operatorname{atan2}(y, x) = -i \log\left(\frac{x + iy}{\sqrt{x^2 + y^2}}\right)$$

Examples

Going counter-clock wise around the origin we find the following angles:

```
>>> from sympy import atan2
>>> atan2(0, 1)
0
>>> atan2(1, 1)
pi/4
>>> atan2(1, 0)
pi/2
>>> atan2(1, -1)
3*pi/4
>>> atan2(0, -1)
pi
>>> atan2(-1, -1)
-3*pi/4
>>> atan2(-1, 0)
-pi/2
>>> atan2(-1, 1)
-pi/4
```

which are all correct. Compare this to the results of the ordinary `atan` function for the point $(x, y) = (-1, 1)$

```
>>> from sympy import atan, S
>>> atan(S(1)/-1)
-pi/4
>>> atan2(1, -1)
3*pi/4
```

where only the `atan2` function returns what we expect. We can differentiate the function with respect to both arguments:

```
>>> from sympy import diff
>>> from sympy.abc import x, y
>>> diff(atan2(y, x), x)
-y/(x**2 + y**2)
```

```
>>> diff(atan2(y, x), y)
x/(x**2 + y**2)
```

We can express the `atan2` function in terms of complex logarithms:

```
>>> from sympy import log
>>> atan2(y, x).rewrite(log)
-I*log((x + I*y)/sqrt(x**2 + y**2))
```

and in terms of (*atan*):

```
>>> from sympy import atan
>>> atan2(y, x).rewrite(atan)
Piecewise((2*atan(y/(x + sqrt(x**2 + y**2))), Ne(y, 0)), (pi, re(x) < 0),
→ (0, Ne(x, 0)), (nan, True))
```

but note that this form is undefined on the negative real axis.

See also:

[sin](#) (page 389), [csc](#) (page 393), [cos](#) (page 390), [sec](#) (page 393), [tan](#) (page 391), [cot](#) (page 392), [asin](#) (page 395), [acsc](#) (page 399), [acos](#) (page 396), [asec](#) (page 398), [atan](#) (page 397), [acot](#) (page 397)

References

[R277], [R278], [R279]

`sympy.functions.elementary.hyperbolic`

Hyperbolic Functions

HyperbolicFunction

class `sympy.functions.elementary.hyperbolic.HyperbolicFunction(*args)`

Base class for hyperbolic functions.

See also:

[sinh](#) (page 402), [cosh](#) (page 403), [tanh](#) (page 403), [coth](#) (page 404)

`sinh`

class `sympy.functions.elementary.hyperbolic.sinh(arg)`

$\sinh(x)$ is the hyperbolic sine of x .

The hyperbolic sine function is $\frac{e^x - e^{-x}}{2}$.

Examples

```
>>> from sympy import sinh
>>> from sympy.abc import x
>>> sinh(x)
sinh(x)
```

See also:

[cosh](#) (page 403), [tanh](#) (page 403), [asinh](#) (page 405)

as_real_imag(*deep=True*, ***hints*)

Returns this function as a complex coordinate.

fdiff(*argindex=1*)

Returns the first derivative of this function.

inverse(*argindex=1*)

Returns the inverse of this function.

static `taylor_term(n, x, *previous_terms)`

Returns the next term in the Taylor series expansion.

cosh

class `sympy.functions.elementary.hyperbolic.cosh(arg)`

`cosh(x)` is the hyperbolic cosine of `x`.

The hyperbolic cosine function is $\frac{e^x + e^{-x}}{2}$.

Examples

```
>>> from sympy import cosh
>>> from sympy.abc import x
>>> cosh(x)
cosh(x)
```

See also:

[`sinh`](#) (page 402), [`tanh`](#) (page 403), [`acosh`](#) (page 406)

tanh

class `sympy.functions.elementary.hyperbolic.tanh(arg)`

`tanh(x)` is the hyperbolic tangent of `x`.

The hyperbolic tangent function is $\frac{\sinh(x)}{\cosh(x)}$.

Examples

```
>>> from sympy import tanh
>>> from sympy.abc import x
>>> tanh(x)
tanh(x)
```

See also:

[`sinh`](#) (page 402), [`cosh`](#) (page 403), [`atanh`](#) (page 406)

inverse(`argindex=1`)

Returns the inverse of this function.

coth

class sympy.functions.elementary.hyperbolic.**coth**(arg)

coth(x) is the hyperbolic cotangent of x.

The hyperbolic cotangent function is $\frac{\cosh(x)}{\sinh(x)}$.

Examples

```
>>> from sympy import coth
>>> from sympy.abc import x
>>> coth(x)
coth(x)
```

See also:

[sinh](#) (page 402), [cosh](#) (page 403), [acoth](#) (page 407)

inverse(argindex=1)

Returns the inverse of this function.

sech

class sympy.functions.elementary.hyperbolic.**sech**(arg)

sech(x) is the hyperbolic secant of x.

The hyperbolic secant function is $\frac{2}{e^x + e^{-x}}$.

Examples

```
>>> from sympy import sech
>>> from sympy.abc import x
>>> sech(x)
sech(x)
```

See also:

[sinh](#) (page 402), [cosh](#) (page 403), [tanh](#) (page 403), [coth](#) (page 404), [csch](#) (page 404), [asinh](#) (page 405), [acosh](#) (page 406)

csch

class sympy.functions.elementary.hyperbolic.**csch**(arg)

csch(x) is the hyperbolic cosecant of x.

The hyperbolic cosecant function is $\frac{2}{e^x - e^{-x}}$.

Examples

```
>>> from sympy import csch
>>> from sympy.abc import x
>>> csch(x)
csch(x)
```

See also:

[sinh](#) (page 402), [cosh](#) (page 403), [tanh](#) (page 403), [sech](#) (page 404), [asinh](#) (page 405), [acosh](#) (page 406)

fdiff(*argindex=1*)

Returns the first derivative of this function

static taylor_term(*n, x, *previous_terms*)

Returns the next term in the Taylor series expansion

Hyperbolic Inverses

asinh

class sympy.functions.elementary.hyperbolic.**asinh**(*arg*)

$\operatorname{asinh}(x)$ is the inverse hyperbolic sine of x .

The inverse hyperbolic sine function.

Examples

```
>>> from sympy import asinh
>>> from sympy.abc import x
>>> asinh(x).diff(x)
1/sqrt(x**2 + 1)
>>> asinh(1)
log(1 + sqrt(2))
```

See also:

[acosh](#) (page 406), [atanh](#) (page 406), [sinh](#) (page 402)

inverse(*argindex=1*)

Returns the inverse of this function.

acosh

class `sympy.functions.elementary.hyperbolic.acosh(arg)`

`acosh(x)` is the inverse hyperbolic cosine of `x`.

The inverse hyperbolic cosine function.

Examples

```
>>> from sympy import acosh
>>> from sympy.abc import x
>>> acosh(x).diff(x)
1/(sqrt(x - 1)*sqrt(x + 1))
>>> acosh(1)
0
```

See also:

[`asinh`](#) (page 405), [`atanh`](#) (page 406), [`cosh`](#) (page 403)

inverse(*argindex*=1)

Returns the inverse of this function.

atanh

class `sympy.functions.elementary.hyperbolic.atanh(arg)`

`atanh(x)` is the inverse hyperbolic tangent of `x`.

The inverse hyperbolic tangent function.

Examples

```
>>> from sympy import atanh
>>> from sympy.abc import x
>>> atanh(x).diff(x)
1/(1 - x**2)
```

See also:

[`asinh`](#) (page 405), [`acosh`](#) (page 406), [`tanh`](#) (page 403)

inverse(*argindex*=1)

Returns the inverse of this function.

acoth

class sympy.functions.elementary.hyperbolic.**acoth**(arg)

acoth(x) is the inverse hyperbolic cotangent of x.

The inverse hyperbolic cotangent function.

Examples

```
>>> from sympy import acoth
>>> from sympy.abc import x
>>> acoth(x).diff(x)
1/(1 - x**2)
```

See also:

[asinh](#) (page 405), [acosh](#) (page 406), [coth](#) (page 404)

inverse(argindex=1)

Returns the inverse of this function.

asech

class sympy.functions.elementary.hyperbolic.**asech**(arg)

asech(x) is the inverse hyperbolic secant of x.

The inverse hyperbolic secant function.

Examples

```
>>> from sympy import asech, sqrt, S
>>> from sympy.abc import x
>>> asech(x).diff(x)
-1/(x*sqrt(1 - x**2))
>>> asech(1).diff(x)
0
>>> asech(1)
0
>>> asech(S(2))
I*pi/3
>>> asech(-sqrt(2))
3*I*pi/4
>>> asech((sqrt(6) - sqrt(2)))
I*pi/12
```

See also:

[asinh](#) (page 405), [atanh](#) (page 406), [cosh](#) (page 403), [acoth](#) (page 407)

References

[R280], [R281], [R282]

inverse(*argindex*=1)

Returns the inverse of this function.

acsch

class sympy.functions.elementary.hyperbolic.**acsch**(*arg*)

acsch(*x*) is the inverse hyperbolic cosecant of *x*.

The inverse hyperbolic cosecant function.

Examples

```
>>> from sympy import acsch, sqrt, S
>>> from sympy.abc import x
>>> acsch(x).diff(x)
-1/(x**2*sqrt(1 + x**(-2)))
>>> acsch(1).diff(x)
0
>>> acsch(1)
log(1 + sqrt(2))
>>> acsch(S.ImaginaryUnit)
-I*pi/2
>>> acsch(-2*S.ImaginaryUnit)
I*pi/6
>>> acsch(S.ImaginaryUnit*(sqrt(6) - sqrt(2)))
-5*I*pi/12
```

See also:

[*asinh*](#) (page 405)

References

[R283], [R284], [R285]

inverse(*argindex*=1)

Returns the inverse of this function.

sympy.functions.elementary.integers

ceiling

class sympy.functions.elementary.integers.**ceiling**(*arg*)

Ceiling is a univariate function which returns the smallest integer value not less than its argument. This implementation generalizes ceiling to complex numbers by taking the ceiling of the real and imaginary parts separately.

Examples

```

>>> from sympy import ceiling, E, I, S, Float, Rational
>>> ceiling(17)
17
>>> ceiling(Rational(23, 10))
3
>>> ceiling(2*E)
6
>>> ceiling(-Float(0.567))
0
>>> ceiling(I/2)
I
>>> ceiling(S(5)/2 + 5*I/2)
3 + 3*I

```

See also:

[*sympy.functions.elementary.integers.floor*](#) (page 409)

References

[R286], [R287]

floor

class sympy.functions.elementary.integers.**floor**(*arg*)

Floor is a univariate function which returns the largest integer value not greater than its argument. This implementation generalizes floor to complex numbers by taking the floor of the real and imaginary parts separately.

Examples

```
>>> from sympy import floor, E, I, S, Float, Rational
>>> floor(17)
17
>>> floor(Rational(23, 10))
2
>>> floor(2*E)
5
>>> floor(-Float(0.567))
-1
>>> floor(-I/2)
-I
>>> floor(S(5)/2 + 5*I/2)
2 + 2*I
```

See also:

[*sympy.functions.elementary.integers.ceiling*](#) (page 409)

References

[R288], [R289]

RoundFunction

class `sympy.functions.elementary.integers.RoundFunction(arg)`
Abstract base class for rounding functions.

frac

class `sympy.functions.elementary.integers.frac(arg)`
Represents the fractional part of x
For real numbers it is defined [R290] as

$$x - \lfloor x \rfloor$$

Examples

```
>>> from sympy import Symbol, frac, Rational, floor, I
>>> frac(Rational(4, 3))
1/3
>>> frac(-Rational(4, 3))
2/3
```

returns zero for integer arguments

```
>>> n = Symbol('n', integer=True)
>>> frac(n)
0
```

rewrite as floor

```
>>> x = Symbol('x')
>>> frac(x).rewrite(floor)
x - floor(x)
```

for complex arguments

```
>>> r = Symbol('r', real=True)
>>> t = Symbol('t', real=True)
>>> frac(t + I*r)
I*frac(r) + frac(t)
```

See also:

[*sympy.functions.elementary.integers.floor*](#) (page 409), [*sympy.functions.elementary.integers.ceiling*](#) (page 409)

References

[R290], [R291]

`sympy.functions.elementary.exponential`

`exp`

class `sympy.functions.elementary.exponential.exp(arg)`

The exponential function, e^x .

Parameters

arg : Expr

Examples

```
>>> from sympy import exp, I, pi
>>> from sympy.abc import x
>>> exp(x)
exp(x)
>>> exp(x).diff(x)
exp(x)
>>> exp(I*pi)
-1
```

See also:

[*log*](#) (page 413)

as_real_imag(*deep=True, **hints*)

Returns this function as a 2-tuple representing a complex number.

Examples

```
>>> from sympy import exp, I
>>> from sympy.abc import x
>>> exp(x).as_real_imag()
(exp(re(x))*cos(im(x)), exp(re(x))*sin(im(x)))
>>> exp(1).as_real_imag()
(E, 0)
>>> exp(I).as_real_imag()
(cos(1), sin(1))
>>> exp(1+I).as_real_imag()
(E*cos(1), E*sin(1))
```

See also:

[sympy.functions.elementary.complexes.re](#) (page 382), [sympy.functions.elementary.complexes.im](#) (page 383)

property base

Returns the base of the exponential function.

fdiff(*argindex=1*)

Returns the first derivative of this function.

static taylor_term(*n, x, *previous_terms*)

Calculates the next term in the Taylor series expansion.

LambertW

class `sympy.functions.elementary.exponential.LambertW`(*x, k=None*)

The Lambert W function $W(z)$ is defined as the inverse function of $w \exp(w)$ [R292].

Explanation

In other words, the value of $W(z)$ is such that $z = W(z) \exp(W(z))$ for any complex number z . The Lambert W function is a multivalued function with infinitely many branches $W_k(z)$, indexed by $k \in \mathbb{Z}$. Each branch gives a different solution w of the equation $z = w \exp(w)$.

The Lambert W function has two partially real branches: the principal branch ($k = 0$) is real for real $z > -1/e$, and the $k = -1$ branch is real for $-1/e < z < 0$. All branches except $k = 0$ have a logarithmic singularity at $z = 0$.

Examples

```
>>> from sympy import LambertW
>>> LambertW(1.2)
0.635564016364870
>>> LambertW(1.2, -1).n()
-1.34747534407696 - 4.41624341514535*I
>>> LambertW(-1).is_real
False
```

References

[R292]

fdiff(*argindex=1*)

Return the first derivative of this function.

log

class sympy.functions.elementary.exponential.**log**(*arg, base=None*)

The natural logarithm function $\ln(x)$ or $\log(x)$.

Explanation

Logarithms are taken with the natural base, e . To get a logarithm of a different base b , use $\log(x, b)$, which is essentially short-hand for $\log(x)/\log(b)$.

$\log$ represents the principal branch of the natural logarithm. As such it has a branch cut along the negative real axis and returns values having a complex argument in $(-\pi, \pi]$.

Examples

```
>>> from sympy import log, sqrt, S, I
>>> log(8, 2)
3
>>> log(S(8)/3, 2)
-log(3)/log(2) + 3
>>> log(-1 + I*sqrt(3))
log(2) + 2*I*pi/3
```

See also:

[exp](#) (page 411)

as_base_exp()

Returns this function in the form (base, exponent).

as_real_imag(*deep=True, **hints*)

Returns this function as a complex coordinate.

Examples

```
>>> from sympy import I, log
>>> from sympy.abc import x
>>> log(x).as_real_imag()
(log(Abs(x)), arg(x))
>>> log(I).as_real_imag()
(0, pi/2)
>>> log(1 + I).as_real_imag()
(log(sqrt(2)), pi/4)
>>> log(I*x).as_real_imag()
(log(Abs(x)), arg(I*x))
```

fdiff(*argindex*=1)

Returns the first derivative of the function.

inverse(*argindex*=1)

Returns e^x , the inverse function of $\log(x)$.

static taylor_term(*n*, *x*, *previous_terms)

Returns the next term in the Taylor series expansion of $\log(1 + x)$.

exp_polar

class sympy.functions.elementary.exponential.exp_polar(*args)

Represent a *polar number* (see g-function Sphinx documentation).

Explanation

`exp_polar` represents the function $\text{Exp} : \mathbb{C} \rightarrow S$, sending the complex number $z = a + bi$ to the polar number $r = \exp(a), \theta = b$. It is one of the main functions to construct polar numbers.

Examples

```
>>> from sympy import exp_polar, pi, I, exp
```

The main difference is that polar numbers do not “wrap around” at 2π :

```
>>> exp(2*pi*I)
1
>>> exp_polar(2*pi*I)
exp_polar(2*I*pi)
```

apart from that they behave mostly like classical complex numbers:

```
>>> exp_polar(2)*exp_polar(3)
exp_polar(5)
```

See also:

[`sympy.simplify.powsimp.powsimp`](#) (page 680), [`polar\_lift`](#) (page 387), [`periodic\_argument`](#) (page 388), [`principal\_branch`](#) (page 389)

`sympy.functions.elementary.piecewise`

`ExprCondPair`

class `sympy.functions.elementary.piecewise.ExprCondPair(expr, cond)`

Represents an expression, condition pair.

property `cond`

Returns the condition of this pair.

property `expr`

Returns the expression of this pair.

`Piecewise`

class `sympy.functions.elementary.piecewise.Piecewise(*args)`

Represents a piecewise function.

Usage:

`Piecewise( (expr,cond), (expr,cond), ... )`

- Each argument is a 2-tuple defining an expression and condition
- The conds are evaluated in turn returning the first that is True. If any of the evaluated conds are not explicitly False, e.g. $x < 1$, the function is returned in symbolic form.
- If the function is evaluated at a place where all conditions are False, nan will be returned.
- Pairs where the cond is explicitly False, will be removed and no pair appearing after a True condition will ever be retained. If a single pair with a True condition remains, it will be returned, even when evaluation is False.

Examples

```
>>> from sympy import Piecewise, log, piecewise_fold
>>> from sympy.abc import x, y
>>> f = x**2
>>> g = log(x)
>>> p = Piecewise((0, x < -1), (f, x <= 1), (g, True))
>>> p.subs(x,1)
1
>>> p.subs(x,5)
log(5)
```

Booleans can contain Piecewise elements:

```
>>> cond = (x < y).subs(x, Piecewise((2, x < 0), (3, True))); cond
Piecewise((2, x < 0), (3, True)) < y
```

The folded version of this results in a Piecewise whose expressions are Booleans:

```
>>> folded_cond = piecewise_fold(cond); folded_cond
Piecewise((2 < y, x < 0), (3 < y, True))
```

When a Boolean containing Piecewise (like `cond`) or a Piecewise with Boolean expressions (like `folded_cond`) is used as a condition, it is converted to an equivalent [ITE](#) (page 1171) object:

```
>>> Piecewise((1, folded_cond))
Piecewise((1, ITE(x < 0, y > 2, y > 3)))
```

When a condition is an ITE, it will be converted to a simplified Boolean expression:

```
>>> piecewise_fold(_)
Piecewise((1, ((x >= 0) | (y > 2)) & ((y > 3) | (x < 0))))
```

See also:

[piecewise_fold](#) (page 419), [piecewise_exclusive](#) (page 418), [ITE](#) (page 1171)

`_eval_integral(x, first=True, **kwargs)`

Return the indefinite integral of the Piecewise such that subsequent substitution of `x` with a value will give the value of the integral (not including the constant of integration) up to that point. To only integrate the individual parts of Piecewise, use the `piecewise_integrate` method.

Examples

```
>>> from sympy import Piecewise
>>> from sympy.abc import x
>>> p = Piecewise((0, x < 0), (1, x < 1), (2, True))
>>> p.integrate(x)
Piecewise((0, x < 0), (x, x < 1), (2*x - 1, True))
>>> p.piecewise_integrate(x)
Piecewise((0, x < 0), (x, x < 1), (2*x, True))
```

See also:

[Piecewise.piecewise_integrate](#) (page 417)

`as_expr_set_pairs(domain=None)`

Return tuples for each argument of self that give the expression and the interval in which it is valid which is contained within the given domain. If a condition cannot be converted to a set, an error will be raised. The variable of the conditions is assumed to be real; sets of real values are returned.

Examples

```
>>> from sympy import Piecewise, Interval
>>> from sympy.abc import x
>>> p = Piecewise(
...     (1, x < 2),
...     (2, (x > 0) & (x < 4)),
...     (3, True))
>>> p.as_expr_set_pairs()
[(1, Interval.open(-oo, 2)),
 (2, Interval.Ropen(2, 4)),
 (3, Interval(4, oo))]
>>> p.as_expr_set_pairs(Interval(0, 3))
[(1, Interval.Ropen(0, 2)),
 (2, Interval(2, 3))]
```

doit(hints)**

Evaluate this piecewise function.

classmethod eval(*_args)

Either return a modified version of the args or, if no modifications were made, return None.

Modifications that are made here:

1. relationals are made canonical
2. any False conditions are dropped
3. any repeat of a previous condition is ignored
4. any args past one with a true condition are dropped

If there are no args left, nan will be returned. If there is a single arg with a True condition, its corresponding expression will be returned.

Examples

```
>>> from sympy import Piecewise
>>> from sympy.abc import x
>>> cond = -x < -1
>>> args = [(1, cond), (4, cond), (3, False), (2, True), (5, x < 1)]
>>> Piecewise(*args, evaluate=False)
Piecewise((1, -x < -1), (4, -x < -1), (2, True))
>>> Piecewise(*args)
Piecewise((1, x > 1), (2, True))
```

piecewise_integrate(x, **kwargs)

Return the Piecewise with each expression being replaced with its antiderivative. To obtain a continuous antiderivative, use the [integrate\(\)](#) (page 598) function or method.

Examples

```
>>> from sympy import Piecewise
>>> from sympy.abc import x
>>> p = Piecewise((0, x < 0), (1, x < 1), (2, True))
>>> p.pieceswise_integrate(x)
Piecewise((0, x < 0), (x, x < 1), (2*x, True))
```

Note that this does not give a continuous function, e.g. at $x = 1$ the 3rd condition applies and the antiderivative there is $2*x$ so the value of the antiderivative is 2:

```
>>> anti = _
>>> anti.subs(x, 1)
2
```

The continuous derivative accounts for the integral *up to* the point of interest, however:

```
>>> p.integrate(x)
Piecewise((0, x < 0), (x, x < 1), (2*x - 1, True))
>>> _.subs(x, 1)
1
```

See also:

[*Piecewise._eval_integral*](#) (page 416)

`sympy.functions.elementary.piecewise.pieceswise_exclusive(expr, *, skip_nan=False, deep=True)`

Rewrite [*Piecewise*](#) (page 415) with mutually exclusive conditions.

Parameters

expr: a SymPy expression.

Any [*Piecewise*](#) (page 415) in the expression will be rewritten.

skip_nan: ``bool`` (default ``False``)

If `skip_nan` is set to `True` then a final [*NaN*](#) (page 997) case will not be included.

deep: ``bool`` (default ``True``)

If `deep` is `True` then [*pieceswise_exclusive\(\)*](#) (page 418) will rewrite any [*Piecewise*](#) (page 415) subexpressions in `expr` rather than just rewriting `expr` itself.

Returns

An expression equivalent to `expr` but where all [*Piecewise*](#) (page 415) have been rewritten with mutually exclusive conditions.

Explanation

SymPy represents the conditions of a *Piecewise* (page 415) in an “if-elif”-fashion, allowing more than one condition to be simultaneously True. The interpretation is that the first condition that is True is the case that holds. While this is a useful representation computationally it is not how a piecewise formula is typically shown in a mathematical text. The *piecewise_exclusive()* (page 418) function can be used to rewrite any *Piecewise* (page 415) with more typical mutually exclusive conditions.

Note that further manipulation of the resulting *Piecewise* (page 415), e.g. simplifying it, will most likely make it non-exclusive. Hence, this is primarily a function to be used in conjunction with printing the Piecewise or if one would like to reorder the expression-condition pairs.

If it is not possible to determine that all possibilities are covered by the different cases of the *Piecewise* (page 415) then a final *NaN* (page 997) case will be included explicitly. This can be prevented by passing `skip_nan=True`.

Examples

```
>>> from sympy import piecewise_exclusive, Symbol, Piecewise, S
>>> x = Symbol('x', real=True)
>>> p = Piecewise((0, x < 0), (S.Half, x <= 0), (1, True))
>>> piecewise_exclusive(p)
Piecewise((0, x < 0), (1/2, Eq(x, 0)), (1, x > 0))
>>> piecewise_exclusive(Piecewise((2, x > 1)))
Piecewise((2, x > 1), (nan, x <= 1))
>>> piecewise_exclusive(Piecewise((2, x > 1)), skip_nan=True)
Piecewise((2, x > 1))
```

See also:

Piecewise (page 415), *piecewise_fold* (page 419)

`sympy.functions.elementary.piecewise.piecewise_fold(expr, evaluate=True)`

Takes an expression containing a piecewise function and returns the expression in piecewise form. In addition, any ITE conditions are rewritten in negation normal form and simplified.

The final Piecewise is evaluated (default) but if the raw form is desired, send `evaluate=False`; if trivial evaluation is desired, send `evaluate=None` and duplicate conditions and processing of True and False will be handled.

Examples

```
>>> from sympy import Piecewise, piecewise_fold, S
>>> from sympy.abc import x
>>> p = Piecewise((x, x < 1), (1, S(1) <= x))
>>> piecewise_fold(x*p)
Piecewise((x**2, x < 1), (x, True))
```

See also:

Piecewise (page 415), *piecewise_exclusive* (page 418)

sympy.functions.elementary.miscellaneous**IdentityFunction****class** sympy.functions.elementary.miscellaneous.**IdentityFunction**

The identity function

Examples

```
>>> from sympy import Id, Symbol
>>> x = Symbol('x')
>>> Id(x)
x
```

Min**class** sympy.functions.elementary.miscellaneous.**Min**(*args)

Return, if possible, the minimum value of the list. It is named Min and not min to avoid conflicts with the built-in function min.

Examples

```
>>> from sympy import Min, Symbol, oo
>>> from sympy.abc import x, y
>>> p = Symbol('p', positive=True)
>>> n = Symbol('n', negative=True)
```

```
>>> Min(x, -2)
Min(-2, x)
>>> Min(x, -2).subs(x, 3)
-2
>>> Min(p, -3)
-3
>>> Min(x, y)
Min(x, y)
>>> Min(n, 8, p, -7, p, oo)
Min(-7, n)
```

See also:**Max** (page 421)

find maximum values

Max

class `sympy.functions.elementary.miscellaneous.Max(*args)`

Return, if possible, the maximum value of the list.

When number of arguments is equal one, then return this argument.

When number of arguments is equal two, then return, if possible, the value from (a, b) that is $\geq$ the other.

In common case, when the length of list greater than 2, the task is more complicated. Return only the arguments, which are greater than others, if it is possible to determine directional relation.

If is not possible to determine such a relation, return a partially evaluated result.

Assumptions are used to make the decision too.

Also, only comparable arguments are permitted.

It is named Max and not max to avoid conflicts with the built-in function max.

Examples

```
>>> from sympy import Max, Symbol, oo
>>> from sympy.abc import x, y, z
>>> p = Symbol('p', positive=True)
>>> n = Symbol('n', negative=True)
```

```
>>> Max(x, -2)
Max(-2, x)
>>> Max(x, -2).subs(x, 3)
3
>>> Max(p, -2)
p
>>> Max(x, y)
Max(x, y)
>>> Max(x, y) == Max(y, x)
True
>>> Max(x, Max(y, z))
Max(x, y, z)
>>> Max(n, 8, p, 7, -oo)
Max(8, p)
>>> Max(1, x, oo)
oo
```

- Algorithm

The task can be considered as searching of supremums in the directed complete partial orders [R293].

The source values are sequentially allocated by the isolated subsets in which supremums are searched and result as Max arguments.

If the resulted supremum is single, then it is returned.

The isolated subsets are the sets of values which are only comparable with each other in the current set. E.g. natural numbers are comparable with each other, but not comparable with the x symbol. Another example: the symbol x with negative assumption is comparable with a natural number.

Also there are “least” elements, which are comparable with all others, and have a zero property (maximum or minimum for all elements). For example, in case of ∞ , the allocation operation is terminated and only this value is returned.

Assumption:

- if $A > B > C$ then $A > C$
- if $A = B$ then B can be removed

See also:***Min* (page 420)**

find minimum values

References

[R293], [R294]

root

`sympy.functions.elementary.miscellaneous.root(arg, n, k=0, evaluate=None)`

Returns the k -th n -th root of arg .

Parameters

k : int, optional

Should be an integer in $\{0, 1, \dots, n-1\}$. Defaults to the principal root if 0.

evaluate : bool, optional

The parameter determines if the expression should be evaluated. If None, its value is taken from `global_parameters.evaluate`.

Examples

```
>>> from sympy import root, Rational
>>> from sympy.abc import x, n
```

```
>>> root(x, 2)
sqrt(x)
```

```
>>> root(x, 3)
x**(1/3)
```

```
>>> root(x, n)
x**(1/n)
```

```
>>> root(x, -Rational(2, 3))
x**(-3/2)
```

To get the k-th n-th root, specify k:

```
>>> root(-2, 3, 2)
-(-1)**(2/3)*2**(1/3)
```

To get all n n-th roots you can use the `rootof` function. The following examples show the roots of unity for n equal 2, 3 and 4:

```
>>> from sympy import rootof
```

```
>>> [rootof(x**2 - 1, i) for i in range(2)]
[-1, 1]
```

```
>>> [rootof(x**3 - 1,i) for i in range(3)]
[1, -1/2 - sqrt(3)*I/2, -1/2 + sqrt(3)*I/2]
```

```
>>> [rootof(x**4 - 1,i) for i in range(4)]
[-1, 1, -I, I]
```

SymPy, like other symbolic algebra systems, returns the complex root of negative numbers. This is the principal root and differs from the text-book result that one might be expecting. For example, the cube root of -8 does not come back as -2:

```
>>> root(-8, 3)
2*(-1)**(1/3)
```

The `real_root` function can be used to either make the principal result real (or simply to return the real root directly):

```
>>> from sympy import real_root
>>> real_root(_)
-2
>>> real_root(-32, 5)
-2
```

Alternatively, the n//2-th n-th root of a negative number can be computed with `root`:

```
>>> root(-32, 5, 5//2)
-2
```

See also:

[`sympy.polys.rootoftools.rootof`](#) (page 2431), [`sympy.core.power.integer\_nthroot`](#) (page 1008), [`sqrt`](#) (page 424), [`real\_root`](#) (page 426)

References

[R295], [R296], [R297], [R298], [R299]

sqrt

`sympy.functions.elementary.miscellaneous.sqrt(arg, evaluate=None)`

Returns the principal square root.

Parameters

evaluate : bool, optional

The parameter determines if the expression should be evaluated. If None, its value is taken from `global_parameters.evaluate`.

Examples

```
>>> from sympy import sqrt, Symbol, S
>>> x = Symbol('x')
```

```
>>> sqrt(x)
sqrt(x)
```

```
>>> sqrt(x)**2
x
```

Note that `sqrt(x**2)` does not simplify to `x`.

```
>>> sqrt(x**2)
sqrt(x**2)
```

This is because the two are not equal to each other in general. For example, consider `x == -1`:

```
>>> from sympy import Eq
>>> Eq(sqrt(x**2), x).subs(x, -1)
False
```

This is because `sqrt` computes the principal square root, so the square may put the argument in a different branch. This identity does hold if `x` is positive:

```
>>> y = Symbol('y', positive=True)
>>> sqrt(y**2)
y
```

You can force this simplification by using the `powdenest()` function with the `force` option set to `True`:

```
>>> from sympy import powdenest
>>> sqrt(x**2)
sqrt(x**2)
```

(continues on next page)

(continued from previous page)

```
>>> powdenest(sqrt(x**2), force=True)
x
```

To get both branches of the square root you can use the `rootof` function:

```
>>> from sympy import rootof
```

```
>>> [rootof(x**2-3,i) for i in (0,1)]
[-sqrt(3), sqrt(3)]
```

Although `sqrt` is printed, there is no `sqrt` function so looking for `sqrt` in an expression will fail:

```
>>> from sympy.utilities.misc import func_name
>>> func_name(sqrt(x))
'Pow'
>>> sqrt(x).has(sqrt)
False
```

To find `sqrt` look for `Pow` with an exponent of $1/2$:

```
>>> (x + 1/sqrt(x)).find(lambda i: i.is_Pow and abs(i.exp) is S.Half)
{1/sqrt(x)}
```

See also:

[*sympy.polys.rootoftools.rootof*](#) (page 2431), [*root*](#) (page 422), [*real_root*](#) (page 426)

References

[R300], [R301]

cbt

`sympy.functions.elementary.miscellaneous.cbt(arg, evaluate=None)`

Returns the principal cube root.

Parameters

evaluate : bool, optional

The parameter determines if the expression should be evaluated. If None, its value is taken from `global_parameters.evaluate`.

Examples

```
>>> from sympy import cbrt, Symbol
>>> x = Symbol('x')
```

```
>>> cbrt(x)
x**(1/3)
```

```
>>> cbrt(x)**3
x
```

Note that `cbrt(x**3)` does not simplify to `x`.

```
>>> cbrt(x**3)
(x**3)**(1/3)
```

This is because the two are not equal to each other in general. For example, consider $x = -1$:

```
>>> from sympy import Eq
>>> Eq(cbrt(x**3), x).subs(x, -1)
False
```

This is because `cbrt` computes the principal cube root, this identity does hold if x is positive:

```
>>> y = Symbol('y', positive=True)
>>> cbrt(y**3)
y
```

See also:

[*sympy.polys.rootoftools.rootof*](#) (page 2431), [*root*](#) (page 422), [*real_root*](#) (page 426)

References

[R302], [R303]

real_root

`sympy.functions.elementary.miscellaneous.real_root(arg, n=None, evaluate=None)`
Return the real n 'th-root of *arg* if possible.

Parameters

n : int or None, optional

If *n* is None, then all instances of $(-n)^{1/\text{odd}}$ will be changed to $-n^{1/\text{odd}}$. This will only create a real root of a principal root. The presence of other factors may cause the result to not be real.

evaluate : bool, optional

The parameter determines if the expression should be evaluated. If None, its value is taken from `global_parameters.evaluate`.

Examples

```
>>> from sympy import root, real_root
```

```
>>> real_root(-8, 3)
-2
>>> root(-8, 3)
2*(-1)**(1/3)
>>> real_root(_)
-2
```

If one creates a non-principal root and applies `real_root`, the result will not be real (so use with caution):

```
>>> root(-8, 3, 2)
-2*(-1)**(2/3)
>>> real_root(_)
-2*(-1)**(2/3)
```

See also:

[`sympy.polys.rootoftools.rootof`](#) (page 2431), [`sympy.core.power.integer\_nthroot`](#) (page 1008), [`root`](#) (page 422), [`sqrt`](#) (page 424)

Combinatorial

This module implements various combinatorial functions.

bell

class `sympy.functions.combinatorial.numbers.bell(n, k_sym=None, symbols=None)`

Bell numbers / Bell polynomials

The Bell numbers satisfy $B_0 = 1$ and

$$B_n = \sum_{k=0}^{n-1} \binom{n-1}{k} B_k.$$

They are also given by:

$$B_n = \frac{1}{e} \sum_{k=0}^{\infty} \frac{k^n}{k!}.$$

The Bell polynomials are given by $B_0(x) = 1$ and

$$B_n(x) = x \sum_{k=1}^{n-1} \binom{n-1}{k-1} B_{k-1}(x).$$

The second kind of Bell polynomials (are sometimes called “partial” Bell polynomials or incomplete Bell polynomials) are defined as

$$B_{n,k}(x_1, x_2, \dots, x_{n-k+1}) = \sum_{\substack{j_1+j_2+j_3+\dots+k \\ j_1+2j_2+3j_3+\dots=n}} \frac{n!}{j_1!j_2!\dots j_{n-k+1}!} \left(\frac{x_1}{1!}\right)^{j_1} \left(\frac{x_2}{2!}\right)^{j_2} \dots \left(\frac{x_{n-k+1}}{(n-k+1)!}\right)^{j_{n-k+1}}.$$

- `bell(n)` gives the n^{th} Bell number, B_n .
- `bell(n, x)` gives the n^{th} Bell polynomial, $B_n(x)$.
- `bell(n, k, (x1, x2, ...))` gives Bell polynomials of the second kind, $B_{n,k}(x_1, x_2, \dots, x_{n-k+1})$.

Notes

Not to be confused with Bernoulli numbers and Bernoulli polynomials, which use the same notation.

Examples

```
>>> from sympy import bell, Symbol, symbols
```

```
>>> [bell(n) for n in range(11)]
[1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975]
>>> bell(30)
846749014511809332450147
>>> bell(4, Symbol('t'))
t**4 + 6*t**3 + 7*t**2 + t
>>> bell(6, 2, symbols('x:6')[1:])
6*x1*x5 + 15*x2*x4 + 10*x3**2
```

See also:

[*bernoulli*](#) (page 428), [*catalan*](#) (page 431), [*euler*](#) (page 433), [*fibonacci*](#) (page 437), [*harmonic*](#) (page 439), [*lucas*](#) (page 441), [*genocchi*](#) (page 442), [*partition*](#) (page 442), [*tribonacci*](#) (page 438)

References

[R193], [R194], [R195]

bernoulli

class `sympy.functions.combinatorial.numbers.bernoulli`(*n*, *sym=None*)

Bernoulli numbers / Bernoulli polynomials

The Bernoulli numbers are a sequence of rational numbers defined by $B_0 = 1$ and the recursive relation ($n > 0$):

$$0 = \sum_{k=0}^n \binom{n+1}{k} B_k$$

They are also commonly defined by their exponential generating function, which is $\frac{x}{e^x - 1}$. For odd indices > 1 , the Bernoulli numbers are zero.

The Bernoulli polynomials satisfy the analogous formula:

$$B_n(x) = \sum_{k=0}^n \binom{n}{k} B_k x^{n-k}$$

Bernoulli numbers and Bernoulli polynomials are related as $B_n(0) = B_n$.

We compute Bernoulli numbers using Ramanujan's formula:

$$B_n = \frac{A(n) - S(n)}{\binom{n+3}{n}}$$

where:

$$A(n) = \begin{cases} \frac{n+3}{3} & n \equiv 0 \text{ or } 2 \pmod{6} \\ -\frac{n+3}{6} & n \equiv 4 \pmod{6} \end{cases}$$

and:

$$S(n) = \sum_{k=1}^{\lfloor n/6 \rfloor} \binom{n+3}{n-6k} B_{n-6k}$$

This formula is similar to the sum given in the definition, but cuts 2/3 of the terms. For Bernoulli polynomials, we use the formula in the definition.

- `bernoulli(n)` gives the n th Bernoulli number, B_n
- `bernoulli(n, x)` gives the n th Bernoulli polynomial in x , $B_n(x)$

Examples

```
>>> from sympy import bernoulli
```

```
>>> [bernoulli(n) for n in range(11)]
[1, -1/2, 1/6, 0, -1/30, 0, 1/42, 0, -1/30, 0, 5/66]
>>> bernoulli(1000001)
0
```

See also:

[bell](#) (page 427), [catalan](#) (page 431), [euler](#) (page 433), [fibonacci](#) (page 437), [harmonic](#) (page 439), [lucas](#) (page 441), [genocchi](#) (page 442), [partition](#) (page 442), [tribonacci](#) (page 438)

References

[R196], [R197], [R198], [R199]

binomial

class sympy.functions.combinatorial.factorials.**binomial**(n , k)

Implementation of the binomial coefficient. It can be defined in two ways depending on its desired interpretation:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \text{ or } \binom{n}{k} = \frac{(n)_k}{k!}$$

First, in a strict combinatorial sense it defines the number of ways we can choose k elements from a set of n elements. In this case both arguments are nonnegative integers and binomial is computed using an efficient algorithm based on prime factorization.

The other definition is generalization for arbitrary n , however k must also be nonnegative. This case is very useful when evaluating summations.

For the sake of convenience, for negative integer k this function will return zero no matter the other argument.

To expand the binomial when n is a symbol, use either `expand_func()` or `expand(func=True)`. The former will keep the polynomial in factored form while the latter will expand the polynomial itself. See examples for details.

Examples

```
>>> from sympy import Symbol, Rational, binomial, expand_func
>>> n = Symbol('n', integer=True, positive=True)
```

```
>>> binomial(15, 8)
6435
```

```
>>> binomial(n, -1)
0
```

Rows of Pascal's triangle can be generated with the binomial function:

```
>>> for N in range(8):
...     print([binomial(N, i) for i in range(N + 1)])
...
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
[1, 5, 10, 10, 5, 1]
[1, 6, 15, 20, 15, 6, 1]
[1, 7, 21, 35, 35, 21, 7, 1]
```

As can a given diagonal, e.g. the 4th diagonal:

```
>>> N = -4
>>> [binomial(N, i) for i in range(1 - N)]
[1, -4, 10, -20, 35]
```

```
>>> binomial(Rational(5, 4), 3)
-5/128
>>> binomial(Rational(-5, 4), 3)
-195/128
```

```
>>> binomial(n, 3)
binomial(n, 3)
```

```
>>> binomial(n, 3).expand(func=True)
n**3/6 - n**2/2 + n/3
```

```
>>> expand_func(binomial(n, 3))
n*(n - 2)*(n - 1)/6
```

References

[R200]

catalan

class sympy.functions.combinatorial.numbers.**catalan**(*n*)
Catalan numbers

The n^{th} catalan number is given by:

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

- **catalan**(*n*) gives the n^{th} Catalan number, C_n

Examples

```
>>> from sympy import (Symbol, binomial, gamma, hyper,
...                    catalan, diff, combsimp, Rational, I)
```

```
>>> [catalan(i) for i in range(1,10)]
[1, 2, 5, 14, 42, 132, 429, 1430, 4862]
```

```
>>> n = Symbol("n", integer=True)
```

```
>>> catalan(n)
catalan(n)
```

Catalan numbers can be transformed into several other, identical expressions involving other mathematical functions

```
>>> catalan(n).rewrite(binomial)
binomial(2*n, n)/(n + 1)
```

```
>>> catalan(n).rewrite(gamma)
4**n*gamma(n + 1/2)/(sqrt(pi)*gamma(n + 2))
```

```
>>> catalan(n).rewrite(hyper)
hyper((1 - n, -n), (2,), 1)
```

For some non-integer values of n we can get closed form expressions by rewriting in terms of gamma functions:

```
>>> catalan(Rational(1, 2)).rewrite(gamma)
8/(3*pi)
```

We can differentiate the Catalan numbers $C(n)$ interpreted as a continuous real function in n :

```
>>> diff(catalan(n), n)
(polygamma(0, n + 1/2) - polygamma(0, n + 2) + log(4))*catalan(n)
```

As a more advanced example consider the following ratio between consecutive numbers:

```
>>> combsimp((catalan(n + 1)/catalan(n)).rewrite(binomial))
2*(2*n + 1)/(n + 2)
```

The Catalan numbers can be generalized to complex numbers:

```
>>> catalan(I).rewrite(gamma)
4**I*gamma(1/2 + I)/(sqrt(pi)*gamma(2 + I))
```

and evaluated with arbitrary precision:

```
>>> catalan(I).evalf(20)
0.39764993382373624267 - 0.020884341620842555705*I
```

See also:

bell (page 427), *bernoulli* (page 428), *euler* (page 433), *fibonacci* (page 437), *harmonic* (page 439), *lucas* (page 441), *genocchi* (page 442), *partition* (page 442), *tribonacci* (page 438), *sympy.functions.combinatorial.factorials.binomial* (page 430)

References

[R201], [R202], [R203], [R204]

euler

class `sympy.functions.combinatorial.numbers.euler(m, sym=None)`

Euler numbers / Euler polynomials

The Euler numbers are given by:

$$E_{2n} = I \sum_{k=1}^{2n+1} \sum_{j=0}^k \binom{k}{j} \frac{(-1)^j (k-2j)^{2n+1}}{2^k I^k k}$$

$$E_{2n+1} = 0$$

Euler numbers and Euler polynomials are related by

$$E_n = 2^n E_n \left(\frac{1}{2} \right).$$

We compute symbolic Euler polynomials using [\[R209\]](#)

$$E_n(x) = \sum_{k=0}^n \binom{n}{k} \frac{E_k}{2^k} \left(x - \frac{1}{2} \right)^{n-k}.$$

However, numerical evaluation of the Euler polynomial is computed more efficiently (and more accurately) using the `mpmath` library.

- `euler(n)` gives the n^{th} Euler number, E_n .
- `euler(n, x)` gives the n^{th} Euler polynomial, $E_n(x)$.

Examples

```
>>> from sympy import euler, Symbol, S
>>> [euler(n) for n in range(10)]
[1, 0, -1, 0, 5, 0, -61, 0, 1385, 0]
>>> n = Symbol("n")
>>> euler(n + 2*n)
euler(3*n)
```

```
>>> x = Symbol("x")
>>> euler(n, x)
euler(n, x)
```

```
>>> euler(0, x)
1
>>> euler(1, x)
x - 1/2
>>> euler(2, x)
x**2 - x
>>> euler(3, x)
x**3 - 3*x**2/2 + 1/4
>>> euler(4, x)
x**4 - 2*x**3 + x
```

```
>>> euler(12, S.Half)
2702765/4096
>>> euler(12)
2702765
```

See also:

[bell](#) (page 427), [bernoulli](#) (page 428), [catalan](#) (page 431), [fibonacci](#) (page 437), [harmonic](#) (page 439), [lucas](#) (page 441), [genocchi](#) (page 442), [partition](#) (page 442), [tribonacci](#) (page 438)

References

[R205], [R206], [R207], [R208], [R209]

factorial

class sympy.functions.combinatorial.factorials.**factorial**(*n*)

Implementation of factorial function over nonnegative integers. By convention (consistent with the gamma function and the binomial coefficients), factorial of a negative integer is complex infinity.

The factorial is very important in combinatorics where it gives the number of ways in which n objects can be permuted. It also arises in calculus, probability, number theory, etc.

There is strict relation of factorial with gamma function. In fact $n! = \text{gamma}(n + 1)$ for nonnegative integers. Rewrite of this kind is very useful in case of combinatorial simplification.

Computation of the factorial is done using two algorithms. For small arguments a pre-computed look up table is used. However for bigger input algorithm Prime-Swing is used. It is the fastest algorithm known and computes $n!$ via prime factorization of special class of numbers, called here the ‘Swing Numbers’.

Examples

```
>>> from sympy import Symbol, factorial, S
>>> n = Symbol('n', integer=True)
```

```
>>> factorial(0)
1
```

```
>>> factorial(7)
5040
```

```
>>> factorial(-2)
zoo
```

```
>>> factorial(n)
factorial(n)
```

```
>>> factorial(2*n)
factorial(2*n)
```

```
>>> factorial(S(1)/2)
factorial(1/2)
```

See also:

[*factorial2*](#) (page 436), [*RisingFactorial*](#) (page 443), [*FallingFactorial*](#) (page 436)

subfactorial

class `sympy.functions.combinatorial.factorials.subfactorial`(*arg*)

The subfactorial counts the derangements of n items and is defined for non-negative integers as:

$$!n = \begin{cases} 1 & n = 0 \\ 0 & n = 1 \\ (n-1)(!(n-1)+!(n-2)) & n > 1 \end{cases}$$

It can also be written as `int(round(n!/exp(1)))` but the recursive definition with caching is implemented for this function.

An interesting analytic expression is the following [R211]

$$!x = \Gamma(x+1, -1)/e$$

which is valid for non-negative integers x . The above formula is not very useful in case of non-integers. $\Gamma(x+1, -1)$ is single-valued only for integral arguments x , elsewhere on the positive real axis it has an infinite number of branches none of which are real.

Examples

```
>>> from sympy import subfactorial
>>> from sympy.abc import n
>>> subfactorial(n + 1)
subfactorial(n + 1)
>>> subfactorial(5)
44
```

See also:

[*factorial*](#) (page 434), [*uppergamma*](#) (page 466), [*sympy.utilities.iterables.generate_derangements*](#) (page 2076)

References

[R210], [R211]

factorial2 / double factorial

class sympy.functions.combinatorial.factorials.**factorial2**(arg)

The double factorial $n!!$, not to be confused with $(n!)!$

The double factorial is defined for nonnegative integers and for odd negative integers as:

$$n!! = \begin{cases} 1 & n = 0 \\ n(n-2)(n-4) \cdots 1 & n \text{ positive odd} \\ n(n-2)(n-4) \cdots 2 & n \text{ positive even} \\ (n+2)!!/(n+2) & n \text{ negative odd} \end{cases}$$

Examples

```
>>> from sympy import factorial2, var
>>> n = var('n')
>>> n
n
>>> factorial2(n + 1)
factorial2(n + 1)
>>> factorial2(5)
15
>>> factorial2(-1)
1
>>> factorial2(-5)
1/3
```

See also:

[factorial](#) (page 434), [RisingFactorial](#) (page 443), [FallingFactorial](#) (page 436)

References

[R212]

FallingFactorial

class sympy.functions.combinatorial.factorials.**FallingFactorial**(x, k)

Falling factorial (related to rising factorial) is a double valued function arising in concrete mathematics, hypergeometric functions and series expansions. It is defined by

$$\text{ff}(x, k) = (x)_k = x \cdot (x-1) \cdots (x-k+1)$$

where x can be arbitrary expression and k is an integer. For more information check “Concrete mathematics” by Graham, pp. 66 or [R213].

When x is a `.Poly` instance of degree ≥ 1 with single variable, $(x)_k = x(y) \cdot x(y-1) \cdots x(y-k+1)$, where y is the variable of x . This is as described in

```
>>> from sympy import ff, Poly, Symbol
>>> from sympy.abc import x
>>> n = Symbol('n', integer=True)

>>> ff(x, 0)
1
>>> ff(5, 5)
120
>>> ff(x, 5) == x*(x - 1)*(x - 2)*(x - 3)*(x - 4)
True
>>> ff(Poly(x**2, x), 2)
Poly(x**4 - 2*x**3 + x**2, x, domain='ZZ')
>>> ff(n, n)
factorial(n)
```

Rewriting is complicated unless the relationship between the arguments is known, but falling factorial can be rewritten in terms of gamma, factorial and binomial and rising factorial.

```
>>> from sympy import factorial, rf, gamma, binomial, Symbol
>>> n = Symbol('n', integer=True, positive=True)
>>> F = ff(n, n - 2)
>>> for i in (rf, ff, factorial, binomial, gamma):
...     F.rewrite(i)
...
RisingFactorial(3, n - 2)
FallingFactorial(n, n - 2)
factorial(n)/2
binomial(n, n - 2)*factorial(n - 2)
gamma(n + 1)/2
```

See also:

[factorial](#) (page 434), [factorial2](#) (page 436), [RisingFactorial](#) (page 443)

References

[R213], [R214]

fibonacci

class `sympy.functions.combinatorial.numbers.fibonacci`(n , `sym=None`)

Fibonacci numbers / Fibonacci polynomials

The Fibonacci numbers are the integer sequence defined by the initial terms $F_0 = 0$, $F_1 = 1$ and the two-term recurrence relation $F_n = F_{n-1} + F_{n-2}$. This definition extended to arbitrary real and complex arguments using the formula

$$F_z = \frac{\phi^z - \cos(\pi z)\phi^{-z}}{\sqrt{5}}$$

The Fibonacci polynomials are defined by $F_1(x) = 1$, $F_2(x) = x$, and $F_n(x) = x * F_{n-1}(x) + F_{n-2}(x)$ for $n > 2$. For all positive integers n , $F_n(1) = F_n$.

- `fibonacci(n)` gives the n^{th} Fibonacci number, F_n
- `fibonacci(n, x)` gives the n^{th} Fibonacci polynomial in x , $F_n(x)$

Examples

```
>>> from sympy import fibonacci, Symbol
```

```
>>> [fibonacci(x) for x in range(11)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> fibonacci(5, Symbol('t'))
t**4 + 3*t**2 + 1
```

See also:

[bell](#) (page 427), [bernoulli](#) (page 428), [catalan](#) (page 431), [euler](#) (page 433), [harmonic](#) (page 439), [lucas](#) (page 441), [genocchi](#) (page 442), [partition](#) (page 442), [tribonacci](#) (page 438)

References

[R216], [R217]

tribonacci

class `sympy.functions.combinatorial.numbers.tribonacci(n, sym=None)`

Tribonacci numbers / Tribonacci polynomials

The Tribonacci numbers are the integer sequence defined by the initial terms $T_0 = 0$, $T_1 = 1$, $T_2 = 1$ and the three-term recurrence relation $T_n = T_{n-1} + T_{n-2} + T_{n-3}$.

The Tribonacci polynomials are defined by $T_0(x) = 0$, $T_1(x) = 1$, $T_2(x) = x^2$, and $T_n(x) = x^2 T_{n-1}(x) + x T_{n-2}(x) + T_{n-3}(x)$ for $n > 2$. For all positive integers n , $T_n(1) = T_n$.

- `tribonacci(n)` gives the n^{th} Tribonacci number, T_n
- `tribonacci(n, x)` gives the n^{th} Tribonacci polynomial in x , $T_n(x)$

Examples

```
>>> from sympy import tribonacci, Symbol
```

```
>>> [tribonacci(x) for x in range(11)]
[0, 1, 1, 2, 4, 7, 13, 24, 44, 81, 149]
>>> tribonacci(5, Symbol('t'))
t**8 + 3*t**5 + 3*t**2
```

See also:

[bell](#) (page 427), [bernoulli](#) (page 428), [catalan](#) (page 431), [euler](#) (page 433), [fibonacci](#) (page 437), [harmonic](#) (page 439), [lucas](#) (page 441), [genocchi](#) (page 442), [partition](#) (page 442)

References

[R218], [R219], [R220]

harmonic

class `sympy.functions.combinatorial.numbers.harmonic(n, m=None)`

Harmonic numbers

The n th harmonic number is given by $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$.

More generally:

$$H_{n,m} = \sum_{k=1}^n \frac{1}{k^m}$$

As $n \rightarrow \infty$, $H_{n,m} \rightarrow \zeta(m)$, the Riemann zeta function.

- `harmonic(n)` gives the n th harmonic number, H_n
- `harmonic(n, m)` gives the n th generalized harmonic number of order m , $H_{n,m}$, where `harmonic(n) == harmonic(n, 1)`

Examples

```
>>> from sympy import harmonic, oo
```

```
>>> [harmonic(n) for n in range(6)]
[0, 1, 3/2, 11/6, 25/12, 137/60]
>>> [harmonic(n, 2) for n in range(6)]
[0, 1, 5/4, 49/36, 205/144, 5269/3600]
>>> harmonic(oo, 2)
pi**2/6
```

```
>>> from sympy import Symbol, Sum
>>> n = Symbol("n")
```

```
>>> harmonic(n).rewrite(Sum)
Sum(1/_k, (_k, 1, n))
```

We can evaluate harmonic numbers for all integral and positive rational arguments:

```
>>> from sympy import S, expand_func, simplify
>>> harmonic(8)
761/280
```

(continues on next page)

(continued from previous page)

```
>>> harmonic(11)
83711/27720
```

```
>>> H = harmonic(1/S(3))
>>> H
harmonic(1/3)
>>> He = expand_func(H)
>>> He
-log(6) - sqrt(3)*pi/6 + 2*Sum(log(sin(_k*pi/3))*cos(2*_k*pi/3), (_k, 1,
→ 1))
                        + 3*Sum(1/(3*_k + 1), (_k, 0, 0))
>>> He.doit()
-log(6) - sqrt(3)*pi/6 - log(sqrt(3)/2) + 3
>>> H = harmonic(25/S(7))
>>> He = simplify(expand_func(H).doit())
>>> He
log(sin(2*pi/7)**(2*cos(16*pi/7))/(14*sin(pi/7)**(2*cos(pi/7))*cos(pi/
→ 14)**(2*sin(pi/14)))) + pi*tan(pi/14)/2 + 30247/9900
>>> He.n(40)
1.983697455232980674869851942390639915940
>>> harmonic(25/S(7)).n(40)
1.983697455232980674869851942390639915940
```

We can rewrite harmonic numbers in terms of polygamma functions:

```
>>> from sympy import digamma, polygamma
>>> m = Symbol("m")
```

```
>>> harmonic(n).rewrite(digamma)
polygamma(0, n + 1) + EulerGamma
```

```
>>> harmonic(n).rewrite(polygamma)
polygamma(0, n + 1) + EulerGamma
```

```
>>> harmonic(n,3).rewrite(polygamma)
polygamma(2, n + 1)/2 - polygamma(2, 1)/2
```

```
>>> harmonic(n,m).rewrite(polygamma)
(-1)**m*(polygamma(m - 1, 1) - polygamma(m - 1, n + 1))/factorial(m - 1)
```

Integer offsets in the argument can be pulled out:

```
>>> from sympy import expand_func
```

```
>>> expand_func(harmonic(n+4))
harmonic(n) + 1/(n + 4) + 1/(n + 3) + 1/(n + 2) + 1/(n + 1)
```

```
>>> expand_func(harmonic(n-4))
harmonic(n) - 1/(n - 1) - 1/(n - 2) - 1/(n - 3) - 1/n
```

Some limits can be computed as well:


```
>>> from sympy import limit, oo
```

```
>>> limit(harmonic(n), n, oo)
oo
```

```
>>> limit(harmonic(n, 2), n, oo)
pi**2/6
```

```
>>> limit(harmonic(n, 3), n, oo)
-polygamma(2, 1)/2
```

However we cannot compute the general relation yet:

```
>>> limit(harmonic(n, m), n, oo)
harmonic(oo, m)
```

which equals $\zeta(m)$ for $m > 1$.

See also:

[bell](#) (page 427), [bernoulli](#) (page 428), [catalan](#) (page 431), [euler](#) (page 433), [fibonacci](#) (page 437), [lucas](#) (page 441), [genocchi](#) (page 442), [partition](#) (page 442), [tribonacci](#) (page 438)

References

[R221], [R222], [R223]

lucas

class sympy.functions.combinatorial.numbers.lucas(*n*)

Lucas numbers

Lucas numbers satisfy a recurrence relation similar to that of the Fibonacci sequence, in which each term is the sum of the preceding two. They are generated by choosing the initial values $L_0 = 2$ and $L_1 = 1$.

- `lucas(n)` gives the n^{th} Lucas number

Examples

```
>>> from sympy import lucas
```

```
>>> [lucas(x) for x in range(11)]
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123]
```

See also:

[bell](#) (page 427), [bernoulli](#) (page 428), [catalan](#) (page 431), [euler](#) (page 433), [fibonacci](#) (page 437), [harmonic](#) (page 439), [genocchi](#) (page 442), [partition](#) (page 442), [tribonacci](#) (page 438)

References

[R224], [R225]

genocchi

class sympy.functions.combinatorial.numbers.genocchi(*n*)

Genocchi numbers

The Genocchi numbers are a sequence of integers G_n that satisfy the relation:

$$\frac{2t}{e^t + 1} = \sum_{n=1}^{\infty} \frac{G_n t^n}{n!}$$

Examples

```
>>> from sympy import genocchi, Symbol
>>> [genocchi(n) for n in range(1, 9)]
[1, -1, 0, 1, 0, -3, 0, 17]
>>> n = Symbol('n', integer=True, positive=True)
>>> genocchi(2*n + 1)
0
```

See also:

bell (page 427), *bernoulli* (page 428), *catalan* (page 431), *euler* (page 433), *fibonacci* (page 437), *harmonic* (page 439), *lucas* (page 441), *partition* (page 442), *tribonacci* (page 438)

References

[R226], [R227]

partition

class sympy.functions.combinatorial.numbers.partition(*n*)

Partition numbers

The Partition numbers are a sequence of integers p_n that represent the number of distinct ways of representing n as a sum of natural numbers (with order irrelevant). The generating function for p_n is given by:

$$\sum_{n=0}^{\infty} p_n x^n = \prod_{k=1}^{\infty} (1 - x^k)^{-1}$$

Examples

```
>>> from sympy import partition, Symbol
>>> [partition(n) for n in range(9)]
[1, 1, 2, 3, 5, 7, 11, 15, 22]
>>> n = Symbol('n', integer=True, negative=True)
>>> partition(n)
0
```

See also:

bell (page 427), *bernoulli* (page 428), *catalan* (page 431), *euler* (page 433), *fibonacci* (page 437), *harmonic* (page 439), *lucas* (page 441), *genocchi* (page 442), *tribonacci* (page 438)

References

[R228], [R229]

MultiFactorial

```
class sympy.functions.combinatorial.factorials.MultiFactorial(*args)
```

RisingFactorial

```
class sympy.functions.combinatorial.factorials.RisingFactorial(x, k)
```

Rising factorial (also called Pochhammer symbol [R230]) is a double valued function arising in concrete mathematics, hypergeometric functions and series expansions. It is defined by:

$$\text{rf}(y, k) = (x)^k = x \cdot (x+1) \cdots (x+k-1)$$

where x can be arbitrary expression and k is an integer. For more information check “Concrete mathematics” by Graham, pp. 66 or visit <http://mathworld.wolfram.com/RisingFactorial.html> page.

When x is a `.Poly` instance of degree ≥ 1 with a single variable, $(x)^k = x(y) \cdot x(y+1) \cdots x(y+k-1)$, where y is the variable of x . This is as described in [R231].

Examples

```
>>> from sympy import rf, Poly
>>> from sympy.abc import x
>>> rf(x, 0)
1
>>> rf(1, 5)
120
>>> rf(x, 5) == x*(1 + x)*(2 + x)*(3 + x)*(4 + x)
```

(continues on next page)

(continued from previous page)

```
True
>>> rf(Poly(x**3, x), 2)
Poly(x**6 + 3*x**5 + 3*x**4 + x**3, x, domain='ZZ')
```

Rewriting is complicated unless the relationship between the arguments is known, but rising factorial can be rewritten in terms of gamma, factorial, binomial, and falling factorial.

```
>>> from sympy import Symbol, factorial, ff, binomial, gamma
>>> n = Symbol('n', integer=True, positive=True)
>>> R = rf(n, n + 2)
>>> for i in (rf, ff, factorial, binomial, gamma):
...     R.rewrite(i)
...
RisingFactorial(n, n + 2)
FallingFactorial(2*n + 1, n + 2)
factorial(2*n + 1)/factorial(n - 1)
binomial(2*n + 1, n + 2)*factorial(n + 2)
gamma(2*n + 2)/gamma(n)
```

See also:

[factorial](#) (page 434), [factorial2](#) (page 436), [FallingFactorial](#) (page 436)

References

[R230], [R231]

stirling

`sympy.functions.combinatorial.numbers.stirling(n, k, d=None, kind=2, signed=False)`

Return Stirling number $S(n, k)$ of the first or second (default) kind.

The sum of all Stirling numbers of the second kind for $k = 1$ through n is `bell(n)`. The recurrence relationship for these numbers is:

$$\begin{aligned} \begin{Bmatrix} 0 \\ 0 \end{Bmatrix} &= 1; \begin{Bmatrix} n \\ 0 \end{Bmatrix} = \begin{Bmatrix} 0 \\ k \end{Bmatrix} = 0; \\ \begin{Bmatrix} n+1 \\ k \end{Bmatrix} &= j \begin{Bmatrix} n \\ k \end{Bmatrix} + \begin{Bmatrix} n \\ k-1 \end{Bmatrix} \end{aligned}$$

where j is:

n for Stirling numbers of the first kind, $-n$ for signed Stirling numbers of the first kind, k for Stirling numbers of the second kind.

The first kind of Stirling number counts the number of permutations of n distinct items that have k cycles; the second kind counts the ways in which n distinct items can be partitioned into k parts. If d is given, the “reduced Stirling number of the second kind” is returned: $S^d(n, k) = S(n - d + 1, k - d + 1)$ with $n \geq k \geq d$. (This counts the ways to partition n consecutive integers into k groups with no pairwise difference less than d . See example below.)

To obtain the signed Stirling numbers of the first kind, use keyword `signed=True`. Using this keyword automatically sets `kind` to 1.

Examples

```
>>> from sympy.functions.combinatorial.numbers import stirling, bell
>>> from sympy.combinatorics import Permutation
>>> from sympy.utilities.iterables import multiset_partitions, p
    ↪ permutations
```

First kind (unsigned by default):

```
>>> [stirling(6, i, kind=1) for i in range(7)]
[0, 120, 274, 225, 85, 15, 1]
>>> perms = list(permutations(range(4)))
>>> [sum(Permutation(p).cycles == i for p in perms) for i in range(5)]
[0, 6, 11, 6, 1]
>>> [stirling(4, i, kind=1) for i in range(5)]
[0, 6, 11, 6, 1]
```

First kind (signed):

```
>>> [stirling(4, i, signed=True) for i in range(5)]
[0, -6, 11, -6, 1]
```

Second kind:

```
>>> [stirling(10, i) for i in range(12)]
[0, 1, 511, 9330, 34105, 42525, 22827, 5880, 750, 45, 1, 0]
>>> sum(_) == bell(10)
True
>>> len(list(multiset_partitions(range(4), 2))) == stirling(4, 2)
True
```

Reduced second kind:

```
>>> from sympy import subsets, oo
>>> def delta(p):
...     if len(p) == 1:
...         return oo
...     return min(abs(i[0] - i[1]) for i in subsets(p, 2))
>>> parts = multiset_partitions(range(5), 3)
>>> d = 2
>>> sum(1 for p in parts if all(delta(i) >= d for i in p))
7
>>> stirling(5, 3, 2)
7
```

See also:

[*sympy.utilities.iterables.multiset_partitions*](#) (page 2084)

References

[R232], [R233]

Enumeration

Three functions are available. Each of them attempts to efficiently compute a given combinatorial quantity for a given set or multiset which can be entered as an integer, sequence or multiset (dictionary with elements as keys and multiplicities as values). The *k* parameter indicates the number of elements to pick (or the number of partitions to make). When *k* is *None*, the sum of the enumeration for all *k* (from 0 through the number of items represented by *n*) is returned. A replacement parameter is recognized for combinations and permutations; this indicates that any item may appear with multiplicity as high as the number of items in the original set.

```
>>> from sympy.functions.combinatorial.numbers import nC, nP, nT
>>> items = 'baby'
```

nC

`sympy.functions.combinatorial.numbers.nC(n, k=None, replacement=False)`

Return the number of combinations of *n* items taken *k* at a time.

Possible values for *n*:

- integer - set of length *n*
- sequence - converted to a multiset internally
- multiset - {element: multiplicity}

If *k* is *None* then the total of all combinations of length 0 through the number of items represented in *n* will be returned.

If replacement is *True* then a given item can appear more than once in the *k* items. (For example, for 'ab' sets of 2 would include 'aa', 'ab', and 'bb'.) The multiplicity of elements in *n* is ignored when replacement is *True* but the total number of elements is considered since no element can appear more times than the number of elements in *n*.

Examples

```
>>> from sympy.functions.combinatorial.numbers import nC
>>> from sympy.utilities.iterables import multiset_combinations
>>> nC(3, 2)
3
>>> nC('abc', 2)
3
>>> nC('aab', 2)
2
```

When replacement is *True*, each item can have multiplicity equal to the length represented by *n*:

```
>>> nC('abc', replacement=True)
35
>>> [len(list(multiset_combinations('aaaabbbbcccc', i))) for i in
→ range(5)]
[1, 3, 6, 10, 15]
>>> sum(_)
35
```

If there are k items with multiplicities $m_1, m_2, \dots, m_k$ then the total of all combinations of length 0 through k is the product, $(m_1 + 1)(m_2 + 1)\dots(m_k + 1)$. When the multiplicity of each item is 1 (i.e., k unique items) then there are 2^k combinations. For example, if there are 4 unique items, the total number of combinations is 16:

```
>>> sum(nC(4, i) for i in range(5))
16
```

See also:

[`sympy.utilities.iterables.multiset\_combinations`](#) (page 2083)

References

[R234], [R235]

nP

`sympy.functions.combinatorial.numbers.nP(n, k=None, replacement=False)`

Return the number of permutations of n items taken k at a time.

Possible values for n :

- integer - set of length n
- sequence - converted to a multiset internally
- multiset - {element: multiplicity}

If k is `None` then the total of all permutations of length 0 through the number of items represented by n will be returned.

If `replacement` is `True` then a given item can appear more than once in the k items. (For example, for 'ab' permutations of 2 would include 'aa', 'ab', 'ba' and 'bb'.) The multiplicity of elements in n is ignored when `replacement` is `True` but the total number of elements is considered since no element can appear more times than the number of elements in n .

Examples

```
>>> from sympy.functions.combinatorial.numbers import nP
>>> from sympy.utilities.iterables import multiset_permutations, multiset
>>> nP(3, 2)
6
>>> nP('abc', 2) == nP(multiset('abc'), 2) == 6
True
>>> nP('aab', 2)
3
>>> nP([1, 2, 2], 2)
3
>>> [nP(3, i) for i in range(4)]
[1, 3, 6, 6]
>>> nP(3) == sum(_)
True
```

When replacement is True, each item can have multiplicity equal to the length represented by n:

```
>>> nP('aabc', replacement=True)
121
>>> [len(list(multiset_permutations('aaaabbbbcccc', i))) for i in
↪ range(5)]
[1, 3, 9, 27, 81]
>>> sum(_)
121
```

See also:

[`sympy.utilities.iterables.multiset\_permutations`](#) (page 2086)

References

[R236]

nT

`sympy.functions.combinatorial.numbers.nT(n, k=None)`

Return the number of k-sized partitions of n items.

Possible values for n:

- integer - n identical items
- sequence - converted to a multiset internally
- multiset - {element: multiplicity}

Note: the convention for nT is different than that of nC and nP in that here an integer indicates n *identical* items instead of a set of length n; this is in keeping with the partitions function which treats its integer-n input like a list of n 1s. One can use range(n) for n to indicate n distinct items.

If `k` is `None` then the total number of ways to partition the elements represented in `n` will be returned.

Examples

```
>>> from sympy.functions.combinatorial.numbers import nT
```

Partitions of the given multiset:

```
>>> [nT('aabbcc', i) for i in range(1, 7)]
[1, 8, 11, 5, 1, 0]
>>> nT('aabbcc') == sum(_)
True
```

```
>>> [nT("mississippi", i) for i in range(1, 12)]
[1, 74, 609, 1521, 1768, 1224, 579, 197, 50, 9, 1]
```

Partitions when all items are identical:

```
>>> [nT(5, i) for i in range(1, 6)]
[1, 2, 2, 1, 1]
>>> nT('1'*5) == sum(_)
True
```

When all items are different:

```
>>> [nT(range(5), i) for i in range(1, 6)]
[1, 15, 25, 10, 1]
>>> nT(range(5)) == sum(_)
True
```

Partitions of an integer expressed as a sum of positive integers:

```
>>> from sympy import partition
>>> partition(4)
5
>>> nT(4, 1) + nT(4, 2) + nT(4, 3) + nT(4, 4)
5
>>> nT('1'*4)
5
```

See also:

[*sympy.utilities.iterables.partitions*](#) (page 2089), [*sympy.utilities.iterables.multiset_partitions*](#) (page 2084), [*sympy.functions.combinatorial.numbers.partition*](#) (page 442)

References

[R237]

Special

DiracDelta

class sympy.functions.special.delta_functions.**DiracDelta**(arg, k=0)

The DiracDelta function and its derivatives.

Explanation

DiracDelta is not an ordinary function. It can be rigorously defined either as a distribution or as a measure.

DiracDelta only makes sense in definite integrals, and in particular, integrals of the form $\text{Integral}(f(x)*\text{DiracDelta}(x - x_0), (x, a, b))$, where it equals $f(x_0)$ if $a \leq x_0 \leq b$ and 0 otherwise. Formally, DiracDelta acts in some ways like a function that is 0 everywhere except at 0, but in many ways it also does not. It can often be useful to treat DiracDelta in formal ways, building up and manipulating expressions with delta functions (which may eventually be integrated), but care must be taken to not treat it as a real function. SymPy's ∞ is similar. It only truly makes sense formally in certain contexts (such as integration limits), but SymPy allows its use everywhere, and it tries to be consistent with operations on it (like $1/\infty$), but it is easy to get into trouble and get wrong results if ∞ is treated too much like a number. Similarly, if DiracDelta is treated too much like a function, it is easy to get wrong or nonsensical results.

DiracDelta function has the following properties:

- 1) $\frac{d}{dx}\theta(x) = \delta(x)$
- 2) $\int_{-\infty}^{\infty} \delta(x - a)f(x) dx = f(a)$ and $\int_{a-\epsilon}^{a+\epsilon} \delta(x - a)f(x) dx = f(a)$
- 3) $\delta(x) = 0$ for all $x \neq 0$
- 4) $\delta(g(x)) = \sum_i \frac{\delta(x-x_i)}{\|g'(x_i)\|}$ where x_i are the roots of g
- 5) $\delta(-x) = \delta(x)$

Derivatives of k-th order of DiracDelta have the following properties:

- 6) $\delta(x, k) = 0$ for all $x \neq 0$
- 7) $\delta(-x, k) = -\delta(x, k)$ for odd k
- 8) $\delta(-x, k) = \delta(x, k)$ for even k

Examples

```
>>> from sympy import DiracDelta, diff, pi
>>> from sympy.abc import x, y

>>> DiracDelta(x)
DiracDelta(x)
>>> DiracDelta(1)
0
>>> DiracDelta(-1)
0
>>> DiracDelta(pi)
0
>>> DiracDelta(x - 4).subs(x, 4)
DiracDelta(0)
>>> diff(DiracDelta(x))
DiracDelta(x, 1)
>>> diff(DiracDelta(x - 1), x, 2)
DiracDelta(x - 1, 2)
>>> diff(DiracDelta(x**2 - 1), x, 2)
2*(2*x**2*DiracDelta(x**2 - 1, 2) + DiracDelta(x**2 - 1, 1))
>>> DiracDelta(3*x).is_simple(x)
True
>>> DiracDelta(x**2).is_simple(x)
False
>>> DiracDelta((x**2 - 1)*y).expand(diracdelta=True, wrt=x)
DiracDelta(x - 1)/(2*Abs(y)) + DiracDelta(x + 1)/(2*Abs(y))
```

See also:

[Heaviside](#) (page 454), [sympy.simplify.simplify.simplify](#) (page 661), [is_simple](#) (page 453), [sympy.functions.special.tensor_functions.KroneckerDelta](#) (page 549)

References

[R304]

classmethod `eval(arg, k=0)`

Returns a simplified form or a value of DiracDelta depending on the argument passed by the DiracDelta object.

Parameters

k : integer

order of derivative

arg : argument passed to DiracDelta

Explanation

The `eval()` method is automatically called when the `DiracDelta` class is about to be instantiated and it returns either some simplified instance or the unevaluated instance depending on the argument passed. In other words, `eval()` method is not needed to be called explicitly, it is being called and evaluated once the object is called.

Examples

```
>>> from sympy import DiracDelta, S
>>> from sympy.abc import x
```

```
>>> DiracDelta(x)
DiracDelta(x)
```

```
>>> DiracDelta(-x, 1)
-DiracDelta(x, 1)
```

```
>>> DiracDelta(1)
0
```

```
>>> DiracDelta(5, 1)
0
```

```
>>> DiracDelta(0)
DiracDelta(0)
```

```
>>> DiracDelta(-1)
0
```

```
>>> DiracDelta(S.NaN)
nan
```

```
>>> DiracDelta(x - 100).subs(x, 5)
0
```

```
>>> DiracDelta(x - 100).subs(x, 100)
DiracDelta(0)
```

`fdiff(argindex=1)`

Returns the first derivative of a `DiracDelta` Function.

Parameters

`argindex` : integer

degree of derivative

Explanation

The difference between `diff()` and `fdiff()` is: `diff()` is the user-level function and `fdiff()` is an object method. `fdiff()` is a convenience method available in the `Function` class. It returns the derivative of the function without considering the chain rule. `diff(function, x)` calls `Function._eval_derivative` which in turn calls `fdiff()` internally to compute the derivative of the function.

Examples

```
>>> from sympy import DiracDelta, diff
>>> from sympy.abc import x
```

```
>>> DiracDelta(x).fdiff()
DiracDelta(x, 1)
```

```
>>> DiracDelta(x, 1).fdiff()
DiracDelta(x, 2)
```

```
>>> DiracDelta(x**2 - 1).fdiff()
DiracDelta(x**2 - 1, 1)
```

```
>>> diff(DiracDelta(x, 1)).fdiff()
DiracDelta(x, 3)
```

`is_simple(x)`

Tells whether the argument(`args[0]`) of `DiracDelta` is a linear expression in `x`.

Parameters

x : can be a symbol

Examples

```
>>> from sympy import DiracDelta, cos
>>> from sympy.abc import x, y
```

```
>>> DiracDelta(x*y).is_simple(x)
True
>>> DiracDelta(x*y).is_simple(y)
True
```

```
>>> DiracDelta(x**2 + x - 2).is_simple(x)
False
```

```
>>> DiracDelta(cos(x)).is_simple(x)
False
```

See also:

[`sympy.simplify.simplify.simplify`](#) (page 661), [`DiracDelta`](#) (page 450)

Heaviside

class sympy.functions.special.delta_functions.**Heaviside**(arg, H0=1 / 2)
Heaviside step function.

Explanation

The Heaviside step function has the following properties:

- 1) $\frac{d}{dx}\theta(x) = \delta(x)$
- 2) $\theta(x) = \begin{cases} 0 & \text{for } x < 0 \\ \frac{1}{2} & \text{for } x = 0 \\ 1 & \text{for } x > 0 \end{cases}$
- 3) $\frac{d}{dx}\max(x, 0) = \theta(x)$

Heaviside(x) is printed as $\theta(x)$ with the SymPy LaTeX printer.

The value at 0 is set differently in different fields. SymPy uses 1/2, which is a convention from electronics and signal processing, and is consistent with solving improper integrals by Fourier transform and convolution.

To specify a different value of Heaviside at $x=0$, a second argument can be given. Using Heaviside(x, nan) gives an expression that will evaluate to nan for $x=0$.

Changed in version 1.9: Heaviside(0) now returns 1/2 (before: undefined)

Examples

```
>>> from sympy import Heaviside, nan
>>> from sympy.abc import x
>>> Heaviside(9)
1
>>> Heaviside(-9)
0
>>> Heaviside(0)
1/2
>>> Heaviside(0, nan)
nan
>>> (Heaviside(x) + 1).replace(Heaviside(x), Heaviside(x, 1))
Heaviside(x, 1) + 1
```

See also:

[*DiracDelta*](#) (page 450)

References

[R305], [R306]

classmethod `eval(arg, H0=1 / 2)`

Returns a simplified form or a value of Heaviside depending on the argument passed by the Heaviside object.

Parameters

arg : argument passed by Heaviside object

H0 : value of Heaviside(0)

Explanation

The `eval()` method is automatically called when the `Heaviside` class is about to be instantiated and it returns either some simplified instance or the unevaluated instance depending on the argument passed. In other words, `eval()` method is not needed to be called explicitly, it is being called and evaluated once the object is called.

Examples

```
>>> from sympy import Heaviside, S
>>> from sympy.abc import x
```

```
>>> Heaviside(x)
Heaviside(x)
```

```
>>> Heaviside(19)
1
```

```
>>> Heaviside(0)
1/2
```

```
>>> Heaviside(0, 1)
1
```

```
>>> Heaviside(-5)
0
```

```
>>> Heaviside(S.NaN)
nan
```

```
>>> Heaviside(x - 100).subs(x, 5)
0
```

```
>>> Heaviside(x - 100).subs(x, 105)
1
```

fdiff(*argindex*=1)

Returns the first derivative of a Heaviside Function.

Parameters**argindex** : integer

order of derivative

Examples

```
>>> from sympy import Heaviside, diff
>>> from sympy.abc import x
```

```
>>> Heaviside(x).fdiff()
DiracDelta(x)
```

```
>>> Heaviside(x**2 - 1).fdiff()
DiracDelta(x**2 - 1)
```

```
>>> diff(Heaviside(x)).fdiff()
DiracDelta(x, 1)
```

property pargs

Args without default S.Half

Singularity Function

```
class sympy.functions.special.singularity_functions.SingularityFunction(variable,
                                                                           offset,
                                                                           expo-
                                                                           nent)
```

Singularity functions are a class of discontinuous functions.

Explanation

Singularity functions take a variable, an offset, and an exponent as arguments. These functions are represented using Macaulay brackets as:

$$\text{SingularityFunction}(x, a, n) := \langle x - a \rangle^n$$

The singularity function will automatically evaluate to $\text{Derivative}(\text{DiracDelta}(x - a), x, -n - 1)$ if $n < 0$ and $(x - a)^n \text{Heaviside}(x - a)$ if $n \geq 0$.

Examples

```
>>> from sympy import SingularityFunction, diff, Piecewise, DiracDelta, \
↳ Heaviside, Symbol
>>> from sympy.abc import x, a, n
>>> SingularityFunction(x, a, n)
SingularityFunction(x, a, n)
>>> y = Symbol('y', positive=True)
>>> n = Symbol('n', nonnegative=True)
>>> SingularityFunction(y, -10, n)
(y + 10)**n
>>> y = Symbol('y', negative=True)
>>> SingularityFunction(y, 10, n)
0
>>> SingularityFunction(x, 4, -1).subs(x, 4)
00
>>> SingularityFunction(x, 10, -2).subs(x, 10)
00
>>> SingularityFunction(4, 1, 5)
243
>>> diff(SingularityFunction(x, 1, 5) + SingularityFunction(x, 1, 4), x)
4*SingularityFunction(x, 1, 3) + 5*SingularityFunction(x, 1, 4)
>>> diff(SingularityFunction(x, 4, 0), x, 2)
SingularityFunction(x, 4, -2)
>>> SingularityFunction(x, 4, 5).rewrite(Piecewise)
Piecewise(((x - 4)**5, x > 4), (0, True))
>>> expr = SingularityFunction(x, a, n)
>>> y = Symbol('y', positive=True)
>>> n = Symbol('n', nonnegative=True)
>>> expr.subs({x: y, a: -10, n: n})
(y + 10)**n
```

The methods `rewrite(DiracDelta)`, `rewrite(Heaviside)`, and `rewrite('HeavisideDiracDelta')` returns the same output. One can use any of these methods according to their choice.

```
>>> expr = SingularityFunction(x, 4, 5) + SingularityFunction(x, -3, -1) \
↳ SingularityFunction(x, 0, -2)
>>> expr.rewrite(Heaviside)
(x - 4)**5*Heaviside(x - 4) + DiracDelta(x + 3) - DiracDelta(x, 1)
>>> expr.rewrite(DiracDelta)
(x - 4)**5*Heaviside(x - 4) + DiracDelta(x + 3) - DiracDelta(x, 1)
>>> expr.rewrite('HeavisideDiracDelta')
(x - 4)**5*Heaviside(x - 4) + DiracDelta(x + 3) - DiracDelta(x, 1)
```

See also:

[DiracDelta](#) (page 450), [Heaviside](#) (page 454)

References

[R307]

classmethod `eval(variable, offset, exponent)`

Returns a simplified form or a value of Singularity Function depending on the argument passed by the object.

Explanation

The `eval()` method is automatically called when the `SingularityFunction` class is about to be instantiated and it returns either some simplified instance or the unevaluated instance depending on the argument passed. In other words, `eval()` method is not needed to be called explicitly, it is being called and evaluated once the object is called.

Examples

```
>>> from sympy import SingularityFunction, Symbol, nan
>>> from sympy.abc import x, a, n
>>> SingularityFunction(x, a, n)
SingularityFunction(x, a, n)
>>> SingularityFunction(5, 3, 2)
4
>>> SingularityFunction(x, a, nan)
nan
>>> SingularityFunction(x, 3, 0).subs(x, 3)
1
>>> SingularityFunction(4, 1, 5)
243
>>> x = Symbol('x', positive = True)
>>> a = Symbol('a', negative = True)
>>> n = Symbol('n', nonnegative = True)
>>> SingularityFunction(x, a, n)
(-a + x)**n
>>> x = Symbol('x', negative = True)
>>> a = Symbol('a', positive = True)
>>> SingularityFunction(x, a, n)
0
```

fdiff(argindex=1)

Returns the first derivative of a DiracDelta Function.

Explanation

The difference between `diff()` and `fdiff()` is: `diff()` is the user-level function and `fdiff()` is an object method. `fdiff()` is a convenience method available in the `Function` class. It returns the derivative of the function without considering the chain rule. `diff(function, x)` calls `Function._eval_derivative` which in turn calls `fdiff()` internally to compute the derivative of the function.

Gamma, Beta and related Functions

class `sympy.functions.special.gamma_functions.gamma(arg)`

The gamma function

$$\Gamma(x) := \int_0^{\infty} t^{x-1} e^{-t} dt.$$

Explanation

The gamma function implements the function which passes through the values of the factorial function (i.e., $\Gamma(n) = (n-1)!$ when n is an integer). More generally, $\Gamma(z)$ is defined in the whole complex plane except at the negative integers where there are simple poles.

Examples

```
>>> from sympy import S, I, pi, gamma
>>> from sympy.abc import x
```

Several special values are known:

```
>>> gamma(1)
1
>>> gamma(4)
6
>>> gamma(S(3)/2)
sqrt(pi)/2
```

The gamma function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(gamma(x))
gamma(conjugate(x))
```

Differentiation with respect to x is supported:

```
>>> from sympy import diff
>>> diff(gamma(x), x)
gamma(x)*polygamma(0, x)
```

Series expansion is also supported:

```
>>> from sympy import series
>>> series(gamma(x), x, 0, 3)
1/x - EulerGamma + x*(EulerGamma**2/2 + pi**2/12) + x**2*(-
→ EulerGamma*pi**2/12 + polygamma(2, 1)/6 - EulerGamma**3/6) + O(x**3)
```

We can numerically evaluate the gamma function to arbitrary precision on the whole complex plane:

```
>>> gamma(pi).evalf(40)
2.288037795340032417959588909060233922890
>>> gamma(1+I).evalf(20)
0.49801566811835604271 - 0.15494982830181068512*I
```

See also:

[lowergamma](#) (page 468)

Lower incomplete gamma function.

[uppergamma](#) (page 466)

Upper incomplete gamma function.

[polygamma](#) (page 462)

Polygamma function.

[loggamma](#) (page 460)

Log Gamma function.

[digamma](#) (page 464)

Digamma function.

[trigamma](#) (page 465)

Trigamma function.

[beta](#) (page 470)

Euler Beta function.

References

[R308], [R309], [R310], [R311]

class sympy.functions.special.gamma_functions.**loggamma**(*z*)

The loggamma function implements the logarithm of the gamma function (i.e., $\log \Gamma(x)$).

Examples

Several special values are known. For numerical integral arguments we have:

```
>>> from sympy import loggamma
>>> loggamma(-2)
00
>>> loggamma(0)
00
>>> loggamma(1)
0
```

(continues on next page)

(continued from previous page)

```
>>> loggamma(2)
0
>>> loggamma(3)
log(2)
```

And for symbolic values:

```
>>> from sympy import Symbol
>>> n = Symbol("n", integer=True, positive=True)
>>> loggamma(n)
log(gamma(n))
>>> loggamma(-n)
oo
```

For half-integral values:

```
>>> from sympy import S
>>> loggamma(S(5)/2)
log(3*sqrt(pi)/4)
>>> loggamma(n/2)
log(2**(1 - n)*sqrt(pi)*gamma(n)/gamma(n/2 + 1/2))
```

And general rational arguments:

```
>>> from sympy import expand_func
>>> L = loggamma(S(16)/3)
>>> expand_func(L).doit()
-5*log(3) + loggamma(1/3) + log(4) + log(7) + log(10) + log(13)
>>> L = loggamma(S(19)/4)
>>> expand_func(L).doit()
-4*log(4) + loggamma(3/4) + log(3) + log(7) + log(11) + log(15)
>>> L = loggamma(S(23)/7)
>>> expand_func(L).doit()
-3*log(7) + log(2) + loggamma(2/7) + log(9) + log(16)
```

The loggamma function has the following limits towards infinity:

```
>>> from sympy import oo
>>> loggamma(oo)
oo
>>> loggamma(-oo)
zoo
```

The loggamma function obeys the mirror symmetry if $x \in \mathbb{C} \setminus \{-\infty, 0\}$:

```
>>> from sympy.abc import x
>>> from sympy import conjugate
>>> conjugate(loggamma(x))
loggamma(conjugate(x))
```

Differentiation with respect to x is supported:

```
>>> from sympy import diff
>>> diff(loggamma(x), x)
polygamma(0, x)
```

Series expansion is also supported:

```
>>> from sympy import series
>>> series(loggamma(x), x, 0, 4).cancel()
-log(x) - EulerGamma*x + pi**2*x**2/12 + x**3*polygamma(2, 1)/6 + O(x**4)
```

We can numerically evaluate the gamma function to arbitrary precision on the whole complex plane:

```
>>> from sympy import I
>>> loggamma(5).evalf(30)
3.17805383034794561964694160130
>>> loggamma(I).evalf(20)
-0.65092319930185633889 - 1.8724366472624298171*I
```

See also:

gamma (page 459)

Gamma function.

lowergamma (page 468)

Lower incomplete gamma function.

uppergamma (page 466)

Upper incomplete gamma function.

polygamma (page 462)

Polygamma function.

digamma (page 464)

Digamma function.

trigamma (page 465)

Trigamma function.

beta (page 470)

Euler Beta function.

References

[R312], [R313], [R314], [R315]

class sympy.functions.special.gamma_functions.**polygamma**(*n*, *z*)

The function `polygamma(n, z)` returns `log(gamma(z)).diff(n + 1)`.

Explanation

It is a meromorphic function on $\mathbb{C}$ and defined as the $(n+1)$ -th derivative of the logarithm of the gamma function:

$$\psi^{(n)}(z) := \frac{d^{n+1}}{dz^{n+1}} \log \Gamma(z).$$

Examples

Several special values are known:

```
>>> from sympy import S, polygamma
>>> polygamma(0, 1)
-EulerGamma
>>> polygamma(0, 1/S(2))
-2*log(2) - EulerGamma
>>> polygamma(0, 1/S(3))
-log(3) - sqrt(3)*pi/6 - EulerGamma - log(sqrt(3))
>>> polygamma(0, 1/S(4))
-pi/2 - log(4) - log(2) - EulerGamma
>>> polygamma(0, 2)
1 - EulerGamma
>>> polygamma(0, 23)
19093197/5173168 - EulerGamma
```

```
>>> from sympy import oo, I
>>> polygamma(0, oo)
oo
>>> polygamma(0, -oo)
oo
>>> polygamma(0, I*oo)
oo
>>> polygamma(0, -I*oo)
oo
```

Differentiation with respect to x is supported:

```
>>> from sympy import Symbol, diff
>>> x = Symbol("x")
>>> diff(polygamma(0, x), x)
polygamma(1, x)
>>> diff(polygamma(0, x), x, 2)
polygamma(2, x)
>>> diff(polygamma(0, x), x, 3)
polygamma(3, x)
>>> diff(polygamma(1, x), x)
polygamma(2, x)
>>> diff(polygamma(1, x), x, 2)
polygamma(3, x)
>>> diff(polygamma(2, x), x)
polygamma(3, x)
```

(continues on next page)

(continued from previous page)

```
>>> diff(polygamma(2, x), x, 2)
polygamma(4, x)
```

```
>>> n = Symbol("n")
>>> diff(polygamma(n, x), x)
polygamma(n + 1, x)
>>> diff(polygamma(n, x), x, 2)
polygamma(n + 2, x)
```

We can rewrite polygamma functions in terms of harmonic numbers:

```
>>> from sympy import harmonic
>>> polygamma(0, x).rewrite(harmonic)
harmonic(x - 1) - EulerGamma
>>> polygamma(2, x).rewrite(harmonic)
2*harmonic(x - 1, 3) - 2*zeta(3)
>>> ni = Symbol("n", integer=True)
>>> polygamma(ni, x).rewrite(harmonic)
(-1)**(n + 1)*(-harmonic(x - 1, n + 1) + zeta(n + 1))*factorial(n)
```

See also:

[gamma](#) (page 459)

Gamma function.

[lowergamma](#) (page 468)

Lower incomplete gamma function.

[uppergamma](#) (page 466)

Upper incomplete gamma function.

[loggamma](#) (page 460)

Log Gamma function.

[digamma](#) (page 464)

Digamma function.

[trigamma](#) (page 465)

Trigamma function.

[beta](#) (page 470)

Euler Beta function.

References

[R316], [R317], [R318], [R319]

class sympy.functions.special.gamma_functions.**digamma**(z)

The digamma function is the first derivative of the loggamma function

$$\psi(x) := \frac{d}{dz} \log \Gamma(z) = \frac{\Gamma'(z)}{\Gamma(z)}.$$

In this case, `digamma(z) = polygamma(0, z)`.

Examples

```
>>> from sympy import digamma
>>> digamma(0)
zoo
>>> from sympy import Symbol
>>> z = Symbol('z')
>>> digamma(z)
polygamma(0, z)
```

To retain digamma as it is:

```
>>> digamma(0, evaluate=False)
digamma(0)
>>> digamma(z, evaluate=False)
digamma(z)
```

See also:

gamma (page 459)

Gamma function.

lowergamma (page 468)

Lower incomplete gamma function.

uppergamma (page 466)

Upper incomplete gamma function.

polygamma (page 462)

Polygamma function.

loggamma (page 460)

Log Gamma function.

trigamma (page 465)

Trigamma function.

beta (page 470)

Euler Beta function.

References

[R320], [R321], [R322]

class sympy.functions.special.gamma_functions.**trigamma**(z)

The trigamma function is the second derivative of the loggamma function

$$\psi^{(1)}(z) := \frac{d^2}{dz^2} \log \Gamma(z).$$

In this case, `trigamma(z) = polygamma(1, z)`.

Examples

```
>>> from sympy import trigamma
>>> trigamma(0)
zoo
>>> from sympy import Symbol
>>> z = Symbol('z')
>>> trigamma(z)
polygamma(1, z)
```

To retain trigamma as it is:

```
>>> trigamma(0, evaluate=False)
trigamma(0)
>>> trigamma(z, evaluate=False)
trigamma(z)
```

See also:

gamma (page 459)

Gamma function.

lowergamma (page 468)

Lower incomplete gamma function.

uppergamma (page 466)

Upper incomplete gamma function.

polygamma (page 462)

Polygamma function.

loggamma (page 460)

Log Gamma function.

digamma (page 464)

Digamma function.

beta (page 470)

Euler Beta function.

References

[R323], [R324], [R325]

class sympy.functions.special.gamma_functions.uppergamma(*a*, *z*)

The upper incomplete gamma function.

Explanation

It can be defined as the meromorphic continuation of

$$\Gamma(s, x) := \int_x^\infty t^{s-1} e^{-t} dt = \Gamma(s) - \gamma(s, x).$$

where $\gamma(s, x)$ is the lower incomplete gamma function, [lowergamma](#) (page 468). This can be shown to be the same as

$$\Gamma(s, x) = \Gamma(s) - \frac{x^s}{s} {}_1F_1\left(\begin{matrix} s \\ s+1 \end{matrix} \middle| -x\right),$$

where ${}_1F_1$ is the (confluent) hypergeometric function.

The upper incomplete gamma function is also essentially equivalent to the generalized exponential integral:

$$E_n(x) = \int_1^\infty \frac{e^{-xt}}{t^n} dt = x^{n-1} \Gamma(1-n, x).$$

Examples

```
>>> from sympy import uppergamma, S
>>> from sympy.abc import s, x
>>> uppergamma(s, x)
uppergamma(s, x)
>>> uppergamma(3, x)
2*(x**2/2 + x + 1)*exp(-x)
>>> uppergamma(-S(1)/2, x)
-2*sqrt(pi)*erfc(sqrt(x)) + 2*exp(-x)/sqrt(x)
>>> uppergamma(-2, x)
expint(3, x)/x**2
```

See also:

[gamma](#) (page 459)

Gamma function.

[lowergamma](#) (page 468)

Lower incomplete gamma function.

[polygamma](#) (page 462)

Polygamma function.

[loggamma](#) (page 460)

Log Gamma function.

[digamma](#) (page 464)

Digamma function.

[trigamma](#) (page 465)

Trigamma function.

[beta](#) (page 470)

Euler Beta function.

References

[R326], [R327], [R328], [R329], [R330], [R331]

class sympy.functions.special.gamma_functions.**lowergamma**(*a*, *x*)

The lower incomplete gamma function.

Explanation

It can be defined as the meromorphic continuation of

$$\gamma(s, x) := \int_0^x t^{s-1} e^{-t} dt = \Gamma(s) - \Gamma(s, x).$$

This can be shown to be the same as

$$\gamma(s, x) = \frac{x^s}{s} {}_1F_1 \left(\begin{matrix} s \\ s+1 \end{matrix} \middle| -x \right),$$

where ${}_1F_1$ is the (confluent) hypergeometric function.

Examples

```
>>> from sympy import lowergamma, S
>>> from sympy.abc import s, x
>>> lowergamma(s, x)
lowergamma(s, x)
>>> lowergamma(3, x)
-2*(x**2/2 + x + 1)*exp(-x) + 2
>>> lowergamma(-S(1)/2, x)
-2*sqrt(pi)*erf(sqrt(x)) - 2*exp(-x)/sqrt(x)
```

See also:

[gamma](#) (page 459)

Gamma function.

[uppergamma](#) (page 466)

Upper incomplete gamma function.

[polygamma](#) (page 462)

Polygamma function.

[loggamma](#) (page 460)

Log Gamma function.

[digamma](#) (page 464)

Digamma function.

[trigamma](#) (page 465)

Trigamma function.

[beta](#) (page 470)

Euler Beta function.

References

[R332], [R333], [R334], [R335], [R336]

class sympy.functions.special.gamma_functions.multigamma(x, p)

The multivariate gamma function is a generalization of the gamma function

$$\Gamma_p(z) = \pi^{p(p-1)/4} \prod_{k=1}^p \Gamma[z + (1 - k)/2].$$

In a special case, `multigamma(x, 1) = gamma(x)`.

Parameters

p : order or dimension of the multivariate gamma function

Examples

```
>>> from sympy import S, multigamma
>>> from sympy import Symbol
>>> x = Symbol('x')
>>> p = Symbol('p', positive=True, integer=True)
```

```
>>> multigamma(x, p)
pi**(p*(p - 1)/4)*Product(gamma(-_k/2 + x + 1/2), (_k, 1, p))
```

Several special values are known:

```
>>> multigamma(1, 1)
1
>>> multigamma(4, 1)
6
>>> multigamma(S(3)/2, 1)
sqrt(pi)/2
```

Writing multigamma in terms of the gamma function:

```
>>> multigamma(x, 1)
gamma(x)
```

```
>>> multigamma(x, 2)
sqrt(pi)*gamma(x)*gamma(x - 1/2)
```

```
>>> multigamma(x, 3)
pi**(3/2)*gamma(x)*gamma(x - 1)*gamma(x - 1/2)
```

See also:

[gamma](#) (page 459), [lowergamma](#) (page 468), [uppergamma](#) (page 466), [polygamma](#) (page 462), [loggamma](#) (page 460), [digamma](#) (page 464), [trigamma](#) (page 465), [beta](#) (page 470)

References

[R337]

class sympy.functions.special.beta_functions.**beta**(x , $y=None$)

The beta integral is called the Eulerian integral of the first kind by Legendre:

$$B(x, y) = \int_0^1 t^{x-1} (1-t)^{y-1} dt.$$

Explanation

The Beta function or Euler's first integral is closely associated with the gamma function. The Beta function is often used in probability theory and mathematical statistics. It satisfies properties like:

$$\begin{aligned} B(a, 1) &= \frac{1}{a} \\ B(a, b) &= B(b, a) \\ B(a, b) &= \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)} \end{aligned}$$

Therefore for integral values of a and b :

$$B = \frac{(a-1)!(b-1)!}{(a+b-1)!}$$

A special case of the Beta function when $x = y$ is the Central Beta function. It satisfies properties like:

$$B(x) = 2^{1-2x} B(x, \frac{1}{2}) B(x) = 2^{1-2x} \cos(\pi x) B(\frac{1}{2} - x, x) B(x) = \int_0^1 \frac{t^x}{(1+t)^{2x}} dt B(x) = \frac{2}{x} \prod_{n=1}^{\infty} \frac{n(n+2x)}{(n+x)^2}$$

Examples

```
>>> from sympy import I, pi
>>> from sympy.abc import x, y
```

The Beta function obeys the mirror symmetry:

```
>>> from sympy import beta, conjugate
>>> conjugate(beta(x, y))
beta(conjugate(x), conjugate(y))
```

Differentiation with respect to both x and y is supported:

```
>>> from sympy import beta, diff
>>> diff(beta(x, y), x)
(polygamma(0, x) - polygamma(0, x + y))*beta(x, y)
```

```
>>> diff(beta(x, y), y)
(polygamma(0, y) - polygamma(0, x + y))*beta(x, y)
```

```
>>> diff(beta(x), x)
2*(polygamma(0, x) - polygamma(0, 2*x))*beta(x, x)
```

We can numerically evaluate the Beta function to arbitrary precision for any complex numbers x and y :

```
>>> from sympy import beta
>>> beta(pi).evalf(40)
0.02671848900111377452242355235388489324562
```

```
>>> beta(1 + I).evalf(20)
-0.2112723729365330143 - 0.7655283165378005676*I
```

See also:

gamma (page 459)

Gamma function.

uppergamma (page 466)

Upper incomplete gamma function.

lowergamma (page 468)

Lower incomplete gamma function.

polygamma (page 462)

Polygamma function.

loggamma (page 460)

Log Gamma function.

digamma (page 464)

Digamma function.

trigamma (page 465)

Trigamma function.

References

[R338], [R339], [R340]

Error Functions and Fresnel Integrals

class sympy.functions.special.error_functions.**erf**(*arg*)

The Gauss error function.

Explanation

This function is defined as:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

Examples

```
>>> from sympy import I, oo, erf
>>> from sympy.abc import z
```

Several special values are known:

```
>>> erf(0)
0
>>> erf(oo)
1
>>> erf(-oo)
-1
>>> erf(I*oo)
oo*I
>>> erf(-I*oo)
-oo*I
```

In general one can pull out factors of -1 and I from the argument:

```
>>> erf(-z)
-erf(z)
```

The error function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(erf(z))
erf(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(erf(z), z)
2*exp(-z**2)/sqrt(pi)
```

We can numerically evaluate the error function to arbitrary precision on the whole complex plane:

```
>>> erf(4).evalf(30)
0.999999984582742099719981147840
```

```
>>> erf(-4*I).evalf(30)
-1296959.73071763923152794095062*I
```

See also:

erfc (page 473)

Complementary error function.

erfi (page 474)

Imaginary error function.

erf2 (page 476)

Two-argument error function.

erfinv (page 477)

Inverse error function.

erfcinv (page 478)

Inverse Complementary error function.

erf2inv (page 479)

Inverse two-argument error function.

References[\[R341\]](#), [\[R342\]](#), [\[R343\]](#), [\[R344\]](#)**inverse(argindex=1)**

Returns the inverse of this function.

class sympy.functions.special.error_functions.**erfc**(arg)

Complementary Error Function.

Explanation

The function is defined as:

$$\operatorname{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2} dt$$

Examples

```
>>> from sympy import I, oo, erfc
>>> from sympy.abc import z
```

Several special values are known:

```
>>> erfc(0)
1
>>> erfc(oo)
0
>>> erfc(-oo)
2
>>> erfc(I*oo)
-oo*I
>>> erfc(-I*oo)
oo*I
```

The error function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(erfc(z))
erfc(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(erfc(z), z)
-2*exp(-z**2)/sqrt(pi)
```

It also follows

```
>>> erfc(-z)
2 - erfc(z)
```

We can numerically evaluate the complementary error function to arbitrary precision on the whole complex plane:

```
>>> erfc(4).evalf(30)
0.0000000154172579002800188521596734869
```

```
>>> erfc(4*I).evalf(30)
1.0 - 1296959.73071763923152794095062*I
```

See also:

erf (page 471)

Gaussian error function.

erfi (page 474)

Imaginary error function.

erf2 (page 476)

Two-argument error function.

erfinv (page 477)

Inverse error function.

erfcinv (page 478)

Inverse Complementary error function.

erf2inv (page 479)

Inverse two-argument error function.

References

[R345], [R346], [R347], [R348]

inverse(*argindex*=1)

Returns the inverse of this function.

class sympy.functions.special.error_functions.***erfi***(*z*)

Imaginary error function.

Explanation

The function `erfi` is defined as:

$$\operatorname{erfi}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{t^2} dt$$

Examples

```
>>> from sympy import I, oo, erfi
>>> from sympy.abc import z
```

Several special values are known:

```
>>> erfi(0)
0
>>> erfi(oo)
oo
>>> erfi(-oo)
-oo
>>> erfi(I*oo)
I
>>> erfi(-I*oo)
-I
```

In general one can pull out factors of `-1` and `I` from the argument:

```
>>> erfi(-z)
-erfi(z)
```

```
>>> from sympy import conjugate
>>> conjugate(erfi(z))
erfi(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(erfi(z), z)
2*exp(z**2)/sqrt(pi)
```

We can numerically evaluate the imaginary error function to arbitrary precision on the whole complex plane:

```
>>> erfi(2).evalf(30)
18.5648024145755525987042919132
```

```
>>> erfi(-2*I).evalf(30)
-0.995322265018952734162069256367*I
```

See also:

[**`erf`**](#) (page 471)

Gaussian error function.

erfc (page 473)

Complementary error function.

erf2 (page 476)

Two-argument error function.

erfinv (page 477)

Inverse error function.

erfcinv (page 478)

Inverse Complementary error function.

erf2inv (page 479)

Inverse two-argument error function.

References

[R349], [R350], [R351]

class sympy.functions.special.error_functions.erf2(x, y)

Two-argument error function.

Explanation

This function is defined as:

$$\operatorname{erf2}(x, y) = \frac{2}{\sqrt{\pi}} \int_x^y e^{-t^2} dt$$

Examples

```
>>> from sympy import oo, erf2
>>> from sympy.abc import x, y
```

Several special values are known:

```
>>> erf2(0, 0)
0
>>> erf2(x, x)
0
>>> erf2(x, oo)
1 - erf(x)
>>> erf2(x, -oo)
-erf(x) - 1
>>> erf2(oo, y)
erf(y) - 1
>>> erf2(-oo, y)
erf(y) + 1
```

In general one can pull out factors of -1:

```
>>> erf2(-x, -y)
-erf2(x, y)
```

The error function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(erf2(x, y))
erf2(conjugate(x), conjugate(y))
```

Differentiation with respect to x, y is supported:

```
>>> from sympy import diff
>>> diff(erf2(x, y), x)
-2*exp(-x**2)/sqrt(pi)
>>> diff(erf2(x, y), y)
2*exp(-y**2)/sqrt(pi)
```

See also:

[erf](#) (page 471)

Gaussian error function.

[erfc](#) (page 473)

Complementary error function.

[erfi](#) (page 474)

Imaginary error function.

[erfinv](#) (page 477)

Inverse error function.

[erfcinv](#) (page 478)

Inverse Complementary error function.

[erf2inv](#) (page 479)

Inverse two-argument error function.

References

[R352]

class sympy.functions.special.error_functions.**erfinv**(z)

Inverse Error Function. The erfinv function is defined as:

$$\operatorname{erf}(x) = y \quad \Rightarrow \quad \operatorname{erfinv}(y) = x$$

Examples

```
>>> from sympy import erfinv
>>> from sympy.abc import x
```

Several special values are known:

```
>>> erfinv(0)
0
>>> erfinv(1)
oo
```

Differentiation with respect to x is supported:

```
>>> from sympy import diff
>>> diff(erfinv(x), x)
sqrt(pi)*exp(erfinv(x)**2)/2
```

We can numerically evaluate the inverse error function to arbitrary precision on $[-1, 1]$:

```
>>> erfinv(0.2).evalf(30)
0.179143454621291692285822705344
```

See also:

[erf](#) (page 471)

Gaussian error function.

[erfc](#) (page 473)

Complementary error function.

[erfi](#) (page 474)

Imaginary error function.

[erf2](#) (page 476)

Two-argument error function.

[erfcinv](#) (page 478)

Inverse Complementary error function.

[erf2inv](#) (page 479)

Inverse two-argument error function.

References

[R353], [R354]

`inverse(argindex=1)`

Returns the inverse of this function.

`class sympy.functions.special.error_functions.erfcinv(z)`

Inverse Complementary Error Function. The `erfcinv` function is defined as:

$$\operatorname{erfc}(x) = y \quad \Rightarrow \quad \operatorname{erfcinv}(y) = x$$

Examples

```
>>> from sympy import erfcinv
>>> from sympy.abc import x
```

Several special values are known:

```
>>> erfcinv(1)
0
>>> erfcinv(0)
oo
```

Differentiation with respect to x is supported:

```
>>> from sympy import diff
>>> diff(erfcinv(x), x)
-sqrt(pi)*exp(erfcinv(x)**2)/2
```

See also:

[erf](#) (page 471)

Gaussian error function.

[erfc](#) (page 473)

Complementary error function.

[erfi](#) (page 474)

Imaginary error function.

[erf2](#) (page 476)

Two-argument error function.

[erfinv](#) (page 477)

Inverse error function.

[erf2inv](#) (page 479)

Inverse two-argument error function.

References

[R355], [R356]

`inverse(argindex=1)`

Returns the inverse of this function.

`class sympy.functions.special.error_functions.erf2inv(x, y)`

Two-argument Inverse error function. The erf2inv function is defined as:

$$\operatorname{erf2}(x, w) = y \quad \Rightarrow \quad \operatorname{erf2inv}(x, y) = w$$

Examples

```
>>> from sympy import erf2inv, oo
>>> from sympy.abc import x, y
```

Several special values are known:

```
>>> erf2inv(0, 0)
0
>>> erf2inv(1, 0)
1
>>> erf2inv(0, 1)
oo
>>> erf2inv(0, y)
erfinv(y)
>>> erf2inv(oo, y)
erfcinv(-y)
```

Differentiation with respect to x and y is supported:

```
>>> from sympy import diff
>>> diff(erf2inv(x, y), x)
exp(-x**2 + erf2inv(x, y)**2)
>>> diff(erf2inv(x, y), y)
sqrt(pi)*exp(erf2inv(x, y)**2)/2
```

See also:

[erf](#) (page 471)

Gaussian error function.

[erfc](#) (page 473)

Complementary error function.

[erfi](#) (page 474)

Imaginary error function.

[erf2](#) (page 476)

Two-argument error function.

[erfinv](#) (page 477)

Inverse error function.

[erfcinv](#) (page 478)

Inverse complementary error function.

References

[R357]

class sympy.functions.special.error_functions.**FresnelIntegral**(z)

Base class for the Fresnel integrals.

class sympy.functions.special.error_functions.**fresnels**(z)

Fresnel integral S.

Explanation

This function is defined by

$$S(z) = \int_0^z \sin \frac{\pi}{2} t^2 dt.$$

It is an entire function.

Examples

```
>>> from sympy import I, oo, fresnels
>>> from sympy.abc import z
```

Several special values are known:

```
>>> fresnels(0)
0
>>> fresnels(oo)
1/2
>>> fresnels(-oo)
-1/2
>>> fresnels(I*oo)
-I/2
>>> fresnels(-I*oo)
I/2
```

In general one can pull out factors of -1 and i from the argument:

```
>>> fresnels(-z)
-fresnels(z)
>>> fresnels(I*z)
-I*fresnels(z)
```

The Fresnel S integral obeys the mirror symmetry $\overline{S(z)} = S(\bar{z})$:

```
>>> from sympy import conjugate
>>> conjugate(fresnels(z))
fresnels(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(fresnels(z), z)
sin(pi*z**2/2)
```

Defining the Fresnel functions via an integral:

```
>>> from sympy import integrate, pi, sin, expand_func
>>> integrate(sin(pi*z**2/2), z)
3*fresnels(z)*gamma(3/4)/(4*gamma(7/4))
>>> expand_func(integrate(sin(pi*z**2/2), z))
fresnels(z)
```

We can numerically evaluate the Fresnel integral to arbitrary precision on the whole complex plane:

```
>>> fresnels(2).evalf(30)
0.343415678363698242195300815958
```

```
>>> fresnels(-2*I).evalf(30)
0.343415678363698242195300815958*I
```

See also:

fresnelc (page 482)

Fresnel cosine integral.

References[\[R358\]](#), [\[R359\]](#), [\[R360\]](#), [\[R361\]](#), [\[R362\]](#)**class** sympy.functions.special.error_functions.fresnelc(*z*)

Fresnel integral C.

Explanation

This function is defined by

$$C(z) = \int_0^z \cos \frac{\pi}{2} t^2 dt.$$

It is an entire function.

Examples

```
>>> from sympy import I, oo, fresnelc
>>> from sympy.abc import z
```

Several special values are known:

```
>>> fresnelc(0)
0
>>> fresnelc(oo)
1/2
>>> fresnelc(-oo)
-1/2
>>> fresnelc(I*oo)
I/2
>>> fresnelc(-I*oo)
-I/2
```

In general one can pull out factors of -1 and *i* from the argument:

```
>>> fresnelc(-z)
-fresnelc(z)
>>> fresnelc(I*z)
I*fresnelc(z)
```

The Fresnel C integral obeys the mirror symmetry $\overline{C(z)} = C(\bar{z})$:

```
>>> from sympy import conjugate
>>> conjugate(fresnelc(z))
fresnelc(conjugate(z))
```

Differentiation with respect to *z* is supported:

```
>>> from sympy import diff
>>> diff(fresnelc(z), z)
cos(pi*z**2/2)
```

Defining the Fresnel functions via an integral:

```
>>> from sympy import integrate, pi, cos, expand_func
>>> integrate(cos(pi*z**2/2), z)
fresnelc(z)*gamma(1/4)/(4*gamma(5/4))
>>> expand_func(integrate(cos(pi*z**2/2), z))
fresnelc(z)
```

We can numerically evaluate the Fresnel integral to arbitrary precision on the whole complex plane:

```
>>> fresnelc(2).evalf(30)
0.488253406075340754500223503357
```

```
>>> fresnelc(-2*I).evalf(30)
-0.488253406075340754500223503357*I
```

See also:

[fresnels](#) (page 480)
Fresnel sine integral.

References

[R363], [R364], [R365], [R366], [R367]

Exponential, Logarithmic and Trigonometric Integrals

class sympy.functions.special.error_functions.**Ei**(z)
The classical exponential integral.

Explanation

For use in SymPy, this function is defined as

$$\operatorname{Ei}(x) = \sum_{n=1}^{\infty} \frac{x^n}{n n!} + \log(x) + \gamma,$$

where γ is the Euler-Mascheroni constant.

If x is a polar number, this defines an analytic function on the Riemann surface of the logarithm. Otherwise this defines an analytic function in the cut plane $\mathbb{C} \setminus (-\infty, 0]$.

Background

The name exponential integral comes from the following statement:

$$\operatorname{Ei}(x) = \int_{-\infty}^x \frac{e^t}{t} dt$$

If the integral is interpreted as a Cauchy principal value, this statement holds for $x > 0$ and $Ei(x)$ as defined above.

Examples

```
>>> from sympy import Ei, polar_lift, exp_polar, I, pi
>>> from sympy.abc import x
```

```
>>> Ei(-1)
Ei(-1)
```

This yields a real value:

```
>>> Ei(-1).n(chop=True)
-0.219383934395520
```

On the other hand the analytic continuation is not real:

```
>>> Ei(polar_lift(-1)).n(chop=True)
-0.21938393439552 + 3.14159265358979*I
```

The exponential integral has a logarithmic branch point at the origin:

```
>>> Ei(x*exp_polar(2*I*pi))
Ei(x) + 2*I*pi
```

Differentiation is supported:

```
>>> Ei(x).diff(x)
exp(x)/x
```

The exponential integral is related to many other special functions. For example:

```
>>> from sympy import expint, Shi
>>> Ei(x).rewrite(expint)
-expint(1, x*exp_polar(I*pi)) - I*pi
>>> Ei(x).rewrite(Shi)
Chi(x) + Shi(x)
```

See also:

[expint](#) (page 485)

Generalised exponential integral.

[E1](#) (page 487)

Special case of the generalised exponential integral.

[li](#) (page 487)

Logarithmic integral.

[Li](#) (page 489)

Offset logarithmic integral.

[Si](#) (page 491)

Sine integral.

***Ci* (page 492)**

Cosine integral.

***Shi* (page 494)**

Hyperbolic sine integral.

***Chi* (page 495)**

Hyperbolic cosine integral.

***uppergamma* (page 466)**

Upper incomplete gamma function.

References[\[R368\]](#), [\[R369\]](#), [\[R370\]](#)**class** `sympy.functions.special.error_functions.expint(nu, z)`

Generalized exponential integral.

Explanation

This function is defined as

$$E_{\nu}(z) = z^{\nu-1} \Gamma(1 - \nu, z),$$

where $\Gamma(1 - \nu, z)$ is the upper incomplete gamma function (`uppergamma`).Hence for z with positive real part we have

$$E_{\nu}(z) = \int_1^{\infty} \frac{e^{-zt}}{t^{\nu}} dt,$$

which explains the name.

The representation as an incomplete gamma function provides an analytic continuation for $E_{\nu}(z)$. If ν is a non-positive integer, the exponential integral is thus an unbranched function of z , otherwise there is a branch point at the origin. Refer to the incomplete gamma function documentation for details of the branching behavior.

Examples

```
>>> from sympy import expint, S
>>> from sympy.abc import nu, z
```

Differentiation is supported. Differentiation with respect to z further explains the name: for integral orders, the exponential integral is an iterated integral of the exponential function.

```
>>> expint(nu, z).diff(z)
-expint(nu - 1, z)
```

Differentiation with respect to ν has no classical expression:

```
>>> expint(nu, z).diff(nu)
-z**(nu - 1)*meijerg((()), (1, 1)), ((0, 0, 1 - nu), ()), z)
```

At non-positive integer orders, the exponential integral reduces to the exponential function:

```
>>> expint(0, z)
exp(-z)/z
>>> expint(-1, z)
exp(-z)/z + exp(-z)/z**2
```

At half-integers it reduces to error functions:

```
>>> expint(S(1)/2, z)
sqrt(pi)*erfc(sqrt(z))/sqrt(z)
```

At positive integer orders it can be rewritten in terms of exponentials and `expint(1, z)`. Use `expand_func()` to do this:

```
>>> from sympy import expand_func
>>> expand_func(expint(5, z))
z**4*expint(1, z)/24 + (-z**3 + z**2 - 2*z + 6)*exp(-z)/24
```

The generalised exponential integral is essentially equivalent to the incomplete gamma function:

```
>>> from sympy import uppgamma
>>> expint(nu, z).rewrite(uppgamma)
z**(nu - 1)*uppgamma(1 - nu, z)
```

As such it is branched at the origin:

```
>>> from sympy import exp_polar, pi, I
>>> expint(4, z*exp_polar(2*pi*I))
I*pi*z**3/3 + expint(4, z)
>>> expint(nu, z*exp_polar(2*pi*I))
z**(nu - 1)*(exp(2*I*pi*nu) - 1)*gamma(1 - nu) + expint(nu, z)
```

See also:

Ei (page 483)

Another related function called exponential integral.

E1 (page 487)

The classical case, returns `expint(1, z)`.

li (page 487)

Logarithmic integral.

Li (page 489)

Offset logarithmic integral.

Si (page 491)

Sine integral.

Ci (page 492)

Cosine integral.

***Shi* (page 494)**

Hyperbolic sine integral.

***Chi* (page 495)**

Hyperbolic cosine integral.

uppergamma (page 466)**References**[\[R371\]](#), [\[R372\]](#), [\[R373\]](#)`sympy.functions.special.error_functions.E1(z)`

Classical case of the generalized exponential integral.

ExplanationThis is equivalent to `expint(1, z)`.**Examples**

```
>>> from sympy import E1
>>> E1(0)
expint(1, 0)
```

```
>>> E1(5)
expint(1, 5)
```

See also:***Ei* (page 483)**

Exponential integral.

***expint* (page 485)**

Generalised exponential integral.

***li* (page 487)**

Logarithmic integral.

***Li* (page 489)**

Offset logarithmic integral.

***Si* (page 491)**

Sine integral.

***Ci* (page 492)**

Cosine integral.

***Shi* (page 494)**

Hyperbolic sine integral.

***Chi* (page 495)**

Hyperbolic cosine integral.

class sympy.functions.special.error_functions.**li**(*z*)

The classical logarithmic integral.

Explanation

For use in SymPy, this function is defined as

$$\operatorname{li}(x) = \int_0^x \frac{1}{\log(t)} dt.$$

Examples

```
>>> from sympy import I, oo, li
>>> from sympy.abc import z
```

Several special values are known:

```
>>> li(0)
0
>>> li(1)
-oo
>>> li(oo)
oo
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(li(z), z)
1/log(z)
```

Defining the `li` function via an integral: `>>> from sympy import integrate` `>>> integrate(li(z), z)` `z*li(z) - Ei(2*log(z))`

```
>>> integrate(li(z), z)
z*li(z) - Ei(2*log(z))
```

The logarithmic integral can also be defined in terms of `Ei`:

```
>>> from sympy import Ei
>>> li(z).rewrite(Ei)
Ei(log(z))
>>> diff(li(z).rewrite(Ei), z)
1/log(z)
```

We can numerically evaluate the logarithmic integral to arbitrary precision on the whole complex plane (except the singular points):

```
>>> li(2).evalf(30)
1.04516378011749278484458888919
```

```
>>> li(2*I).evalf(30)
1.0652795784357498247001125598 + 3.08346052231061726610939702133*I
```


We can even compute Soldner's constant by the help of mpmath:

```
>>> from mpmath import findroot
>>> findroot(li, 2)
1.45136923488338
```

Further transformations include rewriting `li` in terms of the trigonometric integrals `Si`, `Ci`, `Shi` and `Chi`:

```
>>> from sympy import Si, Ci, Shi, Chi
>>> li(z).rewrite(Si)
-log(I*log(z)) - log(1/log(z))/2 + log(log(z))/2 + Ci(I*log(z)) +
→Shi(log(z))
>>> li(z).rewrite(Ci)
-log(I*log(z)) - log(1/log(z))/2 + log(log(z))/2 + Ci(I*log(z)) +
→Shi(log(z))
>>> li(z).rewrite(Shi)
-log(1/log(z))/2 + log(log(z))/2 + Chi(log(z)) - Shi(log(z))
>>> li(z).rewrite(Chi)
-log(1/log(z))/2 + log(log(z))/2 + Chi(log(z)) - Shi(log(z))
```

See also:

Li (page 489)

Offset logarithmic integral.

Ei (page 483)

Exponential integral.

expint (page 485)

Generalised exponential integral.

E1 (page 487)

Special case of the generalised exponential integral.

Si (page 491)

Sine integral.

Ci (page 492)

Cosine integral.

Shi (page 494)

Hyperbolic sine integral.

Chi (page 495)

Hyperbolic cosine integral.

References

[R374], [R375], [R376], [R377]

class `sympy.functions.special.error_functions.Li(z)`

The offset logarithmic integral.

Explanation

For use in SymPy, this function is defined as

$$\text{Li}(x) = \text{li}(x) - \text{li}(2)$$

Examples

```
>>> from sympy import Li
>>> from sympy.abc import z
```

The following special value is known:

```
>>> Li(2)
0
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(Li(z), z)
1/log(z)
```

The shifted logarithmic integral can be written in terms of $\text{li}(z)$:

```
>>> from sympy import li
>>> Li(z).rewrite(li)
li(z) - li(2)
```

We can numerically evaluate the logarithmic integral to arbitrary precision on the whole complex plane (except the singular points):

```
>>> Li(2).evalf(30)
0
```

```
>>> Li(4).evalf(30)
1.92242131492155809316615998938
```

See also:

[li](#) (page 487)
Logarithmic integral.

[Ei](#) (page 483)
Exponential integral.

[expint](#) (page 485)
Generalised exponential integral.

[E1](#) (page 487)
Special case of the generalised exponential integral.

[Si](#) (page 491)
Sine integral.

[Ci](#) (page 492)
Cosine integral.

Shi (page 494)

Hyperbolic sine integral.

Chi (page 495)

Hyperbolic cosine integral.

References[\[R378\]](#), [\[R379\]](#), [\[R380\]](#)**class** `sympy.functions.special.error_functions.Si(z)`

Sine integral.

Explanation

This function is defined by

$$\text{Si}(z) = \int_0^z \frac{\sin t}{t} dt.$$

It is an entire function.

Examples

```
>>> from sympy import Si
>>> from sympy.abc import z
```

The sine integral is an antiderivative of $\sin(z)/z$:

```
>>> Si(z).diff(z)
sin(z)/z
```

It is unbranched:

```
>>> from sympy import exp_polar, I, pi
>>> Si(z*exp_polar(2*I*pi))
Si(z)
```

Sine integral behaves much like ordinary sine under multiplication by I :

```
>>> Si(I*z)
I*Shi(z)
>>> Si(-z)
-Si(z)
```

It can also be expressed in terms of exponential integrals, but beware that the latter is branched:

```
>>> from sympy import expint
>>> Si(z).rewrite(expint)
-I*(-expint(1, z*exp_polar(-I*pi/2))/2 +
    expint(1, z*exp_polar(I*pi/2))/2) + pi/2
```

It can be rewritten in the form of sinc function (by definition):

```
>>> from sympy import sinc
>>> Si(z).rewrite(sinc)
Integral(sinc(t), (t, 0, z))
```

See also:

Ci (page 492)

Cosine integral.

Shi (page 494)

Hyperbolic sine integral.

Chi (page 495)

Hyperbolic cosine integral.

Ei (page 483)

Exponential integral.

expint (page 485)

Generalised exponential integral.

sinc (page 394)

unnormalized sinc function

E1 (page 487)

Special case of the generalised exponential integral.

li (page 487)

Logarithmic integral.

Li (page 489)

Offset logarithmic integral.

References

[R381]

class sympy.functions.special.error_functions.**Ci**(*z*)

Cosine integral.

Explanation

This function is defined for positive x by

$$\text{Ci}(x) = \gamma + \log x + \int_0^x \frac{\cos t - 1}{t} dt = - \int_x^\infty \frac{\cos t}{t} dt,$$

where γ is the Euler-Mascheroni constant.

We have

$$\text{Ci}(z) = - \frac{\text{E}_1(e^{i\pi/2}z) + \text{E}_1(e^{-i\pi/2}z)}{2}$$

which holds for all polar z and thus provides an analytic continuation to the Riemann surface of the logarithm.

The formula also holds as stated for $z \in \mathbb{C}$ with $\Re(z) > 0$. By lifting to the principal branch, we obtain an analytic function on the cut complex plane.

Examples

```
>>> from sympy import Ci
>>> from sympy.abc import z
```

The cosine integral is a primitive of $\cos(z)/z$:

```
>>> Ci(z).diff(z)
cos(z)/z
```

It has a logarithmic branch point at the origin:

```
>>> from sympy import exp_polar, I, pi
>>> Ci(z*exp_polar(2*I*pi))
Ci(z) + 2*I*pi
```

The cosine integral behaves somewhat like ordinary $\cos$ under multiplication by i :

```
>>> from sympy import polar_lift
>>> Ci(polar_lift(I)*z)
Chi(z) + I*pi/2
>>> Ci(polar_lift(-1)*z)
Ci(z) + I*pi
```

It can also be expressed in terms of exponential integrals:

```
>>> from sympy import expint
>>> Ci(z).rewrite(expint)
-expint(1, z*exp_polar(-I*pi/2))/2 - expint(1, z*exp_polar(I*pi/2))/2
```

See also:

***Si* (page 491)**

Sine integral.

***Shi* (page 494)**

Hyperbolic sine integral.

***Chi* (page 495)**

Hyperbolic cosine integral.

***Ei* (page 483)**

Exponential integral.

***expint* (page 485)**

Generalised exponential integral.

***E1* (page 487)**

Special case of the generalised exponential integral.

***li* (page 487)**

Logarithmic integral.

Li (page 489)

Offset logarithmic integral.

References[\[R382\]](#)

class sympy.functions.special.error_functions.**Shi**(*z*)
Sinh integral.

Explanation

This function is defined by

$$\text{Shi}(z) = \int_0^z \frac{\sinh t}{t} dt.$$

It is an entire function.

Examples

```
>>> from sympy import Shi
>>> from sympy.abc import z
```

The Sinh integral is a primitive of $\sinh(z)/z$:

```
>>> Shi(z).diff(z)
sinh(z)/z
```

It is unbranched:

```
>>> from sympy import exp_polar, I, pi
>>> Shi(z*exp_polar(2*I*pi))
Shi(z)
```

The sinh integral behaves much like ordinary $\sinh$ under multiplication by i :

```
>>> Shi(I*z)
I*Si(z)
>>> Shi(-z)
-Shi(z)
```

It can also be expressed in terms of exponential integrals, but beware that the latter is branched:

```
>>> from sympy import expint
>>> Shi(z).rewrite(expint)
expint(1, z)/2 - expint(1, z*exp_polar(I*pi))/2 - I*pi/2
```

See also:

Si (page 491)

Sine integral.

***Ci* (page 492)**

Cosine integral.

***Chi* (page 495)**

Hyperbolic cosine integral.

***Ei* (page 483)**

Exponential integral.

***expint* (page 485)**

Generalised exponential integral.

***E1* (page 487)**

Special case of the generalised exponential integral.

***li* (page 487)**

Logarithmic integral.

***Li* (page 489)**

Offset logarithmic integral.

References[\[R383\]](#)**class** sympy.functions.special.error_functions.**Chi**(*z*)

Cosh integral.

ExplanationThis function is defined for positive x by

$$\text{Chi}(x) = \gamma + \log x + \int_0^x \frac{\cosh t - 1}{t} dt,$$

where γ is the Euler-Mascheroni constant.

We have

$$\text{Chi}(z) = \text{Ci}\left(e^{i\pi/2}z\right) - i\frac{\pi}{2},$$

which holds for all polar z and thus provides an analytic continuation to the Riemann surface of the logarithm. By lifting to the principal branch we obtain an analytic function on the cut complex plane.

Examples

```
>>> from sympy import Chi
>>> from sympy.abc import z
```

The cosh integral is a primitive of $\cosh(z)/z$:

```
>>> Chi(z).diff(z)
cosh(z)/z
```

It has a logarithmic branch point at the origin:

```
>>> from sympy import exp_polar, I, pi
>>> Chi(z*exp_polar(2*I*pi))
Chi(z) + 2*I*pi
```

The cosh integral behaves somewhat like ordinary cosh under multiplication by i :

```
>>> from sympy import polar_lift
>>> Chi(polar_lift(I)*z)
Ci(z) + I*pi/2
>>> Chi(polar_lift(-1)*z)
Chi(z) + I*pi
```

It can also be expressed in terms of exponential integrals:

```
>>> from sympy import expint
>>> Chi(z).rewrite(expint)
-expint(1, z)/2 - expint(1, z*exp_polar(I*pi))/2 - I*pi/2
```

See also:

***Si* (page 491)**

Sine integral.

***Ci* (page 492)**

Cosine integral.

***Shi* (page 494)**

Hyperbolic sine integral.

***Ei* (page 483)**

Exponential integral.

***expint* (page 485)**

Generalised exponential integral.

***E1* (page 487)**

Special case of the generalised exponential integral.

***li* (page 487)**

Logarithmic integral.

***Li* (page 489)**

Offset logarithmic integral.

References

[R384]

Bessel Type Functions

class sympy.functions.special.bessel.BesselBase(*nu*, *z*)

Abstract base class for Bessel-type functions.

This class is meant to reduce code duplication. All Bessel-type functions can 1) be differentiated, with the derivatives expressed in terms of similar functions, and 2) be rewritten in terms of other Bessel-type functions.

Here, Bessel-type functions are assumed to have one complex parameter.

To use this base class, define class attributes `_a` and `_b` such that $2F_n' = -_aF_{n+1} + _bF_{n-1}$.

property argument

The argument of the Bessel-type function.

property order

The order of the Bessel-type function.

class sympy.functions.special.bessel.besselj(*nu*, *z*)

Bessel function of the first kind.

Explanation

The Bessel J function of order ν is defined to be the function satisfying Bessel's differential equation

$$z^2 \frac{d^2 w}{dz^2} + z \frac{dw}{dz} + (z^2 - \nu^2)w = 0,$$

with Laurent expansion

$$J_\nu(z) = z^\nu \left(\frac{1}{\Gamma(\nu + 1)2^\nu} + O(z^2) \right),$$

if ν is not a negative integer. If $\nu = -n \in \mathbb{Z}_{<0}$ is a negative integer, then the definition is

$$J_{-n}(z) = (-1)^n J_n(z).$$

Examples

Create a Bessel function object:

```
>>> from sympy import besselj, jn
>>> from sympy.abc import z, n
>>> b = besselj(n, z)
```

Differentiate it:

```
>>> b.diff(z)
besselj(n - 1, z)/2 - besselj(n + 1, z)/2
```

Rewrite in terms of spherical Bessel functions:

```
>>> b.rewrite(jn)
sqrt(2)*sqrt(z)*jn(n - 1/2, z)/sqrt(pi)
```

Access the parameter and argument:

```
>>> b.order
n
>>> b.argument
z
```

See also:

[*bessely*](#) (page 498), [*besseli*](#) (page 499), [*besselk*](#) (page 499)

References

[R385], [R386], [R387], [R388]

class `sympy.functions.special.bessel.bessely(nu, z)`

Bessel function of the second kind.

Explanation

The Bessel Y function of order ν is defined as

$$Y_{\nu}(z) = \lim_{\mu \rightarrow \nu} \frac{J_{\mu}(z) \cos(\pi\mu) - J_{-\mu}(z)}{\sin(\pi\mu)},$$

where $J_{\mu}(z)$ is the Bessel function of the first kind.

It is a solution to Bessel's equation, and linearly independent from J_{ν} .

Examples

```
>>> from sympy import bessely, yn
>>> from sympy.abc import z, n
>>> b = bessely(n, z)
>>> b.diff(z)
bessely(n - 1, z)/2 - bessely(n + 1, z)/2
>>> b.rewrite(yn)
sqrt(2)*sqrt(z)*yn(n - 1/2, z)/sqrt(pi)
```

See also:

[*besselj*](#) (page 497), [*besseli*](#) (page 499), [*besselk*](#) (page 499)

References

[R389]

class sympy.functions.special.bessel.besseli(*nu*, *z*)
Modified Bessel function of the first kind.

Explanation

The Bessel I function is a solution to the modified Bessel equation

$$z^2 \frac{d^2 w}{dz^2} + z \frac{dw}{dz} + (z^2 + \nu^2)w = 0.$$

It can be defined as

$$I_\nu(z) = i^{-\nu} J_\nu(iz),$$

where $J_\nu(z)$ is the Bessel function of the first kind.

Examples

```
>>> from sympy import besseli
>>> from sympy.abc import z, n
>>> besseli(n, z).diff(z)
besseli(n - 1, z)/2 + besseli(n + 1, z)/2
```

See also:

[*besselj*](#) (page 497), [*bessely*](#) (page 498), [*besselk*](#) (page 499)

References

[R390]

class sympy.functions.special.bessel.besselk(*nu*, *z*)
Modified Bessel function of the second kind.

Explanation

The Bessel K function of order ν is defined as

$$K_\nu(z) = \lim_{\mu \rightarrow \nu} \frac{\pi}{2} \frac{I_{-\mu}(z) - I_\mu(z)}{\sin(\pi\mu)},$$

where $I_\mu(z)$ is the modified Bessel function of the first kind.

It is a solution of the modified Bessel equation, and linearly independent from Y_ν .

Examples

```
>>> from sympy import besselk
>>> from sympy.abc import z, n
>>> besselk(n, z).diff(z)
-besselk(n - 1, z)/2 - besselk(n + 1, z)/2
```

See also:

[*besselj*](#) (page 497), [*besseli*](#) (page 499), [*bessely*](#) (page 498)

References

[R391]

class sympy.functions.special.bessel.hankel1(*nu*, *z*)
Hankel function of the first kind.

Explanation

This function is defined as

$$H_{\nu}^{(1)} = J_{\nu}(z) + iY_{\nu}(z),$$

where $J_{\nu}(z)$ is the Bessel function of the first kind, and $Y_{\nu}(z)$ is the Bessel function of the second kind.

It is a solution to Bessel's equation.

Examples

```
>>> from sympy import hankel1
>>> from sympy.abc import z, n
>>> hankel1(n, z).diff(z)
hankel1(n - 1, z)/2 - hankel1(n + 1, z)/2
```

See also:

[*hankel2*](#) (page 500), [*besselj*](#) (page 497), [*bessely*](#) (page 498)

References

[R392]

class sympy.functions.special.bessel.hankel2(*nu*, *z*)
Hankel function of the second kind.

Explanation

This function is defined as

$$H_{\nu}^{(2)} = J_{\nu}(z) - iY_{\nu}(z),$$

where $J_{\nu}(z)$ is the Bessel function of the first kind, and $Y_{\nu}(z)$ is the Bessel function of the second kind.

It is a solution to Bessel's equation, and linearly independent from $H_{\nu}^{(1)}$.

Examples

```
>>> from sympy import hankel2
>>> from sympy.abc import z, n
>>> hankel2(n, z).diff(z)
hankel2(n - 1, z)/2 - hankel2(n + 1, z)/2
```

See also:

[hankel1](#) (page 500), [besselj](#) (page 497), [bessely](#) (page 498)

References

[R393]

class `sympy.functions.special.bessel.jn(nu, z)`
Spherical Bessel function of the first kind.

Explanation

This function is a solution to the spherical Bessel equation

$$z^2 \frac{d^2 w}{dz^2} + 2z \frac{dw}{dz} + (z^2 - \nu(\nu + 1))w = 0.$$

It can be defined as

$$j_{\nu}(z) = \sqrt{\frac{\pi}{2z}} J_{\nu+\frac{1}{2}}(z),$$

where $J_{\nu}(z)$ is the Bessel function of the first kind.

The spherical Bessel functions of integral order are calculated using the formula:

$$j_n(z) = f_n(z) \sin z + (-1)^{n+1} f_{-n-1}(z) \cos z,$$

where the coefficients $f_n(z)$ are available as [sympy.polys.orthopolys.spherical_bessel_fn\(\)](#) (page 2441).

Examples

```
>>> from sympy import Symbol, jn, sin, cos, expand_func, besselj, bessely
>>> z = Symbol("z")
>>> nu = Symbol("nu", integer=True)
>>> print(expand_func(jn(0, z)))
sin(z)/z
>>> expand_func(jn(1, z)) == sin(z)/z**2 - cos(z)/z
True
>>> expand_func(jn(3, z))
(-6/z**2 + 15/z**4)*sin(z) + (1/z - 15/z**3)*cos(z)
>>> jn(nu, z).rewrite(besselj)
sqrt(2)*sqrt(pi)*sqrt(1/z)*besselj(nu + 1/2, z)/2
>>> jn(nu, z).rewrite(bessely)
(-1)**nu*sqrt(2)*sqrt(pi)*sqrt(1/z)*bessely(-nu - 1/2, z)/2
>>> jn(2, 5.2+0.3j).evalf(20)
0.099419756723640344491 - 0.054525080242173562897*I
```

See also:

[besselj](#) (page 497), [bessely](#) (page 498), [besselk](#) (page 499), [yn](#) (page 502)

References

[R394]

class sympy.functions.special.bessel.**yn**(nu, z)

Spherical Bessel function of the second kind.

Explanation

This function is another solution to the spherical Bessel equation, and linearly independent from j_n . It can be defined as

$$y_\nu(z) = \sqrt{\frac{\pi}{2z}} Y_{\nu+\frac{1}{2}}(z),$$

where $Y_\nu(z)$ is the Bessel function of the second kind.

For integral orders n , y_n is calculated using the formula:

$$y_n(z) = (-1)^{n+1} j_{-n-1}(z)$$

Examples

```
>>> from sympy import Symbol, yn, sin, cos, expand_func, besselj, bessely
>>> z = Symbol("z")
>>> nu = Symbol("nu", integer=True)
>>> print(expand_func(yn(0, z)))
-cos(z)/z
>>> expand_func(yn(1, z)) == -cos(z)/z**2 - sin(z)/z
```

(continues on next page)

(continued from previous page)

```
True
>>> yn(nu, z).rewrite(besselj)
(-1)**(nu + 1)*sqrt(2)*sqrt(pi)*sqrt(1/z)*besselj(-nu - 1/2, z)/2
>>> yn(nu, z).rewrite(bessely)
sqrt(2)*sqrt(pi)*sqrt(1/z)*bessely(nu + 1/2, z)/2
>>> yn(2, 5.2+0.3j).evalf(20)
0.18525034196069722536 + 0.014895573969924817587*I
```

See also:

[besselj](#) (page 497), [bessely](#) (page 498), [besselk](#) (page 499), [jn](#) (page 501)

References

[R395]

`sympy.functions.special.bessel.jn_zeros(n, k, method='sympy', dps=15)`

Zeros of the spherical Bessel function of the first kind.

Parameters

- n** : integer
order of Bessel function
- k** : integer
number of zeros to return

Explanation

This returns an array of zeros of j_n up to the k -th zero.

- `method = "sympy"`: uses `mpmath.besseljzero`
- `method = "scipy"`: uses the SciPy's `sph_jn` and `newton` to find all roots, which is faster than computing the zeros using a general numerical solver, but it requires SciPy and only works with low precision floating point numbers. (The function used with `method="sympy"` is a recent addition to `mpmath`; before that a general solver was used.)

Examples

```
>>> from sympy import jn_zeros
>>> jn_zeros(2, 4, dps=5)
[5.7635, 9.095, 12.323, 15.515]
```

See also:

[jn](#) (page 501), [yn](#) (page 502), [besselj](#) (page 497), [besselk](#) (page 499), [bessely](#) (page 498)

class `sympy.functions.special.bessel.marcumq(m, a, b)`

The Marcum Q-function.

Explanation

The Marcum Q-function is defined by the meromorphic continuation of

$$Q_m(a, b) = a^{-m+1} \int_b^\infty x^m e^{-\frac{a^2}{2} - \frac{x^2}{2}} I_{m-1}(ax) dx$$

Examples

```
>>> from sympy import marcumq
>>> from sympy.abc import m, a, b
>>> marcumq(m, a, b)
marcumq(m, a, b)
```

Special values:

```
>>> marcumq(m, 0, b)
uppergamma(m, b**2/2)/gamma(m)
>>> marcumq(0, 0, 0)
0
>>> marcumq(0, a, 0)
1 - exp(-a**2/2)
>>> marcumq(1, a, a)
1/2 + exp(-a**2)*besseli(0, a**2)/2
>>> marcumq(2, a, a)
1/2 + exp(-a**2)*besseli(0, a**2)/2 + exp(-a**2)*besseli(1, a**2)
```

Differentiation with respect to a and b is supported:

```
>>> from sympy import diff
>>> diff(marcumq(m, a, b), a)
a*(-marcumq(m, a, b) + marcumq(m + 1, a, b))
>>> diff(marcumq(m, a, b), b)
-a**(1 - m)*b**m*exp(-a**2/2 - b**2/2)*besseli(m - 1, a*b)
```

References

[R396], [R397]

Airy Functions

class sympy.functions.special.bessel.AiryBase(*args)

Abstract base class for Airy functions.

This class is meant to reduce code duplication.

class sympy.functions.special.bessel.airyai(arg)

The Airy function Ai of the first kind.

Explanation

The Airy function $\text{Ai}(z)$ is defined to be the function satisfying Airy's differential equation

$$\frac{d^2 w(z)}{dz^2} - zw(z) = 0.$$

Equivalently, for real z

$$\text{Ai}(z) := \frac{1}{\pi} \int_0^\infty \cos\left(\frac{t^3}{3} + zt\right) dt.$$

Examples

Create an Airy function object:

```
>>> from sympy import airyai
>>> from sympy.abc import z
```

```
>>> airyai(z)
airyai(z)
```

Several special values are known:

```
>>> airyai(0)
3**(1/3)/(3*gamma(2/3))
>>> from sympy import oo
>>> airyai(oo)
0
>>> airyai(-oo)
0
```

The Airy function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(airyai(z))
airyai(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(airyai(z), z)
airyaiprime(z)
>>> diff(airyai(z), z, 2)
z*airyai(z)
```

Series expansion is also supported:

```
>>> from sympy import series
>>> series(airyai(z), z, 0, 3)
3**(5/6)*gamma(1/3)/(6*pi) - 3**(1/6)*z*gamma(2/3)/(2*pi) + 0(z**3)
```

We can numerically evaluate the Airy function to arbitrary precision on the whole complex plane:

```
>>> airyai(-2).evalf(50)
0.22740742820168557599192443603787379946077222541710
```

Rewrite $\text{Ai}(z)$ in terms of hypergeometric functions:

```
>>> from sympy import hyper
>>> airyai(z).rewrite(hyper)
-3**(2/3)*z*hyper((), (4/3,), z**3/9)/(3*gamma(1/3)) + 3**(1/3)*hyper((),
→ (2/3,), z**3/9)/(3*gamma(2/3))
```

See also:

[airybi](#) (page 506)

Airy function of the second kind.

[airyaiprime](#) (page 508)

Derivative of the Airy function of the first kind.

[airybiprime](#) (page 509)

Derivative of the Airy function of the second kind.

References

[R398], [R399], [R400], [R401]

class `sympy.functions.special.bessel.airybi(arg)`

The Airy function Bi of the second kind.

Explanation

The Airy function $\text{Bi}(z)$ is defined to be the function satisfying Airy's differential equation

$$\frac{d^2 w(z)}{dz^2} - zw(z) = 0.$$

Equivalently, for real z

$$\text{Bi}(z) := \frac{1}{\pi} \int_0^\infty \exp\left(-\frac{t^3}{3} + zt\right) + \sin\left(\frac{t^3}{3} + zt\right) dt.$$

Examples

Create an Airy function object:

```
>>> from sympy import airybi
>>> from sympy.abc import z
```

```
>>> airybi(z)
airybi(z)
```

Several special values are known:

```
>>> airybi(0)
3**(5/6)/(3*gamma(2/3))
>>> from sympy import oo
>>> airybi(oo)
oo
>>> airybi(-oo)
0
```

The Airy function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(airybi(z))
airybi(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(airybi(z), z)
airybiprime(z)
>>> diff(airybi(z), z, 2)
z*airybi(z)
```

Series expansion is also supported:

```
>>> from sympy import series
>>> series(airybi(z), z, 0, 3)
3**(1/3)*gamma(1/3)/(2*pi) + 3**(2/3)*z*gamma(2/3)/(2*pi) + 0(z**3)
```

We can numerically evaluate the Airy function to arbitrary precision on the whole complex plane:

```
>>> airybi(-2).evalf(50)
-0.41230258795639848808323405461146104203453483447240
```

Rewrite $\text{Bi}(z)$ in terms of hypergeometric functions:

```
>>> from sympy import hyper
>>> airybi(z).rewrite(hyper)
3**(1/6)*z*hyper(( ), (4/3, ), z**3/9)/gamma(1/3) + 3**(5/6)*hyper(( ), (2/
→ 3, ), z**3/9)/(3*gamma(2/3))
```

See also:

[airyai](#) (page 504)

Airy function of the first kind.

[airyprime](#) (page 508)

Derivative of the Airy function of the first kind.

[airybiprime](#) (page 509)

Derivative of the Airy function of the second kind.

References

[R402], [R403], [R404], [R405]

class sympy.functions.special.bessel.airyaiprime(*arg*)

The derivative Ai' of the Airy function of the first kind.

Explanation

The Airy function $Ai'(z)$ is defined to be the function

$$Ai'(z) := \frac{d Ai(z)}{dz}.$$

Examples

Create an Airy function object:

```
>>> from sympy import airyaiprime
>>> from sympy.abc import z
```

```
>>> airyaiprime(z)
airyaiprime(z)
```

Several special values are known:

```
>>> airyaiprime(0)
-3**(2/3)/(3*gamma(1/3))
>>> from sympy import oo
>>> airyaiprime(oo)
0
```

The Airy function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(airyaiprime(z))
airyaiprime(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(airyaiprime(z), z)
z*airyai(z)
>>> diff(airyaiprime(z), z, 2)
z*airyaiprime(z) + airyai(z)
```

Series expansion is also supported:

```
>>> from sympy import series
>>> series(airyaiprime(z), z, 0, 3)
-3**(2/3)/(3*gamma(1/3)) + 3**(1/3)*z**2/(6*gamma(2/3)) + O(z**3)
```

We can numerically evaluate the Airy function to arbitrary precision on the whole complex plane:

```
>>> airyaiprime(-2).evalf(50)
0.61825902074169104140626429133247528291577794512415
```

Rewrite $\text{Ai}'(z)$ in terms of hypergeometric functions:

```
>>> from sympy import hyper
>>> airyaiprime(z).rewrite(hyper)
3**(1/3)*z**2*hyper(( ), (5/3, ), z**3/9)/(6*gamma(2/3)) - 3**(2/
→ 3)*hyper(( ), (1/3, ), z**3/9)/(3*gamma(1/3))
```

See also:

[***airyai***](#) (page 504)

Airy function of the first kind.

[***airybi***](#) (page 506)

Airy function of the second kind.

[***airybiprime***](#) (page 509)

Derivative of the Airy function of the second kind.

References

[R406], [R407], [R408], [R409]

class sympy.functions.special.bessel.**airybiprime**(arg)

The derivative Bi' of the Airy function of the first kind.

Explanation

The Airy function $\text{Bi}'(z)$ is defined to be the function

$$\text{Bi}'(z) := \frac{d \text{Bi}(z)}{dz}.$$

Examples

Create an Airy function object:

```
>>> from sympy import airybiprime
>>> from sympy.abc import z
```

```
>>> airybiprime(z)
airybiprime(z)
```

Several special values are known:

```
>>> airybiprime(0)
3**(1/6)/gamma(1/3)
>>> from sympy import oo
>>> airybiprime(oo)
oo
>>> airybiprime(-oo)
0
```

The Airy function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(airybiprime(z))
airybiprime(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(airybiprime(z), z)
z*airybi(z)
>>> diff(airybiprime(z), z, 2)
z*airybiprime(z) + airybi(z)
```

Series expansion is also supported:

```
>>> from sympy import series
>>> series(airybiprime(z), z, 0, 3)
3**(1/6)/gamma(1/3) + 3**(5/6)*z**2/(6*gamma(2/3)) + O(z**3)
```

We can numerically evaluate the Airy function to arbitrary precision on the whole complex plane:

```
>>> airybiprime(-2).evalf(50)
0.27879516692116952268509756941098324140300059345163
```

Rewrite $\text{Bi}'(z)$ in terms of hypergeometric functions:

```
>>> from sympy import hyper
>>> airybiprime(z).rewrite(hyper)
3**(5/6)*z**2*hyper(( ), (5/3, ), z**3/9)/(6*gamma(2/3)) + 3**(1/
→ 6)*hyper(( ), (1/3, ), z**3/9)/gamma(1/3)
```

See also:

[airyai](#) (page 504)

Airy function of the first kind.

[airybi](#) (page 506)

Airy function of the second kind.

[airyaiprime](#) (page 508)

Derivative of the Airy function of the first kind.

References

[R410], [R411], [R412], [R413]

B-Splines

`sympy.functions.special.bsplines.bspline_basis(d, knots, n, x)`

The n -th B-spline at x of degree d with knots.

Parameters

d : integer

degree of bspline

knots : list of integer values

list of knots points of bspline

n : integer

n -th B-spline

x : symbol

Explanation

B-Splines are piecewise polynomials of degree d . They are defined on a set of knots, which is a sequence of integers or floats.

Examples

The 0th degree splines have a value of 1 on a single interval:

```
>>> from sympy import bspline_basis
>>> from sympy.abc import x
>>> d = 0
>>> knots = tuple(range(5))
>>> bspline_basis(d, knots, 0, x)
Piecewise((1, (x >= 0) & (x <= 1)), (0, True))
```

For a given (d , $knots$) there are $\text{len}(knots)-d-1$ B-splines defined, that are indexed by n (starting at 0).

Here is an example of a cubic B-spline:

```
>>> bspline_basis(3, tuple(range(5)), 0, x)
Piecewise((x**3/6, (x >= 0) & (x <= 1)),
          (-x**3/2 + 2*x**2 - 2*x + 2/3,
           (x >= 1) & (x <= 2)),
          (x**3/2 - 4*x**2 + 10*x - 22/3,
           (x >= 2) & (x <= 3)),
          (-x**3/6 + 2*x**2 - 8*x + 32/3,
           (x >= 3) & (x <= 4)),
          (0, True))
```

By repeating knot points, you can introduce discontinuities in the B-splines and their derivatives:

```
>>> d = 1
>>> knots = (0, 0, 2, 3, 4)
>>> bspline_basis(d, knots, 0, x)
Piecewise((1 - x/2, (x >= 0) & (x <= 2)), (0, True))
```

It is quite time consuming to construct and evaluate B-splines. If you need to evaluate a B-spline many times, it is best to lambdify them first:

```
>>> from sympy import lambdify
>>> d = 3
>>> knots = tuple(range(10))
>>> b0 = bspline_basis(d, knots, 0, x)
>>> f = lambdify(x, b0)
>>> y = f(0.5)
```

See also:

[*bspline_basis_set*](#) (page 512)

References

[R414]

`sympy.functions.special.bsplines.bspline_basis_set(d, knots, x)`

Return the $\text{len}(\text{knots}) - d - 1$ B-splines at x of degree d with *knots*.

Parameters

d : integer

degree of bspline

knots : list of integers

list of knots points of bspline

x : symbol

Explanation

This function returns a list of piecewise polynomials that are the $\text{len}(\text{knots}) - d - 1$ B-splines of degree d for the given knots. This function calls `bspline_basis(d, knots, n, x)` for different values of n .

Examples

```
>>> from sympy import bspline_basis_set
>>> from sympy.abc import x
>>> d = 2
>>> knots = range(5)
>>> splines = bspline_basis_set(d, knots, x)
>>> splines
[Piecewise((x**2/2, (x >= 0) & (x <= 1)),
          (-x**2 + 3*x - 3/2, (x >= 1) & (x <= 2)),
          (x**2/2 - 3*x + 9/2, (x >= 2) & (x <= 3)),
          (0, True)),
 Piecewise((x**2/2 - x + 1/2, (x >= 1) & (x <= 2)),
          (-x**2 + 5*x - 11/2, (x >= 2) & (x <= 3)),
          (x**2/2 - 4*x + 8, (x >= 3) & (x <= 4)),
          (0, True))]
```

See also:

[*bspline_basis*](#) (page 511)

`sympy.functions.special.bsplines.interpolating_spline(d, x, X, Y)`

Return spline of degree *d*, passing through the given *X* and *Y* values.

Parameters

d : integer

Degree of B spline strictly greater than equal to one

x : symbol

X : list of strictly increasing integer values

list of X coordinates through which the spline passes

Y : list of strictly increasing integer values

list of Y coordinates through which the spline passes

Explanation

This function returns a piecewise function such that each part is a polynomial of degree not greater than *d*. The value of *d* must be 1 or greater and the values of *X* must be strictly increasing.

Examples

```
>>> from sympy import interpolating_spline
>>> from sympy.abc import x
>>> interpolating_spline(1, x, [1, 2, 4, 7], [3, 6, 5, 7])
Piecewise((3*x, (x >= 1) & (x <= 2)),
          (7 - x/2, (x >= 2) & (x <= 4)),
          (2*x/3 + 7/3, (x >= 4) & (x <= 7)))
>>> interpolating_spline(3, x, [-2, 0, 1, 3, 4], [4, 2, 1, 1, 3])
```

(continues on next page)

(continued from previous page)

```
Piecewise((7*x**3/117 + 7*x**2/117 - 131*x/117 + 2, (x >= -2) & (x <=
↪1)),
          (10*x**3/117 - 2*x**2/117 - 122*x/117 + 77/39, (x >= 1) & (x <=
↪4)))
```

See also:[bspline_basis_set](#) (page 512), [interpolating_poly](#) (page 2438)

Riemann Zeta and Related Functions

class `sympy.functions.special.zeta_functions.zeta`(*z*, *a_*=None)

Hurwitz zeta function (or Riemann zeta function).

Explanation

For $\operatorname{Re}(a) > 0$ and $\operatorname{Re}(s) > 1$, this function is defined as

$$\zeta(s, a) = \sum_{n=0}^{\infty} \frac{1}{(n+a)^s},$$

where the standard choice of argument for $n+a$ is used. For fixed a with $\operatorname{Re}(a) > 0$ the Hurwitz zeta function admits a meromorphic continuation to all of $\mathbb{C}$, it is an unbranched function with a simple pole at $s = 1$.

Analytic continuation to other a is possible under some circumstances, but this is not typically done.

The Hurwitz zeta function is a special case of the Lerch transcendent:

$$\zeta(s, a) = \Phi(1, s, a).$$

This formula defines an analytic continuation for all possible values of s and a (also $\operatorname{Re}(a) < 0$), see the documentation of [lerchphi](#) (page 517) for a description of the branching behavior.

If no value is passed for a , by this function assumes a default value of $a = 1$, yielding the Riemann zeta function.

Examples

For $a = 1$ the Hurwitz zeta function reduces to the famous Riemann zeta function:

$$\zeta(s, 1) = \zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}.$$

```
>>> from sympy import zeta
>>> from sympy.abc import s
>>> zeta(s, 1)
zeta(s)
>>> zeta(s)
zeta(s)
```

The Riemann zeta function can also be expressed using the Dirichlet eta function:

```
>>> from sympy import dirichlet_eta
>>> zeta(s).rewrite(dirichlet_eta)
dirichlet_eta(s)/(1 - 2**(1 - s))
```

The Riemann zeta function at positive even integer and negative odd integer values is related to the Bernoulli numbers:

```
>>> zeta(2)
pi**2/6
>>> zeta(4)
pi**4/90
>>> zeta(-1)
-1/12
```

The specific formulae are:

$$\zeta(2n) = (-1)^{n+1} \frac{B_{2n}(2\pi)^{2n}}{2(2n)!}$$

$$\zeta(-n) = -\frac{B_{n+1}}{n+1}$$

At negative even integers the Riemann zeta function is zero:

```
>>> zeta(-4)
0
```

No closed-form expressions are known at positive odd integers, but numerical evaluation is possible:

```
>>> zeta(3).n()
1.20205690315959
```

The derivative of $\zeta(s, a)$ with respect to a can be computed:

```
>>> from sympy.abc import a
>>> zeta(s, a).diff(a)
-s*zeta(s + 1, a)
```

However the derivative with respect to s has no useful closed form expression:

```
>>> zeta(s, a).diff(s)
Derivative(zeta(s, a), s)
```

The Hurwitz zeta function can be expressed in terms of the Lerch transcendent, [lerchphi](#) (page 517):

```
>>> from sympy import lerchphi
>>> zeta(s, a).rewrite(lerchphi)
lerchphi(1, s, a)
```

See also:

[dirichlet_eta](#) (page 516), [lerchphi](#) (page 517), [polylog](#) (page 516)

References

[R415], [R416]

class sympy.functions.special.zeta_functions.dirichlet_eta(*s*)
Dirichlet eta function.

Explanation

For $\operatorname{Re}(s) > 0$, this function is defined as

$$\eta(s) = \sum_{n=1}^{\infty} \frac{(-1)^{n-1}}{n^s}.$$

It admits a unique analytic continuation to all of $\mathbb{C}$. It is an entire, unbranched function.

Examples

The Dirichlet eta function is closely related to the Riemann zeta function:

```
>>> from sympy import dirichlet_eta, zeta
>>> from sympy.abc import s
>>> dirichlet_eta(s).rewrite(zeta)
(1 - 2**(1 - s))*zeta(s)
```

See also:

[zeta](#) (page 514)

References

[R417]

class sympy.functions.special.zeta_functions.polylog(*s, z*)
Polylogarithm function.

Explanation

For $|z| < 1$ and $s \in \mathbb{C}$, the polylogarithm is defined by

$$\operatorname{Li}_s(z) = \sum_{n=1}^{\infty} \frac{z^n}{n^s},$$

where the standard branch of the argument is used for n . It admits an analytic continuation which is branched at $z = 1$ (notably not on the sheet of initial definition), $z = 0$ and $z = \infty$.

The name polylogarithm comes from the fact that for $s = 1$, the polylogarithm is related to the ordinary logarithm (see examples), and that

$$\operatorname{Li}_{s+1}(z) = \int_0^z \frac{\operatorname{Li}_s(t)}{t} dt.$$

The polylogarithm is a special case of the Lerch transcendent:

$$\text{Li}_s(z) = z\Phi(z, s, 1).$$

Examples

For $z \in \{0, 1, -1\}$, the polylogarithm is automatically expressed using other functions:

```
>>> from sympy import polylog
>>> from sympy.abc import s
>>> polylog(s, 0)
0
>>> polylog(s, 1)
zeta(s)
>>> polylog(s, -1)
-dirichlet_eta(s)
```

If s is a negative integer, 0 or 1, the polylogarithm can be expressed using elementary functions. This can be done using `expand_func()`:

```
>>> from sympy import expand_func
>>> from sympy.abc import z
>>> expand_func(polylog(1, z))
-log(1 - z)
>>> expand_func(polylog(0, z))
z/(1 - z)
```

The derivative with respect to z can be computed in closed form:

```
>>> polylog(s, z).diff(z)
polylog(s - 1, z)/z
```

The polylogarithm can be expressed in terms of the lerch transcendent:

```
>>> from sympy import lerchphi
>>> polylog(s, z).rewrite(lerchphi)
z*lerchphi(z, s, 1)
```

See also:

[zeta](#) (page 514), [lerchphi](#) (page 517)

class `sympy.functions.special.zeta_functions.lerchphi(*args)`
 Lerch transcendent (Lerch phi function).

Explanation

For $\operatorname{Re}(a) > 0$, $|z| < 1$ and $s \in \mathbb{C}$, the Lerch transcendent is defined as

$$\Phi(z, s, a) = \sum_{n=0}^{\infty} \frac{z^n}{(n+a)^s},$$

where the standard branch of the argument is used for $n+a$, and by analytic continuation for other values of the parameters.

A commonly used related function is the Lerch zeta function, defined by

$$L(q, s, a) = \Phi(e^{2\pi i q}, s, a).$$

Analytic Continuation and Branching Behavior

It can be shown that

$$\Phi(z, s, a) = z\Phi(z, s, a+1) + a^{-s}.$$

This provides the analytic continuation to $\operatorname{Re}(a) \leq 0$.

Assume now $\operatorname{Re}(a) > 0$. The integral representation

$$\Phi_0(z, s, a) = \int_0^{\infty} \frac{t^{s-1} e^{-at}}{1 - ze^{-t}} \frac{dt}{\Gamma(s)}$$

provides an analytic continuation to $\mathbb{C} - [1, \infty)$. Finally, for $x \in (1, \infty)$ we find

$$\lim_{\epsilon \rightarrow 0^+} \Phi_0(x + i\epsilon, s, a) - \lim_{\epsilon \rightarrow 0^+} \Phi_0(x - i\epsilon, s, a) = \frac{2\pi i \log^{s-1} x}{x^a \Gamma(s)},$$

using the standard branch for both $\log x$ and $\log \log x$ (a branch of $\log \log x$ is needed to evaluate $\log x^{s-1}$). This concludes the analytic continuation. The Lerch transcendent is thus branched at $z \in \{0, 1, \infty\}$ and $a \in \mathbb{Z}_{\leq 0}$. For fixed z, a outside these branch points, it is an entire function of s .

Examples

The Lerch transcendent is a fairly general function, for this reason it does not automatically evaluate to simpler functions. Use `expand_func()` to achieve this.

If $z = 1$, the Lerch transcendent reduces to the Hurwitz zeta function:

```
>>> from sympy import lerchphi, expand_func
>>> from sympy.abc import z, s, a
>>> expand_func(lerchphi(1, s, a))
zeta(s, a)
```

More generally, if z is a root of unity, the Lerch transcendent reduces to a sum of Hurwitz zeta functions:

```
>>> expand_func(lerchphi(-1, s, a))
zeta(s, a/2)/2**s - zeta(s, a/2 + 1/2)/2**s
```

If $a = 1$, the Lerch transcendent reduces to the polylogarithm:

```
>>> expand_func(lerchphi(z, s, 1))
polylog(s, z)/z
```

More generally, if a is rational, the Lerch transcendent reduces to a sum of polylogarithms:

```
>>> from sympy import S
>>> expand_func(lerchphi(z, s, S(1)/2))
2**(s - 1)*(polylog(s, sqrt(z))/sqrt(z) -
            polylog(s, sqrt(z)*exp_polar(I*pi))/sqrt(z))
>>> expand_func(lerchphi(z, s, S(3)/2))
-2**s/z + 2**(s - 1)*(polylog(s, sqrt(z))/sqrt(z) -
            polylog(s, sqrt(z)*exp_polar(I*pi))/sqrt(z))/z
```

The derivatives with respect to z and a can be computed in closed form:

```
>>> lerchphi(z, s, a).diff(z)
(-a*lerchphi(z, s, a) + lerchphi(z, s - 1, a))/z
>>> lerchphi(z, s, a).diff(a)
-s*lerchphi(z, s + 1, a)
```

See also:

[polylog](#) (page 516), [zeta](#) (page 514)

References

[R418], [R419], [R420]

class sympy.functions.special.zeta_functions.**stieltjes**(n , $a=None$)

Represents Stieltjes constants, γ_k that occur in Laurent Series expansion of the Riemann zeta function.

Examples

```
>>> from sympy import stieltjes
>>> from sympy.abc import n, m
>>> stieltjes(n)
stieltjes(n)
```

The zero'th stieltjes constant:

```
>>> stieltjes(0)
EulerGamma
>>> stieltjes(0, 1)
EulerGamma
```

For generalized stieltjes constants:

```
>>> stieltjes(n, m)
stieltjes(n, m)
```

Constants are only defined for integers ≥ 0 :

```
>>> stieltjes(-1)
200
```

References

[R421]

Hypergeometric Functions

class `sympy.functions.special.hyper.hyper(ap, bq, z)`

The generalized hypergeometric function is defined by a series where the ratios of successive terms are a rational function of the summation index. When convergent, it is continued analytically to the largest possible domain.

Explanation

The hypergeometric function depends on two vectors of parameters, called the numerator parameters a_p , and the denominator parameters b_q . It also has an argument z . The series definition is

$${}_pF_q \left(\begin{matrix} a_1, \dots, a_p \\ b_1, \dots, b_q \end{matrix} \middle| z \right) = \sum_{n=0}^{\infty} \frac{(a_1)_n \cdots (a_p)_n}{(b_1)_n \cdots (b_q)_n} \frac{z^n}{n!},$$

where $(a)_n = (a)(a+1) \cdots (a+n-1)$ denotes the rising factorial.

If one of the b_q is a non-positive integer then the series is undefined unless one of the a_p is a larger (i.e., smaller in magnitude) non-positive integer. If none of the b_q is a non-positive integer and one of the a_p is a non-positive integer, then the series reduces to a polynomial. To simplify the following discussion, we assume that none of the a_p or b_q is a non-positive integer. For more details, see the references.

The series converges for all z if $p \leq q$, and thus defines an entire single-valued function in this case. If $p = q + 1$ the series converges for $|z| < 1$, and can be continued analytically into a half-plane. If $p > q + 1$ the series is divergent for all z .

Please note the hypergeometric function constructor currently does *not* check if the parameters actually yield a well-defined function.

Examples

The parameters a_p and b_q can be passed as arbitrary iterables, for example:

```
>>> from sympy import hyper
>>> from sympy.abc import x, n, a
>>> hyper((1, 2, 3), [3, 4], x)
hyper((1, 2, 3), (3, 4), x)
```

There is also pretty printing (it looks better using Unicode):


```
>>> from sympy import pprint
>>> pprint(hyper((1, 2, 3), [3, 4], x), use_unicode=False)

  |  /1, 2, 3 |  \
  |  |      |  | x|
 3 2 \ 3, 4  |  /
```

The parameters must always be iterables, even if they are vectors of length one or zero:

```
>>> hyper((1, ), [], x)
hyper((1,), (), x)
```

But of course they may be variables (but if they depend on x then you should not expect much implemented functionality):

```
>>> hyper((n, a), (n**2,), x)
hyper((n, a), (n**2,), x)
```

The hypergeometric function generalizes many named special functions. The function `hyperexpand()` tries to express a hypergeometric function using named special functions. For example:

```
>>> from sympy import hyperexpand
>>> hyperexpand(hyper([], [], x))
exp(x)
```

You can also use `expand_func()`:

```
>>> from sympy import expand_func
>>> expand_func(x*hyper([1, 1], [2], -x))
log(x + 1)
```

More examples:

```
>>> from sympy import S
>>> hyperexpand(hyper([], [S(1)/2], -x**2/4))
cos(x)
>>> hyperexpand(x*hyper([S(1)/2, S(1)/2], [S(3)/2], x**2))
asin(x)
```

We can also sometimes `hyperexpand()` parametric functions:

```
>>> from sympy.abc import a
>>> hyperexpand(hyper([-a], [], x))
(1 - x)**a
```

See also:

[`sympy.simplify.hyperexpand`](#) (page 687), [`gamma`](#) (page 459), [`meijerg`](#) (page 522)

References

[R422], [R423]

property `ap`

Numerator parameters of the hypergeometric function.

property `argument`

Argument of the hypergeometric function.

property `bq`

Denominator parameters of the hypergeometric function.

property `convergence_statement`

Return a condition on z under which the series converges.

property `eta`

A quantity related to the convergence of the series.

property `radius_of_convergence`

Compute the radius of convergence of the defining series.

Explanation

Note that even if this is not `oo`, the function may still be evaluated outside of the radius of convergence by analytic continuation. But if this is zero, then the function is not actually defined anywhere else.

Examples

```
>>> from sympy import hyper
>>> from sympy.abc import z
>>> hyper((1, 2), [3], z).radius_of_convergence
1
>>> hyper((1, 2, 3), [4], z).radius_of_convergence
0
>>> hyper((1, 2), (3, 4), z).radius_of_convergence
oo
```

class `sympy.functions.special.hyper.meijerg(*args)`

The Meijer G-function is defined by a Mellin-Barnes type integral that resembles an inverse Mellin transform. It generalizes the hypergeometric functions.

Explanation

The Meijer G-function depends on four sets of parameters. There are “*numerator parameters*” $a_1, \dots, a_n$ and $a_{n+1}, \dots, a_p$, and there are “*denominator parameters*” $b_1, \dots, b_m$ and $b_{m+1}, \dots, b_q$. Confusingly, it is traditionally denoted as follows (note the position of m, n, p, q , and how they relate to the lengths of the four parameter vectors):

$$G_{p,q}^{m,n} \left(\begin{matrix} a_1, \dots, a_n & a_{n+1}, \dots, a_p \\ b_1, \dots, b_m & b_{m+1}, \dots, b_q \end{matrix} \middle| z \right).$$

However, in SymPy the four parameter vectors are always available separately (see examples), so that there is no need to keep track of the decorating sub- and super-scripts on the G symbol.

The G function is defined as the following integral:

$$\frac{1}{2\pi i} \int_L \frac{\prod_{j=1}^m \Gamma(b_j - s) \prod_{j=1}^n \Gamma(1 - a_j + s)}{\prod_{j=m+1}^q \Gamma(1 - b_j + s) \prod_{j=n+1}^p \Gamma(a_j - s)} z^s ds,$$

where $\Gamma(z)$ is the gamma function. There are three possible contours which we will not describe in detail here (see the references). If the integral converges along more than one of them, the definitions agree. The contours all separate the poles of $\Gamma(1 - a_j + s)$ from the poles of $\Gamma(b_k - s)$, so in particular the G function is undefined if $a_j - b_k \in \mathbb{Z}_{>0}$ for some $j \leq n$ and $k \leq m$.

The conditions under which one of the contours yields a convergent integral are complicated and we do not state them here, see the references.

Please note currently the Meijer G-function constructor does *not* check any convergence conditions.

Examples

You can pass the parameters either as four separate vectors:

```
>>> from sympy import meijerg, Tuple, pprint
>>> from sympy.abc import x, a
>>> pprint(meijerg((1, 2), (a, 4), (5,), [], x), use_unicode=False)
__1, 2 /1, 2 a, 4 | \
/_|         |         | x|
\_|4, 1 \ 5         | /
```

Or as two nested vectors:

```
>>> pprint(meijerg([(1, 2), (3, 4)], ([5], Tuple()), x), use_
→unicode=False)
__1, 2 /1, 2 3, 4 | \
/_|         |         | x|
\_|4, 1 \ 5         | /
```

As with the hypergeometric function, the parameters may be passed as arbitrary iterables. Vectors of length zero and one also have to be passed as iterables. The parameters need not be constants, but if they depend on the argument then not much implemented functionality should be expected.

All the subvectors of parameters are available:

```
>>> from sympy import pprint
>>> g = meijerg([1], [2], [3], [4], x)
>>> pprint(g, use_unicode=False)
      1, 1 /1  2 | \
    /___|   |   | x|
   \| 2, 2 \3  4 | /
>>> g.an
(1,)
>>> g.ap
(1, 2)
>>> g.aother
(2,)
>>> g.bm
(3,)
>>> g.bq
(3, 4)
>>> g.bother
(4,)
```

The Meijer G-function generalizes the hypergeometric functions. In some cases it can be expressed in terms of hypergeometric functions, using Slater's theorem. For example:

```
>>> from sympy import hyperexpand
>>> from sympy.abc import a, b, c
>>> hyperexpand(meijerg([a], [], [c], [b], x), allow_hyper=True)
x**c*gamma(-a + c + 1)*hyper((-a + c + 1,),
                             (-b + c + 1,), -x)/gamma(-b + c + 1)
```

Thus the Meijer G-function also subsumes many named functions as special cases. You can use `expand_func()` or `hyperexpand()` to (try to) rewrite a Meijer G-function in terms of named special functions. For example:

```
>>> from sympy import expand_func, S
>>> expand_func(meijerg([], [], [[0], []], -x))
exp(x)
>>> hyperexpand(meijerg([], [], [[S(1)/2], [0]], (x/2)**2))
sin(x)/sqrt(pi)
```

See also:

[hyper](#) (page 520), [sympy.simplify.hyperexpand](#) (page 687)

References

[R424], [R425]

property `an`

First set of numerator parameters.

property `aother`

Second set of numerator parameters.

property `ap`

Combined numerator parameters.

property argument

Argument of the Meijer G-function.

property bm

First set of denominator parameters.

property bother

Second set of denominator parameters.

property bq

Combined denominator parameters.

property delta

A quantity related to the convergence region of the integral, c.f. references.

get_period()

Return a number P such that $G(x * \exp(I * P)) == G(x)$.

Examples

```
>>> from sympy import meijerg, pi, S
>>> from sympy.abc import z
```

```
>>> meijerg([1], [], [], [], z).get_period()
2*pi
>>> meijerg([pi], [], [], [], z).get_period()
oo
>>> meijerg([1, 2], [], [], [], z).get_period()
oo
>>> meijerg([1,1], [2], [1, S(1)/2, S(1)/3], [1], z).get_period()
12*pi
```

integrand(s)

Get the defining integrand $D(s)$.

property is_number

Returns true if expression has numeric data only.

property nu

A quantity related to the convergence region of the integral, c.f. references.

class sympy.functions.special.hyper.**appellf1**($a, b1, b2, c, x, y$)

This is the Appell hypergeometric function of two variables as:

$$F_1(a, b_1, b_2, c, x, y) = \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} \frac{(a)_{m+n} (b_1)_m (b_2)_n}{(c)_{m+n}} \frac{x^m y^n}{m! n!}.$$

Examples

```
>>> from sympy import appellf1, symbols
>>> x, y, a, b1, b2, c = symbols('x y a b1 b2 c')
>>> appellf1(2., 1., 6., 4., 5., 6.)
0.0063339426292673
>>> appellf1(12., 12., 6., 4., 0.5, 0.12)
172870711.659936
>>> appellf1(40, 2, 6, 4, 15, 60)
appellf1(40, 2, 6, 4, 15, 60)
>>> appellf1(20., 12., 10., 3., 0.5, 0.12)
15605338197184.4
>>> appellf1(40, 2, 6, 4, x, y)
appellf1(40, 2, 6, 4, x, y)
>>> appellf1(a, b1, b2, c, x, y)
appellf1(a, b1, b2, c, x, y)
```

References

[R426], [R427]

Elliptic integrals

class sympy.functions.special.elliptic_integrals.elliptic_k(*m*)

The complete elliptic integral of the first kind, defined by

$$K(m) = F\left(\frac{\pi}{2} \middle| m\right)$$

where $F(z|m)$ is the Legendre incomplete elliptic integral of the first kind.

Explanation

The function $K(m)$ is a single-valued function on the complex plane with branch cut along the interval $(1, \infty)$.

Note that our notation defines the incomplete elliptic integral in terms of the parameter m instead of the elliptic modulus (eccentricity) k . In this case, the parameter m is defined as $m = k^2$.

Examples

```
>>> from sympy import elliptic_k, I
>>> from sympy.abc import m
>>> elliptic_k(0)
pi/2
>>> elliptic_k(1.0 + I)
1.50923695405127 + 0.625146415202697*I
>>> elliptic_k(m).series(n=3)
pi/2 + pi*m/8 + 9*pi*m**2/128 + O(m**3)
```

See also:

[`elliptic\_f`](#) (page 527)

References

[R428], [R429]

class `sympy.functions.special.elliptic_integrals.elliptic_f(z, m)`

The Legendre incomplete elliptic integral of the first kind, defined by

$$F(z|m) = \int_0^z \frac{dt}{\sqrt{1 - m \sin^2 t}}$$

Explanation

This function reduces to a complete elliptic integral of the first kind, $K(m)$, when $z = \pi/2$.

Note that our notation defines the incomplete elliptic integral in terms of the parameter m instead of the elliptic modulus (eccentricity) k . In this case, the parameter m is defined as $m = k^2$.

Examples

```
>>> from sympy import elliptic_f, I
>>> from sympy.abc import z, m
>>> elliptic_f(z, m).series(z)
z + z**5*(3*m**2/40 - m/30) + m*z**3/6 + O(z**6)
>>> elliptic_f(3.0 + I/2, 1.0 + I)
2.909449841483 + 1.74720545502474*I
```

See also:

[`elliptic\_k`](#) (page 526)

References

[R430], [R431]

class `sympy.functions.special.elliptic_integrals.elliptic_e(m, z=None)`

Called with two arguments z and m , evaluates the incomplete elliptic integral of the second kind, defined by

$$E(z|m) = \int_0^z \sqrt{1 - m \sin^2 t} dt$$

Called with a single argument m , evaluates the Legendre complete elliptic integral of the second kind

$$E(m) = E\left(\frac{\pi}{2}|m\right)$$

Explanation

The function $E(m)$ is a single-valued function on the complex plane with branch cut along the interval $(1, \infty)$.

Note that our notation defines the incomplete elliptic integral in terms of the parameter m instead of the elliptic modulus (eccentricity) k . In this case, the parameter m is defined as $m = k^2$.

Examples

```
>>> from sympy import elliptic_e, I
>>> from sympy.abc import z, m
>>> elliptic_e(z, m).series(z)
z + z**5*(-m**2/40 + m/30) - m*z**3/6 + O(z**6)
>>> elliptic_e(m).series(n=4)
pi/2 - pi*m/8 - 3*pi*m**2/128 - 5*pi*m**3/512 + O(m**4)
>>> elliptic_e(1 + I, 2 - I/2).n()
1.55203744279187 + 0.290764986058437*I
>>> elliptic_e(0)
pi/2
>>> elliptic_e(2.0 - I)
0.991052601328069 + 0.81879421395609*I
```

References

[R432], [R433], [R434]

class sympy.functions.special.elliptic_integrals.elliptic_pi($n, m, z=None$)

Called with three arguments n , z and m , evaluates the Legendre incomplete elliptic integral of the third kind, defined by

$$\Pi(n; z|m) = \int_0^z \frac{dt}{(1 - n \sin^2 t) \sqrt{1 - m \sin^2 t}}$$

Called with two arguments n and m , evaluates the complete elliptic integral of the third kind:

$$\Pi(n|m) = \Pi\left(n; \frac{\pi}{2} |m\right)$$

Explanation

Note that our notation defines the incomplete elliptic integral in terms of the parameter m instead of the elliptic modulus (eccentricity) k . In this case, the parameter m is defined as $m = k^2$.

Examples

```
>>> from sympy import elliptic_pi, I
>>> from sympy.abc import z, n, m
>>> elliptic_pi(n, z, m).series(z, n=4)
z + z**3*(m/6 + n/3) + O(z**4)
>>> elliptic_pi(0.5 + I, 1.0 - I, 1.2)
2.50232379629182 - 0.760939574180767*I
>>> elliptic_pi(0, 0)
pi/2
>>> elliptic_pi(1.0 - I/3, 2.0 + I)
3.29136443417283 + 0.32555634906645*I
```

References

[R435], [R436], [R437]

Mathieu Functions

class sympy.functions.special.mathieu_functions.**MathieuBase**(*args)

Abstract base class for Mathieu functions.

This class is meant to reduce code duplication.

class sympy.functions.special.mathieu_functions.**mathieus**(a, q, z)

The Mathieu Sine function $S(a, q, z)$.

Explanation

This function is one solution of the Mathieu differential equation:

$$y(x)'' + (a - 2q \cos(2x))y(x) = 0$$

The other solution is the Mathieu Cosine function.

Examples

```
>>> from sympy import diff, mathieus
>>> from sympy.abc import a, q, z
```

```
>>> mathieus(a, q, z)
mathieus(a, q, z)
```

```
>>> mathieus(a, 0, z)
sin(sqrt(a)*z)
```

```
>>> diff(mathieus(a, q, z), z)
mathieusprime(a, q, z)
```

See also:***mathieuc* (page 530)**

Mathieu cosine function.

***mathieusprime* (page 531)**

Derivative of Mathieu sine function.

***mathieucprime* (page 531)**

Derivative of Mathieu cosine function.

References

[R438], [R439], [R440], [R441]

class sympy.functions.special.mathieu_functions.**mathieuc**(*a*, *q*, *z*)The Mathieu Cosine function $C(a, q, z)$.**Explanation**

This function is one solution of the Mathieu differential equation:

$$y(x)'' + (a - 2q \cos(2x))y(x) = 0$$

The other solution is the Mathieu Sine function.

Examples

```
>>> from sympy import diff, mathieuc
>>> from sympy.abc import a, q, z
```

```
>>> mathieuc(a, q, z)
mathieuc(a, q, z)
```

```
>>> mathieuc(a, 0, z)
cos(sqrt(a)*z)
```

```
>>> diff(mathieuc(a, q, z), z)
mathieucprime(a, q, z)
```

See also:***mathieus* (page 529)**

Mathieu sine function

***mathieusprime* (page 531)**

Derivative of Mathieu sine function

***mathieucprime* (page 531)**

Derivative of Mathieu cosine function

References

[R442], [R443], [R444], [R445]

class sympy.functions.special.mathieu_functions.mathieusprime(*a*, *q*, *z*)

The derivative $S'(a, q, z)$ of the Mathieu Sine function.

Explanation

This function is one solution of the Mathieu differential equation:

$$y(x)'' + (a - 2q \cos(2x))y(x) = 0$$

The other solution is the Mathieu Cosine function.

Examples

```
>>> from sympy import diff, mathieusprime
>>> from sympy.abc import a, q, z
```

```
>>> mathieusprime(a, q, z)
mathieusprime(a, q, z)
```

```
>>> mathieusprime(a, 0, z)
sqrt(a)*cos(sqrt(a)*z)
```

```
>>> diff(mathieusprime(a, q, z), z)
(-a + 2*q*cos(2*z))*mathieus(a, q, z)
```

See also:

***mathieus* (page 529)**

Mathieu sine function

***mathieuc* (page 530)**

Mathieu cosine function

***mathieucprime* (page 531)**

Derivative of Mathieu cosine function

References

[R446], [R447], [R448], [R449]

class sympy.functions.special.mathieu_functions.mathieucprime(*a*, *q*, *z*)

The derivative $C'(a, q, z)$ of the Mathieu Cosine function.

Explanation

This function is one solution of the Mathieu differential equation:

$$y(x)'' + (a - 2q \cos(2x))y(x) = 0$$

The other solution is the Mathieu Sine function.

Examples

```
>>> from sympy import diff, mathieucprime
>>> from sympy.abc import a, q, z
```

```
>>> mathieucprime(a, q, z)
mathieucprime(a, q, z)
```

```
>>> mathieucprime(a, 0, z)
-sqrt(a)*sin(sqrt(a)*z)
```

```
>>> diff(mathieucprime(a, q, z), z)
(-a + 2*q*cos(2*z))*mathieuc(a, q, z)
```

See also:

mathieus (page 529)

Mathieu sine function

mathieuc (page 530)

Mathieu cosine function

mathieusprime (page 531)

Derivative of Mathieu sine function

References

[R450], [R451], [R452], [R453]

Orthogonal Polynomials

This module mainly implements special orthogonal polynomials.

See also `functions.combinatorial.numbers` which contains some combinatorial polynomials.

Jacobi Polynomials

class sympy.functions.special.polynomials.jacobi(n, a, b, x)

Jacobi polynomial $P_n^{(\alpha, \beta)}(x)$.

Explanation

`jacobi( $n$ ,  $\alpha$ ,  $\beta$ ,  $x$ )` gives the n th Jacobi polynomial in x , $P_n^{(\alpha, \beta)}(x)$.

The Jacobi polynomials are orthogonal on $[-1, 1]$ with respect to the weight $(1 - x)^\alpha (1 + x)^\beta$.

Examples

```
>>> from sympy import jacobi, S, conjugate, diff
>>> from sympy.abc import a, b, n, x
```

```
>>> jacobi(0, a, b, x)
1
>>> jacobi(1, a, b, x)
a/2 - b/2 + x*(a/2 + b/2 + 1)
>>> jacobi(2, a, b, x)
a**2/8 - a*b/4 - a/8 + b**2/8 - b/8 + x**2*(a**2/8 + a*b/4 + 7*a/8 +
↪ b**2/8 + 7*b/8 + 3/2) + x*(a**2/4 + 3*a/4 - b**2/4 - 3*b/4) - 1/2
```

```
>>> jacobi(n, a, b, x)
jacobi(n, a, b, x)
```

```
>>> jacobi(n, a, a, x)
RisingFactorial(a + 1, n)*gegenbauer(n,
a + 1/2, x)/RisingFactorial(2*a + 1, n)
```

```
>>> jacobi(n, 0, 0, x)
legendre(n, x)
```

```
>>> jacobi(n, S(1)/2, S(1)/2, x)
RisingFactorial(3/2, n)*chebyshevu(n, x)/factorial(n + 1)
```

```
>>> jacobi(n, -S(1)/2, -S(1)/2, x)
RisingFactorial(1/2, n)*chebyshevt(n, x)/factorial(n)
```

```
>>> jacobi(n, a, b, -x)
(-1)**n*jacobi(n, b, a, x)
```

```
>>> jacobi(n, a, b, 0)
gamma(a + n + 1)*hyper((-b - n, -n), (a + 1,), -1)/
↪ (2**n*factorial(n)*gamma(a + 1))
>>> jacobi(n, a, b, 1)
RisingFactorial(a + 1, n)/factorial(n)
```

```
>>> conjugate(jacobi(n, a, b, x))
jacobi(n, conjugate(a), conjugate(b), conjugate(x))
```

```
>>> diff(jacobi(n,a,b,x), x)
(a/2 + b/2 + n/2 + 1/2)*jacobi(n - 1, a + 1, b + 1, x)
```

See also:

[gegenbauer](#) (page 535), [chebyshevt_root](#) (page 539), [chebyshevu](#) (page 538), [chebyshevu_root](#) (page 539), [legendre](#) (page 540), [assoc_legendre](#) (page 540), [hermite](#) (page 541), [laguerre](#) (page 542), [assoc_laguerre](#) (page 543), [sympy.polys.orthopolys.jacobi_poly](#) (page 2440), [sympy.polys.orthopolys.gegenbauer_poly](#) (page 2440), [sympy.polys.orthopolys.chebyshevt_poly](#) (page 2439), [sympy.polys.orthopolys.chebyshevu_poly](#) (page 2439), [sympy.polys.orthopolys.hermite_poly](#) (page 2440), [sympy.polys.orthopolys.legendre_poly](#) (page 2440), [sympy.polys.orthopolys.laguerre_poly](#) (page 2441)

References

[R454], [R455], [R456]

`sympy.functions.special.polynomials.jacobi_normalized(n, a, b, x)`

Jacobi polynomial $P_n^{(\alpha,\beta)}(x)$.

Parameters

- n** : integer degree of polynomial
- a** : alpha value
- b** : beta value
- x** : symbol

Explanation

`jacobi_normalized(n, alpha, beta, x)` gives the n th Jacobi polynomial in x , $P_n^{(\alpha,\beta)}(x)$.

The Jacobi polynomials are orthogonal on $[-1,1]$ with respect to the weight $(1-x)^\alpha(1+x)^\beta$.

This functions returns the polynomials normilzed:

$$\int_{-1}^1 P_m^{(\alpha,\beta)}(x)P_n^{(\alpha,\beta)}(x)(1-x)^\alpha(1+x)^\beta dx = \delta_{m,n}$$

Examples

```
>>> from sympy import jacobi_normalized
>>> from sympy.abc import n,a,b,x
```

```
>>> jacobi_normalized(n, a, b, x)
jacobi(n, a, b, x)/sqrt(2**(a + b + 1)*gamma(a + n + 1)*gamma(b + n + 1)/
↳ ((a + b + 2*n + 1)*factorial(n)*gamma(a + b + n + 1)))
```

See also:

[gegenbauer](#) (page 535), [chebyshevt_root](#) (page 539), [chebyshevu](#) (page 538), [chebyshevu_root](#) (page 539), [legendre](#) (page 540), [assoc_legendre](#) (page 540), [hermite](#) (page 541), [laguerre](#) (page 542), [assoc_laguerre](#) (page 543), [sympy.polys.orthopolys.jacobi_poly](#) (page 2440), [sympy.polys.orthopolys.gegenbauer_poly](#) (page 2440), [sympy.polys.orthopolys.chebyshevt_poly](#) (page 2439), [sympy.polys.orthopolys.chebyshevu_poly](#) (page 2439), [sympy.polys.orthopolys.hermite_poly](#) (page 2440), [sympy.polys.orthopolys.legendre_poly](#) (page 2440), [sympy.polys.orthopolys.laguerre_poly](#) (page 2441)

References

[R457], [R458], [R459]

Gegenbauer Polynomials

class sympy.functions.special.polynomials.**gegenbauer**(n , a , x)

Gegenbauer polynomial $C_n^{(\alpha)}(x)$.

Explanation

`gegenbauer( $n$ ,  $\alpha$ ,  $x$ )` gives the n th Gegenbauer polynomial in x , $C_n^{(\alpha)}(x)$.

The Gegenbauer polynomials are orthogonal on $[-1,1]$ with respect to the weight $(1 - x^2)^{\alpha - \frac{1}{2}}$.

Examples

```
>>> from sympy import gegenbauer, conjugate, diff
>>> from sympy.abc import n,a,x
>>> gegenbauer(0, a, x)
1
>>> gegenbauer(1, a, x)
2*a*x
>>> gegenbauer(2, a, x)
-a + x**2*(2*a**2 + 2*a)
>>> gegenbauer(3, a, x)
x**3*(4*a**3/3 + 4*a**2 + 8*a/3) + x*(-2*a**2 - 2*a)
```

```
>>> gegenbauer(n, a, x)
gegenbauer(n, a, x)
>>> gegenbauer(n, a, -x)
(-1)**n*gegenbauer(n, a, x)
```

```
>>> gegenbauer(n, a, 0)
2**n*sqrt(pi)*gamma(a + n/2)/(gamma(a)*gamma(1/2 - n/2)*gamma(n + 1))
>>> gegenbauer(n, a, 1)
gamma(2*a + n)/(gamma(2*a)*gamma(n + 1))
```

```
>>> conjugate(gegenbauer(n, a, x))
gegenbauer(n, conjugate(a), conjugate(x))
```

```
>>> diff(gegenbauer(n, a, x), x)
2*a*gegenbauer(n - 1, a + 1, x)
```

See also:

[jacobi](#) (page 533), [chebyshevt_root](#) (page 539), [chebyshevu](#) (page 538), [chebyshevu_root](#) (page 539), [legendre](#) (page 540), [assoc_legendre](#) (page 540), [hermite](#) (page 541), [laguerre](#) (page 542), [assoc_laguerre](#) (page 543), [sympy.polys.orthopolys.jacobi_poly](#) (page 2440), [sympy.polys.orthopolys.gegenbauer_poly](#) (page 2440), [sympy.polys.orthopolys.chebyshevt_poly](#) (page 2439), [sympy.polys.orthopolys.chebyshevu_poly](#) (page 2439), [sympy.polys.orthopolys.hermite_poly](#) (page 2440), [sympy.polys.orthopolys.legendre_poly](#) (page 2440), [sympy.polys.orthopolys.laguerre_poly](#) (page 2441)

References

[R460], [R461], [R462]

Chebyshev Polynomials

class `sympy.functions.special.polynomials.chebyshevt`(*n*, *x*)

Chebyshev polynomial of the first kind, $T_n(x)$.

Explanation

`chebyshevt`(*n*, *x*) gives the *n*th Chebyshev polynomial (of the first kind) in *x*, $T_n(x)$.

The Chebyshev polynomials of the first kind are orthogonal on $[-1, 1]$ with respect to the weight $\frac{1}{\sqrt{1-x^2}}$.

Examples

```
>>> from sympy import chebyshevt, diff
>>> from sympy.abc import n,x
>>> chebyshevt(0, x)
1
>>> chebyshevt(1, x)
x
>>> chebyshevt(2, x)
2*x**2 - 1
```

```
>>> chebyshevt(n, x)
chebyshevt(n, x)
>>> chebyshevt(n, -x)
(-1)**n*chebyshevt(n, x)
>>> chebyshevt(-n, x)
chebyshevt(n, x)
```

```
>>> chebyshevt(n, 0)
cos(pi*n/2)
>>> chebyshevt(n, -1)
(-1)**n
```

```
>>> diff(chebyshevt(n, x), x)
n*chebyshevu(n - 1, x)
```

See also:

[jacobi](#) (page 533), [gegenbauer](#) (page 535), [chebyshevt_root](#) (page 539), [chebyshevu](#) (page 538), [chebyshevu_root](#) (page 539), [legendre](#) (page 540), [assoc_legendre](#) (page 540), [hermite](#) (page 541), [laguerre](#) (page 542), [assoc_laguerre](#) (page 543), [sympy.polys.orthopolys.jacobi_poly](#) (page 2440), [sympy.polys.orthopolys.gegenbauer_poly](#) (page 2440), [sympy.polys.orthopolys.chebyshevt_poly](#) (page 2439), [sympy.polys.orthopolys.chebyshevu_poly](#) (page 2439), [sympy.polys.orthopolys.hermite_poly](#) (page 2440), [sympy.polys.orthopolys.legendre_poly](#) (page 2440), [sympy.polys.orthopolys.laguerre_poly](#) (page 2441)

References

[R463], [R464], [R465], [R466], [R467]

class `sympy.functions.special.polynomials.chebyshevu(n, x)`

Chebyshev polynomial of the second kind, $U_n(x)$.

Explanation

`chebyshevu(n, x)` gives the n th Chebyshev polynomial of the second kind in x , $U_n(x)$.

The Chebyshev polynomials of the second kind are orthogonal on $[-1, 1]$ with respect to the weight $\sqrt{1 - x^2}$.

Examples

```
>>> from sympy import chebyshevu, diff
>>> from sympy.abc import n,x
>>> chebyshevu(0, x)
1
>>> chebyshevu(1, x)
2*x
>>> chebyshevu(2, x)
4*x**2 - 1
```

```
>>> chebyshevu(n, x)
chebyshevu(n, x)
>>> chebyshevu(n, -x)
(-1)**n*chebyshevu(n, x)
>>> chebyshevu(-n, x)
-chebyshevu(n - 2, x)
```

```
>>> chebyshevu(n, 0)
cos(pi*n/2)
>>> chebyshevu(n, 1)
n + 1
```

```
>>> diff(chebyshevu(n, x), x)
(-x*chebyshevu(n, x) + (n + 1)*chebyshevt(n + 1, x))/(x**2 - 1)
```

See also:

[*jacobi*](#) (page 533), [*gegenbauer*](#) (page 535), [*chebyshevt*](#) (page 536), [*chebyshevt_root*](#) (page 539), [*chebyshevu_root*](#) (page 539), [*legendre*](#) (page 540), [*assoc_legendre*](#) (page 540), [*hermite*](#) (page 541), [*laguerre*](#) (page 542), [*assoc_laguerre*](#) (page 543), [*sympy.polys.orthopolys.jacobi_poly*](#) (page 2440), [*sympy.polys.orthopolys.gegenbauer_poly*](#) (page 2440), [*sympy.polys.orthopolys.chebyshevt_poly*](#) (page 2439), [*sympy.polys.orthopolys.chebyshevu_poly*](#) (page 2439), [*sympy.polys.orthopolys.hermite_poly*](#) (page 2440), [*sympy.polys.orthopolys.legendre_poly*](#) (page 2440), [*sympy.polys.orthopolys.laguerre_poly*](#) (page 2441)

References

[R468], [R469], [R470], [R471], [R472]

class sympy.functions.special.polynomials.chebyshevt_root(*n*, *k*)

chebyshevt_root(*n*, *k*) returns the *k*th root (indexed from zero) of the *n*th Chebyshev polynomial of the first kind; that is, if $0 \leq k < n$, `chebyshevt(n, chebyshevt_root(n, k)) == 0`.

Examples

```
>>> from sympy import chebyshevt, chebyshevt_root
>>> chebyshevt_root(3, 2)
-sqrt(3)/2
>>> chebyshevt(3, chebyshevt_root(3, 2))
0
```

See also:

[jacobi](#) (page 533), [gegenbauer](#) (page 535), [chebyshevt](#) (page 536), [chebyshevu](#) (page 538), [chebyshevu_root](#) (page 539), [legendre](#) (page 540), [assoc_legendre](#) (page 540), [hermite](#) (page 541), [laguerre](#) (page 542), [assoc_laguerre](#) (page 543), [sympy.polys.orthopolys.jacobi_poly](#) (page 2440), [sympy.polys.orthopolys.gegenbauer_poly](#) (page 2440), [sympy.polys.orthopolys.chebyshevt_poly](#) (page 2439), [sympy.polys.orthopolys.chebyshevu_poly](#) (page 2439), [sympy.polys.orthopolys.hermite_poly](#) (page 2440), [sympy.polys.orthopolys.legendre_poly](#) (page 2440), [sympy.polys.orthopolys.laguerre_poly](#) (page 2441)

class sympy.functions.special.polynomials.chebyshevu_root(*n*, *k*)

chebyshevu_root(*n*, *k*) returns the *k*th root (indexed from zero) of the *n*th Chebyshev polynomial of the second kind; that is, if $0 \leq k < n$, `chebyshevu(n, chebyshevu_root(n, k)) == 0`.

Examples

```
>>> from sympy import chebyshevu, chebyshevu_root
>>> chebyshevu_root(3, 2)
-sqrt(2)/2
>>> chebyshevu(3, chebyshevu_root(3, 2))
0
```

See also:

[chebyshevt](#) (page 536), [chebyshevt_root](#) (page 539), [chebyshevu](#) (page 538), [legendre](#) (page 540), [assoc_legendre](#) (page 540), [hermite](#) (page 541), [laguerre](#) (page 542), [assoc_laguerre](#) (page 543), [sympy.polys.orthopolys.jacobi_poly](#) (page 2440), [sympy.polys.orthopolys.gegenbauer_poly](#) (page 2440), [sympy.polys.orthopolys.chebyshevt_poly](#) (page 2439), [sympy.polys.orthopolys.chebyshevu_poly](#) (page 2439), [sympy.polys.orthopolys.hermite_poly](#) (page 2440), [sympy.polys.orthopolys.legendre_poly](#) (page 2440), [sympy.polys.orthopolys.laguerre_poly](#) (page 2441)

Legendre Polynomials

class `sympy.functions.special.polynomials.Legendre(n, x)`
`legendre(n, x)` gives the n th Legendre polynomial of x , $P_n(x)$

Explanation

The Legendre polynomials are orthogonal on $[-1, 1]$ with respect to the constant weight 1. They satisfy $P_n(1) = 1$ for all n ; further, P_n is odd for odd n and even for even n .

Examples

```
>>> from sympy import legendre, diff
>>> from sympy.abc import x, n
>>> legendre(0, x)
1
>>> legendre(1, x)
x
>>> legendre(2, x)
3*x**2/2 - 1/2
>>> legendre(n, x)
legendre(n, x)
>>> diff(legendre(n,x), x)
n*(x*legendre(n, x) - legendre(n - 1, x))/(x**2 - 1)
```

See also:

[jacobi](#) (page 533), [gegenbauer](#) (page 535), [chebyshevt](#) (page 536), [chebyshevt_root](#) (page 539), [chebyshevu](#) (page 538), [chebyshevu_root](#) (page 539), [assoc_legendre](#) (page 540), [hermite](#) (page 541), [laguerre](#) (page 542), [assoc_laguerre](#) (page 543), [sympy.polys.orthopolys.jacobi_poly](#) (page 2440), [sympy.polys.orthopolys.gegenbauer_poly](#) (page 2440), [sympy.polys.orthopolys.chebyshevt_poly](#) (page 2439), [sympy.polys.orthopolys.chebyshevu_poly](#) (page 2439), [sympy.polys.orthopolys.hermite_poly](#) (page 2440), [sympy.polys.orthopolys.legendre_poly](#) (page 2440), [sympy.polys.orthopolys.laguerre_poly](#) (page 2441)

References

[R473], [R474], [R475], [R476]

class `sympy.functions.special.polynomials.assoc_legendre(n, m, x)`
`assoc_legendre(n, m, x)` gives $P_n^m(x)$, where n and m are the degree and order or an expression which is related to the n th order Legendre polynomial, $P_n(x)$ in the following manner:

$$P_n^m(x) = (-1)^m (1 - x^2)^{\frac{m}{2}} \frac{d^m P_n(x)}{dx^m}$$

Explanation

Associated Legendre polynomials are orthogonal on $[-1, 1]$ with:

- weight = 1 for the same m and different n .
- weight = $\frac{1}{1-x^2}$ for the same n and different m .

Examples

```
>>> from sympy import assoc_legendre
>>> from sympy.abc import x, m, n
>>> assoc_legendre(0,0, x)
1
>>> assoc_legendre(1,0, x)
x
>>> assoc_legendre(1,1, x)
-sqrt(1 - x**2)
>>> assoc_legendre(n,m,x)
assoc_legendre(n, m, x)
```

See also:

[jacobi](#) (page 533), [gegenbauer](#) (page 535), [chebyshevt](#) (page 536), [chebyshevt_root](#) (page 539), [chebyshevu](#) (page 538), [chebyshevu_root](#) (page 539), [legendre](#) (page 540), [hermite](#) (page 541), [laguerre](#) (page 542), [assoc_laguerre](#) (page 543), [sympy.polys.orthopolys.jacobi_poly](#) (page 2440), [sympy.polys.orthopolys.gegenbauer_poly](#) (page 2440), [sympy.polys.orthopolys.chebyshevt_poly](#) (page 2439), [sympy.polys.orthopolys.chebyshevu_poly](#) (page 2439), [sympy.polys.orthopolys.hermite_poly](#) (page 2440), [sympy.polys.orthopolys.legendre_poly](#) (page 2440), [sympy.polys.orthopolys.laguerre_poly](#) (page 2441)

References

[R477], [R478], [R479], [R480]

Hermite Polynomials

class `sympy.functions.special.polynomials.hermite( $n, x$ )`
`hermite( $n, x$ )` gives the n th Hermite polynomial in x , $H_n(x)$

Explanation

The Hermite polynomials are orthogonal on $(-\infty, \infty)$ with respect to the weight $\exp(-x^2)$.

Examples

```
>>> from sympy import hermite, diff
>>> from sympy.abc import x, n
>>> hermite(0, x)
1
>>> hermite(1, x)
2*x
>>> hermite(2, x)
4*x**2 - 2
>>> hermite(n, x)
hermite(n, x)
>>> diff(hermite(n,x), x)
2*n*hermite(n - 1, x)
>>> hermite(n, -x)
(-1)**n*hermite(n, x)
```

See also:

[jacobi](#) (page 533), [gegenbauer](#) (page 535), [chebyshevt](#) (page 536), [chebyshevt_root](#) (page 539), [chebyshevu](#) (page 538), [chebyshevu_root](#) (page 539), [legendre](#) (page 540), [assoc_legendre](#) (page 540), [laguerre](#) (page 542), [assoc_laguerre](#) (page 543), [sympy.polys.orthopolys.jacobi_poly](#) (page 2440), [sympy.polys.orthopolys.gegenbauer_poly](#) (page 2440), [sympy.polys.orthopolys.chebyshevt_poly](#) (page 2439), [sympy.polys.orthopolys.chebyshevu_poly](#) (page 2439), [sympy.polys.orthopolys.hermite_poly](#) (page 2440), [sympy.polys.orthopolys.legendre_poly](#) (page 2440), [sympy.polys.orthopolys.laguerre_poly](#) (page 2441)

References

[R481], [R482], [R483]

Laguerre Polynomials

class sympy.functions.special.polynomials.**laguerre**(*n*, *x*)

Returns the *n*th Laguerre polynomial in *x*, $L_n(x)$.

Parameters

n : int

Degree of Laguerre polynomial. Must be $n \geq 0$.

Examples

```
>>> from sympy import laguerre, diff
>>> from sympy.abc import x, n
>>> laguerre(0, x)
1
>>> laguerre(1, x)
1 - x
>>> laguerre(2, x)
x**2/2 - 2*x + 1
>>> laguerre(3, x)
-x**3/6 + 3*x**2/2 - 3*x + 1
```

```
>>> laguerre(n, x)
laguerre(n, x)
```

```
>>> diff(laguerre(n, x), x)
-assoc_laguerre(n - 1, 1, x)
```

See also:

[jacobi](#) (page 533), [gegenbauer](#) (page 535), [chebyshevt](#) (page 536), [chebyshevt_root](#) (page 539), [chebyshevu](#) (page 538), [chebyshevu_root](#) (page 539), [legendre](#) (page 540), [assoc_legendre](#) (page 540), [hermite](#) (page 541), [assoc_laguerre](#) (page 543), [sympy.polys.orthopolys.jacobi_poly](#) (page 2440), [sympy.polys.orthopolys.gegenbauer_poly](#) (page 2440), [sympy.polys.orthopolys.chebyshevt_poly](#) (page 2439), [sympy.polys.orthopolys.chebyshevu_poly](#) (page 2439), [sympy.polys.orthopolys.hermite_poly](#) (page 2440), [sympy.polys.orthopolys.legendre_poly](#) (page 2440), [sympy.polys.orthopolys.laguerre_poly](#) (page 2441)

References

[R484], [R485], [R486], [R487]

class sympy.functions.special.polynomials.**assoc_laguerre**(*n*, *alpha*, *x*)

Returns the n th generalized Laguerre polynomial in x , $L_n(x)$.

Parameters

n : int

Degree of Laguerre polynomial. Must be $n \geq 0$.

alpha : Expr

Arbitrary expression. For $\alpha=0$ regular Laguerre polynomials will be generated.

Examples

```
>>> from sympy import assoc_laguerre, diff
>>> from sympy.abc import x, n, a
>>> assoc_laguerre(0, a, x)
1
>>> assoc_laguerre(1, a, x)
a - x + 1
>>> assoc_laguerre(2, a, x)
a**2/2 + 3*a/2 + x**2/2 + x*(-a - 2) + 1
>>> assoc_laguerre(3, a, x)
a**3/6 + a**2 + 11*a/6 - x**3/6 + x**2*(a/2 + 3/2) +
    x*(-a**2/2 - 5*a/2 - 3) + 1
```

```
>>> assoc_laguerre(n, a, 0)
binomial(a + n, a)
```

```
>>> assoc_laguerre(n, a, x)
assoc_laguerre(n, a, x)
```

```
>>> assoc_laguerre(n, 0, x)
laguerre(n, x)
```

```
>>> diff(assoc_laguerre(n, a, x), x)
-assoc_laguerre(n - 1, a + 1, x)
```

```
>>> diff(assoc_laguerre(n, a, x), a)
Sum(assoc_laguerre(_k, a, x)/(-a + n), (_k, 0, n - 1))
```

See also:

[jacobi](#) (page 533), [gegenbauer](#) (page 535), [chebyshevt](#) (page 536), [chebyshevt_root](#) (page 539), [chebyshevu](#) (page 538), [chebyshevu_root](#) (page 539), [legendre](#) (page 540), [assoc_legendre](#) (page 540), [hermite](#) (page 541), [laguerre](#) (page 542), [sympy.polys.orthopolys.jacobi_poly](#) (page 2440), [sympy.polys.orthopolys.gegenbauer_poly](#) (page 2440), [sympy.polys.orthopolys.chebyshevt_poly](#) (page 2439), [sympy.polys.orthopolys.chebyshevu_poly](#) (page 2439), [sympy.polys.orthopolys.hermite_poly](#) (page 2440), [sympy.polys.orthopolys.legendre_poly](#) (page 2440), [sympy.polys.orthopolys.laguerre_poly](#) (page 2441)

References

[R488], [R489], [R490], [R491]

Spherical Harmonics

class `sympy.functions.special.spherical_harmonics.Ynm(n, m, theta, phi)`
 Spherical harmonics defined as

$$Y_n^m(\theta, \varphi) := \sqrt{\frac{(2n+1)(n-m)!}{4\pi(n+m)!}} \exp(im\varphi) P_n^m(\cos(\theta))$$

Explanation

`Ynm()` gives the spherical harmonic function of order n and m in θ and φ , $Y_n^m(\theta, \varphi)$. The four parameters are as follows: $n \geq 0$ an integer and m an integer such that $-n \leq m \leq n$ holds. The two angles are real-valued with $\theta \in [0, \pi]$ and $\varphi \in [0, 2\pi]$.

Examples

```
>>> from sympy import Ynm, Symbol, simplify
>>> from sympy.abc import n,m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")
```

```
>>> Ynm(n, m, theta, phi)
Ynm(n, m, theta, phi)
```

Several symmetries are known, for the order:

```
>>> Ynm(n, -m, theta, phi)
(-1)**m*exp(-2*I*m*phi)*Ynm(n, m, theta, phi)
```

As well as for the angles:

```
>>> Ynm(n, m, -theta, phi)
Ynm(n, m, theta, phi)
```

```
>>> Ynm(n, m, theta, -phi)
exp(-2*I*m*phi)*Ynm(n, m, theta, phi)
```

For specific integers n and m we can evaluate the harmonics to more useful expressions:

```
>>> simplify(Ynm(0, 0, theta, phi).expand(func=True))
1/(2*sqrt(pi))
```

```
>>> simplify(Ynm(1, -1, theta, phi).expand(func=True))
sqrt(6)*exp(-I*phi)*sin(theta)/(4*sqrt(pi))
```

```
>>> simplify(Ynm(1, 0, theta, phi).expand(func=True))
sqrt(3)*cos(theta)/(2*sqrt(pi))
```

```
>>> simplify(Ynm(1, 1, theta, phi).expand(func=True))
-sqrt(6)*exp(I*phi)*sin(theta)/(4*sqrt(pi))
```

```
>>> simplify(Ynm(2, -2, theta, phi).expand(func=True))
sqrt(30)*exp(-2*I*phi)*sin(theta)**2/(8*sqrt(pi))
```

```
>>> simplify(Ynm(2, -1, theta, phi).expand(func=True))
sqrt(30)*exp(-I*phi)*sin(2*theta)/(8*sqrt(pi))
```

```
>>> simplify(Ynm(2, 0, theta, phi).expand(func=True))
sqrt(5)*(3*cos(theta)**2 - 1)/(4*sqrt(pi))
```

```
>>> simplify(Ynm(2, 1, theta, phi).expand(func=True))
-sqrt(30)*exp(I*phi)*sin(2*theta)/(8*sqrt(pi))
```

```
>>> simplify(Ynm(2, 2, theta, phi).expand(func=True))
sqrt(30)*exp(2*I*phi)*sin(theta)**2/(8*sqrt(pi))
```

We can differentiate the functions with respect to both angles:

```
>>> from sympy import Ynm, Symbol, diff
>>> from sympy.abc import n,m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")
```

```
>>> diff(Ynm(n, m, theta, phi), theta)
m*cot(theta)*Ynm(n, m, theta, phi) + sqrt((-m + n)*(m + n + 1))*exp(-
→ I*phi)*Ynm(n, m + 1, theta, phi)
```

```
>>> diff(Ynm(n, m, theta, phi), phi)
I*m*Ynm(n, m, theta, phi)
```

Further we can compute the complex conjugation:

```
>>> from sympy import Ynm, Symbol, conjugate
>>> from sympy.abc import n,m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")
```

```
>>> conjugate(Ynm(n, m, theta, phi))
(-1)**(2*m)*exp(-2*I*m*phi)*Ynm(n, m, theta, phi)
```

To get back the well known expressions in spherical coordinates, we use full expansion:

```
>>> from sympy import Ynm, Symbol, expand_func
>>> from sympy.abc import n,m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")
```

```
>>> expand_func(Ynm(n, m, theta, phi))
sqrt((2*n + 1)*factorial(-m + n)/factorial(m + n))*exp(I*m*phi)*assoc_
↳ legendre(n, m, cos(theta))/(2*sqrt(pi))
```

See also:

[Ynm_c](#) (page 547), [Znm](#) (page 547)

References

[R492], [R493], [R494], [R495]

`sympy.functions.special.spherical_harmonics.Ynm_c(n, m, theta, phi)`

Conjugate spherical harmonics defined as

$$\overline{Y_n^m(\theta, \varphi)} := (-1)^m Y_n^{-m}(\theta, \varphi).$$

Examples

```
>>> from sympy import Ynm_c, Symbol, simplify
>>> from sympy.abc import n,m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")
>>> Ynm_c(n, m, theta, phi)
(-1)**(2*m)*exp(-2*I*m*phi)*Ynm(n, m, theta, phi)
>>> Ynm_c(n, m, -theta, phi)
(-1)**(2*m)*exp(-2*I*m*phi)*Ynm(n, m, theta, phi)
```

For specific integers n and m we can evaluate the harmonics to more useful expressions:

```
>>> simplify(Ynm_c(0, 0, theta, phi).expand(func=True))
1/(2*sqrt(pi))
>>> simplify(Ynm_c(1, -1, theta, phi).expand(func=True))
sqrt(6)*exp(I*(-phi + 2*conjugate(phi)))*sin(theta)/(4*sqrt(pi))
```

See also:

[Ynm](#) (page 545), [Znm](#) (page 547)

References

[R496], [R497], [R498]

class `sympy.functions.special.spherical_harmonics.Znm(n, m, theta, phi)`

Real spherical harmonics defined as

$$Z_n^m(\theta, \varphi) := \begin{cases} \frac{Y_n^m(\theta, \varphi) + \overline{Y_n^m(\theta, \varphi)}}{\sqrt{2}} & m > 0 \\ Y_n^m(\theta, \varphi) & m = 0 \\ \frac{Y_n^m(\theta, \varphi) - \overline{Y_n^m(\theta, \varphi)}}{i\sqrt{2}} & m < 0 \end{cases}$$

which gives in simplified form

$$Z_n^m(\theta, \varphi) = \begin{cases} \frac{Y_n^m(\theta, \varphi) + (-1)^m Y_n^{-m}(\theta, \varphi)}{\sqrt{2}} & m > 0 \\ Y_n^m(\theta, \varphi) & m = 0 \\ \frac{Y_n^m(\theta, \varphi) - (-1)^m Y_n^{-m}(\theta, \varphi)}{i\sqrt{2}} & m < 0 \end{cases}$$

Examples

```
>>> from sympy import Znm, Symbol, simplify
>>> from sympy.abc import n, m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")
>>> Znm(n, m, theta, phi)
Znm(n, m, theta, phi)
```

For specific integers n and m we can evaluate the harmonics to more useful expressions:

```
>>> simplify(Znm(0, 0, theta, phi).expand(func=True))
1/(2*sqrt(pi))
>>> simplify(Znm(1, 1, theta, phi).expand(func=True))
-sqrt(3)*sin(theta)*cos(phi)/(2*sqrt(pi))
>>> simplify(Znm(2, 1, theta, phi).expand(func=True))
-sqrt(15)*sin(2*theta)*cos(phi)/(4*sqrt(pi))
```

See also:

[Ynm](#) (page 545), [Ynm_c](#) (page 547)

References

[R499], [R500], [R501]

Tensor Functions

`sympy.functions.special.tensor_functions.Eijk(*args, **kwargs)`

Represent the Levi-Civita symbol.

This is a compatibility wrapper to `LeviCivita()`.

See also:

[LeviCivita](#) (page 548)

`sympy.functions.special.tensor_functions.eval_levicivita(*args)`

Evaluate Levi-Civita symbol.

class `sympy.functions.special.tensor_functions.LeviCivita(*args)`

Represent the Levi-Civita symbol.

Explanation

For even permutations of indices it returns 1, for odd permutations -1, and for everything else (a repeated index) it returns 0.

Thus it represents an alternating pseudotensor.

Examples

```
>>> from sympy import LeviCivita
>>> from sympy.abc import i, j, k
>>> LeviCivita(1, 2, 3)
1
>>> LeviCivita(1, 3, 2)
-1
>>> LeviCivita(1, 2, 2)
0
>>> LeviCivita(i, j, k)
LeviCivita(i, j, k)
>>> LeviCivita(i, j, i)
0
```

See also:

[Eijk](#) (page 548)

class sympy.functions.special.tensor_functions.**KroneckerDelta**(*i, j*,
delta_range=None)

The discrete, or Kronecker, delta function.

Parameters

i : Number, Symbol

The first index of the delta function.

j : Number, Symbol

The second index of the delta function.

Explanation

A function that takes in two integers i and j . It returns 0 if i and j are not equal, or it returns 1 if i and j are equal.

Examples

An example with integer indices:

```
>>> from sympy import KroneckerDelta
>>> KroneckerDelta(1, 2)
0
>>> KroneckerDelta(3, 3)
1
```

Symbolic indices:

```
>>> from sympy.abc import i, j, k
>>> KroneckerDelta(i, j)
KroneckerDelta(i, j)
>>> KroneckerDelta(i, i)
1
>>> KroneckerDelta(i, i + 1)
0
>>> KroneckerDelta(i, i + 1 + k)
KroneckerDelta(i, i + k + 1)
```

See also:

[eval](#) (page 550), [DiracDelta](#) (page 450)

References

[R502]

classmethod `eval(i, j, delta_range=None)`

Evaluates the discrete delta function.

Examples

```
>>> from sympy import KroneckerDelta
>>> from sympy.abc import i, j, k
```

```
>>> KroneckerDelta(i, j)
KroneckerDelta(i, j)
>>> KroneckerDelta(i, i)
1
>>> KroneckerDelta(i, i + 1)
0
>>> KroneckerDelta(i, i + 1 + k)
KroneckerDelta(i, i + k + 1)
```

indirect doctest

property `indices_contain_equal_information`

Returns True if indices are either both above or below fermi.

Examples

```
>>> from sympy import KroneckerDelta, Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, q).indices_contain_equal_information
True
>>> KroneckerDelta(p, q+1).indices_contain_equal_information
True
>>> KroneckerDelta(i, p).indices_contain_equal_information
False
```

property `is_above_fermi`

True if Delta can be non-zero above fermi.

Examples

```
>>> from sympy import KroneckerDelta, Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_above_fermi
True
>>> KroneckerDelta(p, i).is_above_fermi
False
>>> KroneckerDelta(p, q).is_above_fermi
True
```

See also:

[`is\_below\_fermi`](#) (page 551), [`is\_only\_below\_fermi`](#) (page 552),
[`is\_only\_above\_fermi`](#) (page 552)

property `is_below_fermi`

True if Delta can be non-zero below fermi.

Examples

```
>>> from sympy import KroneckerDelta, Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_below_fermi
False
>>> KroneckerDelta(p, i).is_below_fermi
True
```

(continues on next page)

(continued from previous page)

```
>>> KroneckerDelta(p, q).is_below_fermi
True
```

See also:

[*is_above_fermi*](#) (page 551), [*is_only_above_fermi*](#) (page 552),
[*is_only_below_fermi*](#) (page 552)

property `is_only_above_fermi`

True if Delta is restricted to above fermi.

Examples

```
>>> from sympy import KroneckerDelta, Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_only_above_fermi
True
>>> KroneckerDelta(p, q).is_only_above_fermi
False
>>> KroneckerDelta(p, i).is_only_above_fermi
False
```

See also:

[*is_above_fermi*](#) (page 551), [*is_below_fermi*](#) (page 551), [*is_only_below_fermi*](#)
(page 552)

property `is_only_below_fermi`

True if Delta is restricted to below fermi.

Examples

```
>>> from sympy import KroneckerDelta, Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, i).is_only_below_fermi
True
>>> KroneckerDelta(p, q).is_only_below_fermi
False
>>> KroneckerDelta(p, a).is_only_below_fermi
False
```

See also:

[*is_above_fermi*](#) (page 551), [*is_below_fermi*](#) (page 551), [*is_only_above_fermi*](#)
(page 552)

property killable_index

Returns the index which is preferred to substitute in the final expression.

Explanation

The index to substitute is the index with less information regarding fermi level. If indices contain the same information, 'a' is preferred before 'b'.

Examples

```
>>> from sympy import KroneckerDelta, Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> j = Symbol('j', below_fermi=True)
>>> p = Symbol('p')
>>> KroneckerDelta(p, i).killable_index
p
>>> KroneckerDelta(p, a).killable_index
p
>>> KroneckerDelta(i, j).killable_index
j
```

See also:

[*preferred_index*](#) (page 553)

property preferred_index

Returns the index which is preferred to keep in the final expression.

Explanation

The preferred index is the index with more information regarding fermi level. If indices contain the same information, 'a' is preferred before 'b'.

Examples

```
>>> from sympy import KroneckerDelta, Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> j = Symbol('j', below_fermi=True)
>>> p = Symbol('p')
>>> KroneckerDelta(p, i).preferred_index
i
>>> KroneckerDelta(p, a).preferred_index
a
>>> KroneckerDelta(i, j).preferred_index
i
```

See also:

[*killable_index*](#) (page 552)

Integrals

This module documentation contains details about Meijer G-functions and SymPy integrals. functions.

Contents

Computing Integrals using Meijer G-Functions

This text aims to describe in some detail the steps (and subtleties) involved in using Meijer G-functions for computing definite and indefinite integrals. We shall ignore proofs completely.

Overview

The algorithm to compute $\int f(x)dx$ or $\int_0^\infty f(x)dx$ generally consists of three steps:

1. Rewrite the integrand using Meijer G-functions (one or sometimes two).
2. Apply an integration theorem, to get the answer (usually expressed as another G-function).
3. Expand the result in named special functions.

Step (3) is implemented in the function `hyperexpand` (q.v.). Steps (1) and (2) are described below. Moreover, G-functions are usually branched. Thus our treatment of branched functions is described first.

Some other integrals (e.g. $\int_{-\infty}^\infty$) can also be computed by first recasting them into one of the above forms. There is a lot of choice involved here, and the algorithm is heuristic at best.

Polar Numbers and Branched Functions

Both Meijer G-Functions and Hypergeometric functions are typically branched (possible branchpoints being $0, \pm 1, \infty$). This is not very important when e.g. expanding a single hypergeometric function into named special functions, since sorting out the branches can be left to the human user. However this algorithm manipulates and transforms G-functions, and to do this correctly it needs at least some crude understanding of the branchings involved.

To begin, we consider the set $S = \{(r, \theta) : r > 0, \theta \in \mathbb{R}\}$. We have a map $p : S \rightarrow \mathbb{C} - \{0\}, (r, \theta) \mapsto re^{i\theta}$. Decreeing this to be a local biholomorphism gives S both a topology and a complex structure. This Riemann Surface is usually referred to as the Riemann Surface of the logarithm, for the following reason: We can define maps $\text{Exp} : \mathbb{C} \rightarrow S, (x + iy) \mapsto (\exp(x), y)$ and $\text{Log} : S \rightarrow \mathbb{C}, (e^x, y) \mapsto x + iy$. These can both be shown to be holomorphic, and are indeed mutual inverses.

We also sometimes formally attach a point “zero” (0) to S and denote the resulting object S_0 . Notably there is no complex structure defined near 0 . A fundamental system of neighbourhoods is given by $\{\text{Exp}(z) : \Re(z) < k\}$, which at least defines a topology. Elements of S_0 shall be called polar numbers. We further define functions $\text{Arg} : S \rightarrow \mathbb{R}, (r, \theta) \mapsto \theta$ and $|\cdot| : S_0 \rightarrow \mathbb{R}_{>0}, (r, \theta) \mapsto r$. These have evident meaning and are both continuous everywhere.

Using these maps many operations can be extended from $\mathbb{C}$ to S . We define $\text{Exp}(a)\text{Exp}(b) = \text{Exp}(a + b)$ for $a, b \in \mathbb{C}$, also for $a \in S$ and $b \in \mathbb{C}$ we define $a^b = \text{Exp}(b \text{Log}(a))$. It can be checked easily that using these definitions, many algebraic properties holding for positive reals (e.g.

$(ab)^c = a^c b^c$ which hold in $\mathbb{C}$ only for some numbers (because of branch cuts) hold indeed for all polar numbers.

As one peculiarity it should be mentioned that addition of polar numbers is not usually defined. However, formal sums of polar numbers can be used to express branching behaviour. For example, consider the functions $F(z) = \sqrt{1+z}$ and $G(a,b) = \sqrt{a+b}$, where a, b, z are polar numbers. The general rule is that functions of a single polar variable are defined in such a way that they are continuous on circles, and agree with the usual definition for positive reals. Thus if $S(z)$ denotes the standard branch of the square root function on $\mathbb{C}$, we are forced to define

$$F(z) = \begin{cases} S(p(z)) & : |z| < 1 \\ S(p(z)) & : -\pi < \text{Arg}(z) + 4\pi n \leq \pi \text{ for some } n \in \mathbb{Z} \\ -S(p(z)) & : \text{else} \end{cases}$$

(We are omitting $|z| = 1$ here, this does not matter for integration.) Finally we define $G(a,b) = \sqrt{a}F(b/a)$.

Representing Branched Functions on the Argand Plane

Suppose $f : \mathcal{S} \rightarrow \mathbb{C}$ is a holomorphic function. We wish to define a function F on (part of) the complex numbers $\mathbb{C}$ that represents f as closely as possible. This process is known as “introducing branch cuts”. In our situation, there is actually a canonical way of doing this (which is adhered to in all of SymPy), as follows: Introduce the “cut complex plane” $C = \mathbb{C} \setminus \mathbb{R}_{\leq 0}$. Define a function $l : C \rightarrow \mathcal{S}$ via $re^{i\theta} \mapsto r \text{Exp}(i\theta)$. Here $r > 0$ and $-\pi < \theta \leq \pi$. Then l is holomorphic, and we define $G = f \circ l$. This called “lifting to the principal branch” throughout the SymPy documentation.

Table Lookups and Inverse Mellin Transforms

Suppose we are given an integrand $f(x)$ and are trying to rewrite it as a single G-function. To do this, we first split $f(x)$ into the form $x^s g(x)$ (where $g(x)$ is supposed to be simpler than $f(x)$). This is because multiplicative powers can be absorbed into the G-function later. This splitting is done by `_split_mul(f, x)`. Then we assemble a tuple of functions that occur in f (e.g. if $f(x) = e^x \cos x$, we would assemble the tuple $(\cos, \exp)$). This is done by the function `_mytype(f, x)`. Next we index a lookup table (created using `_create_lookup_table()`) with this tuple. This (hopefully) yields a list of Meijer G-function formulae involving these functions, we then pattern-match all of them. If one fits, we were successful, otherwise not and we have to try something else.

Suppose now we want to rewrite as a product of two G-functions. To do this, we (try to) find all inequivalent ways of splitting $f(x)$ into a product $f_1(x)f_2(x)$. We could try these splittings in any order, but it is often a good idea to minimize (a) the number of powers occurring in $f_i(x)$ and (b) the number of different functions occurring in $f_i(x)$. Thus given e.g. $f(x) = \sin x e^x \sin 2x$ we should try $f_1(x) = \sin x \sin 2x$, $f_2(x) = e^x$ first. All of this is done by the function `_mul_as_two_parts(f)`.

Finally, we can try a recursive Mellin transform technique. Since the Meijer G-function is defined essentially as a certain inverse mellin transform, if we want to write a function $f(x)$ as a G-function, we can compute its mellin transform $F(s)$. If $F(s)$ is in the right form, the G-function expression can be read off. This technique generalises many standard rewritings, e.g. $e^{ax}e^{bx} = e^{(a+b)x}$.

One twist is that some functions don't have mellin transforms, even though they can be written as G-functions. This is true for example for $f(x) = e^x \sin x$ (the function grows too rapidly to have a mellin transform). However if the function is recognised to be analytic, then we can try to compute the mellin-transform of $f(ax)$ for a parameter a , and deduce the G-function expression by analytic continuation. (Checking for analyticity is easy. Since we can only deal with a certain subset of functions anyway, we only have to filter out those which are not analytic.)

The function `_rewrite_single` does the table lookup and recursive mellin transform. The functions `_rewrite1` and `_rewrite2` respectively use above-mentioned helpers and `_rewrite_single` to rewrite their argument as respectively one or two G-functions.

Applying the Integral Theorems

If the integrand has been recast into G-functions, evaluating the integral is relatively easy. We first do some substitutions to reduce e.g. the exponent of the argument of the G-function to unity (see `_rewrite_saxena_1` and `_rewrite_saxena`, respectively, for one or two G-functions). Next we go through a list of conditions under which the integral theorem applies. It can fail for basically two reasons: either the integral does not exist, or the manipulations in deriving the theorem may not be allowed (for more details, see this [\[BlogPost\]](#)).

Sometimes this can be remedied by reducing the argument of the G-functions involved. For example it is clear that the G-function representing e^z satisfies $G(\text{Exp}(2\pi i)z) = G(z)$ for all $z \in \mathcal{S}$. The function `meijerg.get_period()` can be used to discover this, and the function `principal_branch(z, period)` in `functions/elementary/complexes.py` can be used to exploit the information. This is done transparently by the integration code.

The G-Function Integration Theorems

This section intends to display in detail the definite integration theorems used in the code. The following two formulae go back to Meijer (In fact he proved more general formulae; indeed in the literature formulae are usually stated in more general form. However it is very easy to deduce the general formulae from the ones we give here. It seemed best to keep the theorems as simple as possible, since they are very complicated anyway.):

1.

$$\int_0^\infty G_{p,q}^{m,n} \left(\begin{matrix} a_1, \dots, a_p \\ b_1, \dots, b_q \end{matrix} \middle| \eta x \right) dx = \frac{\prod_{j=1}^m \Gamma(b_j + 1) \prod_{j=1}^n \Gamma(-a_j)}{\eta \prod_{j=m+1}^q \Gamma(-b_j) \prod_{j=n+1}^p \Gamma(a_j + 1)}$$

2.

$$\int_0^\infty G_{u,v}^{s,t} \left(\begin{matrix} c_1, \dots, c_u \\ d_1, \dots, d_v \end{matrix} \middle| \sigma x \right) G_{p,q}^{m,n} \left(\begin{matrix} a_1, \dots, a_p \\ b_1, \dots, b_q \end{matrix} \middle| \omega x \right) dx = G_{v+p,u+q}^{m+t,n+s} \left(\begin{matrix} a_1, \dots, a_n, -d_1, \dots, -d_v, a_{n+1}, \dots, a_p \\ b_1, \dots, b_m, -c_1, \dots, -c_u, b_{m+1}, \dots, b_q \end{matrix} \middle| \frac{\omega}{\sigma} \right)$$

The more interesting question is under what conditions these formulae are valid. Below we detail the conditions implemented in SymPy. They are an amalgamation of conditions found in [\[Prudnikov1990\]](#) and [\[Luke1969\]](#); please let us know if you find any errors.

Conditions of Convergence for Integral (1)

We can without loss of generality assume $p \leq q$, since the G-functions of indices m, n, p, q and of indices n, m, q, p can be related easily (see e.g. [Luke1969], section 5.3). We introduce the following notation:

$$\begin{aligned}\xi &= m + n - p \\ \delta &= m + n - \frac{p+q}{2} \\ C_3 &: -\Re(b_j) < 1 \text{ for } j = 1, \dots, m \\ &\quad 0 < -\Re(a_j) \text{ for } j = 1, \dots, n \\ C_3^* &: -\Re(b_j) < 1 \text{ for } j = 1, \dots, q \\ &\quad 0 < -\Re(a_j) \text{ for } j = 1, \dots, p \\ C_4 &: -\Re(\delta) + \frac{q+1-p}{2} > q-p\end{aligned}$$

The convergence conditions will be detailed in several “cases”, numbered one to five. For later use it will be helpful to separate conditions “at infinity” from conditions “at zero”. By conditions “at infinity” we mean conditions that only depend on the behaviour of the integrand for large, positive values of x , whereas by conditions “at zero” we mean conditions that only depend on the behaviour of the integrand on $(0, \epsilon)$ for any $\epsilon > 0$. Since all our conditions are specified in terms of parameters of the G-functions, this distinction is not immediately visible. They are, however, of very distinct character mathematically; the conditions at infinity being in particular much harder to control.

In order for the integral theorem to be valid, conditions n “at zero” and “at infinity” both have to be fulfilled, for some n .

These are the conditions “at infinity”:

1.

$$\delta > 0 \wedge |\arg(\eta)| < \delta\pi \wedge (A \vee B \vee C),$$

where

$$\begin{aligned}A &= 1 \leq n \wedge p < q \wedge 1 \leq m \\ B &= 1 \leq p \wedge 1 \leq m \wedge q = p+1 \wedge \neg(n=0 \wedge m=p+1) \\ C &= 1 \leq n \wedge q = p \wedge |\arg(\eta)| \neq (\delta - 2k)\pi \text{ for } k = 0, 1, \dots \left\lceil \frac{\delta}{2} \right\rceil.\end{aligned}$$

2.

$$n = 0 \wedge p+1 \leq m \wedge |\arg(\eta)| < \delta\pi$$

3.

$$(p < q \wedge 1 \leq m \wedge \delta > 0 \wedge |\arg(\eta)| = \delta\pi) \vee (p \leq q-2 \wedge \delta = 0 \wedge \arg(\eta) = 0)$$

4.

$$p = q \wedge \delta = 0 \wedge \arg(\eta) = 0 \wedge \eta \neq 0 \wedge \Re\left(\sum_{j=1}^p b_j - a_j\right) < 0$$

5.

$$\delta > 0 \wedge |\arg(\eta)| < \delta\pi$$

And these are the conditions “at zero”:

1.

$$\eta \neq 0 \wedge C_3$$

2.

$$C_3$$

3.

$$C_3 \wedge C_4$$

4.

$$C_3$$

5.

$$C_3$$

Conditions of Convergence for Integral (2)

We introduce the following notation:

$$b^* = s + t - \frac{u + v}{2}$$

$$c^* = m + n - \frac{p + q}{2}$$

$$\rho = \sum_{j=1}^v d_j - \sum_{j=1}^u c_j + \frac{u - v}{2} + 1$$

$$\mu = \sum_{j=1}^q b_j - \sum_{j=1}^p a_j + \frac{p - q}{2} + 1$$

$$\phi = q - p - \frac{u - v}{2} + 1$$

$$\eta = 1 - (v - u) - \mu - \rho$$

$$\psi = \frac{\pi(q - m - n) + |\arg(\omega)|}{q - p}$$

$$\theta = \frac{\pi(v - s - t) + |\arg(\sigma)|}{v - u}$$

$$\lambda_c = (q - p)|\omega|^{1/(q-p)} \cos \psi + (v - u)|\sigma|^{1/(v-u)} \cos \theta$$

$$\lambda_{s0}(c_1, c_2) = c_1(q - p)|\omega|^{1/(q-p)} \sin \psi + c_2(v - u)|\sigma|^{1/(v-u)} \sin \theta$$

$$\lambda_s = \begin{cases} \lambda_{s0}(-1, -1) \lambda_{s0}(1, 1) & \text{for } \arg(\omega) = 0 \wedge \arg(\sigma) = 0 \\ \lambda_{s0}(\text{sign}(\arg(\omega)), -1) \lambda_{s0}(\text{sign}(\arg(\omega)), 1) & \text{for } \arg(\omega) \neq 0 \wedge \arg(\sigma) = 0 \\ \lambda_{s0}(-1, \text{sign}(\arg(\sigma))) \lambda_{s0}(1, \text{sign}(\arg(\sigma))) & \text{for } \arg(\omega) = 0 \wedge \arg(\sigma) \neq 0 \\ \lambda_{s0}(\text{sign}(\arg(\omega)), \text{sign}(\arg(\sigma))) & \text{otherwise} \end{cases}$$

$$z_0 = \frac{\omega}{\sigma} e^{-i\pi(b^* + c^*)}$$

$$z_1 = \frac{\sigma}{\omega} e^{-i\pi(b^* + c^*)}$$

The following conditions will be helpful:

$$\begin{aligned} C_1 : & (a_i - b_j \notin \mathbb{Z}_{>0} \text{ for } i = 1, \dots, n, j = 1, \dots, m) \\ & \wedge (c_i - d_j \notin \mathbb{Z}_{>0} \text{ for } i = 1, \dots, t, j = 1, \dots, s) \\ C_2 : & \Re(1 + b_i + d_j) > 0 \text{ for } i = 1, \dots, m, j = 1, \dots, s \\ C_3 : & \Re(a_i + c_j) < 1 \text{ for } i = 1, \dots, n, j = 1, \dots, t \\ C_4 : & (p - q)\Re(c_i) - \Re(\mu) > -\frac{3}{2} \text{ for } i = 1, \dots, t \\ C_5 : & (p - q)\Re(1 + d_i) - \Re(\mu) > -\frac{3}{2} \text{ for } i = 1, \dots, s \\ C_6 : & (u - v)\Re(a_i) - \Re(\rho) > -\frac{3}{2} \text{ for } i = 1, \dots, n \\ C_7 : & (u - v)\Re(1 + b_i) - \Re(\rho) > -\frac{3}{2} \text{ for } i = 1, \dots, m \\ C_8 : & 0 < |\phi| + 2\Re((\mu - 1)(-u + v) + (-p + q)(\rho - 1) + (-p + q)(-u + v)) \\ C_9 : & 0 < |\phi| - 2\Re((\mu - 1)(-u + v) + (-p + q)(\rho - 1) + (-p + q)(-u + v)) \\ C_{10} : & |\arg(\sigma)| < \pi b^* \\ C_{11} : & |\arg(\sigma)| = \pi b^* \\ C_{12} : & |\arg(\omega)| < c^* \pi \\ C_{13} : & |\arg(\omega)| = c^* \pi \\ C_{14}^1 : & (z_0 \neq 1 \wedge |\arg(1 - z_0)| < \pi) \vee (z_0 = 1 \wedge \Re(\mu + \rho - u + v) < 1) \\ C_{14}^2 : & (z_1 \neq 1 \wedge |\arg(1 - z_1)| < \pi) \vee (z_1 = 1 \wedge \Re(\mu + \rho - p + q) < 1) \\ C_{14} : & \phi = 0 \wedge b^* + c^* \leq 1 \wedge (C_{14}^1 \vee C_{14}^2) \\ C_{15} : & \lambda_c > 0 \vee (\lambda_c = 0 \wedge \lambda_s \neq 0 \wedge \Re(\eta) > -1) \vee (\lambda_c = 0 \wedge \lambda_s = 0 \wedge \Re(\eta) > 0) \\ C_{16} : & \int_0^\infty G_{u,v}^{s,t}(\sigma x) dx \text{ converges at infinity} \\ C_{17} : & \int_0^\infty G_{p,q}^{m,n}(\omega x) dx \text{ converges at infinity} \end{aligned}$$

Note that C_{16} and C_{17} are the reason we split the convergence conditions for integral (1).

With this notation established, the implemented convergence conditions can be enumerated as follows:

1.

$$mnst \neq 0 \wedge 0 < b^* \wedge 0 < c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_{10} \wedge C_{12}$$

2.

$$u = v \wedge b^* = 0 \wedge 0 < c^* \wedge 0 < \sigma \wedge \Re \rho < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_{12}$$

3.

$$p = q \wedge u = v \wedge b^* = 0 \wedge c^* = 0 \wedge 0 < \sigma \wedge 0 < \omega \wedge \Re \mu < 1 \wedge \Re \rho < 1 \wedge \sigma \neq \omega \wedge C_1 \wedge C_2 \wedge C_3$$

4.

$$p = q \wedge u = v \wedge b^* = 0 \wedge c^* = 0 \wedge 0 < \sigma \wedge 0 < \omega \wedge \Re(\mu + \rho) < 1 \wedge \omega \neq \sigma \wedge C_1 \wedge C_2 \wedge C_3$$

5.

$$p = q \wedge u = v \wedge b^* = 0 \wedge c^* = 0 \wedge 0 < \sigma \wedge 0 < \omega \wedge \Re(\mu + \rho) < 1 \wedge \omega \neq \sigma \wedge C_1 \wedge C_2 \wedge C_3$$

6.

$$q < p \wedge 0 < s \wedge 0 < b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_5 \wedge C_{10} \wedge C_{13}$$

7.

$$p < q \wedge 0 < t \wedge 0 < b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_{10} \wedge C_{13}$$

8.

$$v < u \wedge 0 < m \wedge 0 < c^* \wedge 0 \leq b^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_7 \wedge C_{11} \wedge C_{12}$$

9.

$$u < v \wedge 0 < n \wedge 0 < c^* \wedge 0 \leq b^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_6 \wedge C_{11} \wedge C_{12}$$

10.

$$q < p \wedge u = v \wedge b^* = 0 \wedge 0 \leq c^* \wedge 0 < \sigma \wedge \Re \rho < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_5 \wedge C_{13}$$

11.

$$p < q \wedge u = v \wedge b^* = 0 \wedge 0 \leq c^* \wedge 0 < \sigma \wedge \Re \rho < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_{13}$$

12.

$$p = q \wedge v < u \wedge 0 \leq b^* \wedge c^* = 0 \wedge 0 < \omega \wedge \Re \mu < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_7 \wedge C_{11}$$

13.

$$p = q \wedge u < v \wedge 0 \leq b^* \wedge c^* = 0 \wedge 0 < \omega \wedge \Re \mu < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_6 \wedge C_{11}$$

14.

$$p < q \wedge v < u \wedge 0 \leq b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_7 \wedge C_{11} \wedge C_{13}$$

15.

$$q < p \wedge u < v \wedge 0 \leq b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_5 \wedge C_6 \wedge C_{11} \wedge C_{13}$$

16.

$$q < p \wedge v < u \wedge 0 \leq b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_5 \wedge C_7 \wedge C_8 \wedge C_{11} \wedge C_{13} \wedge C_{14}$$

17.

$$p < q \wedge u < v \wedge 0 \leq b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_6 \wedge C_9 \wedge C_{11} \wedge C_{13} \wedge C_{14}$$

18.

$$t = 0 \wedge 0 < s \wedge 0 < b^* \wedge 0 < \phi \wedge C_1 \wedge C_2 \wedge C_{10}$$

19.

$$s = 0 \wedge 0 < t \wedge 0 < b^* \wedge \phi < 0 \wedge C_1 \wedge C_3 \wedge C_{10}$$

20.

$$n = 0 \wedge 0 < m \wedge 0 < c^* \wedge \phi < 0 \wedge C_1 \wedge C_2 \wedge C_{12}$$

21.

$$m = 0 \wedge 0 < n \wedge 0 < c^* \wedge 0 < \phi \wedge C_1 \wedge C_3 \wedge C_{12}$$

22.

$$st = 0 \wedge 0 < b^* \wedge 0 < c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_{10} \wedge C_{12}$$

23.

$$mn = 0 \wedge 0 < b^* \wedge 0 < c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_{10} \wedge C_{12}$$

24.

$$p < m + n \wedge t = 0 \wedge \phi = 0 \wedge 0 < s \wedge 0 < b^* \wedge c^* < 0 \wedge |\arg(\omega)| < \pi(m + n - p + 1) \wedge C_1 \wedge C_2 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

25.

$$q < m + n \wedge s = 0 \wedge \phi = 0 \wedge 0 < t \wedge 0 < b^* \wedge c^* < 0 \wedge |\arg(\omega)| < \pi(m + n - q + 1) \wedge C_1 \wedge C_3 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

26.

$$p = q - 1 \wedge t = 0 \wedge \phi = 0 \wedge 0 < s \wedge 0 < b^* \wedge 0 \leq c^* \wedge \pi c^* < |\arg(\omega)| \wedge C_1 \wedge C_2 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

27.

$$p = q + 1 \wedge s = 0 \wedge \phi = 0 \wedge 0 < t \wedge 0 < b^* \wedge 0 \leq c^* \wedge \pi c^* < |\arg(\omega)| \wedge C_1 \wedge C_3 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

28.

$$p < q - 1 \wedge t = 0 \wedge \phi = 0 \wedge 0 < s \wedge 0 < b^* \wedge 0 \leq c^* \wedge \pi c^* < |\mathbf{arg}(\omega)| \wedge |\mathbf{arg}(\omega)| < \pi(m + n - p + 1) \wedge C_1 \wedge C_2 \wedge C_{10} \wedge$$

29.

$$q + 1 < p \wedge s = 0 \wedge \phi = 0 \wedge 0 < t \wedge 0 < b^* \wedge 0 \leq c^* \wedge \pi c^* < |\mathbf{arg}(\omega)| \wedge |\mathbf{arg}(\omega)| < \pi(m + n - q + 1) \wedge C_1 \wedge C_3 \wedge C_{10} \wedge$$

30.

$$n = 0 \wedge \phi = 0 \wedge 0 < s + t \wedge 0 < m \wedge 0 < c^* \wedge b^* < 0 \wedge |\mathbf{arg}(\sigma)| < \pi(s + t - u + 1) \wedge C_1 \wedge C_2 \wedge C_{12} \wedge C_{14} \wedge C_{15}$$

31.

$$m = 0 \wedge \phi = 0 \wedge v < s + t \wedge 0 < n \wedge 0 < c^* \wedge b^* < 0 \wedge |\mathbf{arg}(\sigma)| < \pi(s + t - v + 1) \wedge C_1 \wedge C_3 \wedge C_{12} \wedge C_{14} \wedge C_{15}$$

32.

$$n = 0 \wedge \phi = 0 \wedge u = v - 1 \wedge 0 < m \wedge 0 < c^* \wedge 0 \leq b^* \wedge \pi b^* < |\mathbf{arg}(\sigma)| \wedge |\mathbf{arg}(\sigma)| < \pi(b^* + 1) \wedge C_1 \wedge C_2 \wedge C_{12} \wedge C_{14} \wedge$$

33.

$$m = 0 \wedge \phi = 0 \wedge u = v + 1 \wedge 0 < n \wedge 0 < c^* \wedge 0 \leq b^* \wedge \pi b^* < |\mathbf{arg}(\sigma)| \wedge |\mathbf{arg}(\sigma)| < \pi(b^* + 1) \wedge C_1 \wedge C_3 \wedge C_{12} \wedge C_{14} \wedge$$

34.

$$n = 0 \wedge \phi = 0 \wedge u < v - 1 \wedge 0 < m \wedge 0 < c^* \wedge 0 \leq b^* \wedge \pi b^* < |\mathbf{arg}(\sigma)| \wedge |\mathbf{arg}(\sigma)| < \pi(s + t - u + 1) \wedge C_1 \wedge C_2 \wedge C_{12} \wedge$$

35.

$$m = 0 \wedge \phi = 0 \wedge v + 1 < u \wedge 0 < n \wedge 0 < c^* \wedge 0 \leq b^* \wedge \pi b^* < |\mathbf{arg}(\sigma)| \wedge |\mathbf{arg}(\sigma)| < \pi(s + t - v + 1) \wedge C_1 \wedge C_3 \wedge C_{12} \wedge$$

36.

$$C_{17} \wedge t = 0 \wedge u < s \wedge 0 < b^* \wedge C_{10} \wedge C_1 \wedge C_2 \wedge C_3$$

37.

$$C_{17} \wedge s = 0 \wedge v < t \wedge 0 < b^* \wedge C_{10} \wedge C_1 \wedge C_2 \wedge C_3$$

38.

$$C_{16} \wedge n = 0 \wedge p < m \wedge 0 < c^* \wedge C_{12} \wedge C_1 \wedge C_2 \wedge C_3$$

39.

$$C_{16} \wedge m = 0 \wedge q < n \wedge 0 < c^* \wedge C_{12} \wedge C_1 \wedge C_2 \wedge C_3$$

The Inverse Laplace Transform of a G-function

The inverse laplace transform of a Meijer G-function can be expressed as another G-function. This is a fairly versatile method for computing this transform. However, I could not find the details in the literature, so I work them out here. In [Luke1969], section 5.6.3, there is a formula for the inverse Laplace transform of a G-function of argument bz , and convergence conditions are also given. However, we need a formula for argument bz^a for rational a .

We are asked to compute

$$f(t) = \frac{1}{2\pi i} \int_{c-i\infty}^{c+i\infty} e^{zt} G(bz^a) dz,$$

for positive real t . Three questions arise:

1. When does this integral converge?
2. How can we compute the integral?
3. When is our computation valid?

How to compute the integral

We shall work formally for now. Denote by $\Delta(s)$ the product of gamma functions appearing in the definition of G , so that

$$G(z) = \frac{1}{2\pi i} \int_L \Delta(s) z^s ds.$$

Thus

$$f(t) = \frac{1}{(2\pi i)^2} \int_{c-i\infty}^{c+i\infty} \int_L e^{zt} \Delta(s) b^s z^{as} ds dz.$$

We interchange the order of integration to get

$$f(t) = \frac{1}{2\pi i} \int_L b^s \Delta(s) \int_{c-i\infty}^{c+i\infty} e^{zt} z^{as} \frac{dz}{2\pi i} ds.$$

The inner integral is easily seen to be $\frac{1}{\Gamma(-as)} \frac{1}{t^{1+as}}$. (Using Cauchy's theorem and Jordan's lemma deform the contour to run from $-\infty$ to $-\infty$, encircling 0 once in the negative sense. For as real and greater than one, this contour can be pushed onto the negative real axis and the integral is recognised as a product of a sine and a gamma function. The formula is then proved using the functional equation of the gamma function, and extended to the entire domain of convergence of the original integral by appealing to analytic continuation.) Hence we find

$$f(t) = \frac{1}{t} \frac{1}{2\pi i} \int_L \Delta(s) \frac{1}{\Gamma(-as)} \left(\frac{b}{t^a}\right)^s ds,$$

which is a so-called Fox H function (of argument $\frac{b}{t^a}$). For rational a , this can be expressed as a Meijer G-function using the gamma function multiplication theorem.

When this computation is valid

There are a number of obstacles in this computation. Interchange of integrals is only valid if all integrals involved are absolutely convergent. In particular the inner integral has to converge. Also, for our identification of the final integral as a Fox H / Meijer G-function to be correct, the poles of the newly obtained gamma function must be separated properly.

It is easy to check that the inner integral converges absolutely for $\Re(as) < -1$. Thus the contour L has to run left of the line $\Re(as) = -1$. Under this condition, the poles of the newly-introduced gamma function are separated properly.

It remains to observe that the Meijer G-function is an analytic, unbranched function of its parameters, and of the coefficient b . Hence so is $f(t)$. Thus the final computation remains valid as long as the initial integral converges, and if there exists a changed set of parameters where the computation is valid. If we assume w.l.o.g. that $a > 0$, then the latter condition is fulfilled if G converges along contours (2) or (3) of [Luke1969], section 5.2, i.e. either $\delta \geq \frac{a}{2}$ or $p \geq 1, p \geq q$.

When the integral exists

Using [Luke1969], section 5.10, for any given meijer G-function we can find a dominant term of the form $z^a e^{bz^c}$ (although this expression might not be the best possible, because of cancellation).

We must thus investigate

$$\lim_{T \rightarrow \infty} \int_{c-iT}^{c+iT} e^{zt} z^a e^{bz^c} dz.$$

(This principal value integral is the exact statement used in the Laplace inversion theorem.) We write $z = c + i\tau$. Then $\arg(z) \rightarrow \pm \frac{\pi}{2}$, and so $e^{zt} \sim e^{it\tau}$ (where $\sim$ shall always mean “asymptotically equivalent up to a positive real multiplicative constant”). Also $z^{x+iy} \sim |\tau|^x e^{iy \log |\tau|} e^{\pm xi \frac{\pi}{2}}$.

Set $\omega_{\pm} = be^{\pm i\Re(c)\frac{\pi}{2}}$. We have three cases:

1. $b = 0$ or $\Re(c) \leq 0$. In this case the integral converges if $\Re(a) \leq -1$.
2. $b \neq 0$, $\Im(c) = 0$, $\Re(c) > 0$. In this case the integral converges if $\Re(\omega_{\pm}) < 0$.
3. $b \neq 0$, $\Im(c) = 0$, $\Re(c) > 0$, $\Re(\omega_{\pm}) \leq 0$, and at least one of $\Re(\omega_{\pm}) = 0$. Here the same condition as in (1) applies.

Implemented G-Function Formulae

An important part of the algorithm is a table expressing various functions as Meijer G-functions. This is essentially a table of Mellin Transforms in disguise. The following automatically generated table shows the formulae currently implemented in SymPy. An entry “generated” means that the corresponding G-function has a variable number of parameters. This table is intended to shrink in future, when the algorithm’s capabilities of deriving new formulae improve. Of course it has to grow whenever a new class of special functions is to be dealt with.

Elementary functions:

$$a = aG_{1,1}^{1,0} \left(\begin{matrix} 1 \\ 0 \end{matrix} \middle| z \right) + aG_{1,1}^{0,1} \left(\begin{matrix} 1 \\ 0 \end{matrix} \middle| z \right)$$

$$\begin{aligned}
(z^q p + b)^{-a} &= \frac{b^{-a} G_{1,1}^{1,1} \left(\begin{matrix} 1-a \\ 0 \end{matrix} \middle| \frac{z^q p}{b} \right)}{\Gamma(a)} \\
\frac{-b^a + (z^q p)^a}{z^q p - b} &= \frac{b^{a-1} G_{2,2}^{2,2} \left(\begin{matrix} 0, a \\ 0, a \end{matrix} \middle| \frac{z^q p}{b} \right) \sin(\pi a)}{\pi} \\
\left(a + \sqrt{z^q p + a^2} \right)^b &= - \frac{a^b b G_{2,2}^{1,2} \left(\begin{matrix} \frac{b}{2} + \frac{1}{2}, \frac{b}{2} + 1 \\ 0 \end{matrix} \middle| \frac{z^q p}{a^2} \right)}{2\sqrt{\pi}} \\
\left(-a + \sqrt{z^q p + a^2} \right)^b &= \frac{a^b b G_{2,2}^{1,2} \left(\begin{matrix} \frac{b}{2} + \frac{1}{2}, \frac{b}{2} + 1 \\ b \end{matrix} \middle| \frac{z^q p}{a^2} \right)}{2\sqrt{\pi}} \\
\frac{\left(a + \sqrt{z^q p + a^2} \right)^b}{\sqrt{z^q p + a^2}} &= \frac{a^{b-1} G_{2,2}^{1,2} \left(\begin{matrix} \frac{b}{2} + \frac{1}{2}, \frac{b}{2} \\ 0 \end{matrix} \middle| \frac{z^q p}{a^2} \right)}{\sqrt{\pi}} \\
\frac{\left(-a + \sqrt{z^q p + a^2} \right)^b}{\sqrt{z^q p + a^2}} &= \frac{a^{b-1} G_{2,2}^{1,2} \left(\begin{matrix} \frac{b}{2} + \frac{1}{2}, \frac{b}{2} \\ b \end{matrix} \middle| \frac{z^q p}{a^2} \right)}{\sqrt{\pi}} \\
\left(z^{\frac{q}{2}} \sqrt{p} + \sqrt{z^q p + a} \right)^b &= - \frac{a^{\frac{b}{2}} b G_{2,2}^{2,1} \left(\begin{matrix} \frac{b}{2} + 1, 1 - \frac{b}{2} \\ 0, \frac{1}{2} \end{matrix} \middle| \frac{z^q p}{a} \right)}{2\sqrt{\pi}} \\
\left(-z^{\frac{q}{2}} \sqrt{p} + \sqrt{z^q p + a} \right)^b &= \frac{a^{\frac{b}{2}} b G_{2,2}^{2,1} \left(\begin{matrix} 1 - \frac{b}{2}, \frac{b}{2} + 1 \\ 0, \frac{1}{2} \end{matrix} \middle| \frac{z^q p}{a} \right)}{2\sqrt{\pi}} \\
\frac{\left(z^{\frac{q}{2}} \sqrt{p} + \sqrt{z^q p + a} \right)^b}{\sqrt{z^q p + a}} &= \frac{a^{\frac{b}{2} - \frac{1}{2}} G_{2,2}^{2,1} \left(\begin{matrix} \frac{b}{2} + \frac{1}{2}, \frac{1}{2} - \frac{b}{2} \\ 0, \frac{1}{2} \end{matrix} \middle| \frac{z^q p}{a} \right)}{\sqrt{\pi}} \\
\frac{\left(-z^{\frac{q}{2}} \sqrt{p} + \sqrt{z^q p + a} \right)^b}{\sqrt{z^q p + a}} &= \frac{a^{\frac{b}{2} - \frac{1}{2}} G_{2,2}^{2,1} \left(\begin{matrix} \frac{1}{2} - \frac{b}{2}, \frac{b}{2} + \frac{1}{2} \\ 0, \frac{1}{2} \end{matrix} \middle| \frac{z^q p}{a} \right)}{\sqrt{\pi}}
\end{aligned}$$

Functions involving $|z^q p - b|$:

$$|z^q p - b|^{-a} = 2 G_{2,2}^{1,1} \left(\begin{matrix} 1-a \\ 0 \end{matrix} \middle| \frac{\frac{1}{2} - \frac{a}{2}}{\frac{1}{2} - \frac{a}{2}} \frac{z^q p}{b} \right) \sin\left(\frac{\pi a}{2}\right) |b|^{-a} \Gamma(1-a), \text{ if } \operatorname{re}(a) < 1$$

Functions involving $\operatorname{Chi}(z^q p)$:

$$\operatorname{Chi}(z^q p) = - \frac{\pi^{\frac{3}{2}} G_{2,4}^{2,0} \left(\begin{matrix} \frac{1}{2}, 1 \\ 0, 0 \end{matrix} \middle| \frac{z^{2q} p^2}{4} \right)}{2}$$

Functions involving $\operatorname{Ci}(z^q p)$:

$$\operatorname{Ci}(z^q p) = - \frac{\sqrt{\pi} G_{1,3}^{2,0} \left(\begin{matrix} 1 \\ 0, 0 \end{matrix} \middle| \frac{z^{2q} p^2}{4} \right)}{2}$$

Functions involving $\operatorname{Ei}(z^q p)$:

$$\operatorname{Ei}(z^q p) = -i\pi G_{1,1}^{1,0} \left(\begin{matrix} 1 \\ 0 \end{matrix} \middle| z \right) - G_{1,2}^{2,0} \left(\begin{matrix} 1 \\ 0, 0 \end{matrix} \middle| z^q p e^{i\pi} \right) - i\pi G_{1,1}^{0,1} \left(\begin{matrix} 1 \\ 0 \end{matrix} \middle| z \right)$$

Functions involving $\theta(z^q p - b)$:

$$(z^q p - b)^{a-1} \theta(z^q p - b) = b^{a-1} G_{1,1}^{0,1} \left(a \left| \frac{z^q p}{b} \right. \right) \Gamma(a), \text{ if } b > 0$$

$$(-z^q p + b)^{a-1} \theta(-z^q p + b) = b^{a-1} G_{1,1}^{1,0} \left(0 \left| \frac{z^q p}{b} \right. \right) \Gamma(a), \text{ if } b > 0$$

$$(z^q p - b)^{a-1} \theta \left(z - \left(\frac{b}{p} \right)^{\frac{1}{q}} \right) = b^{a-1} G_{1,1}^{0,1} \left(a \left| \frac{z^q p}{b} \right. \right) \Gamma(a), \text{ if } b > 0$$

$$(-z^q p + b)^{a-1} \theta \left(-z + \left(\frac{b}{p} \right)^{\frac{1}{q}} \right) = b^{a-1} G_{1,1}^{1,0} \left(0 \left| \frac{z^q p}{b} \right. \right) \Gamma(a), \text{ if } b > 0$$

Functions involving $\theta(-z^q p + 1)$, $\log(z^q p)$:

$$\log(z^q p)^n \theta(-z^q p + 1) = \text{generated}$$

$$\log(z^q p)^n \theta(z^q p - 1) = \text{generated}$$

Functions involving Shi($z^q p$):

$$\text{Shi}(z^q p) = \frac{z^q \sqrt{\pi} p G_{1,3}^{1,1} \left(\frac{1}{2} \left| \frac{z^{2q} p^2 e^{i\pi}}{4} \right. \right)}{4}$$

Functions involving Si($z^q p$):

$$\text{Si}(z^q p) = \frac{\sqrt{\pi} G_{1,3}^{1,1} \left(\frac{1}{2} \left| \frac{z^{2q} p^2}{4} \right. \right)}{2}$$

Functions involving $I_a(z^q p)$:

$$I_a(z^q p) = \pi G_{1,3}^{1,0} \left(\frac{a}{2} \left| \frac{z^{2q} p^2}{4} \right. \right)$$

Functions involving $J_a(z^q p)$:

$$J_a(z^q p) = G_{0,2}^{1,0} \left(\frac{a}{2} \left| \frac{z^{2q} p^2}{4} \right. \right)$$

Functions involving $K_a(z^q p)$:

$$K_a(z^q p) = \frac{G_{0,2}^{2,0} \left(\frac{a}{2}, -\frac{a}{2} \left| \frac{z^{2q} p^2}{4} \right. \right)}{2}$$

Functions involving $Y_a(z^q p)$:

$$Y_a(z^q p) = G_{1,3}^{2,0} \left(\frac{a}{2}, -\frac{a}{2} \left| \frac{z^{2q} p^2}{4} \right. \right)$$

Functions involving $\cos(z^q p)$:

$$\cos(z^q p) = \sqrt{\pi} G_{0,2}^{1,0} \left(0 \left| \frac{z^{2q} p^2}{4} \right. \right)$$

Functions involving $\cosh(z^q p)$:

$$\cosh(z^q p) = \pi^{\frac{3}{2}} G_{1,3}^{1,0} \left(0 \quad \frac{1}{2}, \frac{1}{2} \left| \frac{z^{2q} p^2}{4} \right. \right)$$

Functions involving $E(z^q p)$:

$$E(z^q p) = -\frac{G_{2,2}^{1,2} \left(\frac{1}{2}, \frac{3}{2} \left| -z^q p \right. \right)}{4}$$

Functions involving $K(z^q p)$:

$$K(z^q p) = \frac{G_{2,2}^{1,2} \left(\frac{1}{2}, \frac{1}{2} \left| -z^q p \right. \right)}{2}$$

Functions involving $\operatorname{erf}(z^q p)$:

$$\operatorname{erf}(z^q p) = \frac{G_{1,2}^{1,1} \left(\frac{1}{2} \left| z^{2q} p^2 \right. \right)}{\sqrt{\pi}}$$

Functions involving $\operatorname{erfc}(z^q p)$:

$$\operatorname{erfc}(z^q p) = \frac{G_{1,2}^{2,0} \left(0, \frac{1}{2} \left| z^{2q} p^2 \right. \right)}{\sqrt{\pi}}$$

Functions involving $\operatorname{erfi}(z^q p)$:

$$\operatorname{erfi}(z^q p) = \frac{z^q p G_{1,2}^{1,1} \left(\frac{1}{2}, -\frac{1}{2} \left| -z^{2q} p^2 \right. \right)}{\sqrt{\pi}}$$

Functions involving $e^{z^q p e^{i\pi}}$:

$$e^{z^q p e^{i\pi}} = G_{0,1}^{1,0} \left(0 \left| z^q p \right. \right)$$

Functions involving $E_a(z^q p)$:

$$E_a(z^q p) = G_{1,2}^{2,0} \left(a-1, 0 \left| z^q p \right. \right)$$

Functions involving $C(z^q p)$:

$$C(z^q p) = \frac{G_{1,3}^{1,1} \left(\frac{1}{4}, 0, \frac{3}{4} \left| \frac{z^{4q} \pi^2 p^4}{16} \right. \right)}{2}$$

Functions involving $S(z^q p)$:

$$S(z^q p) = \frac{G_{1,3}^{1,1} \left(\frac{3}{4}, 0, \frac{1}{4} \left| \frac{z^{4q} \pi^2 p^4}{16} \right. \right)}{2}$$

Functions involving $\log(z^q p)$:

$$\log(z^q p)^n = \text{generated}$$

$$\begin{aligned}\log(z^q p + a) &= G_{1,1}^{1,0} \left(\begin{matrix} 1 \\ 0 \end{matrix} \middle| z \right) \log(a) + G_{1,1}^{0,1} \left(\begin{matrix} 1 \\ 0 \end{matrix} \middle| z \right) \log(a) + G_{2,2}^{1,2} \left(\begin{matrix} 1, 1 \\ 1 \end{matrix} \middle| \frac{z^q p}{a} \right) \\ \log(|z^q p - a|) &= G_{1,1}^{1,0} \left(\begin{matrix} 1 \\ 0 \end{matrix} \middle| z \right) \log(|a|) + G_{1,1}^{0,1} \left(\begin{matrix} 1 \\ 0 \end{matrix} \middle| z \right) \log(|a|) + \pi G_{3,3}^{1,2} \left(\begin{matrix} 1, 1 \\ 1 \end{matrix} \middle| \frac{\frac{1}{2}}{0, \frac{1}{2}} \middle| \frac{z^q p}{a} \right)\end{aligned}$$

Functions involving $\sin(z^q p)$:

$$\sin(z^q p) = \sqrt{\pi} G_{0,2}^{1,0} \left(\begin{matrix} \frac{1}{2} \\ 0 \end{matrix} \middle| \frac{z^{2q} p^2}{4} \right)$$

Functions involving $\operatorname{sinc}(z^q p)$:

$$\operatorname{sinc}(z^q p) = \frac{\sqrt{\pi} G_{0,2}^{1,0} \left(\begin{matrix} \frac{1}{2} \\ 0 \end{matrix} \middle| \frac{z^{2q} p^2}{4} \right)}{2}$$

Functions involving $\sinh(z^q p)$:

$$\sinh(z^q p) = \pi^{\frac{3}{2}} G_{1,3}^{1,0} \left(\begin{matrix} 1 \\ \frac{1}{2} \end{matrix} \middle| \frac{z^{2q} p^2}{4} \right)$$

Internal API Reference

Integrate functions by rewriting them as Meijer G-functions.

There are three user-visible functions that can be used by other parts of the sympy library to solve various integration problems:

- `meijerint_indefinite`
- `meijerint_definite`
- `meijerint_inversion`

They can be used to compute, respectively, indefinite integrals, definite integrals over intervals of the real line, and inverse laplace-type integrals (from $c-I*\infty$ to $c+I*\infty$). See the respective docstrings for details.

The main references for this are:

[L] Luke, Y. L. (1969), The Special Functions and Their Approximations,
Volume 1

[R] Kelly B. Roach. Meijer G Function Representations.

In: Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation, pages 205-211, New York, 1997. ACM.

[P] A. P. Prudnikov, Yu. A. Brychkov and O. I. Marichev (1990).

Integrals and Series: More Special Functions, Vol. 3,. Gordon and Breach Science Publisher

exception `sympy.integrals.meijerint._CoeffExpValueError`

Exception raised by `_get_coeff_exp`, for internal use only.

`sympy.integrals.meijerint._check_antecedents(g1, g2, x)`

Return a condition under which the integral theorem applies.

`sympy.integrals.meijerint._check_antecedents_1(g, x, helper=False)`

Return a condition under which the mellin transform of g exists. Any power of x has already been absorbed into the G function, so this is just $\int_0^\infty g \, dx$.

See [L, section 5.6.1]. (Note that $s=1$.)

If `helper` is `True`, only check if the MT exists at infinity, i.e. if $\int_1^\infty g \, dx$ exists.

`sympy.integrals.meijerint._check_antecedents_inversion(g, x)`

Check antecedents for the laplace inversion integral.

`sympy.integrals.meijerint._condsimp(cond, first=True)`

Do naive simplifications on `cond`.

Explanation

Note that this routine is completely ad-hoc, simplification rules being added as need arises rather than following any logical pattern.

Examples

```
>>> from sympy.integrals.meijerint import _condsimp as simp
>>> from sympy import Or, Eq
>>> from sympy.abc import x, y
>>> simp(Or(x < y, Eq(x, y)))
x <= y
```

`sympy.integrals.meijerint._create_lookup_table(table)`

Add formulae for the function $\rightarrow$ meijerg lookup table.

`sympy.integrals.meijerint._dummy(name, token, expr, **kwargs)`

Return a dummy. This will return the same dummy if the same token+name is requested more than once, and it is not already in `expr`. This is for being cache-friendly.

`sympy.integrals.meijerint._dummy_(name, token, **kwargs)`

Return a dummy associated to `name` and `token`. Same effect as declaring it globally.

`sympy.integrals.meijerint._eval_cond(cond)`

Re-evaluate the conditions.

`sympy.integrals.meijerint._exponents(expr, x)`

Find the exponents of x (not including zero) in `expr`.

Examples

```
>>> from sympy.integrals.meijerint import _exponents
>>> from sympy.abc import x, y
>>> from sympy import sin
>>> _exponents(x, x)
{1}
>>> _exponents(x**2, x)
```

(continues on next page)

(continued from previous page)

```
{2}
>>> _exponents(x**2 + x, x)
{1, 2}
>>> _exponents(x**3*sin(x + x*y) + 1/x, x)
{-1, 1, 3, y}
```

`sympy.integrals.meijerint._find_splitting_points(expr, x)`

Find numbers a such that a linear substitution $x \rightarrow x + a$ would (hopefully) simplify expr .

Examples

```
>>> from sympy.integrals.meijerint import _find_splitting_points as fsp
>>> from sympy import sin
>>> from sympy.abc import x
>>> fsp(x, x)
{0}
>>> fsp((x-1)**3, x)
{1}
>>> fsp(sin(x+3)*x, x)
{-3, 0}
```

`sympy.integrals.meijerint._flip_g(g)`

Turn the G function into one of inverse argument (i.e. $G(1/x) \rightarrow G'(x)$)

`sympy.integrals.meijerint._functions(expr, x)`

Find the types of functions in expr , to estimate the complexity.

`sympy.integrals.meijerint._get_coeff_exp(expr, x)`

When expr is known to be of the form $c*x**b$, with c and/or b possibly 1, return c, b .

Examples

```
>>> from sympy.abc import x, a, b
>>> from sympy.integrals.meijerint import _get_coeff_exp
>>> _get_coeff_exp(a*x**b, x)
(a, b)
>>> _get_coeff_exp(x, x)
(1, 1)
>>> _get_coeff_exp(2*x, x)
(2, 1)
>>> _get_coeff_exp(x**3, x)
(1, 3)
```

`sympy.integrals.meijerint._guess_expansion(f, x)`

Try to guess sensible rewritings for integrand $f(x)$.

`sympy.integrals.meijerint._inflate_fox_h(g, a)`

Let d denote the integrand in the definition of the G function g . Consider the function H which is defined in the same way, but with integrand $d/\Gamma(a*s)$ (contour conventions as usual).

If a is rational, the function H can be written as $C \cdot G$, for a constant C and a G -function G .

This function returns C, G .

`sympy.integrals.meijerint._inflate_g(g, n)`

Return C, h such that h is a G function of argument z^n and $g = C \cdot h$.

`sympy.integrals.meijerint._int0oo(g1, g2, x)`

Express integral from zero to infinity $g1 \cdot g2$ using a G function, assuming the necessary conditions are fulfilled.

Examples

```
>>> from sympy.integrals.meijerint import _int0oo
>>> from sympy.abc import s, t, m
>>> from sympy import meijerg, S
>>> g1 = meijerg([], [], [-S(1)/2, 0], [], s**2*t/4)
>>> g2 = meijerg([], [], [m/2], [-m/2], t/4)
>>> _int0oo(g1, g2, t)
4*meijerg(((1/2, 0), ()), ((m/2,), (-m/2,)), s**(-2))/s**2
```

`sympy.integrals.meijerint._int0oo_1(g, x)`

Evaluate $\int_0^\infty g \, dx$ using G functions, assuming the necessary conditions are fulfilled.

Examples

```
>>> from sympy.abc import a, b, c, d, x, y
>>> from sympy import meijerg
>>> from sympy.integrals.meijerint import _int0oo_1
>>> _int0oo_1(meijerg([a], [b], [c], [d], x*y), x)
gamma(-a)*gamma(c + 1)/(y*gamma(-d)*gamma(b + 1))
```

`sympy.integrals.meijerint._int_inversion(g, x, t)`

Compute the laplace inversion integral, assuming the formula applies.

`sympy.integrals.meijerint._is_analytic(f, x)`

Check if $f(x)$, when expressed using G functions on the positive reals, will in fact agree with the G functions almost everywhere

`sympy.integrals.meijerint._meijerint_definite_2(f, x)`

Try to integrate $f \, dx$ from zero to infinity.

The body of this function computes various ‘simplifications’ $f_1, f_2, \dots$ of f (e.g. by calling `expand_mul()`, `trigexpand()` - see `_guess_expansion`) and calls `_meijerint_definite_3` with each of these in succession. If `_meijerint_definite_3` succeeds with any of the simplified functions, returns this result.

`sympy.integrals.meijerint._meijerint_definite_3(f, x)`

Try to integrate $f \, dx$ from zero to infinity.

This function calls `_meijerint_definite_4` to try to compute the integral. If this fails, it tries using linearity.

`sympy.integrals.meijerint._meijerint_definite_4(f, x, only_double=False)`

Try to integrate $f dx$ from zero to infinity.

Explanation

This function tries to apply the integration theorems found in literature, i.e. it tries to rewrite f as either one or a product of two G-functions.

The parameter `only_double` is used internally in the recursive algorithm to disable trying to rewrite f as a single G-function.

`sympy.integrals.meijerint._meijerint_indefinite_1(f, x)`

Helper that does not attempt any substitution.

`sympy.integrals.meijerint._mul_args(f)`

Return a list L such that $Mul(*L) == f$.

If f is not a `Mul` or `Pow`, $L=[f]$. If $f=g^n$ for an integer n , $L=[g]*n$. If f is a `Mul`, L comes from applying `_mul_args` to all factors of f .

`sympy.integrals.meijerint._mul_as_two_parts(f)`

Find all the ways to split f into a product of two terms. Return `None` on failure.

Explanation

Although the order is canonical from multiset partitions, this is not necessarily the best order to process the terms. For example, if the case of $\text{len}(gs) == 2$ is removed and multiset is allowed to sort the terms, some tests fail.

Examples

```
>>> from sympy.integrals.meijerint import _mul_as_two_parts
>>> from sympy import sin, exp, ordered
>>> from sympy.abc import x
>>> list(ordered(_mul_as_two_parts(x*sin(x)*exp(x))))
[(x, exp(x)*sin(x)), (x*exp(x), sin(x)), (x*sin(x), exp(x))]
```

`sympy.integrals.meijerint._my_principal_branch(expr, period, full_pb=False)`

Bring expr nearer to its principal branch by removing superfluous factors. This function does *not* guarantee to yield the principal branch, to avoid introducing opaque `principal_branch()` objects, unless `full_pb=True`.

`sympy.integrals.meijerint._mytype(f, x)`

Create a hashable entity describing the type of f .

`sympy.integrals.meijerint._rewritel(f, x, recursive=True)`

Try to rewrite f using a (sum of) single G functions with argument $a*x^b$. Return fac , po , g such that $f = \text{fac}*\text{po}*g$, fac is independent of x . and $\text{po} = x^s$. Here g is a result from `_rewrite_single`. Return `None` on failure.

`sympy.integrals.meijerint._rewrite2(f, x)`

Try to rewrite f as a product of two G functions of arguments $a*x**b$. Return fac , po , $g1$, $g2$ such that $f = fac*po*g1*g2$, where fac is independent of x and po is $x**s$. Here $g1$ and $g2$ are results of `_rewrite_single`. Returns None on failure.

`sympy.integrals.meijerint._rewrite_inversion(fac, po, g, x)`

Absorb $po == x**s$ into g .

`sympy.integrals.meijerint._rewrite_saxena(fac, po, g1, g2, x, full_pb=False)`

Rewrite the integral $fac*po*g1*g2$ from 0 to ∞ in terms of G functions with argument $c*x$.

Explanation

Return C , $f1$, $f2$ such that $\int_0^\infty C f1 f2 dx = \int_0^\infty fac po g1 g2 dx$.

Examples

```
>>> from sympy.integrals.meijerint import _rewrite_saxena
>>> from sympy.abc import s, t, m
>>> from sympy import meijerg
>>> g1 = meijerg([], [], [0], [], s*t)
>>> g2 = meijerg([], [], [m/2], [-m/2], t**2/4)
>>> r = _rewrite_saxena(1, t**0, g1, g2, t)
>>> r[0]
s/(4*sqrt(pi))
>>> r[1]
meijerg((()), ()), ((-1/2, 0), ()), s**2*t/4)
>>> r[2]
meijerg((()), ()), ((m/2,), (-m/2,)), t/4)
```

`sympy.integrals.meijerint._rewrite_saxena_1(fac, po, g, x)`

Rewrite the integral $fac*po*g dx$, from zero to infinity, as integral $fac*G$, where G has argument $a*x$. Note $po=x**s$. Return fac , G .

`sympy.integrals.meijerint._rewrite_single(f, x, recursive=True)`

Try to rewrite f as a sum of single G functions of the form $C*x**s*G(a*x**b)$, where b is a rational number and C is independent of x . We guarantee that `result.argument.as_coeff_mul(x)` returns $(a, (x**b,))$ or $(a, ())$. Returns a list of tuples (C, s, G) and a condition $cond$. Returns None on failure.

`sympy.integrals.meijerint._split_mul(f, x)`

Split expression f into fac , po , g , where fac is a constant factor, $po = x**s$ for some s independent of x , and g is "the rest".

Examples

```
>>> from sympy.integrals.meijerint import _split_mul
>>> from sympy import sin
>>> from sympy.abc import s, x
>>> _split_mul((3*x)**s*sin(x**2)*x, x)
(3**s, x*x**s, sin(x**2))
```

`sympy.integrals.meijerint.meijerint_definite(f, x, a, b)`

Integrate f over the interval $[a, b]$, by rewriting it as a product of two G functions, or as a single G function.

Return $res, cond$, where $cond$ are convergence conditions.

Examples

```
>>> from sympy.integrals.meijerint import meijerint_definite
>>> from sympy import exp, oo
>>> from sympy.abc import x
>>> meijerint_definite(exp(-x**2), x, -oo, oo)
(sqrt(pi), True)
```

This function is implemented as a succession of functions `meijerint_definite`, `_meijerint_definite_2`, `_meijerint_definite_3`, `_meijerint_definite_4`. Each function in the list calls the next one (presumably) several times. This means that calling `meijerint_definite` can be very costly.

`sympy.integrals.meijerint.meijerint_indefinite(f, x)`

Compute an indefinite integral of f by rewriting it as a G function.

Examples

```
>>> from sympy.integrals.meijerint import meijerint_indefinite
>>> from sympy import sin
>>> from sympy.abc import x
>>> meijerint_indefinite(sin(x), x)
-cos(x)
```

`sympy.integrals.meijerint.meijerint_inversion(f, x, t)`

Compute the inverse laplace transform $\int_{c-i\infty}^{c+i\infty} f(x)e^{tx} dx$, for real c larger than the real part of all singularities of f .

Note that t is always assumed real and positive.

Return `None` if the integral does not exist or could not be evaluated.

Examples

```
>>> from sympy.abc import x, t
>>> from sympy.integrals.meijerint import meijerint_inversion
>>> meijerint_inversion(1/x, x, t)
Heaviside(t)
```

Integrals

The integrals module in SymPy implements methods to calculate definite and indefinite integrals of expressions.

Principal method in this module is `integrate()` (page 598)

- `integrate(f, x)` returns the indefinite integral $\int f dx$
- `integrate(f, (x, a, b))` returns the definite integral $\int_a^b f dx$

Examples

SymPy can integrate a vast array of functions. It can integrate polynomial functions:

```
>>> from sympy import *
>>> init_printing(use_unicode=False, wrap_line=False)
>>> x = Symbol('x')
>>> integrate(x**2 + x + 1, x)
 3      2
x      x
-- + -- + x
3      2
```

Rational functions:

```
>>> integrate(x/(x**2+2*x+1), x)
          1
log(x + 1) + ----
          x + 1
```

Exponential-polynomial functions. These multiplicative combinations of polynomials and the functions `exp`, `cos` and `sin` can be integrated by hand using repeated integration by parts, which is an extremely tedious process. Happily, SymPy will deal with these integrals.

```
>>> integrate(x**2 * exp(x) * cos(x), x)
 2 x      2 x      x      x
x *e *sin(x)  x *e *cos(x)  x      e *sin(x)  e *cos(x)
----- + ----- - x*e *sin(x) + ----- - -----
      2          2          2          2
```

even a few nonelementary integrals (in particular, some integrals involving the error function) can be evaluated:

```
>>> integrate(exp(-x**2)*erf(x), x)
      2
\ / pi *erf (x)
-----
      4
```

Integral Transforms

SymPy has special support for definite integrals, and integral transforms.

`sympy.integrals.transforms.mellin_transform(f, x, s, **hints)`

Compute the Mellin transform $F(s)$ of $f(x)$,

$$F(s) = \int_0^{\infty} x^{s-1} f(x) dx.$$

For all “sensible” functions, this converges absolutely in a strip

$$a < \operatorname{Re}(s) < b.$$

Explanation

The Mellin transform is related via change of variables to the Fourier transform, and also to the (bilateral) Laplace transform.

This function returns $(F, (a, b), \text{cond})$ where F is the Mellin transform of f , (a, b) is the fundamental strip (as above), and cond are auxiliary convergence conditions.

If the integral cannot be computed in closed form, this function returns an unevaluated *MellinTransform* (page 576) object.

For a description of possible hints, refer to the docstring of `sympy.integrals.transforms.IntegralTransform.doit()` (page 586). If `noconds=False`, then only F will be returned (i.e. not cond , and also not the strip (a, b)).

Examples

```
>>> from sympy import mellin_transform, exp
>>> from sympy.abc import x, s
>>> mellin_transform(exp(-x), x, s)
(gamma(s), (0, oo), True)
```

See also:

inverse_mellin_transform (page 577), *laplace_transform* (page 577),
fourier_transform (page 579), *hankel_transform* (page 584),
inverse_hankel_transform (page 585)

class `sympy.integrals.transforms.MellinTransform(*args)`

Class representing unevaluated Mellin transforms.

For usage of this class, see the *IntegralTransform* (page 586) docstring.

For how to compute Mellin transforms, see the *mellin_transform()* (page 576) docstring.

`sympy.integrals.transforms.inverse_mellin_transform(F, s, x, strip, **hints)`

Compute the inverse Mellin transform of $F(s)$ over the fundamental strip given by `strip=(a, b)`.

Explanation

This can be defined as

$$f(x) = \frac{1}{2\pi i} \int_{c-i\infty}^{c+i\infty} x^{-s} F(s) ds,$$

for any c in the fundamental strip. Under certain regularity conditions on F and/or f , this recovers f from its Mellin transform F (and vice versa), for positive real x .

One of a or b may be passed as `None`; a suitable c will be inferred.

If the integral cannot be computed in closed form, this function returns an unevaluated [InverseMellinTransform](#) (page 577) object.

Note that this function will assume x to be positive and real, regardless of the SymPy assumptions!

For a description of possible hints, refer to the docstring of [sympy.integrals.transforms.IntegralTransform.doit\(\)](#) (page 586).

Examples

```
>>> from sympy import inverse_mellin_transform, oo, gamma
>>> from sympy.abc import x, s
>>> inverse_mellin_transform(gamma(s), s, x, (0, oo))
exp(-x)
```

The fundamental strip matters:

```
>>> f = 1/(s**2 - 1)
>>> inverse_mellin_transform(f, s, x, (-oo, -1))
x*(1 - 1/x**2)*Heaviside(x - 1)/2
>>> inverse_mellin_transform(f, s, x, (-1, 1))
-x*Heaviside(1 - x)/2 - Heaviside(x - 1)/(2*x)
>>> inverse_mellin_transform(f, s, x, (1, oo))
(1/2 - x**2/2)*Heaviside(1 - x)/x
```

See also:

[mellin_transform](#) (page 576), [hankel_transform](#) (page 584),
[inverse_hankel_transform](#) (page 585)

class `sympy.integrals.transforms.InverseMellinTransform(*args)`

Class representing unevaluated inverse Mellin transforms.

For usage of this class, see the [IntegralTransform](#) (page 586) docstring.

For how to compute inverse Mellin transforms, see the [inverse_mellin_transform\(\)](#) (page 577) docstring.

`sympy.integrals.transforms.laplace_transform(f, t, s, legacy_matrix=True, **hints)`

Compute the Laplace Transform $F(s)$ of $f(t)$,

$$F(s) = \int_{0-}^{\infty} e^{-st} f(t) dt.$$

Explanation

For all sensible functions, this converges absolutely in a half-plane

$$a < \operatorname{Re}(s)$$

This function returns (F, a, cond) where F is the Laplace transform of f , a is the half-plane of convergence, and cond are auxiliary convergence conditions.

The implementation is rule-based, and if you are interested in which rules are applied, and whether integration is attempted, you can switch debug information on by setting `sympy.SYMPY_DEBUG=True`.

The lower bound is $0-$, meaning that this bound should be approached from the lower side. This is only necessary if distributions are involved. At present, it is only done if $f(t)$ contains `DiracDelta`, in which case the Laplace transform is computed implicitly as

$$F(s) = \lim_{\tau \rightarrow 0-} \int_{\tau}^{\infty} e^{-st} f(t) dt$$

by applying rules.

If the integral cannot be fully computed in closed form, this function returns an unevaluated [LaplaceTransform](#) (page 579) object.

For a description of possible hints, refer to the docstring of [sympy.integrals.transforms.IntegralTransform.doit\(\)](#) (page 586). If `noconds=True`, only F will be returned (i.e. not cond , and also not the plane a).

Deprecated since version 1.9: Legacy behavior for matrices where `laplace_transform` with `noconds=False` (the default) returns a Matrix whose elements are tuples. The behavior of `laplace_transform` for matrices will change in a future release of SymPy to return a tuple of the transformed Matrix and the convergence conditions for the matrix as a whole. Use `legacy_matrix=False` to enable the new behavior.

Examples

```
>>> from sympy import DiracDelta, exp, laplace_transform
>>> from sympy.abc import t, s, a
>>> laplace_transform(t**4, t, s)
(24/s**5, 0, True)
>>> laplace_transform(t**a, t, s)
(gamma(a + 1)/(s*s**a), 0, re(a) > -1)
>>> laplace_transform(DiracDelta(t)-a*exp(-a*t), t, s)
(s/(a + s), Max(0, -a), True)
```

See also:

[inverse_laplace_transform](#) (page 579), [mellin_transform](#) (page 576),
[fourier_transform](#) (page 579), [hankel_transform](#) (page 584),
[inverse_hankel_transform](#) (page 585)

class `sympy.integrals.transforms.LaplaceTransform(*args)`

Class representing unevaluated Laplace transforms.

For usage of this class, see the [IntegralTransform](#) (page 586) docstring.

For how to compute Laplace transforms, see the [laplace_transform\(\)](#) (page 577) docstring.

`sympy.integrals.transforms.inverse_laplace_transform(F, s, t, plane=None, **hints)`

Compute the inverse Laplace transform of $F(s)$, defined as

$$f(t) = \frac{1}{2\pi i} \int_{c-i\infty}^{c+i\infty} e^{st} F(s) ds,$$

for c so large that $F(s)$ has no singularities in the half-plane $\operatorname{Re}(s) > c - \epsilon$.

Explanation

The plane can be specified by argument `plane`, but will be inferred if passed as `None`.

Under certain regularity conditions, this recovers $f(t)$ from its Laplace Transform $F(s)$, for non-negative t , and vice versa.

If the integral cannot be computed in closed form, this function returns an unevaluated [InverseLaplaceTransform](#) (page 579) object.

Note that this function will always assume t to be real, regardless of the SymPy assumption on t .

For a description of possible hints, refer to the docstring of [sympy.integrals.transforms.IntegralTransform.doit\(\)](#) (page 586).

Examples

```
>>> from sympy import inverse_laplace_transform, exp, Symbol
>>> from sympy.abc import s, t
>>> a = Symbol('a', positive=True)
>>> inverse_laplace_transform(exp(-a*s)/s, s, t)
Heaviside(-a + t)
```

See also:

[laplace_transform](#) (page 577), [_fast_inverse_laplace](#) (page 579), [hankel_transform](#) (page 584), [inverse_hankel_transform](#) (page 585)

class `sympy.integrals.transforms.InverseLaplaceTransform(*args)`

Class representing unevaluated inverse Laplace transforms.

For usage of this class, see the [IntegralTransform](#) (page 586) docstring.

For how to compute inverse Laplace transforms, see the [inverse_laplace_transform\(\)](#) (page 579) docstring.

`sympy.integrals.transforms._fast_inverse_laplace(e, s, t)`

Fast inverse Laplace transform of rational function including RootSum

`sympy.integrals.transforms.fourier_transform(f, x, k, **hints)`

Compute the unitary, ordinary-frequency Fourier transform of f , defined as

$$F(k) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x k} dx.$$

Explanation

If the transform cannot be computed in closed form, this function returns an unevaluated *FourierTransform* (page 580) object.

For other Fourier transform conventions, see the function `sympy.integrals.transforms._fourier_transform()` (page 580).

For a description of possible hints, refer to the docstring of `sympy.integrals.transforms.IntegralTransform.doit()` (page 586). Note that for this transform, by default `noconds=True`.

Examples

```
>>> from sympy import fourier_transform, exp
>>> from sympy.abc import x, k
>>> fourier_transform(exp(-x**2), x, k)
sqrt(pi)*exp(-pi**2*k**2)
>>> fourier_transform(exp(-x**2), x, k, noconds=False)
(sqrt(pi)*exp(-pi**2*k**2), True)
```

See also:

inverse_fourier_transform (page 580), *sine_transform* (page 581),
inverse_sine_transform (page 582), *cosine_transform* (page 583),
inverse_cosine_transform (page 583), *hankel_transform* (page 584),
inverse_hankel_transform (page 585), *mellin_transform* (page 576),
laplace_transform (page 577)

`sympy.integrals.transforms._fourier_transform(f, x, k, a, b, name, simplify=True)`

Compute a general Fourier-type transform

$$F(k) = a \int_{-\infty}^{\infty} e^{bixk} f(x) dx.$$

For suitable choice of a and b , this reduces to the standard Fourier and inverse Fourier transforms.

class `sympy.integrals.transforms.FourierTransform(*args)`

Class representing unevaluated Fourier transforms.

For usage of this class, see the *IntegralTransform* (page 586) docstring.

For how to compute Fourier transforms, see the *fourier_transform()* (page 579) docstring.

`sympy.integrals.transforms.inverse_fourier_transform(F, k, x, **hints)`

Compute the unitary, ordinary-frequency inverse Fourier transform of F , defined as

$$f(x) = \int_{-\infty}^{\infty} F(k) e^{2\pi i x k} dk.$$

Explanation

If the transform cannot be computed in closed form, this function returns an unevaluated *InverseFourierTransform* (page 581) object.

For other Fourier transform conventions, see the function *sympy.integrals.transforms._fourier_transform()* (page 580).

For a description of possible hints, refer to the docstring of *sympy.integrals.transforms.IntegralTransform.doit()* (page 586). Note that for this transform, by default *noconds=True*.

Examples

```
>>> from sympy import inverse_fourier_transform, exp, sqrt, pi
>>> from sympy.abc import x, k
>>> inverse_fourier_transform(sqrt(pi)*exp(-(pi*k)**2), k, x)
exp(-x**2)
>>> inverse_fourier_transform(sqrt(pi)*exp(-(pi*k)**2), k, x,
→noconds=False)
(exp(-x**2), True)
```

See also:

fourier_transform (page 579), *sine_transform* (page 581), *inverse_sine_transform* (page 582), *cosine_transform* (page 583), *inverse_cosine_transform* (page 583), *hankel_transform* (page 584), *inverse_hankel_transform* (page 585), *mellin_transform* (page 576), *laplace_transform* (page 577)

class *sympy.integrals.transforms.InverseFourierTransform*(*args)

Class representing unevaluated inverse Fourier transforms.

For usage of this class, see the *IntegralTransform* (page 586) docstring.

For how to compute inverse Fourier transforms, see the *inverse_fourier_transform()* (page 580) docstring.

sympy.integrals.transforms.sine_transform(*f*, *x*, *k*, ***hints*)

Compute the unitary, ordinary-frequency sine transform of *f*, defined as

$$F(k) = \sqrt{\frac{2}{\pi}} \int_0^{\infty} f(x) \sin(2\pi x k) dx.$$

Explanation

If the transform cannot be computed in closed form, this function returns an unevaluated *SineTransform* (page 582) object.

For a description of possible hints, refer to the docstring of *sympy.integrals.transforms.IntegralTransform.doit()* (page 586). Note that for this transform, by default *noconds=True*.

Examples

```
>>> from sympy import sine_transform, exp
>>> from sympy.abc import x, k, a
>>> sine_transform(x*exp(-a*x**2), x, k)
sqrt(2)*k*exp(-k**2/(4*a))/(4*a**(3/2))
>>> sine_transform(x**(-a), x, k)
2**(1/2 - a)*k**(a - 1)*gamma(1 - a/2)/gamma(a/2 + 1/2)
```

See also:

[fourier_transform](#) (page 579), [inverse_fourier_transform](#) (page 580),
[inverse_sine_transform](#) (page 582), [cosine_transform](#) (page 583),
[inverse_cosine_transform](#) (page 583), [hankel_transform](#) (page 584),
[inverse_hankel_transform](#) (page 585), [mellin_transform](#) (page 576),
[laplace_transform](#) (page 577)

class `sympy.integrals.transforms.SineTransform(*args)`

Class representing unevaluated sine transforms.

For usage of this class, see the [IntegralTransform](#) (page 586) docstring.

For how to compute sine transforms, see the [sine_transform\(\)](#) (page 581) docstring.

`sympy.integrals.transforms.inverse_sine_transform(F, k, x, **hints)`

Compute the unitary, ordinary-frequency inverse sine transform of F , defined as

$$f(x) = \sqrt{\frac{2}{\pi}} \int_0^{\infty} F(k) \sin(2\pi x k) dk.$$

Explanation

If the transform cannot be computed in closed form, this function returns an unevaluated [InverseSineTransform](#) (page 583) object.

For a description of possible hints, refer to the docstring of [sympy.integrals.transforms.IntegralTransform.doit\(\)](#) (page 586). Note that for this transform, by default `noconds=True`.

Examples

```
>>> from sympy import inverse_sine_transform, exp, sqrt, gamma
>>> from sympy.abc import x, k, a
>>> inverse_sine_transform(2**((1-2*a)/2)*k**(a - 1)*
...     gamma(-a/2 + 1)/gamma((a+1)/2), k, x)
x**(-a)
>>> inverse_sine_transform(sqrt(2)*k*exp(-k**2/(4*a))/(4*sqrt(a)**3), k,
↪ x)
x*exp(-a*x**2)
```

See also:

[fourier_transform](#) (page 579), [inverse_fourier_transform](#) (page 580),
[sine_transform](#) (page 581), [cosine_transform](#) (page 583),

[inverse_cosine_transform](#) (page 583), [hankel_transform](#) (page 584),
[inverse_hankel_transform](#) (page 585), [mellin_transform](#) (page 576),
[laplace_transform](#) (page 577)

class `sympy.integrals.transforms.InverseSineTransform(*args)`

Class representing unevaluated inverse sine transforms.

For usage of this class, see the [IntegralTransform](#) (page 586) docstring.

For how to compute inverse sine transforms, see the [inverse_sine_transform\(\)](#) (page 582) docstring.

`sympy.integrals.transforms.cosine_transform(f, x, k, **hints)`

Compute the unitary, ordinary-frequency cosine transform of f , defined as

$$F(k) = \sqrt{\frac{2}{\pi}} \int_0^{\infty} f(x) \cos(2\pi x k) dx.$$

Explanation

If the transform cannot be computed in closed form, this function returns an unevaluated [CosineTransform](#) (page 583) object.

For a description of possible hints, refer to the docstring of [sympy.integrals.transforms.IntegralTransform.doit\(\)](#) (page 586). Note that for this transform, by default `noconds=True`.

Examples

```
>>> from sympy import cosine_transform, exp, sqrt, cos
>>> from sympy.abc import x, k, a
>>> cosine_transform(exp(-a*x), x, k)
sqrt(2)*a/(sqrt(pi)*(a**2 + k**2))
>>> cosine_transform(exp(-a*sqrt(x))*cos(a*sqrt(x)), x, k)
a*exp(-a**2/(2*k))/(2*k**(3/2))
```

See also:

[fourier_transform](#) (page 579), [inverse_fourier_transform](#) (page 580),
[sine_transform](#) (page 581), [inverse_sine_transform](#) (page 582),
[inverse_cosine_transform](#) (page 583), [hankel_transform](#) (page 584),
[inverse_hankel_transform](#) (page 585), [mellin_transform](#) (page 576),
[laplace_transform](#) (page 577)

class `sympy.integrals.transforms.CosineTransform(*args)`

Class representing unevaluated cosine transforms.

For usage of this class, see the [IntegralTransform](#) (page 586) docstring.

For how to compute cosine transforms, see the [cosine_transform\(\)](#) (page 583) docstring.

`sympy.integrals.transforms.inverse_cosine_transform(F, k, x, **hints)`

Compute the unitary, ordinary-frequency inverse cosine transform of F , defined as

$$f(x) = \sqrt{\frac{2}{\pi}} \int_0^{\infty} F(k) \cos(2\pi x k) dk.$$

Explanation

If the transform cannot be computed in closed form, this function returns an unevaluated *InverseCosineTransform* (page 584) object.

For a description of possible hints, refer to the docstring of *sympy.integrals.transforms.IntegralTransform.doit()* (page 586). Note that for this transform, by default `noconds=True`.

Examples

```
>>> from sympy import inverse_cosine_transform, sqrt, pi
>>> from sympy.abc import x, k, a
>>> inverse_cosine_transform(sqrt(2)*a/(sqrt(pi)*(a**2 + k**2)), k, x)
exp(-a*x)
>>> inverse_cosine_transform(1/sqrt(k), k, x)
1/sqrt(x)
```

See also:

fourier_transform (page 579), *inverse_fourier_transform* (page 580), *sine_transform* (page 581), *inverse_sine_transform* (page 582), *cosine_transform* (page 583), *hankel_transform* (page 584), *inverse_hankel_transform* (page 585), *mellin_transform* (page 576), *laplace_transform* (page 577)

class `sympy.integrals.transforms.InverseCosineTransform(*args)`

Class representing unevaluated inverse cosine transforms.

For usage of this class, see the *IntegralTransform* (page 586) docstring.

For how to compute inverse cosine transforms, see the *inverse_cosine_transform()* (page 583) docstring.

`sympy.integrals.transforms.hankel_transform(f, r, k, nu, **hints)`

Compute the Hankel transform of f , defined as

$$F_{\nu}(k) = \int_0^{\infty} f(r) J_{\nu}(kr) r dr.$$

Explanation

If the transform cannot be computed in closed form, this function returns an unevaluated *HankelTransform* (page 585) object.

For a description of possible hints, refer to the docstring of *sympy.integrals.transforms.IntegralTransform.doit()* (page 586). Note that for this transform, by default `noconds=True`.

Examples

```
>>> from sympy import hankel_transform, inverse_hankel_transform
>>> from sympy import exp
>>> from sympy.abc import r, k, m, nu, a
```

```
>>> ht = hankel_transform(1/r**m, r, k, nu)
>>> ht
2*k**(m - 2)*gamma(-m/2 + nu/2 + 1)/(2**m*gamma(m/2 + nu/2))
```

```
>>> inverse_hankel_transform(ht, k, r, nu)
r**(-m)
```

```
>>> ht = hankel_transform(exp(-a*r), r, k, 0)
>>> ht
a/(k**3*(a**2/k**2 + 1)**(3/2))
```

```
>>> inverse_hankel_transform(ht, k, r, 0)
exp(-a*r)
```

See also:

[fourier_transform](#) (page 579), [inverse_fourier_transform](#) (page 580), [sine_transform](#) (page 581), [inverse_sine_transform](#) (page 582), [cosine_transform](#) (page 583), [inverse_cosine_transform](#) (page 583), [inverse_hankel_transform](#) (page 585), [mellin_transform](#) (page 576), [laplace_transform](#) (page 577)

class sympy.integrals.transforms.HankelTransform(*args)

Class representing unevaluated Hankel transforms.

For usage of this class, see the [IntegralTransform](#) (page 586) docstring.

For how to compute Hankel transforms, see the [hankel_transform\(\)](#) (page 584) docstring.

sympy.integrals.transforms.inverse_hankel_transform(*F*, *k*, *r*, *nu*, ***hints*)

Compute the inverse Hankel transform of *F* defined as

$$f(r) = \int_0^{\infty} F_{\nu}(k) J_{\nu}(kr) k dk.$$

Explanation

If the transform cannot be computed in closed form, this function returns an unevaluated [InverseHankelTransform](#) (page 586) object.

For a description of possible hints, refer to the docstring of [sympy.integrals.transforms.IntegralTransform.doit\(\)](#) (page 586). Note that for this transform, by default `noconds=True`.

Examples

```
>>> from sympy import hankel_transform, inverse_hankel_transform
>>> from sympy import exp
>>> from sympy.abc import r, k, m, nu, a
```

```
>>> ht = hankel_transform(1/r**m, r, k, nu)
>>> ht
2*k**(m - 2)*gamma(-m/2 + nu/2 + 1)/(2**m*gamma(m/2 + nu/2))
```

```
>>> inverse_hankel_transform(ht, k, r, nu)
r**(-m)
```

```
>>> ht = hankel_transform(exp(-a*r), r, k, 0)
>>> ht
a/(k**3*(a**2/k**2 + 1)**(3/2))
```

```
>>> inverse_hankel_transform(ht, k, r, 0)
exp(-a*r)
```

See also:

[fourier_transform](#) (page 579), [inverse_fourier_transform](#) (page 580), [sine_transform](#) (page 581), [inverse_sine_transform](#) (page 582), [cosine_transform](#) (page 583), [inverse_cosine_transform](#) (page 583), [hankel_transform](#) (page 584), [mellin_transform](#) (page 576), [laplace_transform](#) (page 577)

class sympy.integrals.transforms.**InverseHankelTransform**(*args)

Class representing unevaluated inverse Hankel transforms.

For usage of this class, see the [IntegralTransform](#) (page 586) docstring.

For how to compute inverse Hankel transforms, see the [inverse_hankel_transform\(\)](#) (page 585) docstring.

class sympy.integrals.transforms.**IntegralTransform**(*args)

Base class for integral transforms.

Explanation

This class represents unevaluated transforms.

To implement a concrete transform, derive from this class and implement the `_compute_transform(f, x, s, **hints)` and `_as_integral(f, x, s)` functions. If the transform cannot be computed, raise [IntegralTransformError](#) (page 587).

Also set `cls._name`. For instance,

```
>>> from sympy import LaplaceTransform
>>> LaplaceTransform._name
'Laplace'
```

Implement `self._collapse_extra` if your function returns more than just a number and possibly a convergence condition.

doit(hints)**

Try to evaluate the transform in closed form.

Explanation

This general function handles linearity, but apart from that leaves pretty much everything to `_compute_transform`.

Standard hints are the following:

- `simplify`: whether or not to simplify the result
- `noconds`: if True, do not return convergence conditions
- **needeval**: if True, raise `IntegralTransformError` instead of returning `IntegralTransform` objects

The default values of these hints depend on the concrete transform, usually the default is `(simplify, noconds, needeval) = (True, False, False)`.

property function

The function to be transformed.

property function_variable

The dependent variable of the function to be transformed.

property transform_variable

The independent transform variable.

exception `sympy.integrals.transforms.IntegralTransformError(transform, function, msg)`

Exception raised in relation to problems computing transforms.

Explanation

This class is mostly used internally; if integrals cannot be computed objects representing unevaluated transforms are usually returned.

The hint `needeval=True` can be used to disable returning transform objects, and instead raise this exception if an integral cannot be computed.

Internals

SymPy uses a number of algorithms to compute integrals. Algorithms are tried in order until one produces an answer. Most of these algorithms can be enabled or disabled manually using various flags to `integrate()` (page 598) or `doit()` (page 603).

SymPy first applies several heuristic algorithms, as these are the fastest:

1. If the function is a rational function, there is a complete algorithm for integrating rational functions called the Lazard-Rioboo-Trager and the Horowitz-Ostrogradsky algorithms. They are implemented in `ratint()` (page 587).

`sympy.integrals.rationaltools.ratint(f, x, **flags)`

Performs indefinite integration of rational functions.

Explanation

Given a field K and a rational function $f = p/q$, where p and q are polynomials in $K[x]$, returns a function g such that $f = g'$.

Examples

```
>>> from sympy.integrals.rationaltools import ratint
>>> from sympy.abc import x
```

```
>>> ratint(36/(x**5 - 2*x**4 - 2*x**3 + 4*x**2 + x - 2), x)
(12*x + 6)/(x**2 - 1) + 4*log(x - 2) - 4*log(x + 1)
```

See also:

[*sympy.integrals.integrals.Integral.doit*](#) (page 603), [*sympy.integrals.rationaltools.ratint_logpart*](#) (page 588), [*sympy.integrals.rationaltools.ratint_ratpart*](#) (page 588)

References

[R525]

`sympy.integrals.rationaltools.ratint_ratpart(f, g, x)`
Horowitz-Ostrogradsky algorithm.

Explanation

Given a field K and polynomials f and g in $K[x]$, such that f and g are coprime and $\deg(f) < \deg(g)$, returns fractions A and B in $K(x)$, such that $f/g = A' + B$ and B has square-free denominator.

Examples

```
>>> from sympy.integrals.rationaltools import ratint_ratpart
>>> from sympy.abc import x, y
>>> from sympy import Poly
>>> ratint_ratpart(Poly(1, x, domain='ZZ'),
... Poly(x + 1, x, domain='ZZ'), x)
(0, 1/(x + 1))
>>> ratint_ratpart(Poly(1, x, domain='EX'),
... Poly(x**2 + y**2, x, domain='EX'), x)
(0, 1/(x**2 + y**2))
>>> ratint_ratpart(Poly(36, x, domain='ZZ'),
... Poly(x**5 - 2*x**4 - 2*x**3 + 4*x**2 + x - 2, x, domain='ZZ'), x)
((12*x + 6)/(x**2 - 1), 12/(x**2 - x - 2))
```

See also:

[*ratint*](#) (page 587), [*ratint_logpart*](#) (page 588)

`sympy.integrals.rationaltools.ratint_logpart(f, g, x, t=None)`

Lazard-Rioboo-Trager algorithm.

Explanation

Given a field K and polynomials f and g in $K[x]$, such that f and g are coprime, $\deg(f) < \deg(g)$ and g is square-free, returns a list of tuples (s_i, q_i) of polynomials, for $i = 1..n$, such that s_i in $K[t, x]$ and q_i in $K[t]$, and:

$$\frac{d}{dx} \frac{f}{g} = \sum_{i=1..n} \frac{d}{dx} \left(\frac{s_i}{q_i} \right) + \sum_{i=1..n} a_i \log(s_i(a, x))$$

$\frac{d}{dx} \frac{f}{g} = \sum_{i=1..n} \frac{d}{dx} \left(\frac{s_i}{q_i} \right) + \sum_{i=1..n} a_i \log(s_i(a, x))$

Examples

```
>>> from sympy.integrals.rationaltools import ratint_logpart
>>> from sympy.abc import x
>>> from sympy import Poly
>>> ratint_logpart(Poly(1, x, domain='ZZ'),
... Poly(x**2 + x + 1, x, domain='ZZ'), x)
[(Poly(x + 3*_t/2 + 1/2, x, domain='QQ[_t]'),
...Poly(3*_t**2 + 1, _t, domain='ZZ'))]
>>> ratint_logpart(Poly(12, x, domain='ZZ'),
... Poly(x**2 - x - 2, x, domain='ZZ'), x)
[(Poly(x - 3*_t/8 - 1/2, x, domain='QQ[_t]'),
...Poly(-_t**2 + 16, _t, domain='ZZ'))]
```

See also:

[ratint](#) (page 587), [ratint_ratpart](#) (page 588)

2. [trigintegrate\(\)](#) (page 589) solves integrals of trigonometric functions using pattern matching

`sympy.integrals.trigonometry.trigintegrate(f, x, conds='piecewise')`

Integrate $f = \text{Mul}(\text{trig})$ over x .

Examples

```
>>> from sympy import sin, cos, tan, sec
>>> from sympy.integrals.trigonometry import trigintegrate
>>> from sympy.abc import x
```

```
>>> trigintegrate(sin(x)*cos(x), x)
sin(x)**2/2
```

```
>>> trigintegrate(sin(x)**2, x)
x/2 - sin(x)*cos(x)/2
```

```
>>> trigintegrate(tan(x)*sec(x), x)
1/cos(x)
```

```
>>> trigintegrate(sin(x)*tan(x), x)
-log(sin(x) - 1)/2 + log(sin(x) + 1)/2 - sin(x)
```

See also:

[*sympy.integrals.integrals.Integral.doit*](#) (page 603), [*sympy.integrals.integrals.Integral*](#) (page 601)

References

[R526]

3. [*deltaintegrate\(\)*](#) (page 590) solves integrals with [*DiracDelta*](#) (page 450) objects.
`sympy.integrals.deltafunctions.deltaintegrate(f, x)`

Explanation

The idea for integration is the following:

- If we are dealing with a `DiracDelta` expression, i.e. `DiracDelta(g(x))`, we try to simplify it.

If we could simplify it, then we integrate the resulting expression. We already know we can integrate a simplified expression, because only simple `DiracDelta` expressions are involved.

If we couldn't simplify it, there are two cases:

- 1) The expression is a simple expression: we return the integral, taking care if we are dealing with a `Derivative` or with a proper `DiracDelta`.
- 2) The expression is not simple (i.e. `DiracDelta(cos(x))`): we can do nothing at all.

- If the node is a multiplication node having a `DiracDelta` term:

First we expand it.

If the expansion did work, then we try to integrate the expansion.

If not, we try to extract a simple `DiracDelta` term, then we have two cases:

- 1) We have a simple `DiracDelta` term, so we return the integral.
- 2) We didn't have a simple term, but we do have an expression with simplified `DiracDelta` terms, so we integrate this expression.

Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.integrals.deltafunctions import deltaintegrate
>>> from sympy import sin, cos, DiracDelta
>>> deltaintegrate(x*sin(x)*cos(x)*DiracDelta(x - 1), x)
sin(1)*cos(1)*Heaviside(x - 1)
>>> deltaintegrate(y**2*DiracDelta(x - z)*DiracDelta(y - z), y)
z**2*DiracDelta(x - z)*Heaviside(y - z)
```

See also:

[*sympy.functions.special.delta_functions.DiracDelta*](#) (page 450), [*sympy.integrals.integrals.Integral*](#) (page 601)

4. [*singularityintegrate\(\)*](#) (page 591) is applied if the function contains a [*SingularityFunction*](#) (page 456)

```
sympy.integrals.singularityfunctions.singularityintegrate(f, x)
```

This function handles the indefinite integrations of Singularity functions. The `integrate` function calls this function internally whenever an instance of `SingularityFunction` is passed as argument.

Explanation

The idea for integration is the following:

- If we are dealing with a `SingularityFunction` expression, i.e. `SingularityFunction(x, a, n)`, we just return `SingularityFunction(x, a, n + 1)/(n + 1)` if $n \geq 0$ and `SingularityFunction(x, a, n + 1)` if $n < 0$.
- If the node is a multiplication or power node having a `SingularityFunction` term we rewrite the whole expression in terms of `Heaviside` and `DiracDelta` and then integrate the output. Lastly, we rewrite the output of integration back in terms of `SingularityFunction`.
- If none of the above case arises, we return `None`.

Examples

```
>>> from sympy.integrals.singularityfunctions import _
↳ singularityintegrate
>>> from sympy import SingularityFunction, symbols, Function
>>> x, a, n, y = symbols('x a n y')
>>> f = Function('f')
>>> singularityintegrate(SingularityFunction(x, a, 3), x)
SingularityFunction(x, a, 4)/4
>>> singularityintegrate(5*SingularityFunction(x, 5, -2), x)
5*SingularityFunction(x, 5, -1)
>>> singularityintegrate(6*SingularityFunction(x, 5, -1), x)
6*SingularityFunction(x, 5, 0)
>>> singularityintegrate(x*SingularityFunction(x, 0, -1), x)
```

(continues on next page)

(continued from previous page)

```
0
>>> singularityintegrate(SingularityFunction(x, 1, -1) * f(x), x)
f(1)*SingularityFunction(x, 1, 0)
```

5. If the heuristic algorithms cannot be applied, `risch_integrate()` (page 592) is tried next. The *Risch algorithm* is a general method for calculating antiderivatives of elementary functions. The Risch algorithm is a decision procedure that can determine whether an elementary solution exists, and in that case calculate it. It can be extended to handle many nonelementary functions in addition to the elementary ones. However, the version implemented in SymPy only supports a small subset of the full algorithm, particularly, on part of the transcendental algorithm for exponentials and logarithms is implemented. An advantage of `risch_integrate()` (page 592) over other methods is that if it returns an instance of `NonElementaryIntegral` (page 594), the integral is proven to be nonelementary by the algorithm, meaning the integral cannot be represented using a combination of exponentials, logarithms, trig functions, powers, rational functions, algebraic functions, and function composition.

```
sympy.integrals.risch.risch_integrate(f, x, extension=None, handle_first='log',
                                       separate_integral=False,
                                       rewrite_complex=None,
                                       conds='piecewise')
```

The Risch Integration Algorithm.

Explanation

Only transcendental functions are supported. Currently, only exponentials and logarithms are supported, but support for trigonometric functions is forthcoming.

If this function returns an unevaluated `Integral` in the result, it means that it has proven that integral to be nonelementary. Any errors will result in raising `NotImplementedError`. The unevaluated `Integral` will be an instance of `NonElementaryIntegral`, a subclass of `Integral`.

`handle_first` may be either `'exp'` or `'log'`. This changes the order in which the extension is built, and may result in a different (but equivalent) solution (for an example of this, see issue 5109). It is also possible that the integral may be computed with one but not the other, because not all cases have been implemented yet. It defaults to `'log'` so that the outer extension is exponential when possible, because more of the exponential case has been implemented.

If `separate_integral` is `True`, the result is returned as a tuple `(ans, i)`, where the integral is `ans + i`, `ans` is elementary, and `i` is either a `NonElementaryIntegral` or `0`. This is useful if you want to try further integrating the `NonElementaryIntegral` part using other algorithms to possibly get a solution in terms of special functions. It is `False` by default.

Examples

```
>>> from sympy.integrals.risch import risch_integrate
>>> from sympy import exp, log, pprint
>>> from sympy.abc import x
```

First, we try integrating $\exp(-x^2)$. Except for a constant factor of $2/\sqrt{\pi}$, this is the famous error function.

```
>>> pprint(risch_integrate(exp(-x**2), x))
/
|
|      2
|    -x
|   e    dx
|
/
```

The unevaluated Integral in the result means that `risch_integrate()` has proven that $\exp(-x^2)$ does not have an elementary anti-derivative.

In many cases, `risch_integrate()` can split out the elementary anti-derivative part from the nonelementary anti-derivative part. For example,

```
>>> pprint(risch_integrate((2*log(x)**2 - log(x) - x**2)/(log(x)**3 -
... x**2*log(x)), x))
      log(-x + log(x))      log(x + log(x))      /
----- + ----- + |  1
      2              2      | ----- dx
                        |  log(x)
                        |
                        /
```

This means that it has proven that the integral of $1/\log(x)$ is nonelementary. This function is also known as the logarithmic integral, and is often denoted as $\text{Li}(x)$.

`risch_integrate()` currently only accepts purely transcendental functions with exponentials and logarithms, though note that this can include nested exponentials and logarithms, as well as exponentials with bases other than E .

```
>>> pprint(risch_integrate(exp(x)*exp(exp(x)), x))
/ x\
\e /
e
>>> pprint(risch_integrate(exp(exp(x)), x))
/
|
|      / x\
|     \e /
|    e    dx
|
/
```

```
>>> pprint(risch_integrate(x*x**x*log(x) + x**x + x*x**x, x))
      x
x*x
>>> pprint(risch_integrate(x**x, x))
/
|
|      x
|      x  dx
|
/
```

```
>>> pprint(risch_integrate(-1/(x*log(x)*log(log(x))**2), x))
      1
-----
log(log(x))
```

class `sympy.integrals.risch.NonElementaryIntegral`(*function*, **symbols*, ***assumptions*)

Represents a nonelementary Integral.

Explanation

If the result of `integrate()` is an instance of this class, it is guaranteed to be nonelementary. Note that `integrate()` by default will try to find any closed-form solution, even in terms of special functions which may themselves not be elementary. To make `integrate()` only give elementary solutions, or, in the cases where it can prove the integral to be nonelementary, instances of this class, use `integrate(risch=True)`. In this case, `integrate()` may raise `NotImplementedError` if it cannot make such a determination.

`integrate()` uses the deterministic Risch algorithm to integrate elementary functions or prove that they have no elementary integral. In some cases, this algorithm can split an integral into an elementary and nonelementary part, so that the result of `integrate` will be the sum of an elementary expression and a `NonElementaryIntegral`.

Examples

```
>>> from sympy import integrate, exp, log, Integral
>>> from sympy.abc import x
```

```
>>> a = integrate(exp(-x**2), x, risch=True)
>>> print(a)
Integral(exp(-x**2), x)
>>> type(a)
<class 'sympy.integrals.risch.NonElementaryIntegral'>
```

```
>>> expr = (2*log(x)**2 - log(x) - x**2)/(log(x)**3 - x**2*log(x))
>>> b = integrate(expr, x, risch=True)
>>> print(b)
```

(continues on next page)

(continued from previous page)

```
-log(-x + log(x))/2 + log(x + log(x))/2 + Integral(1/log(x), x)
>>> type(b.atoms(Integral).pop())
<class 'sympy.integrals.risch.NonElementaryIntegral'>
```

6. For non-elementary definite integrals, SymPy uses so-called Meijer G-functions. Details are described in [Computing Integrals using Meijer G-Functions](#) (page 554).
7. All the algorithms mentioned thus far are either pattern-matching based heuristic, or solve integrals using algorithms that are much different from the way most people are taught in their calculus courses. SymPy also implements a method that can solve integrals in much the same way you would in calculus. The advantage of this method is that it is possible to extract the integration steps from, so that one can see how to compute the integral “by hand”. This is used by [SymPy Gamma](#). This is implemented in the [manualintegrate\(\)](#) (page 595) function. The steps for an integral can be seen with the [integral_steps\(\)](#) (page 596) function.

```
sympy.integrals.manualintegrate.manualintegrate(f, var)
```

Explanation

Compute indefinite integral of a single variable using an algorithm that resembles what a student would do by hand.

Unlike [integrate\(\)](#) (page 598), var can only be a single symbol.

Examples

```
>>> from sympy import sin, cos, tan, exp, log, integrate
>>> from sympy.integrals.manualintegrate import manualintegrate
>>> from sympy.abc import x
>>> manualintegrate(1 / x, x)
log(x)
>>> integrate(1/x)
log(x)
>>> manualintegrate(log(x), x)
x*log(x) - x
>>> integrate(log(x))
x*log(x) - x
>>> manualintegrate(exp(x) / (1 + exp(2 * x)), x)
atan(exp(x))
>>> integrate(exp(x) / (1 + exp(2 * x)))
RootSum(4*_z**2 + 1, Lambda(_i, _i*log(2*_i + exp(x))))
>>> manualintegrate(cos(x)**4 * sin(x), x)
-cos(x)**5/5
>>> integrate(cos(x)**4 * sin(x), x)
-cos(x)**5/5
>>> manualintegrate(cos(x)**4 * sin(x)**3, x)
cos(x)**7/7 - cos(x)**5/5
>>> integrate(cos(x)**4 * sin(x)**3, x)
cos(x)**7/7 - cos(x)**5/5
>>> manualintegrate(tan(x), x)
```

(continues on next page)

(continued from previous page)

```
-log(cos(x))
>>> integrate(tan(x), x)
-log(cos(x))
```

See also:

[`sympy.integrals.integrals.integrate`](#) (page 598), [`sympy.integrals.integrals.Integral.doit`](#) (page 603), [`sympy.integrals.integrals.Integral`](#) (page 601)

`sympy.integrals.manualintegrate.integral_steps(integrand, symbol, **options)`

Returns the steps needed to compute an integral.

Returns

rule : namedtuple

The first step; most rules have substeps that must also be considered. These substeps can be evaluated using `manualintegrate` to obtain a result.

Explanation

This function attempts to mirror what a student would do by hand as closely as possible.

SymPy Gamma uses this to provide a step-by-step explanation of an integral. The code it uses to format the results of this function can be found at https://github.com/sympy/sympy_gamma/blob/master/app/logic/intsteps.py.

Examples

```
>>> from sympy import exp, sin
>>> from sympy.integrals.manualintegrate import integral_steps
>>> from sympy.abc import x
>>> print(repr(integral_steps(exp(x) / (1 + exp(2 * x)), x)))
URule(u_var=_u, u_func=exp(x), constant=1,
substep=ArctanRule(a=1, b=1, c=1, context=1/(_u**2 + 1), symbol=_u),
context=exp(x)/(exp(2*x) + 1), symbol=x)
>>> print(repr(integral_steps(sin(x), x)))
TrigRule(func='sin', arg=x, context=sin(x), symbol=x)
>>> print(repr(integral_steps((x**2 + 3)**2, x)))
RewriteRule(rewritten=x**4 + 6*x**2 + 9,
substep=AddRule(substeps=[PowerRule(base=x, exp=4, context=x**4,
↳ symbol=x),
    ConstantTimesRule(constant=6, other=x**2,
        substep=PowerRule(base=x, exp=2, context=x**2, symbol=x),
        context=6*x**2, symbol=x),
    ConstantRule(constant=9, context=9, symbol=x)],
context=x**4 + 6*x**2 + 9, symbol=x), context=(x**2 + 3)**2, symbol=x)
```

8. Finally, if all the above fail, SymPy also uses a simplified version of the Risch algorithm, called the *Risch-Norman algorithm*. This algorithm is tried last because it is often the slowest to compute. This is implemented in [`heurisch\(\)`](#) (page 596):

```
sympy.integrals.heurisch.heurisch(f, x, rewrite=False, hints=None,
                                     mappings=None, retries=3, degree_offset=0,
                                     unnecessary_permutations=None,
                                     _try_heurisch=None)
```

Compute indefinite integral using heuristic Risch algorithm.

Explanation

This is a heuristic approach to indefinite integration in finite terms using the extended heuristic (parallel) Risch algorithm, based on Manuel Bronstein's "Poor Man's Integrator".

The algorithm supports various classes of functions including transcendental elementary or special functions like Airy, Bessel, Whittaker and Lambert.

Note that this algorithm is not a decision procedure. If it isn't able to compute the antiderivative for a given function, then this is not a proof that such a function does not exist. One should use recursive Risch algorithm in such case. It's an open question if this algorithm can be made a full decision procedure.

This is an internal integrator procedure. You should use top level 'integrate' function in most cases, as this procedure needs some preprocessing steps and otherwise may fail.

Specification

```
heurisch(f, x, rewrite=False, hints=None)
```

where

f : expression x : symbol

rewrite -> force rewrite 'f' in terms of 'tan' and 'tanh' hints -> a list of functions that may appear in anti-derivate

- hints = None -> no suggestions at all
- hints = [] -> try to figure out
- hints = [f1, ..., fn] -> we know better

Examples

```
>>> from sympy import tan
>>> from sympy.integrals.heurisch import heurisch
>>> from sympy.abc import x, y
```

```
>>> heurisch(y*tan(x), x)
y*log(tan(x)**2 + 1)/2
```

See Manuel Bronstein's "Poor Man's Integrator":

See also:

[sympy.integrals.integrals.Integral.doit](#) (page 603), [sympy.integrals.integrals.Integral](#) (page 601), [sympy.integrals.heurisch.components](#) (page 598)

References

For more information on the implemented algorithm refer to:

[\[R527\]](#), [\[R528\]](#), [\[R529\]](#), [\[R530\]](#), [\[R531\]](#)

`sympy.integrals.heurisch.components(f, x)`

Returns a set of all functional components of the given expression which includes symbols, function applications and compositions and non-integer powers. Fractional powers are collected with minimal, positive exponents.

Examples

```
>>> from sympy import cos, sin
>>> from sympy.abc import x
>>> from sympy.integrals.heurisch import components
```

```
>>> components(sin(x)*cos(x)**2, x)
{x, sin(x), cos(x)}
```

See also:

[heurisch](#) (page 596)

API reference

`sympy.integrals.integrals.integrate(f, var, ...)`

Deprecated since version 1.6: Using `integrate()` with [Poly](#) (page 2378) is deprecated. Use [Poly.integrate\(\)](#) (page 2398) instead. See [Using integrate with Poly](#) (page 176).

Explanation

Compute definite or indefinite integral of one or more variables using Risch-Norman algorithm and table lookup. This procedure is able to handle elementary algebraic and transcendental functions and also a huge class of special functions, including Airy, Bessel, Whittaker and Lambert.

var can be:

- a symbol - indefinite integration
- **a tuple (symbol, a) - indefinite integration with result**
given with a replacing symbol
- a tuple (symbol, a, b) - definite integration

Several variables can be specified, in which case the result is multiple integration. (If var is omitted and the integrand is univariate, the indefinite integral in that variable will be performed.)

Indefinite integrals are returned without terms that are independent of the integration variables. (see examples)

Definite improper integrals often entail delicate convergence conditions. Pass `conds='piecewise'`, `'separate'` or `'none'` to have these returned, respectively, as a Piecewise function, as a separate result (i.e. result will be a tuple), or not at all (default is `'piecewise'`).

Strategy

SymPy uses various approaches to definite integration. One method is to find an antiderivative for the integrand, and then use the fundamental theorem of calculus. Various functions are implemented to integrate polynomial, rational and trigonometric functions, and integrands containing DiracDelta terms.

SymPy also implements the part of the Risch algorithm, which is a decision procedure for integrating elementary functions, i.e., the algorithm can either find an elementary antiderivative, or prove that one does not exist. There is also a (very successful, albeit somewhat slow) general implementation of the heuristic Risch algorithm. This algorithm will eventually be phased out as more of the full Risch algorithm is implemented. See the docstring of `Integral._eval_integral()` for more details on computing the antiderivative using algebraic methods.

The option `risch=True` can be used to use only the (full) Risch algorithm. This is useful if you want to know if an elementary function has an elementary antiderivative. If the indefinite `Integral` returned by this function is an instance of `NonElementaryIntegral`, that means that the Risch algorithm has proven that integral to be non-elementary. Note that by default, additional methods (such as the Meijer G method outlined below) are tried on these integrals, as they may be expressible in terms of special functions, so if you only care about elementary answers, use `risch=True`. Also note that an unevaluated `Integral` returned by this function is not necessarily a `NonElementaryIntegral`, even with `risch=True`, as it may just be an indication that the particular part of the Risch algorithm needed to integrate that function is not yet implemented.

Another family of strategies comes from re-writing the integrand in terms of so-called Meijer G-functions. Indefinite integrals of a single G-function can always be computed, and the definite integral of a product of two G-functions can be computed from zero to infinity. Various strategies are implemented to rewrite integrands as G-functions, and use this information to compute integrals (see the `meijerint` module).

The option `manual=True` can be used to use only an algorithm that tries to mimic integration by hand. This algorithm does not handle as many integrands as the other algorithms implemented but may return results in a more familiar form. The `manualintegrate` module has functions that return the steps used (see the module docstring for more information).

In general, the algebraic methods work best for computing antiderivatives of (possibly complicated) combinations of elementary functions. The G-function methods work best for computing definite integrals from zero to infinity of moderately complicated combinations of special functions, or indefinite integrals of very simple combinations of special functions.

The strategy employed by the integration code is as follows:

- If computing a definite integral, and both limits are real, and at least one limit is $+\infty$, try the G-function method of definite integration first.
- Try to find an antiderivative, using all available methods, ordered by performance (that is try fastest method first, slowest last; in particular polynomial integration is tried first, Meijer G-functions second to last, and heuristic Risch last).
- If still not successful, try G-functions irrespective of the limits.

The option `meijerg=True, False, None` can be used to, respectively: always use G-function methods and no others, never use G-function methods, or use all available methods (in order as described above). It defaults to `None`.

Examples

```
>>> from sympy import integrate, log, exp, oo
>>> from sympy.abc import a, x, y
```

```
>>> integrate(x*y, x)
x**2*y/2
```

```
>>> integrate(log(x), x)
x*log(x) - x
```

```
>>> integrate(log(x), (x, 1, a))
a*log(a) - a + 1
```

```
>>> integrate(x)
x**2/2
```

Terms that are independent of x are dropped by indefinite integration:

```
>>> from sympy import sqrt
>>> integrate(sqrt(1 + x), (x, 0, x))
2*(x + 1)**(3/2)/3 - 2/3
>>> integrate(sqrt(1 + x), x)
2*(x + 1)**(3/2)/3
```

```
>>> integrate(x*y)
Traceback (most recent call last):
...
ValueError: specify integration variables to integrate x*y
```

Note that `integrate(x)` syntax is meant only for convenience in interactive sessions and should be avoided in library code.

```
>>> integrate(x**a*exp(-x), (x, 0, oo)) # same as conds='piecewise'
Piecewise((gamma(a + 1), re(a) > -1),
          (Integral(x**a*exp(-x), (x, 0, oo)), True))
```

```
>>> integrate(x**a*exp(-x), (x, 0, oo), conds='none')
gamma(a + 1)
```



```
>>> integrate(x**a*exp(-x), (x, 0, oo), conds='separate')
(gamma(a + 1), re(a) > -1)
```

See also:

[*Integral*](#) (page 601), [*Integral.doit*](#) (page 603)

`sympy.integrals.integrals.line_integrate(field, Curve, variables)`

Compute the line integral.

Examples

```
>>> from sympy import Curve, line_integrate, E, ln
>>> from sympy.abc import x, y, t
>>> C = Curve([E**t + 1, E**t - 1], (t, 0, ln(2)))
>>> line_integrate(x + y, C, [x, y])
3*sqrt(2)
```

See also:

[*sympy.integrals.integrals.integrate*](#) (page 598), [*Integral*](#) (page 601)

The class [*Integral*](#) (page 601) represents an unevaluated integral and has some methods that help in the integration of an expression.

class `sympy.integrals.integrals.Integral(function, *symbols, **assumptions)`

Represents unevaluated integral.

is_commutative

Returns whether all the free symbols in the integral are commutative.

as_sum(*n=None*, *method='midpoint'*, *evaluate=True*)

Approximates a definite integral by a sum.

Parameters**n :**

The number of subintervals to use, optional.

method :

One of: 'left', 'right', 'midpoint', 'trapezoid'.

evaluate : bool

If False, returns an unevaluated Sum expression. The default is True, evaluate the sum.

Notes

These methods of approximate integration are described in [1].

Examples

```
>>> from sympy import Integral, sin, sqrt
>>> from sympy.abc import x, n
>>> e = Integral(sin(x), (x, 3, 7))
>>> e
Integral(sin(x), (x, 3, 7))
```

For demonstration purposes, this interval will only be split into 2 regions, bounded by [3, 5] and [5, 7].

The left-hand rule uses function evaluations at the left of each interval:

```
>>> e.as_sum(2, 'left')
2*sin(5) + 2*sin(3)
```

The midpoint rule uses evaluations at the center of each interval:

```
>>> e.as_sum(2, 'midpoint')
2*sin(4) + 2*sin(6)
```

The right-hand rule uses function evaluations at the right of each interval:

```
>>> e.as_sum(2, 'right')
2*sin(5) + 2*sin(7)
```

The trapezoid rule uses function evaluations on both sides of the intervals. This is equivalent to taking the average of the left and right hand rule results:

```
>>> e.as_sum(2, 'trapezoid')
2*sin(5) + sin(3) + sin(7)
>>> (e.as_sum(2, 'left') + e.as_sum(2, 'right'))/2 == _
True
```

Here, the discontinuity at $x = 0$ can be avoided by using the midpoint or right-hand method:

```
>>> e = Integral(1/sqrt(x), (x, 0, 1))
>>> e.as_sum(5).n(4)
1.730
>>> e.as_sum(10).n(4)
1.809
>>> e.doit().n(4) # the actual value is 2
2.000
```

The left- or trapezoid method will encounter the discontinuity and return infinity:

```
>>> e.as_sum(5, 'left')
zoo
```

The number of intervals can be symbolic. If omitted, a dummy symbol will be used for it.

```
>>> e = Integral(x**2, (x, 0, 2))
>>> e.as_sum(n, 'right').expand()
8/3 + 4/n + 4/(3*n**2)
```

This shows that the midpoint rule is more accurate, as its error term decays as the square of n :

```
>>> e.as_sum(method='midpoint').expand()
8/3 - 2/(3*_n**2)
```

A symbolic sum is returned with `evaluate=False`:

```
>>> e.as_sum(n, 'midpoint', evaluate=False)
2*Sum((2*_k/n - 1/n)**2, (_k, 1, n))/n
```

See also:

***Integral.doit* (page 603)**

Perform the integration using any hints

References

[R532]

doit(hints)**

Perform the integration using any hints given.

Examples

```
>>> from sympy import Piecewise, S
>>> from sympy.abc import x, t
>>> p = x**2 + Piecewise((0, x/t < 0), (1, True))
>>> p.integrate((t, S(4)/5, 1), (x, -1, 1))
1/3
```

See also:

sympy.integrals.trigonometry.trigintegrate (page 589), *sympy.integrals.heurisch.heurisch* (page 596), *sympy.integrals.rationaltools.ratint* (page 587)

***as_sum* (page 601)**

Approximate the integral using a sum

property free_symbols

This method returns the symbols that will exist when the integral is evaluated. This is useful if one is trying to determine whether an integral depends on a certain symbol or not.

Examples

```
>>> from sympy import Integral
>>> from sympy.abc import x, y
>>> Integral(x, (x, y, 1)).free_symbols
{y}
```

See also:

[*sympy.concrete.expr_with_limits.ExprWithLimits.function*](#) (page 607), [*sympy.concrete.expr_with_limits.ExprWithLimits.limits*](#) (page 608), [*sympy.concrete.expr_with_limits.ExprWithLimits.variables*](#) (page 608)

principal_value(**kwargs)

Compute the Cauchy Principal Value of the definite integral of a real function in the given interval on the real axis.

Explanation

In mathematics, the Cauchy principal value, is a method for assigning values to certain improper integrals which would otherwise be undefined.

Examples

```
>>> from sympy import Integral, oo
>>> from sympy.abc import x
>>> Integral(x+1, (x, -oo, oo)).principal_value()
oo
>>> f = 1 / (x**3)
>>> Integral(f, (x, -oo, oo)).principal_value()
0
>>> Integral(f, (x, -10, 10)).principal_value()
0
>>> Integral(f, (x, -10, oo)).principal_value() + Integral(f, (x, -oo,
→ 10)).principal_value()
0
```

References

[R533], [R534]

transform(x , u)

Performs a change of variables from x to u using the relationship given by x and u which will define the transformations f and F (which are inverses of each other) as follows:

- 1) If x is a Symbol (which is a variable of integration) then u will be interpreted as some function, $f(u)$, with inverse $F(u)$. This, in effect, just makes the substitution of x with $f(x)$.
- 2) If u is a Symbol then x will be interpreted as some function, $F(x)$, with inverse $f(u)$. This is commonly referred to as u -substitution.

Once f and F have been identified, the transformation is made as follows:

$$\int_a^b x dx \rightarrow \int_{F(a)}^{F(b)} f(x) \frac{d}{dx}$$

where $F(x)$ is the inverse of $f(x)$ and the limits and integrand have been corrected so as to retain the same value after integration.

Notes

The mappings, $F(x)$ or $f(u)$, must lead to a unique integral. Linear or rational linear expression, $2*x$, $1/x$ and $\text{sqrt}(x)$, will always work; quadratic expressions like $x**2 - 1$ are acceptable as long as the resulting integrand does not depend on the sign of the solutions (see examples).

The integral will be returned unchanged if x is not a variable of integration.

x must be (or contain) only one of the integration variables. If u has more than one free symbol then it should be sent as a tuple $(u, uvar)$ where $uvar$ identifies which variable is replacing the integration variable. XXX can it contain another integration variable?

Examples

```
>>> from sympy.abc import a, x, u
>>> from sympy import Integral, cos, sqrt
```

```
>>> i = Integral(x*cos(x**2 - 1), (x, 0, 1))
```

transform can change the variable of integration

```
>>> i.transform(x, u)
Integral(u*cos(u**2 - 1), (u, 0, 1))
```

transform can perform u-substitution as long as a unique integrand is obtained:

```
>>> i.transform(x**2 - 1, u)
Integral(cos(u)/2, (u, -1, 0))
```

This attempt fails because $x = \pm\sqrt{u + 1}$ and the sign does not cancel out of the integrand:

```
>>> Integral(cos(x**2 - 1), (x, 0, 1)).transform(x**2 - 1, u)
Traceback (most recent call last):
...
ValueError:
The mapping between F(x) and f(u) did not give a unique integrand.
```

transform can do a substitution. Here, the previous result is transformed back into the original expression using “u-substitution”:

```
>>> ui = _
>>> _.transform(sqrt(u + 1), x) == i
True
```

We can accomplish the same with a regular substitution:

```
>>> ui.transform(u, x**2 - 1) == i
True
```

If the x does not contain a symbol of integration then the integral will be returned unchanged. Integral i does not have an integration variable a so no change is made:

```
>>> i.transform(a, x) == i
True
```

When u has more than one free symbol the symbol that is replacing x must be identified by passing u as a tuple:

```
>>> Integral(x, (x, 0, 1)).transform(x, (u + a, u))
Integral(a + u, (u, -a, 1 - a))
>>> Integral(x, (x, 0, 1)).transform(x, (u + a, a))
Integral(a + u, (a, -u, 1 - u))
```

See also:

[*sympy.concrete.expr_with_limits.ExprWithLimits.variables*](#) (page 608)

Lists the integration variables

[*as_dummy*](#) (page 929)

Replace integration variables with dummy ones

Integral (page 601) subclasses from *ExprWithLimits* (page 606), which is a common superclass of *Integral* (page 601) and *Sum* (page 900).

```
class sympy.concrete.expr_with_limits.ExprWithLimits(function, *symbols,
**assumptions)
```

property bound_symbols

Return only variables that are dummy variables.

Examples

```
>>> from sympy import Integral
>>> from sympy.abc import x, i, j, k
>>> Integral(x**i, (i, 1, 3), (j, 2), k).bound_symbols
[i, j]
```

See also:

[*function*](#) (page 607), [*limits*](#) (page 608), [*free_symbols*](#) (page 606)

[*as_dummy*](#) (page 929)

Rename dummy variables

[*sympy.integrals.integrals.Integral.transform*](#) (page 604)

Perform mapping on the dummy variable

property free_symbols

This method returns the symbols in the object, excluding those that take on a specific value (i.e. the dummy symbols).

Examples

```
>>> from sympy import Sum
>>> from sympy.abc import x, y
>>> Sum(x, (x, y, 1)).free_symbols
{y}
```

property function

Return the function applied across limits.

Examples

```
>>> from sympy import Integral
>>> from sympy.abc import x
>>> Integral(x**2, (x,)).function
x**2
```

See also:

[limits](#) (page 608), [variables](#) (page 608), [free_symbols](#) (page 606)

property has_finite_limits

Returns True if the limits are known to be finite, either by the explicit bounds, assumptions on the bounds, or assumptions on the variables. False if known to be infinite, based on the bounds. None if not enough information is available to determine.

Examples

```
>>> from sympy import Sum, Integral, Product, oo, Symbol
>>> x = Symbol('x')
>>> Sum(x, (x, 1, 8)).has_finite_limits
True
```

```
>>> Integral(x, (x, 1, oo)).has_finite_limits
False
```

```
>>> M = Symbol('M')
>>> Sum(x, (x, 1, M)).has_finite_limits
```

```
>>> N = Symbol('N', integer=True)
>>> Product(x, (x, 1, N)).has_finite_limits
True
```

See also:

[has_reversed_limits](#) (page 607)

property has_reversed_limits

Returns True if the limits are known to be in reversed order, either by the explicit bounds, assumptions on the bounds, or assumptions on the variables. False if known

to be in normal order, based on the bounds. None if not enough information is available to determine.

Examples

```
>>> from sympy import Sum, Integral, Product, oo, Symbol
>>> x = Symbol('x')
>>> Sum(x, (x, 8, 1)).has_reversed_limits
True
```

```
>>> Sum(x, (x, 1, oo)).has_reversed_limits
False
```

```
>>> M = Symbol('M')
>>> Integral(x, (x, 1, M)).has_reversed_limits
```

```
>>> N = Symbol('N', integer=True, positive=True)
>>> Sum(x, (x, 1, N)).has_reversed_limits
False
```

```
>>> Product(x, (x, 2, N)).has_reversed_limits
```

```
>>> Product(x, (x, 2, N)).subs(N, N + 2).has_reversed_limits
False
```

See also:

[*sympy.concrete.expr_with_intlimits.ExprWithIntLimits.has_empty_sequence*](#) (page 912)

property `is_number`

Return True if the Sum has no free symbols, else False.

property `limits`

Return the limits of expression.

Examples

```
>>> from sympy import Integral
>>> from sympy.abc import x, i
>>> Integral(x**i, (i, 1, 3)).limits
((i, 1, 3),)
```

See also:

[*function*](#) (page 607), [*variables*](#) (page 608), [*free_symbols*](#) (page 606)

property `variables`

Return a list of the limit variables.


```
>>> from sympy import Sum
>>> from sympy.abc import x, i
>>> Sum(x**i, (i, 1, 3)).variables
[i]
```

See also:

function (page 607), *limits* (page 608), *free_symbols* (page 606)

as_dummy (page 929)

Rename dummy variables

sympy.integrals.integrals.Integral.transform (page 604)

Perform mapping on the dummy variable

TODO and Bugs

There are still lots of functions that SymPy does not know how to integrate. For bugs related to this module, see <https://github.com/sympy/sympy/issues?q=is%3Aissue+is%3Aopen+label%3Aintegrals>

Numeric Integrals

SymPy has functions to calculate points and weights for Gaussian quadrature of any order and any precision:

`sympy.integrals.quadrature.gauss_legendre(n, n_digits)`

Computes the Gauss-Legendre quadrature [R535] points and weights.

Parameters

n :

The order of quadrature.

n_digits :

Number of significant digits of the points and weights to return.

Returns

(x, w) : the x and w are lists of points and weights as Floats.

The points x_i and weights w_i are returned as (x, w) tuple of lists.

Explanation

The Gauss-Legendre quadrature approximates the integral:

$$\int_{-1}^1 f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of P_n and the weights w_i are given by:

$$w_i = \frac{2}{(1 - x_i^2) (P'_n(x_i))^2}$$

Examples

```
>>> from sympy.integrals.quadrature import gauss_legendre
>>> x, w = gauss_legendre(3, 5)
>>> x
[-0.7746, 0, 0.7746]
>>> w
[0.55556, 0.88889, 0.55556]
>>> x, w = gauss_legendre(4, 5)
>>> x
[-0.86114, -0.33998, 0.33998, 0.86114]
>>> w
[0.34785, 0.65215, 0.65215, 0.34785]
```

See also:

[*gauss_laguerre*](#) (page 610), [*gauss_gen_laguerre*](#) (page 612), [*gauss_hermite*](#) (page 611), [*gauss_chebyshev_t*](#) (page 613), [*gauss_chebyshev_u*](#) (page 614), [*gauss_jacobi*](#) (page 615), [*gauss_lobatto*](#) (page 616)

References

[R535], [R536]

`sympy.integrals.quadrature.gauss_laguerre(n, n_digits)`

Computes the Gauss-Laguerre quadrature [R537] points and weights.

Parameters

n :

The order of quadrature.

n_digits :

Number of significant digits of the points and weights to return.

Returns

(x, w) : The x and w are lists of points and weights as Floats.

The points x_i and weights w_i are returned as (x, w) tuple of lists.

Explanation

The Gauss-Laguerre quadrature approximates the integral:

$$\int_0^{\infty} e^{-x} f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of L_n and the weights w_i are given by:

$$w_i = \frac{x_i}{(n+1)^2 (L_{n+1}(x_i))^2}$$

Examples

```
>>> from sympy.integrals.quadrature import gauss_laguerre
>>> x, w = gauss_laguerre(3, 5)
>>> x
[0.41577, 2.2943, 6.2899]
>>> w
[0.71109, 0.27852, 0.010389]
>>> x, w = gauss_laguerre(6, 5)
>>> x
[0.22285, 1.1889, 2.9927, 5.7751, 9.8375, 15.983]
>>> w
[0.45896, 0.417, 0.11337, 0.010399, 0.00026102, 8.9855e-7]
```

See also:

[gauss_legendre](#) (page 609), [gauss_gen_laguerre](#) (page 612), [gauss_hermite](#) (page 611), [gauss_chebyshev_t](#) (page 613), [gauss_chebyshev_u](#) (page 614), [gauss_jacobi](#) (page 615), [gauss_lobatto](#) (page 616)

References

[R537], [R538]

`sympy.integrals.quadrature.gauss_hermite(n, n_digits)`

Computes the Gauss-Hermite quadrature [R539] points and weights.

Parameters

n :

The order of quadrature.

n_digits :

Number of significant digits of the points and weights to return.

Returns

(x, w) : The x and w are lists of points and weights as Floats.

The points x_i and weights w_i are returned as (x, w) tuple of lists.

Explanation

The Gauss-Hermite quadrature approximates the integral:

$$\int_{-\infty}^{\infty} e^{-x^2} f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of H_n and the weights w_i are given by:

$$w_i = \frac{2^{n-1} n! \sqrt{\pi}}{n^2 (H_{n-1}(x_i))^2}$$

Examples

```
>>> from sympy.integrals.quadrature import gauss_hermite
>>> x, w = gauss_hermite(3, 5)
>>> x
[-1.2247, 0, 1.2247]
>>> w
[0.29541, 1.1816, 0.29541]
```

```
>>> x, w = gauss_hermite(6, 5)
>>> x
[-2.3506, -1.3358, -0.43608, 0.43608, 1.3358, 2.3506]
>>> w
[0.00453, 0.15707, 0.72463, 0.72463, 0.15707, 0.00453]
```

See also:

[*gauss_legendre*](#) (page 609), [*gauss_laguerre*](#) (page 610), [*gauss_gen_laguerre*](#) (page 612), [*gauss_chebyshev_t*](#) (page 613), [*gauss_chebyshev_u*](#) (page 614), [*gauss_jacobi*](#) (page 615), [*gauss_lobatto*](#) (page 616)

References

[R539], [R540], [R541]

`sympy.integrals.quadrature.gauss_gen_laguerre(n, alpha, n_digits)`

Computes the generalized Gauss-Laguerre quadrature [R542] points and weights.

Parameters

n :

The order of quadrature.

alpha :

The exponent of the singularity, $\alpha > -1$.

n_digits :

Number of significant digits of the points and weights to return.

Returns

(x, w) : the x and w are lists of points and weights as Floats.

The points x_i and weights w_i are returned as (x, w) tuple of lists.

Explanation

The generalized Gauss-Laguerre quadrature approximates the integral:

$$\int_0^{\infty} x^{\alpha} e^{-x} f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of L_n^{α} and the weights w_i are given by:

$$w_i = \frac{\Gamma(\alpha + n)}{n\Gamma(n)L_{n-1}^{\alpha}(x_i)L_{n-1}^{\alpha+1}(x_i)}$$

Examples

```
>>> from sympy import S
>>> from sympy.integrals.quadrature import gauss_gen_laguerre
>>> x, w = gauss_gen_laguerre(3, -S.Half, 5)
>>> x
[0.19016, 1.7845, 5.5253]
>>> w
[1.4493, 0.31413, 0.00906]
```

```
>>> x, w = gauss_gen_laguerre(4, 3*S.Half, 5)
>>> x
[0.97851, 2.9904, 6.3193, 11.712]
>>> w
[0.53087, 0.67721, 0.11895, 0.0023152]
```

See also:

[*gauss_legendre*](#) (page 609), [*gauss_laguerre*](#) (page 610), [*gauss_hermite*](#) (page 611), [*gauss_chebyshev_t*](#) (page 613), [*gauss_chebyshev_u*](#) (page 614), [*gauss_jacobi*](#) (page 615), [*gauss_lobatto*](#) (page 616)

References

[R542], [R543]

`sympy.integrals.quadrature.gauss_chebyshev_t(n, n_digits)`

Computes the Gauss-Chebyshev quadrature [R544] points and weights of the first kind.

Parameters

n :

The order of quadrature.

n_digits :

Number of significant digits of the points and weights to return.

Returns

(x, w) : the x and w are lists of points and weights as Floats.

The points x_i and weights w_i are returned as (x, w) tuple of lists.

Explanation

The Gauss-Chebyshev quadrature of the first kind approximates the integral:

$$\int_{-1}^1 \frac{1}{\sqrt{1-x^2}} f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of T_n and the weights w_i are given by:

$$w_i = \frac{\pi}{n}$$

Examples

```
>>> from sympy.integrals.quadrature import gauss_chebyshev_t
>>> x, w = gauss_chebyshev_t(3, 5)
>>> x
[0.86602, 0, -0.86602]
>>> w
[1.0472, 1.0472, 1.0472]
```

```
>>> x, w = gauss_chebyshev_t(6, 5)
>>> x
[0.96593, 0.70711, 0.25882, -0.25882, -0.70711, -0.96593]
>>> w
[0.5236, 0.5236, 0.5236, 0.5236, 0.5236, 0.5236]
```

See also:

[*gauss_legendre*](#) (page 609), [*gauss_laguerre*](#) (page 610), [*gauss_hermite*](#) (page 611), [*gauss_gen_laguerre*](#) (page 612), [*gauss_chebyshev_u*](#) (page 614), [*gauss_jacobi*](#) (page 615), [*gauss_lobatto*](#) (page 616)

References

[R544], [R545]

`sympy.integrals.quadrature.gauss_chebyshev_u(n, n_digits)`

Computes the Gauss-Chebyshev quadrature [R546] points and weights of the second kind.

Parameters

n : the order of quadrature

n_digits : number of significant digits of the points and weights to return

Returns

(x, w) : the x and w are lists of points and weights as Floats.

The points x_i and weights w_i are returned as (x, w) tuple of lists.

Explanation

The Gauss-Chebyshev quadrature of the second kind approximates the integral:

$$\int_{-1}^1 \sqrt{1-x^2} f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of U_n and the weights w_i are given by:

$$w_i = \frac{\pi}{n+1} \sin^2 \left(\frac{i}{n+1} \pi \right)$$

Examples

```
>>> from sympy.integrals.quadrature import gauss_chebyshev_u
>>> x, w = gauss_chebyshev_u(3, 5)
>>> x
[0.70711, 0, -0.70711]
>>> w
[0.3927, 0.7854, 0.3927]
```

```
>>> x, w = gauss_chebyshev_u(6, 5)
>>> x
[0.90097, 0.62349, 0.22252, -0.22252, -0.62349, -0.90097]
>>> w
[0.084489, 0.27433, 0.42658, 0.42658, 0.27433, 0.084489]
```

See also:

[gauss_legendre](#) (page 609), [gauss_laguerre](#) (page 610), [gauss_hermite](#) (page 611), [gauss_gen_laguerre](#) (page 612), [gauss_chebyshev_t](#) (page 613), [gauss_jacobi](#) (page 615), [gauss_lobatto](#) (page 616)

References

[R546], [R547]

`sympy.integrals.quadrature.gauss_jacobi(n, alpha, beta, n_digits)`

Computes the Gauss-Jacobi quadrature [R548] points and weights.

Parameters

n : the order of quadrature

alpha : the first parameter of the Jacobi Polynomial, $\alpha > -1$

beta : the second parameter of the Jacobi Polynomial, $\beta > -1$

n_digits : number of significant digits of the points and weights to return

Returns

(x, w) : the x and w are lists of points and weights as Floats.

The points x_i and weights w_i are returned as (x, w) tuple of lists.

Explanation

The Gauss-Jacobi quadrature of the first kind approximates the integral:

$$\int_{-1}^1 (1-x)^\alpha (1+x)^\beta f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of $P_n^{(\alpha, \beta)}$ and the weights w_i are given by:

$$w_i = -\frac{2n + \alpha + \beta + 2}{n + \alpha + \beta + 1} \frac{\Gamma(n + \alpha + 1)\Gamma(n + \beta + 1)}{\Gamma(n + \alpha + \beta + 1)(n + 1)!} \frac{2^{\alpha + \beta}}{P_n'(x_i)P_{n+1}^{(\alpha, \beta)}(x_i)}$$

Examples

```
>>> from sympy import S
>>> from sympy.integrals.quadrature import gauss_jacobi
>>> x, w = gauss_jacobi(3, S.Half, -S.Half, 5)
>>> x
[-0.90097, -0.22252, 0.62349]
>>> w
[1.7063, 1.0973, 0.33795]
```

```
>>> x, w = gauss_jacobi(6, 1, 1, 5)
>>> x
[-0.87174, -0.5917, -0.2093, 0.2093, 0.5917, 0.87174]
>>> w
[0.050584, 0.22169, 0.39439, 0.39439, 0.22169, 0.050584]
```

See also:

[*gauss_legendre*](#) (page 609), [*gauss_laguerre*](#) (page 610), [*gauss_hermite*](#) (page 611), [*gauss_gen_laguerre*](#) (page 612), [*gauss_chebyshev_t*](#) (page 613), [*gauss_chebyshev_u*](#) (page 614), [*gauss_lobatto*](#) (page 616)

References

[R548], [R549], [R550]

`sympy.integrals.quadrature.gauss_lobatto(n, n_digits)`

Computes the Gauss-Lobatto quadrature [R551] points and weights.

Parameters

n : the order of quadrature

n_digits : number of significant digits of the points and weights to return

Returns

(x, w) : the x and w are lists of points and weights as Floats.

The points x_i and weights w_i are returned as (x, w) tuple of lists.

Explanation

The Gauss-Lobatto quadrature approximates the integral:

$$\int_{-1}^1 f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of $P'_n - 1$ and the weights w_i are given by:

$$w_i = \frac{2}{n(n-1) [P_{n-1}(x_i)]^2}, \quad x \neq \pm 1$$
$$w_i = \frac{2}{n(n-1)}, \quad x = \pm 1$$

Examples

```
>>> from sympy.integrals.quadrature import gauss_lobatto
>>> x, w = gauss_lobatto(3, 5)
>>> x
[-1, 0, 1]
>>> w
[0.33333, 1.3333, 0.33333]
>>> x, w = gauss_lobatto(4, 5)
>>> x
[-1, -0.44721, 0.44721, 1]
>>> w
[0.16667, 0.83333, 0.83333, 0.16667]
```

See also:

[gauss_legendre](#) (page 609), [gauss_laguerre](#) (page 610), [gauss_gen_laguerre](#) (page 612), [gauss_hermite](#) (page 611), [gauss_chebyshev_t](#) (page 613), [gauss_chebyshev_u](#) (page 614), [gauss_jacobi](#) (page 615)

References

[R551], [R552]

Integration over Polytopes

The `intpoly` module in SymPy implements methods to calculate the integral of a polynomial over 2/3-Polytopes. Uses evaluation techniques as described in Chin et al. (2015) [1].

The input for 2-Polytope or Polygon uses the already existing Polygon data structure in SymPy. See [sympy.geometry.polygon](#) (page 2273) for how to create a polygon.

For the 3-Polytope or Polyhedron, the most economical representation is to specify a list of vertices and then to provide each constituting face(Polygon) as a list of vertex indices.

For example, consider the unit cube. Here is how it would be represented.

```
unit_cube = [(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1),
              (1, 1, 0), (1, 1, 1)],
              [3, 7, 6, 2], [1, 5, 7, 3], [5, 4, 6, 7], [0, 4, 5, 1], [2, 0, 1, 3], [2,
              6, 4, 0]]
```

Here, the first sublist is the list of vertices. The other smaller lists such as [3, 7, 6, 2] represent a 2D face of the polyhedra with vertices having index 3, 7, 6 and 2 in the first sublist(in that order).

Principal method in this module is [polytope_integrate\(\)](#) (page 619)

- `polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), x)` returns the integral of x over the triangle with vertices (0, 0), (0, 1) and (1, 0)
- `polytope_integrate(unit_cube, x + y + z)` returns the integral of $x + y + z$ over the unit cube.

References

[1]: Chin, Eric B., Jean B. Lasserre, and N. Sukumar. "Numerical integration of homogeneous functions on convex and nonconvex polygons and polyhedra." Computational Mechanics 56.6 (2015): 967-981

PDF link : <http://dilbert.engr.ucdavis.edu/~suku/quadrature/cls-integration.pdf>

Examples

For 2D Polygons

Single Polynomial:

```
>>> from sympy.integrals.intpoly import *
>>> init_printing(use_unicode=False, wrap_line=False)
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), x)
1/6
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), x + x*y + y**2)
7/24
```

List of specified polynomials:

```
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), [3, x*y + y**2, x**4],
→ max_degree=4)
          4          2
{3: 3/2, x : 1/30, x*y + y : 1/8}
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), [1.125, x, x**2, 6.
→ 89*x**3, x*y + y**2, x**4], max_degree=4)
          2          3 689      4          2
{1.125: 9/16, x: 1/6, x : 1/12, 6.89*x : ----, x : 1/30, x*y + y : 1/8}
          2000
```

Computing all monomials up to a maximum degree:

```
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), max_degree=3)
          2          3          2          3
→ {0: 0, 1: 1/2, x: 1/6, x : 1/12, x : 1/20, y: 1/6, y : 1/12, y : 1/20, x*y: 1/
→ 24, x*y : 1/60, x *y: 1/60}
```

For 3-Polytopes/Polyhedra

Single Polynomial:

```
>>> from sympy.integrals.intpoly import *
>>> cube = [(0, 0, 0), (0, 0, 5), (0, 5, 0), (0, 5, 5), (5, 0, 0), (5, 0, 5),
→ (5, 5, 0), (5, 5, 5)], [2, 6, 7, 3], [3, 7, 5, 1], [7, 6, 4, 5], [1, 5, 4,
→ 0], [3, 1, 0, 2], [0, 4, 6, 2]]
>>> polytope_integrate(cube, x**2 + y**2 + z**2 + x*y + y*z + x*z)
-21875/4
```

(continues on next page)

(continued from previous page)

```
>>> octahedron = [(S(-1) / sqrt(2), 0, 0), (0, S(1) / sqrt(2), 0), (0, 0, S(-1) / sqrt(2)), (0, 0, S(1) / sqrt(2)), (0, S(-1) / sqrt(2), 0), (S(1) / sqrt(2), 0, 0)], [3, 4, 5], [3, 5, 1], [3, 1, 0], [3, 0, 4], [4, 0, 2], [4, 2, 5], [2, 0, 1], [5, 2, 1]]
>>> polytope_integrate(octahedron, x**2 + y**2 + z**2 + x*y + y*z + x*z)
\sqrt{2}
-----
20
```

List of specified polynomials:

```
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), [3, x*y + y**2, x**4],
→ max_degree=4)
4          2
{3: 3/2, x : 1/30, x*y + y : 1/8}
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), [1.125, x, x**2, 6.
→ 89*x**3, x*y + y**2, x**4], max_degree=4)
2          3 689      4          2
{1.125: 9/16, x: 1/6, x : 1/12, 6.89*x : ----, x : 1/30, x*y + y : 1/8}
2000
```

Computing all monomials up to a maximum degree:

```
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), max_degree=3)
2          3          2          3
{0: 0, 1: 1/2, x: 1/6, x : 1/12, x : 1/20, y: 1/6, y : 1/12, y : 1/20, x*y: 1/
→ 24, x*y : 1/60, x *y: 1/60}
```

API reference

`sympy.integrals.intpoly.polytope_integrate(poly, expr=None, *, clockwise=False, max_degree=None)`

Integrates polynomials over 2/3-Polytopes.

Parameters

poly : The input Polygon.

expr : The input polynomial.

clockwise : Binary value to sort input points of 2-Polytope clockwise.(Optional)

max_degree : The maximum degree of any monomial of the input polynomial.(Optional)

Explanation

This function accepts the polytope in `poly` and the function in `expr` (uni/bi/trivariate polynomials are implemented) and returns the exact integral of `expr` over `poly`.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy import Point, Polygon
>>> from sympy.integrals.intpoly import polytope_integrate
>>> polygon = Polygon(Point(0, 0), Point(0, 1), Point(1, 1), Point(1, 0))
>>> polys = [1, x, y, x*y, x**2*y, x*y**2]
>>> expr = x*y
>>> polytope_integrate(polygon, expr)
1/4
>>> polytope_integrate(polygon, polys, max_degree=3)
{1: 1, x: 1/2, y: 1/2, x*y: 1/4, x*y**2: 1/6, x**2*y: 1/6}
```

Series

The series module implements series expansions as a function and many related functions.

Contents

Series Expansions

Limits

The main purpose of this module is the computation of limits.

`sympy.series.limits.limit(e, z, z0, dir='+')`

Computes the limit of $e(z)$ at the point z_0 .

Parameters

e : expression, the limit of which is to be taken

z : symbol representing the variable in the limit.

Other symbols are treated as constants. Multivariate limits are not supported.

z0 : the value toward which z tends. Can be any expression, including ∞ and $-\infty$.

dir : string, optional (default: "+")

The limit is bi-directional if `dir="+-"`, from the right ($z \rightarrow z_0+$) if `dir="+"`, and from the left ($z \rightarrow z_0-$) if `dir="-"`. For infinite z_0 (∞ or $-\infty$), the `dir` argument is determined from the direction of the infinity (i.e., `dir="-"` for ∞).

Examples

```
>>> from sympy import limit, sin, oo
>>> from sympy.abc import x
>>> limit(sin(x)/x, x, 0)
1
>>> limit(1/x, x, 0) # default dir='+'
oo
>>> limit(1/x, x, 0, dir="-")
-oo
>>> limit(1/x, x, 0, dir='+-')
zoo
>>> limit(1/x, x, oo)
0
```

Notes

First we try some heuristics for easy and frequent cases like “ x ”, “ $1/x$ ”, “ $x**2$ ” and similar, so that it’s fast. For all other cases, we use the Gruntz algorithm (see the `gruntz()` function).

See also:

[`limit\_seq` \(page 660\)](#)

returns the limit of a sequence.

class `sympy.series.limits.Limit( $e, z, z0, dir='+'$ )`

Represents an unevaluated limit.

Examples

```
>>> from sympy import Limit, sin
>>> from sympy.abc import x
>>> Limit(sin(x)/x, x, 0)
Limit(sin(x)/x, x, 0)
>>> Limit(1/x, x, 0, dir="-")
Limit(1/x, x, 0, dir='-')
```

doit(***hints*)

Evaluates the limit.

Parameters

deep : bool, optional (default: True)

Invoke the `doit` method of the expressions involved before taking the limit.

hints : optional keyword arguments

To be passed to `doit` methods; only used if `deep` is True.

As is explained above, the workhorse for limit computations is the function `gruntz()` which implements Gruntz’ algorithm for computing limits.

The Gruntz Algorithm

This section explains the basics of the algorithm used for computing limits. Most of the time the `limit()` function should just work. However it is still useful to keep in mind how it is implemented in case something does not work as expected.

First we define an ordering on functions. Suppose $f(x)$ and $g(x)$ are two real-valued functions such that $\lim_{x \rightarrow \infty} f(x) = \infty$ and similarly $\lim_{x \rightarrow \infty} g(x) = \infty$. We shall say that $f(x)$ *dominates* $g(x)$, written $f(x) \succ g(x)$, if for all $a, b \in \mathbb{R}_{>0}$ we have $\lim_{x \rightarrow \infty} \frac{f(x)^a}{g(x)^b} = \infty$. We also say that $f(x)$ and $g(x)$ are *of the same comparability class* if neither $f(x) \succ g(x)$ nor $g(x) \succ f(x)$ and shall denote it as $f(x) \asymp g(x)$.

Note that whenever $a, b \in \mathbb{R}_{>0}$ then $af(x)^b \asymp f(x)$, and we shall use this to extend the definition of $\succ$ to all functions which tend to 0 or $\pm\infty$ as $x \rightarrow \infty$. Thus we declare that $f(x) \asymp 1/f(x)$ and $f(x) \asymp -f(x)$.

It is easy to show the following examples:

- $e^x \succ x^m$
- $e^{x^2} \succ e^{mx}$
- $e^{e^x} \succ e^{x^m}$
- $x^m \asymp x^n$
- $e^{x+\frac{1}{x}} \asymp e^{x+\log x} \asymp e^x$.

From the above definition, it is possible to prove the following property:

Suppose $\omega, g_1, g_2, \dots$ are functions of x , $\lim_{x \rightarrow \infty} \omega = 0$ and $\omega \succ g_i$ for all i . Let $c_1, c_2, \dots \in \mathbb{R}$ with $c_1 < c_2 < \dots$.

Then $\lim_{x \rightarrow \infty} \sum_i g_i \omega^{c_i} = \lim_{x \rightarrow \infty} g_1 \omega^{c_1}$.

For $g_1 = g$ and ω as above we also have the following easy result:

- $\lim_{x \rightarrow \infty} g \omega^c = 0$ for $c > 0$
- $\lim_{x \rightarrow \infty} g \omega^c = \pm\infty$ for $c < 0$, where the sign is determined by the (eventual) sign of g
- $\lim_{x \rightarrow \infty} g \omega^0 = \lim_{x \rightarrow \infty} g$.

Using these results yields the following strategy for computing $\lim_{x \rightarrow \infty} f(x)$:

1. Find the set of *most rapidly varying subexpressions* (MRV set) of $f(x)$. That is, from the set of all subexpressions of $f(x)$, find the elements that are maximal under the relation $\succ$.
2. Choose a function ω that is in the same comparability class as the elements in the MRV set, such that $\lim_{x \rightarrow \infty} \omega = 0$.
3. Expand $f(x)$ as a series in ω in such a way that the antecedents of the above theorem are satisfied.
4. Apply the theorem and conclude the computation of $\lim_{x \rightarrow \infty} f(x)$, possibly by recursively working on $g_1(x)$.

Notes

This exposition glossed over several details. Many are described in the file `gruntz.py`, and all can be found in Gruntz' very readable thesis. The most important points that have not been explained are:

1. Given $f(x)$ and $g(x)$, how do we determine if $f(x) \succ g(x)$, $g(x) \succ f(x)$ or $g(x) \asymp f(x)$?
2. How do we find the MRV set of an expression?
3. How do we compute series expansions?
4. Why does the algorithm terminate?

If you are interested, be sure to take a look at [Gruntz Thesis](#).

Reference

`sympy.series.gruntz.gruntz(e, z, z0, dir='+')`

Compute the limit of $e(z)$ at the point z_0 using the Gruntz algorithm.

Explanation

z_0 can be any expression, including ∞ and $-\infty$.

For `dir="+"` (default) it calculates the limit from the right ($z \rightarrow z_0+$) and for `dir="-"` the limit from the left ($z \rightarrow z_0-$). For infinite z_0 (∞ or $-\infty$), the `dir` argument does not matter.

This algorithm is fully described in the module docstring in the `gruntz.py` file. It relies heavily on the series expansion. Most frequently, `gruntz()` is only used if the faster `limit()` function (which uses heuristics) fails.

`sympy.series.gruntz.compare(a, b, x)`

Returns " $<$ " if $a < b$, " $=$ " for $a == b$, " $>$ " for $a > b$

`sympy.series.gruntz.rewrite(e, Omega, x, wsym)`

$e(x)$... the function Ω ... the mrv set ω ... the symbol which is going to be used for w

Returns the rewritten e in terms of w and $\log(w)$. See `test_rewrite1()` for examples and correct results.

`sympy.series.gruntz.build_expression_tree(Omega, rewrites)`

Helper function for `rewrite`.

We need to sort Ω (mrv set) so that we replace an expression before we replace any expression in terms of which it has to be rewritten:

```
e1 ---> e2 ---> e3
      \
      -> e4
```

Here we can do e_1, e_2, e_3, e_4 or e_1, e_2, e_4, e_3 . To do this we assemble the nodes into a tree, and sort them by height.

This function builds the tree, rewrites then sorts the nodes.

`sympy.series.gruntz.mrv_leadterm(e, x)`

Returns (c0, e0) for e.

`sympy.series.gruntz.calculate_series(e, x, logx=None)`

Calculates at least one term of the series of e in x.

This is a place that fails most often, so it is in its own function.

`sympy.series.gruntz.limitinf(e, x, leadsimp=False)`

Limit e(x) for $x \rightarrow \infty$.

Explanation

If `leadsimp` is True, an attempt is made to simplify the leading term of the series expansion of e. That may succeed even if e cannot be simplified.

`sympy.series.gruntz.sign(e, x)`

Returns a sign of an expression e(x) for $x \rightarrow \infty$.

```
e > 0 for x sufficiently large ... 1
e == 0 for x sufficiently large ... 0
e < 0 for x sufficiently large ... -1
```

The result of this function is currently undefined if e changes sign arbitrarily often for arbitrarily large x (e.g. $\sin(x)$).

Note that this returns zero only if e is *constantly* zero for x sufficiently large. [If e is constant, of course, this is just the same thing as the sign of e.]

`sympy.series.gruntz.mrv(e, x)`

Returns a SubSet of most rapidly varying (mrv) subexpressions of 'e', and e rewritten in terms of these

`sympy.series.gruntz.mrv_max1(f, g, exps, x)`

Computes the maximum of two sets of expressions f and g, which are in the same comparability class, i.e. `mrv_max1()` compares (two elements of) f and g and returns the set, which is in the higher comparability class of the union of both, if they have the same order of variation. Also returns exps, with the appropriate substitutions made.

`sympy.series.gruntz.mrv_max3(f, expsf, g, expsg, union, expsboth, x)`

Computes the maximum of two sets of expressions f and g, which are in the same comparability class, i.e. `max()` compares (two elements of) f and g and returns either (f, expsf) [if f is larger], (g, expsg) [if g is larger] or (union, expsboth) [if f, g are of the same class].

class `sympy.series.gruntz.SubSet`

Stores (expr, dummy) pairs, and how to rewrite expr-s.

Explanation

The gruntz algorithm needs to rewrite certain expressions in term of a new variable w . We cannot use subs, because it is just too smart for us. For example:

```
> Omega=[exp(exp(_p - exp(-_p))/(1 - 1/_p)), exp(exp(_p))]
> O2=[exp(-exp(_p) + exp(-exp(-_p))*exp(_p)/(1 - 1/_p))/_w, 1/_w]
> e = exp(exp(_p - exp(-_p))/(1 - 1/_p)) - exp(exp(_p))
> e.subs(Omega[0],O2[0]).subs(Omega[1],O2[1])
-1/w + exp(exp(p)*exp(-exp(-p))/(1 - 1/p))
```

is really not what we want!

So we do it the hard way and keep track of all the things we potentially want to substitute by dummy variables. Consider the expression:

```
exp(x - exp(-x)) + exp(x) + x.
```

The mrv set is $\{\exp(x), \exp(-x), \exp(x - \exp(-x))\}$. We introduce corresponding dummy variables $d1, d2, d3$ and rewrite:

```
d3 + d1 + x.
```

This class first of all keeps track of the mapping $\text{expr} \rightarrow \text{variable}$, i.e. will at this stage be a dictionary:

```
{exp(x): d1, exp(-x): d2, exp(x - exp(-x)): d3}.
```

[It turns out to be more convenient this way round.] But sometimes expressions in the mrv set have other expressions from the mrv set as subexpressions, and we need to keep track of that as well. In this case, $d3$ is really $\exp(x - d2)$, so rewrites at this stage is:

```
{d3: exp(x-d2)}.
```

The function `rewrite` uses all this information to correctly rewrite our expression in terms of w . In this case w can be chosen to be $\exp(-x)$, i.e. $d2$. The correct rewriting then is:

```
exp(-w)/w + 1/w + x.
```

`copy()`

Create a shallow copy of `SubsSet`

`do_subs(e)`

Substitute the variables with expressions

`meets(s2)`

Tell whether or not self and $s2$ have non-empty intersection

`union(s2, exps=None)`

Compute the union of self and $s2$, adjusting exps

More Intuitive Series Expansion

This is achieved by creating a wrapper around `Basic.series()`. This allows for the use of `series(x*cos(x),x)`, which is possibly more intuitive than `(x*cos(x)).series(x)`.

Examples

```
>>> from sympy import Symbol, cos, series
>>> x = Symbol('x')
>>> series(cos(x),x)
1 - x**2/2 + x**4/24 + O(x**6)
```

Reference

`sympy.series.series.series(expr, x=None, x0=0, n=6, dir='+')`

Series expansion of `expr` around point $x = x_0$.

Parameters

expr : Expression

The expression whose series is to be expanded.

x : Symbol

It is the variable of the expression to be calculated.

x0 : Value

The value around which `x` is calculated. Can be any value from $-\infty$ to ∞ .

n : Value

The number of terms upto which the series is to be expanded.

dir : String, optional

The series-expansion can be bi-directional. If `dir="+"`, then $(x \rightarrow x_0 +)$. If `dir="-"`, then $(x \rightarrow x_0 -)$. For infinite `x0` (∞ or $-\infty$), the `dir` argument is determined from the direction of the infinity (i.e., `dir="-"` for ∞).

Returns

Expr

Series expansion of the expression about `x0`

Examples

```
>>> from sympy import series, tan, oo
>>> from sympy.abc import x
>>> f = tan(x)
>>> series(f, x, 2, 6, "+")
tan(2) + (1 + tan(2)**2)*(x - 2) + (x - 2)**2*(tan(2)**3 + tan(2)) +
(x - 2)**3*(1/3 + 4*tan(2)**2/3 + tan(2)**4) + (x - 2)**4*(tan(2)**5 +
5*tan(2)**3/3 + 2*tan(2)/3) + (x - 2)**5*(2/15 + 17*tan(2)**2/15 +
2*tan(2)**4 + tan(2)**6) + O((x - 2)**6, (x, 2))
```

```
>>> series(f, x, 2, 3, "-")
tan(2) + (2 - x)*(-tan(2)**2 - 1) + (2 - x)**2*(tan(2)**3 + tan(2))
+ O((x - 2)**3, (x, 2))
```

```
>>> series(f, x, 2, oo, "+")
Traceback (most recent call last):
...
TypeError: 'Infinity' object cannot be interpreted as an integer
```

See also:

[*sympy.core.expr.Expr.series* \(page 973\)](#)

See the docstring of `Expr.series()` for complete details of this wrapper.

Order Terms

This module also implements automatic keeping track of the order of your expansion.

Examples

```
>>> from sympy import Symbol, Order
>>> x = Symbol('x')
>>> Order(x) + x**2
O(x)
>>> Order(x) + 1
1 + O(x)
```

Reference

class `sympy.series.order.Order`(*expr*, **args*, ***kwargs*)

Represents the limiting behavior of some function.

Explanation

The order of a function characterizes the function based on the limiting behavior of the function as it goes to some limit. Only taking the limit point to be a number is currently supported. This is expressed in big O notation [R742].

The formal definition for the order of a function $g(x)$ about a point a is such that $g(x) = O(f(x))$ as $x \rightarrow a$ if and only if for any $\delta > 0$ there exists a $M > 0$ such that $|g(x)| \leq M|f(x)|$ for $|x - a| < \delta$. This is equivalent to $\lim_{x \rightarrow a} \sup |g(x)/f(x)| < \infty$.

Let's illustrate it on the following example by taking the expansion of $\sin(x)$ about 0:

$$\sin(x) = x - x^3/3! + O(x^5)$$

where in this case $O(x^5) = x^5/5! - x^7/7! + \dots$. By the definition of O , for any $\delta > 0$ there is an M such that:

$$|x^5/5! - x^7/7! + \dots| \leq M|x^5| \text{ for } |x| < \delta$$

or by the alternate definition:

$$\lim_{x \rightarrow 0} |(x^5/5! - x^7/7! + \dots)/x^5| < \infty$$

which surely is true, because

$$\lim_{x \rightarrow 0} |(x^5/5! - x^7/7! + \dots)/x^5| = 1/5!$$

As it is usually used, the order of a function can be intuitively thought of representing all terms of powers greater than the one specified. For example, $O(x^3)$ corresponds to any terms proportional to $x^3, x^4, \dots$ and any higher power. For a polynomial, this leaves terms proportional to x^2, x and constants.

Examples

```
>>> from sympy import 0, oo, cos, pi
>>> from sympy.abc import x, y
```

```
>>> O(x + x**2)
O(x)
>>> O(x + x**2, (x, 0))
O(x)
>>> O(x + x**2, (x, oo))
O(x**2, (x, oo))
```

```
>>> O(1 + x*y)
O(1, x, y)
>>> O(1 + x*y, (x, 0), (y, 0))
O(1, x, y)
>>> O(1 + x*y, (x, oo), (y, oo))
O(x*y, (x, oo), (y, oo))
```

```
>>> O(1) in O(1, x)
True
>>> O(1, x) in O(1)
False
>>> O(x) in O(1, x)
True
>>> O(x**2) in O(x)
True
```

```
>>> O(x)*x
O(x**2)
>>> O(x) - O(x)
O(x)
>>> O(cos(x))
O(1)
>>> O(cos(x), (x, pi/2))
O(x - pi/2, (x, pi/2))
```

Notes

In $O(f(x), x)$ the expression $f(x)$ is assumed to have a leading term. $O(f(x), x)$ is automatically transformed to $O(f(x).as\_leading\_term(x), x)$.

$O(expr*f(x), x)$ is $O(f(x), x)$

$O(expr, x)$ is $O(1)$

$O(0, x)$ is 0.

Multivariate O is also supported:

$O(f(x, y), x, y)$ is transformed to $O(f(x, y).as\_leading\_term(x, y).as\_leading\_term(y), x, y)$

In the multivariate case, it is assumed the limits w.r.t. the various symbols commute.

If no symbols are passed then all symbols in the expression are used and the limit point is assumed to be zero.

References

[R742]

contains(*expr*)

Return True if *expr* belongs to $\text{Order}(\text{self.expr}, * \text{self.variables})$. Return False if *self* belongs to *expr*. Return None if the inclusion relation cannot be determined (e.g. when *self* and *expr* have different symbols).

Series Acceleration

TODO

Reference

`sympy.series.acceleration.richardson(A, k, n, N)`

Calculate an approximation for $\lim_{k \rightarrow \infty} A(k)$ using Richardson extrapolation with the terms $A(n)$, $A(n+1)$, ..., $A(n+N+1)$. Choosing $N \sim 2^n$ often gives good results.

Examples

A simple example is to calculate $\exp(1)$ using the limit definition. This limit converges slowly; $n = 100$ only produces two accurate digits:

```
>>> from sympy.abc import n
>>> e = (1 + 1/n)**n
>>> print(round(e.subs(n, 100).evalf(), 10))
2.7048138294
```

Richardson extrapolation with 11 appropriately chosen terms gives a value that is accurate to the indicated precision:

```
>>> from sympy import E
>>> from sympy.series.acceleration import richardson
>>> print(round(richardson(e, n, 10, 20).evalf(), 10))
2.7182818285
>>> print(round(E.evalf(), 10))
2.7182818285
```

Another useful application is to speed up convergence of series. Computing 100 terms of the zeta(2) series $1/k^{**2}$ yields only two accurate digits:

```
>>> from sympy.abc import k, n
>>> from sympy import Sum
>>> A = Sum(k** -2, (k, 1, n))
>>> print(round(A.subs(n, 100).evalf(), 10))
1.6349839002
```

Richardson extrapolation performs much better:

```
>>> from sympy import pi
>>> print(round(richardson(A, n, 10, 20).evalf(), 10))
1.6449340668
>>> print(round(((pi**2)/6).evalf(), 10))      # Exact value
1.6449340668
```

`sympy.series.acceleration.shanks(A, k, n, m=1)`

Calculate an approximation for $\lim_{k \rightarrow \infty} A(k)$ using the n -term Shanks transformation $S(A)(n)$. With $m > 1$, calculate the m -fold recursive Shanks transformation $S(S(\dots S(A)\dots))(n)$.

The Shanks transformation is useful for summing Taylor series that converge slowly near a pole or singularity, e.g. for $\log(2)$:

```
>>> from sympy.abc import k, n
>>> from sympy import Sum, Integer
>>> from sympy.series.acceleration import shanks
>>> A = Sum(Integer(-1)**(k+1) / k, (k, 1, n))
>>> print(round(A.subs(n, 100).doit().evalf(), 10))
0.6881721793
>>> print(round(shanks(A, n, 25).evalf(), 10))
0.6931396564
>>> print(round(shanks(A, n, 25, 5).evalf(), 10))
0.6931471806
```

The correct value is 0.6931471805599453094172321215.

Residues

TODO

Reference

`sympy.series.residues.residue(expr, x, x0)`

Finds the residue of `expr` at the point $x=x_0$.

The residue is defined as the coefficient of $1/(x-x_0)$ in the power series expansion about $x=x_0$.

Examples

```
>>> from sympy import Symbol, residue, sin
>>> x = Symbol("x")
>>> residue(1/x, x, 0)
1
>>> residue(1/x**2, x, 0)
0
>>> residue(2/sin(x), x, 0)
2
```

This function is essential for the Residue Theorem [1].

References

[R743]

Sequences

A sequence is a finite or infinite lazily evaluated list.

`sympy.series.sequences.sequence(seq, limits=None)`

Returns appropriate sequence object.

Explanation

If `seq` is a SymPy sequence, returns [SeqPer](#) (page 635) object otherwise returns [SeqFormula](#) (page 634) object.

Examples

```
>>> from sympy import sequence
>>> from sympy.abc import n
>>> sequence(n**2, (n, 0, 5))
SeqFormula(n**2, (n, 0, 5))
>>> sequence((1, 2, 3), (n, 0, 5))
SeqPer((1, 2, 3), (n, 0, 5))
```

See also:

[sympy.series.sequences.SeqPer](#) (page 635), [sympy.series.sequences.SeqFormula](#) (page 634)

Sequences Base

class `sympy.series.sequences.SeqBase(*args)`

Base class for sequences

coeff(*pt*)

Returns the coefficient at point *pt*

coeff_mul(*other*)

Should be used when *other* is not a sequence. Should be defined to define custom behaviour.

Examples

```
>>> from sympy import SeqFormula
>>> from sympy.abc import n
>>> SeqFormula(n**2).coeff_mul(2)
SeqFormula(2*n**2, (n, 0, oo))
```

Notes

'*' defines multiplication of sequences with sequences only.

find_linear_recurrence(*n*, *d=None*, *gfvar=None*)

Finds the shortest linear recurrence that satisfies the first *n* terms of sequence of order $\leq n/2$ if possible. If *d* is specified, find shortest linear recurrence of order $\leq \min(d, n/2)$ if possible. Returns list of coefficients $[b(1), b(2), \dots]$ corresponding to the recurrence relation $x(n) = b(1)*x(n-1) + b(2)*x(n-2) + \dots$. Returns $[]$ if no recurrence is found. If *gfvar* is specified, also returns ordinary generating function as a function of *gfvar*.

Examples

```
>>> from sympy import sequence, sqrt, oo, lucas
>>> from sympy.abc import n, x, y
>>> sequence(n**2).find_linear_recurrence(10, 2)
[]
>>> sequence(n**2).find_linear_recurrence(10)
[3, -3, 1]
>>> sequence(2**n).find_linear_recurrence(10)
[2]
>>> sequence(23*n**4+91*n**2).find_linear_recurrence(10)
[5, -10, 10, -5, 1]
>>> sequence(sqrt(5)*((1 + sqrt(5))/2)**n - (-(1 + sqrt(5))/2)**(-
→n))/5).find_linear_recurrence(10)
[1, 1]
>>> sequence(x+y*(-2)**(-n), (n, 0, oo)).find_linear_recurrence(30)
[1/2, 1/2]
>>> sequence(3*5**n + 12).find_linear_recurrence(20,gfvar=x)
([6, -5], 3*(5 - 21*x)/((x - 1)*(5*x - 1)))
>>> sequence(lucas(n)).find_linear_recurrence(15,gfvar=x)
([1, 1], (x - 2)/(x**2 + x - 1))
```

property free_symbols

This method returns the symbols in the object, excluding those that take on a specific value (i.e. the dummy symbols).

Examples

```
>>> from sympy import SeqFormula
>>> from sympy.abc import n, m
>>> SeqFormula(m*n**2, (n, 0, 5)).free_symbols
{m}
```

property gen

Returns the generator for the sequence

property interval

The interval on which the sequence is defined

property length

Length of the sequence

property start

The starting point of the sequence. This point is included

property stop

The ending point of the sequence. This point is included

property variables

Returns a tuple of variables that are bounded

Elementary Sequences

class sympy.series.sequences.**SeqFormula**(*formula*, *limits=None*)

Represents sequence based on a formula.

Elements are generated using a formula.

Examples

```
>>> from sympy import SeqFormula, oo, Symbol
>>> n = Symbol('n')
>>> s = SeqFormula(n**2, (n, 0, 5))
>>> s.formula
n**2
```

For value at a particular point

```
>>> s.coeff(3)
9
```

supports slicing

```
>>> s[:]
[0, 1, 4, 9, 16, 25]
```

iterable

```
>>> list(s)
[0, 1, 4, 9, 16, 25]
```

sequence starts from negative infinity

```
>>> SeqFormula(n**2, (-oo, 0))[0:6]
[0, 1, 4, 9, 16, 25]
```

See also:

[`sympy.series.sequences.SeqPer`](#) (page 635)

coeff_mul(*coeff*)

See docstring of SeqBase.coeff_mul

class `sympy.series.sequences.SeqPer`(*periodical*, *limits=None*)

Represents a periodic sequence.

The elements are repeated after a given period.

Examples

```
>>> from sympy import SeqPer, oo
>>> from sympy.abc import k
```

```
>>> s = SeqPer((1, 2, 3), (0, 5))
>>> s.periodical
(1, 2, 3)
>>> s.period
3
```

For value at a particular point

```
>>> s.coeff(3)
1
```

supports slicing

```
>>> s[:]
[1, 2, 3, 1, 2, 3]
```

iterable

```
>>> list(s)
[1, 2, 3, 1, 2, 3]
```

sequence starts from negative infinity

```
>>> SeqPer((1, 2, 3), (-oo, 0))[0:6]
[1, 2, 3, 1, 2, 3]
```

Periodic formulas

```
>>> SeqPer((k, k**2, k**3), (k, 0, oo))[0:6]
[0, 1, 8, 3, 16, 125]
```

See also:

[*sympy.series.sequences.SeqFormula*](#) (page 634)

coeff_mul(*coeff*)

See docstring of SeqBase.coeff_mul

Singleton Sequences

class sympy.series.sequences.**EmptySequence**

Represents an empty sequence.

The empty sequence is also available as a singleton as `S.EmptySequence`.

Examples

```
>>> from sympy import EmptySequence, SeqPer
>>> from sympy.abc import x
>>> EmptySequence
EmptySequence
>>> SeqPer((1, 2), (x, 0, 10)) + EmptySequence
SeqPer((1, 2), (x, 0, 10))
>>> SeqPer((1, 2)) * EmptySequence
EmptySequence
>>> EmptySequence.coeff_mul(-1)
EmptySequence
```

coeff_mul(*coeff*)

See docstring of SeqBase.coeff_mul

Compound Sequences

class sympy.series.sequences.**SeqAdd**(*args, **kwargs)

Represents term-wise addition of sequences.

Rules:

- The interval on which sequence is defined is the intersection of respective intervals of sequences.
- Anything + [*EmptySequence*](#) (page 636) remains unchanged.
- Other rules are defined in `_add` methods of sequence classes.

Examples

```
>>> from sympy import EmptySequence, oo, SeqAdd, SeqPer, SeqFormula
>>> from sympy.abc import n
>>> SeqAdd(SeqPer((1, 2), (n, 0, oo)), EmptySequence)
SeqPer((1, 2), (n, 0, oo))
>>> SeqAdd(SeqPer((1, 2), (n, 0, 5)), SeqPer((1, 2), (n, 6, 10)))
EmptySequence
>>> SeqAdd(SeqPer((1, 2), (n, 0, oo)), SeqFormula(n**2, (n, 0, oo)))
SeqAdd(SeqFormula(n**2, (n, 0, oo)), SeqPer((1, 2), (n, 0, oo)))
>>> SeqAdd(SeqFormula(n**3), SeqFormula(n**2))
SeqFormula(n**3 + n**2, (n, 0, oo))
```

See also:

[sympy.series.sequences.SeqMul](#) (page 637)

static reduce(args)

Simplify [SeqAdd](#) (page 636) using known rules.

Iterates through all pairs and ask the constituent sequences if they can simplify themselves with any other constituent.

Notes

adapted from `Union.reduce`

class `sympy.series.sequences.SeqMul(*args, **kwargs)`

Represents term-wise multiplication of sequences.

Explanation

Handles multiplication of sequences only. For multiplication with other objects see [SeqBase.coeff_mul\(\)](#) (page 632).

Rules:

- The interval on which sequence is defined is the intersection of respective intervals of sequences.
- Anything * [EmptySequence](#) (page 636) returns [EmptySequence](#) (page 636).
- Other rules are defined in `_mul` methods of sequence classes.

Examples

```
>>> from sympy import EmptySequence, oo, SeqMul, SeqPer, SeqFormula
>>> from sympy.abc import n
>>> SeqMul(SeqPer((1, 2), (n, 0, oo)), EmptySequence)
EmptySequence
>>> SeqMul(SeqPer((1, 2), (n, 0, 5)), SeqPer((1, 2), (n, 6, 10)))
EmptySequence
>>> SeqMul(SeqPer((1, 2), (n, 0, oo)), SeqFormula(n**2))
```

(continues on next page)

(continued from previous page)

```
SeqMul(SeqFormula(n**2, (n, 0, oo)), SeqPer((1, 2), (n, 0, oo)))
>>> SeqMul(SeqFormula(n**3), SeqFormula(n**2))
SeqFormula(n**5, (n, 0, oo))
```

See also:

[*sympy.series.sequences.SeqAdd*](#) (page 636)

static reduce(args)

Simplify a [*SeqMul*](#) (page 637) using known rules.

Explanation

Iterates through all pairs and ask the constituent sequences if they can simplify themselves with any other constituent.

Notes

adapted from `Union.reduce`

Recursive Sequences

class `sympy.series.sequences.RecursiveSeq`(*recurrence, yn, n, initial=None, start=0*)

A finite degree recursive sequence.

Parameters

recurrence : SymPy expression defining recurrence

This is *not* an equality, only the expression that the n th term is equal to. For example, if $a(n) = f(a(n - 1), \dots, a(n - d))$, then the expression should be $f(a(n - 1), \dots, a(n - d))$.

yn : applied undefined function

Represents the n th term of the sequence as e.g. $y(n)$ where y is an undefined function and n is the sequence index.

n : symbolic argument

The name of the variable that the recurrence is in, e.g., n if the recurrence function is $y(n)$.

initial : iterable with length equal to the degree of the recurrence

The initial values of the recurrence.

start : start value of sequence (inclusive)

Explanation

That is, a sequence $a(n)$ that depends on a fixed, finite number of its previous values. The general form is

$$a(n) = f(a(n-1), a(n-2), \dots, a(n-d))$$

for some fixed, positive integer d , where f is some function defined by a SymPy expression.

Examples

```
>>> from sympy import Function, symbols
>>> from sympy.series.sequences import RecursiveSeq
>>> y = Function("y")
>>> n = symbols("n")
>>> fib = RecursiveSeq(y(n - 1) + y(n - 2), y(n), n, [0, 1])
```

```
>>> fib.coeff(3) # Value at a particular point
2
```

```
>>> fib[:6] # supports slicing
[0, 1, 1, 2, 3, 5]
```

```
>>> fib.recurrence # inspect recurrence
Eq(y(n), y(n - 2) + y(n - 1))
```

```
>>> fib.degree # automatically determine degree
2
```

```
>>> for x in zip(range(10), fib): # supports iteration
...     print(x)
(0, 0)
(1, 1)
(2, 1)
(3, 2)
(4, 3)
(5, 5)
(6, 8)
(7, 13)
(8, 21)
(9, 34)
```

See also:

[`sympy.series.sequences.SeqFormula`](#) (page 634)

property initial

The initial values of the sequence

property interval

Interval on which sequence is defined.

property n

Sequence index symbol

property recurrence

Equation defining recurrence.

property start

The starting point of the sequence. This point is included

property stop

The ending point of the sequence. (oo)

property y

Undefined function for the nth term of the sequence

property yn

Applied function representing the nth term

Fourier Series

Provides methods to compute Fourier series.

class `sympy.series.fourier.FourierSeries(*args)`

Represents Fourier sine/cosine series.

Explanation

This class only represents a fourier series. No computation is performed.

For how to compute Fourier series, see the [*fourier_series\(\)*](#) (page 643) docstring.

See also:

[*sympy.series.fourier.fourier_series*](#) (page 643)

scale(s)

Scale the function by a term independent of x.

Explanation

$f(x) \rightarrow s * f(x)$

This is fast, if Fourier series of $f(x)$ is already computed.

Examples

```
>>> from sympy import fourier_series, pi
>>> from sympy.abc import x
>>> s = fourier_series(x**2, (x, -pi, pi))
>>> s.scale(2).truncate()
-8*cos(x) + 2*cos(2*x) + 2*pi**2/3
```

scalex(s)

Scale x by a term independent of x.

Explanation

$f(x) \rightarrow f(s*x)$

This is fast, if Fourier series of $f(x)$ is already computed.

Examples

```
>>> from sympy import fourier_series, pi
>>> from sympy.abc import x
>>> s = fourier_series(x**2, (x, -pi, pi))
>>> s.scalex(2).truncate()
-4*cos(2*x) + cos(4*x) + pi**2/3
```

shift(s)

Shift the function by a term independent of x .

Explanation

$f(x) \rightarrow f(x) + s$

This is fast, if Fourier series of $f(x)$ is already computed.

Examples

```
>>> from sympy import fourier_series, pi
>>> from sympy.abc import x
>>> s = fourier_series(x**2, (x, -pi, pi))
>>> s.shift(1).truncate()
-4*cos(x) + cos(2*x) + 1 + pi**2/3
```

shiftx(s)

Shift x by a term independent of x .

Explanation

$f(x) \rightarrow f(x + s)$

This is fast, if Fourier series of $f(x)$ is already computed.

Examples

```
>>> from sympy import fourier_series, pi
>>> from sympy.abc import x
>>> s = fourier_series(x**2, (x, -pi, pi))
>>> s.shiftx(1).truncate()
-4*cos(x + 1) + cos(2*x + 2) + pi**2/3
```

`sigma_approximation(n=3)`

Return σ -approximation of Fourier series with respect to order n .

Parameters

n : int

Highest order of the terms taken into account in approximation.

Returns

Expr :

Sigma approximation of function expanded into Fourier series.

Explanation

Sigma approximation adjusts a Fourier summation to eliminate the Gibbs phenomenon which would otherwise occur at discontinuities. A sigma-approximated summation for a Fourier series of a T -periodical function can be written as

$$s(\theta) = \frac{1}{2}a_0 + \sum_{k=1}^{m-1} \text{sinc}\left(\frac{k}{m}\right) \cdot \left[a_k \cos\left(\frac{2\pi k}{T}\theta\right) + b_k \sin\left(\frac{2\pi k}{T}\theta\right) \right],$$

where $a_0, a_k, b_k, k = 1, \dots, m-1$ are standard Fourier series coefficients and $\text{sinc}\left(\frac{k}{m}\right)$ is a Lanczos σ factor (expressed in terms of normalized sinc function).

Examples

```
>>> from sympy import fourier_series, pi
>>> from sympy.abc import x
>>> s = fourier_series(x, (x, -pi, pi))
>>> s.sigma_approximation(4)
2*sin(x)*sinc(pi/4) - 2*sin(2*x)/pi + 2*sin(3*x)*sinc(3*pi/4)/3
```

Notes

The behaviour of `sigma_approximation()` (page 642) is different from `truncate()` (page 643) - it takes all nonzero terms of degree smaller than n , rather than first n nonzero ones.

See also:

`sympy.series.fourier.FourierSeries.truncate` (page 643)

References

[R737], [R738]

truncate(*n*=3)

Return the first *n* nonzero terms of the series.

If *n* is None return an iterator.

Parameters

n : int or None

Amount of non-zero terms in approximation or None.

Returns

Expr or iterator :

Approximation of function expanded into Fourier series.

Examples

```
>>> from sympy import fourier_series, pi
>>> from sympy.abc import x
>>> s = fourier_series(x, (x, -pi, pi))
>>> s.truncate(4)
2*sin(x) - sin(2*x) + 2*sin(3*x)/3 - sin(4*x)/2
```

See also:

[*sympy.series.fourier.FourierSeries.sigma_approximation*](#) (page 642)

`sympy.series.fourier.fourier_series(f, limits=None, finite=True)`

Computes the Fourier trigonometric series expansion.

Parameters

limits : (sym, start, end), optional

sym denotes the symbol the series is computed with respect to.

start and *end* denotes the start and the end of the interval where the fourier series converges to the given function.

Default range is specified as $-\pi$ and π .

Returns

FourierSeries

A symbolic object representing the Fourier trigonometric series.

Explanation

Fourier trigonometric series of $f(x)$ over the interval (a, b) is defined as:

$$\frac{a_0}{2} + \sum_{n=1}^{\infty} \left(a_n \cos\left(\frac{2n\pi x}{L}\right) + b_n \sin\left(\frac{2n\pi x}{L}\right) \right)$$

where the coefficients are:

$$\begin{aligned} L &= b - a \\ a_0 &= \frac{2}{L} \int_a^b f(x) dx \\ a_n &= \frac{2}{L} \int_a^b f(x) \cos\left(\frac{2n\pi x}{L}\right) dx \\ b_n &= \frac{2}{L} \int_a^b f(x) \sin\left(\frac{2n\pi x}{L}\right) dx \end{aligned}$$

The condition whether the function $f(x)$ given should be periodic or not is more than necessary, because it is sufficient to consider the series to be converging to $f(x)$ only in the given interval, not throughout the whole real line.

This also brings a lot of ease for the computation because you do not have to make $f(x)$ artificially periodic by wrapping it with piecewise, modulo operations, but you can shape the function to look like the desired periodic function only in the interval (a, b) , and the computed series will automatically become the series of the periodic version of $f(x)$.

This property is illustrated in the examples section below.

Examples

Computing the Fourier series of $f(x) = x^2$:

```
>>> from sympy import fourier_series, pi
>>> from sympy.abc import x
>>> f = x**2
>>> s = fourier_series(f, (x, -pi, pi))
>>> s1 = s.truncate(n=3)
>>> s1
-4*cos(x) + cos(2*x) + pi**2/3
```

Shifting of the Fourier series:

```
>>> s.shift(1).truncate()
-4*cos(x) + cos(2*x) + 1 + pi**2/3
>>> s.shiftx(1).truncate()
-4*cos(x + 1) + cos(2*x + 2) + pi**2/3
```

Scaling of the Fourier series:

```
>>> s.scale(2).truncate()
-8*cos(x) + 2*cos(2*x) + 2*pi**2/3
>>> s.scalex(2).truncate()
-4*cos(2*x) + cos(4*x) + pi**2/3
```

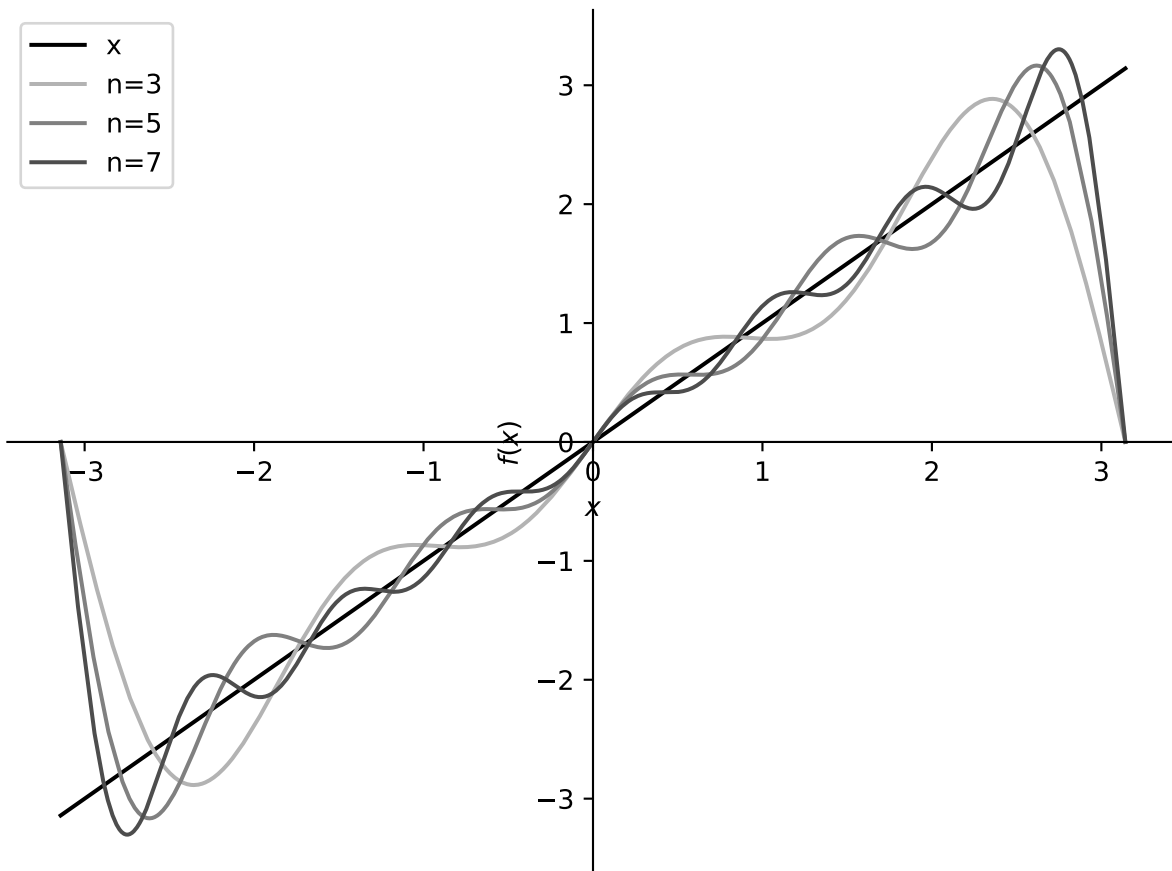
Computing the Fourier series of $f(x) = x$:

This illustrates how truncating to the higher order gives better convergence.

```
>>> from sympy import fourier_series, pi, plot
>>> from sympy.abc import x
>>> f = x
>>> s = fourier_series(f, (x, -pi, pi))
>>> s1 = s.truncate(n = 3)
>>> s2 = s.truncate(n = 5)
>>> s3 = s.truncate(n = 7)
>>> p = plot(f, s1, s2, s3, (x, -pi, pi), show=False, legend=True)
```

```
>>> p[0].line_color = (0, 0, 0)
>>> p[0].label = 'x'
>>> p[1].line_color = (0.7, 0.7, 0.7)
>>> p[1].label = 'n=3'
>>> p[2].line_color = (0.5, 0.5, 0.5)
>>> p[2].label = 'n=5'
>>> p[3].line_color = (0.3, 0.3, 0.3)
>>> p[3].label = 'n=7'
```

```
>>> p.show()
```

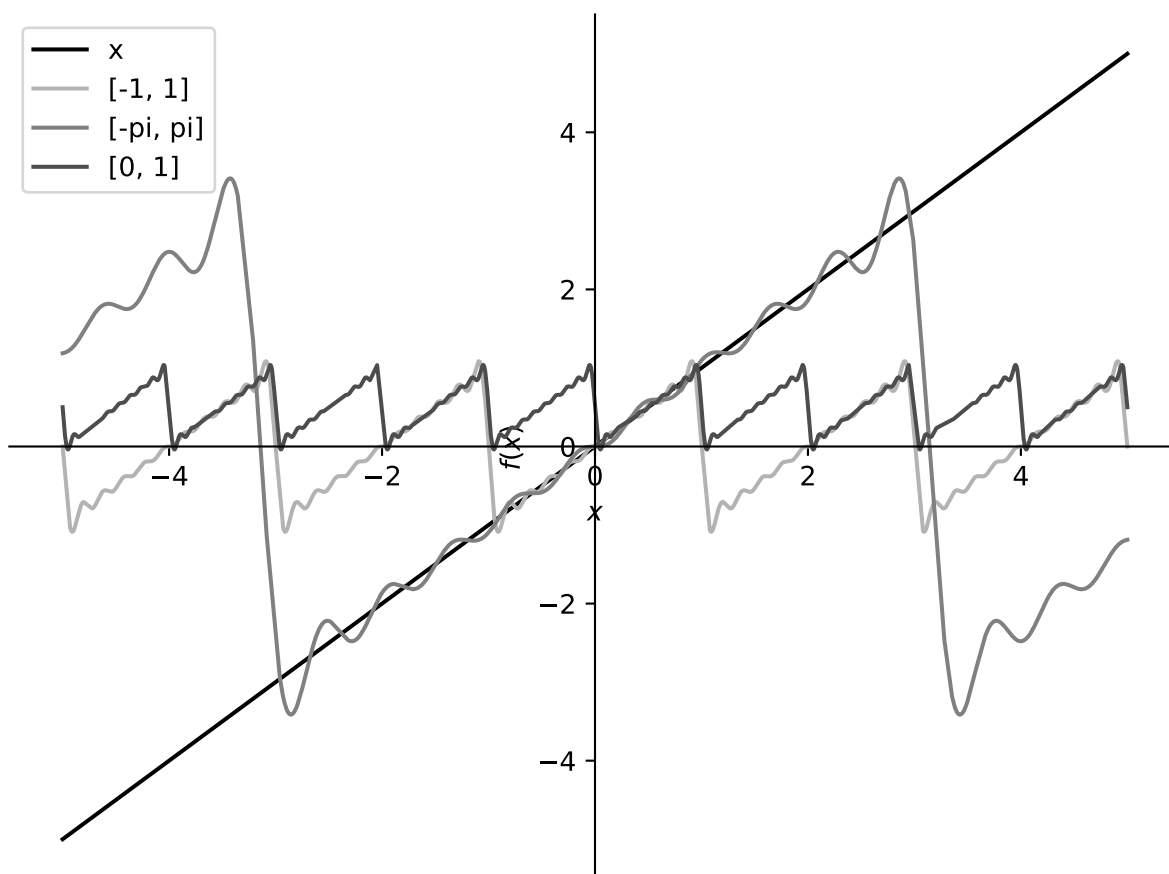


This illustrates how the series converges to different sawtooth waves if the different ranges are specified.

```
>>> s1 = fourier_series(x, (x, -1, 1)).truncate(10)
>>> s2 = fourier_series(x, (x, -pi, pi)).truncate(10)
>>> s3 = fourier_series(x, (x, 0, 1)).truncate(10)
>>> p = plot(x, s1, s2, s3, (x, -5, 5), show=False, legend=True)
```

```
>>> p[0].line_color = (0, 0, 0)
>>> p[0].label = 'x'
>>> p[1].line_color = (0.7, 0.7, 0.7)
>>> p[1].label = '[-1, 1]'
>>> p[2].line_color = (0.5, 0.5, 0.5)
>>> p[2].label = '[-pi, pi]'
>>> p[3].line_color = (0.3, 0.3, 0.3)
>>> p[3].label = '[0, 1]'
```

```
>>> p.show()
```



Notes

Computing Fourier series can be slow due to the integration required in computing a_n , b_n .

It is faster to compute Fourier series of a function by using shifting and scaling on an already computed Fourier series rather than computing again.

e.g. If the Fourier series of x^2 is known the Fourier series of $x^2 - 1$ can be found by shifting by -1 .

See also:

[`sympy.series.fourier.FourierSeries`](#) (page 640)

References

[R739]

Formal Power Series

Methods for computing and manipulating Formal Power Series.

class `sympy.series.formal.FormalPowerSeries(*args)`

Represents Formal Power Series of a function.

Explanation

No computation is performed. This class should only to be used to represent a series. No checks are performed.

For computing a series use [`fps\(\)`](#) (page 650).

See also:

[`sympy.series.formal.fps`](#) (page 650)

coeff_bell(n)

`self.coeff_bell(n)` returns a sequence of Bell polynomials of the second kind. Note that n should be a integer.

The second kind of Bell polynomials (are sometimes called “partial” Bell polynomials or incomplete Bell polynomials) are defined as

$$B_{n,k}(x_1, x_2, \dots, x_{n-k+1}) = \sum_{\substack{j_1+j_2+j_3+\dots=k \\ j_1+2j_2+3j_3+\dots=n}} \frac{n!}{j_1!j_2!\dots j_{n-k+1}!} \left(\frac{x_1}{1!}\right)^{j_1} \left(\frac{x_2}{2!}\right)^{j_2} \dots \left(\frac{x_{n-k+1}}{(n-k+1)!}\right)^{j_{n-k+1}}.$$

- `bell(n, k, (x1, x2, ...))` gives Bell polynomials of the second kind, $B_{n,k}(x_1, x_2, \dots, x_{n-k+1})$.

See also:

[`sympy.functions.combinatorial.numbers.bell`](#) (page 427)

compose(*other*, *x=None*, *n=6*)

Returns the truncated terms of the formal power series of the composed function, up to specified *n*.

Parameters

n : Number, optional

Specifies the order of the term up to which the polynomial should be truncated.

Explanation

If *f* and *g* are two formal power series of two different functions, then the coefficient sequence *a_k* of the composed formal power series *f_p* will be as follows.

$$\sum_{k=0}^n b_k B_{n,k}(x_1, x_2, \dots, x_{n-k+1})$$

Examples

```
>>> from sympy import fps, sin, exp
>>> from sympy.abc import x
>>> f1 = fps(exp(x))
>>> f2 = fps(sin(x))
```

```
>>> f1.compose(f2, x).truncate()
1 + x + x**2/2 - x**4/8 - x**5/15 + O(x**6)
```

```
>>> f1.compose(f2, x).truncate(8)
1 + x + x**2/2 - x**4/8 - x**5/15 - x**6/240 + x**7/90 + O(x**8)
```

See also:

[*sympy.functions.combinatorial.numbers.bell*](#) (page 427), [*sympy.series.formal.FormalPowerSeriesCompose*](#) (page 653)

References

[R731]

property infinite

Returns an infinite representation of the series

integrate(*x=None*, ***kwargs*)

Integrate Formal Power Series.

Examples

```
>>> from sympy import fps, sin, integrate
>>> from sympy.abc import x
>>> f = fps(sin(x))
>>> f.integrate(x).truncate()
-1 + x**2/2 - x**4/24 + 0(x**6)
>>> integrate(f, (x, 0, 1))
1 - cos(1)
```

inverse(*x=None, n=6*)

Returns the truncated terms of the inverse of the formal power series, up to specified *n*.

Parameters

n : Number, optional

Specifies the order of the term up to which the polynomial should be truncated.

Explanation

If *f* and *g* are two formal power series of two different functions, then the coefficient sequence *a_k* of the composed formal power series *fp* will be as follows.

$$\sum_{k=0}^n (-1)^k x_0^{-k-1} B_{n,k}(x_1, x_2, \dots, x_{n-k+1})$$

Examples

```
>>> from sympy import fps, exp, cos
>>> from sympy.abc import x
>>> f1 = fps(exp(x))
>>> f2 = fps(cos(x))
```

```
>>> f1.inverse(x).truncate()
1 - x + x**2/2 - x**3/6 + x**4/24 - x**5/120 + 0(x**6)
```

```
>>> f2.inverse(x).truncate(8)
1 + x**2/2 + 5*x**4/24 + 61*x**6/720 + 0(x**8)
```

See also:

[sympy.functions.combinatorial.numbers.bell](#) (page 427), [sympy.series.formal.FormalPowerSeriesInverse](#) (page 653)

References

[R732]

polynomial(*n*=6)

Truncated series as polynomial.

Explanation

Returns series expansion of *f* upto order $O(x^n)$ as a polynomial(without 0 term).

product(*other*, *x*=None, *n*=6)

Multiplies two Formal Power Series, using discrete convolution and return the truncated terms upto specified order.

Parameters

n : Number, optional

Specifies the order of the term up to which the polynomial should be truncated.

Examples

```
>>> from sympy import fps, sin, exp
>>> from sympy.abc import x
>>> f1 = fps(sin(x))
>>> f2 = fps(exp(x))
```

```
>>> f1.product(f2, x).truncate(4)
x + x**2 + x**3/3 + O(x**4)
```

See also:

[sympy.discrete.convolutions](#) (page 1089), [sympy.series.formal.FormalPowerSeriesProduct](#) (page 653)

truncate(*n*=6)

Truncated series.

Explanation

Returns truncated series expansion of *f* upto order $O(x^n)$.

If *n* is None, returns an infinite iterator.

sympy.series.formal.fps(*f*, *x*=None, *x0*=0, *dir*=1, *hyper*=True, *order*=4, *rational*=True, *full*=False)

Generates Formal Power Series of *f*.

Parameters

x : Symbol, optional

If *x* is None and *f* is univariate, the univariate symbols will be supplied, otherwise an error will be raised.

x0 : number, optional

Point to perform series expansion about. Default is 0.

dir : {1, -1, '+', '-'}, optional

If dir is 1 or '+' the series is calculated from the right and for -1 or '-' the series is calculated from the left. For smooth functions this flag will not alter the results. Default is 1.

hyper : {True, False}, optional

Set hyper to False to skip the hypergeometric algorithm. By default it is set to False.

order : int, optional

Order of the derivative of f, Default is 4.

rational : {True, False}, optional

Set rational to False to skip rational algorithm. By default it is set to True.

full : {True, False}, optional

Set full to True to increase the range of rational algorithm. See [rational_algorithm\(\)](#) (page 654) for details. By default it is set to False.

Explanation

Returns the formal series expansion of f around $x = x_0$ with respect to x in the form of a FormalPowerSeries object.

Formal Power Series is represented using an explicit formula computed using different algorithms.

See [compute_fps\(\)](#) (page 652) for the more details regarding the computation of formula.

Examples

```
>>> from sympy import fps, ln, atan, sin
>>> from sympy.abc import x, n
```

Rational Functions

```
>>> fps(ln(1 + x)).truncate()
x - x**2/2 + x**3/3 - x**4/4 + x**5/5 + O(x**6)
```

```
>>> fps(atan(x), full=True).truncate()
x - x**3/3 + x**5/5 + O(x**6)
```

Symbolic Functions

```
>>> fps(x**n*sin(x**2), x).truncate(8)
-x**(n + 6)/6 + x**(n + 2) + O(x**(n + 8))
```

See also:

[`sympy.series.formal.FormalPowerSeries`](#) (page 647), [`sympy.series.formal.compute\_fps`](#) (page 652)

`sympy.series.formal.compute_fps(f, x, x0=0, dir=1, hyper=True, order=4, rational=True, full=False)`

Computes the formula for Formal Power Series of a function.

Parameters

x : Symbol

x0 : number, optional

Point to perform series expansion about. Default is 0.

dir : {1, -1, '+', '-'}, optional

If dir is 1 or '+' the series is calculated from the right and for -1 or '-' the series is calculated from the left. For smooth functions this flag will not alter the results. Default is 1.

hyper : {True, False}, optional

Set hyper to False to skip the hypergeometric algorithm. By default it is set to False.

order : int, optional

Order of the derivative of f, Default is 4.

rational : {True, False}, optional

Set rational to False to skip rational algorithm. By default it is set to True.

full : {True, False}, optional

Set full to True to increase the range of rational algorithm. See [`rational\_algorithm\(\)`](#) (page 654) for details. By default it is set to False.

Returns

ak : sequence

Sequence of coefficients.

xk : sequence

Sequence of powers of x.

ind : Expr

Independent terms.

mul : Pow

Common terms.

Explanation

Tries to compute the formula by applying the following techniques (in order):

- `rational_algorithm`
- Hypergeometric algorithm

See also:

[`sympy.series.formal.rational\_algorithm`](#) (page 654), [`sympy.series.formal.hyper\_algorithm`](#) (page 658)

class `sympy.series.formal.FormalPowerSeriesCompose(*args)`

Represents the composed formal power series of two functions.

Explanation

No computation is performed. Terms are calculated using a term by term logic, instead of a point by point logic.

There are two differences between a [`FormalPowerSeries`](#) (page 647) object and a [`FormalPowerSeriesCompose`](#) (page 653) object. The first argument contains the outer function and the inner function involved in the composition. Also, the coefficient sequence contains the generic sequence which is to be multiplied by a custom `bell_seq` finite sequence. The finite terms will then be added up to get the final terms.

See also:

[`sympy.series.formal.FormalPowerSeries`](#) (page 647), [`sympy.series.formal.FiniteFormalPowerSeries`](#) (page 654)

property function

Function for the composed formal power series.

class `sympy.series.formal.FormalPowerSeriesInverse(*args)`

Represents the Inverse of a formal power series.

Explanation

No computation is performed. Terms are calculated using a term by term logic, instead of a point by point logic.

There is a single difference between a [`FormalPowerSeries`](#) (page 647) object and a [`FormalPowerSeriesInverse`](#) (page 653) object. The coefficient sequence contains the generic sequence which is to be multiplied by a custom `bell_seq` finite sequence. The finite terms will then be added up to get the final terms.

See also:

[`sympy.series.formal.FormalPowerSeries`](#) (page 647), [`sympy.series.formal.FiniteFormalPowerSeries`](#) (page 654)

property function

Function for the inverse of a formal power series.

class `sympy.series.formal.FormalPowerSeriesProduct(*args)`

Represents the product of two formal power series of two functions.

Explanation

No computation is performed. Terms are calculated using a term by term logic, instead of a point by point logic.

There are two differences between a *FormalPowerSeries* (page 647) object and a *FormalPowerSeriesProduct* (page 653) object. The first argument contains the two functions involved in the product. Also, the coefficient sequence contains both the coefficient sequence of the formal power series of the involved functions.

See also:

sympy.series.formal.FormalPowerSeries (page 647), *sympy.series.formal.FiniteFormalPowerSeries* (page 654)

property function

Function of the product of two formal power series.

class `sympy.series.formal.FiniteFormalPowerSeries(*args)`

Base Class for Product, Compose and Inverse classes

Rational Algorithm

`sympy.series.formal.rational_independent(terms, x)`

Returns a list of all the rationally independent terms.

Examples

```
>>> from sympy import sin, cos
>>> from sympy.series.formal import rational_independent
>>> from sympy.abc import x
```

```
>>> rational_independent([cos(x), sin(x)], x)
[cos(x), sin(x)]
>>> rational_independent([x**2, sin(x), x*sin(x), x**3], x)
[x**3 + x**2, x*sin(x) + sin(x)]
```

`sympy.series.formal.rational_algorithm(f, x, k, order=4, full=False)`

Rational algorithm for computing formula of coefficients of Formal Power Series of a function.

Parameters

x : Symbol

order : int, optional

Order of the derivative of f, Default is 4.

full : bool

Returns

formula : Expr

ind : Expr

Independent terms.

order : int**full** : bool

Explanation

Applicable when $f(x)$ or some derivative of $f(x)$ is a rational function in x .

[`rational\_algorithm\(\)`](#) (page 654) uses [`apart\(\)`](#) (page 2443) function for partial fraction decomposition. [`apart\(\)`](#) (page 2443) by default uses 'undetermined coefficients method'. By setting `full=True`, 'Bronstein's algorithm' can be used instead.

Looks for derivative of a function up to 4'th order (by default). This can be overridden using `order` option.

Examples

```
>>> from sympy import log, atan
>>> from sympy.series.formal import rational_algorithm as ra
>>> from sympy.abc import x, k
```

```
>>> ra(1 / (1 - x), x, k)
(1, 0, 0)
>>> ra(log(1 + x), x, k)
(-1/((-1)**k*k), 0, 1)
```

```
>>> ra(atan(x), x, k, full=True)
((-I/(2*(-I)**k) + I/(2*I**k))/k, 0, 1)
```

Notes

By setting `full=True`, range of admissible functions to be solved using `rational_algorithm` can be increased. This option should be used carefully as it can significantly slow down the computation as `doit` is performed on the [`RootSum`](#) (page 2436) object returned by the [`apart\(\)`](#) (page 2443) function. Use `full=False` whenever possible.

See also:

[`sympy.polys.partfrac.apart`](#) (page 2443)

References

[R733], [R734]

Hypergeometric Algorithm

`sympy.series.formal.simpleDE(f, x, g, order=4)`

Generates simple DE.

Explanation

DE is of the form

$$f^k(x) + \sum_{j=0}^{k-1} A_j f^j(x) = 0$$

where A_j should be rational function in x .

Generates DE's upto order 4 (default). DE's can also have free parameters.

By increasing order, higher order DE's can be found.

Yields a tuple of (DE, order).

`sympy.series.formal.exp_re(DE, r, k)`

Converts a DE with constant coefficients (exlike) into a RE.

Explanation

Performs the substitution:

$$f^j(x) \rightarrow r(k+j)$$

Normalises the terms so that lowest order of a term is always $r(k)$.

Examples

```
>>> from sympy import Function, Derivative
>>> from sympy.series.formal import exp_re
>>> from sympy.abc import x, k
>>> f, r = Function('f'), Function('r')
```

```
>>> exp_re(-f(x) + Derivative(f(x)), r, k)
-r(k) + r(k + 1)
>>> exp_re(Derivative(f(x), x) + Derivative(f(x), (x, 2)), r, k)
r(k) + r(k + 1)
```

See also:

[`sympy.series.formal.hyper\_re`](#) (page 656)

`sympy.series.formal.hyper_re(DE, r, k)`

Converts a DE into a RE.

Explanation

Performs the substitution:

$$x^l f^j(x) \rightarrow (k+1-l)_j \cdot a_{k+j-l}$$

Normalises the terms so that lowest order of a term is always $r(k)$.

Examples

```
>>> from sympy import Function, Derivative
>>> from sympy.series.formal import hyper_re
>>> from sympy.abc import x, k
>>> f, r = Function('f'), Function('r')
```

```
>>> hyper_re(-f(x) + Derivative(f(x)), r, k)
(k + 1)*r(k + 1) - r(k)
>>> hyper_re(-x*f(x) + Derivative(f(x), (x, 2)), r, k)
(k + 2)*(k + 3)*r(k + 3) - r(k)
```

See also:

[*sympy.series.formal.exp_re*](#) (page 656)

`sympy.series.formal.rsolve_hypergeometric(f, x, P, Q, k, m)`

Solves RE of hypergeometric type.

Returns

formula : Expr

ind : Expr

Independent terms.

order : int

Explanation

Attempts to solve RE of the form

$$Q(k) \cdot a(k+m) - P(k) \cdot a(k)$$

Transformations that preserve Hypergeometric type:

- $x^{**n}f(x)$: $b(k+m) = R(k-n)*b(k)$
- $f(A*x)$: $b(k+m) = A^{**m}*R(k)*b(k)$
- $f(x^{**n})$: $b(k+n*m) = R(k/n)*b(k)$
- $f(x^{**(1/m)})$: $b(k+1) = R(k*m)*b(k)$
- $f'(x)$: $b(k+m) = ((k+m+1)/(k+1))*R(k+1)*b(k)$

Some of these transformations have been used to solve the RE.

Examples

```
>>> from sympy import exp, ln, S
>>> from sympy.series.formal import rsolve_hypergeometric as rh
>>> from sympy.abc import x, k
```

```
>>> rh(exp(x), x, -S.One, (k + 1), k, 1)
(Piecewise((1/factorial(k), Eq(Mod(k, 1), 0)), (0, True)), 1, 1)
```

```
>>> rh(ln(1 + x), x, k**2, k*(k + 1), k, 1)
(Piecewise((-1)**(k - 1)*factorial(k - 1)/RisingFactorial(2, k - 1),
Eq(Mod(k, 1), 0)), (0, True)), x, 2)
```

References

[R735], [R736]

`sympy.series.formal.solve_de(f, x, DE, order, g, k)`

Solves the DE.

Returns

formula : Expr

ind : Expr

Independent terms.

order : int

Explanation

Tries to solve DE by either converting into a RE containing two terms or converting into a DE having constant coefficients.

Examples

```
>>> from sympy import Derivative as D, Function
>>> from sympy import exp, ln
>>> from sympy.series.formal import solve_de
>>> from sympy.abc import x, k
>>> f = Function('f')
```

```
>>> solve_de(exp(x), x, D(f(x), x) - f(x), 1, f, k)
(Piecewise((1/factorial(k), Eq(Mod(k, 1), 0)), (0, True)), 1, 1)
```

```
>>> solve_de(ln(1 + x), x, (x + 1)*D(f(x), x, 2) + D(f(x)), 2, f, k)
(Piecewise((-1)**(k - 1)*factorial(k - 1)/RisingFactorial(2, k - 1),
Eq(Mod(k, 1), 0)), (0, True)), x, 2)
```

`sympy.series.formal.hyper_algorithm(f, x, k, order=4)`

Hypergeometric algorithm for computing Formal Power Series.

Explanation

Steps:

- Generates DE
- Convert the DE into RE
- Solves the RE

Examples

```
>>> from sympy import exp, ln
>>> from sympy.series.formal import hyper_algorithm
```

```
>>> from sympy.abc import x, k
```

```
>>> hyper_algorithm(exp(x), x, k)
(Piecewise((1/factorial(k), Eq(Mod(k, 1), 0)), (0, True)), 1, 1)
```

```
>>> hyper_algorithm(ln(1 + x), x, k)
(Piecewise((((-1)**(k - 1)*factorial(k - 1)/RisingFactorial(2, k - 1),
Eq(Mod(k, 1), 0)), (0, True)), x, 2)
```

See also:

[*sympy.series.formal.simpleDE*](#) (page 656), [*sympy.series.formal.solve_de*](#) (page 658)

Limits of Sequences

Provides methods to compute limit of terms having sequences at infinity.

`sympy.series.limitseq.difference_delta(expr, n=None, step=1)`
Difference Operator.

Explanation

Discrete analog of differential operator. Given a sequence $x[n]$, returns the sequence $x[n + \text{step}] - x[n]$.

Examples

```
>>> from sympy import difference_delta as dd
>>> from sympy.abc import n
>>> dd(n*(n + 1), n)
2*n + 2
>>> dd(n*(n + 1), n, 2)
4*n + 6
```

References

[R740]

`sympy.series.limitseq.dominant(expr, n)`

Finds the dominant term in a sum, that is a term that dominates every other term.

Explanation

If $\lim(a/b, n, \infty)$ is ∞ then a dominates b . If $\lim(a/b, n, \infty)$ is 0 then b dominates a . Otherwise, a and b are comparable.

If there is no unique dominant term, then returns `None`.

Examples

```
>>> from sympy import Sum
>>> from sympy.series.limitseq import dominant
>>> from sympy.abc import n, k
>>> dominant(5*n**3 + 4*n**2 + n + 1, n)
5*n**3
>>> dominant(2**n + Sum(k, (k, 0, n)), n)
2**n
```

See also:

[`sympy.series.limitseq.dominant`](#) (page 660)

`sympy.series.limitseq.limit_seq(expr, n=None, trials=5)`

Finds the limit of a sequence as index n tends to infinity.

Parameters

expr : Expr

SymPy expression for the n -th term of the sequence

n : Symbol, optional

The index of the sequence, an integer that tends to positive infinity.
If `None`, inferred from the expression unless it has multiple symbols.

trials: int, optional

The algorithm is highly recursive. `trials` is a safeguard from infinite recursion in case the limit is not easily computed by the algorithm.
Try increasing `trials` if the algorithm returns `None`.

Admissible Terms

The algorithm is designed for sequences built from rational functions, indefinite sums, and indefinite products over an indeterminate n . Terms of alternating sign are also allowed, but more complex oscillatory behavior is not supported.

Examples

```
>>> from sympy import limit_seq, Sum, binomial
>>> from sympy.abc import n, k, m
>>> limit_seq((5*n**3 + 3*n**2 + 4) / (3*n**3 + 4*n - 5), n)
5/3
>>> limit_seq(binomial(2*n, n) / Sum(binomial(2*k, k), (k, 1, n)), n)
3/4
>>> limit_seq(Sum(k**2 * Sum(2**m/m, (m, 1, k)), (k, 1, n)) / (2**n*n),
→n)
4
```

See also:

[`sympy.series.limitseq.dominant`](#) (page 660)

References

[R741]

Simplify

Simplify

simplify

`sympy.simplify.simplify.simplify(expr, ratio=1.7, measure=<function count_ops>, rational=False, inverse=False, doit=True, **kwargs)`

Simplifies the given expression.

Explanation

Simplification is not a well defined term and the exact strategies this function tries can change in the future versions of SymPy. If your algorithm relies on “simplification” (whatever it is), try to determine what you need exactly - is it `powsimp()`?, `radsimp()`?, `together()`?, `logcombine()`?, or something else? And use this particular function directly, because those are well defined and thus your algorithm will be robust.

Nonetheless, especially for interactive use, or when you do not know anything about the structure of the expression, `simplify()` tries to apply intelligent heuristics to make the input expression “simpler”. For example:

```
>>> from sympy import simplify, cos, sin
>>> from sympy.abc import x, y
>>> a = (x + x**2)/(x*sin(y)**2 + x*cos(y)**2)
>>> a
(x**2 + x)/(x*sin(y)**2 + x*cos(y)**2)
>>> simplify(a)
x + 1
```

Note that we could have obtained the same result by using specific simplification functions:

```
>>> from sympy import trigsimp, cancel
>>> trigsimp(a)
(x**2 + x)/x
>>> cancel(_)
x + 1
```

In some cases, applying `simplify()` (page 661) may actually result in some more complicated expression. The default `ratio=1.7` prevents more extreme cases: if `(result length)/(input length) > ratio`, then input is returned unmodified. The `measure` parameter lets you specify the function used to determine how complex an expression is. The function should take a single argument as an expression and return a number such that if expression `a` is more complex than expression `b`, then `measure(a) > measure(b)`. The default measure function is `count_ops()` (page 1059), which returns the total number of operations in the expression.

For example, if `ratio=1`, `simplify` output cannot be longer than input.

```
>>> from sympy import sqrt, simplify, count_ops, oo
>>> root = 1/(sqrt(2)+3)
```

Since `simplify(root)` would result in a slightly longer expression, `root` is returned unchanged instead:

```
>>> simplify(root, ratio=1) == root
True
```

If `ratio=oo`, `simplify` will be applied anyway:

```
>>> count_ops(simplify(root, ratio=oo)) > count_ops(root)
True
```

Note that the shortest expression is not necessarily the simplest, so setting `ratio` to 1 may not be a good idea. Heuristically, the default value `ratio=1.7` seems like a reasonable choice.

You can easily define your own measure function based on what you feel should represent the “size” or “complexity” of the input expression. Note that some choices, such as `lambda expr: len(str(expr))` may appear to be good metrics, but have other problems (in this case, the measure function may slow down `simplify` too much for very large expressions). If you do not know what a good metric would be, the default, `count_ops`, is a good one.

For example:

```

>>> from sympy import symbols, log
>>> a, b = symbols('a b', positive=True)
>>> g = log(a) + log(b) + log(a)*log(1/b)
>>> h = simplify(g)
>>> h
log(a*b**(1 - log(a)))
>>> count_ops(g)
8
>>> count_ops(h)
5

```

So you can see that `h` is simpler than `g` using the `count_ops` metric. However, we may not like how `simplify` (in this case, using `logcombine`) has created the `b**(log(1/a) + 1)` term. A simple way to reduce this would be to give more weight to powers as operations in `count_ops`. We can do this by using the `visual=True` option:

```

>>> print(count_ops(g, visual=True))
2*ADD + DIV + 4*LOG + MUL
>>> print(count_ops(h, visual=True))
2*LOG + MUL + POW + SUB

```

```

>>> from sympy import Symbol, S
>>> def my_measure(expr):
...     POW = Symbol('POW')
...     # Discourage powers by giving POW a weight of 10
...     count = count_ops(expr, visual=True).subs(POW, 10)
...     # Every other operation gets a weight of 1 (the default)
...     count = count.replace(Symbol, type(S.One))
...     return count
>>> my_measure(g)
8
>>> my_measure(h)
14
>>> 15./8 > 1.7 # 1.7 is the default ratio
True
>>> simplify(g, measure=my_measure)
-log(a)*log(b) + log(a) + log(b)

```

Note that because `simplify()` internally tries many different simplification strategies and then compares them using the measure function, we get a completely different result that is still different from the input expression by doing this.

If `rational=True`, Floats will be recast as Rationals before simplification. If `rational=None`, Floats will be recast as Rationals but the result will be recast as Floats. If `rational=False` (default) then nothing will be done to the Floats.

If `inverse=True`, it will be assumed that a composition of inverse functions, such as `sin` and `asin`, can be cancelled in any order. For example, `asin(sin(x))` will yield `x` without checking whether `x` belongs to the set where this relation is true. The default is `False`.

Note that `simplify()` automatically calls `doit()` on the final expression. You can avoid this behavior by passing `doit=False` as an argument.

Also, it should be noted that simplifying the boolean expression is not well defined. If the expression prefers automatic evaluation (such as `Eq()` (page 1022) or `Or()` (page 1167)),

simplification will return True or False if truth value can be determined. If the expression is not evaluated by default (such as `Predicate()` (page 195)), simplification will not reduce it and you should use `refine()` (page 197) or `ask()` (page 191) function. This inconsistency will be resolved in future version.

See also:

[`sympy.assumptions.refine.refine`](#) (page 197)

Simplification using assumptions.

[`sympy.assumptions.ask.ask`](#) (page 191)

Query for boolean expressions using assumptions.

separatevars

`sympy.simplify.simplify.separatevars(expr, symbols=[], dict=False, force=False)`

Separates variables in an expression, if possible. By default, it separates with respect to all symbols in an expression and collects constant coefficients that are independent of symbols.

Explanation

If `dict=True` then the separated terms will be returned in a dictionary keyed to their corresponding symbols. By default, all symbols in the expression will appear as keys; if symbols are provided, then all those symbols will be used as keys, and any terms in the expression containing other symbols or non-symbols will be returned keyed to the string 'coeff'. (Passing None for symbols will return the expression in a dictionary keyed to 'coeff'.)

If `force=True`, then bases of powers will be separated regardless of assumptions on the symbols involved.

Notes

The order of the factors is determined by Mul, so that the separated expressions may not necessarily be grouped together.

Although factoring is necessary to separate variables in some expressions, it is not necessary in all cases, so one should not count on the returned factors being factored.

Examples

```
>>> from sympy.abc import x, y, z, alpha
>>> from sympy import separatevars, sin
>>> separatevars((x*y)**y)
(x*y)**y
>>> separatevars((x*y)**y, force=True)
x**y*y**y
```



```
>>> e = 2*x**2*z*sin(y)+2*z*x**2
>>> separatevars(e)
2*x**2*z*(sin(y) + 1)
>>> separatevars(e, symbols=(x, y), dict=True)
{'coeff': 2*z, x: x**2, y: sin(y) + 1}
>>> separatevars(e, [x, y, alpha], dict=True)
{'coeff': 2*z, alpha: 1, x: x**2, y: sin(y) + 1}
```

If the expression is not really separable, or is only partially separable, `separatevars` will do the best it can to separate it by using factoring.

```
>>> separatevars(x + x*y - 3*x**2)
-x*(3*x - y - 1)
```

If the expression is not separable then `expr` is returned unchanged or (if `dict=True`) then `None` is returned.

```
>>> eq = 2*x + y*sin(x)
>>> separatevars(eq) == eq
True
>>> separatevars(2*x + y*sin(x), symbols=(x, y), dict=True) is None
True
```

nthroot

`sympy.simplify.simplify.nthroot(expr, n, max_len=4, prec=15)`

Compute a real nth-root of a sum of surds.

Parameters

expr : sum of surds

n : integer

max_len : maximum number of surds passed as constants to `nsimplify`

Algorithm

First `nsimplify` is used to get a candidate root; if it is not a root the minimal polynomial is computed; the answer is one of its roots.

Examples

```
>>> from sympy.simplify.simplify import nthroot
>>> from sympy import sqrt
>>> nthroot(90 + 34*sqrt(7), 3)
sqrt(7) + 3
```

kroneckersimp

`sympy.simplify.simplify.kroneckersimp(expr)`

Simplify expressions with KroneckerDelta.

The only simplification currently attempted is to identify multiplicative cancellation:

Examples

```
>>> from sympy import KroneckerDelta, kroneckersimp
>>> from sympy.abc import i
>>> kroneckersimp(1 + KroneckerDelta(0, i) * KroneckerDelta(1, i))
1
```

besselsimp

`sympy.simplify.simplify.besselsimp(expr)`

Simplify bessel-type functions.

Explanation

This routine tries to simplify bessel-type functions. Currently it only works on the Bessel J and I functions, however. It works by looking at all such functions in turn, and eliminating factors of “I” and “-1” (actually their polar equivalents) in front of the argument. Then, functions of half-integer order are rewritten using strigonometric functions and functions of integer order (> 1) are rewritten using functions of low order. Finally, if the expression was changed, compute factorization of the result with `factor()`.

```
>>> from sympy import besselj, besseli, besselsimp, polar_lift, I, S
>>> from sympy.abc import z, nu
>>> besselsimp(besselj(nu, z*polar_lift(-1)))
exp(I*pi*nu)*besselj(nu, z)
>>> besselsimp(besseli(nu, z*polar_lift(-I)))
exp(-I*pi*nu/2)*besselj(nu, z)
>>> besselsimp(besseli(S(-1)/2, z))
sqrt(2)*cosh(z)/(sqrt(pi)*sqrt(z))
>>> besselsimp(z*besseli(0, z) + z*(besseli(2, z))/2 + besseli(1, z))
3*z*besseli(0, z)/2
```

hypersimp

`sympy.simplify.simplify.hypersimp(f, k)`

Given combinatorial term $f(k)$ simplify its consecutive term ratio i.e. $f(k+1)/f(k)$. The input term can be composed of functions and integer sequences which have equivalent representation in terms of gamma special function.

Explanation

The algorithm performs three basic steps:

1. Rewrite all functions in terms of gamma, if possible.
2. Rewrite all occurrences of gamma in terms of products of gamma and rising factorial with integer, absolute constant exponent.
3. Perform simplification of nested fractions, powers and if the resulting expression is a quotient of polynomials, reduce their total degree.

If $f(k)$ is hypergeometric then as result we arrive with a quotient of polynomials of minimal degree. Otherwise None is returned.

For more information on the implemented algorithm refer to:

1. W. Koepf, Algorithms for m-fold Hypergeometric Summation, Journal of Symbolic Computation (1995) 20, 399-417

hypersimilar

`sympy.simplify.simplify.hypersimilar(f, g, k)`

Returns True if f and g are hyper-similar.

Explanation

Similarity in hypergeometric sense means that a quotient of $f(k)$ and $g(k)$ is a rational function in k . This procedure is useful in solving recurrence relations.

For more information see `hypersimp()`.

nsimplify

`sympy.simplify.simplify.nsimplify(expr, constants=(), tolerance=None, full=False, rational=None, rational_conversion='base10')`

Find a simple representation for a number or, if there are free symbols or if `rational=True`, then replace Floats with their Rational equivalents. If no change is made and `rational` is not False then Floats will at least be converted to Rationals.

Explanation

For numerical expressions, a simple formula that numerically matches the given numerical expression is sought (and the input should be possible to evalf to a precision of at least 30 digits).

Optionally, a list of (rationally independent) constants to include in the formula may be given.

A lower tolerance may be set to find less exact matches. If no tolerance is given then the least precise value will set the tolerance (e.g. Floats default to 15 digits of precision, so would be `tolerance=10**-15`).

With `full=True`, a more extensive search is performed (this is useful to find simpler numbers when the tolerance is set low).

When converting to rational, if `rational_conversion='base10'` (the default), then convert floats to rationals using their base-10 (string) representation. When `rational_conversion='exact'` it uses the exact, base-2 representation.

Examples

```
>>> from sympy import nsimplify, sqrt, GoldenRatio, exp, I, pi
>>> nsimplify(4/(1+sqrt(5)), [GoldenRatio])
-2 + 2*GoldenRatio
>>> nsimplify((1/(exp(3*pi*I/5)+1)))
1/2 - I*sqrt(sqrt(5)/10 + 1/4)
>>> nsimplify(I**I, [pi])
exp(-pi/2)
>>> nsimplify(pi, tolerance=0.01)
22/7
```

```
>>> nsimplify(0.333333333333333, rational=True, rational_conversion=
↳ 'exact')
6004799503160655/18014398509481984
>>> nsimplify(0.333333333333333, rational=True)
1/3
```

See also:

[*sympy.core.function.nfloat*](#) (page 1064)

posify

`sympy.simplify.simplify.posify(eq)`

Return `eq` (with generic symbols made positive) and a dictionary containing the mapping between the old and new symbols.

Explanation

Any symbol that has `positive=None` will be replaced with a positive dummy symbol having the same name. This replacement will allow more symbolic processing of expressions, especially those involving powers and logarithms.

A dictionary that can be sent to `subs` to restore `eq` to its original symbols is also returned.

```
>>> from sympy import posify, Symbol, log, solve
>>> from sympy.abc import x
>>> posify(x + Symbol('p', positive=True) + Symbol('n', negative=True))
(_x + n + p, {_x: x})
```

```
>>> eq = 1/x
>>> log(eq).expand()
log(1/x)
>>> log(posify(eq)[0]).expand()
-log(_x)
>>> p, rep = posify(eq)
>>> log(p).expand().subs(rep)
-log(x)
```

It is possible to apply the same transformations to an iterable of expressions:

```
>>> eq = x**2 - 4
>>> solve(eq, x)
[-2, 2]
>>> eq_x, reps = posify([eq, x]); eq_x
[_x**2 - 4, _x]
>>> solve(*eq_x)
[2]
```

logcombine

`sympy.simplify.simplify.logcombine(expr, force=False)`

Takes logarithms and combines them using the following rules:

- $\log(x) + \log(y) == \log(x*y)$ if both are positive
- $a*\log(x) == \log(x**a)$ if x is positive and a is real

If `force` is `True` then the assumptions above will be assumed to hold if there is no assumption already in place on a quantity. For example, if a is imaginary or the argument negative, `force` will not perform a combination but if a is a symbol with no assumptions the change will take place.

Examples

```
>>> from sympy import Symbol, symbols, log, logcombine, I
>>> from sympy.abc import a, x, y, z
>>> logcombine(a*log(x) + log(y) - log(z))
a*log(x) + log(y) - log(z)
>>> logcombine(a*log(x) + log(y) - log(z), force=True)
log(x**a*y/z)
>>> x,y,z = symbols('x,y,z', positive=True)
>>> a = Symbol('a', real=True)
>>> logcombine(a*log(x) + log(y) - log(z))
log(x**a*y/z)
```

The transformation is limited to factors and/or terms that contain logs, so the result depends on the initial state of expansion:

```
>>> eq = (2 + 3*I)*log(x)
>>> logcombine(eq, force=True) == eq
True
>>> logcombine(eq.expand(), force=True)
log(x**2) + I*log(x**3)
```

See also:

[*posify* \(page 668\)](#)

replace all symbols with symbols having positive assumptions

[*sympy.core.function.expand_log* \(page 1061\)](#)

expand the logarithms of products and powers; the opposite of `logcombine`

Radsimp

radsimp

`sympy.simplify.radsimp.radsimp(expr, symbolic=True, max_terms=4)`

Rationalize the denominator by removing square roots.

Explanation

The expression returned from `radsimp` must be used with caution since if the denominator contains symbols, it will be possible to make substitutions that violate the assumptions of the simplification process: that for a denominator matching $a + b\sqrt{c}$, $a \neq \pm b\sqrt{c}$. (If there are no symbols, this assumption is made valid by collecting terms of $\sqrt{c}$ so the match variable a does not contain $\sqrt{c}$.) If you do not want the simplification to occur for symbolic denominators, set `symbolic` to `False`.

If there are more than `max_terms` radical terms then the expression is returned unchanged.

Examples

```
>>> from sympy import radsimp, sqrt, Symbol, pprint
>>> from sympy import factor_terms, fraction, signsimp
>>> from sympy.simplify.radsimp import collect_sqrt
>>> from sympy.abc import a, b, c
```

```
>>> radsimp(1/(2 + sqrt(2)))
(2 - sqrt(2))/2
>>> x,y = map(Symbol, 'xy')
>>> e = ((2 + 2*sqrt(2))*x + (2 + sqrt(8))*y)/(2 + sqrt(2))
>>> radsimp(e)
sqrt(2)*(x + y)
```

No simplification beyond removal of the gcd is done. One might want to polish the result a little, however, by collecting square root terms:

```
>>> r2 = sqrt(2)
>>> r5 = sqrt(5)
>>> ans = radsimp(1/(y*r2 + x*r2 + a*r5 + b*r5)); pprint(ans)
```

$$\frac{\sqrt{5}^2 a + \sqrt{5}^2 b - \sqrt{2}^2 x - \sqrt{2}^2 y}{5^2 a^2 + 10^2 a b + 5^2 b^2 - 2^2 x^2 - 4^2 x y - 2^2 y^2}$$

```
>>> n, d = fraction(ans)
>>> pprint(factor_terms(signsimp(collect_sqrt(n))/d, radical=True))
```

$$\frac{\sqrt{5}^2 (a + b) - \sqrt{2}^2 (x + y)}{5^2 a^2 + 10^2 a b + 5^2 b^2 - 2^2 x^2 - 4^2 x y - 2^2 y^2}$$

If radicals in the denominator cannot be removed or there is no denominator, the original expression will be returned.

```
>>> radsimp(sqrt(2)*x + sqrt(2))
sqrt(2)*x + sqrt(2)
```

Results with symbols will not always be valid for all substitutions:

```
>>> eq = 1/(a + b*sqrt(c))
>>> eq.subs(a, b*sqrt(c))
1/(2*b*sqrt(c))
>>> radsimp(eq).subs(a, b*sqrt(c))
nan
```

If symbolic=False, symbolic denominators will not be transformed (but numeric denominators will still be processed):

```
>>> radsimp(eq, symbolic=False)
1/(a + b*sqrt(c))
```

rad_rationalize

`sympy.simplify.radsimp.rad_rationalize(num, den)`

Rationalize num/den by removing square roots in the denominator; num and den are sum of terms whose squares are positive rationals.

Examples

```
>>> from sympy import sqrt
>>> from sympy.simplify.radsimp import rad_rationalize
>>> rad_rationalize(sqrt(3), 1 + sqrt(2)/3)
(-sqrt(3) + sqrt(6)/3, -7/9)
```

collect

`sympy.simplify.radsimp.collect(expr, syms, func=None, evaluate=None, exact=False, distribute_order_term=True)`

Collect additive terms of an expression.

Explanation

This function collects additive terms of an expression with respect to a list of expression up to powers with rational exponents. By the term symbol here are meant arbitrary expressions, which can contain powers, products, sums etc. In other words symbol is a pattern which will be searched for in the expression's terms.

The input expression is not expanded by `collect()` (page 672), so user is expected to provide an expression in an appropriate form. This makes `collect()` (page 672) more predictable as there is no magic happening behind the scenes. However, it is important to note, that powers of products are converted to products of powers using the `expand_power_base()` (page 1063) function.

There are two possible types of output. First, if `evaluate` flag is set, this function will return an expression with collected terms or else it will return a dictionary with expressions up to rational powers as keys and collected coefficients as values.

Examples

```
>>> from sympy import S, collect, expand, factor, Wild
>>> from sympy.abc import a, b, c, x, y
```

This function can collect symbolic coefficients in polynomials or rational expressions. It will manage to find all integer or rational powers of collection variable:

```
>>> collect(a*x**2 + b*x**2 + a*x - b*x + c, x)
c + x**2*(a + b) + x*(a - b)
```

The same result can be achieved in dictionary form:


```
>>> d = collect(a*x**2 + b*x**2 + a*x - b*x + c, x, evaluate=False)
>>> d[x**2]
a + b
>>> d[x]
a - b
>>> d[S.One]
c
```

You can also work with multivariate polynomials. However, remember that this function is greedy so it will care only about a single symbol at time, in specification order:

```
>>> collect(x**2 + y*x**2 + x*y + y + a*y, [x, y])
x**2*(y + 1) + x*y + y*(a + 1)
```

Also more complicated expressions can be used as patterns:

```
>>> from sympy import sin, log
>>> collect(a*sin(2*x) + b*sin(2*x), sin(2*x))
(a + b)*sin(2*x)

>>> collect(a*x*log(x) + b*(x*log(x)), x*log(x))
x*(a + b)*log(x)
```

You can use wildcards in the pattern:

```
>>> w = Wild('w1')
>>> collect(a*x**y - b*x**y, w**y)
x**y*(a - b)
```

It is also possible to work with symbolic powers, although it has more complicated behavior, because in this case power's base and symbolic part of the exponent are treated as a single symbol:

```
>>> collect(a*x**c + b*x**c, x)
a*x**c + b*x**c
>>> collect(a*x**c + b*x**c, x**c)
x**c*(a + b)
```

However if you incorporate rationals to the exponents, then you will get well known behavior:

```
>>> collect(a*x**(2*c) + b*x**(2*c), x**c)
x**(2*c)*(a + b)
```

Note also that all previously stated facts about `collect()` (page 672) function apply to the exponential function, so you can get:

```
>>> from sympy import exp
>>> collect(a*exp(2*x) + b*exp(2*x), exp(x))
(a + b)*exp(2*x)
```

If you are interested only in collecting specific powers of some symbols then set `exact` flag to `True`:

```
>>> collect(a*x**7 + b*x**7, x, exact=True)
a*x**7 + b*x**7
>>> collect(a*x**7 + b*x**7, x**7, exact=True)
x**7*(a + b)
```

If you want to collect on any object containing symbols, set `exact` to `None`:

```
>>> collect(x*exp(x) + sin(x)*y + sin(x)*2 + 3*x, x, exact=None)
x*exp(x) + 3*x + (y + 2)*sin(x)
>>> collect(a*x*y + x*y + b*x + x, [x, y], exact=None)
x*y*(a + 1) + x*(b + 1)
```

You can also apply this function to differential equations, where derivatives of arbitrary order can be collected. Note that if you collect with respect to a function or a derivative of a function, all derivatives of that function will also be collected. Use `exact=True` to prevent this from happening:

```
>>> from sympy import Derivative as D, collect, Function
>>> f = Function('f')(x)

>>> collect(a*D(f,x) + b*D(f,x), D(f,x))
(a + b)*Derivative(f(x), x)

>>> collect(a*D(D(f,x),x) + b*D(D(f,x),x), f)
(a + b)*Derivative(f(x), (x, 2))

>>> collect(a*D(D(f,x),x) + b*D(D(f,x),x), D(f,x), exact=True)
a*Derivative(f(x), (x, 2)) + b*Derivative(f(x), (x, 2))

>>> collect(a*D(f,x) + b*D(f,x) + a*f + b*f, f)
(a + b)*f(x) + (a + b)*Derivative(f(x), x)
```

Or you can even match both derivative order and exponent at the same time:

```
>>> collect(a*D(D(f,x),x)**2 + b*D(D(f,x),x)**2, D(f,x))
(a + b)*Derivative(f(x), (x, 2))**2
```

Finally, you can apply a function to each of the collected coefficients. For example you can factorize symbolic coefficients of polynomial:

```
>>> f = expand((x + a + 1)**3)

>>> collect(f, x, factor)
x**3 + 3*x**2*(a + 1) + 3*x*(a + 1)**2 + (a + 1)**3
```

Note: Arguments are expected to be in expanded form, so you might have to call `expand()` (page 1053) prior to calling this function.

See also:

[`collect\_const`](#) (page 676), [`collect\_sqrt`](#) (page 675), [`rcollect`](#) (page 674)

`sympy.simplify.radsimp.rcollect(expr, *vars)`

Recursively collect sums in an expression.

Examples

```
>>> from sympy.simplify import rcollect
>>> from sympy.abc import x, y
```

```
>>> expr = (x**2*y + x*y + x + y)/(x + y)
```

```
>>> rcollect(expr, y)
(x + y*(x**2 + x + 1))/(x + y)
```

See also:

[collect](#) (page 672), [collect_const](#) (page 676), [collect_sqrt](#) (page 675)

collect_sqrt

`sympy.simplify.radsimp.collect_sqrt(expr, evaluate=None)`

Return `expr` with terms having common square roots collected together. If `evaluate` is `False` a count indicating the number of `sqrt`-containing terms will be returned and, if non-zero, the terms of the Add will be returned, else the expression itself will be returned as a single term. If `evaluate` is `True`, the expression with any collected terms will be returned.

Note: since $I = \sqrt{-1}$, it is collected, too.

Examples

```
>>> from sympy import sqrt
>>> from sympy.simplify.radsimp import collect_sqrt
>>> from sympy.abc import a, b
```

```
>>> r2, r3, r5 = [sqrt(i) for i in [2, 3, 5]]
>>> collect_sqrt(a*r2 + b*r2)
sqrt(2)*(a + b)
>>> collect_sqrt(a*r2 + b*r2 + a*r3 + b*r3)
sqrt(2)*(a + b) + sqrt(3)*(a + b)
>>> collect_sqrt(a*r2 + b*r2 + a*r3 + b*r5)
sqrt(3)*a + sqrt(5)*b + sqrt(2)*(a + b)
```

If `evaluate` is `False` then the arguments will be sorted and returned as a list and a count of the number of `sqrt`-containing terms will be returned:

```
>>> collect_sqrt(a*r2 + b*r2 + a*r3 + b*r5, evaluate=False)
((sqrt(3)*a, sqrt(5)*b, sqrt(2)*(a + b)), 3)
>>> collect_sqrt(a*sqrt(2) + b, evaluate=False)
((b, sqrt(2)*a), 1)
>>> collect_sqrt(a + b, evaluate=False)
((a + b,), 0)
```

See also:

[collect](#) (page 672), [collect_const](#) (page 676), [rcollect](#) (page 674)

collect_const

`sympy.simplify.radsimp.collect_const(expr, *vars, Numbers=True)`

A non-greedy collection of terms with similar number coefficients in an Add expr. If vars is given then only those constants will be targeted. Although any Number can also be targeted, if this is not desired set Numbers=False and no Float or Rational will be collected.

Parameters

expr : SymPy expression

This parameter defines the expression the expression from which terms with similar coefficients are to be collected. A non-Add expression is returned as it is.

vars : variable length collection of Numbers, optional

Specifies the constants to target for collection. Can be multiple in number.

Numbers : bool

Specifies to target all instance of `sympy.core.numbers.Number` (page 981) class. If Numbers=False, then no Float or Rational will be collected.

Returns

expr : Expr

Returns an expression with similar coefficient terms collected.

Examples

```
>>> from sympy import sqrt
>>> from sympy.abc import s, x, y, z
>>> from sympy.simplify.radsimp import collect_const
>>> collect_const(sqrt(3) + sqrt(3)*(1 + sqrt(2)))
sqrt(3)*(sqrt(2) + 2)
>>> collect_const(sqrt(3)*s + sqrt(7)*s + sqrt(3) + sqrt(7))
(sqrt(3) + sqrt(7))*(s + 1)
>>> s = sqrt(2) + 2
>>> collect_const(sqrt(3)*s + sqrt(3) + sqrt(7)*s + sqrt(7))
(sqrt(2) + 3)*(sqrt(3) + sqrt(7))
>>> collect_const(sqrt(3)*s + sqrt(3) + sqrt(7)*s + sqrt(7), sqrt(3))
sqrt(7) + sqrt(3)*(sqrt(2) + 3) + sqrt(7)*(sqrt(2) + 2)
```

The collection is sign-sensitive, giving higher precedence to the unsigned values:

```
>>> collect_const(x - y - z)
x - (y + z)
>>> collect_const(-y - z)
-(y + z)
>>> collect_const(2*x - 2*y - 2*z, 2)
2*(x - y - z)
>>> collect_const(2*x - 2*y - 2*z, -2)
2*x - 2*(y + z)
```

See also:

[`collect`](#) (page 672), [`collect\_sqrt`](#) (page 675), [`rcollect`](#) (page 674)

fraction

`sympy.simplify.radsimp.fraction(expr, exact=False)`

Returns a pair with expression's numerator and denominator. If the given expression is not a fraction then this function will return the tuple `(expr, 1)`.

This function will not make any attempt to simplify nested fractions or to do any term rewriting at all.

If only one of the numerator/denominator pair is needed then use `numer(expr)` or `denom(expr)` functions respectively.

```
>>> from sympy import fraction, Rational, Symbol
>>> from sympy.abc import x, y
```

```
>>> fraction(x/y)
(x, y)
>>> fraction(x)
(x, 1)
```

```
>>> fraction(1/y**2)
(1, y**2)
```

```
>>> fraction(x*y/2)
(x*y, 2)
>>> fraction(Rational(1, 2))
(1, 2)
```

This function will also work fine with assumptions:

```
>>> k = Symbol('k', negative=True)
>>> fraction(x * y**k)
(x, y**(-k))
```

If we know nothing about sign of some exponent and `exact` flag is unset, then structure this exponent's structure will be analyzed and pretty fraction will be returned:

```
>>> from sympy import exp, Mul
>>> fraction(2*x**(-y))
(2, x**y)
```

```
>>> fraction(exp(-x))
(1, exp(x))
```

```
>>> fraction(exp(-x), exact=True)
(exp(-x), 1)
```

The `exact` flag will also keep any unevaluated Muls from being evaluated:

```
>>> u = Mul(2, x + 1, evaluate=False)
>>> fraction(u)
(2*x + 2, 1)
>>> fraction(u, exact=True)
(2*(x + 1), 1)
```

Ratsimp

ratsimp

`sympy.simplify.ratsimp.ratsimp(expr)`

Put an expression over a common denominator, cancel and reduce.

Examples

```
>>> from sympy import ratsimp
>>> from sympy.abc import x, y
>>> ratsimp(1/x + 1/y)
(x + y)/(x*y)
```

ratsimpmodprime

`sympy.simplify.ratsimp.ratsimpmodprime(expr, G, *gens, quick=True, polynomial=False, **args)`

Simplifies a rational expression `expr` modulo the prime ideal generated by `G`. `G` should be a Groebner basis of the ideal.

Examples

```
>>> from sympy.simplify.ratsimp import ratsimpmodprime
>>> from sympy.abc import x, y
>>> eq = (x + y**5 + y)/(x - y)
>>> ratsimpmodprime(eq, [x*y**5 - x - y], x, y, order='lex')
(-x**2 - x*y - x - y)/(-x**2 + x*y)
```

If `polynomial` is `False`, the algorithm computes a rational simplification which minimizes the sum of the total degrees of the numerator and the denominator.

If `polynomial` is `True`, this function just brings numerator and denominator into a canonical form. This is much faster, but has potentially worse results.

References

[R762]

Trigonometric simplification

trigsimp

`sympy.simplify.trigsimp.trigsimp(expr, **opts)`

Returns a reduced expression by using known trig identities.

Parameters

method : string, optional

Specifies the method to use. Valid choices are:

- 'matching', default
- 'groebner'
- 'combined'
- 'fu'
- 'old'

If 'matching', simplify the expression recursively by targeting common patterns. If 'groebner', apply an experimental groebner basis algorithm. In this case further options are forwarded to `trigsimp_groebner`, please refer to its docstring. If 'combined', it first runs the groebner basis algorithm with small default parameters, then runs the 'matching' algorithm. If 'fu', run the collection of trigonometric transformations described by Fu, et al. (see the [fu\(\)](#) (page 712) docstring). If 'old', the original SymPy trig simplification function is run.

opts :

Optional keyword arguments passed to the method. See each method's function docstring for details.

Examples

```
>>> from sympy import trigsimp, sin, cos, log
>>> from sympy.abc import x
>>> e = 2*sin(x)**2 + 2*cos(x)**2
>>> trigsimp(e)
2
```

Simplification occurs wherever trigonometric functions are located.

```
>>> trigsimp(log(e))
log(2)
```

Using `method='groebner'` (or `method='combined'`) might lead to greater simplification. The old `trigsimp` routine can be accessed as with `method='old'`.

```
>>> from sympy import coth, tanh
>>> t = 3*tanh(x)**7 - 2/coth(x)**7
>>> trigsimp(t, method='old') == t
True
>>> trigsimp(t)
tanh(x)**7
```

Power simplification

powsimp

`sympy.simplify.powsimp.powsimp(expr, deep=False, combine='all', force=False, measure=<function count_ops>)`

reduces expression by combining powers with similar bases and exponents.

Explanation

If `deep` is `True` then `powsimp()` will also simplify arguments of functions. By default `deep` is set to `False`.

If `force` is `True` then bases will be combined without checking for assumptions, e.g. `sqrt(x)*sqrt(y) -> sqrt(x*y)` which is not true if `x` and `y` are both negative.

You can make `powsimp()` only combine bases or only combine exponents by changing `combine='base'` or `combine='exp'`. By default, `combine='all'`, which does both. `combine='base'` will only combine:

```
  a   a           a           2x      x
x * y => (x*y)  as well as things like 2  => 4
```

and `combine='exp'` will only combine

```
  a   b      (a + b)
x * x => x
```

`combine='exp'` will strictly only combine exponents in the way that used to be automatic. Also use `deep=True` if you need the old behavior.

When `combine='all'`, `'exp'` is evaluated first. Consider the first example below for when there could be an ambiguity relating to this. This is done so things like the second example can be completely combined. If you want `'base'` combined first, do something like `powsimp(powsimp(expr, combine='base'), combine='exp')`.

Examples

```
>>> from sympy import powsimp, exp, log, symbols
>>> from sympy.abc import x, y, z, n
>>> powsimp(x**y*x**z*y**z, combine='all')
x**(y + z)*y**z
>>> powsimp(x**y*x**z*y**z, combine='exp')
x**(y + z)*y**z
>>> powsimp(x**y*x**z*y**z, combine='base', force=True)
x**y*(x*y)**z
```

```
>>> powsimp(x**z*x**y*n**z*n**y, combine='all', force=True)
(n*x)**(y + z)
>>> powsimp(x**z*x**y*n**z*n**y, combine='exp')
n**(y + z)*x**(y + z)
>>> powsimp(x**z*x**y*n**z*n**y, combine='base', force=True)
(n*x)**y*(n*x)**z
```

```
>>> x, y = symbols('x y', positive=True)
>>> powsimp(log(exp(x)*exp(y)))
log(exp(x)*exp(y))
>>> powsimp(log(exp(x)*exp(y)), deep=True)
x + y
```

Radicals with Mul bases will be combined if combine='exp'

```
>>> from sympy import sqrt
>>> x, y = symbols('x y')
```

Two radicals are automatically joined through Mul:

```
>>> a=sqrt(x*sqrt(y))
>>> a*a**3 == a**4
True
```

But if an integer power of that radical has been autoexpanded then Mul does not join the resulting factors:

```
>>> a**4 # auto expands to a Mul, no longer a Pow
x**2*y
>>> _*a # so Mul doesn't combine them
x**2*y*sqrt(x*sqrt(y))
>>> powsimp(_) # but powsimp will
(x*sqrt(y))**(5/2)
>>> powsimp(x*y*a) # but won't when doing so would violate assumptions
x*y*sqrt(x*sqrt(y))
```

powdenest

`sympy.simplify.powsimp.powdenest(eq, force=False, polar=False)`

Collect exponents on powers as assumptions allow.

Explanation

Given $(bb^{**be})^{**e}$, this can be simplified as follows:

- if bb is positive, or
- e is an integer, or
- $|be| < 1$ then this simplifies to $bb^{*(be*e)}$

Given a product of powers raised to a power, $(bb1^{*be1} * bb2^{*be2} \dots)^{**e}$, simplification can be done as follows:

- if e is positive, the gcd of all be_i can be joined with e ;
- all non-negative bb can be separated from those that are negative and their gcd can be joined with e ; autosimplification already handles this separation.
- integer factors from powers that have integers in the denominator of the exponent can be removed from any term and the gcd of such integers can be joined with e

Setting `force` to `True` will make symbols that are not explicitly negative behave as though they are positive, resulting in more denesting.

Setting `polar` to `True` will do simplifications on the Riemann surface of the logarithm, also resulting in more denestings.

When there are sums of logs in `exp()` then a product of powers may be obtained e.g. `exp(3*(log(a) + 2*log(b))) -> a**3*b**6`.

Examples

```
>>> from sympy.abc import a, b, x, y, z
>>> from sympy import Symbol, exp, log, sqrt, symbols, powdenest
```

```
>>> powdenest((x**(2*a/3))**(3*x))
(x**(2*a/3))**(3*x)
>>> powdenest(exp(3*x*log(2)))
2**(3*x)
```

Assumptions may prevent expansion:

```
>>> powdenest(sqrt(x**2))
sqrt(x**2)
```

```
>>> p = symbols('p', positive=True)
>>> powdenest(sqrt(p**2))
p
```

No other expansion is done.

```
>>> i, j = symbols('i,j', integer=True)
>>> powdenest((x**x)**(i + j)) # -X-> (x**x)**i*(x**x)**j
x**(x*(i + j))
```

But `exp()` will be denested by moving all non-log terms outside of the function; this may result in the collapsing of the `exp` to a power with a different base:

```
>>> powdenest(exp(3*y*log(x)))
x**(3*y)
>>> powdenest(exp(y*(log(a) + log(b))))
(a*b)**y
>>> powdenest(exp(3*(log(a) + log(b))))
a**3*b**3
```

If assumptions allow, symbols can also be moved to the outermost exponent:

```
>>> i = Symbol('i', integer=True)
>>> powdenest(((x**(2*i))**(3*y))**x)
((x**(2*i))**(3*y))**x
>>> powdenest(((x**(2*i))**(3*y))**x, force=True)
x**(6*i*x*y)
```

```
>>> powdenest(((x**(2*a/3))**(3*y/i))**x)
((x**(2*a/3))**(3*y/i))**x
>>> powdenest((x**(2*i)*y**(4*i))**z, force=True)
(x*y**2)**(2*i*z)
```

```
>>> n = Symbol('n', negative=True)
```

```
>>> powdenest((x**i)**y, force=True)
x**(i*y)
>>> powdenest((n**i)**x, force=True)
(n**i)**x
```

Combinatorial simplification

combsimp

`sympy.simplify.combsimp.combsimp(expr)`
Simplify combinatorial expressions.

Explanation

This function takes as input an expression containing factorials, binomials, Pochhammer symbol and other “combinatorial” functions, and tries to minimize the number of those functions and reduce the size of their arguments.

The algorithm works by rewriting all combinatorial functions as gamma functions and applying `gammasimp()` except simplification steps that may make an integer argument non-integer. See docstring of `gammasimp` for more information.

Then it rewrites expression in terms of factorials and binomials by rewriting gammas as factorials and converting $(a+b)!/a!b!$ into binomials.

If expression has gamma functions or combinatorial functions with non-integer argument, it is automatically passed to `gammasimp`.

Examples

```
>>> from sympy.simplify import combsimp
>>> from sympy import factorial, binomial, symbols
>>> n, k = symbols('n k', integer = True)
```

```
>>> combsimp(factorial(n)/factorial(n - 3))
n*(n - 2)*(n - 1)
>>> combsimp(binomial(n+1, k+1)/binomial(n, k))
(n + 1)/(k + 1)
```

Square Root Denesting

`sqrtdenest`

`sympy.simplify.sqrtdenest.sqrtdenest(expr, max_iter=3)`

Denests sqrts in an expression that contain other square roots if possible, otherwise returns the expr unchanged. This is based on the algorithms of [1].

Examples

```
>>> from sympy.simplify.sqrtdenest import sqrtdenest
>>> from sympy import sqrt
>>> sqrtdenest(sqrt(5 + 2 * sqrt(6)))
sqrt(2) + sqrt(3)
```

See also:

[`sympy.solvers.solvers.unrad`](#) (page 851)

References

[R763], [R764]

Common Subexpression Elimination

cse

```
sympy.simplify.cse_main.cse(exprs, symbols=None, optimizations=None,
                             postprocess=None, order='canonical', ignore=(),
                             list=True)
```

Perform common subexpression elimination on an expression.

Parameters

exprs : list of SymPy expressions, or a single SymPy expression

The expressions to reduce.

symbols : infinite iterator yielding unique Symbols

The symbols used to label the common subexpressions which are pulled out. The `numbered_symbols` generator is useful. The default is a stream of symbols of the form “x0”, “x1”, etc. This must be an infinite iterator.

optimizations : list of (callable, callable) pairs

The (preprocessor, postprocessor) pairs of external optimization functions. Optionally ‘basic’ can be passed for a set of predefined basic optimizations. Such ‘basic’ optimizations were used by default in old implementation, however they can be really slow on larger expressions. Now, no pre or post optimizations are made by default.

postprocess : a function which accepts the two return values of cse and

returns the desired form of output from cse, e.g. if you want the replacements reversed the function might be the following lambda:
lambda r, e: return reversed(r), e

order : string, ‘none’ or ‘canonical’

The order by which Mul and Add arguments are processed. If set to ‘canonical’, arguments will be canonically ordered. If set to ‘none’, ordering will be faster but dependent on expressions hashes, thus machine dependent and variable. For large expressions where speed is a concern, use the setting `order='none'`.

ignore : iterable of Symbols

Substitutions containing any Symbol from ignore will be ignored.

list : bool, (default True)

Returns expression in list or else with same type as input (when False).

Returns

replacements : list of (Symbol, expression) pairs

All of the common subexpressions that were replaced. Subexpressions earlier in this list might show up in subexpressions later in this list.

reduced_exprs : list of SymPy expressions

The reduced expressions with all of the replacements above.

Examples

```
>>> from sympy import cse, SparseMatrix
>>> from sympy.abc import x, y, z, w
>>> cse((w + x + y + z)*(w + y + z))/(w + x)**3)
([(x0, y + z), (x1, w + x)], [(w + x0)*(x0 + x1)/x1**3])
```

List of expressions with recursive substitutions:

```
>>> m = SparseMatrix([x + y, x + y + z])
>>> cse([(x+y)**2, x + y + z, y + z, x + z + y, m])
([(x0, x + y), (x1, x0 + z)], [x0**2, x1, y + z, x1, Matrix([
[x0],
[x1]])])
```

Note: the type and mutability of input matrices is retained.

```
>>> isinstance(_[1][-1], SparseMatrix)
True
```

The user may disallow substitutions containing certain symbols:

```
>>> cse([y**2*(x + 1), 3*y**2*(x + 1)], ignore=(y,))
([(x0, x + 1)], [x0*y**2, 3*x0*y**2])
```

The default return value for the reduced expression(s) is a list, even if there is only one expression. The *list* flag preserves the type of the input in the output:

```
>>> cse(x)
([], [x])
>>> cse(x, list=False)
([], x)
```

opt_cse

`sympy.simplify.cse_main.opt_cse(exprs, order='canonical')`

Find optimization opportunities in Adds, Muls, Pows and negative coefficient Muls.

Parameters

exprs : list of SymPy expressions

The expressions to optimize.

order : string, 'none' or 'canonical'

The order by which Mul and Add arguments are processed. For large expressions where speed is a concern, use the setting `order='none'`.

Returns**opt_subs** : dictionary of expression substitutions

The expression substitutions which can be useful to optimize CSE.

Examples

```
>>> from sympy.simplify.cse_main import opt_cse
>>> from sympy.abc import x
>>> opt_subs = opt_cse([x**2])
>>> k, v = list(opt_subs.keys())[0], list(opt_subs.values())[0]
>>> print((k, v.as_unevaluated_basic()))
(x**2, 1/(x**2))
```

tree_cse

`sympy.simplify.cse_main.tree_cse(exprs, symbols, opt_subs=None, order='canonical', ignore=())`

Perform raw CSE on expression tree, taking `opt_subs` into account.**Parameters****exprs** : list of SymPy expressions

The expressions to reduce.

symbols : infinite iterator yielding unique Symbols

The symbols used to label the common subexpressions which are pulled out.

opt_subs : dictionary of expression substitutions

The expressions to be substituted before any CSE action is performed.

order : string, 'none' or 'canonical'The order by which Mul and Add arguments are processed. For large expressions where speed is a concern, use the setting `order='none'`.**ignore** : iterable of SymbolsSubstitutions containing any Symbol from `ignore` will be ignored.**Hypergeometric Function Expansion****hyperexpand**

`sympy.simplify.hyperexpand.hyperexpand(f, allow_hyper=False, rewrite='default', place=None)`

Expand hypergeometric functions. If `allow_hyper` is True, allow partial simplification (that is a result different from input, but still containing hypergeometric functions).If a G-function has expansions both at zero and at infinity, `place` can be set to 0 or `zoo` to indicate the preferred choice.

Examples

```
>>> from sympy.simplify.hyperexpand import hyperexpand
>>> from sympy.functions import hyper
>>> from sympy.abc import z
>>> hyperexpand(hyper([], [], z))
exp(z)
```

Non-hypergeometric parts of the expression and hypergeometric expressions that are not recognised are left unchanged:

```
>>> hyperexpand(1 + hyper([1, 1, 1], [], z))
hyper((1, 1, 1), (), z) + 1
```

EPath Tools

EPath class

class sympy.simplify.epathtools.EPath(*path*)

Manipulate expressions using paths.

EPath grammar in EBNF notation:

```
literal    ::= /[A-Za-z_][A-Za-z_0-9]*/
number     ::= /-?\d+/
type       ::= literal
attribute  ::= literal "?"
all        ::= "*"
slice      ::= "[" number? (":" number? (":" number?))?" "]"
range      ::= all | slice
query      ::= (type | attribute) ("|" (type | attribute))*
selector   ::= range | query range?
path       ::= "/" selector ("/" selector)*
```

See the docstring of the epath() function.

apply(*expr*, *func*, *args=None*, *kwargs=None*)

Modify parts of an expression selected by a path.

Examples

```
>>> from sympy.simplify.epathtools import EPath
>>> from sympy import sin, cos, E
>>> from sympy.abc import x, y, z, t
```

```
>>> path = EPath("*/[0]/Symbol")
>>> expr = [(x, 1), 2), ((3, y), z)]
```

```
>>> path.apply(expr, lambda expr: expr**2)
[(x**2, 1), 2), ((3, y**2), z)]
```



```
>>> path = EPath("/*/*/Symbol")
>>> expr = t + sin(x + 1) + cos(x + y + E)
```

```
>>> path.apply(expr, lambda expr: 2*expr)
t + sin(2*x + 1) + cos(2*x + 2*y + E)
```

select(*expr*)

Retrieve parts of an expression selected by a path.

Examples

```
>>> from sympy.simplify.epathtools import EPath
>>> from sympy import sin, cos, E
>>> from sympy.abc import x, y, z, t
```

```
>>> path = EPath("/*/[0]/Symbol")
>>> expr = [(x, 1), 2), ((3, y), z)]
```

```
>>> path.select(expr)
[x, y]
```

```
>>> path = EPath("/*/*/Symbol")
>>> expr = t + sin(x + 1) + cos(x + y + E)
```

```
>>> path.select(expr)
[x, x, y]
```

epath

`sympy.simplify.epathtools.epath`(*path*, *expr=None*, *func=None*, *args=None*, *kwargs=None*)

Manipulate parts of an expression selected by a path.

Parameters

path : str | EPath

A path as a string or a compiled EPath.

expr : Basic | iterable

An expression or a container of expressions.

func : callable (optional)

A callable that will be applied to matching parts.

args : tuple (optional)

Additional positional arguments to func.

kwargs : dict (optional)

Additional keyword arguments to func.

Explanation

This function allows to manipulate large nested expressions in single line of code, utilizing techniques to those applied in XML processing standards (e.g. XPath).

If `func` is `None`, `epath()` (page 689) retrieves elements selected by the path. Otherwise it applies `func` to each matching element.

Note that it is more efficient to create an `EPath` object and use the `select` and `apply` methods of that object, since this will compile the path string only once. This function should only be used as a convenient shortcut for interactive use.

This is the supported syntax:

- **select all:** `/*`
Equivalent of `for arg in args:`.
- **select slice:** `/[0]` or `/[1:5]` or `/[1:5:2]`
Supports standard Python's slice syntax.
- **select by type:** `/list` or `/list|tuple`
Emulates `isinstance()`.
- **select by attribute:** `/__iter__?`
Emulates `hasattr()`.

Examples

```
>>> from sympy.simplify.epathtools import epath
>>> from sympy import sin, cos, E
>>> from sympy.abc import x, y, z, t
```

```
>>> path = "/*/[0]/Symbol"
>>> expr = [(x, 1), 2), ((3, y), z)]
```

```
>>> epath(path, expr)
[x, y]
>>> epath(path, expr, lambda expr: expr**2)
[((x**2, 1), 2), ((3, y**2), z)]
```

```
>>> path = "/*/*Symbol"
>>> expr = t + sin(x + 1) + cos(x + y + E)
```

```
>>> epath(path, expr)
[x, x, y]
>>> epath(path, expr, lambda expr: 2*expr)
t + sin(2*x + 1) + cos(2*x + 2*y + E)
```

Hypergeometric Expansion

This page describes how the function `hyperexpand()` (page 687) and related code work. For usage, see the documentation of the `sympify` module.

Hypergeometric Function Expansion Algorithm

This section describes the algorithm used to expand hypergeometric functions. Most of it is based on the papers [Roach1996] and [Roach1997].

Recall that the hypergeometric function is (initially) defined as

$${}_pF_q \left(\begin{matrix} a_1, \dots, a_p \\ b_1, \dots, b_q \end{matrix} \middle| z \right) = \sum_{n=0}^{\infty} \frac{(a_1)_n \cdots (a_p)_n}{(b_1)_n \cdots (b_q)_n} \frac{z^n}{n!}.$$

It turns out that there are certain differential operators that can change the a_p and p_q parameters by integers. If a sequence of such operators is known that converts the set of indices a_r^0 and b_s^0 into a_p and b_q , then we shall say the pair a_p, b_q is reachable from a_r^0, b_s^0 . Our general strategy is thus as follows: given a set a_p, b_q of parameters, try to look up an origin a_r^0, b_s^0 for which we know an expression, and then apply the sequence of differential operators to the known expression to find an expression for the Hypergeometric function we are interested in.

Notation

In the following, the symbol a will always denote a numerator parameter and the symbol b will always denote a denominator parameter. The subscripts p, q, r, s denote vectors of that length, so e.g. a_p denotes a vector of p numerator parameters. The subscripts i and j denote “running indices”, so they should usually be used in conjunction with a “for all i ”. E.g. $a_i < 4$ for all i . Uppercase subscripts I and J denote a chosen, fixed index. So for example $a_I > 0$ is true if the inequality holds for the one index I we are currently interested in.

Incrementing and decrementing indices

Suppose $a_i \neq 0$. Set $A(a_i) = \frac{z}{a_i} \frac{d}{dz} + 1$. It is then easy to show that $A(a_i) {}_pF_q \left(\begin{matrix} a_p \\ b_q \end{matrix} \middle| z \right) = {}_pF_q \left(\begin{matrix} a_p + e_i \\ b_q \end{matrix} \middle| z \right)$, where e_i is the i -th unit vector. Similarly for $b_j \neq 1$ we set $B(b_j) = \frac{z}{b_j - 1} \frac{d}{dz} + 1$ and find $B(b_j) {}_pF_q \left(\begin{matrix} a_p \\ b_q \end{matrix} \middle| z \right) = {}_pF_q \left(\begin{matrix} a_p \\ b_q - e_j \end{matrix} \middle| z \right)$. Thus we can increment upper and decrement lower indices at will, as long as we don’t go through zero. The $A(a_i)$ and $B(b_j)$ are called shift operators.

It is also easy to show that $\frac{d}{dz} {}_pF_q \left(\begin{matrix} a_p \\ b_q \end{matrix} \middle| z \right) = \frac{a_1 \cdots a_p}{b_1 \cdots b_q} {}_pF_q \left(\begin{matrix} a_p + 1 \\ b_q + 1 \end{matrix} \middle| z \right)$, where $a_p + 1$ is the vector $a_1 + 1, a_2 + 1, \dots$ and similarly for $b_q + 1$. Combining this with the shift operators, we arrive at one form of the Hypergeometric differential equation: $\left[\frac{d}{dz} \prod_{j=1}^q B(b_j) - \frac{a_1 \cdots a_p}{(b_1 - 1) \cdots (b_q - 1)} \prod_{i=1}^p A(a_i) \right] {}_pF_q \left(\begin{matrix} a_p \\ b_q \end{matrix} \middle| z \right) = 0$. This holds if all shift operators are defined, i.e. if no $a_i = 0$ and no $b_j = 1$. Clearing denominators and multiplying through by z we arrive at the following equation: $\left[z \frac{d}{dz} \prod_{j=1}^q \left(z \frac{d}{dz} + b_j - 1 \right) - z \prod_{i=1}^p \left(z \frac{d}{dz} + a_i \right) \right] {}_pF_q \left(\begin{matrix} a_p \\ b_q \end{matrix} \middle| z \right) = 0$. Even though our derivation does not show it, it can be checked that this equation holds whenever the ${}_pF_q$ is defined.

Notice that, under suitable conditions on a_I, b_J , each of the operators $A(a_i)$, $B(b_j)$ and $z \frac{d}{dz}$ can be expressed in terms of $A(a_I)$ or $B(b_J)$. Our next aim is to write the Hypergeometric

differential equation as follows: $[XA(a_I) - r]_p F_q \left(\begin{smallmatrix} a_p \\ b_q \end{smallmatrix} \middle| z \right) = 0$, for some operator X and some constant r to be determined. If $r \neq 0$, then we can write this as $\frac{-1}{r} X_p F_q \left(\begin{smallmatrix} a_p + e_I \\ b_q \end{smallmatrix} \middle| z \right) = {}_p F_q \left(\begin{smallmatrix} a_p \\ b_q \end{smallmatrix} \middle| z \right)$, and so $\frac{-1}{r} X$ undoes the shifting of $A(a_I)$, whence it will be called an inverse-shift operator.

Now $A(a_I)$ exists if $a_I \neq 0$, and then $z \frac{d}{dz} = a_I A(a_I) - a_I$. Observe also that all the operators $A(a_i)$, $B(b_j)$ and $z \frac{d}{dz}$ commute. We have $\prod_{i=1}^p \left(z \frac{d}{dz} + a_i \right) = \left(\prod_{i=1, i \neq I}^p \left(z \frac{d}{dz} + a_i \right) \right) a_I A(a_I)$, so this gives us the first half of X . The other half does not have such a nice expression. We find $z \frac{d}{dz} \prod_{j=1}^q \left(z \frac{d}{dz} + b_j - 1 \right) = (a_I A(a_I) - a_I) \prod_{j=1}^q (a_I A(a_I) - a_I + b_j - 1)$. Since the first half had no constant term, we infer $r = -a_I \prod_{j=1}^q (b_j - 1 - a_I)$.

This tells us under which conditions we can “un-shift” $A(a_I)$, namely when $a_I \neq 0$ and $r \neq 0$. Substituting $a_I - 1$ for a_I then tells us under what conditions we can decrement the index a_I . Doing a similar analysis for $B(a_J)$, we arrive at the following rules:

- An index a_I can be decremented if $a_I \neq 1$ and $a_I \neq b_j$ for all b_j .
- An index b_J can be incremented if $b_J \neq -1$ and $b_J \neq a_i$ for all a_i .

Combined with the conditions (stated above) for the existence of shift operators, we have thus established the rules of the game!

Reduction of Order

Notice that, quite trivially, if $a_I = b_J$, we have ${}_p F_q \left(\begin{smallmatrix} a_p \\ b_q \end{smallmatrix} \middle| z \right) = {}_{p-1} F_{q-1} \left(\begin{smallmatrix} a_p^* \\ b_q^* \end{smallmatrix} \middle| z \right)$, where a_p^* means a_p with a_I omitted, and similarly for b_q^* . We call this reduction of order.

In fact, we can do even better. If $a_I - b_J \in \mathbb{Z}_{>0}$, then it is easy to see that $\frac{(a_I)_n}{(b_J)_n}$ is actually a polynomial in n . It is also easy to see that $(z \frac{d}{dz})^k z^n = n^k z^n$. Combining these two remarks we find:

If $a_I - b_J \in \mathbb{Z}_{>0}$, then there exists a polynomial $p(n) = p_0 + p_1 n + \dots$ (of degree $a_I - b_J$) such that $\frac{(a_I)_n}{(b_J)_n} = p(n)$ and ${}_p F_q \left(\begin{smallmatrix} a_p \\ b_q \end{smallmatrix} \middle| z \right) = \left(p_0 + p_1 z \frac{d}{dz} + p_2 \left(z \frac{d}{dz} \right)^2 + \dots \right) {}_{p-1} F_{q-1} \left(\begin{smallmatrix} a_p^* \\ b_q^* \end{smallmatrix} \middle| z \right)$.

Thus any set of parameters a_p, b_q is reachable from a set of parameters c_r, d_s where $c_i - d_j \in \mathbb{Z}$ implies $c_i < d_j$. Such a set of parameters c_r, d_s is called suitable. Our database of known formulae should only contain suitable origins. The reasons are twofold: firstly, working from suitable origins is easier, and secondly, a formula for a non-suitable origin can be deduced from a lower order formula, and we should put this one into the database instead.

Moving Around in the Parameter Space

It remains to investigate the following question: suppose a_p, b_q and a_p^0, b_q^0 are both suitable, and also $a_i - a_i^0 \in \mathbb{Z}$, $b_j - b_j^0 \in \mathbb{Z}$. When is a_p, b_q reachable from a_p^0, b_q^0 ? It is clear that we can treat all parameters independently that are incongruent mod 1. So assume that a_i and b_j are congruent to r mod 1, for all i and j . The same then follows for a_i^0 and b_j^0 .

If $r \neq 0$, then any such a_p, b_q is reachable from any a_p^0, b_q^0 . To see this notice that there exist constants c, c^0 , congruent mod 1, such that $a_i < c < b_j$ for all i and j , and similarly $a_i^0 < c^0 < b_j^0$. If $n = c - c^0 > 0$ then we first inverse-shift all the b_j^0 n times up, and then similarly shift up all the a_i^0 n times. If $n < 0$ then we first inverse-shift down the a_i^0 and then shift down the b_j^0 . This reduces to the case $c = c^0$. But evidently we can now shift or inverse-shift around the a_i^0

arbitrarily so long as we keep them less than c , and similarly for the b_j^0 so long as we keep them bigger than c . Thus a_p, b_q is reachable from a_p^0, b_q^0 .

If $r = 0$ then the problem is slightly more involved. WLOG no parameter is zero. We now have one additional complication: no parameter can ever move through zero. Hence a_p, b_q is reachable from a_p^0, b_q^0 if and only if the number of $a_i < 0$ equals the number of $a_i^0 < 0$, and similarly for the b_i and b_i^0 . But in a suitable set of parameters, all $b_j > 0$! This is because the Hypergeometric function is undefined if one of the b_j is a non-positive integer and all a_i are smaller than the b_j . Hence the number of $b_j \leq 0$ is always zero.

We can thus associate to every suitable set of parameters a_p, b_q , where no $a_i = 0$, the following invariants:

- For every $r \in [0, 1)$ the number α_r of parameters $a_i \equiv r \pmod{1}$, and similarly the number β_r of parameters $b_i \equiv r \pmod{1}$.
- The number γ of integers a_i with $a_i < 0$.

The above reasoning shows that a_p, b_q is reachable from a_p^0, b_q^0 if and only if the invariants $\alpha_r, \beta_r, \gamma$ all agree. Thus in particular “being reachable from” is a symmetric relation on suitable parameters without zeros.

Applying the Operators

If all goes well then for a given set of parameters we find an origin in our database for which we have a nice formula. We now have to apply (potentially) many differential operators to it. If we do this blindly then the result will be very messy. This is because with Hypergeometric type functions, the derivative is usually expressed as a sum of two contiguous functions. Hence if we compute N derivatives, then the answer will involve $2N$ contiguous functions! This is clearly undesirable. In fact we know from the Hypergeometric differential equation that we need at most $\max(p, q + 1)$ contiguous functions to express all derivatives.

Hence instead of differentiating blindly, we will work with a $\mathbb{C}(z)$ -module basis: for an origin a_r^0, b_s^0 we either store (for particularly pretty answers) or compute a set of N functions (typically $N = \max(r, s + 1)$) with the property that the derivative of any of them is a $\mathbb{C}(z)$ -linear combination of them. In formulae, we store a vector B of N functions, a matrix M and a vector C (the latter two with entries in $\mathbb{C}(z)$), with the following properties:

- ${}_rF_s \left(\begin{smallmatrix} a_r^0 \\ b_s^0 \end{smallmatrix} \middle| z \right) = CB$
- $z \frac{d}{dz} B = MB$.

Then we can compute as many derivatives as we want and we will always end up with $\mathbb{C}(z)$ -linear combination of at most N special functions.

As hinted above, B , M and C can either all be stored (for particularly pretty answers) or computed from a single ${}_pF_q$ formula.

Loose Ends

This describes the bulk of the hypergeometric function algorithm. There are a few further tricks, described in the `hyperexpand.py` source file. The extension to Meijer G -functions is also described there.

Meijer G -Functions of Finite Confluence

Slater's theorem essentially evaluates a G -function as a sum of residues. If all poles are simple, the resulting series can be recognised as hypergeometric series. Thus a G -function can be evaluated as a sum of Hypergeometric functions.

If the poles are not simple, the resulting series are not hypergeometric. This is known as the "confluent" or "logarithmic" case (the latter because the resulting series tend to contain logarithms). The answer depends in a complicated way on the multiplicities of various poles, and there is no accepted notation for representing it (as far as I know). However if there are only finitely many multiple poles, we can evaluate the G function as a sum of hypergeometric functions, plus finitely many extra terms. I could not find any good reference for this, which is why I work it out here.

Recall the general setup. We define

$$G(z) = \frac{1}{2\pi i} \int_L \frac{\prod_{j=1}^m \Gamma(b_j - s) \prod_{j=1}^n \Gamma(1 - a_j + s)}{\prod_{j=m+1}^q \Gamma(1 - b_j + s) \prod_{j=n+1}^p \Gamma(a_j - s)} z^s ds,$$

where L is a contour starting and ending at $+\infty$, enclosing all of the poles of $\Gamma(b_j - s)$ for $j = 1, \dots, n$ once in the negative direction, and no other poles. Also the integral is assumed absolutely convergent.

In what follows, for any complex numbers a, b , we write $a \equiv b \pmod{1}$ if and only if there exists an integer k such that $a - b = k$. Thus there are double poles iff $a_i \equiv a_j \pmod{1}$ for some $i \neq j \leq n$.

We now assume that whenever $b_j \equiv a_i \pmod{1}$ for $i \leq m$, $j > n$ then $b_j < a_i$. This means that no quotient of the relevant gamma functions is a polynomial, and can always be achieved by "reduction of order". Fix a complex number c such that $\{b_i | b_i \equiv c \pmod{1}, i \leq m\}$ is not empty. Enumerate this set as $b, b + k_1, \dots, b + k_u$, with k_i non-negative integers. Enumerate similarly $\{a_j | a_j \equiv c \pmod{1}, j > n\}$ as $b + l_1, \dots, b + l_v$. Then $l_i > k_j$ for all i, j . For finite confluence, we need to assume $v \geq u$ for all such c .

Let $c_1, \dots, c_w$ be distinct $\pmod{1}$ and exhaust the congruence classes of the b_i . I claim

$$G(z) = - \sum_{j=1}^w (F_j(z) + R_j(z)),$$

where $F_j(z)$ is a hypergeometric function and $R_j(z)$ is a finite sum, both to be specified later. Indeed corresponding to every c_j there is a sequence of poles, at mostly finitely many of them multiple poles. This is where the j -th term comes from.

Hence fix again c , enumerate the relevant b_i as $b, b + k_1, \dots, b + k_u$. We will look at the a_j corresponding to $a + l_1, \dots, a + l_u$. The other a_i are not treated specially. The corresponding gamma functions have poles at (potentially) $s = b + r$ for $r = 0, 1, \dots$. For $r \geq l_u$, pole of the integrand is simple. We thus set

$$R(z) = \sum_{r=0}^{l_u-1} \text{res}_{s=b+r}.$$

We finally need to investigate the other poles. Set $r = l_u + t$, $t \geq 0$. A computation shows

$$\frac{\Gamma(k_i - l_u - t)}{\Gamma(l_i - l_u - t)} = \frac{1}{(k_i - l_u - t)_{l_i - k_i}} = \frac{(-1)^{\delta_i}}{(l_u - l_i + 1)_{\delta_i}} \frac{(l_u - l_i + 1)_t}{(l_u - k_i + 1)_t},$$

where $\delta_i = l_i - k_i$.

Also

$$\Gamma(b_j - l_u - b - t) = \frac{\Gamma(b_j - l_u - b)}{(-1)^t (l_u + b + 1 - b_j)_t},$$

$$\Gamma(1 - a_j + l_u + b + t) = \Gamma(1 - a_j + l_u + b) \Gamma(1 - a_j + l_u + b)_t$$

and

$$res_{s=b+l_u+t} \Gamma(b - s) = -\frac{(-1)^{l_u+t}}{(l_u + t)!} = -\frac{(-1)^{l_u}}{l_u!} \frac{(-1)^t}{(l_u + 1)_t}.$$

Hence

$$res_{s=b+l_u+t} = -z^{b+l_u} \frac{(-1)^{l_u}}{l_u!} \prod_{i=1}^u \frac{(-1)^{\delta_i}}{(l_u - k_i + 1)_{\delta_i}} \frac{\prod_{j=1}^n \Gamma(1 - a_j + l_u + b) \prod_{j=1}^m \Gamma(b_j - l_u - b)^*}{\prod_{j=n+1}^p \Gamma(a_j - l_u - b)^* \prod_{j=m+1}^q \Gamma(1 - b_j + l_u + b)} \\ \times z^t \frac{(-1)^t}{(l_u + 1)_t} \prod_{i=1}^u \frac{(l_u - l_i + 1)_t}{(l_u - k_i + 1)_t} \frac{\prod_{j=1}^n (1 - a_j + l_u + b)_t \prod_{j=n+1}^p (-1)^t (l_u + b + 1 - a_j)_t^*}{\prod_{j=1}^m (-1)^t (l_u + b + 1 - b_j)_t^* \prod_{j=m+1}^q (1 - b_j + l_u + b)_t^*},$$

where the $*$ means to omit the terms we treated specially.

We thus arrive at

$$F(z) = C \times {}_{p+1}F_q \left(\begin{matrix} 1, (1 + l_u - l_i), (1 + l_u + b - a_i)^* \\ 1 + l_u, (1 + l_u - k_i), (1 + l_u + b - b_i)^* \end{matrix} \middle| (-1)^{p-m-n} z \right),$$

where C designates the factor in the residue independent of t . (This result can also be written in slightly simpler form by converting all the l_u etc back to a_* - b_* , but doing so is going to require more notation still and is not helpful for computation.)

Extending The Hypergeometric Tables

Adding new formulae to the tables is straightforward. At the top of the file `sympy/simplify/hyperexpand.py`, there is a function called `add_formulae()`. Nested in it are defined two helpers, `add(ap, bq, res)` and `addb(ap, bq, B, C, M)`, as well as dummies `a`, `b`, `c`, and `z`.

The first step in adding a new formula is by using `add(ap, bq, res)`. This declares `hyper(ap, bq, z) == res`. Here `ap` and `bq` may use the dummies `a`, `b`, and `c` as free symbols. For example the well-known formula $\sum_0^\infty \frac{(-a)_n z^n}{n!} = (1 - z)^a$ is declared by the following line: `add((-a, ), (), (1-z)**a)`.

From the information provided, the matrices B , C and M will be computed, and the formula is now available when expanding hypergeometric functions. Next the test file `sympy/simplify/tests/test_hyperexpand.py` should be run, in particular the test `test_formulae()`. This will test the newly added formula numerically. If it fails, there is (presumably) a typo in what was entered.

Since all newly-added formulae are probably relatively complicated, chances are that the automatically computed basis is rather suboptimal (there is no good way of testing this, other than observing very messy output). In this case the matrices B , C and M should be computed by hand. Then the helper `addb` can be used to declare a hypergeometric formula with hand-computed basis.

An example

Because this explanation so far might be very theoretical and difficult to understand, we walk through an explicit example now. We take the Fresnel function $C(z)$ which obeys the following hypergeometric representation:

$$C(z) = z \cdot {}_1F_2 \left(\frac{1}{2} \middle| \frac{5}{4} - \frac{\pi^2 z^4}{16} \right).$$

First we try to add this formula to the lookup table by using the (simpler) function `add(ap, bq, res)`. The first two arguments are simply the lists containing the parameter sets of ${}_1F_2$. The `res` argument is a little bit more complicated. We only know $C(z)$ in terms of ${}_1F_2(\dots | f(z))$ with f a function of z , in our case

$$f(z) = -\frac{\pi^2 z^4}{16}.$$

What we need is a formula where the hypergeometric function has only z as argument ${}_1F_2(\dots | z)$. We introduce the new complex symbol w and search for a function $g(w)$ such that

$$f(g(w)) = w$$

holds. Then we can replace every z in $C(z)$ by $g(w)$. In the case of our example the function g could look like

$$g(w) = \frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) w^{\frac{1}{4}}.$$

We get these functions mainly by guessing and testing the result. Hence we proceed by computing $f(g(w))$ (and simplifying naively)

$$\begin{aligned} f(g(w)) &= -\frac{\pi^2 g(w)^4}{16} \\ &= -\frac{\pi^2 g \left(\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) w^{\frac{1}{4}} \right)^4}{16} \\ &= -\frac{\pi^2 \frac{2^4}{\sqrt{\pi}^4} \exp\left(\frac{i\pi}{4}\right)^4 w^{\frac{1}{4}^4}}{16} \\ &= -\exp(i\pi) w \\ &= w \end{aligned}$$

and indeed get back w . (In case of branched functions we have to be aware of branch cuts. In that case we take w to be a positive real number and check the formula. If what we have found works for positive w , then just replace `exp` (page 411) inside any branched function by `exp_polar` (page 414) and what we get is right for *all* w .) Hence we can write the formula as

$$C(g(w)) = g(w) \cdot {}_1F_2 \left(\frac{1}{2} \middle| \frac{5}{4} \middle| w \right).$$

and trivially

$${}_1F_2 \left(\frac{1}{2} \middle| \frac{5}{4} \middle| w \right) = \frac{C(g(w))}{g(w)} = \frac{C \left(\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) w^{\frac{1}{4}} \right)}{\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) w^{\frac{1}{4}}}$$

which is exactly what is needed for the third parameter, `res`, in `add`. Finally, the whole function call to add this rule to the table looks like:


```
add([S(1)/4],
    [S(1)/2, S(5)/4],
    fresnelc(exp(pi*I/4)*root(z,4)*2/sqrt(pi)) / (exp(pi*I/4)*root(z,4)*2/
    ↪ sqrt(pi))
    )
```

Using this rule we will find that it works but the results are not really nice in terms of simplicity and number of special function instances included. We can obtain much better results by adding the formula to the lookup table in another way. For this we use the (more complicated) function `addb(ap, bq, B, C, M)`. The first two arguments are again the lists containing the parameter sets of ${}_1F_2$. The remaining three are the matrices mentioned earlier on this page.

We know that the $n = \max(p, q + 1)$ -th derivative can be expressed as a linear combination of lower order derivatives. The matrix B contains the basis $\{B_0, B_1, \dots\}$ and is of shape $n \times 1$. The best way to get B_i is to take the first $n = \max(p, q + 1)$ derivatives of the expression for ${}_pF_q$ and take out useful pieces. In our case we find that $n = \max(1, 2 + 1) = 3$. For computing the derivatives, we have to use the operator $z \frac{d}{dz}$. The first basis element B_0 is set to the expression for ${}_1F_2$ from above:

$$B_0 = \frac{\sqrt{\pi} \exp\left(-\frac{i\pi}{4}\right) C\left(\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) z^{\frac{1}{4}}\right)}{2z^{\frac{1}{4}}}$$

Next we compute $z \frac{d}{dz} B_0$. For this we can directly use SymPy!

```
>>> from sympy import Symbol, sqrt, exp, I, pi, fresnelc, root, diff, expand
>>> z = Symbol("z")
>>> B0 = sqrt(pi)*exp(-I*pi/4)*fresnelc(2*root(z,4)*exp(I*pi/4)/sqrt(pi))/\
...     (2*root(z,4))
>>> z * diff(B0, z)
z*(cosh(2*sqrt(z))/(4*z) - sqrt(pi)*exp(-I*pi/4)*fresnelc(2*z**(1/4)*exp(I*pi/
↪ 4)/sqrt(pi))/(8*z**(5/4)))
>>> expand(_)
cosh(2*sqrt(z))/4 - sqrt(pi)*exp(-I*pi/4)*fresnelc(2*z**(1/4)*exp(I*pi/4)/
↪ sqrt(pi))/(8*z**(1/4))
```

Formatting this result nicely we obtain

$$B'_1 = -\frac{1}{4} \frac{\sqrt{\pi} \exp\left(-\frac{i\pi}{4}\right) C\left(\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) z^{\frac{1}{4}}\right)}{2z^{\frac{1}{4}}} + \frac{1}{4} \cosh(2\sqrt{z})$$

Computing the second derivative we find

```
>>> from sympy import (Symbol, cosh, sqrt, pi, exp, I, fresnelc, root,
...                     diff, expand)
>>> z = Symbol("z")
>>> B1prime = cosh(2*sqrt(z))/4 - sqrt(pi)*exp(-I*pi/4)*\
...           fresnelc(2*root(z,4)*exp(I*pi/4)/sqrt(pi))/(8*root(z,4))
>>> z * diff(B1prime, z)
z*(-cosh(2*sqrt(z))/(16*z) + sinh(2*sqrt(z))/(4*sqrt(z)) + sqrt(pi)*exp(-I*pi/
↪ 4)*fresnelc(2*z**(1/4)*exp(I*pi/4)/sqrt(pi))/(32*z**(5/4)))
>>> expand(_)
sqrt(z)*sinh(2*sqrt(z))/4 - cosh(2*sqrt(z))/16 + sqrt(pi)*exp(-I*pi/
↪ 4)*fresnelc(2*z**(1/4)*exp(I*pi/4)/sqrt(pi))/(32*z**(1/4))
```

which can be printed as

$$B'_2 = \frac{1}{16} \frac{\sqrt{\pi} \exp\left(-\frac{i\pi}{4}\right) C\left(\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) z^{\frac{1}{4}}\right)}{2z^{\frac{1}{4}}} - \frac{1}{16} \cosh(2\sqrt{z}) + \frac{1}{4} \sinh(2\sqrt{z})\sqrt{z}$$

We see the common pattern and can collect the pieces. Hence it makes sense to choose B_1 and B_2 as follows

$$B = \begin{pmatrix} B_0 \\ B_1 \\ B_2 \end{pmatrix} = \begin{pmatrix} \frac{\sqrt{\pi} \exp\left(-\frac{i\pi}{4}\right) C\left(\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) z^{\frac{1}{4}}\right)}{2z^{\frac{1}{4}}} \\ \cosh(2\sqrt{z}) \\ \sinh(2\sqrt{z})\sqrt{z} \end{pmatrix}$$

(This is in contrast to the basis $B = (B_0, B'_1, B'_2)$ that would have been computed automatically if we used just `add(ap, bq, res)`.)

Because it must hold that ${}_pF_q(\cdots|z) = CB$ the entries of C are obviously

$$C = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

Finally we have to compute the entries of the 3×3 matrix M such that $z \frac{d}{dz} B = MB$ holds. This is easy. We already computed the first part $z \frac{d}{dz} B_0$ above. This gives us the first row of M . For the second row we have:

```
>>> from sympy import Symbol, cosh, sqrt, diff
>>> z = Symbol("z")
>>> B1 = cosh(2*sqrt(z))
>>> z * diff(B1, z)
sqrt(z)*sinh(2*sqrt(z))
```

and for the third one

```
>>> from sympy import Symbol, sinh, sqrt, expand, diff
>>> z = Symbol("z")
>>> B2 = sinh(2*sqrt(z))*sqrt(z)
>>> expand(z * diff(B2, z))
sqrt(z)*sinh(2*sqrt(z))/2 + z*cosh(2*sqrt(z))
```

Now we have computed the entries of this matrix to be

$$M = \begin{pmatrix} -\frac{1}{4} & \frac{1}{4} & 0 \\ 0 & 0 & 1 \\ 0 & z & \frac{1}{2} \end{pmatrix}$$

Note that the entries of C and M should typically be rational functions in z , with rational coefficients. This is all we need to do in order to add a new formula to the lookup table for `hyperexpand`.

Implemented Hypergeometric Formulae

A vital part of the algorithm is a relatively large table of hypergeometric function representations. The following automatically generated list contains all the representations implemented in SymPy (of course many more are derived from them). These formulae are mostly taken from [Luke1969] and [Prudnikov1990]. They are all tested numerically.

$$\begin{aligned}
{}_0F_0(|z) &= e^z \\
{}_1F_0(a|z) &= (1-z)^{-a} \\
{}_2F_1\left(a, a - \frac{1}{2} \middle| 2a; z\right) &= 2^{2a-1} (\sqrt{1-z} + 1)^{1-2a} \\
{}_2F_1\left(1, 1 \middle| 2; z\right) &= -\frac{\log(1-z)}{z} \\
{}_2F_1\left(\frac{1}{2}, 1 \middle| \frac{3}{2}; z\right) &= \frac{\operatorname{atanh}(\sqrt{z})}{\sqrt{z}} \\
{}_2F_1\left(\frac{1}{2}, \frac{1}{2} \middle| \frac{3}{2}; z\right) &= \frac{\operatorname{asin}(\sqrt{z})}{\sqrt{z}} \\
{}_2F_1\left(a, a + \frac{1}{2} \middle| \frac{1}{2}; z\right) &= \frac{(\sqrt{z} + 1)^{-2a}}{2} + \frac{(1 - \sqrt{z})^{-2a}}{2} \\
{}_2F_1\left(a, -a \middle| \frac{1}{2}; z\right) &= \cos(2a \operatorname{asin}(\sqrt{z})) \\
{}_2F_1\left(1, 1 \middle| \frac{3}{2}; z\right) &= \frac{\operatorname{asin}(\sqrt{z})}{\sqrt{z}\sqrt{1-z}} \\
{}_2F_1\left(\frac{1}{2}, \frac{1}{2} \middle| 1; z\right) &= \frac{2K(z)}{\pi} \\
{}_2F_1\left(-\frac{1}{2}, \frac{1}{2} \middle| 1; z\right) &= \frac{2E(z)}{\pi} \\
{}_3F_2\left(-\frac{1}{2}, 1, 1 \middle| \frac{1}{2}, 2; z\right) &= -\frac{2\sqrt{z} \operatorname{atanh}(\sqrt{z})}{3} + \frac{2}{3} - \frac{\log(1-z)}{3z} \\
{}_3F_2\left(-\frac{1}{2}, 1, 1 \middle| 2, 2; z\right) &= \left(\frac{4}{9} - \frac{16}{9z}\right) \sqrt{1-z} + \frac{4 \log\left(\frac{\sqrt{1-z}}{2} + \frac{1}{2}\right)}{3z} + \frac{16}{9z} \\
{}_1F_1\left(1 \middle| b; z\right) &= z^{1-b} (b-1) e^z \gamma(b-1, z) \\
{}_1F_1\left(a \middle| 2a; z\right) &= 4^{a-\frac{1}{2}} z^{\frac{1}{2}-a} e^{\frac{z}{2}} I_{a-\frac{1}{2}}\left(\frac{z}{2}\right) \Gamma\left(a + \frac{1}{2}\right) \\
{}_1F_1\left(a \middle| a+1; z\right) &= a (ze^{i\pi})^{-a} \gamma(a, ze^{i\pi}) \\
{}_1F_1\left(-\frac{1}{2} \middle| \frac{1}{2}; z\right) &= \sqrt{z} i \sqrt{\pi} \operatorname{erf}(\sqrt{z} i) + e^z \\
{}_1F_2\left(\frac{1}{4}, \frac{5}{4} \middle| \frac{3}{4}, \frac{7}{4}; z\right) &= \frac{\sqrt{\pi} \left(i \sinh(2\sqrt{z}) S\left(\frac{2\sqrt[4]{z} e^{\frac{i\pi}{4}}}{\sqrt{\pi}}\right) + \cosh(2\sqrt{z}) C\left(\frac{2\sqrt[4]{z} e^{\frac{i\pi}{4}}}{\sqrt{\pi}}\right) \right)}{2\sqrt[4]{z}} e^{-\frac{i\pi}{4}}
\end{aligned}$$

$$\begin{aligned}
 {}_2F_2\left(\frac{1}{2}, a \middle| \frac{3}{2}, a+1 \middle| z\right) &= -\frac{ai\sqrt{\pi}\sqrt{\frac{1}{z}}\operatorname{erf}(\sqrt{z}i)}{2a-1} - \frac{a(z e^{i\pi})^{-a}\gamma(a, z e^{i\pi})}{2a-1} \\
 {}_2F_2\left(1, 1 \middle| 2, 2 \middle| z\right) &= \frac{-\log(z) + \operatorname{Ei}(z)}{z} - \frac{\gamma}{z} \\
 {}_0F_1\left(\frac{1}{2} \middle| z\right) &= \cosh(2\sqrt{z}) \\
 {}_0F_1\left(b \middle| z\right) &= z^{\frac{1}{2}-\frac{b}{2}} I_{b-1}(2\sqrt{z}) \Gamma(b) \\
 {}_0F_3\left(\frac{1}{2}, a, a+\frac{1}{2} \middle| z\right) &= 2^{-2a} z^{\frac{1}{4}-\frac{a}{2}} (I_{2a-1}(4\sqrt[4]{z}) + J_{2a-1}(4\sqrt[4]{z})) \Gamma(2a) \\
 {}_0F_3\left(a, a+\frac{1}{2}, 2a \middle| z\right) &= \left(2\sqrt{z} e^{\frac{i\pi}{2}}\right)^{1-2a} I_{2a-1}\left(2\sqrt{2}\sqrt[4]{z} e^{\frac{i\pi}{4}}\right) J_{2a-1}\left(2\sqrt{2}\sqrt[4]{z} e^{\frac{i\pi}{4}}\right) \Gamma^2(2a) \\
 {}_1F_2\left(a-\frac{1}{2}, 2a \middle| z\right) &= 2 \cdot 4^{a-1} z^{1-a} I_{a-\frac{3}{2}}(\sqrt{z}) I_{a-\frac{1}{2}}(\sqrt{z}) \Gamma\left(a-\frac{1}{2}\right) \Gamma\left(a+\frac{1}{2}\right) - 4^{a-\frac{1}{2}} z^{\frac{1}{2}-a} I_{a-\frac{1}{2}}^2(\sqrt{z}) \Gamma^2\left(a+\frac{1}{2}\right) \\
 {}_1F_2\left(b, 2-b \middle| z\right) &= \frac{\pi(1-b) I_{1-b}(\sqrt{z}) I_{b-1}(\sqrt{z})}{\sin(b\pi)} \\
 {}_1F_2\left(\frac{1}{2}, \frac{3}{2} \middle| z\right) &= \frac{\operatorname{Shi}(2\sqrt{z})}{2\sqrt{z}} \\
 {}_1F_2\left(\frac{3}{2}, \frac{7}{4} \middle| z\right) &= \frac{3\sqrt{\pi} e^{-\frac{3i\pi}{4}} S\left(\frac{2\sqrt[4]{z} e^{\frac{i\pi}{4}}}{\sqrt{\pi}}\right)}{4z^{\frac{3}{4}}} \\
 {}_1F_2\left(\frac{1}{2}, \frac{5}{4} \middle| z\right) &= \frac{\sqrt{\pi} e^{-\frac{i\pi}{4}} C\left(\frac{2\sqrt[4]{z} e^{\frac{i\pi}{4}}}{\sqrt{\pi}}\right)}{2\sqrt[4]{z}} \\
 {}_2F_3\left(\frac{a}{2}, a+\frac{1}{2}, 2a-b+1 \middle| z\right) &= \left(\frac{\sqrt{z}}{2}\right)^{1-2a} I_{2a-b}(\sqrt{z}) I_{b-1}(\sqrt{z}) \Gamma(b) \Gamma(2a-b+1) \\
 {}_2F_3\left(1, 1 \middle| 2, 2, \frac{3}{2} \middle| z\right) &= \frac{-\log(2\sqrt{z}) + \operatorname{Chi}(2\sqrt{z})}{z} - \frac{\gamma}{z} \\
 {}_3F_3\left(\frac{1}{2}, 1, a \middle| 2, 2, a+1 \middle| z\right) &= \frac{a(-z)^{-a}(\Gamma(a) - \Gamma(a, -z))}{(a-1)^2} + \frac{a(1-a)(\log(-z) + \operatorname{E}_1(-z) + \gamma)}{z(a^2-2a+1)} - \frac{ae^z}{z(a^2-2a+1)} + \frac{a}{z(a^2-2a+1)}
 \end{aligned}$$

References

Hongguang Fu's Trigonometric Simplification

Implementation of the trigsimp algorithm by Fu et al.

The idea behind the Fu algorithm is to use a sequence of rules that students learn during their pre-calculus courses. The rules are applied heuristically and it uses a greedy algorithm to apply multiple rules simultaneously and choose the result with the least leaf counts.

There are transform rules in which a single rule is applied to the expression tree. The following are just mnemonic in nature; see the docstrings for examples.

- `TR0()` (page 704) - simplify expression

- `TR1()` (page 704) - sec-csc to cos-sin
- `TR2()` (page 704) - tan-cot to sin-cos ratio
- `TR2i()` (page 704) - sin-cos ratio to tan
- `TR3()` (page 705) - angle canonicalization
- `TR4()` (page 705) - functions at special angles
- `TR5()` (page 705) - powers of sin to powers of cos
- `TR6()` (page 706) - powers of cos to powers of sin
- `TR7()` (page 706) - reduce cos power (increase angle)
- `TR8()` (page 706) - expand products of sin-cos to sums
- `TR9()` (page 707) - contract sums of sin-cos to products
- `TR10()` (page 707) - separate sin-cos arguments
- `TR10i()` (page 707) - collect sin-cos arguments
- `TR11()` (page 708) - reduce double angles
- `TR12()` (page 708) - separate tan arguments
- `TR12i()` (page 709) - collect tan arguments
- `TR13()` (page 709) - expand product of tan-cot
- `TRmorrie()` (page 709) - $\text{prod}(\cos(x^{2^i}), (i, 0, k - 1)) \rightarrow \sin(2^k x) / (2^k \sin(x))$
- `TR14()` (page 711) - factored powers of sin or cos to cos or sin power
- `TR15()` (page 711) - negative powers of sin to cot power
- `TR16()` (page 711) - negative powers of cos to tan power
- `TR22()` (page 712) - tan-cot powers to negative powers of sec-csc functions
- `TR111()` (page 711) - negative sin-cos-tan powers to csc-sec-cot

There are 4 combination transforms (CTR1 - CTR4) in which a sequence of transformations are applied and the simplest expression is selected from a few options.

Finally, there are the 2 rule lists (RL1 and RL2), which apply a sequence of transformations and combined transformations, and the fu algorithm itself, which applies rules and rule lists and selects the best expressions. There is also a function L which counts the number of trigonometric functions that appear in the expression.

Other than TR0, re-writing of expressions is not done by the transformations. e.g. TR10i finds pairs of terms in a sum that are in the form like $\cos(x)\cos(y) + \sin(x)\sin(y)$. Such expression are targeted in a bottom-up traversal of the expression, but no manipulation to make them appear is attempted. For example,

Set-up for examples below:

```
>>> from sympy.simplify.fu import fu, L, TR9, TR10i, TR11
>>> from sympy import factor, sin, cos, powsimp
>>> from sympy.abc import x, y, z, a
>>> from time import time
```

```
>>> eq = cos(x + y)/cos(x)
>>> TR10i(eq.expand(trig=True))
-sin(x)*sin(y)/cos(x) + cos(y)
```

If the expression is put in “normal” form (with a common denominator) then the transformation is successful:

```
>>> TR10i(_.normal())
cos(x + y)/cos(x)
```

TR11’s behavior is similar. It rewrites double angles as smaller angles but doesn’t do any simplification of the result.

```
>>> TR11(sin(2)**a*cos(1)**(-a), 1)
(2*sin(1)*cos(1))**a/cos(1)**a
>>> powsimp(_)
(2*sin(1))**a
```

The temptation is to try make these TR rules “smarter” but that should really be done at a higher level; the TR rules should try maintain the “do one thing well” principle. There is one exception, however. In TR10i and TR9 terms are recognized even when they are each multiplied by a common factor:

```
>>> fu(a*cos(x)*cos(y) + a*sin(x)*sin(y))
a*cos(x - y)
```

Factoring with `factor_terms` is used but it is “JIT”-like, being delayed until it is deemed necessary. Furthermore, if the factoring does not help with the simplification, it is not retained, so `a*cos(x)*cos(y) + a*sin(x)*sin(z)` does not become a factored (but unsimplified in the trigonometric sense) expression:

```
>>> fu(a*cos(x)*cos(y) + a*sin(x)*sin(z))
a*sin(x)*sin(z) + a*cos(x)*cos(y)
```

In some cases factoring might be a good idea, but the user is left to make that decision. For example:

```
>>> expr=((15*sin(2*x) + 19*sin(x + y) + 17*sin(x + z) + 19*cos(x - z) +
... 25)*(20*sin(2*x) + 15*sin(x + y) + sin(y + z) + 14*cos(x - z) +
... 14*cos(y - z))*(9*sin(2*y) + 12*sin(y + z) + 10*cos(x - y) + 2*cos(y -
... z) + 18)).expand(trig=True).expand()
```

In the expanded state, there are nearly 1000 trig functions:

```
>>> L(expr)
932
```

If the expression were factored first, this would take time but the resulting expression would be transformed very quickly:

```
>>> def clock(f, n=2):
...     t=time(); f(); return round(time()-t, n)
...
>>> clock(lambda: factor(expr))
```

(continues on next page)

(continued from previous page)

```
0.86
>>> clock(lambda: TR10i(expr), 3)
0.016
```

If the unexpanded expression is used, the transformation takes longer but not as long as it took to factor it and then transform it:

```
>>> clock(lambda: TR10i(expr), 2)
0.28
```

So neither expansion nor factoring is used in TR10i: if the expression is already factored (or partially factored) then expansion with `trig=True` would destroy what is already known and take longer; if the expression is expanded, factoring may take longer than simply applying the transformation itself.

Although the algorithms should be canonical, always giving the same result, they may not yield the best result. This, in general, is the nature of simplification where searching all possible transformation paths is very expensive. Here is a simple example. There are 6 terms in the following sum:

```
>>> expr = (sin(x)**2*cos(y)*cos(z) + sin(x)*sin(y)*cos(x)*cos(z) +
... sin(x)*sin(z)*cos(x)*cos(y) + sin(y)*sin(z)*cos(x)**2 + sin(y)*sin(z) +
... cos(y)*cos(z))
>>> args = expr.args
```

Serendipitously, `fu` gives the best result:

```
>>> fu(expr)
3*cos(y - z)/2 - cos(2*x + y + z)/2
```

But if different terms were combined, a less-optimal result might be obtained, requiring some additional work to get better simplification, but still less than optimal. The following shows an alternative form of `expr` that resists optimal simplification once a given step is taken since it leads to a dead end:

```
>>> TR9(-cos(x)**2*cos(y + z) + 3*cos(y - z)/2 +
...      cos(y + z)/2 + cos(-2*x + y + z)/4 - cos(2*x + y + z)/4)
sin(2*x)*sin(y + z)/2 - cos(x)**2*cos(y + z) + 3*cos(y - z)/2 + cos(y + z)/2
```

Here is a smaller expression that exhibits the same behavior:

```
>>> a = sin(x)*sin(z)*cos(x)*cos(y) + sin(x)*sin(y)*cos(x)*cos(z)
>>> TR10i(a)
sin(x)*sin(y + z)*cos(x)
>>> newa = _
>>> TR10i(expr - a) # this combines two more of the remaining terms
sin(x)**2*cos(y)*cos(z) + sin(y)*sin(z)*cos(x)**2 + cos(y - z)
>>> TR10i(_ + newa) == _ + newa # but now there is no more simplification
True
```

Without getting lucky or trying all possible pairings of arguments, the final result may be less than optimal and impossible to find without better heuristics or brute force trial of all possibilities.

Rules

`sympy.simplify.fu.TR0(rv)`

Simplification of rational polynomials, trying to simplify the expression, e.g. combine things like $3*x + 2*x$, etc....

`sympy.simplify.fu.TR1(rv)`

Replace sec, csc with $1/\cos$, $1/\sin$

Examples

```
>>> from sympy.simplify.fu import TR1, sec, csc
>>> from sympy.abc import x
>>> TR1(2*csc(x) + sec(x))
1/cos(x) + 2/sin(x)
```

`sympy.simplify.fu.TR2(rv)`

Replace tan and cot with $\sin/\cos$ and $\cos/\sin$

Examples

```
>>> from sympy.simplify.fu import TR2
>>> from sympy.abc import x
>>> from sympy import tan, cot, sin, cos
>>> TR2(tan(x))
sin(x)/cos(x)
>>> TR2(cot(x))
cos(x)/sin(x)
>>> TR2(tan(tan(x) - sin(x)/cos(x)))
0
```

`sympy.simplify.fu.TR2i(rv, half=False)`

Converts ratios involving sin and cos as follows::

$\sin(x)/\cos(x) \rightarrow \tan(x)$ $\sin(x)/(\cos(x) + 1) \rightarrow \tan(x/2)$ if `half=True`

Examples

```
>>> from sympy.simplify.fu import TR2i
>>> from sympy.abc import x, a
>>> from sympy import sin, cos
>>> TR2i(sin(x)/cos(x))
tan(x)
```

Powers of the numerator and denominator are also recognized

```
>>> TR2i(sin(x)**2/(cos(x) + 1)**2, half=True)
tan(x/2)**2
```


The transformation does not take place unless assumptions allow (i.e. the base must be positive or the exponent must be an integer for both numerator and denominator)

```
>>> TR2i(sin(x)**a/(cos(x) + 1)**a)
sin(x)**a/(cos(x) + 1)**a
```

`sympy.simplify.fu.TR3(rv)`

Induced formula: example $\sin(-a) = -\sin(a)$

Examples

```
>>> from sympy.simplify.fu import TR3
>>> from sympy.abc import x, y
>>> from sympy import pi
>>> from sympy import cos
>>> TR3(cos(y - x*(y - x)))
cos(x*(x - y) + y)
>>> cos(pi/2 + x)
-sin(x)
>>> cos(30*pi/2 + x)
-cos(x)
```

`sympy.simplify.fu.TR4(rv)`

Identify values of special angles.

A= 0 Pi/6 Pi/4 Pi/3 Pi/2

$\sin(a)$ 0 1/2 $\sqrt{2}/2$ $\sqrt{3}/2$ 1 $\cos(a)$ 1 $\sqrt{3}/2$ $\sqrt{2}/2$ 1/2 0 $\tan(a)$ 0 $\sqrt{3}/3$ 1 $\sqrt{3}$ -

Examples

```
>>> from sympy import pi
>>> from sympy import cos, sin, tan, cot
>>> for s in (0, pi/6, pi/4, pi/3, pi/2):
...     print('%s %s %s %s' % (cos(s), sin(s), tan(s), cot(s)))
...
1 0 0 zoo
sqrt(3)/2 1/2 sqrt(3)/3 sqrt(3)
sqrt(2)/2 sqrt(2)/2 1 1
1/2 sqrt(3)/2 sqrt(3) sqrt(3)/3
0 1 zoo 0
```

`sympy.simplify.fu.TR5(rv, max=4, pow=False)`

Replacement of $\sin^2$ with $1 - \cos(x)^2$.

See `_TR56` docstring for advanced use of `max` and `pow`.

Examples

```
>>> from sympy.simplify.fu import TR5
>>> from sympy.abc import x
>>> from sympy import sin
>>> TR5(sin(x)**2)
1 - cos(x)**2
>>> TR5(sin(x)**-2) # unchanged
sin(x)**(-2)
>>> TR5(sin(x)**4)
(1 - cos(x)**2)**2
```

`sympy.simplify.fu.TR6(rv, max=4, pow=False)`

Replacement of $\cos^2$ with $1 - \sin^2$.

See `_TR56` docstring for advanced use of `max` and `pow`.

Examples

```
>>> from sympy.simplify.fu import TR6
>>> from sympy.abc import x
>>> from sympy import cos
>>> TR6(cos(x)**2)
1 - sin(x)**2
>>> TR6(cos(x)**-2) #unchanged
cos(x)**(-2)
>>> TR6(cos(x)**4)
(1 - sin(x)**2)**2
```

`sympy.simplify.fu.TR7(rv)`

Lowering the degree of $\cos(x)^2$.

Examples

```
>>> from sympy.simplify.fu import TR7
>>> from sympy.abc import x
>>> from sympy import cos
>>> TR7(cos(x)**2)
cos(2*x)/2 + 1/2
>>> TR7(cos(x)**2 + 1)
cos(2*x)/2 + 3/2
```

`sympy.simplify.fu.TR8(rv, first=True)`

Converting products of $\cos$ and/or $\sin$ to a sum or difference of $\cos$ and or $\sin$ terms.

Examples

```
>>> from sympy.simplify.fu import TR8
>>> from sympy import cos, sin
>>> TR8(cos(2)*cos(3))
cos(5)/2 + cos(1)/2
>>> TR8(cos(2)*sin(3))
sin(5)/2 + sin(1)/2
>>> TR8(sin(2)*sin(3))
-cos(5)/2 + cos(1)/2
```

`sympy.simplify.fu.TR9(rv)`

Sum of cos or sin terms as a product of cos or sin.

Examples

```
>>> from sympy.simplify.fu import TR9
>>> from sympy import cos, sin
>>> TR9(cos(1) + cos(2))
2*cos(1/2)*cos(3/2)
>>> TR9(cos(1) + 2*sin(1) + 2*sin(2))
cos(1) + 4*sin(3/2)*cos(1/2)
```

If no change is made by TR9, no re-arrangement of the expression will be made. For example, though factoring of common term is attempted, if the factored expression was not changed, the original expression will be returned:

```
>>> TR9(cos(3) + cos(3)*cos(2))
cos(3) + cos(2)*cos(3)
```

`sympy.simplify.fu.TR10(rv, first=True)`

Separate sums in cos and sin.

Examples

```
>>> from sympy.simplify.fu import TR10
>>> from sympy.abc import a, b, c
>>> from sympy import cos, sin
>>> TR10(cos(a + b))
-sin(a)*sin(b) + cos(a)*cos(b)
>>> TR10(sin(a + b))
sin(a)*cos(b) + sin(b)*cos(a)
>>> TR10(sin(a + b + c))
(-sin(a)*sin(b) + cos(a)*cos(b))*sin(c) + (sin(a)*cos(b) +
→ sin(b)*cos(a))*cos(c)
```

`sympy.simplify.fu.TR10i(rv)`

Sum of products to function of sum.

Examples

```
>>> from sympy.simplify.fu import TR10i
>>> from sympy import cos, sin, sqrt
>>> from sympy.abc import x
```

```
>>> TR10i(cos(1)*cos(3) + sin(1)*sin(3))
cos(2)
>>> TR10i(cos(1)*sin(3) + sin(1)*cos(3) + cos(3))
cos(3) + sin(4)
>>> TR10i(sqrt(2)*cos(x)*x + sqrt(6)*sin(x)*x)
2*sqrt(2)*x*sin(x + pi/6)
```

`sympy.simplify.fu.TR11(rv, base=None)`

Function of double angle to product. The base argument can be used to indicate what is the un-doubled argument, e.g. if $3\pi/7$ is the base then cosine and sine functions with argument $6\pi/7$ will be replaced.

Examples

```
>>> from sympy.simplify.fu import TR11
>>> from sympy import cos, sin, pi
>>> from sympy.abc import x
>>> TR11(sin(2*x))
2*sin(x)*cos(x)
>>> TR11(cos(2*x))
-sin(x)**2 + cos(x)**2
>>> TR11(sin(4*x))
4*(-sin(x)**2 + cos(x)**2)*sin(x)*cos(x)
>>> TR11(sin(4*x/3))
4*(-sin(x/3)**2 + cos(x/3)**2)*sin(x/3)*cos(x/3)
```

If the arguments are simply integers, no change is made unless a base is provided:

```
>>> TR11(cos(2))
cos(2)
>>> TR11(cos(4), 2)
-sin(2)**2 + cos(2)**2
```

There is a subtle issue here in that autosimplification will convert some higher angles to lower angles

```
>>> cos(6*pi/7) + cos(3*pi/7)
-cos(pi/7) + cos(3*pi/7)
```

The $6\pi/7$ angle is now $\pi/7$ but can be targeted with TR11 by supplying the $3\pi/7$ base:

```
>>> TR11(_, 3*pi/7)
-sin(3*pi/7)**2 + cos(3*pi/7)**2 + cos(3*pi/7)
```

`sympy.simplify.fu.TR12(rv, first=True)`

Separate sums in tan.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy import tan
>>> from sympy.simplify.fu import TR12
>>> TR12(tan(x + y))
(tan(x) + tan(y))/(-tan(x)*tan(y) + 1)
```

`sympy.simplify.fu.TR12i(rv)`

Combine tan arguments as $(\tan(y) + \tan(x))/(\tan(x)\tan(y) - 1) \rightarrow -\tan(x + y)$.

Examples

```
>>> from sympy.simplify.fu import TR12i
>>> from sympy import tan
>>> from sympy.abc import a, b, c
>>> ta, tb, tc = [tan(i) for i in (a, b, c)]
>>> TR12i((ta + tb)/(-ta*tb + 1))
tan(a + b)
>>> TR12i((ta + tb)/(ta*tb - 1))
-tan(a + b)
>>> TR12i((-ta - tb)/(ta*tb - 1))
tan(a + b)
>>> eq = (ta + tb)/(-ta*tb + 1)**2*(-3*ta - 3*tc)/(2*(ta*tc - 1))
>>> TR12i(eq.expand())
-3*tan(a + b)*tan(a + c)/(2*(tan(a) + tan(b) - 1))
```

`sympy.simplify.fu.TR13(rv)`

Change products of tan or cot.

Examples

```
>>> from sympy.simplify.fu import TR13
>>> from sympy import tan, cot
>>> TR13(tan(3)*tan(2))
-tan(2)/tan(5) - tan(3)/tan(5) + 1
>>> TR13(cot(3)*cot(2))
cot(2)*cot(5) + 1 + cot(3)*cot(5)
```

`sympy.simplify.fu.TRmorrie(rv)`

Returns $\cos(x)\cos(2x)\dots\cos(2^{(k-1)}x) \rightarrow \sin(2^k x)/(2^k \sin(x))$

Examples

```
>>> from sympy.simplify.fu import TRmorrie, TR8, TR3
>>> from sympy.abc import x
>>> from sympy import Mul, cos, pi
>>> TRmorrie(cos(x)*cos(2*x))
sin(4*x)/(4*sin(x))
>>> TRmorrie(7*Mul(*[cos(x) for x in range(10)]))
7*sin(12)*sin(16)*cos(5)*cos(7)*cos(9)/(64*sin(1)*sin(3))
```

Sometimes autosimplification will cause a power to be not recognized. e.g. in the following, $\cos(4\pi/7)$ automatically simplifies to $-\cos(3\pi/7)$ so only 2 of the 3 terms are recognized:

```
>>> TRmorrie(cos(pi/7)*cos(2*pi/7)*cos(4*pi/7))
-sin(3*pi/7)*cos(3*pi/7)/(4*sin(pi/7))
```

A touch by TR8 resolves the expression to a Rational

```
>>> TR8(_)
-1/8
```

In this case, if eq is unsimplified, the answer is obtained directly:

```
>>> eq = cos(pi/9)*cos(2*pi/9)*cos(3*pi/9)*cos(4*pi/9)
>>> TRmorrie(eq)
1/16
```

But if angles are made canonical with TR3 then the answer is not simplified without further work:

```
>>> TR3(eq)
sin(pi/18)*cos(pi/9)*cos(2*pi/9)/2
>>> TRmorrie(_)
sin(pi/18)*sin(4*pi/9)/(8*sin(pi/9))
>>> TR8(_)
cos(7*pi/18)/(16*sin(pi/9))
>>> TR3(_)
1/16
```

The original expression would have resolved to 1/16 directly with TR8, however:

```
>>> TR8(eq)
1/16
```

References

[R759]

`sympy.simplify.fu.TR14(rv, first=True)`

Convert factored powers of sin and cos identities into simpler expressions.

Examples

```
>>> from sympy.simplify.fu import TR14
>>> from sympy.abc import x, y
>>> from sympy import cos, sin
>>> TR14((cos(x) - 1)*(cos(x) + 1))
-sin(x)**2
>>> TR14((sin(x) - 1)*(sin(x) + 1))
-cos(x)**2
>>> p1 = (cos(x) + 1)*(cos(x) - 1)
>>> p2 = (cos(y) - 1)**2*(cos(y) + 1)
>>> p3 = (3*(cos(y) - 1))*(3*(cos(y) + 1))
>>> TR14(p1*p2*p3*(x - 1))
-18*(x - 1)*sin(x)**2*sin(y)**4
```

`sympy.simplify.fu.TR15(rv, max=4, pow=False)`

Convert $\sin(x)^{-2}$ to $1 + \cot(x)^2$.

See `_TR56` docstring for advanced use of `max` and `pow`.

Examples

```
>>> from sympy.simplify.fu import TR15
>>> from sympy.abc import x
>>> from sympy import sin
>>> TR15(1 - 1/sin(x)**2)
-cot(x)**2
```

`sympy.simplify.fu.TR16(rv, max=4, pow=False)`

Convert $\cos(x)^{-2}$ to $1 + \tan(x)^2$.

See `_TR56` docstring for advanced use of `max` and `pow`.

Examples

```
>>> from sympy.simplify.fu import TR16
>>> from sympy.abc import x
>>> from sympy import cos
>>> TR16(1 - 1/cos(x)**2)
-tan(x)**2
```

`sympy.simplify.fu.TR111(rv)`

Convert $f(x)^{-i}$ to $g(x)^i$ where either i is an integer or the base is positive and f, g are: $\tan, \cot$; $\sin, \csc$; or $\cos, \sec$.

Examples

```
>>> from sympy.simplify.fu import TR111
>>> from sympy.abc import x
>>> from sympy import tan
>>> TR111(1 - 1/tan(x)**2)
1 - cot(x)**2
```

`sympy.simplify.fu.TR22(rv, max=4, pow=False)`

Convert $\tan(x)^2$ to $\sec(x)^2 - 1$ and $\cot(x)^2$ to $\csc(x)^2 - 1$.

See `_TR56` docstring for advanced use of `max` and `pow`.

Examples

```
>>> from sympy.simplify.fu import TR22
>>> from sympy.abc import x
>>> from sympy import tan, cot
>>> TR22(1 + tan(x)**2)
sec(x)**2
>>> TR22(1 + cot(x)**2)
csc(x)**2
```

`sympy.simplify.fu.TRpower(rv)`

Convert $\sin(x)^n$ and $\cos(x)^n$ with positive n to sums.

Examples

```
>>> from sympy.simplify.fu import TRpower
>>> from sympy.abc import x
>>> from sympy import cos, sin
>>> TRpower(sin(x)**6)
-15*cos(2*x)/32 + 3*cos(4*x)/16 - cos(6*x)/32 + 5/16
>>> TRpower(sin(x)**3*cos(2*x)**4)
(3*sin(x)/4 - sin(3*x)/4)*(cos(4*x)/2 + cos(8*x)/8 + 3/8)
```

References

[R760]

`sympy.simplify.fu.fu(rv, measure=<function <lambda>>)`

Attempt to simplify expression by using transformation rules given in the algorithm by Fu et al.

`fu()` (page 712) will try to minimize the objective function measure. By default this first minimizes the number of trig terms and then minimizes the number of total operations.

Examples

```
>>> from sympy.simplify.fu import fu
>>> from sympy import cos, sin, tan, pi, S, sqrt
>>> from sympy.abc import x, y, a, b
```

```
>>> fu(sin(50)**2 + cos(50)**2 + sin(pi/6))
3/2
>>> fu(sqrt(6)*cos(x) + sqrt(2)*sin(x))
2*sqrt(2)*sin(x + pi/3)
```

CTR1 example

```
>>> eq = sin(x)**4 - cos(y)**2 + sin(y)**2 + 2*cos(x)**2
>>> fu(eq)
cos(x)**4 - 2*cos(y)**2 + 2
```

CTR2 example

```
>>> fu(S.Half - cos(2*x)/2)
sin(x)**2
```

CTR3 example

```
>>> fu(sin(a)*(cos(b) - sin(b)) + cos(a)*(sin(b) + cos(b)))
sqrt(2)*sin(a + b + pi/4)
```

CTR4 example

```
>>> fu(sqrt(3)*cos(x)/2 + sin(x)/2)
sin(x + pi/3)
```

Example 1

```
>>> fu(1-sin(2*x)**2/4-sin(y)**2-cos(x)**4)
-cos(x)**2 + cos(y)**2
```

Example 2

```
>>> fu(cos(4*pi/9))
sin(pi/18)
>>> fu(cos(pi/9)*cos(2*pi/9)*cos(3*pi/9)*cos(4*pi/9))
1/16
```

Example 3

```
>>> fu(tan(7*pi/18)+tan(5*pi/18)-sqrt(3)*tan(5*pi/18)*tan(7*pi/18))
-sqrt(3)
```

Objective function example

```
>>> fu(sin(x)/cos(x)) # default objective function
tan(x)
>>> fu(sin(x)/cos(x), measure=lambda x: -x.count_ops()) # maximize op_
```

(continues on next page)

(continued from previous page)

```
↪count  
sin(x)/cos(x)
```

References

[R761]

Notes

This work was started by Dimitar Vlahovski at the Technological School “Electronic systems” (30.11.2011).

Beyond TR13, other rules are not from the original paper, but extended in SymPy.

References

Solvers

This module documentation contains details about the `sympy.solvers` module. functions.

Contents

Diophantine

Diophantine equations

The word “Diophantine” comes with the name Diophantus, a mathematician lived in the great city of Alexandria sometime around 250 AD. Often referred to as the “father of Algebra”, Diophantus in his famous work “Arithmetica” presented 150 problems that marked the early beginnings of number theory, the field of study about integers and their properties. Diophantine equations play a central and an important part in number theory.

We call a “Diophantine equation” to an equation of the form, $f(x_1, x_2, \dots, x_n) = 0$ where $n \geq 2$ and $x_1, x_2, \dots, x_n$ are integer variables. If we can find n integers $a_1, a_2, \dots, a_n$ such that $x_1 = a_1, x_2 = a_2, \dots, x_n = a_n$ satisfies the above equation, we say that the equation is solvable. You can read more about Diophantine equations in¹ and².

Currently, following five types of Diophantine equations can be solved using `diophantine()` (page 720) and other helper functions of the Diophantine module.

- Linear Diophantine equations: $a_1x_1 + a_2x_2 + \dots + a_nx_n = b$.
- General binary quadratic equation: $ax^2 + bxy + cy^2 + dx + ey + f = 0$
- Homogeneous ternary quadratic equation: $ax^2 + by^2 + cz^2 + dxy + eyz + fzx = 0$
- Extended Pythagorean equation: $a_1x_1^2 + a_2x_2^2 + \dots + a_nx_n^2 = a_{n+1}x_{n+1}^2$

¹ Andreescu, Titu. Andrica, Dorin. Cucurezeanu, Ion. An Introduction to Diophantine Equations

² Diophantine Equation, Wolfram Mathworld, [online]. Available: <http://mathworld.wolfram.com/DiophantineEquation.html>

- General sum of squares: $x_1^2 + x_2^2 + \dots + x_n^2 = k$

Module structure

This module contains `diophantine()` (page 720) and helper functions that are needed to solve certain Diophantine equations. It's structured in the following manner.

- `diophantine()` (page 720)
 - `diop_solve()` (page 721)
 - * `classify_diop()` (page 721)
 - * `diop_linear()` (page 722)
 - * `diop_quadratic()` (page 723)
 - * `diop_ternary_quadratic()` (page 729)
 - * `diop_ternary_quadratic_normal()` (page 741)
 - * `diop_general_pythagorean()` (page 731)
 - * `diop_general_sum_of_squares()` (page 731)
 - * `diop_general_sum_of_even_powers()` (page 732)
 - `merge_solution()` (page 737)

When an equation is given to `diophantine()` (page 720), it factors the equation(if possible) and solves the equation given by each factor by calling `diop_solve()` (page 721) separately. Then all the results are combined using `merge_solution()` (page 737).

`diop_solve()` (page 721) internally uses `classify_diop()` (page 721) to find the type of the equation(and some other details) given to it and then calls the appropriate solver function based on the type returned. For example, if `classify_diop()` (page 721) returned "linear" as the type of the equation, then `diop_solve()` (page 721) calls `diop_linear()` (page 722) to solve the equation.

Each of the functions, `diop_linear()` (page 722), `diop_quadratic()` (page 723), `diop_ternary_quadratic()` (page 729), `diop_general_pythagorean()` (page 731) and `diop_general_sum_of_squares()` (page 731) solves a specific type of equations and the type can be easily guessed by it's name.

Apart from these functions, there are a considerable number of other functions in the "Diophantine Module" and all of them are listed under User functions and Internal functions.

Tutorial

First, let's import the highest API of the Diophantine module.

```
>>> from sympy.solvers.diophantine import diophantine
```

Before we start solving the equations, we need to define the variables.

```
>>> from sympy import symbols
>>> x, y, z = symbols("x, y, z", integer=True)
```

Let's start by solving the easiest type of Diophantine equations, i.e. linear Diophantine equations. Let's solve $2x + 3y = 5$. Note that although we write the equation in the above form, when we input the equation to any of the functions in Diophantine module, it needs to be in the form $eq = 0$.

```
>>> diophantine(2*x + 3*y - 5)
{(3*t_0 - 5, 5 - 2*t_0)}
```

Note that stepping one more level below the highest API, we can solve the very same equation by calling `diop_solve()` (page 721).

```
>>> from sympy.solvers.diophantine.diophantine import diop_solve
>>> diop_solve(2*x + 3*y - 5)
(3*t_0 - 5, 5 - 2*t_0)
```

Note that it returns a tuple rather than a set. `diophantine()` (page 720) always return a set of tuples. But `diop_solve()` (page 721) may return a single tuple or a set of tuples depending on the type of the equation given.

We can also solve this equation by calling `diop_linear()` (page 722), which is what `diop_solve()` (page 721) calls internally.

```
>>> from sympy.solvers.diophantine.diophantine import diop_linear
>>> diop_linear(2*x + 3*y - 5)
(3*t_0 - 5, 5 - 2*t_0)
```

If the given equation has no solutions then the outputs will look like below.

```
>>> diophantine(2*x + 4*y - 3)
set()
>>> diop_solve(2*x + 4*y - 3)
(None, None)
>>> diop_linear(2*x + 4*y - 3)
(None, None)
```

Note that except for the highest level API, in case of no solutions, a tuple of `None` are returned. Size of the tuple is the same as the number of variables. Also, one can specifically set the parameter to be used in the solutions by passing a customized parameter. Consider the following example:

```
>>> m = symbols("m", integer=True)
>>> diop_solve(2*x + 3*y - 5, m)
(3*m_0 - 5, 5 - 2*m_0)
```

For linear Diophantine equations, the customized parameter is the prefix used for each free variable in the solution. Consider the following example:

```
>>> diop_solve(2*x + 3*y - 5*z + 7, m)
(m_0, m_0 + 5*m_1 - 14, m_0 + 3*m_1 - 7)
```

In the solution above, `m_0` and `m_1` are independent free variables.

Please note that for the moment, users can set the parameter only for linear Diophantine equations and binary quadratic equations.

Let's try solving a binary quadratic equation which is an equation with two variables and has a degree of two. Before trying to solve these equations, an idea about various cases

associated with the equation would help a lot. Please refer³ and⁴ for detailed analysis of different cases and the nature of the solutions. Let us define $\Delta = b^2 - 4ac$ w.r.t. the binary quadratic $ax^2 + bxy + cy^2 + dx + ey + f = 0$.

When $\Delta < 0$, there are either no solutions or only a finite number of solutions.

```
>>> diophantine(x**2 - 4*x*y + 8*y**2 - 3*x + 7*y - 5)
{(2, 1), (5, 1)}
```

In the above equation $\Delta = (-4)^2 - 4 * 1 * 8 = -16$ and hence only a finite number of solutions exist.

When $\Delta = 0$ we might have either no solutions or parameterized solutions.

```
>>> diophantine(3*x**2 - 6*x*y + 3*y**2 - 3*x + 7*y - 5)
set()
>>> diophantine(x**2 - 4*x*y + 4*y**2 - 3*x + 7*y - 5)
{(-2*t**2 - 7*t + 10, -t**2 - 3*t + 5)}
>>> diophantine(x**2 + 2*x*y + y**2 - 3*x - 3*y)
{(t_0, -t_0), (t_0, 3 - t_0)}
```

The most interesting case is when $\Delta > 0$ and it is not a perfect square. In this case, the equation has either no solutions or an infinite number of solutions. Consider the below cases where $\Delta = 8$.

```
>>> diophantine(x**2 - 4*x*y + 2*y**2 - 3*x + 7*y - 5)
set()
>>> from sympy import sqrt
>>> n = symbols("n", integer=True)
>>> s = diophantine(x**2 - 2*y**2 - 2*x - 4*y, n)
>>> x_1, y_1 = s.pop()
>>> x_2, y_2 = s.pop()
>>> x_n = -(-2*sqrt(2) + 3)**n/2 + sqrt(2)*(-2*sqrt(2) + 3)**n/2 - 1
↳ sqrt(2)*(2*sqrt(2) + 3)**n/2 - (2*sqrt(2) + 3)**n/2 + 1
>>> x_1 == x_n or x_2 == x_n
True
>>> y_n = -sqrt(2)*(-2*sqrt(2) + 3)**n/4 + (-2*sqrt(2) + 3)**n/2 + 1
↳ sqrt(2)*(2*sqrt(2) + 3)**n/4 + (2*sqrt(2) + 3)**n/2 - 1
>>> y_1 == y_n or y_2 == y_n
True
```

Here n is an integer. Although x_n and y_n may not look like integers, substituting in specific values for n (and simplifying) shows that they are. For example consider the following example where we set n equal to 9.

```
>>> from sympy import simplify
>>> simplify(x_n.subs({n: 9}))
-9369318
```

Any binary quadratic of the form $ax^2 + bxy + cy^2 + dx + ey + f = 0$ can be transformed to an equivalent form $X^2 - DY^2 = N$.

³ Methods to solve $Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$, [online], Available: <http://www.alpertron.com.ar/METHODS.HTM>

⁴ Solving the equation $ax^2 + bxy + cy^2 + dx + ey + f = 0$, [online], Available: <https://web.archive.org/web/20160323033111/http://www.jpr2718.org/ax2p.pdf>

```
>>> from sympy.solvers.diophantine.diophantine import find_DN, diop_DN, \
    transformation_to_DN
>>> find_DN(x**2 - 3*x*y + y**2 - 7*x + 5*y - 3)
(5, 920)
```

So, the above equation is equivalent to the equation $X^2 - 5Y^2 = 920$ after a linear transformation. If we want to find the linear transformation, we can use [transformation_to_DN\(\)](#) (page 727)

```
>>> A, B = transformation_to_DN(x**2 - 3*x*y + y**2 - 7*x + 5*y - 3)
```

Here A is a 2 X 2 matrix and B is a 2 X 1 matrix such that the transformation

$$\begin{bmatrix} X \\ Y \end{bmatrix} = A \begin{bmatrix} x \\ y \end{bmatrix} + B$$

gives the equation $X^2 - 5Y^2 = 920$. Values of A and B are as follows.

```
>>> A
Matrix([
[1/10, 3/10],
[ 0, 1/5]])
>>> B
Matrix([
[ 1/5],
[-11/5]])
```

We can solve an equation of the form $X^2 - DY^2 = N$ by passing D and N to [diop_DN\(\)](#) (page 724)

```
>>> diop_DN(5, 920)
[]
```

Unfortunately, our equation has no solution.

Now let's turn to homogeneous ternary quadratic equations. These equations are of the form $ax^2 + by^2 + cz^2 + dxy + eyz + fzx = 0$. These type of equations either have infinitely many solutions or no solutions (except the obvious solution (0, 0, 0))

```
>>> diophantine(3*x**2 + 4*y**2 - 5*z**2 + 4*x*y + 6*y*z + 7*z*x)
{(0, 0, 0)}
>>> diophantine(3*x**2 + 4*y**2 - 5*z**2 + 4*x*y - 7*y*z + 7*z*x)
{(-16*p**2 + 28*p*q + 20*q**2, 3*p**2 + 38*p*q - 25*q**2, 4*p**2 - 24*p*q + \
    68*q**2)}
```

If you are only interested in a base solution rather than the parameterized general solution (to be more precise, one of the general solutions), you can use [diop_ternary_quadratic\(\)](#) (page 729).

```
>>> from sympy.solvers.diophantine.diophantine import diop_ternary_quadratic
>>> diop_ternary_quadratic(3*x**2 + 4*y**2 - 5*z**2 + 4*x*y - 7*y*z + 7*z*x)
(-4, 5, 1)
```

[diop_ternary_quadratic\(\)](#) (page 729) first converts the given equation to an equivalent equation of the form $w^2 = AX^2 + BY^2$ and then it uses [descent\(\)](#) (page 730) to solve the latter equation. You can refer to the docs of [transformation_to_normal\(\)](#) (page 728) to find more

on this. The equation $w^2 = AX^2 + BY^2$ can be solved more easily by using the Aforementioned `descent()` (page 730).

```
>>> from sympy.solvers.diophantine.diophantine import descent
>>> descent(3, 1) # solves the equation w**2 = 3*Y**2 + Z**2
(1, 0, 1)
```

Here the solution tuple is in the order (w, Y, Z)

The extended Pythagorean equation, $a_1x_1^2 + a_2x_2^2 + \dots + a_nx_n^2 = a_{n+1}x_{n+1}^2$ and the general sum of squares equation, $x_1^2 + x_2^2 + \dots + x_n^2 = k$ can also be solved using the Diophantine module.

```
>>> from sympy.abc import a, b, c, d, e, f
>>> diophantine(9*a**2 + 16*b**2 + c**2 + 49*d**2 + 4*e**2 - 25*f**2)
{(70*t1**2 + 70*t2**2 + 70*t3**2 + 70*t4**2 - 70*t5**2, 105*t1*t5, 420*t2*t5,
 60*t3*t5, 210*t4*t5, 42*t1**2 + 42*t2**2 + 42*t3**2 + 42*t4**2 + 42*t5**2)}
```

function `diop_general_pythagorean()` (page 731) can also be called directly to solve the same equation. Either you can call `diop_general_pythagorean()` (page 731) or use the high level API. For the general sum of squares, this is also true, but one advantage of calling `diop_general_sum_of_squares()` (page 731) is that you can control how many solutions are returned.

```
>>> from sympy.solvers.diophantine.diophantine import diop_general_sum_of_
  squares
>>> eq = a**2 + b**2 + c**2 + d**2 - 18
>>> diophantine(eq)
{(0, 0, 3, 3), (0, 1, 1, 4), (1, 2, 2, 3)}
>>> diop_general_sum_of_squares(eq, 2)
{(0, 0, 3, 3), (1, 2, 2, 3)}
```

The `sum_of_squares()` (page 737) routine will provide an iterator that returns solutions and one may control whether the solutions contain zeros or not (and the solutions not containing zeros are returned first):

```
>>> from sympy.solvers.diophantine.diophantine import sum_of_squares
>>> sos = sum_of_squares(18, 4, zeros=True)
>>> next(sos)
(1, 2, 2, 3)
>>> next(sos)
(0, 0, 3, 3)
```

Simple Egyptian fractions can be found with the Diophantine module, too. For example, here are the ways that one might represent $1/2$ as a sum of two unit fractions:

```
>>> from sympy import Eq, S
>>> diophantine(Eq(1/x + 1/y, S(1)/2))
{(-2, 1), (1, -2), (3, 6), (4, 4), (6, 3)}
```

To get a more thorough understanding of the Diophantine module, please refer to the following blog.

<http://thilinaatsympy.wordpress.com/>

References

User Functions

This functions is imported into the global namespace with `from sympy import *`:

diophantine

`sympy.solvers.diophantine.diophantine.diophantine(eq, param=t, syms=None, permute=False)`

Simplify the solution procedure of diophantine equation `eq` by converting it into a product of terms which should equal zero.

Explanation

For example, when solving, $x^2 - y^2 = 0$ this is treated as $(x + y)(x - y) = 0$ and $x + y = 0$ and $x - y = 0$ are solved independently and combined. Each term is solved by calling `diop_solve()`. (Although it is possible to call `diop_solve()` directly, one must be careful to pass an equation in the correct form and to interpret the output correctly; `diophantine()` is the public-facing function to use in general.)

Output of `diophantine()` is a set of tuples. The elements of the tuple are the solutions for each variable in the equation and are arranged according to the alphabetic ordering of the variables. e.g. For an equation with two variables, a and b , the first element of the tuple is the solution for a and the second for b .

Usage

`diophantine(eq, t, syms)`: Solve the diophantine equation `eq`. `t` is the optional parameter to be used by `diop_solve()`. `syms` is an optional list of symbols which determines the order of the elements in the returned tuple.

By default, only the base solution is returned. If `permute` is set to `True` then permutations of the base solution and/or permutations of the signs of the values will be returned when applicable.

Examples

```
>>> from sympy.abc import x, y, z
>>> diophantine(x**2 - y**2)
{(t_0, -t_0), (t_0, t_0)}
```

```
>>> diophantine(x*(2*x + 3*y - z))
{(0, n1, n2), (t_0, t_1, 2*t_0 + 3*t_1)}
>>> diophantine(x**2 + 3*x*y + 4*x)
{(0, n1), (3*t_0 - 4, -t_0)}
```


Details

`eq` should be an expression which is assumed to be zero. `t` is the parameter to be used in the solution.

See also:

[*diop_solve*](#) (page 721), [*sympy.utilities.iterables.permute_signs*](#) (page 2090), [*sympy.utilities.iterables.signed_permutations*](#) (page 2094)

And this function is imported with `from sympy.solvers.diophantine import *`:

`classify_diop`

```
sympy.solvers.diophantine.diophantine.classify_diop(eq, _dict=True)
```

Internal Functions

These functions are intended for internal use in the Diophantine module.

`diop_solve`

```
sympy.solvers.diophantine.diophantine.diop_solve(eq, param=t)
```

Solves the diophantine equation `eq`.

Explanation

Unlike `diophantine()`, factoring of `eq` is not attempted. Uses `classify_diop()` to determine the type of the equation and calls the appropriate solver function.

Use of `diophantine()` is recommended over other helper functions. `diop_solve()` can return either a set or a tuple depending on the nature of the equation.

Usage

`diop_solve(eq, t)`: Solve diophantine equation, `eq` using `t` as a parameter if needed.

Details

`eq` should be an expression which is assumed to be zero. `t` is a parameter to be used in the solution.

Examples

```
>>> from sympy.solvers.diophantine import diop_solve
>>> from sympy.abc import x, y, z, w
>>> diop_solve(2*x + 3*y - 5)
(3*t_0 - 5, 5 - 2*t_0)
>>> diop_solve(4*x + 3*y - 4*z + 5)
(t_0, 8*t_0 + 4*t_1 + 5, 7*t_0 + 3*t_1 + 5)
>>> diop_solve(x + 3*y - 4*z + w - 6)
(t_0, t_0 + t_1, 6*t_0 + 5*t_1 + 4*t_2 - 6, 5*t_0 + 4*t_1 + 3*t_2 - 6)
>>> diop_solve(x**2 + y**2 - 5)
{(-2, -1), (-2, 1), (-1, -2), (-1, 2), (1, -2), (1, 2), (2, -1), (2, 1)}
```

See also:

[*diophantine*](#) (page 720)

diop_linear

`sympy.solvers.diophantine.diophantine.diop_linear(eq, param=t)`

Solves linear diophantine equations.

A linear diophantine equation is an equation of the form $a_1x_1 + a_2x_2 + \dots + a_nx_n = 0$ where $a_1, a_2, \dots, a_n$ are integer constants and $x_1, x_2, \dots, x_n$ are integer variables.

Usage

`diop_linear(eq)`: Returns a tuple containing solutions to the diophantine equation `eq`. Values in the tuple is arranged in the same order as the sorted variables.

Details

`eq` is a linear diophantine equation which is assumed to be zero. `param` is the parameter to be used in the solution.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import diop_linear
>>> from sympy.abc import x, y, z
>>> diop_linear(2*x - 3*y - 5) # solves equation 2*x - 3*y - 5 == 0
(3*t_0 - 5, 2*t_0 - 5)
```

Here $x = -3t_0 - 5$ and $y = -2t_0 - 5$

```
>>> diop_linear(2*x - 3*y - 4*z - 3)
(t_0, 2*t_0 + 4*t_1 + 3, -t_0 - 3*t_1 - 3)
```

See also:

[*diop_quadratic*](#) (page 723), [*diop_ternary_quadratic*](#) (page 729),
[*diop_general_pythagorean*](#) (page 731), [*diop_general_sum_of_squares*](#) (page 731)

base_solution_linear

`sympy.solvers.diophantine.diophantine.base_solution_linear(c, a, b, t=None)`

Return the base solution for the linear equation, $ax + by = c$.

Explanation

Used by `diop_linear()` to find the base solution of a linear Diophantine equation. If `t` is given then the parametrized solution is returned.

Usage

`base_solution_linear(c, a, b, t)`: `a`, `b`, `c` are coefficients in $ax + by = c$ and `t` is the parameter to be used in the solution.

Examples

```

>>> from sympy.solvers.diophantine.diophantine import base_solution_
    ↳ linear
>>> from sympy.abc import t
>>> base_solution_linear(5, 2, 3) # equation 2*x + 3*y = 5
(-5, 5)
>>> base_solution_linear(0, 5, 7) # equation 5*x + 7*y = 0
(0, 0)
>>> base_solution_linear(5, 2, 3, t) # equation 2*x + 3*y = 5
(3*t - 5, 5 - 2*t)
>>> base_solution_linear(0, 5, 7, t) # equation 5*x + 7*y = 0
(7*t, -5*t)

```

diop_quadratic

`sympy.solvers.diophantine.diophantine.diop_quadratic(eq, param=t)`

Solves quadratic diophantine equations.

i.e. equations of the form $Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$. Returns a set containing the tuples (x, y) which contains the solutions. If there are no solutions then $(None, None)$ is returned.

Usage

`diop_quadratic(eq, param)`: `eq` is a quadratic binary diophantine equation. `param` is used to indicate the parameter to be used in the solution.

Details

eq should be an expression which is assumed to be zero. param is a parameter to be used in the solution.

Examples

```
>>> from sympy.abc import x, y, t
>>> from sympy.solvers.diophantine.diophantine import diop_quadratic
>>> diop_quadratic(x**2 + y**2 + 2*x + 2*y + 2, t)
{(-1, -1)}
```

See also:

[*diop_linear*](#) (page 722), [*diop_ternary_quadratic*](#) (page 729),
[*diop_general_sum_of_squares*](#) (page 731), [*diop_general_pythagorean*](#) (page 731)

References

[R765], [R766]

diop_DN

`sympy.solvers.diophantine.diophantine.diop_DN(D, N, t=t)`

Solves the equation $x^2 - Dy^2 = N$.

Explanation

Mainly concerned with the case $D > 0$, D is not a perfect square, which is the same as the generalized Pell equation. The LMM algorithm [R767] is used to solve this equation.

Returns one solution tuple, (x, y) for each class of the solutions. Other solutions of the class can be constructed according to the values of D and N .

Usage

`diop_DN(D, N, t)`: D and N are integers as in $x^2 - Dy^2 = N$ and t is the parameter to be used in the solutions.

Details

D and N correspond to D and N in the equation. t is the parameter to be used in the solutions.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import diop_DN
>>> diop_DN(13, -4) # Solves equation x**2 - 13*y**2 = -4
[(3, 1), (393, 109), (36, 10)]
```

The output can be interpreted as follows: There are three fundamental solutions to the equation $x^2 - 13y^2 = -4$ given by (3, 1), (393, 109) and (36, 10). Each tuple is in the form (x, y), i.e. solution (3, 1) means that $x = 3$ and $y = 1$.

```
>>> diop_DN(986, 1) # Solves equation x**2 - 986*y**2 = 1
[(49299, 1570)]
```

See also:

[*find_DN*](#) (page 728), [*diop_bf_DN*](#) (page 726)

References

[R767]

cornacchia

`sympy.solvers.diophantine.diophantine.cornacchia(a, b, m)`

Solves $ax^2 + by^2 = m$ where $\gcd(a, b) = 1 = \gcd(a, m)$ and $a, b > 0$.

Explanation

Uses the algorithm due to Cornacchia. The method only finds primitive solutions, i.e. ones with $\gcd(x, y) = 1$. So this method cannot be used to find the solutions of $x^2 + y^2 = 20$ since the only solution to former is $(x, y) = (4, 2)$ and it is not primitive. When $a = b$, only the solutions with $x \leq y$ are found. For more details, see the References.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import cornacchia
>>> cornacchia(2, 3, 35) # equation 2x**2 + 3y**2 = 35
{(2, 3), (4, 1)}
>>> cornacchia(1, 1, 25) # equation x**2 + y**2 = 25
{(4, 3)}
```

See also:

[*sympy.utilities.iterables.signed_permutations*](#) (page 2094)

References

[R768], [R769]

diop_bf_DN

`sympy.solvers.diophantine.diophantine.diop_bf_DN(D, N, t=t)`

Uses brute force to solve the equation, $x^2 - Dy^2 = N$.

Explanation

Mainly concerned with the generalized Pell equation which is the case when $D > 0$, D is not a perfect square. For more information on the case refer [R770]. Let (t, u) be the minimal positive solution of the equation $x^2 - Dy^2 = 1$. Then this method requires $\sqrt{\text{frac}|N| (t \pm 1)2D}$ to be small.

Usage

`diop_bf_DN(D, N, t)`: D and N are coefficients in $x^2 - Dy^2 = N$ and t is the parameter to be used in the solutions.

Details

D and N correspond to D and N in the equation. t is the parameter to be used in the solutions.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import diop_bf_DN
>>> diop_bf_DN(13, -4)
[(3, 1), (-3, 1), (36, 10)]
>>> diop_bf_DN(986, 1)
[(49299, 1570)]
```

See also:

[`diop\_DN`](#) (page 724)

References

[R770]

transformation_to_DN

`sympy.solvers.diophantine.diophantine.transformation_to_DN(eq)`

This function transforms general quadratic, $ax^2 + bxy + cy^2 + dx + ey + f = 0$ to more easy to deal with $X^2 - DY^2 = N$ form.

Explanation

This is used to solve the general quadratic equation by transforming it to the latter form. Refer to [R771] for more detailed information on the transformation. This function returns a tuple (A, B) where A is a 2 X 2 matrix and B is a 2 X 1 matrix such that,

$$\text{Transpose}([x \ y]) = A * \text{Transpose}([X \ Y]) + B$$

Usage

`transformation_to_DN(eq)`: where eq is the quadratic to be transformed.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy.solvers.diophantine.diophantine import transformation_to_
→DN
>>> A, B = transformation_to_DN(x**2 - 3*x*y - y**2 - 2*y + 1)
>>> A
Matrix([
[1/26, 3/26],
[ 0, 1/13]])
>>> B
Matrix([
[-6/13],
[-4/13]])
```

A, B returned are such that $\text{Transpose}([x \ y]) = A * \text{Transpose}([X \ Y]) + B$. Substituting these values for x and y and a bit of simplifying work will give an equation of the form $x^2 - Dy^2 = N$.

```
>>> from sympy.abc import X, Y
>>> from sympy import Matrix, simplify
>>> u = (A*Matrix([X, Y]) + B)[0] # Transformation for x
>>> u
X/26 + 3*Y/26 - 6/13
>>> v = (A*Matrix([X, Y]) + B)[1] # Transformation for y
>>> v
Y/13 - 4/13
```

Next we will substitute these formulas for x and y and do `simplify()`.

```
>>> eq = simplify((x**2 - 3*x*y - y**2 - 2*y + 1).subs(zip((x, y), (u, v))))
>>> eq
X**2/676 - Y**2/52 + 17/13
```

By multiplying the denominator appropriately, we can get a Pell equation in the standard form.

```
>>> eq * 676
X**2 - 13*Y**2 + 884
```

If only the final equation is needed, `find_DN()` can be used.

See also:

[`find\_DN`](#) (page 728)

References

[R771]

transformation_to_normal

`sympy.solvers.diophantine.diophantine.transformation_to_normal(eq)`

Returns the transformation Matrix that converts a general ternary quadratic equation eq ($ax^2 + by^2 + cz^2 + dxy + eyz + fzx$) to a form without cross terms: $ax^2 + by^2 + cz^2 = 0$. This is not used in solving ternary quadratics; it is only implemented for the sake of completeness.

find_DN

`sympy.solvers.diophantine.diophantine.find_DN(eq)`

This function returns a tuple, (D, N) of the simplified form, $x^2 - Dy^2 = N$, corresponding to the general quadratic, $ax^2 + bxy + cy^2 + dx + ey + f = 0$.

Solving the general quadratic is then equivalent to solving the equation $X^2 - DY^2 = N$ and transforming the solutions by using the transformation matrices returned by `transformation_to_DN()`.

Usage

`find_DN(eq)`: where eq is the quadratic to be transformed.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy.solvers.diophantine.diophantine import find_DN
>>> find_DN(x**2 - 3*x*y - y**2 - 2*y + 1)
(13, -884)
```

Interpretation of the output is that we get $X^2 - 13Y^2 = -884$ after transforming $x^2 - 3xy - y^2 - 2y + 1$ using the transformation returned by `transformation_to_DN()`.

See also:

[`transformation\_to\_DN`](#) (page 727)

References

[R772]

diop_ternary_quadratic

`sympy.solvers.diophantine.diophantine.diop_ternary_quadratic(eq, parameterize=False)`

Solves the general quadratic ternary form, $ax^2 + by^2 + cz^2 + fxy + gyz + hzx = 0$.

Returns a tuple (x, y, z) which is a base solution for the above equation. If there are no solutions, $(None, None, None)$ is returned.

Usage

`diop_ternary_quadratic(eq)`: Return a tuple containing a basic solution to eq.

Details

eq should be an homogeneous expression of degree two in three variables and it is assumed to be zero.

Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.solvers.diophantine.diophantine import diop_ternary_
↳ quadratic
>>> diop_ternary_quadratic(x**2 + 3*y**2 - z**2)
(1, 0, 1)
>>> diop_ternary_quadratic(4*x**2 + 5*y**2 - z**2)
(1, 0, 2)
>>> diop_ternary_quadratic(45*x**2 - 7*y**2 - 8*x*y - z**2)
(28, 45, 105)
>>> diop_ternary_quadratic(x**2 - 49*y**2 - z**2 + 13*z*y - 8*x*y)
(9, 1, 5)
```

square_factor

`sympy.solvers.diophantine.diophantine.square_factor(a)`

Returns an integer c s.t. $a = c^2k$, $c, k \in \mathbb{Z}$. Here k is square free. a can be given as an integer or a dictionary of factors.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import square_factor
>>> square_factor(24)
2
>>> square_factor(-36*3)
6
>>> square_factor(1)
1
>>> square_factor({3: 2, 2: 1, -1: 1}) # -18
3
```

See also:

[`sympy.ntheory.factor\_.core`](#) (page 1504)

descent

`sympy.solvers.diophantine.diophantine.descent(A, B)`

Returns a non-trivial solution, (x, y, z) , to $x^2 = Ay^2 + Bz^2$ using Lagrange's descent method with lattice-reduction. A and B are assumed to be valid for such a solution to exist.

This is faster than the normal Lagrange's descent algorithm because the Gaussian reduction is used.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import descent
>>> descent(3, 1) # x**2 = 3*y**2 + z**2
(1, 0, 1)
```

$(x, y, z) = (1, 0, 1)$ is a solution to the above equation.

```
>>> descent(41, -113)
(-16, -3, 1)
```

References

[R773]

diop_general_pythagorean

`sympy.solvers.diophantine.diophantine.diop_general_pythagorean(eq, param=m)`

Solves the general pythagorean equation, $a_1^2x_1^2 + a_2^2x_2^2 + \dots + a_n^2x_n^2 - a_{n+1}^2x_{n+1}^2 = 0$.

Returns a tuple which contains a parametrized solution to the equation, sorted in the same order as the input variables.

Usage

`diop_general_pythagorean(eq, param)`: where `eq` is a general pythagorean equation which is assumed to be zero and `param` is the base parameter used to construct other parameters by subscripting.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import diop_general_
    ↳ pythagorean
>>> from sympy.abc import a, b, c, d, e
>>> diop_general_pythagorean(a**2 + b**2 + c**2 - d**2)
(m1**2 + m2**2 - m3**2, 2*m1*m3, 2*m2*m3, m1**2 + m2**2 + m3**2)
>>> diop_general_pythagorean(9*a**2 - 4*b**2 + 16*c**2 + 25*d**2 + e**2)
(10*m1**2 + 10*m2**2 + 10*m3**2 - 10*m4**2, 15*m1**2 + 15*m2**2 +
    ↳ 15*m3**2 + 15*m4**2, 15*m1*m4, 12*m2*m4, 60*m3*m4)
```

diop_general_sum_of_squares

`sympy.solvers.diophantine.diophantine.diop_general_sum_of_squares(eq, limit=1)`

Solves the equation $x_1^2 + x_2^2 + \dots + x_n^2 - k = 0$.

Returns at most `limit` number of solutions.

Usage

`general_sum_of_squares(eq, limit)`: Here `eq` is an expression which is assumed to be zero. Also, `eq` should be in the form, $x_1^2 + x_2^2 + \dots + x_n^2 - k = 0$.

Details

When $n = 3$ if $k = 4^a(8m + 7)$ for some $a, m \in \mathbb{Z}$ then there will be no solutions. Refer to [R774] for more details.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import diop_general_sum_
    ↳ of_squares
>>> from sympy.abc import a, b, c, d, e
>>> diop_general_sum_of_squares(a**2 + b**2 + c**2 + d**2 + e**2 - 2345)
{(15, 22, 22, 24, 24)}
```

Reference

diop_general_sum_of_even_powers

`sympy.solvers.diophantine.diophantine.diop_general_sum_of_even_powers(eq, limit=1)`

Solves the equation $x_1^e + x_2^e + \dots + x_n^e - k = 0$ where e is an even, integer power.

Returns at most `limit` number of solutions.

Usage

`general_sum_of_even_powers(eq, limit)`: Here `eq` is an expression which is assumed to be zero. Also, `eq` should be in the form, $x_1^e + x_2^e + \dots + x_n^e - k = 0$.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import diop_general_sum_
    ↳ of_even_powers
>>> from sympy.abc import a, b
>>> diop_general_sum_of_even_powers(a**4 + b**4 - (2**4 + 3**4))
{(2, 3)}
```

See also:

[*power_representation*](#) (page 733)

power_representation

`sympy.solvers.diophantine.diophantine.power_representation(n, p, k, zeros=False)`

Returns a generator for finding k-tuples of integers, $(n_1, n_2, \dots, n_k)$, such that $n = n_1^p + n_2^p + \dots + n_k^p$.

Usage

`power_representation(n, p, k, zeros)`: Represent non-negative number n as a sum of k p th powers. If `zeros` is true, then the solutions is allowed to contain zeros.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import power_
      representation
```

Represent 1729 as a sum of two cubes:

```
>>> f = power_representation(1729, 3, 2)
>>> next(f)
(9, 10)
>>> next(f)
(1, 12)
```

If the flag `zeros` is True, the solution may contain tuples with zeros; any such solutions will be generated after the solutions without zeros:

```
>>> list(power_representation(125, 2, 3, zeros=True))
[(5, 6, 8), (3, 4, 10), (0, 5, 10), (0, 2, 11)]
```

For even p the `permute_sign` function can be used to get all signed values:

```
>>> from sympy.utilities.iterables import permute_signs
>>> list(permute_signs((1, 12)))
[(1, 12), (-1, 12), (1, -12), (-1, -12)]
```

All possible signed permutations can also be obtained:

```
>>> from sympy.utilities.iterables import signed_permutations
>>> list(signed_permutations((1, 12)))
[(1, 12), (-1, 12), (1, -12), (-1, -12), (12, 1), (-12, 1), (12, -1), (-
→ 12, -1)]
```

partition

`sympy.solvers.diophantine.diophantine.partition(n, k=None, zeros=False)`

Returns a generator that can be used to generate partitions of an integer n .

Explanation

A partition of n is a set of positive integers which add up to n . For example, partitions of 3 are 3, 1 + 2, 1 + 1 + 1. A partition is returned as a tuple. If k equals `None`, then all possible partitions are returned irrespective of their size, otherwise only the partitions of size k are returned. If the `zero` parameter is set to `True` then a suitable number of zeros are added at the end of every partition of size less than k .

`zero` parameter is considered only if k is not `None`. When the partitions are over, the last `next()` call throws the `StopIteration` exception, so this function should always be used inside a `try - except` block.

Details

`partition(n, k)`: Here n is a positive integer and k is the size of the partition which is also positive integer.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import partition
>>> f = partition(5)
>>> next(f)
(1, 1, 1, 1, 1)
>>> next(f)
(1, 1, 1, 2)
>>> g = partition(5, 3)
>>> next(g)
(1, 1, 3)
>>> next(g)
(1, 2, 2)
>>> g = partition(5, 3, zeros=True)
>>> next(g)
(0, 0, 5)
```

sum_of_three_squares

`sympy.solvers.diophantine.diophantine.sum_of_three_squares(n)`

Returns a 3-tuple (a, b, c) such that $a^2 + b^2 + c^2 = n$ and $a, b, c \geq 0$.

Returns `None` if $n = 4^a(8m + 7)$ for some $a, m \in \mathbb{Z}$. See [R775] for more details.

Usage

`sum_of_three_squares(n)`: Here n is a non-negative integer.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import sum_of_three_
      ↪ squares
>>> sum_of_three_squares(44542)
(18, 37, 207)
```

See also:

[`sum\_of\_squares`](#) (page 737)

References

[R775]

`sum_of_four_squares`

`sympy.solvers.diophantine.diophantine.sum_of_four_squares(n)`

Returns a 4-tuple (a, b, c, d) such that $a^2 + b^2 + c^2 + d^2 = n$.

Here $a, b, c, d \geq 0$.

Usage

`sum_of_four_squares(n)`: Here n is a non-negative integer.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import sum_of_four_squares
>>> sum_of_four_squares(3456)
(8, 8, 32, 48)
>>> sum_of_four_squares(1294585930293)
(0, 1234, 2161, 1137796)
```

See also:

[`sum\_of\_squares`](#) (page 737)

References

[R776]

sum_of_powers

`sympy.solvers.diophantine.diophantine.sum_of_powers(n, p, k, zeros=False)`

Returns a generator for finding k-tuples of integers, $(n_1, n_2, \dots, n_k)$, such that $n = n_1^p + n_2^p + \dots + n_k^p$.

Usage

`power_representation(n, p, k, zeros)`: Represent non-negative number n as a sum of k p th powers. If `zeros` is true, then the solutions is allowed to contain zeros.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import power_
      representation
```

Represent 1729 as a sum of two cubes:

```
>>> f = power_representation(1729, 3, 2)
>>> next(f)
(9, 10)
>>> next(f)
(1, 12)
```

If the flag `zeros` is True, the solution may contain tuples with zeros; any such solutions will be generated after the solutions without zeros:

```
>>> list(power_representation(125, 2, 3, zeros=True))
[(5, 6, 8), (3, 4, 10), (0, 5, 10), (0, 2, 11)]
```

For even p the `permute_sign` function can be used to get all signed values:

```
>>> from sympy.utilities.iterables import permute_signs
>>> list(permute_signs((1, 12)))
[(1, 12), (-1, 12), (1, -12), (-1, -12)]
```

All possible signed permutations can also be obtained:

```
>>> from sympy.utilities.iterables import signed_permutations
>>> list(signed_permutations((1, 12)))
[(1, 12), (-1, 12), (1, -12), (-1, -12), (12, 1), (-12, 1), (12, -1), (-
→ 12, -1)]
```


sum_of_squares

`sympy.solvers.diophantine.diophantine.sum_of_squares(n, k, zeros=False)`

Return a generator that yields the k -tuples of nonnegative values, the squares of which sum to n . If `zeros` is `False` (default) then the solution will not contain zeros. The nonnegative elements of a tuple are sorted.

- If $k == 1$ and n is square, $(n,)$ is returned.
- If $k == 2$ then n can only be written as a sum of squares if every prime in the factorization of n that has the form $4*k + 3$ has an even multiplicity. If n is prime then it can only be written as a sum of two squares if it is in the form $4*k + 1$.
- if $k == 3$ then n can be written as a sum of squares if it does not have the form $4*m*(8*k + 7)$.
- all integers can be written as the sum of 4 squares.
- if $k > 4$ then n can be partitioned and each partition can be written as a sum of 4 squares; if n is not evenly divisible by 4 then n can be written as a sum of squares only if the an additional partition can be written as sum of squares. For example, if $k = 6$ then n is partitioned into two parts, the first being written as a sum of 4 squares and the second being written as a sum of 2 squares - which can only be done if the condition above for $k = 2$ can be met, so this will automatically reject certain partitions of n .

Examples

```
>>> from sympy.solvers.diophantine.diophantine import sum_of_squares
>>> list(sum_of_squares(25, 2))
[(3, 4)]
>>> list(sum_of_squares(25, 2, True))
[(3, 4), (0, 5)]
>>> list(sum_of_squares(25, 4))
[(1, 2, 2, 4)]
```

See also:

[`sympy.utilities.iterables.signed\_permutations`](#) (page 2094)

merge_solution

`sympy.solvers.diophantine.diophantine.merge_solution(var, var_t, solution)`

This is used to construct the full solution from the solutions of sub equations.

Explanation

For example when solving the equation $(x - y)(x^2 + y^2 - z^2) = 0$, solutions for each of the equations $x - y = 0$ and $x^2 + y^2 - z^2$ are found independently. Solutions for $x - y = 0$ are $(x, y) = (t, t)$. But we should introduce a value for z when we output the solution for the original equation. This function converts (t, t) into (t, t, n_1) where n_1 is an integer parameter.

divisible

`sympy.solvers.diophantine.diophantine.divisible(a, b)`

Returns *True* if a is divisible by b and *False* otherwise.

PQa

`sympy.solvers.diophantine.diophantine.PQa(P_0, Q_0, D)`

Returns useful information needed to solve the Pell equation.

Explanation

There are six sequences of integers defined related to the continued fraction representation of $\frac{P}{Q} + \sqrt{D}Q$, namely $\{P_i\}$, $\{Q_i\}$, $\{a_i\}$, $\{A_i\}$, $\{B_i\}$, $\{G_i\}$. `PQa()` Returns these values as a 6-tuple in the same order as mentioned above. Refer [R777] for more detailed information.

Usage

`PQa(P_0, Q_0, D)`: P_0 , Q_0 and D are integers corresponding to P_0 , Q_0 and D in the continued fraction $\frac{P_0}{Q_0} + \sqrt{D}Q_0$. Also it's assumed that $P_0^2 \equiv D \pmod{|Q_0|}$ and D is square free.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import PQa
>>> pqa = PQa(13, 4, 5) # (13 + sqrt(5))/4
>>> next(pqa) # (P_0, Q_0, a_0, A_0, B_0, G_0)
(13, 4, 3, 3, 1, -1)
>>> next(pqa) # (P_1, Q_1, a_1, A_1, B_1, G_1)
(-1, 1, 1, 4, 1, 3)
```

References

[R777]

equivalent

`sympy.solvers.diophantine.diophantine.equivalent(u, v, r, s, D, N)`

Returns True if two solutions (u, v) and (r, s) of $x^2 - Dy^2 = N$ belongs to the same equivalence class and False otherwise.

Explanation

Two solutions (u, v) and (r, s) to the above equation fall to the same equivalence class iff both $(ur - Dvs)$ and $(us - vr)$ are divisible by N . See reference [R778]. No test is performed to test whether (u, v) and (r, s) are actually solutions to the equation. User should take care of this.

Usage

`equivalent(u, v, r, s, D, N)`: (u, v) and (r, s) are two solutions of the equation $x^2 - Dy^2 = N$ and all parameters involved are integers.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import equivalent
>>> equivalent(18, 5, -18, -5, 13, -1)
True
>>> equivalent(3, 1, -18, 393, 109, -4)
False
```

References

[R778]

parametrize_ternary_quadratic

`sympy.solvers.diophantine.diophantine.parametrize_ternary_quadratic(eq)`

Returns the parametrized general solution for the ternary quadratic equation `eq` which has the form $ax^2 + by^2 + cz^2 + fxy + gyz + hzx = 0$.

Examples

```
>>> from sympy import Tuple, ordered
>>> from sympy.abc import x, y, z
>>> from sympy.solvers.diophantine.diophantine import parametrize_
    ↳ ternary_quadratic
```

The parametrized solution may be returned with three parameters:

```
>>> parametrize_ternary_quadratic(2*x**2 + y**2 - 2*z**2)
(p**2 - 2*q**2, -2*p**2 + 4*p*q - 4*p*r - 4*q**2, p**2 - 4*p*q + 2*q**2 -
    ↳ 4*q*r)
```

There might also be only two parameters:

```
>>> parametrize_ternary_quadratic(4*x**2 + 2*y**2 - 3*z**2)
(2*p**2 - 3*q**2, -4*p**2 + 12*p*q - 6*q**2, 4*p**2 - 8*p*q + 6*q**2)
```

Notes

Consider p and q in the previous 2-parameter solution and observe that more than one solution can be represented by a given pair of parameters. If p and q are not coprime, this is trivially true since the common factor will also be a common factor of the solution values. But it may also be true even when p and q are coprime:

```
>>> sol = Tuple(*_)
>>> p, q = ordered(sol.free_symbols)
>>> sol.subs([(p, 3), (q, 2)])
(6, 12, 12)
>>> sol.subs([(q, 1), (p, 1)])
(-1, 2, 2)
>>> sol.subs([(q, 0), (p, 1)])
(2, -4, 4)
>>> sol.subs([(q, 1), (p, 0)])
(-3, -6, 6)
```

Except for sign and a common factor, these are equivalent to the solution of (1, 2, 2).

References

[R779]

diop_ternary_quadratic_normal

`sympy.solvers.diophantine.diophantine.diop_ternary_quadratic_normal(eq, parameterize=False)`

Solves the quadratic ternary diophantine equation, $ax^2 + by^2 + cz^2 = 0$.

Explanation

Here the coefficients a , b , and c should be non zero. Otherwise the equation will be a quadratic binary or univariate equation. If solvable, returns a tuple (x, y, z) that satisfies the given equation. If the equation does not have integer solutions, $(None, None, None)$ is returned.

Usage

`diop_ternary_quadratic_normal(eq)`: where `eq` is an equation of the form $ax^2 + by^2 + cz^2 = 0$.

Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.solvers.diophantine.diophantine import diop_ternary_
    quadratic_normal
>>> diop_ternary_quadratic_normal(x**2 + 3*y**2 - z**2)
(1, 0, 1)
>>> diop_ternary_quadratic_normal(4*x**2 + 5*y**2 - z**2)
(1, 0, 2)
>>> diop_ternary_quadratic_normal(34*x**2 - 3*y**2 - 301*z**2)
(4, 9, 1)
```

ldescent

`sympy.solvers.diophantine.diophantine.ldescent(A, B)`

Return a non-trivial solution to $w^2 = Ax^2 + By^2$ using Lagrange's method; return `None` if there is no such solution. .

Here, $A \neq 0$ and $B \neq 0$ and A and B are square free. Output a tuple (w_0, x_0, y_0) which is a solution to the above equation.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import ldescent
>>> ldescent(1, 1) # w^2 = x^2 + y^2
(1, 1, 0)
>>> ldescent(4, -7) # w^2 = 4x^2 - 7y^2
(2, -1, 0)
```

This means that $x = -1, y = 0$ and $w = 2$ is a solution to the equation $w^2 = 4x^2 - 7y^2$

```
>>> ldescent(5, -1) # w^2 = 5x^2 - y^2
(2, 1, -1)
```

References

[R780], [R781]

gaussian_reduce

`sympy.solvers.diophantine.diophantine.gaussian_reduce(w, a, b)`

Returns a reduced solution (x, z) to the congruence $X^2 - aZ^2 \equiv 0 \pmod{b}$ so that $x^2 + |a|z^2$ is minimal.

Details

Here w is a solution of the congruence $x^2 \equiv a \pmod{b}$

References

[R782], [R783]

holzer

`sympy.solvers.diophantine.diophantine.holzer(x, y, z, a, b, c)`

Simplify the solution (x, y, z) of the equation $ax^2 + by^2 = cz^2$ with $a, b, c > 0$ and $z^2 \geq |ab|$ to a new reduced solution (x', y', z') such that $z'^2 \leq |ab|$.

The algorithm is an interpretation of Mordell's reduction as described on page 8 of Cremona and Rusin's paper [R784] and the work of Mordell in reference [R785].

References

[R784], [R785]

prime_as_sum_of_two_squares

`sympy.solvers.diophantine.diophantine.prime_as_sum_of_two_squares(p)`

Represent a prime p as a unique sum of two squares; this can only be done if the prime is congruent to 1 mod 4.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import prime_as_sum_of_
    ↪ two_squares
>>> prime_as_sum_of_two_squares(7)  # can't be done
>>> prime_as_sum_of_two_squares(5)
(1, 2)
```

Reference

See also:

[`sum\_of\_squares`](#) (page 737)

sqf_normal

`sympy.solvers.diophantine.diophantine.sqf_normal(a, b, c, steps=False)`

Return a', b', c' , the coefficients of the square-free normal form of $ax^2 + by^2 + cz^2 = 0$, where a', b', c' are pairwise prime. If `steps` is True then also return three tuples: sq , sqf , and (a', b', c') where sq contains the square factors of a , b and c after removing the $\gcd(a, b, c)$; sqf contains the values of a , b and c after removing both the $\gcd(a, b, c)$ and the square factors.

The solutions for $ax^2 + by^2 + cz^2 = 0$ can be recovered from the solutions of $a'x^2 + b'y^2 + c'z^2 = 0$.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import sqf_normal
>>> sqf_normal(2 * 3**2 * 5, 2 * 5 * 11, 2 * 7**2 * 11)
(11, 1, 5)
>>> sqf_normal(2 * 3**2 * 5, 2 * 5 * 11, 2 * 7**2 * 11, True)
((3, 1, 7), (5, 55, 11), (11, 1, 5))
```

See also:

[`reconstruct`](#) (page 744)

References

[R787]

reconstruct

`sympy.solvers.diophantine.diophantine.reconstruct(A, B, z)`

Reconstruct the z value of an equivalent solution of $ax^2 + by^2 + cz^2$ from the z value of a solution of the square-free normal form of the equation, $a' * x^2 + b' * y^2 + c' * z^2$, where a' , b' and c' are square free and $\gcd(a', b', c') == 1$.

Internal Classes

These classes are intended for internal use in the Diophantine module.

DiophantineSolutionSet

class `sympy.solvers.diophantine.diophantine.DiophantineSolutionSet`(*symbols_seq*,
parameters)

Container for a set of solutions to a particular diophantine equation.

The base representation is a set of tuples representing each of the solutions.

Parameters

symbols : list

List of free symbols in the original equation.

parameters: list

List of parameters to be used in the solution.

Examples

Adding solutions:

```
>>> from sympy.solvers.diophantine.diophantine import DiophantineSolutionSet
>>> from sympy.abc import x, y, t, u
>>> s1 = DiophantineSolutionSet([x, y], [t, u])
>>> s1
set()
>>> s1.add((2, 3))
>>> s1.add((-1, u))
>>> s1
{(-1, u), (2, 3)}
>>> s2 = DiophantineSolutionSet([x, y], [t, u])
>>> s2.add((3, 4))
>>> s1.update(*s2)
>>> s1
{(-1, u), (2, 3), (3, 4)}
```


Conversion of solutions into dicts:

```
>>> list(s1.dict_iterator())
[{x: -1, y: u}, {x: 2, y: 3}, {x: 3, y: 4}]
```

Substituting values:

```
>>> s3 = DiophantineSolutionSet([x, y], [t, u])
>>> s3.add((t**2, t + u))
>>> s3
{(t**2, t + u)}
>>> s3.subs({t: 2, u: 3})
{(4, 5)}
>>> s3.subs(t, -1)
{(1, u - 1)}
>>> s3.subs(t, 3)
{(9, u + 3)}
```

Evaluation at specific values. Positional arguments are given in the same order as the parameters:

```
>>> s3(-2, 3)
{(4, 1)}
>>> s3(5)
{(25, u + 5)}
>>> s3(None, 2)
{(t**2, t + 2)}
```

DiophantineEquationType

class sympy.solvers.diophantine.diophantine.**DiophantineEquationType**(*equation*,
free_symbols=None)

Internal representation of a particular diophantine equation type.

Parameters

equation :

The diophantine equation that is being solved.

free_symbols : list (optional)

The symbols being solved for.

Attributes

total_degree :	The maximum of the degrees of all terms in the equation
homogeneous :	Does the equation contain a term of degree 0
homogeneous_order :	Does the equation contain any coefficient that is in the symbols being solved for
dimension :	The number of symbols being solved for

matches()

Determine whether the given equation can be matched to the particular equation type.

Univariate

class sympy.solvers.diophantine.diophantine.**Univariate**(equation,
free_symbols=None)

Representation of a univariate diophantine equation.

A univariate diophantine equation is an equation of the form $a_0 + a_1x + a_2x^2 + \dots + a_nx^n = 0$ where $a_1, a_2, \dots, a_n$ are integer constants and x is an integer variable.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import Univariate
>>> from sympy.abc import x
>>> Univariate((x - 2)*(x - 3)**2).solve() # solves equation (x - 2)*(x -
→ 3)**2 == 0
{(2,), (3,)}
```

Linear

class sympy.solvers.diophantine.diophantine.**Linear**(equation, free_symbols=None)
Representation of a linear diophantine equation.

A linear diophantine equation is an equation of the form $a_1x_1 + a_2x_2 + \dots + a_nx_n = 0$ where $a_1, a_2, \dots, a_n$ are integer constants and $x_1, x_2, \dots, x_n$ are integer variables.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import Linear
>>> from sympy.abc import x, y, z
>>> l1 = Linear(2*x - 3*y - 5)
>>> l1.matches() # is this equation linear
True
>>> l1.solve() # solves equation 2*x - 3*y - 5 == 0
{(3*t_0 - 5, 2*t_0 - 5)}
```

Here $x = -3*t_0 - 5$ and $y = -2*t_0 - 5$

```
>>> Linear(2*x - 3*y - 4*z - 3).solve()
{(t_0, 2*t_0 + 4*t_1 + 3, -t_0 - 3*t_1 - 3)}
```

BinaryQuadratic

class sympy.solvers.diophantine.diophantine.**BinaryQuadratic**(*equation*,
free_symbols=None)

Representation of a binary quadratic diophantine equation.

A binary quadratic diophantine equation is an equation of the form $Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$, where A, B, C, D, E, F are integer constants and x and y are integer variables.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy.solvers.diophantine.diophantine import BinaryQuadratic
>>> b1 = BinaryQuadratic(x**3 + y**2 + 1)
>>> b1.matches()
False
>>> b2 = BinaryQuadratic(x**2 + y**2 + 2*x + 2*y + 2)
>>> b2.matches()
True
>>> b2.solve()
{(-1, -1)}
```

References

[R788], [R789]

InhomogeneousTernaryQuadratic

class sympy.solvers.diophantine.diophantine.**InhomogeneousTernaryQuadratic**(*equation*,
free_symbols=None)

Representation of an inhomogeneous ternary quadratic.

No solver is currently implemented for this equation type.

HomogeneousTernaryQuadraticNormal

class sympy.solvers.diophantine.diophantine.**HomogeneousTernaryQuadraticNormal**(*equation*,
free_symbols=None)

Representation of a homogeneous ternary quadratic normal diophantine equation.

Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.solvers.diophantine.diophantine import HomogeneousTernaryQuadraticNormal
>>> HomogeneousTernaryQuadraticNormal(4*x**2 - 5*y**2 + z**2).solve()
{(1, 2, 4)}
```

HomogeneousTernaryQuadratic

class sympy.solvers.diophantine.diophantine.**HomogeneousTernaryQuadratic**(*equation*,
free_symbols=None)

Representation of a homogeneous ternary quadratic diophantine equation.

Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.solvers.diophantine.diophantine import HomogeneousTernaryQuadratic
>>> HomogeneousTernaryQuadratic(x**2 + y**2 - 3*z**2 + x*y).solve()
{(-1, 2, 1)}
>>> HomogeneousTernaryQuadratic(3*x**2 + y**2 - 3*z**2 + 5*x*y + y*z).  
    solve()  
{(3, 12, 13)}
```

InhomogeneousGeneralQuadratic

class sympy.solvers.diophantine.diophantine.**InhomogeneousGeneralQuadratic**(*equation*,
free_symbols=None)

Representation of an inhomogeneous general quadratic.

No solver is currently implemented for this equation type.

HomogeneousGeneralQuadratic

class sympy.solvers.diophantine.diophantine.**HomogeneousGeneralQuadratic**(*equation*,
free_symbols=None)

Representation of a homogeneous general quadratic.

No solver is currently implemented for this equation type.

GeneralSumOfSquares

class sympy.solvers.diophantine.diophantine.**GeneralSumOfSquares**(*equation*,
free_symbols=None)

Representation of the diophantine equation

$$x_1^2 + x_2^2 + \dots + x_n^2 - k = 0.$$

Details

When $n = 3$ if $k = 4^a(8m + 7)$ for some $a, m \in \mathbb{Z}$ then there will be no solutions. Refer [R790] for more details.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import GeneralSumOfSquares
>>> from sympy.abc import a, b, c, d, e
>>> GeneralSumOfSquares(a**2 + b**2 + c**2 + d**2 + e**2 - 2345).solve()
{(15, 22, 22, 24, 24)}
```

By default only 1 solution is returned. Use the *limit* keyword for more:

```
>>> sorted(GeneralSumOfSquares(a**2 + b**2 + c**2 + d**2 + e**2 - 2345).
↳ solve(limit=3))
[(15, 22, 22, 24, 24), (16, 19, 24, 24, 24), (16, 20, 22, 23, 26)]
```

References

[R790]

GeneralPythagorean

class sympy.solvers.diophantine.diophantine.**GeneralPythagorean**(*equation*,
free_symbols=None)

Representation of the general pythagorean equation, $a_1^2x_1^2 + a_2^2x_2^2 + \dots + a_n^2x_n^2 - a_{n+1}^2x_{n+1}^2 = 0$.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import GeneralPythagorean
>>> from sympy.abc import a, b, c, d, e, x, y, z, t
>>> GeneralPythagorean(a**2 + b**2 + c**2 - d**2).solve()
{(t_0**2 + t_1**2 - t_2**2, 2*t_0*t_2, 2*t_1*t_2, t_0**2 + t_1**2 + t_
↳ 2**2)}
>>> GeneralPythagorean(9*a**2 - 4*b**2 + 16*c**2 + 25*d**2 + e**2).
↳ solve(parameters=[x, y, z, t])
{(-10*t**2 + 10*x**2 + 10*y**2 + 10*z**2, 15*t**2 + 15*x**2 + 15*y**2 +
↳ 15*z**2, 15*t*x, 12*t*y, 60*t*z)}
```

CubicThue

class sympy.solvers.diophantine.diophantine.**CubicThue**(*equation*,
free_symbols=None)

Representation of a cubic Thue diophantine equation.

A cubic Thue diophantine equation is a polynomial of the form $f(x, y) = r$ of degree 3, where x and y are integers and r is a rational number.

No solver is currently implemented for this equation type.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy.solvers.diophantine.diophantine import CubicThue
>>> c1 = CubicThue(x**3 + y**2 + 1)
>>> c1.matches()
True
```

GeneralSumOfEvenPowers

class sympy.solvers.diophantine.diophantine.**GeneralSumOfEvenPowers**(*equation*,
free_symbols=None)

Representation of the diophantine equation

$$x_1^e + x_2^e + \dots + x_n^e - k = 0$$

where e is an even, integer power.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import _
↳ GeneralSumOfEvenPowers
>>> from sympy.abc import a, b
>>> GeneralSumOfEvenPowers(a**4 + b**4 - (2**4 + 3**4)).solve()
{(2, 3)}
```

Inequality Solvers

sympy.solvers.inequalities.**solve_rational_inequalities**(*eqs*)

Solve a system of rational inequalities with rational coefficients.

Examples

```
>>> from sympy.abc import x
>>> from sympy import solve_rational_inequalities, Poly
```

```
>>> solve_rational_inequalities([[
... ((Poly(-x + 1), Poly(1, x)), '>='),
... ((Poly(-x + 1), Poly(1, x)), '<=')]])
{1}
```

```
>>> solve_rational_inequalities([[
... ((Poly(x), Poly(1, x)), '!='),
... ((Poly(-x + 1), Poly(1, x)), '>=')]])
Union(Interval.open(-oo, 0), Interval.Lopen(0, 1))
```

See also:

[*solve_poly_inequality*](#) (page 751)

`sympy.solvers.inequalities.solve_poly_inequality(poly, rel)`

Solve a polynomial inequality with rational coefficients.

Examples

```
>>> from sympy import solve_poly_inequality, Poly
>>> from sympy.abc import x
```

```
>>> solve_poly_inequality(Poly(x, x, domain='ZZ'), '==')
[{0}]
```

```
>>> solve_poly_inequality(Poly(x**2 - 1, x, domain='ZZ'), '!=')
[Interval.open(-oo, -1), Interval.open(-1, 1), Interval.open(1, oo)]
```

```
>>> solve_poly_inequality(Poly(x**2 - 1, x, domain='ZZ'), '==')
[{-1}, {1}]
```

See also:

[*solve_poly_inequalities*](#) (page 751)

`sympy.solvers.inequalities.solve_poly_inequalities(polys)`

Solve polynomial inequalities with rational coefficients.

Examples

```
>>> from sympy import Poly
>>> from sympy.solvers.inequalities import solve_poly_inequalities
>>> from sympy.abc import x
>>> solve_poly_inequalities(((
... Poly(x**2 - 3), ">"), (
... Poly(-x**2 + 1), ">")))
Union(Interval.open(-oo, -sqrt(3)), Interval.open(-1, 1), Interval.
->open(sqrt(3), oo))
```

`sympy.solvers.inequalities.reduce_rational_inequalities(exprs, gen, relational=True)`

Reduce a system of rational inequalities with rational coefficients.

Examples

```
>>> from sympy import Symbol
>>> from sympy.solvers.inequalities import reduce_rational_inequalities
```

```
>>> x = Symbol('x', real=True)
```

```
>>> reduce_rational_inequalities([[x**2 <= 0]], x)
Eq(x, 0)
```

```
>>> reduce_rational_inequalities([[x + 2 > 0]], x)
-2 < x
>>> reduce_rational_inequalities([(x + 2, ">")], x)
-2 < x
>>> reduce_rational_inequalities([[x + 2]], x)
Eq(x, -2)
```

This function find the non-infinite solution set so if the unknown symbol is declared as extended real rather than real then the result may include finiteness conditions:

```
>>> y = Symbol('y', extended_real=True)
>>> reduce_rational_inequalities([y + 2 > 0], y)
(-2 < y) & (y < oo)
```

`sympy.solvers.inequalities.reduce_abs_inequality(expr, rel, gen)`

Reduce an inequality with nested absolute values.

Examples

```
>>> from sympy import reduce_abs_inequality, Abs, Symbol
>>> x = Symbol('x', real=True)
```

```
>>> reduce_abs_inequality(Abs(x - 5) - 3, '<', x)
(2 < x) & (x < 8)
```

```
>>> reduce_abs_inequality(Abs(x + 2)*3 - 13, '<', x)
(-19/3 < x) & (x < 7/3)
```

See also:

[*reduce_abs_inequalities*](#) (page 753)

`sympy.solvers.inequalities.reduce_abs_inequalities(exprs, gen)`

Reduce a system of inequalities with nested absolute values.

Examples

```
>>> from sympy import reduce_abs_inequalities, Abs, Symbol
>>> x = Symbol('x', extended_real=True)
```

```
>>> reduce_abs_inequalities([(Abs(3*x - 5) - 7, '<'),
... (Abs(x + 25) - 13, '>')], x)
(-2/3 < x) & (x < 4) & (((-oo < x) & (x < -38)) | ((-12 < x) & (x < oo)))
```

```
>>> reduce_abs_inequalities([(Abs(x - 4) + Abs(3*x - 5) - 7, '<')], x)
(1/2 < x) & (x < 4)
```

See also:

[*reduce_abs_inequality*](#) (page 752)

`sympy.solvers.inequalities.reduce_inequalities(inequalities, symbols=[])`

Reduce a system of inequalities with rational coefficients.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy import reduce_inequalities
```

```
>>> reduce_inequalities(0 <= x + 3, [])
(-3 <= x) & (x < oo)
```

```
>>> reduce_inequalities(0 <= x + y*2 - 1, [x])
(x < oo) & (x >= 1 - 2*y)
```

`sympy.solvers.inequalities.solve_univariate_inequality(expr, gen, relational=True, domain=Reals, continuous=False)`

Solves a real univariate inequality.

Parameters

expr : Relational

The target inequality

gen : Symbol

The variable for which the inequality is solved

relational : bool

A Relational type output is expected or not

domain : Set

The domain over which the equation is solved

continuous: bool

True if *expr* is known to be continuous over the given domain (and so `continuous_domain()` does not need to be called on it)

Raises

NotImplementedError

The solution of the inequality cannot be determined due to limitation in `sympy.solvers.solve.set.solve()` (page 869).

Notes

Currently, we cannot solve all the inequalities due to limitations in `sympy.solvers.solve.set.solve()` (page 869). Also, the solution returned for trigonometric inequalities are restricted in its periodic interval.

Examples

```
>>> from sympy import solve_univariate_inequality, Symbol, sin, Interval, S
>>> x = Symbol('x')
```

```
>>> solve_univariate_inequality(x**2 >= 4, x)
((2 <= x) & (x < oo)) | ((-oo < x) & (x <= -2))
```

```
>>> solve_univariate_inequality(x**2 >= 4, x, relational=False)
Union(Interval(-oo, -2), Interval(2, oo))
```

```
>>> domain = Interval(0, S.Infinity)
>>> solve_univariate_inequality(x**2 >= 4, x, False, domain)
Interval(2, oo)
```

```
>>> solve_univariate_inequality(sin(x) > 0, x, relational=False)
Interval.open(0, pi)
```

See also:

[*sympy.solvers.solveset.solvify*](#) (page 869)

solver returning solveset solutions with solve's output API

ODE

User Functions

These are functions that are imported into the global namespace with `from sympy import *`. These functions (unlike [Hint Functions](#) (page 766), below) are intended for use by ordinary users of SymPy.

dsolve

```
sympy.solvers.ode.dsolve(eq, func=None, hint='default', simplify=True, ics=None,
                          xi=None, eta=None, x0=0, n=6, **kwargs)
```

Solves any (supported) kind of ordinary differential equation and system of ordinary differential equations.

For Single Ordinary Differential Equation

It is classified under this when number of equation in eq is one. **Usage**

`dsolve(eq, f(x), hint)` -> Solve ordinary differential equation eq for function `f(x)`, using method `hint`.

Details

eq can be any supported ordinary differential equation (see the [ode](#) (page 755) docstring for supported methods). This can either be an [Equality](#) (page 1023), or an expression, which is assumed to be equal to 0.

f(x) is a function of one variable whose derivatives in that variable make up the ordinary differential equation eq. In many cases it is not necessary to provide this; it will be autodetected (and an error raised if it could not be detected).

hint is the solving method that you want dsolve to use. Use `classify_ode(eq, f(x))` to get all of the possible hints for an ODE. The default hint, `default`, will use whatever hint is returned first by [classify_ode\(\)](#) (page 760). See Hints below for more options that you can use for hint.

simplify enables simplification by `odesimp()` (page 767). See its docstring for more information. Turn this off, for example, to disable solving of solutions for `func` or simplification of arbitrary constants. It will still integrate with this hint. Note that the

solution may contain more arbitrary constants than the order of the ODE with this option enabled.

xi and eta are the infinitesimal functions of an ordinary

differential equation. They are the infinitesimals of the Lie group of point transformations for which the differential equation is invariant. The user can specify values for the infinitesimals. If nothing is specified, xi and eta are calculated using [infinitesimals\(\)](#) (page 764) with the help of various heuristics.

ics is the set of initial/boundary conditions for the differential equation.

It should be given in the form of `{f(x0): x1, f(x).diff(x).subs(x, x2): x3}` and so on. For power series solutions, if no initial conditions are specified `f(0)` is assumed to be `C0` and the power series solution is calculated about 0.

x0 is the point about which the power series solution of a differential equation is to be evaluated.

n gives the exponent of the dependent variable up to which the power series

solution of a differential equation is to be evaluated.

Hints

Aside from the various solving methods, there are also some meta-hints that you can pass to [dsolve\(\)](#) (page 755):

default:

This uses whatever hint is returned first by [classify_ode\(\)](#) (page 760). This is the default argument to [dsolve\(\)](#) (page 755).

all:

To make [dsolve\(\)](#) (page 755) apply all relevant classification hints, use `dsolve(ODE, func, hint="all")`. This will return a dictionary of hint:solution terms. If a hint causes `dsolve` to raise the `NotImplementedError`, value of that hint's key will be the exception object raised. The dictionary will also include some special keys:

- `order`: The order of the ODE. See also [ode_order\(\)](#) (page 852) in `deutils.py`.
- `best`: The simplest hint; what would be returned by `best` below.
- `best_hint`: The hint that would produce the solution given by `best`. If more than one hint produces the best solution, the first one in the tuple returned by [classify_ode\(\)](#) (page 760) is chosen.
- `default`: The solution that would be returned by default. This is the one produced by the hint that appears first in the tuple returned by [classify_ode\(\)](#) (page 760).

all_Integral:

This is the same as `all`, except if a hint also has a corresponding `_Integral` hint, it only returns the `_Integral` hint. This is useful if `all` causes [dsolve\(\)](#) (page 755) to hang because of a difficult or impossible integral. This meta-hint will also be much faster than `all`, because [integrate\(\)](#) (page 964) is an expensive routine.

best:

To have `dsolve()` (page 755) try all methods and return the simplest one. This takes into account whether the solution is solvable in the function, whether it contains any Integral classes (i.e. unevaluable integrals), and which one is the shortest in size.

See also the `classify_ode()` (page 760) docstring for more info on hints, and the `ode` (page 755) docstring for a list of all supported hints.

Tips

- You can declare the derivative of an unknown function this way:

```
>>> from sympy import Function, Derivative
>>> from sympy.abc import x # x is the independent variable
>>> f = Function("f")(x) # f is a function of x
>>> # f_ will be the derivative of f with respect to x
>>> f_ = Derivative(f, x)
```

- See `test_ode.py` for many tests, which serves also as a set of examples for how to use `dsolve()` (page 755).
- `dsolve()` (page 755) always returns an `Equality` (page 1023) class (except for the case when the hint is `all` or `all_Integral`). If possible, it solves the solution explicitly for the function being solved for. Otherwise, it returns an implicit solution.
- Arbitrary constants are symbols named `C1`, `C2`, and so on.
- Because all solutions should be mathematically equivalent, some hints may return the exact same result for an ODE. Often, though, two different hints will return the same solution formatted differently. The two should be equivalent. Also note that sometimes the values of the arbitrary constants in two different solutions may not be the same, because one constant may have “absorbed” other constants into it.
- Do `help(ode.ode_<hintname>)` to get help more information on a specific hint, where `<hintname>` is the name of a hint without `_Integral`.

For System Of Ordinary Differential Equations**Usage**

`dsolve(eq, func)` -> Solve a system of ordinary differential equations `eq` for `func` being list of functions including $x(t)$, $y(t)$, $z(t)$ where number of functions in the list depends upon the number of equations provided in `eq`.

Details

`eq` can be any supported system of ordinary differential equations This can either be an `Equality` (page 1023), or an expression, which is assumed to be equal to 0.

`func` holds $x(t)$ and $y(t)$ being functions of one variable which together with some of their derivatives make up the system of ordinary differential equation `eq`. It is not necessary to provide this; it will be autodetected (and an error raised if it could not be detected).

Hints

The hints are formed by parameters returned by `classify_sysode`, combining them give hints name used later for forming method name.

Examples

```
>>> from sympy import Function, dsolve, Eq, Derivative, sin, cos, symbols
>>> from sympy.abc import x
>>> f = Function('f')
>>> dsolve(Derivative(f(x), x, x) + 9*f(x), f(x))
Eq(f(x), C1*sin(3*x) + C2*cos(3*x))
```

```
>>> eq = sin(x)*cos(f(x)) + cos(x)*sin(f(x))*f(x).diff(x)
>>> dsolve(eq, hint='lst_exact')
[Eq(f(x), -acos(C1/cos(x)) + 2*pi), Eq(f(x), acos(C1/cos(x)))]
>>> dsolve(eq, hint='almost_linear')
[Eq(f(x), -acos(C1/cos(x)) + 2*pi), Eq(f(x), acos(C1/cos(x)))]
>>> t = symbols('t')
>>> x, y = symbols('x, y', cls=Function)
>>> eq = (Eq(Derivative(x(t), t), 12*t*x(t) + 8*y(t)), Eq(Derivative(y(t),
↳ t), 21*x(t) + 7*t*y(t)))
>>> dsolve(eq)
[Eq(x(t), C1*x0(t) + C2*x0(t)*Integral(8*exp(Integral(7*t,
↳ t))*exp(Integral(12*t, t))/x0(t)**2, t)),
Eq(y(t), C1*y0(t) + C2*(y0(t)*Integral(8*exp(Integral(7*t,
↳ t))*exp(Integral(12*t, t))/x0(t)**2, t) +
exp(Integral(7*t, t))*exp(Integral(12*t, t))/x0(t)))]
>>> eq = (Eq(Derivative(x(t), t), x(t)*y(t)*sin(t)), Eq(Derivative(y(t), t),
↳ y(t)**2*sin(t)))
>>> dsolve(eq)
{Eq(x(t), -exp(C1)/(C2*exp(C1) - cos(t))), Eq(y(t), -1/(C1 - cos(t)))}
```

dsolve_system

`sympy.solvers.ode.systems.dsolve_system(eqs, funcs=None, t=None, ics=None, doit=False, simplify=True)`

Solves any(supported) system of Ordinary Differential Equations

Parameters

eqs : List

system of ODEs to be solved

funcs : List or None

List of dependent variables that make up the system of ODEs

t : Symbol or None

Independent variable in the system of ODEs

ics : Dict or None

Set of initial boundary/conditions for the system of ODEs

doit : Boolean

Evaluate the solutions if True. Default value is True. Can be set to false if the integral evaluation takes too much time and/or is not required.

simplify: Boolean

Simplify the solutions for the systems. Default value is True. Can be set to false if simplification takes too much time and/or is not required.

Returns

List of List of Equations

Raises**NotImplementedError**

When the system of ODEs is not solvable by this function.

ValueError

When the parameters passed are not in the required form.

Explanation

This function takes a system of ODEs as an input, determines if the it is solvable by this function, and returns the solution if found any.

This function can handle: 1. Linear, First Order, Constant coefficient homogeneous system of ODEs 2. Linear, First Order, Constant coefficient non-homogeneous system of ODEs 3. Linear, First Order, non-constant coefficient homogeneous system of ODEs 4. Linear, First Order, non-constant coefficient non-homogeneous system of ODEs 5. Any implicit system which can be divided into system of ODEs which is of the above 4 forms 6. Any higher order linear system of ODEs that can be reduced to one of the 5 forms of systems described above.

The types of systems described above are not limited by the number of equations, i.e. this function can solve the above types irrespective of the number of equations in the system passed. But, the bigger the system, the more time it will take to solve the system.

This function returns a list of solutions. Each solution is a list of equations where LHS is the dependent variable and RHS is an expression in terms of the independent variable.

Among the non constant coefficient types, not all the systems are solvable by this function. Only those which have either a coefficient matrix with a commutative antiderivative or those systems which may be divided further so that the divided systems may have coefficient matrix with commutative antiderivative.

Examples

```
>>> from sympy import symbols, Eq, Function
>>> from sympy.solvers.ode.systems import dsolve_system
>>> f, g = symbols("f g", cls=Function)
>>> x = symbols("x")

>>> eqs = [Eq(f(x).diff(x), g(x)), Eq(g(x).diff(x), f(x))]
>>> dsolve_system(eqs)
[[Eq(f(x), -C1*exp(-x) + C2*exp(x)), Eq(g(x), C1*exp(-x) + C2*exp(x))]]
```

You can also pass the initial conditions for the system of ODEs:

```
>>> dsolve_system(eqs, ics={f(0): 1, g(0): 0})
[[Eq(f(x), exp(x)/2 + exp(-x)/2), Eq(g(x), exp(x)/2 - exp(-x)/2)]]
```

Optionally, you can pass the dependent variables and the independent variable for which the system is to be solved:

```
>>> funcs = [f(x), g(x)]
>>> dsolve_system(eqs, funcs=funcs, t=x)
[[Eq(f(x), -C1*exp(-x) + C2*exp(x)), Eq(g(x), C1*exp(-x) + C2*exp(x))]]
```

Lets look at an implicit system of ODEs:

```
>>> eqs = [Eq(f(x).diff(x)**2, g(x)**2), Eq(g(x).diff(x), g(x))]
>>> dsolve_system(eqs)
[[Eq(f(x), C1 - C2*exp(x)), Eq(g(x), C2*exp(x))], [Eq(f(x), C1 +
→ C2*exp(x)), Eq(g(x), C2*exp(x))]]
```

classify_ode

`sympy.solvers.ode.classify_ode(eq, func=None, dict=False, ics=None, *, prep=True, xi=None, eta=None, n=None, **kwargs)`

Returns a tuple of possible [dsolve\(\)](#) (page 755) classifications for an ODE.

The tuple is ordered so that first item is the classification that [dsolve\(\)](#) (page 755) uses to solve the ODE by default. In general, classifications at the near the beginning of the list will produce better solutions faster than those near the end, though there are always exceptions. To make [dsolve\(\)](#) (page 755) use a different classification, use `dsolve(ODE, func, hint=<classification>)`. See also the [dsolve\(\)](#) (page 755) docstring for different meta-hints you can use.

If `dict` is true, [classify_ode\(\)](#) (page 760) will return a dictionary of `hint:match` expression terms. This is intended for internal use by [dsolve\(\)](#) (page 755). Note that because dictionaries are ordered arbitrarily, this will most likely not be in the same order as the tuple.

You can get help on different hints by executing `help(ode.ode_hintname)`, where `hintname` is the name of the hint without `_Integral`.

See [allhints](#) (page 767) or the [ode](#) (page 755) docstring for a list of all supported hints that can be returned from [classify_ode\(\)](#) (page 760).

Notes

These are remarks on hint names.

`_Integral`

If a classification has `_Integral` at the end, it will return the expression with an unevaluated [Integral](#) (page 601) class in it. Note that a hint may do this anyway if [integrate\(\)](#) (page 964) cannot do the integral, though just using an `_Integral` will do so much faster. Indeed, an `_Integral` hint will always be faster than its corresponding hint without `_Integral` because [integrate\(\)](#) (page 964) is an expensive routine. If [dsolve\(\)](#) (page 755) hangs, it is probably

because `integrate()` (page 964) is hanging on a tough or impossible integral. Try using an `_Integral` hint or `all_Integral` to get it return something.

Note that some hints do not have `_Integral` counterparts. This is because `integrate()` (page 598) is not used in solving the ODE for those method. For example, `nth` order linear homogeneous ODEs with constant coefficients do not require integration to solve, so there is no `nth_linear_homogeneous_constant_coeff_Integrate` hint. You can easily evaluate any unevaluated `Integral` (page 601)s in an expression by doing `expr.doit()`.

Ordinals

Some hints contain an ordinal such as `1st_linear`. This is to help differentiate them from other hints, as well as from other methods that may not be implemented yet. If a hint has `nth` in it, such as the `nth_linear` hints, this means that the method used to applies to ODEs of any order.

indep and dep

Some hints contain the words `indep` or `dep`. These reference the independent variable and the dependent function, respectively. For example, if an ODE is in terms of $f(x)$, then `indep` will refer to x and `dep` will refer to f .

subs

If a hints has the word `subs` in it, it means that the ODE is solved by substituting the expression given after the word `subs` for a single dummy variable. This is usually in terms of `indep` and `dep` as above. The substituted expression will be written only in characters allowed for names of Python objects, meaning operators will be spelled out. For example, `indep/dep` will be written as `indep_div_dep`.

coeff

The word `coeff` in a hint refers to the coefficients of something in the ODE, usually of the derivative terms. See the docstring for the individual methods for more info (`help(ode)`). This is contrast to `coefficients`, as in `undetermined_coefficients`, which refers to the common name of a method.

_best

Methods that have more than one fundamental way to solve will have a hint for each sub-method and a `_best` meta-classification. This will evaluate all hints and return the best, using the same considerations as the normal `best` meta-hint.

Examples

```
>>> from sympy import Function, classify_ode, Eq
>>> from sympy.abc import x
>>> f = Function('f')
>>> classify_ode(Eq(f(x).diff(x), 0), f(x))
('nth_algebraic',
'separable',
'1st_exact',
'1st_linear',
```

(continues on next page)

(continued from previous page)

```
'Bernoulli',
'1st_homogeneous_coeff_best',
'1st_homogeneous_coeff_subs_indep_div_dep',
'1st_homogeneous_coeff_subs_dep_div_indep',
'1st_power_series', 'lie_group', 'nth_linear_constant_coeff_homogeneous',
'nth_linear_euler_eq_homogeneous',
'nth_algebraic_Integral', 'separable_Integral', '1st_exact_Integral',
'1st_linear_Integral', 'Bernoulli_Integral',
'1st_homogeneous_coeff_subs_indep_div_dep_Integral',
'1st_homogeneous_coeff_subs_dep_div_indep_Integral')
>>> classify_ode(f(x).diff(x, 2) + 3*f(x).diff(x) + 2*f(x) - 4)
('factorable', 'nth_linear_constant_coeff_undetermined_coefficients',
'nth_linear_constant_coeff_variation_of_parameters',
'nth_linear_constant_coeff_variation_of_parameters_Integral')
```

checkodesol

`sympy.solvers.ode.checkodesol(ode, sol, func=None, order='auto',
solve_for_func=True)`

Substitutes `sol` into `ode` and checks that the result is 0.

This works when `func` is one function, like $f(x)$ or a list of functions like $[f(x), g(x)]$ when `ode` is a system of ODEs. `sol` can be a single solution or a list of solutions. Each solution may be an [Equality](#) (page 1023) that the solution satisfies, e.g. `Eq(f(x), C1)`, `Eq(f(x) + C1, 0)`; or simply an [Expr](#) (page 947), e.g. `f(x) - C1`. In most cases it will not be necessary to explicitly identify the function, but if the function cannot be inferred from the original equation it can be supplied through the `func` argument.

If a sequence of solutions is passed, the same sort of container will be used to return the result for each solution.

It tries the following methods, in order, until it finds zero equivalence:

1. Substitute the solution for f in the original equation. This only works if `ode` is solved for f . It will attempt to solve it first unless `solve_for_func == False`.
2. Take n derivatives of the solution, where n is the order of `ode`, and check to see if that is equal to the solution. This only works on exact ODEs.
3. Take the 1st, 2nd, ..., n th derivatives of the solution, each time solving for the derivative of f of that order (this will always be possible because f is a linear operator). Then back substitute each derivative into `ode` in reverse order.

This function returns a tuple. The first item in the tuple is `True` if the substitution results in 0, and `False` otherwise. The second item in the tuple is what the substitution results in. It should always be 0 if the first item is `True`. Sometimes this function will return `False` even when an expression is identically equal to 0. This happens when [simplify\(\)](#) (page 661) does not reduce the expression to 0. If an expression returned by this function vanishes identically, then `sol` really is a solution to the `ode`.

If this function seems to hang, it is probably because of a hard simplification.

To use this function to test, test the first item of the tuple.

Examples

```
>>> from sympy import (Eq, Function, checkodesol, symbols,
...                     Derivative, exp)
>>> x, C1, C2 = symbols('x,C1,C2')
>>> f, g = symbols('f g', cls=Function)
>>> checkodesol(f(x).diff(x), Eq(f(x), C1))
(True, 0)
>>> assert checkodesol(f(x).diff(x), C1)[0]
>>> assert not checkodesol(f(x).diff(x), x)[0]
>>> checkodesol(f(x).diff(x, 2), x**2)
(False, 2)
```

```
>>> eqs = [Eq(Derivative(f(x), x), f(x)), Eq(Derivative(g(x), x), g(x))]
>>> sol = [Eq(f(x), C1*exp(x)), Eq(g(x), C2*exp(x))]
>>> checkodesol(eqs, sol)
(True, [0, 0])
```

homogeneous_order

`sympy.solvers.ode.homogeneous_order(eq, *symbols)`

Returns the order n if g is homogeneous and None if it is not homogeneous.

Determines if a function is homogeneous and if so of what order. A function $f(x, y, \dots)$ is homogeneous of order n if $f(tx, ty, \dots) = t^n f(x, y, \dots)$.

If the function is of two variables, $F(x, y)$, then f being homogeneous of any order is equivalent to being able to rewrite $F(x, y)$ as $G(x/y)$ or $H(y/x)$. This fact is used to solve 1st order ordinary differential equations whose coefficients are homogeneous of the same order (see the docstrings of [HomogeneousCoeffSubsDepDivIndep](#) (page 773) and [HomogeneousCoeffSubsIndepDivDep](#) (page 774)).

Symbols can be functions, but every argument of the function must be a symbol, and the arguments of the function that appear in the expression must match those given in the list of symbols. If a declared function appears with different arguments than given in the list of symbols, None is returned.

Examples

```
>>> from sympy import Function, homogeneous_order, sqrt
>>> from sympy.abc import x, y
>>> f = Function('f')
>>> homogeneous_order(f(x), f(x)) is None
True
>>> homogeneous_order(f(x,y), f(y, x), x, y) is None
True
>>> homogeneous_order(f(x), f(x), x)
1
>>> homogeneous_order(x**2*f(x)/sqrt(x**2+f(x)**2), x, f(x))
2
```

(continues on next page)

(continued from previous page)

```
>>> homogeneous_order(x**2+f(x), x, f(x)) is None
True
```

infinitesimals

`sympy.solvers.ode.infinitesimals(eq, func=None, order=None, hint='default', match=None)`

The infinitesimal functions of an ordinary differential equation, $\xi(x, y)$ and $\eta(x, y)$, are the infinitesimals of the Lie group of point transformations for which the differential equation is invariant. So, the ODE $y' = f(x, y)$ would admit a Lie group $x^* = X(x, y; \varepsilon) = x + \varepsilon \xi(x, y)$, $y^* = Y(x, y; \varepsilon) = y + \varepsilon \eta(x, y)$ such that $(y^*)' = f(x^*, y^*)$. A change of coordinates, to $r(x, y)$ and $s(x, y)$, can be performed so this Lie group becomes the translation group, $r^* = r$ and $s^* = s + \varepsilon$. They are tangents to the coordinate curves of the new system.

Consider the transformation $(x, y) \rightarrow (X, Y)$ such that the differential equation remains invariant. ξ and η are the tangents to the transformed coordinates X and Y , at $\varepsilon = 0$.

$$\left(\frac{\partial X(x, y; \varepsilon)}{\partial \varepsilon} \right) \Big|_{\varepsilon=0} = \xi, \left(\frac{\partial Y(x, y; \varepsilon)}{\partial \varepsilon} \right) \Big|_{\varepsilon=0} = \eta,$$

The infinitesimals can be found by solving the following PDE:

```
>>> from sympy import Function, Eq, pprint
>>> from sympy.abc import x, y
>>> xi, eta, h = map(Function, ['xi', 'eta', 'h'])
>>> h = h(x, y) # dy/dx = h
>>> eta = eta(x, y)
>>> xi = xi(x, y)
>>> genform = Eq(eta.diff(x) + (eta.diff(y) - xi.diff(x))*h
... - (xi.diff(y))*h**2 - xi*(h.diff(x)) - eta*(h.diff(y)), 0)
>>> pprint(genform)
/d          d          \          d          2
  ↳ d
|--(eta(x, y)) - --(xi(x, y))*h(x, y) - eta(x, y)*--(h(x, y)) - h (x, y)
  ↳ y)*--(x
\dy          dx          /          dy
  ↳ dy

          d          d
i(x, y)) - xi(x, y)*--(h(x, y)) + --(eta(x, y)) = 0
          dx          dx
```

Solving the above mentioned PDE is not trivial, and can be solved only by making intelligent assumptions for ξ and η (heuristics). Once an infinitesimal is found, the attempt to find more heuristics stops. This is done to optimise the speed of solving the differential equation. If a list of all the infinitesimals is needed, `hint` should be flagged as `all`, which gives the complete list of infinitesimals. If the infinitesimals for a particular heuristic needs to be found, it can be passed as a flag to `hint`.

Examples

```
>>> from sympy import Function
>>> from sympy.solvers.ode.lie_group import infinitesimals
>>> from sympy.abc import x
>>> f = Function('f')
>>> eq = f(x).diff(x) - x**2*f(x)
>>> infinitesimals(eq)
[{eta(x, f(x)): exp(x**3/3), xi(x, f(x)): 0}]
```

References

- Solving differential equations by Symmetry Groups, John Starrett, pp. 1 - pp. 14

checkinfsol

`sympy.solvers.ode.checkinfsol(eq, infinitesimals, func=None, order=None)`

This function is used to check if the given infinitesimals are the actual infinitesimals of the given first order differential equation. This method is specific to the Lie Group Solver of ODEs.

As of now, it simply checks, by substituting the infinitesimals in the partial differential equation.

$$\frac{\partial \eta}{\partial x} + \left(\frac{\partial \eta}{\partial y} - \frac{\partial \xi}{\partial x} \right) * h - \frac{\partial \xi}{\partial y} * h^2 - \xi \frac{\partial h}{\partial x} - \eta \frac{\partial h}{\partial y} = 0$$

where η , and ξ are the infinitesimals and $h(x, y) = \frac{dy}{dx}$

The infinitesimals should be given in the form of a list of dicts `[{xi(x, y): inf, eta(x, y): inf}]`, corresponding to the output of the function `infinitesimals`. It returns a list of values of the form `[(True/False, sol)]` where `sol` is the value obtained after substituting the infinitesimals in the PDE. If it is True, then `sol` would be 0.

constantsimp

`sympy.solvers.ode.constantsimp(expr, constants)`

Simplifies an expression with arbitrary constants in it.

This function is written specifically to work with `dsolve()` (page 755), and is not intended for general use.

Simplification is done by “absorbing” the arbitrary constants into other arbitrary constants, numbers, and symbols that they are not independent of.

The symbols must all have the same name with numbers after it, for example, C1, C2, C3. The symbolname here would be ‘C’, the startnumber would be 1, and the endnumber would be 3. If the arbitrary constants are independent of the variable x, then the independent symbol would be x. There is no need to specify the dependent function, such as f(x), because it already has the independent symbol, x, in it.

Because terms are “absorbed” into arbitrary constants and because constants are renumbered after simplifying, the arbitrary constants in `expr` are not necessarily equal to the ones of the same name in the returned result.

If two or more arbitrary constants are added, multiplied, or raised to the power of each other, they are first absorbed together into a single arbitrary constant. Then the new constant is combined into other terms if necessary.

Absorption of constants is done with limited assistance:

1. terms of [Add](#) (page 1013)s are collected to try join constants so $e^x(C_1 \cos(x) + C_2 \cos(x))$ will simplify to $e^x C_1 \cos(x)$;
2. powers with exponents that are [Add](#) (page 1013)s are expanded so e^{C_1+x} will be simplified to $C_1 e^x$.

Use `constant_renumber()` (page 768) to renumber constants after simplification or else arbitrary numbers on constants may appear, e.g. $C_1 + C_3 x$.

In rare cases, a single constant can be “simplified” into two constants. Every differential equation solution should have as many arbitrary constants as the order of the differential equation. The result here will be technically correct, but it may, for example, have C_1 and C_2 in an expression, when C_1 is actually equal to C_2 . Use your discretion in such situations, and also take advantage of the ability to use hints in `dsolve()` (page 755).

Examples

```
>>> from sympy import symbols
>>> from sympy.solvers.ode.ode import constantsimp
>>> C1, C2, C3, x, y = symbols('C1, C2, C3, x, y')
>>> constantsimp(2*C1*x, {C1, C2, C3})
C1*x
>>> constantsimp(C1 + 2 + x, {C1, C2, C3})
C1 + x
>>> constantsimp(C1*C2 + 2 + C2 + C3*x, {C1, C2, C3})
C1 + C3*x
```

Hint Functions

These functions are intended for internal use by `dsolve()` (page 755) and others. Unlike [User Functions](#) (page 755), above, these are not intended for every-day use by ordinary SymPy users. Instead, functions such as `dsolve()` (page 755) should be used. Nonetheless, these functions contain useful information in their docstrings on the various ODE solving methods. For this reason, they are documented here.

allhints

```
sympy.solvers.ode.allhints = ('factorable', 'nth_algebraic', 'separable',
'1st_exact', '1st_linear', 'Bernoulli', '1st_rational_riccati',
'Riccati_special_minus2', '1st_homogeneous_coeff_best',
'1st_homogeneous_coeff_subs_indep_div_dep',
'1st_homogeneous_coeff_subs_dep_div_indep', 'almost_linear',
'linear_coefficients', 'separable_reduced', '1st_power_series', 'lie_group',
'nth_linear_constant_coeff_homogeneous', 'nth_linear_euler_eq_homogeneous',
'nth_linear_constant_coeff_undetermined_coefficients',
'nth_linear_euler_eq_nonhomogeneous_undetermined_coefficients',
'nth_linear_constant_coeff_variation_of_parameters',
'nth_linear_euler_eq_nonhomogeneous_variation_of_parameters', 'Liouville',
'2nd_linear_airy', '2nd_linear_bessel', '2nd_hypergeometric',
'2nd_hypergeometric_Integral', 'nth_order_reducible',
'2nd_power_series_ordinary', '2nd_power_series_regular',
'nth_algebraic_Integral', 'separable_Integral', '1st_exact_Integral',
'1st_linear_Integral', 'Bernoulli_Integral',
'1st_homogeneous_coeff_subs_indep_div_dep_Integral',
'1st_homogeneous_coeff_subs_dep_div_indep_Integral', 'almost_linear_Integral',
'linear_coefficients_Integral', 'separable_reduced_Integral',
'nth_linear_constant_coeff_variation_of_parameters_Integral',
'nth_linear_euler_eq_nonhomogeneous_variation_of_parameters_Integral',
'Liouville_Integral', '2nd_nonlinear_autonomous_conserved',
'2nd_nonlinear_autonomous_conserved_Integral')
```

Built-in immutable sequence.

If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable's items.

If the argument is a tuple, the return value is the same object.

odesimp

```
sympy.solvers.ode.ode.odesimp(ode, eq, func, hint)
```

Simplifies solutions of ODEs, including trying to solve for func and running `constantsimp()` (page 765).

It may use knowledge of the type of solution that the hint returns to apply additional simplifications.

It also attempts to integrate any *Integral* (page 601)s in the expression, if the hint is not an `_Integral` hint.

This function should have no effect on expressions returned by `dsolve()` (page 755), as `dsolve()` (page 755) already calls `odesimp()` (page 767), but the individual hint functions do not call `odesimp()` (page 767) (because the `dsolve()` (page 755) wrapper does). Therefore, this function is designed for mainly internal use.

Examples

```
>>> from sympy import sin, symbols, dsolve, pprint, Function
>>> from sympy.solvers.ode.ode import odesimp
>>> x, u2, C1 = symbols('x,u2,C1')
>>> f = Function('f')
```

```
>>> eq = dsolve(x*f(x).diff(x) - f(x) - x*sin(f(x)/x), f(x),
... hint='1st_homogeneous_coeff_subs_indep_div_dep_Integral',
... simplify=False)
>>> pprint(eq, wrap_line=False)
```

$$\log(f(x)) = \log(C1) + \frac{x}{f(x)} \frac{1}{u1^2} \frac{\sin\left(\frac{1}{u1}\right)}{u1} d(u1)$$

```
>>> pprint(odesimp(eq, f(x), 1, {C1},
... hint='1st_homogeneous_coeff_subs_indep_div_dep'
... ))
x
----- = C1
/f(x)\
tan|----|
\2*x /
```

constant_renumber

`sympy.solvers.ode.ode.constant_renumber(expr, variables=None, newconstants=None)`

Renumber arbitrary constants in `expr` to use the symbol names as given in `newconstants`. In the process, this reorders expression terms in a standard way.

If `newconstants` is not provided then the new constant names will be `C1`, `C2` etc. Otherwise `newconstants` should be an iterable giving the new symbols to use for the constants in order.

The `variables` argument is a list of non-constant symbols. All other free symbols found in `expr` are assumed to be constants and will be renumbered. If `variables` is not given then any numbered symbol beginning with `C` (e.g. `C1`) is assumed to be a constant.

Symbols are renumbered based on `.sort_key()`, so they should be numbered roughly in the order that they appear in the final, printed expression. Note that this ordering is based in part on hashes, so it can produce different results on different machines.

The structure of this function is very similar to that of `constantsimp()` (page 765).

Examples

```
>>> from sympy import symbols
>>> from sympy.solvers.ode.ode import constant_renumber
>>> x, C1, C2, C3 = symbols('x,C1:4')
>>> expr = C3 + C2*x + C1*x**2
>>> expr
C1*x**2 + C2*x + C3
>>> constant_renumber(expr)
C1 + C2*x + C3*x**2
```

The variables argument specifies which are constants so that the other symbols will not be renumbered:

```
>>> constant_renumber(expr, [C1, x])
C1*x**2 + C2 + C3*x
```

The newconstants argument is used to specify what symbols to use when replacing the constants:

```
>>> constant_renumber(expr, [x], newconstants=symbols('E1:4'))
E1 + E2*x + E3*x**2
```

sol_simplicity

`sympy.solvers.ode.ode.ode_sol_simplicity(sol, func, trysolving=True)`

Returns an extended integer representing how simple a solution to an ODE is.

The following things are considered, in order from most simple to least:

- `sol` is solved for `func`.
- `sol` is not solved for `func`, but can be if passed to `solve` (e.g., a solution returned by `dsolve(ode, func, simplify=False)`).
- If `sol` is not solved for `func`, then base the result on the length of `sol`, as computed by `len(str(sol))`.
- If `sol` has any unevaluated *Integral* (page 601)s, this will automatically be considered less simple than any of the above.

This function returns an integer such that if solution A is simpler than solution B by above metric, then `ode_sol_simplicity(sola, func) < ode_sol_simplicity(solb, func)`.

Currently, the following are the numbers returned, but if the heuristic is ever improved, this may change. Only the ordering is guaranteed.

Simplicity	Return
<code>sol</code> solved for <code>func</code>	-2
<code>sol</code> not solved for <code>func</code> but can be	-1
<code>sol</code> is not solved nor solvable for <code>func</code>	<code>len(str(sol))</code>
<code>sol</code> contains an <i>Integral</i> (page 601)	oo

oo here means the SymPy infinity, which should compare greater than any integer.

If you already know `solve()` (page 836) cannot solve `sol`, you can use `trysolving=False` to skip that step, which is the only potentially slow step. For example, `dsolve()` (page 755) with the `simplify=False` flag should do this.

If `sol` is a list of solutions, if the worst solution in the list returns oo it returns that, otherwise it returns `len(str(sol))`, that is, the length of the string representation of the whole list.

Examples

This function is designed to be passed to `min` as the key argument, such as `min(listofsolutions, key=lambda i: ode_sol_simplicity(i, f(x)))`.

```
>>> from sympy import symbols, Function, Eq, tan, Integral
>>> from sympy.solvers.ode.ode import ode_sol_simplicity
>>> x, C1, C2 = symbols('x, C1, C2')
>>> f = Function('f')

>>> ode_sol_simplicity(Eq(f(x), C1*x**2), f(x))
-2
>>> ode_sol_simplicity(Eq(x**2 + f(x), C1), f(x))
-1
>>> ode_sol_simplicity(Eq(f(x), C1*Integral(2*x, x)), f(x))
oo
>>> eq1 = Eq(f(x)/tan(f(x)/(2*x)), C1)
>>> eq2 = Eq(f(x)/tan(f(x)/(2*x) + f(x)), C2)
>>> [ode_sol_simplicity(eq, f(x)) for eq in [eq1, eq2]]
[28, 35]
>>> min([eq1, eq2], key=lambda i: ode_sol_simplicity(i, f(x)))
Eq(f(x)/tan(f(x)/(2*x)), C1)
```

factorable

class `sympy.solvers.ode.single.Factorable(ode_problem)`

Solves equations having a solvable factor.

This function is used to solve the equation having factors. Factors may be of type algebraic or ode. It will try to solve each factor independently. Factors will be solved by calling `dsolve`. We will return the list of solutions.

Examples

```
>>> from sympy import Function, dsolve, pprint
>>> from sympy.abc import x
>>> f = Function('f')
>>> eq = (f(x)**2-4)*(f(x).diff(x)+f(x))
>>> pprint(dsolve(eq, f(x)))
[ $f(x) = 2$ ,  $f(x) = -2$ ,  $f(x) = C_1 e^{-x}$ ]
```

1st_exact

class sympy.solvers.ode.single.**FirstExact**(ode_problem)

Solves 1st order exact ordinary differential equations.

A 1st order differential equation is called exact if it is the total differential of a function. That is, the differential equation

$$P(x, y) \partial x + Q(x, y) \partial y = 0$$

is exact if there is some function $F(x, y)$ such that $P(x, y) = \partial F / \partial x$ and $Q(x, y) = \partial F / \partial y$. It can be shown that a necessary and sufficient condition for a first order ODE to be exact is that $\partial P / \partial y = \partial Q / \partial x$. Then, the solution will be as given below:

```
>>> from sympy import Function, Eq, Integral, symbols, pprint
>>> x, y, t, x0, y0, C1 = symbols('x,y,t,x0,y0,C1')
>>> P, Q, F = map(Function, ['P', 'Q', 'F'])
>>> pprint(Eq(Eq(F(x, y), Integral(P(t, y), (t, x0, x)) +
... Integral(Q(x0, t), (t, y0, y))), C1))
```

$$F(x, y) = \int_{x_0}^x P(t, y) dt + \int_{y_0}^y Q(x_0, t) dt = C_1$$

Where the first partials of P and Q exist and are continuous in a simply connected region.

A note: SymPy currently has no way to represent inert substitution on an expression, so the hint `1st_exact_Integral` will return an integral with dy . This is supposed to represent the function that you are solving for.

Examples

```
>>> from sympy import Function, dsolve, cos, sin
>>> from sympy.abc import x
>>> f = Function('f')
>>> dsolve(cos(f(x)) - (x*sin(f(x)) - f(x)**2)*f(x).diff(x),
... f(x), hint='1st_exact')
Eq(x*cos(f(x)) + f(x)**3/3, C1)
```

References

- https://en.wikipedia.org/wiki/Exact_differential_equation
- M. Tenenbaum & H. Pollard, “Ordinary Differential Equations”, Dover 1963, pp. 73

indirect doctest

1st_homogeneous_coeff_best

class sympy.solvers.ode.single.HomogeneousCoeffBest(ode_problem)

Returns the best solution to an ODE from the two hints 1st_homogeneous_coeff_subs_dep_div_indep and 1st_homogeneous_coeff_subs_indep_div_dep.

This is as determined by *ode_sol_simplicity()* (page 769).

See the *HomogeneousCoeffSubsIndepDivDep* (page 774) and *HomogeneousCoeffSubsDepDivIndep* (page 773) docstrings for more information on these hints. Note that there is no *ode_1st_homogeneous_coeff_best_Integral* hint.

Examples

```
>>> from sympy import Function, dsolve, pprint
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(2*x*f(x) + (x**2 + f(x)**2)*f(x).diff(x), f(x),
... hint='1st_homogeneous_coeff_best', simplify=False))
```

$$\log(f(x)) = \log(C1) - \frac{\log\left(\frac{3x}{\sqrt{f(x)}} + 1\right)}{3}$$

References

- https://en.wikipedia.org/wiki/Homogeneous_differential_equation
- M. Tenenbaum & H. Pollard, “Ordinary Differential Equations”, Dover 1963, pp. 59

indirect doctest

1st_homogeneous_coeff_subs_dep_div_indep

class `sympy.solvers.ode.single.HomogeneousCoeffSubsDepDivIndep(ode_problem)`

Solves a 1st order differential equation with homogeneous coefficients using the substitution $u_1 = \frac{\langle \text{dependent variable} \rangle}{\langle \text{independent variable} \rangle}$.

This is a differential equation

$$P(x, y) + Q(x, y)dy/dx = 0$$

such that P and Q are homogeneous and of the same order. A function $F(x, y)$ is homogeneous of order n if $F(xt, yt) = t^n F(x, y)$. Equivalently, $F(x, y)$ can be rewritten as $G(y/x)$ or $H(x/y)$. See also the docstring of [homogeneous_order\(\)](#) (page 763).

If the coefficients P and Q in the differential equation above are homogeneous functions of the same order, then it can be shown that the substitution $y = u_1 x$ (i.e. $u_1 = y/x$) will turn the differential equation into an equation separable in the variables x and u . If $h(u_1)$ is the function that results from making the substitution $u_1 = f(x)/x$ on $P(x, f(x))$ and $g(u_2)$ is the function that results from the substitution on $Q(x, f(x))$ in the differential equation $P(x, f(x)) + Q(x, f(x))f'(x) = 0$, then the general solution is:

```
>>> from sympy import Function, dsolve, pprint
>>> from sympy.abc import x
>>> f, g, h = map(Function, ['f', 'g', 'h'])
>>> genform = g(f(x)/x) + h(f(x)/x)*f(x).diff(x)
>>> pprint(genform)
      /f(x)\      /f(x)\ d
g|-----| + h|-----|*--(f(x))
 \ x /      \ x / dx
>>> pprint(dsolve(genform, f(x),
... hint='1st_homogeneous_coeff_subs_dep_div_indep_Integral'))
      f(x)
      ----
      x
      /
      |
      |          -h(u1)
      |          ----- d(u1)
      |          u1*h(u1) + g(u1)
log(x) = C1 +  -----
      /
```

Where $u_1 h(u_1) + g(u_1) \neq 0$ and $x \neq 0$.

See also the docstrings of [HomogeneousCoeffBest](#) (page 772) and [HomogeneousCoeffSubsIndepDivDep](#) (page 774).

Examples

```
>>> from sympy import Function, dsolve
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(2*x*f(x) + (x**2 + f(x)**2)*f(x).diff(x), f(x),
... hint='1st_homogeneous_coeff_subs_dep_div_indep', simplify=False))
```

$$\log(x) = \log(C1) - \frac{\log\left(\frac{3f(x)}{x} + \frac{f^3(x)}{x^3}\right)}{3}$$

References

- https://en.wikipedia.org/wiki/Homogeneous_differential_equation
- M. Tenenbaum & H. Pollard, “Ordinary Differential Equations”, Dover 1963, pp. 59

indirect doctest

1st_homogeneous_coeff_subs_indep_div_dep

class sympy.solvers.ode.single.HomogeneousCoeffSubsIndepDivDep(*ode_problem*)

Solves a 1st order differential equation with homogeneous coefficients using the substitution $u_2 = \frac{\text{<independent variable>}}{\text{<dependent variable>}}$.

This is a differential equation

$$P(x, y) + Q(x, y)dy/dx = 0$$

such that P and Q are homogeneous and of the same order. A function $F(x, y)$ is homogeneous of order n if $F(xt, yt) = t^n F(x, y)$. Equivalently, $F(x, y)$ can be rewritten as $G(y/x)$ or $H(x/y)$. See also the docstring of [homogeneous_order\(\)](#) (page 763).

If the coefficients P and Q in the differential equation above are homogeneous functions of the same order, then it can be shown that the substitution $x = u_2 y$ (i.e. $u_2 = x/y$) will turn the differential equation into an equation separable in the variables y and u_2 . If $h(u_2)$ is the function that results from making the substitution $u_2 = x/f(x)$ on $P(x, f(x))$ and $g(u_2)$ is the function that results from the substitution on $Q(x, f(x))$ in the differential equation $P(x, f(x)) + Q(x, f(x))f'(x) = 0$, then the general solution is:

```
>>> from sympy import Function, dsolve, pprint
>>> from sympy.abc import x
>>> f, g, h = map(Function, ['f', 'g', 'h'])
>>> genform = g(x/f(x)) + h(x/f(x))*f(x).diff(x)
>>> pprint(genform)
```

$$g\left|\frac{x}{f(x)}\right| + h\left|\frac{x}{f(x)}\right| * f'(x)$$

(continues on next page)

(continued from previous page)

```

\ f(x) /      \ f(x) / dx
>>> pprint(dsolve(genform, f(x),
... hint='lst_homogeneous_coeff_subs_indep_div_dep_Integral'))
      x
      ----
      f(x)
      /
      |
      |      -g(u1)
      |      ----- d(u1)
      |      u1*g(u1) + h(u1)
      |
      /

f(x) = C1*e

```

Where $u_1 g(u_1) + h(u_1) \neq 0$ and $f(x) \neq 0$.

See also the docstrings of [HomogeneousCoeffBest](#) (page 772) and [HomogeneousCoeffSubsDepDivIndep](#) (page 773).

Examples

```

>>> from sympy import Function, pprint, dsolve
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(2*x*f(x) + (x**2 + f(x)**2)*f(x).diff(x), f(x),
... hint='lst_homogeneous_coeff_subs_indep_div_dep',
... simplify=False))
      /      2      \
      | 3*x      |
log|----- + 1|
      | 2      |
      \f (x)    /
log(f(x)) = log(C1) - -----
                        3

```

References

- https://en.wikipedia.org/wiki/Homogeneous_differential_equation
- M. Tenenbaum & H. Pollard, "Ordinary Differential Equations", Dover 1963, pp. 59

indirect doctest

1st_rational_riccati

class sympy.solvers.ode.single.**RationalRiccati**(ode_problem)

Gives general solutions to the first order Riccati differential equations that have atleast one rational particular solution.

$$y' = b_0(x) + b_1(x)y + b_2(x)y^2$$

where b_0 , b_1 and b_2 are rational functions of x with $b_2 \neq 0$ ($b_2 = 0$ would make it a Bernoulli equation).

Examples

```
>>> from sympy import Symbol, Function, dsolve, checkodesol
>>> f = Function('f')
>>> x = Symbol('x')
```

```
>>> eq = -x**4*f(x)**2 + x**3*f(x).diff(x) + x**2*f(x) + 20
>>> sol = dsolve(eq, hint="1st_rational_riccati")
>>> sol
Eq(f(x), (4*C1 - 5*x**9 - 4)/(x**2*(C1 + x**9 - 1)))
>>> checkodesol(eq, sol)
(True, 0)
```

References

- Riccati ODE: https://en.wikipedia.org/wiki/Riccati_equation
- N. Thieu Vo - Rational and Algebraic Solutions of First-Order Algebraic ODEs: Algorithm 11, pp. 78 - https://www3.risc.jku.at/publications/download/risc_5387/PhDThesisThieu.pdf

2nd_linear_airy

class sympy.solvers.ode.single.**SecondLinearAiry**(ode_problem)

Gives solution of the Airy differential equation

$$\frac{d^2y}{dx^2} + (a + bx)y(x) = 0$$

in terms of Airy special functions `airyai` and `airybi`.

Examples

```
>>> from sympy import dsolve, Function
>>> from sympy.abc import x
>>> f = Function("f")
>>> eq = f(x).diff(x, 2) - x*f(x)
>>> dsolve(eq)
Eq(f(x), C1*airyai(x) + C2*airybi(x))
```

2nd_linear_bessel

class sympy.solvers.ode.single.**SecondLinearBessel**(ode_problem)

Gives solution of the Bessel differential equation

$$x^2 \frac{d^2 y}{dx^2} + x \frac{dy}{dx} y(x) + (x^2 - n^2) y(x)$$

if n is integer then the solution is of the form $\text{Eq}(f(x), C0 \text{besselj}(n,x) + C1 \text{bessely}(n,x))$ as both the solutions are linearly independent else if n is a fraction then the solution is of the form $\text{Eq}(f(x), C0 \text{besselj}(n,x) + C1 \text{besselj}(-n,x))$ which can also transform into $\text{Eq}(f(x), C0 \text{besselj}(n,x) + C1 \text{bessely}(n,x))$.

Examples

```
>>> from sympy.abc import x
>>> from sympy import Symbol
>>> v = Symbol('v', positive=True)
>>> from sympy import dsolve, Function
>>> f = Function('f')
>>> y = f(x)
>>> genform = x**2*y.diff(x, 2) + x*y.diff(x) + (x**2 - v**2)*y
>>> dsolve(genform)
Eq(f(x), C1*besselj(v, x) + C2*bessely(v, x))
```

References

<https://www.math24.net/bessel-differential-equation/>

Bernoulli

class sympy.solvers.ode.single.**Bernoulli**(ode_problem)

Solves Bernoulli differential equations.

These are equations of the form

$$dy/dx + P(x)y = Q(x)y^n, n \neq 1.$$

The substitution $w = 1/y^{1-n}$ will transform an equation of this form into one that is linear (see the docstring of *FirstLinear* (page 776)). The general solution is:

```

>>> from sympy import Function, dsolve, Eq, pprint
>>> from sympy.abc import x, n
>>> f, P, Q = map(Function, ['f', 'P', 'Q'])
>>> genform = Eq(f(x).diff(x) + P(x)*f(x), Q(x)*f(x)**n)
>>> pprint(genform)

```

$$P(x)f(x) + \frac{d}{dx}(f(x)) = Q(x)f(x)^n$$

```

>>> pprint(dsolve(genform, f(x), hint='Bernoulli_Integral'), num_
columns=110)

```

$$f(x) = \frac{C_1 - n \int \frac{P(x)}{Q(x)} dx + \int \frac{(-P(x) + Q(x))}{Q(x)^{n-1}} dx}{\int \frac{1}{Q(x)^{n-1}} dx}$$

Note that the equation is separable when $n = 1$ (see the docstring of [Separable](#) (page 789)).

```

>>> pprint(dsolve(Eq(f(x).diff(x) + P(x)*f(x), Q(x)*f(x)), f(x),
... hint='separable_Integral'))
f(x)

```

$$\frac{1}{y} dy = C_1 + \int (-P(x) + Q(x)) dx$$

Examples

```
>>> from sympy import Function, dsolve, Eq, pprint, log
>>> from sympy.abc import x
>>> f = Function('f')
```

```
>>> pprint(dsolve(Eq(x*f(x).diff(x) + f(x), log(x)*f(x)**2),
... f(x), hint='Bernoulli'))
f(x) = 
$$\frac{1}{C1*x + \log(x) + 1}$$

```

References

- https://en.wikipedia.org/wiki/Bernoulli_differential_equation
- M. Tenenbaum & H. Pollard, “Ordinary Differential Equations”, Dover 1963, pp. 95

indirect doctest

Liouville

class sympy.solvers.ode.single.**Liouville**(ode_problem)

Solves 2nd order Liouville differential equations.

The general form of a Liouville ODE is

$$\frac{d^2 y}{dx^2} + g(y) \left(\frac{dy}{dx} \right)^2 + h(x) \frac{dy}{dx}.$$

The general solution is:

```
>>> from sympy import Function, dsolve, Eq, pprint, diff
>>> from sympy.abc import x
>>> f, g, h = map(Function, ['f', 'g', 'h'])
>>> genform = Eq(diff(f(x), x, x) + g(f(x))*diff(f(x), x)**2 +
... h(x)*diff(f(x), x), 0)
>>> pprint(genform)

$$g(f(x)) \left| \frac{d}{dx} \right|^2 (f(x)) + h(x) \left| \frac{d}{dx} \right| (f(x)) + \frac{d^2}{dx^2} (f(x)) = 0$$

>>> pprint(dsolve(genform, f(x), hint='Liouville_Integral'))

$$\int \frac{1}{\left| \frac{d}{dx} \right| (f(x))} - \int \frac{h(x)}{\left| \frac{d}{dx} \right| (f(x))} dx = \int \frac{1}{\left| \frac{d}{dx} \right| (f(x))} g(y) dy$$

```

(continues on next page)

(continued from previous page)

$$C1 + C2 \frac{e^{\int dx}}{e^{\int \frac{1}{x} dx}} = 0$$

Examples

```
>>> from sympy import Function, dsolve, Eq, pprint
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(diff(f(x), x, x) + diff(f(x), x)**2/f(x) +
... diff(f(x), x)/x, f(x), hint='Liouville'))
```

$$[f(x) = -\sqrt{C1 + C2 \log(x)}, f(x) = \sqrt{C1 + C2 \log(x)}]$$

References

- Goldstein and Braun, “Advanced Methods for the Solution of Differential Equations”, pp. 98
- <http://www.maplesoft.com/support/help/Maple/view.aspx?path=odeadvisor/Liouville>

indirect doctest

Riccati_special_minus2

class sympy.solvers.ode.single.**RiccatiSpecial**(ode_problem)

The general Riccati equation has the form

$$dy/dx = f(x)y^2 + g(x)y + h(x).$$

While it does not have a general solution [1], the “special” form, $dy/dx = ay^2 - bx^c$, does have solutions in many cases [2]. This routine returns a solution for $a(dy/dx) = by^2 + cy/x + d/x^2$ that is obtained by using a suitable change of variables to reduce it to the special form and is valid when neither a nor b are zero and either c or d is zero.

```
>>> from sympy.abc import x, a, b, c, d
>>> from sympy import dsolve, checkodesol, pprint, Function
>>> f = Function('f')
>>> y = f(x)
>>> genform = a*y.diff(x) - (b*y**2 + c*y/x + d/x**2)
>>> sol = dsolve(genform, y, hint="Riccati_special_minus2")
>>> pprint(sol, wrap_line=False)
```

$$\frac{y}{\sqrt{C1 + C2 \log(x)}} = \frac{\sqrt{C1 + C2 \log(x)}}{x^2}$$

(continues on next page)

(continued from previous page)

$$f(x) = \frac{\log(x) \sqrt{-a + c - \sqrt{4bd - (a + c)^2}} \tan\left(C_1 + \frac{\sqrt{2a}}{2bx}\right)}{2bx}$$

```
>>> checkodesol(genform, sol, order=1)[0]
True
```

References

- <http://www.maplesoft.com/support/help/Maple/view.aspx?path=odeadvisor/Riccati>
- <http://eqworld.ipmnet.ru/en/solutions/ode/ode0106.pdf> - <http://eqworld.ipmnet.ru/en/solutions/ode/ode0123.pdf>

nth_linear_constant_coeff_homogeneous

class `sympy.solvers.ode.single.NthLinearConstantCoeffHomogeneous(ode_problem)`

Solves an n th order linear homogeneous differential equation with constant coefficients.

This is an equation of the form

$$a_n f^{(n)}(x) + a_{n-1} f^{(n-1)}(x) + \cdots + a_1 f'(x) + a_0 f(x) = 0.$$

These equations can be solved in a general manner, by taking the roots of the characteristic equation $a_n m^n + a_{n-1} m^{n-1} + \cdots + a_1 m + a_0 = 0$. The solution will then be the sum of $C_n x^i e^{rx}$ terms, for each where C_n is an arbitrary constant, r is a root of the characteristic equation and i is one of each from 0 to the multiplicity of the root - 1 (for example, a root 3 of multiplicity 2 would create the terms $C_1 e^{3x} + C_2 x e^{3x}$). The exponential is usually expanded for complex roots using Euler's equation $e^{Ix} = \cos(x) + I \sin(x)$. Complex roots always come in conjugate pairs in polynomials with real coefficients, so the two roots will be represented (after simplifying the constants) as $e^{ax} (C_1 \cos(bx) + C_2 \sin(bx))$.

If SymPy cannot find exact roots to the characteristic equation, a [ComplexRootOf](#) (page 2431) instance will be return instead.

```
>>> from sympy import Function, dsolve
>>> from sympy.abc import x
>>> f = Function('f')
>>> dsolve(f(x).diff(x, 5) + 10*f(x).diff(x) - 2*f(x), f(x),
... hint='nth_linear_constant_coeff_homogeneous')
...
Eq(f(x), C5*exp(x*RootOf(_x**5 + 10*_x - 2, 0))
+ (C1*sin(x*im(RootOf(_x**5 + 10*_x - 2, 1)))
+ C2*cos(x*im(RootOf(_x**5 + 10*_x - 2, 1))))*exp(x*re(RootOf(_x**5 +
```

(continues on next page)

(continued from previous page)

```

→10*_x - 2, 1)))
+ (C3*sin(x*im(CRootOf(_x**5 + 10*_x - 2, 3)))
+ C4*cos(x*im(CRootOf(_x**5 + 10*_x - 2, 3))))*exp(x*re(CRootOf(_x**5 +
→10*_x - 2, 3)))

```

Note that because this method does not involve integration, there is no `nth_linear_constant_coeff_homogeneous_Integral` hint.

Examples

```

>>> from sympy import Function, dsolve, pprint
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(f(x).diff(x, 4) + 2*f(x).diff(x, 3) -
... 2*f(x).diff(x, 2) - 6*f(x).diff(x) + 5*f(x), f(x),
... hint='nth_linear_constant_coeff_homogeneous'))
              x
f(x) = (C1 + C2*x)*e  + (C3*sin(x) + C4*cos(x))*e
              -2*x

```

References

- https://en.wikipedia.org/wiki/Linear_differential_equation section: Nonhomogeneous equation with constant coefficients
- M. Tenenbaum & H. Pollard, "Ordinary Differential Equations", Dover 1963, pp. 211

indirect doctest

nth_linear_constant_coeff_undetermined_coefficients

class `sympy.solvers.ode.single.NthLinearConstantCoeffUndeterminedCoefficients(ode_problem)`

Solves an n th order linear differential equation with constant coefficients using the method of undetermined coefficients.

This method works on differential equations of the form

$$a_n f^{(n)}(x) + a_{n-1} f^{(n-1)}(x) + \cdots + a_1 f'(x) + a_0 f(x) = P(x),$$

where $P(x)$ is a function that has a finite number of linearly independent derivatives.

Functions that fit this requirement are finite sums functions of the form $ax^i e^{bx} \sin(cx + d)$ or $ax^i e^{bx} \cos(cx + d)$, where i is a non-negative integer and a , b , c , and d are constants. For example any polynomial in x , functions like $x^2 e^{2x}$, $x \sin(x)$, and $e^x \cos(x)$ can all be used. Products of sin's and cos's have a finite number of derivatives, because they can be expanded into $\sin(ax)$ and $\cos(bx)$ terms. However, SymPy currently cannot do that expansion, so you will need to manually rewrite the expression in terms of the above to use this method. So, for example, you will need to manually convert $\sin^2(x)$ into $(1 + \cos(2x))/2$ to properly apply the method of undetermined coefficients on it.

This method works by creating a trial function from the expression and all of its linear independent derivatives and substituting them into the original ODE. The coefficients for

each term will be a system of linear equations, which are be solved for and substituted, giving the solution. If any of the trial functions are linearly dependent on the solution to the homogeneous equation, they are multiplied by sufficient x to make them linearly independent.

Examples

```
>>> from sympy import Function, dsolve, pprint, exp, cos
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(f(x).diff(x, 2) + 2*f(x).diff(x) + f(x) -
... 4*exp(-x)*x**2 + cos(2*x), f(x),
... hint='nth_linear_constant_coeff_undetermined_coefficients'))
```

$$f(x) = \frac{1}{3}C_1 + x\frac{1}{3}C_2 + \frac{x^3}{3}e^{-x} - \frac{4\sin(2x)}{25} + \frac{3\cos(2x)}{25}$$

References

- https://en.wikipedia.org/wiki/Method_of_undetermined_coefficients
- M. Tenenbaum & H. Pollard, “Ordinary Differential Equations”, Dover 1963, pp. 221

indirect doctest

`nth_linear_constant_coeff_variation_of_parameters`

class `sympy.solvers.ode.single.NthLinearConstantCoeffVariationOfParameters(ode_problem)`

Solves an n th order linear differential equation with constant coefficients using the method of variation of parameters.

This method works on any differential equations of the form

$$f^{(n)}(x) + a_{n-1}f^{(n-1)}(x) + \cdots + a_1f'(x) + a_0f(x) = P(x).$$

This method works by assuming that the particular solution takes the form

$$\sum_{i=1}^n c_i(x)y_i(x),$$

where y_i is the i th solution to the homogeneous equation. The solution is then solved using Wronskian’s and Cramer’s Rule. The particular solution is given by

$$\sum_{i=1}^n \left(\int \frac{W_i(x)}{W(x)} dx \right) y_i(x),$$

where $W(x)$ is the Wronskian of the fundamental system (the system of n linearly independent solutions to the homogeneous equation), and $W_i(x)$ is the Wronskian of the fundamental system with the i th column replaced with $[0, 0, \dots, 0, P(x)]$.

This method is general enough to solve any n th order inhomogeneous linear differential equation with constant coefficients, but sometimes SymPy cannot simplify the Wronskian well enough to integrate it. If this method hangs, try using the `nth_linear_constant_coeff_variation_of_parameters_Integral` hint and simplifying the integrals manually. Also, prefer using `nth_linear_constant_coeff_undetermined_coefficients` when it applies, because it does not use integration, making it faster and more reliable.

Warning, using `simplify=False` with `'nth_linear_constant_coeff_variation_of_parameters'` in `dsolve()` (page 755) may cause it to hang, because it will not attempt to simplify the Wronskian before integrating. It is recommended that you only use `simplify=False` with `'nth_linear_constant_coeff_variation_of_parameters_Integral'` for this method, especially if the solution to the homogeneous equation has trigonometric functions in it.

Examples

```
>>> from sympy import Function, dsolve, pprint, exp, log
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(f(x).diff(x, 3) - 3*f(x).diff(x, 2) +
... 3*f(x).diff(x) - f(x) - exp(x)*log(x), f(x),
... hint='nth_linear_constant_coeff_variation_of_parameters'))
      /      /      /      x*log(x)  11*x\\ \\  x
f(x) = |C1 + x*|C2 + x*|C3 + ----- - ----|||*e
      \      \      \              6      36 ///
```

References

- https://en.wikipedia.org/wiki/Variation_of_parameters
- <http://planetmath.org/VariationOfParameters>
- M. Tenenbaum & H. Pollard, "Ordinary Differential Equations", Dover 1963, pp. 233

indirect doctest

nth_linear_euler_eq_homogeneous

class `sympy.solvers.ode.single.NthLinearEulerEqHomogeneous(ode_problem)`

Solves an n th order linear homogeneous variable-coefficient Cauchy-Euler equidimensional ordinary differential equation.

This is an equation with form $0 = a_0 f(x) + a_1 x f'(x) + a_2 x^2 f''(x) \dots$.

These equations can be solved in a general manner, by substituting solutions of the form $f(x) = x^r$, and deriving a characteristic equation for r . When there are repeated roots, we include extra terms of the form $C_{rk} \ln^k(x) x^r$, where C_{rk} is an arbitrary integration constant, r is a root of the characteristic equation, and k ranges over the multiplicity of r . In the cases where the roots are complex, solutions of the form $C_1 x^a \sin(b \log(x)) + C_2 x^a \cos(b \log(x))$ are returned, based on expansions with Euler's formula. The general solution is the sum of the terms found. If SymPy cannot find exact

roots to the characteristic equation, a *ComplexRootOf* (page 2431) instance will be returned instead.

```
>>> from sympy import Function, dsolve
>>> from sympy.abc import x
>>> f = Function('f')
>>> dsolve(4*x**2*f(x).diff(x, 2) + f(x), f(x),
... hint='nth_linear_euler_eq_homogeneous')
...
Eq(f(x), sqrt(x)*(C1 + C2*log(x)))
```

Note that because this method does not involve integration, there is no `nth_linear_euler_eq_homogeneous_Integral` hint.

The following is for internal use:

- `returns = 'sol'` returns the solution to the ODE.
- `returns = 'list'` returns a list of linearly independent solutions, corresponding to the fundamental solution set, for use with non homogeneous solution methods like variation of parameters and undetermined coefficients. Note that, though the solutions should be linearly independent, this function does not explicitly check that. You can do `assert simplify(wronskian(sollist)) != 0` to check for linear independence. Also, `assert len(sollist) == order` will need to pass.
- `returns = 'both'`, return a dictionary `{'sol': <solution to ODE>, 'list': <list of linearly independent solutions>}`.

Examples

```
>>> from sympy import Function, dsolve, pprint
>>> from sympy.abc import x
>>> f = Function('f')
>>> eq = f(x).diff(x, 2)*x**2 - 4*f(x).diff(x)*x + 6*f(x)
>>> pprint(dsolve(eq, f(x),
... hint='nth_linear_euler_eq_homogeneous'))
      2
f(x) = x *(C1 + C2*x)
```

References

- https://en.wikipedia.org/wiki/Cauchy%E2%80%93Euler_equation
- C. Bender & S. Orszag, “Advanced Mathematical Methods for Scientists and Engineers”, Springer 1999, pp. 12

indirect doctest

nth_linear_euler_eq_nonhomogeneous_variation_of_parameters

class `sympy.solvers.ode.single.NthLinearEulerEqNonhomogeneousVariationOfParameters(ode_prob)`

Solves an n th order linear non homogeneous Cauchy-Euler equidimensional ordinary differential equation using variation of parameters.

This is an equation with form $g(x) = a_0 f(x) + a_1 x f'(x) + a_2 x^2 f''(x) \cdots$.

This method works by assuming that the particular solution takes the form

$$\sum_{i=1}^n c_i(x) y_i(x) a_n x^n,$$

where y_i is the i th solution to the homogeneous equation. The solution is then solved using Wronskian's and Cramer's Rule. The particular solution is given by multiplying eq given below with $a_n x^n$

$$\sum_{i=1}^n \left(\int \frac{W_i(x)}{W(x)} dx \right) y_i(x),$$

where $W(x)$ is the Wronskian of the fundamental system (the system of n linearly independent solutions to the homogeneous equation), and $W_i(x)$ is the Wronskian of the fundamental system with the i th column replaced with $[0, 0, \dots, 0, \frac{x^{-n}}{a_n} g(x)]$.

This method is general enough to solve any n th order inhomogeneous linear differential equation, but sometimes SymPy cannot simplify the Wronskian well enough to integrate it. If this method hangs, try using the `nth_linear_constant_coeff_variation_of_parameters_Integral` hint and simplifying the integrals manually. Also, prefer using `nth_linear_constant_coeff_undetermined_coefficients` when it applies, because it does not use integration, making it faster and more reliable.

Warning, using `simplify=False` with `'nth_linear_constant_coeff_variation_of_parameters'` in `dsolve()` (page 755) may cause it to hang, because it will not attempt to simplify the Wronskian before integrating. It is recommended that you only use `simplify=False` with `'nth_linear_constant_coeff_variation_of_parameters_Integral'` for this method, especially if the solution to the homogeneous equation has trigonometric functions in it.

Examples

```

>>> from sympy import Function, dsolve, Derivative
>>> from sympy.abc import x
>>> f = Function('f')
>>> eq = x**2*Derivative(f(x), x, x) - 2*x*Derivative(f(x), x) + 2*f(x) -
→ x**4
>>> dsolve(eq, f(x),
... hint='nth_linear_euler_eq_nonhomogeneous_variation_of_parameters').
→ expand()
Eq(f(x), C1*x + C2*x**2 + x**4/6)

```

`nth_linear_euler_eq_nonhomogeneous_undetermined_coefficients`

class `sympy.solvers.ode.single.NthLinearEulerEqNonhomogeneousUndeterminedCoefficients(ode_`

Solves an n th order linear non homogeneous Cauchy-Euler equidimensional ordinary differential equation using undetermined coefficients.

This is an equation with form $g(x) = a_0f(x) + a_1xf'(x) + a_2x^2f''(x) \cdots$.

These equations can be solved in a general manner, by substituting solutions of the form $x = \exp(t)$, and deriving a characteristic equation of form $g(\exp(t)) = b_0f(t) + b_1f'(t) + b_2f''(t) \cdots$ which can be then solved by `nth_linear_constant_coeff_undetermined_coefficients` if $g(\exp(t))$ has finite number of linearly independent derivatives.

Functions that fit this requirement are finite sums functions of the form $ax^ie^{bx} \sin(cx + d)$ or $ax^ie^{bx} \cos(cx + d)$, where i is a non-negative integer and a , b , c , and d are constants. For example any polynomial in x , functions like x^2e^{2x} , $x \sin(x)$, and $e^x \cos(x)$ can all be used. Products of sin's and cos's have a finite number of derivatives, because they can be expanded into $\sin(ax)$ and $\cos(bx)$ terms. However, SymPy currently cannot do that expansion, so you will need to manually rewrite the expression in terms of the above to use this method. So, for example, you will need to manually convert $\sin^2(x)$ into $(1 + \cos(2x))/2$ to properly apply the method of undetermined coefficients on it.

After replacement of x by $\exp(t)$, this method works by creating a trial function from the expression and all of its linear independent derivatives and substituting them into the original ODE. The coefficients for each term will be a system of linear equations, which are be solved for and substituted, giving the solution. If any of the trial functions are linearly dependent on the solution to the homogeneous equation, they are multiplied by sufficient x to make them linearly independent.

Examples

```
>>> from sympy import dsolve, Function, Derivative, log
>>> from sympy.abc import x
>>> f = Function('f')
>>> eq = x**2*Derivative(f(x), x, x) - 2*x*Derivative(f(x), x) + 2*f(x) -
  ↳ log(x)
>>> dsolve(eq, f(x),
... hint='nth_linear_euler_eq_nonhomogeneous_undetermined_coefficients').
  ↳ expand()
Eq(f(x), C1*x + C2*x**2 + log(x)/2 + 3/4)
```

`nth_algebraic`

class `sympy.solvers.ode.single.NthAlgebraic(ode_problem)`

Solves an n th order ordinary differential equation using algebra and integrals.

There is no general form for the kind of equation that this can solve. The the equation is solved algebraically treating differentiation as an invertible algebraic function.

Examples

```
>>> from sympy import Function, dsolve, Eq
>>> from sympy.abc import x
>>> f = Function('f')
>>> eq = Eq(f(x) * (f(x).diff(x)**2 - 1), 0)
>>> dsolve(eq, f(x), hint='nth_algebraic')
[Eq(f(x), 0), Eq(f(x), C1 - x), Eq(f(x), C1 + x)]
```

Note that this solver can return algebraic solutions that do not have any integration constants ($f(x) = 0$ in the above example).

nth_order_reducible

class `sympy.solvers.ode.single.NthOrderReducible(ode_problem)`

Solves ODEs that only involve derivatives of the dependent variable using a substitution of the form $f^n(x) = g(x)$.

For example any second order ODE of the form $f''(x) = h(f'(x), x)$ can be transformed into a pair of 1st order ODEs $g'(x) = h(g(x), x)$ and $f'(x) = g(x)$. Usually the 1st order ODE for g is easier to solve. If that gives an explicit solution for g then f is found simply by integration.

Examples

```
>>> from sympy import Function, dsolve, Eq
>>> from sympy.abc import x
>>> f = Function('f')
>>> eq = Eq(x*f(x).diff(x)**2 + f(x).diff(x, 2), 0)
>>> dsolve(eq, f(x), hint='nth_order_reducible')
...
Eq(f(x), C1 - sqrt(-1/C2)*log(-C2*sqrt(-1/C2) + x) + sqrt(-1/
- C2)*log(C2*sqrt(-1/C2) + x))
```

separable

class `sympy.solvers.ode.single.Separable(ode_problem)`

Solves separable 1st order differential equations.

This is any differential equation that can be written as $P(y) \frac{dy}{dx} = Q(x)$. The solution can then just be found by rearranging terms and integrating: $\int P(y) dy = \int Q(x) dx$. This hint uses `sympy.simplify.simplify.separatevars()` (page 664) as its back end, so if a separable equation is not caught by this solver, it is most likely the fault of that function. `separatevars()` (page 664) is smart enough to do most expansion and factoring necessary to convert a separable equation $F(x, y)$ into the proper form $P(x) \cdot Q(y)$. The general solution is:

```

>>> from sympy import Function, dsolve, Eq, pprint
>>> from sympy.abc import x
>>> a, b, c, d, f = map(Function, ['a', 'b', 'c', 'd', 'f'])
>>> genform = Eq(a(x)*b(f(x))*f(x).diff(x), c(x)*d(f(x)))
>>> pprint(genform)
      d
a(x)*b(f(x))*--(f(x)) = c(x)*d(f(x))
      dx
>>> pprint(dsolve(genform, f(x), hint='separable_Integral'))
      /
      |
      |  b(y)
      |  ---- dy = C1 +
      |  d(y)
      |
      /
      |
      |  c(x)
      |  ---- dx
      |  a(x)
      |
      /

```

Examples

```

>>> from sympy import Function, dsolve, Eq
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(Eq(f(x)*f(x).diff(x) + x, 3*x*f(x)**2), f(x),
... hint='separable', simplify=False))
      / 2 \      2
log\3*f (x) - 1/  x
----- = C1 + --
      6          2

```

References

- M. Tenenbaum & H. Pollard, “Ordinary Differential Equations”, Dover 1963, pp. 52

indirect doctest

almost_linear

class sympy.solvers.ode.single.AlmostLinear(ode_problem)

Solves an almost-linear differential equation.

The general form of an almost linear differential equation is

$$a(x)g'(f(x))f'(x) + b(x)g(f(x)) + c(x)$$

Here $f(x)$ is the function to be solved for (the dependent variable). The substitution $g(f(x)) = u(x)$ leads to a linear differential equation for $u(x)$ of the form $a(x)u' + b(x)u + c(x) = 0$. This can be solved for $u(x)$ by the *first_linear* hint and then $f(x)$ is found by solving $g(f(x)) = u(x)$.

Examples

```
>>> from sympy import dsolve, Function, pprint, sin, cos
>>> from sympy.abc import x
>>> f = Function('f')
>>> d = f(x).diff(x)
>>> eq = x*d + x*f(x) + 1
>>> dsolve(eq, f(x), hint='almost_linear')
Eq(f(x), (C1 - Ei(x))*exp(-x))
>>> pprint(dsolve(eq, f(x), hint='almost_linear'))
      -x
f(x) = (C1 - Ei(x))*e
>>> example = cos(f(x))*f(x).diff(x) + sin(f(x)) + 1
>>> pprint(example)
      d
sin(f(x)) + cos(f(x))*--(f(x)) + 1
      dx
>>> pprint(dsolve(example, f(x), hint='almost_linear'))
      /      -x      \      /      -x      \
[f(x) = pi - asin\C1*e  - 1/, f(x) = asin\C1*e  - 1/]
```

See also:

[*sympy.solvers.ode.single.FirstLinear*](#) (page 776)

References

- Joel Moses, "Symbolic Integration - The Stormy Decade", Communications of the ACM, Volume 14, Number 8, August 1971, pp. 558

linear_coefficients

class `sympy.solvers.ode.single.LinearCoefficients(ode_problem)`

Solves a differential equation with linear coefficients.

The general form of a differential equation with linear coefficients is

$$y' + F\left(\frac{a_1x + b_1y + c_1}{a_2x + b_2y + c_2}\right) = 0,$$

where $a_1, b_1, c_1, a_2, b_2, c_2$ are constants and $a_1b_2 - a_2b_1 \neq 0$.

This can be solved by substituting:

$$x = x' + \frac{b_2c_1 - b_1c_2}{a_2b_1 - a_1b_2}$$

$$y = y' + \frac{a_1c_2 - a_2c_1}{a_2b_1 - a_1b_2}.$$

This substitution reduces the equation to a homogeneous differential equation.

Examples

```
>>> from sympy import dsolve, Function, pprint
>>> from sympy.abc import x
>>> f = Function('f')
>>> df = f(x).diff(x)
>>> eq = (x + f(x) + 1)*df + (f(x) - 6*x + 1)
>>> dsolve(eq, hint='linear_coefficients')
[Eq(f(x), -x - sqrt(C1 + 7*x**2) - 1), Eq(f(x), -x + sqrt(C1 + 7*x**2) -
→ 1)]
>>> pprint(dsolve(eq, hint='linear_coefficients'))
```

$$[f(x) = -x - \sqrt{C1 + 7x^2} - 1, f(x) = -x + \sqrt{C1 + 7x^2} - 1]$$

See also:

[*sympy.solvers.ode.single.HomogeneousCoeffBest*](#) (page 772), [*sympy.solvers.ode.single.HomogeneousCoeffSubsIndepDivDep*](#) (page 774), [*sympy.solvers.ode.single.HomogeneousCoeffSubsDepDivIndep*](#) (page 773)

References

- Joel Moses, “Symbolic Integration - The Stormy Decade”, Communications of the ACM, Volume 14, Number 8, August 1971, pp. 558

separable_reduced

class `sympy.solvers.ode.single.SeparableReduced(ode_problem)`

Solves a differential equation that can be reduced to the separable form.

The general form of this equation is

$$y' + (y/x)H(x^n y) = 0.$$

This can be solved by substituting $u(y) = x^n y$. The equation then reduces to the separable form $\frac{u'}{u(\text{power}-H(u))} - \frac{1}{x} = 0$.

The general solution is:

```
>>> from sympy import Function, dsolve, pprint
>>> from sympy.abc import x, n
>>> f, g = map(Function, ['f', 'g'])
>>> genform = f(x).diff(x) + (f(x)/x)*g(x**n*f(x))
>>> pprint(genform)
```

$$d \frac{f(x)}{dx} + \frac{f(x)^{n+1} g(f(x)^n)}{x}$$

```
>>> pprint(dsolve(genform, hint='separable_reduced'))
```

$$\frac{n}{x} f(x)$$

(continues on next page)

(continued from previous page)

```

/
|
|      1
|----- dy = C1 + log(x)
| y*(n - g(y))
|
/

```

Examples

```

>>> from sympy import dsolve, Function, pprint
>>> from sympy.abc import x
>>> f = Function('f')
>>> d = f(x).diff(x)
>>> eq = (x - x**2*f(x))*d - f(x)
>>> dsolve(eq, hint='separable_reduced')
[Eq(f(x), (1 - sqrt(C1*x**2 + 1))/x), Eq(f(x), (sqrt(C1*x**2 + 1) + 1)/
→x)]
>>> pprint(dsolve(eq, hint='separable_reduced'))

```

```

      1 - \ /  C1*x  + 1      \ /  C1*x  + 1 + 1
[f(x) = -----, f(x) = -----]
           x                  x

```

See also:

[*sympy.solvers.ode.single.Separable*](#) (page 789)

References

- Joel Moses, “Symbolic Integration - The Stormy Decade”, Communications of the ACM, Volume 14, Number 8, August 1971, pp. 558

lie_group

class `sympy.solvers.ode.single.LieGroup(ode_problem)`

This hint implements the Lie group method of solving first order differential equations. The aim is to convert the given differential equation from the given coordinate system into another coordinate system where it becomes invariant under the one-parameter Lie group of translations. The converted ODE can be easily solved by quadrature. It makes use of the [*sympy.solvers.ode.infinitesimals\(\)*](#) (page 764) function which returns the infinitesimals of the transformation.

The coordinates r and s can be found by solving the following Partial Differential Equations.

$$\xi \frac{\partial r}{\partial x} + \eta \frac{\partial r}{\partial y} = 0$$

$$\xi \frac{\partial s}{\partial x} + \eta \frac{\partial s}{\partial y} = 1$$

The differential equation becomes separable in the new coordinate system

$$\frac{ds}{dr} = \frac{\frac{\partial s}{\partial x} + h(x, y) \frac{\partial s}{\partial y}}{\frac{\partial r}{\partial x} + h(x, y) \frac{\partial r}{\partial y}}$$

After finding the solution by integration, it is then converted back to the original coordinate system by substituting r and s in terms of x and y again.

Examples

```
>>> from sympy import Function, dsolve, exp, pprint
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(f(x).diff(x) + 2*x*f(x) - x*exp(-x**2), f(x),
... hint='lie_group'))
      /      2\      2
      |      x |      -x
f(x) = | C1 + -- | * e
      \      2 /
```

References

- Solving differential equations by Symmetry Groups, John Starrett, pp. 1 - pp. 14

2nd_hypergeometric

class sympy.solvers.ode.single.**SecondHypergeometric**(*ode_problem*)

Solves 2nd order linear differential equations.

It computes special function solutions which can be expressed using the 2F1, 1F1 or 0F1 hypergeometric functions.

$$y'' + A(x)y' + B(x)y = 0,$$

where A and B are rational functions.

These kinds of differential equations have solution of non-Liouvillian form.

Given linear ODE can be obtained from 2F1 given by

$$(x^2 - x)y'' + ((a + b + 1)x - c)y' + bay = 0,$$

where $\{a, b, c\}$ are arbitrary constants.

Notes

The algorithm should find any solution of the form

$$y = P(x) {}_pF_q(\dots; \dots; \frac{\alpha x^k + \beta}{\gamma x^k + \delta}),$$

where ${}_pF_q$ is any of ${}_2F_1$, ${}_1F_1$ or ${}_0F_1$ and P is an “arbitrary function”. Currently only the ${}_2F_1$ case is implemented in SymPy but the other cases are described in the paper and could be implemented in future (contributions welcome!).

Examples

```
>>> from sympy import Function, dsolve, pprint
>>> from sympy.abc import x
>>> f = Function('f')
>>> eq = (x*x - x)*f(x).diff(x,2) + (5*x - 1)*f(x).diff(x) + 4*f(x)
>>> pprint(dsolve(eq, f(x), '2nd_hypergeometric'))
```

$$f(x) = \frac{\frac{C1 + C2 \log(x) + \frac{4}{x+1} \sqrt{\frac{-1}{2}} \sqrt{-1} \sqrt{x}}{(x-1)^3}}$$

References

- “Non-Liouvillian solutions for second order linear ODEs” by L. Chan, E.S. Cheb-Terrab

1st_power_series

`sympy.solvers.ode.ode.ode_1st_power_series(eq, func, order, match)`

The power series solution is a method which gives the Taylor series expansion to the solution of a differential equation.

For a first order differential equation $\frac{dy}{dx} = h(x, y)$, a power series solution exists at a point $x = x_0$ if $h(x, y)$ is analytic at x_0 . The solution is given by

$$y(x) = y(x_0) + \sum_{n=1}^{\infty} \frac{F_n(x_0, b)(x - x_0)^n}{n!},$$

where $y(x_0) = b$ is the value of y at the initial value of x_0 . To compute the values of the $F_n(x_0, b)$ the following algorithm is followed, until the required number of terms are generated.

1. $F_1 = h(x_0, b)$
2. $F_{n+1} = \frac{\partial F_n}{\partial x} + \frac{\partial F_n}{\partial y} F_1$

Examples

```
>>> from sympy import Function, pprint, exp, dsolve
>>> from sympy.abc import x
>>> f = Function('f')
>>> eq = exp(x)*(f(x).diff(x)) - f(x)
>>> pprint(dsolve(eq, hint='1st_power_series'))
```

$$f(x) = C_1 + C_1 x - \frac{C_1 x^3}{6} + \frac{C_1 x^4}{24} + \frac{C_1 x^5}{60} + O(x^6)$$

References

- Travis W. Walker, Analytic power series technique for solving first-order differential equations, p.p 17, 18

2nd_power_series_ordinary

`sympy.solvers.ode.ode_2nd_power_series_ordinary(eq, func, order, match)`

Gives a power series solution to a second order homogeneous differential equation with polynomial coefficients at an ordinary point. A homogeneous differential equation is of the form

$$P(x)\frac{d^2y}{dx^2} + Q(x)\frac{dy}{dx} + R(x)y(x) = 0$$

For simplicity it is assumed that $P(x)$, $Q(x)$ and $R(x)$ are polynomials, it is sufficient that $\frac{Q(x)}{P(x)}$ and $\frac{R(x)}{P(x)}$ exists at x_0 . A recurrence relation is obtained by substituting y as $\sum_{n=0}^{\infty} a_n x^n$, in the differential equation, and equating the n th term. Using this relation various terms can be generated.

Examples

```
>>> from sympy import dsolve, Function, pprint
>>> from sympy.abc import x
>>> f = Function("f")
>>> eq = f(x).diff(x, 2) + f(x)
>>> pprint(dsolve(eq, hint='2nd_power_series_ordinary'))
```

$$f(x) = C_2 \sqrt[4]{x} - \frac{x^2}{2} + 1 + C_1 x \sqrt[4]{1 - \frac{x^2}{6}} + O(x^6)$$

References

- <http://tutorial.math.lamar.edu/Classes/DE/SeriesSolutions.aspx>
- George E. Simmons, "Differential Equations with Applications and Historical Notes", p.p 176 - 184

2nd_power_series_regular

`sympy.solvers.ode.ode.ode_2nd_power_series_regular(eq, func, order, match)`

Gives a power series solution to a second order homogeneous differential equation with polynomial coefficients at a regular point. A second order homogeneous differential equation is of the form

$$P(x)\frac{d^2y}{dx^2} + Q(x)\frac{dy}{dx} + R(x)y(x) = 0$$

A point is said to regular singular at x_0 if $x - x_0 \frac{Q(x)}{P(x)}$ and $(x - x_0)^2 \frac{R(x)}{P(x)}$ are analytic at x_0 . For simplicity $P(x)$, $Q(x)$ and $R(x)$ are assumed to be polynomials. The algorithm for finding the power series solutions is:

1. Try expressing $(x - x_0)P(x)$ and $((x - x_0)^2)Q(x)$ as power series solutions about x_0 . Find p_0 and q_0 which are the constants of the power series expansions.
2. Solve the indicial equation $f(m) = m(m - 1) + m * p_0 + q_0$, to obtain the roots m_1 and m_2 of the indicial equation.
3. If $m_1 - m_2$ is a non integer there exists two series solutions. If $m_1 = m_2$, there exists only one solution. If $m_1 - m_2$ is an integer, then the existence of one solution is confirmed. The other solution may or may not exist.

The power series solution is of the form $x^m \sum_{n=0}^{\infty} a_n x^n$. The coefficients are determined by the following recurrence relation. $a_n = -\frac{\sum_{k=0}^{n-1} q_{n-k} + (m+k)p_{n-k}}{f(m+n)}$. For the case in which $m_1 - m_2$ is an integer, it can be seen from the recurrence relation that for the lower root m , when n equals the difference of both the roots, the denominator becomes zero. So if the numerator is not equal to zero, a second series solution exists.

Examples

```
>>> from sympy import dsolve, Function, pprint
>>> from sympy.abc import x
>>> f = Function("f")
>>> eq = x*(f(x).diff(x, 2)) + 2*(f(x).diff(x)) + x*f(x)
>>> pprint(dsolve(eq, hint='2nd_power_series_regular'))
```

$$f(x) = C_2 \frac{x^4}{120} - \frac{x^2}{6} + 1 + C_1 \frac{x^6}{720} + \frac{x^4}{24} - \frac{x^2}{2} + 1 + \frac{x^6}{0!x}$$

References

- George E. Simmons, “Differential Equations with Applications and Historical Notes”, p.p 176 - 184

Lie heuristics

These functions are intended for internal use of the Lie Group Solver. Nonetheless, they contain useful information in their docstrings on the algorithms implemented for the various heuristics.

abaco1_simple

`sympy.solvers.ode.lie_group.lie_heuristic_abaco1_simple(match, comp=False)`

The first heuristic uses the following four sets of assumptions on ξ and η

$$\xi = 0, \eta = f(x)$$

$$\xi = 0, \eta = f(y)$$

$$\xi = f(x), \eta = 0$$

$$\xi = f(y), \eta = 0$$

The success of this heuristic is determined by algebraic factorisation. For the first assumption $\xi = 0$ and η to be a function of x , the PDE

$$\frac{\partial \eta}{\partial x} + \left(\frac{\partial \eta}{\partial y} - \frac{\partial \xi}{\partial x} \right) * h - \frac{\partial \xi}{\partial y} * h^2 - \xi * \frac{\partial h}{\partial x} - \eta * \frac{\partial h}{\partial y} = 0$$

reduces to $f'(x) - f \frac{\partial h}{\partial y} = 0$. If $\frac{\partial h}{\partial y}$ is a function of x , then this can usually be integrated easily. A similar idea is applied to the other 3 assumptions as well.

References

- E.S Cheb-Terrab, L.G.S Duarte and L.A.C.P da Mota, Computer Algebra Solving of First Order ODEs Using Symmetry Methods, pp. 8

abaco1_product

`sympy.solvers.ode.lie_group.lie_heuristic_abaco1_product(match, comp=False)`

The second heuristic uses the following two assumptions on ξ and η

$$\eta = 0, \xi = f(x) * g(y)$$

$$\eta = f(x) * g(y), \xi = 0$$

The first assumption of this heuristic holds good if $\frac{1}{h^2} \frac{\partial^2}{\partial x \partial y} \log(h)$ is separable in x and y , then the separated factors containing x is $f(x)$, and $g(y)$ is obtained by

$$e^{\int f \frac{\partial}{\partial x} \left(\frac{1}{f * h} \right) dy}$$

provided $f \frac{\partial}{\partial x} \left(\frac{1}{f * h} \right)$ is a function of y only.

The second assumption holds good if $\frac{dy}{dx} = h(x, y)$ is rewritten as $\frac{dy}{dx} = \frac{1}{h(y, x)}$ and the same properties of the first assumption satisfies. After obtaining $f(x)$ and $g(y)$, the coordinates are again interchanged, to get η as $f(x) * g(y)$

References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 7 - pp. 8

bivariate

`sympy.solvers.ode.lie_group.lie_heuristic_bivariate(match, comp=False)`

The third heuristic assumes the infinitesimals ξ and η to be bi-variate polynomials in x and y . The assumption made here for the logic below is that h is a rational function in x and y though that may not be necessary for the infinitesimals to be bivariate polynomials. The coefficients of the infinitesimals are found out by substituting them in the PDE and grouping similar terms that are polynomials and since they form a linear system, solve and check for non trivial solutions. The degree of the assumed bivariates are increased till a certain maximum value.

References

- Lie Groups and Differential Equations pp. 327 - pp. 329

chi

`sympy.solvers.ode.lie_group.lie_heuristic_chi(match, comp=False)`

The aim of the fourth heuristic is to find the function $\chi(x, y)$ that satisfies the PDE $\frac{d\chi}{dx} + h \frac{d\chi}{dy} - \frac{\partial h}{\partial y} \chi = 0$.

This assumes χ to be a bivariate polynomial in x and y . By intuition, h should be a rational function in x and y . The method used here is to substitute a general binomial for χ up to a certain maximum degree is reached. The coefficients of the polynomials, are calculated by by collecting terms of the same order in x and y .

After finding χ , the next step is to use $\eta = \xi * h + \chi$, to determine ξ and η . This can be done by dividing χ by h which would give $-\xi$ as the quotient and η as the remainder.

References

- E.S Cheb-Terrab, L.G.S Duarte and L.A.C.P da Mota, Computer Algebra Solving of First Order ODEs Using Symmetry Methods, pp. 8

abaco2_similar

`sympy.solvers.ode.lie_group.lie_heuristic_abaco2_similar(match, comp=False)`

This heuristic uses the following two assumptions on ξ and η

$$\eta = g(x), \xi = f(x)$$

$$\eta = f(y), \xi = g(y)$$

For the first assumption,

1. First $\frac{\frac{\partial h}{\partial y}}{\frac{\partial^2 h}{\partial y^2}}$ is calculated. Let us say this value is A
2. If this is constant, then h is matched to the form $A(x) + B(x)e^{\frac{y}{C}}$ then, $\frac{e^{\int \frac{A(x)}{C} dx}}{B(x)}$ gives $f(x)$ and $A(x) * f(x)$ gives $g(x)$
3. Otherwise $\frac{\frac{\partial A}{\partial X}}{\frac{\partial A}{\partial Y}} = \gamma$ is calculated. If
 - a] γ is a function of x alone
 - b] $\frac{\gamma \frac{\partial h}{\partial y} - \gamma'(x) - \frac{\partial h}{\partial x}}{h + \gamma} = G$ is a function of x alone. then, $e^{\int G dx}$ gives $f(x)$ and $-\gamma * f(x)$ gives $g(x)$

The second assumption holds good if $\frac{dy}{dx} = h(x, y)$ is rewritten as $\frac{dy}{dx} = \frac{1}{h(y, x)}$ and the same properties of the first assumption satisfies. After obtaining $f(x)$ and $g(x)$, the coordinates are again interchanged, to get ξ as $f(x^*)$ and η as $g(y^*)$

References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 10 - pp. 12

function_sum

`sympy.solvers.ode.lie_group.lie_heuristic_function_sum(match, comp=False)`

This heuristic uses the following two assumptions on ξ and η

$$\eta = 0, \xi = f(x) + g(y)$$

$$\eta = f(x) + g(y), \xi = 0$$

The first assumption of this heuristic holds good if

$$\frac{\partial}{\partial y} [(h \frac{\partial^2}{\partial x^2} (h^{-1}))^{-1}]$$

is separable in x and y ,

1. The separated factors containing y is $\frac{\partial g}{\partial y}$. From this $g(y)$ can be determined.
2. The separated factors containing x is $f''(x)$.
3. $h \frac{\partial^2}{\partial x^2}(h^{-1})$ equals $\frac{f''(x)}{f(x)+g(y)}$. From this $f(x)$ can be determined.

The second assumption holds good if $\frac{dy}{dx} = h(x, y)$ is rewritten as $\frac{dy}{dx} = \frac{1}{h(y, x)}$ and the same properties of the first assumption satisfies. After obtaining $f(x)$ and $g(y)$, the coordinates are again interchanged, to get η as $f(x) + g(y)$.

For both assumptions, the constant factors are separated among $g(y)$ and $f''(x)$, such that $f''(x)$ obtained from 3] is the same as that obtained from 2]. If not possible, then this heuristic fails.

References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 7 - pp. 8

abaco2_unique_unknown

```
sympy.solvers.ode.lie_group.lie_heuristic_abaco2_unique_unknown(match,
                                                                comp=False)
```

This heuristic assumes the presence of unknown functions or known functions with non-integer powers.

1. A list of all functions and non-integer powers containing x and y
2. Loop over each element f in the list, find $\frac{\partial f}{\partial x} = R$

If it is separable in x and y , let X be the factors containing x . Then

a] Check if $\xi = X$ and $\eta = -\frac{X}{R}$ satisfy the PDE. If yes, then return ξ and η

b] Check if $\xi = \frac{-R}{X}$ and $\eta = -\frac{1}{X}$ satisfy the PDE.
If yes, then return ξ and η

If not, then check if

a] $\xi = -R, \eta = 1$

b] $\xi = 1, \eta = -\frac{1}{R}$

are solutions.

References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 10 - pp. 12

abaco2_unique_general

`sympy.solvers.ode.lie_group.lie_heuristic_abaco2_unique_general(match, comp=False)`

This heuristic finds if infinitesimals of the form $\eta = f(x)$, $\xi = g(y)$ without making any assumptions on h .

The complete sequence of steps is given in the paper mentioned below.

References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 10 - pp. 12

linear

`sympy.solvers.ode.lie_group.lie_heuristic_linear(match, comp=False)`

This heuristic assumes

1. $\xi = ax + by + c$ and
2. $\eta = fx + gy + h$

After substituting the following assumptions in the determining PDE, it reduces to

$$f + (g - a)h - bh^2 - (ax + by + c)\frac{\partial h}{\partial x} - (fx + gy + c)\frac{\partial h}{\partial y}$$

Solving the reduced PDE obtained, using the method of characteristics, becomes impractical. The method followed is grouping similar terms and solving the system of linear equations obtained. The difference between the bivariate heuristic is that h need not be a rational function in this case.

References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 10 - pp. 12

Rational Riccati Solver

These functions are intended for internal use to solve a first order Riccati differential equation with atleast one rational particular solution.

`riccati_normal`

`sympy.solvers.ode.riccati.riccati_normal(w, x, b1, b2)`

Given a solution $w(x)$ to the equation

$$w'(x) = b_0(x) + b_1(x) * w(x) + b_2(x) * w(x)^2$$

and rational function coefficients $b_1(x)$ and $b_2(x)$, this function transforms the solution to give a solution $y(x)$ for its corresponding normal Riccati ODE

$$y'(x) + y(x)^2 = a(x)$$

using the transformation

$$y(x) = -b_2(x) * w(x) - b_2'(x)/(2 * b_2(x)) - b_1(x)/2$$

`riccati_inverse_normal`

`sympy.solvers.ode.riccati.riccati_inverse_normal(y, x, b1, b2, bp=None)`

Inverse transforming the solution to the normal Riccati ODE to get the solution to the Riccati ODE.

`riccati_reduced`

`sympy.solvers.ode.riccati.riccati_reduced(eq, f, x)`

Convert a Riccati ODE into its corresponding normal Riccati ODE.

`construct_c`

`sympy.solvers.ode.riccati.construct_c(num, den, x, poles, muls)`

Helper function to calculate the coefficients in the c-vector for each pole.

`construct_d`

`sympy.solvers.ode.riccati.construct_d(num, den, x, val_inf)`

Helper function to calculate the coefficients in the d-vector based on the valuation of the function at oo.

rational_laurent_series

`sympy.solvers.ode.riccati.rational_laurent_series(num, den, x, r, m, n)`

The function computes the Laurent series coefficients of a rational function.

Parameters

num: A Poly object that is the numerator of $f(x)$.

den: A Poly object that is the denominator of $f(x)$.

x: The variable of expansion of the series.

r: The point of expansion of the series.

m: Multiplicity of r if r is a pole of $f(x)$. Should be zero otherwise.

n: Order of the term upto which the series is expanded.

Returns

series: A dictionary that has power of the term as key and coefficient of that term as value.

Below is a basic outline of how the Laurent series of a rational function $f(x)$ about x_0 is being calculated -

1. Substitute $x + x_0$ in place of x . If x_0 is a pole of $f(x)$, multiply the expression by x^m where m is the multiplicity of x_0 . Denote the resulting expression as $g(x)$. We do this substitution so that we can now find the Laurent series of $g(x)$ about $x = 0$.
2. We can then assume that the Laurent series of $g(x)$ takes the following form -

$$g(x) = \frac{num(x)}{den(x)} = \sum_{m=0}^{\infty} a_m x^m$$

where a_m denotes the Laurent series coefficients.

3. Multiply the denominator to the RHS of the equation and form a recurrence relation for the coefficients a_m .

compute_m_ybar

`sympy.solvers.ode.riccati.compute_m_ybar(x, poles, choice, N)`

Helper function to calculate -

1. m - The degree bound for the polynomial solution that must be found for the auxiliary differential equation.
2. ybar - Part of the solution which can be computed using the poles, c and d vectors.

solve_aux_eq

`sympy.solvers.ode.riccati.solve_aux_eq(numa, dena, numy, deny, x, m)`

Helper function to find a polynomial solution of degree m for the auxiliary differential equation.

remove_redundant_sols

`sympy.solvers.ode.riccati.remove_redundant_sols(sol1, sol2, x)`

Helper function to remove redundant solutions to the differential equation.

get_gen_sol_from_part_sol

`sympy.solvers.ode.riccati.get_gen_sol_from_part_sol(part_sols, a, x)`

" Helper function which computes the general solution for a Riccati ODE from its particular solutions.

There are 3 cases to find the general solution from the particular solutions for a Riccati ODE depending on the number of particular solution(s) we have - 1, 2 or 3.

For more information, see Section 6 of "Methods of Solution of the Riccati Differential Equation" by D. R. Haaheim and F. M. Stein

solve_riccati

`sympy.solvers.ode.riccati.solve_riccati(fx, x, b0, b1, b2, gensol=False)`

The main function that gives particular/general solutions to Riccati ODEs that have atleast 1 rational particular solution.

System of ODEs

These functions are intended for internal use by `dsolve()` (page 755) for system of differential equations.

Linear, 2 equations, Order 1, Type 6

`sympy.solvers.ode.ode._linear_2eq_order1_type6(x, y, t, r, eq)`

The equations of this type of ode are .

$$x' = f(t)x + g(t)y$$

$$y' = a[f(t) + ah(t)]x + a[g(t) - h(t)]y$$

This is solved by first multiplying the first equation by $-a$ and adding it to the second equation to obtain

$$y' - ax' = -ah(t)(y - ax)$$

Setting $U = y - ax$ and integrating the equation we arrive at

$$y - ax = C_1 e^{-a \int h(t) dt}$$

and on substituting the value of y in first equation give rise to first order ODEs. After solving for x , we can obtain y by substituting the value of x in second equation.

Linear, 2 equations, Order 1, Type 7

`sympy.solvers.ode.ode._linear_2eq_order1_type7(x, y, t, r, eq)`

The equations of this type of ode are .

$$x' = f(t)x + g(t)y$$

$$y' = h(t)x + p(t)y$$

Differentiating the first equation and substituting the value of y from second equation will give a second-order linear equation

$$gx'' - (fg + gp + g')x' + (fgp - g^2h + fg' - f'g)x = 0$$

This above equation can be easily integrated if following conditions are satisfied.

1. $fgp - g^2h + fg' - f'g = 0$
2. $fgp - g^2h + fg' - f'g = ag, fg + gp + g' = bg$

If first condition is satisfied then it is solved by current dsolve solver and in second case it becomes a constant coefficient differential equation which is also solved by current solver.

Otherwise if the above condition fails then, a particular solution is assumed as $x = x_0(t)$ and $y = y_0(t)$ Then the general solution is expressed as

$$x = C_1 x_0(t) + C_2 x_0(t) \int \frac{g(t)F(t)P(t)}{x_0^2(t)} dt$$

$$y = C_1 y_0(t) + C_2 \left[\frac{F(t)P(t)}{x_0(t)} + y_0(t) \int \frac{g(t)F(t)P(t)}{x_0^2(t)} dt \right]$$

where C_1 and C_2 are arbitrary constants and

$$F(t) = e^{\int f(t) dt}, P(t) = e^{\int p(t) dt}$$

Linear ODE to matrix

`sympy.solvers.ode.systems.linear_ode_to_matrix(eqs, funcs, t, order)`

Convert a linear system of ODEs to matrix form

Parameters

eqs : list of SymPy expressions or equalities

The equations as expressions (assumed equal to zero).

funcs : list of applied functions

The dependent variables of the system of ODEs.

t : symbol

The independent variable.

order : int

The order of the system of ODEs.

Returns

The tuple (A_s, b) where A_s is a tuple of matrices and b is the matrix representing the rhs of the matrix equation.

Raises

ODEOrderError

When the system of ODEs have an order greater than what was specified

ODENonlinearError

When the system of ODEs is nonlinear

Explanation

Express a system of linear ordinary differential equations as a single matrix differential equation [1]. For example the system $x' = x + y + 1$ and $y' = x - y$ can be represented as

$$A_1 X' = A_0 X + b$$

where A_1 and A_0 are 2×2 matrices and b , X and X' are 2×1 matrices with $X = [x, y]^T$.

Higher-order systems are represented with additional matrices e.g. a second-order system would look like

$$A_2 X'' = A_1 X' + A_0 X + b$$

Examples

```
>>> from sympy import Function, Symbol, Matrix, Eq
>>> from sympy.solvers.ode.systems import linear_ode_to_matrix
>>> t = Symbol('t')
>>> x = Function('x')
>>> y = Function('y')
```

We can create a system of linear ODEs like

```
>>> eqs = [
...     Eq(x(t).diff(t), x(t) + y(t) + 1),
...     Eq(y(t).diff(t), x(t) - y(t)),
... ]
>>> funcs = [x(t), y(t)]
>>> order = 1 # 1st order system
```

Now `linear_ode_to_matrix` can represent this as a matrix differential equation.

```
>>> (A1, A0), b = linear_ode_to_matrix(eqs, funcs, t, order)
>>> A1
Matrix([
[1, 0],
[0, 1]])
>>> A0
Matrix([
[1, 1],
[1, -1]])
>>> b
Matrix([
[1],
[0]])
```

The original equations can be recovered from these matrices:

```
>>> eqs_mat = Matrix([eq.lhs - eq.rhs for eq in eqs])
>>> X = Matrix(funcs)
>>> A1 * X.diff(t) - A0 * X - b == eqs_mat
True
```

If the system of equations has a maximum order greater than the order of the system specified, a `ODEOrderError` exception is raised.

```
>>> eqs = [Eq(x(t).diff(t, 2), x(t).diff(t) + x(t)), Eq(y(t).diff(t),
↳ y(t) + x(t))]
>>> linear_ode_to_matrix(eqs, funcs, t, 1)
Traceback (most recent call last):
...
ODEOrderError: Cannot represent system in 1-order form
```

If the system of equations is nonlinear, then `ODENonlinearError` is raised.

```
>>> eqs = [Eq(x(t).diff(t), x(t) + y(t)), Eq(y(t).diff(t), y(t)**2 +
↳ x(t))]
```

(continues on next page)

(continued from previous page)

```
>>> linear_ode_to_matrix(eqs, funcs, t, 1)
Traceback (most recent call last):
...
ODENonlinearError: The system of ODEs is nonlinear.
```

See also:

[*linear_eq_to_matrix*](#) (page 870)
for systems of linear algebraic equations.

References

[R791]

Canonical Equations Converter

`sympy.solvers.ode.systems.canonical_odes(eqs, funcs, t)`

Function that solves for highest order derivatives in a system

Parameters

eqs : List

List of the ODEs

funcs : List

List of dependent variables

t : Symbol

Independent variable

Returns

List

Explanation

This function inputs a system of ODEs and based on the system, the dependent variables and their highest order, returns the system in the following form:

$$X'(t) = A(t)X(t) + b(t)$$

Here, $X(t)$ is the vector of dependent variables of lower order, $A(t)$ is the coefficient matrix, $b(t)$ is the non-homogeneous term and $X'(t)$ is the vector of dependent variables in their respective highest order. We use the term canonical form to imply the system of ODEs which is of the above form.

If the system passed has a non-linear term with multiple solutions, then a list of systems is returned in its canonical form.

Examples

```
>>> from sympy import symbols, Function, Eq, Derivative
>>> from sympy.solvers.ode.systems import canonical_odes
>>> f, g = symbols("f g", cls=Function)
>>> x, y = symbols("x y")
>>> funcs = [f(x), g(x)]
>>> eqs = [Eq(f(x).diff(x) - 7*f(x), 12*g(x)), Eq(g(x).diff(x) + g(x),
↳ 20*f(x))]
```

```
>>> canonical_eqs = canonical_odes(eqs, funcs, x)
>>> canonical_eqs
[[Eq(Derivative(f(x), x), 7*f(x) + 12*g(x)), Eq(Derivative(g(x), x),
↳ 20*f(x) - g(x))]]
```

```
>>> system = [Eq(Derivative(f(x), x)**2 - 2*Derivative(f(x), x) + 1, 4),
↳ Eq(-y*f(x) + Derivative(g(x), x), 0)]
```

```
>>> canonical_system = canonical_odes(system, funcs, x)
>>> canonical_system
[[Eq(Derivative(f(x), x), -1), Eq(Derivative(g(x), x), y*f(x))],
↳ [Eq(Derivative(f(x), x), 3), Eq(Derivative(g(x), x), y*f(x))]]
```

LinODESolve Systems Information

`sympy.solvers.ode.systems.linodesolve_type(A, t, b=None)`

Helper function that determines the type of the system of ODEs for solving with *sympy.solvers.ode.systems.linodesolve()* (page 814)

Parameters

A : Matrix

Coefficient matrix of the system of ODEs

b : Matrix or None

Non-homogeneous term of the system. The default value is None. If this argument is None, then the system is assumed to be homogeneous.

Returns

Dict

Raises

NotImplementedError

When the coefficient matrix does not have a commutative antiderivative

Explanation

This function takes in the coefficient matrix and/or the non-homogeneous term and returns the type of the equation that can be solved by `sympy.solvers.ode.systems.lnodesolve()` (page 814).

If the system is constant coefficient homogeneous, then “type1” is returned

If the system is constant coefficient non-homogeneous, then “type2” is returned

If the system is non-constant coefficient homogeneous, then “type3” is returned

If the system is non-constant coefficient non-homogeneous, then “type4” is returned

If the system has a non-constant coefficient matrix which can be factorized into constant coefficient matrix, then “type5” or “type6” is returned for when the system is homogeneous or non-homogeneous respectively.

Note that, if the system of ODEs is of “type3” or “type4”, then along with the type, the commutative antiderivative of the coefficient matrix is also returned.

If the system cannot be solved by `sympy.solvers.ode.systems.lnodesolve()` (page 814), then `NotImplementedError` is raised.

Examples

```
>>> from sympy import symbols, Matrix
>>> from sympy.solvers.ode.systems import lnodesolve_type
>>> t = symbols("t")
>>> A = Matrix([[1, 1], [2, 3]])
>>> b = Matrix([t, 1])
```

```
>>> lnodesolve_type(A, t)
{'antiderivative': None, 'type_of_equation': 'type1'}
```

```
>>> lnodesolve_type(A, t, b=b)
{'antiderivative': None, 'type_of_equation': 'type2'}
```

```
>>> A_t = Matrix([[1, t], [-t, 1]])
```

```
>>> lnodesolve_type(A_t, t)
{'antiderivative': Matrix([
[      t, t**2/2],
[-t**2/2,      t]]), 'type_of_equation': 'type3'}
```

```
>>> lnodesolve_type(A_t, t, b=b)
{'antiderivative': Matrix([
[      t, t**2/2],
[-t**2/2,      t]]), 'type_of_equation': 'type4'}
```

```
>>> A_non_commutative = Matrix([[1, t], [t, -1]])
>>> lnodesolve_type(A_non_commutative, t)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
NotImplementedError:
The system does not have a commutative antiderivative, it cannot be
solved by linsolve.
```

See also:

[**`linsolve`**](#) (page 814)

Function for which `linsolve_type` gets the information

Matrix Exponential Jordan Form

`sympy.solvers.ode.systems.matrix_exp_jordan_form(A, t)`

Matrix exponential $\exp(A * t)$ for the matrix A and scalar t .

Parameters

A : Matrix

The matrix A in the expression $\exp(A * t)$

t : Symbol

The independent variable

Explanation

Returns the Jordan form of the $\exp(A * t)$ along with the matrix P such that:

$$\exp(A * t) = P * \exp J * P^{-1}$$

Examples

```
>>> from sympy import Matrix, Symbol
>>> from sympy.solvers.ode.systems import matrix_exp, matrix_exp_jordan_
    form
>>> t = Symbol('t')
```

We will consider a 2x2 defective matrix. This shows that our method works even for defective matrices.

```
>>> A = Matrix([[1, 1], [0, 1]])
```

It can be observed that this function gives us the Jordan normal form and the required invertible matrix P .

```
>>> P, expJ = matrix_exp_jordan_form(A, t)
```

Here, it is shown that P and $\exp J$ returned by this function is correct as they satisfy the formula: $P * \exp J * P_{\text{inverse}} = \exp(A * t)$.

```
>>> P * expJ * P.inv() == matrix_exp(A, t)
True
```

References

[R792], [R793], [R794]

Matrix Exponential

`sympy.solvers.ode.systems.matrix_exp(A, t)`

Matrix exponential $\exp(A * t)$ for the matrix A and scalar t .

Parameters

A : Matrix

The matrix A in the expression $\exp(A * t)$

t : Symbol

The independent variable

Explanation

This functions returns the $\exp(A * t)$ by doing a simple matrix multiplication:

$$\exp(A * t) = P * \exp J * P^{-1}$$

where $\exp J$ is $\exp(J * t)$. J is the Jordan normal form of A and P is matrix such that:

$$A = P * J * P^{-1}$$

The matrix exponential $\exp(A * t)$ appears in the solution of linear differential equations. For example if x is a vector and A is a matrix then the initial value problem

$$\frac{dx(t)}{dt} = A \times x(t), x(0) = x_0$$

has the unique solution

$$x(t) = \exp(At)x_0$$

Examples

```
>>> from sympy import Symbol, Matrix, pprint
>>> from sympy.solvers.ode.systems import matrix_exp
>>> t = Symbol('t')
```

We will consider a 2x2 matrix for computing the exponential

```
>>> A = Matrix([[2, -5], [2, -4]])
>>> pprint(A)
[2  -5]
[   ]
[2  -4]
```

Now, $\exp(A*t)$ is given as follows:

```
>>> pprint(matrix_exp(A, t))
[  -t      -t      -t      ]
[3*e  *sin(t) + e  *cos(t)  -5*e  *sin(t)  ]
[                               ]
[  -t      -t      -t      ]
[ 2*e  *sin(t)  - 3*e  *sin(t) + e  *cos(t)]
```

See also:

[*matrix_exp_jordan_form*](#) (page 812)

For exponential of Jordan normal form

References

[R795], [R796]

Linear, n equations, Order 1 Solver

`sympy.solvers.ode.systems.linsolve(A, t, b=None, B=None, type='auto',
doit=False, tau=None)`

System of n equations linear first-order differential equations

Parameters

A : Matrix

Coefficient matrix of the system of linear first order ODEs.

t : Symbol

Independent variable in the system of ODEs.

b : Matrix or None

Non-homogeneous term in the system of ODEs. If None is passed, a homogeneous system of ODEs is assumed.

B : Matrix or None

Antiderivative of the coefficient matrix. If the antiderivative is not passed and the solution requires the term, then the solver would compute it internally.

type : String

Type of the system of ODEs passed. Depending on the type, the solution is evaluated. The type values allowed and the corresponding system it solves are: "type1" for constant coefficient homogeneous

“type2” for constant coefficient non-homogeneous, “type3” for non-constant coefficient homogeneous, “type4” for non-constant coefficient non-homogeneous, “type5” and “type6” for non-constant coefficient homogeneous and non-homogeneous systems respectively where the coefficient matrix can be factorized to a constant coefficient matrix. The default value is “auto” which will let the solver decide the correct type of the system passed.

doit : Boolean

Evaluate the solution if True, default value is False

tau: Expression

Used to substitute for the value of t after we get the solution of the system.

Returns

List

Raises

ValueError

This error is raised when the coefficient matrix, non-homogeneous term or the antiderivative, if passed, are not a matrix or do not have correct dimensions

NonSquareMatrixError

When the coefficient matrix or its antiderivative, if passed is not a square matrix

NotImplementedError

If the coefficient matrix does not have a commutative antiderivative

Explanation

This solver solves the system of ODEs of the following form:

$$X'(t) = A(t)X(t) + b(t)$$

Here, $A(t)$ is the coefficient matrix, $X(t)$ is the vector of n independent variables, $b(t)$ is the non-homogeneous term and $X'(t)$ is the derivative of $X(t)$

Depending on the properties of $A(t)$ and $b(t)$, this solver evaluates the solution differently.

When $A(t)$ is constant coefficient matrix and $b(t)$ is zero vector i.e. system is homogeneous, the system is “type1”. The solution is:

$$X(t) = \exp(At)C$$

Here, C is a vector of constants and A is the constant coefficient matrix.

When $A(t)$ is constant coefficient matrix and $b(t)$ is non-zero i.e. system is non-homogeneous, the system is “type2”. The solution is:

$$X(t) = e^{At} \left(\int e^{-At} b \, dt + C \right)$$

When $A(t)$ is coefficient matrix such that its commutative with its antiderivative $B(t)$ and $b(t)$ is a zero vector i.e. system is homogeneous, the system is “type3”. The solution is:

$$X(t) = \exp(B(t))C$$

When $A(t)$ is commutative with its antiderivative $B(t)$ and $b(t)$ is non-zero i.e. system is non-homogeneous, the system is “type4”. The solution is:

$$X(t) = e^{B(t)} \left(\int e^{-B(t)} b(t) dt + C \right)$$

When $A(t)$ is a coefficient matrix such that it can be factorized into a scalar and a constant coefficient matrix:

$$A(t) = f(t) * A$$

Where $f(t)$ is a scalar expression in the independent variable t and A is a constant matrix, then we can do the following substitutions:

$$\tau = \int f(t) dt, X(t) = Y(\tau), b(t) = b(f^{-1}(\tau))$$

Here, the substitution for the non-homogeneous term is done only when its non-zero. Using these substitutions, our original system becomes:

$$Y'(\tau) = A * Y(\tau) + b(\tau)/f(\tau)$$

The above system can be easily solved using the solution for “type1” or “type2” depending on the homogeneity of the system. After we get the solution for $Y(\tau)$, we substitute the solution for τ as t to get back $X(t)$

$$X(t) = Y(\tau)$$

Systems of “type5” and “type6” have a commutative antiderivative but we use this solution because its faster to compute.

The final solution is the general solution for all the four equations since a constant coefficient matrix is always commutative with its antiderivative.

An additional feature of this function is, if someone wants to substitute for value of the independent variable, they can pass the substitution τ and the solution will have the independent variable substituted with the passed expression(τ).

Examples

To solve the system of ODEs using this function directly, several things must be done in the right order. Wrong inputs to the function will lead to incorrect results.

```
>>> from sympy import symbols, Function, Eq
>>> from sympy.solvers.ode.systems import canonical_odes, linear_ode_to_
↳ matrix, linodesolve, linodesolve_type
>>> from sympy.solvers.ode.subscheck import checkodesol
>>> f, g = symbols("f, g", cls=Function)
>>> x, a = symbols("x, a")
>>> funcs = [f(x), g(x)]
>>> eqs = [Eq(f(x).diff(x) - f(x), a*g(x) + 1), Eq(g(x).diff(x) + g(x),
↳ a*f(x))]
```


Here, it is important to note that before we derive the coefficient matrix, it is important to get the system of ODEs into the desired form. For that we will use `sympy.solvers.ode.systems.canonical_odes()` (page 809).

```
>>> eqs = canonical_odes(eqs, funcs, x)
>>> eqs
[[Eq(Derivative(f(x), x), a*g(x) + f(x) + 1), Eq(Derivative(g(x), x),
↳a*f(x) - g(x))]]
```

Now, we will use `sympy.solvers.ode.systems.linear_ode_to_matrix()` (page 807) to get the coefficient matrix and the non-homogeneous term if it is there.

```
>>> eqs = eqs[0]
>>> (A1, A0), b = linear_ode_to_matrix(eqs, funcs, x, 1)
>>> A = A0
```

We have the coefficient matrices and the non-homogeneous term ready. Now, we can use `sympy.solvers.ode.systems.linodesolve_type()` (page 810) to get the information for the system of ODEs to finally pass it to the solver.

```
>>> system_info = linodesolve_type(A, x, b=b)
>>> sol_vector = linodesolve(A, x, b=b, B=system_info['antiderivative'],
↳type=system_info['type_of_equation'])
```

Now, we can prove if the solution is correct or not by using `sympy.solvers.ode.checkodesol()` (page 762)

```
>>> sol = [Eq(f, s) for f, s in zip(funcs, sol_vector)]
>>> checkodesol(eqs, sol)
(True, [0, 0])
```

We can also use the `doit` method to evaluate the solutions passed by the function.

```
>>> sol_vector_evaluated = linodesolve(A, x, b=b, type="type2",
↳doit=True)
```

Now, we will look at a system of ODEs which is non-constant.

```
>>> eqs = [Eq(f(x).diff(x), f(x) + x*g(x)), Eq(g(x).diff(x), -x*f(x) +
↳g(x))]
```

The system defined above is already in the desired form, so we do not have to convert it.

```
>>> (A1, A0), b = linear_ode_to_matrix(eqs, funcs, x, 1)
>>> A = A0
```

A user can also pass the commutative antiderivative required for `type3` and `type4` system of ODEs. Passing an incorrect one will lead to incorrect results. If the coefficient matrix is not commutative with its antiderivative, then `sympy.solvers.ode.systems.linodesolve_type()` (page 810) raises a `NotImplementedError`. If it does have a commutative antiderivative, then the function just returns the information about the system.

```
>>> system_info = linodesolve_type(A, x, b=b)
```

Now, we can pass the antiderivative as an argument to get the solution. If the system information is not passed, then the solver will compute the required arguments internally.

```
>>> sol_vector = linodesolve(A, x, b=b)
```

Once again, we can verify the solution obtained.

```
>>> sol = [Eq(f, s) for f, s in zip(funcs, sol_vector)]
>>> checkodesol(eqs, sol)
(True, [0, 0])
```

See also:

[***linear_ode_to_matrix***](#) (page 807)

Coefficient matrix computation function

[***canonical_odes***](#) (page 809)

System of ODEs representation change

[***linodesolve_type***](#) (page 810)

Getting information about systems of ODEs to pass in this solver

Nonlinear, 2 equations, Order 1, Type 1

`sympy.solvers.ode.ode._nonlinear_2eq_order1_type1(x, y, t, eq)`

Equations:

$$x' = x^n F(x, y)$$

$$y' = g(y) F(x, y)$$

Solution:

$$x = \varphi(y), \int \frac{1}{g(y)F(\varphi(y), y)} dy = t + C_2$$

where

if $n \neq 1$

$$\varphi = [C_1 + (1 - n) \int \frac{1}{g(y)} dy]^{\frac{1}{1-n}}$$

if $n = 1$

$$\varphi = C_1 e^{\int \frac{1}{g(y)} dy}$$

where C_1 and C_2 are arbitrary constants.

Nonlinear, 2 equations, Order 1, Type 2

`sympy.solvers.ode.ode._nonlinear_2eq_order1_type2(x, y, t, eq)`

Equations:

$$x' = e^{\lambda x} F(x, y)$$

$$y' = g(y) F(x, y)$$

Solution:

$$x = \varphi(y), \int \frac{1}{g(y)F(\varphi(y), y)} dy = t + C_2$$

where

if $\lambda \neq 0$

$$\varphi = -\frac{1}{\lambda} \log(C_1 - \lambda \int \frac{1}{g(y)} dy)$$

if $\lambda = 0$

$$\varphi = C_1 + \int \frac{1}{g(y)} dy$$

where C_1 and C_2 are arbitrary constants.

Nonlinear, 2 equations, Order 1, Type 3

`sympy.solvers.ode.ode._nonlinear_2eq_order1_type3(x, y, t, eq)`

Autonomous system of general form

$$x' = F(x, y)$$

$$y' = G(x, y)$$

Assuming $y = y(x, C_1)$ where C_1 is an arbitrary constant is the general solution of the first-order equation

$$F(x, y)y'_x = G(x, y)$$

Then the general solution of the original system of equations has the form

$$\int \frac{1}{F(x, y(x, C_1))} dx = t + C_1$$

Nonlinear, 2 equations, Order 1, Type 4

`sympy.solvers.ode.ode._nonlinear_2eq_order1_type4(x, y, t, eq)`

Equation:

$$x' = f_1(x)g_1(y)\phi(x, y, t)$$

$$y' = f_2(x)g_2(y)\phi(x, y, t)$$

First integral:

$$\int \frac{f_2(x)}{f_1(x)} dx - \int \frac{g_1(y)}{g_2(y)} dy = C$$

where C is an arbitrary constant.

On solving the first integral for x (resp., y) and on substituting the resulting expression into either equation of the original solution, one arrives at a first-order equation for determining y (resp., x).

Nonlinear, 2 equations, Order 1, Type 5

`sympy.solvers.ode.ode._nonlinear_2eq_order1_type5(func, t, eq)`

Clairaut system of ODEs

$$x = tx' + F(x', y')$$

$$y = ty' + G(x', y')$$

The following are solutions of the system

(i) straight lines:

$$x = C_1 t + F(C_1, C_2), y = C_2 t + G(C_1, C_2)$$

where C_1 and C_2 are arbitrary constants;

(ii) envelopes of the above lines;

(iii) continuously differentiable lines made up from segments of the lines (i) and (ii).

Nonlinear, 3 equations, Order 1, Type 1

`sympy.solvers.ode.ode._nonlinear_3eq_order1_type1(x, y, z, t, eq)`

Equations:

$$ax' = (b - c)yz, \quad by' = (c - a)zx, \quad cz' = (a - b)xy$$

First Integrals:

$$ax^2 + by^2 + cz^2 = C_1$$

$$a^2x^2 + b^2y^2 + c^2z^2 = C_2$$

where C_1 and C_2 are arbitrary constants. On solving the integrals for y and z and on substituting the resulting expressions into the first equation of the system, we arrive at a separable first-order equation on x . Similarly doing that for other two equations, we will arrive at first order equation on y and z too.

References

-<http://eqworld.ipmnet.ru/en/solutions/sysode/sode0401.pdf>

Nonlinear, 3 equations, Order 1, Type 2

`sympy.solvers.ode.ode._nonlinear_3eq_order1_type2(x, y, z, t, eq)`

Equations:

$$ax' = (b - c)yzf(x, y, z, t)$$

$$by' = (c - a)zxf(x, y, z, t)$$

$$cz' = (a - b)xyf(x, y, z, t)$$

First Integrals:

$$ax^2 + by^2 + cz^2 = C_1$$

$$a^2x^2 + b^2y^2 + c^2z^2 = C_2$$

where C_1 and C_2 are arbitrary constants. On solving the integrals for y and z and on substituting the resulting expressions into the first equation of the system, we arrive at a first-order differential equation on x . Similarly doing that for other two equations we will arrive at first order equation on y and z .

References

-<http://eqworld.ipmnet.ru/en/solutions/sysode/sode0402.pdf>

Nonlinear, 3 equations, Order 1, Type 3

`sympy.solvers.ode.ode._nonlinear_3eq_order1_type3(x, y, z, t, eq)`

Equations:

$$x' = cF_2 - bF_3, \quad y' = aF_3 - cF_1, \quad z' = bF_1 - aF_2$$

where $F_n = F_n(x, y, z, t)$.

1. First Integral:

$$ax + by + cz = C_1,$$

where C is an arbitrary constant.

2. If we assume function F_n to be independent of t , i.e., $F_n = F_n(x, y, z)$ Then, on eliminating t and z from the first two equations of the system, one arrives at the first-order equation

$$\frac{dy}{dx} = \frac{aF_3(x, y, z) - cF_1(x, y, z)}{cF_2(x, y, z) - bF_3(x, y, z)}$$

where $z = \frac{1}{c}(C_1 - ax - by)$

References

-<http://eqworld.ipmnet.ru/en/solutions/sysode/sode0404.pdf>

Nonlinear, 3 equations, Order 1, Type 4

`sympy.solvers.ode.ode._nonlinear_3eq_order1_type4(x, y, z, t, eq)`

Equations:

$$x' = czF_2 - byF_3, \quad y' = axF_3 - czF_1, \quad z' = byF_1 - axF_2$$

where $F_n = F_n(x, y, z, t)$

1. First integral:

$$ax^2 + by^2 + cz^2 = C_1$$

where C is an arbitrary constant.

2. Assuming the function F_n is independent of t : $F_n = F_n(x, y, z)$. Then on eliminating t and z from the first two equations of the system, one arrives at the first-order equation

$$\frac{dy}{dx} = \frac{axF_3(x, y, z) - czF_1(x, y, z)}{czF_2(x, y, z) - byF_3(x, y, z)}$$

where $z = \pm \sqrt{\frac{1}{c}(C_1 - ax^2 - by^2)}$

References

-<http://eqworld.ipmnet.ru/en/solutions/sysode/sode0405.pdf>

Nonlinear, 3 equations, Order 1, Type 5

`sympy.solvers.ode.ode._nonlinear_3eq_order1_type5(x, y, z, t, eq)`

$$x' = x(cF_2 - bF_3), \quad y' = y(aF_3 - cF_1), \quad z' = z(bF_1 - aF_2)$$

where $F_n = F_n(x, y, z, t)$ and are arbitrary functions.

First Integral:

$$|x|^a |y|^b |z|^c = C_1$$

where C is an arbitrary constant. If the function F_n is independent of t , then, by eliminating t and z from the first two equations of the system, one arrives at a first-order equation.

References

-<http://eqworld.ipmnet.ru/en/solutions/sysode/sode0406.pdf>

Information on the ode module

This module contains `dsolve()` (page 755) and different helper functions that it uses.

`dsolve()` (page 755) solves ordinary differential equations. See the docstring on the various functions for their uses. Note that partial differential equations support is in `pde.py`. Note that hint functions have docstrings describing their various methods, but they are intended for internal use. Use `dsolve(ode, func, hint=hint)` to solve an ODE using a specific hint. See also the docstring on `dsolve()` (page 755).

Functions in this module

These are the user functions in this module:

- `dsolve()` (page 755) - Solves ODEs.
- `classify_ode()` (page 760) - Classifies ODEs into possible hints for `dsolve()` (page 755).

- `checkodesol()` (page 762) - Checks if an equation is the solution to an ODE.
- `homogeneous_order()` (page 763) - Returns the homogeneous order of an expression.
- `infinitesimals()` (page 764) - Returns the infinitesimals of the Lie group of point transformations of an ODE, such that it is invariant.
- `checkinfsol()` (page 765) - Checks if the given infinitesimals are the actual infinitesimals of a first order ODE.

These are the non-solver helper functions that are for internal use. The user should use the various options to `dsolve()` (page 755) to obtain the functionality provided by these functions:

- `odesimp()` (page 767) - Does all forms of ODE simplification.
- `ode_sol_simplicity()` (page 769) - A key function for comparing solutions by simplicity.
- `constantsimp()` (page 765) - Simplifies arbitrary constants.
- `constant_renumber()` (page 768) - Renumber arbitrary constants.
- `_handle_Integral()` (page 826) - Evaluate unevaluated Integrals.

See also the docstrings of these functions.

Currently implemented solver methods

The following methods are implemented for solving ordinary differential equations. See the docstrings of the various hint functions for more information on each (run `help(ode)`):

- 1st order separable differential equations.
- 1st order differential equations whose coefficients or dx and dy are functions homogeneous of the same order.
- 1st order exact differential equations.
- 1st order linear differential equations.
- 1st order Bernoulli differential equations.
- Power series solutions for first order differential equations.
- Lie Group method of solving first order differential equations.
- 2nd order Liouville differential equations.
- Power series solutions for second order differential equations at ordinary and regular singular points.
- n th order differential equation that can be solved with algebraic rearrangement and integration.
- n th order linear homogeneous differential equation with constant coefficients.
- n th order linear inhomogeneous differential equation with constant coefficients using the method of undetermined coefficients.
- n th order linear inhomogeneous differential equation with constant coefficients using the method of variation of parameters.

Philosophy behind this module

This module is designed to make it easy to add new ODE solving methods without having to mess with the solving code for other methods. The idea is that there is a `classify_ode()` (page 760) function, which takes in an ODE and tells you what hints, if any, will solve the ODE. It does this without attempting to solve the ODE, so it is fast. Each solving method is a hint, and it has its own function, named `ode_<hint>`. That function takes in the ODE and any match expression gathered by `classify_ode()` (page 760) and returns a solved result. If this result has any integrals in it, the hint function will return an unevaluated `Integral` (page 601) class. `dsolve()` (page 755), which is the user wrapper function around all of this, will then call `odesimp()` (page 767) on the result, which, among other things, will attempt to solve the equation for the dependent variable (the function we are solving for), simplify the arbitrary constants in the expression, and evaluate any integrals, if the hint allows it.

How to add new solution methods

If you have an ODE that you want `dsolve()` (page 755) to be able to solve, try to avoid adding special case code here. Instead, try finding a general method that will solve your ODE, as well as others. This way, the `ode` (page 755) module will become more robust, and unhindered by special case hacks. WolphramAlpha and Maple's DETools[odeadvisor] function are two resources you can use to classify a specific ODE. It is also better for a method to work with an n th order ODE instead of only with specific orders, if possible.

To add a new method, there are a few things that you need to do. First, you need a hint name for your method. Try to name your hint so that it is unambiguous with all other methods, including ones that may not be implemented yet. If your method uses integrals, also include a `hint_Integral` hint. If there is more than one way to solve ODEs with your method, include a hint for each one, as well as a `<hint>_best` hint. Your `ode_<hint>_best()` function should choose the best using `min` with `ode_sol_simplicity` as the key argument. See `HomogeneousCoeffBest` (page 772), for example. The function that uses your method will be called `ode_<hint>()`, so the hint must only use characters that are allowed in a Python function name (alphanumeric characters and the underscore `'_'` character). Include a function for every hint, except for `_Integral` hints (`dsolve()` (page 755) takes care of those automatically). Hint names should be all lowercase, unless a word is commonly capitalized (such as `Integral` or `Bernoulli`). If you have a hint that you do not want to run with `all_Integral` that does not have an `_Integral` counterpart (such as a `best` hint that would defeat the purpose of `all_Integral`), you will need to remove it manually in the `dsolve()` (page 755) code. See also the `classify_ode()` (page 760) docstring for guidelines on writing a hint name.

Determine *in general* how the solutions returned by your method compare with other methods that can potentially solve the same ODEs. Then, put your hints in the `allhints` (page 767) tuple in the order that they should be called. The ordering of this tuple determines which hints are default. Note that exceptions are ok, because it is easy for the user to choose individual hints with `dsolve()` (page 755). In general, `_Integral` variants should go at the end of the list, and `_best` variants should go before the various hints they apply to. For example, the `undetermined_coefficients` hint comes before the `variation_of_parameters` hint because, even though variation of parameters is more general than undetermined coefficients, undetermined coefficients generally returns cleaner results for the ODEs that it can solve than variation of parameters does, and it does not require integration, so it is much faster.

Next, you need to have a match expression or a function that matches the type of the ODE, which you should put in `classify_ode()` (page 760) (if the match function is more than just a few lines. It should match the ODE without solving for it as much as possible, so that `classify_ode()` (page 760) remains fast and is not hindered by bugs in solving code. Be sure to consider corner cases. For example, if your solution method involves dividing by something, make sure you exclude the case where that division will be 0.

In most cases, the matching of the ODE will also give you the various parts that you need to solve it. You should put that in a dictionary (`.match()` will do this for

you), and add that as `matching_hints['hint'] = matchdict` in the relevant part of `classify_ode()` (page 760). `classify_ode()` (page 760) will then send this to `dsolve()` (page 755), which will send it to your function as the `match` argument. Your function should be named `ode_<hint>(eq, func, order, match)`. If you need to send more information, put it in the `match` dictionary. For example, if you had to substitute in a dummy variable in `classify_ode()` (page 760) to match the ODE, you will need to pass it to your function using the `match` dict to access it. You can access the independent variable using `func.args[0]`, and the dependent variable (the function you are trying to solve for) as `func.func`. If, while trying to solve the ODE, you find that you cannot, raise `NotImplementedError`. `dsolve()` (page 755) will catch this error with the `all` meta-hint, rather than causing the whole routine to fail.

Add a docstring to your function that describes the method employed. Like with anything else in SymPy, you will need to add a doctest to the docstring, in addition to real tests in `test_ode.py`. Try to maintain consistency with the other hint functions' docstrings. Add your method to the list at the top of this docstring. Also, add your method to `ode.rst` in the `docs/src` directory, so that the Sphinx docs will pull its docstring into the main SymPy documentation. Be sure to make the Sphinx documentation by running `make html` from within the doc directory to verify that the docstring formats correctly.

If your solution method involves integrating, use `Integral` (page 601) instead of `integrate()` (page 964). This allows the user to bypass hard/slow integration by using the `_Integral` variant of your hint. In most cases, calling `sympy.core.basic.Basic.doit()` (page 932) will integrate your solution. If this is not the case, you will need to write special code in `_handle_Integral()` (page 826). Arbitrary constants should be symbols named `C1`, `C2`, and so on. All solution methods should return an equality instance. If you need an arbitrary number of arbitrary constants, you can use `constants = numbered_symbols(prefix='C', cls=Symbol, start=1)`. If it is possible to solve for the dependent function in a general way, do so. Otherwise, do as best as you can, but do not call `solve` in your `ode_<hint>()` function. `odesimp()` (page 767) will attempt to solve the solution for you, so you do not need to do that. Lastly, if your ODE has a common simplification that can be applied to your solutions, you can add a special case in `odesimp()` (page 767) for it. For example, solutions returned from the `1st_homogeneous_coeff` hints often have many `log` (page 413) terms, so `odesimp()` (page 767) calls `logcombine()` (page 669) on them (it also helps to write the arbitrary constant as `log(C1)` instead of `C1` in this case). Also consider common ways that you can rearrange your solution to have `constantsimp()` (page 765) take better advantage of it. It is better to put simplification in `odesimp()` (page 767) than in your method, because it can then be turned off with the `simplify` flag in `dsolve()` (page 755). If you have any extraneous simplification in your function, be sure to only run it using `if match.get('simplify', True):`, especially if it can be slow or if it can reduce the domain of the solution.

Finally, as with every contribution to SymPy, your method will need to be tested. Add a test for each method in `test_ode.py`. Follow the conventions there, i.e., test the solver using `dsolve(eq, f(x), hint=your_hint)`, and also test the solution using `checkodesol()` (page 762) (you can put these in a separate tests and skip/XFAIL if it runs too slow/does not work). Be sure to call your hint specifically in `dsolve()` (page 755), that way the test will not be broken simply by the introduction of another matching hint. If your method works for higher order (>1) ODEs, you will need to run `sol = constant_renumber(sol, 'C', 1, order)` for each solution, where `order` is the order of the ODE. This is because `constant_renumber` renumbers the arbitrary constants by printing order, which is platform dependent. Try to test every corner case of your solver, including a range of orders if it is a n th order solver, but if your solver is slow, such as if it involves hard integration, try to keep the test run time down.

Feel free to refactor existing hints to avoid duplicating code or creating inconsistencies. If you can show that your method exactly duplicates an existing method, including in the simplicity

and speed of obtaining the solutions, then you can remove the old, less general method. The existing code is tested extensively in `test_ode.py`, so if anything is broken, one of those tests will surely fail.

Internal functions

These functions are not intended for end-user use.

`sympy.solvers.ode.ode._handle_Integral(expr, func, hint)`

Converts a solution with Integrals in it into an actual solution.

For most hints, this simply runs `expr.doit()`.

PDE

User Functions

These are functions that are imported into the global namespace with `from sympy import *`. They are intended for user use.

pde_separate

`sympy.solvers.pde.pde_separate(eq, fun, sep, strategy='mul')`

Separate variables in partial differential equation either by additive or multiplicative separation approach. It tries to rewrite an equation so that one of the specified variables occurs on a different side of the equation than the others.

Parameters

- **eq** – Partial differential equation
- **fun** – Original function $F(x, y, z)$
- **sep** – List of separated functions $[X(x), u(y, z)]$
- **strategy** – Separation strategy. You can choose between additive separation ('add') and multiplicative separation ('mul') which is default.

Examples

```
>>> from sympy import E, Eq, Function, pde_separate, Derivative as D
>>> from sympy.abc import x, t
>>> u, X, T = map(Function, 'uXT')
```

```
>>> eq = Eq(D(u(x, t), x), E**(u(x, t))*D(u(x, t), t))
>>> pde_separate(eq, u(x, t), [X(x), T(t)], strategy='add')
[exp(-X(x))*Derivative(X(x), x), exp(T(t))*Derivative(T(t), t)]
```

```
>>> eq = Eq(D(u(x, t), x, 2), D(u(x, t), t, 2))
>>> pde_separate(eq, u(x, t), [X(x), T(t)], strategy='mul')
[Derivative(X(x), (x, 2))/X(x), Derivative(T(t), (t, 2))/T(t)]
```

See also:

[pde_separate_add](#) (page 827), [pde_separate_mul](#) (page 827)

pde_separate_add

`sympy.solvers.pde.pde_separate_add(eq, fun, sep)`

Helper function for searching additive separable solutions.

Consider an equation of two independent variables x, y and a dependent variable w , we look for the product of two functions depending on different arguments:

$$w(x, y, z) = X(x) + y(y, z)$$

Examples

```
>>> from sympy import E, Eq, Function, pde_separate_add, Derivative as D
>>> from sympy.abc import x, t
>>> u, X, T = map(Function, 'uXT')
```

```
>>> eq = Eq(D(u(x, t), x), E**(u(x, t))*D(u(x, t), t))
>>> pde_separate_add(eq, u(x, t), [X(x), T(t)])
[exp(-X(x))*Derivative(X(x), x), exp(T(t))*Derivative(T(t), t)]
```

pde_separate_mul

`sympy.solvers.pde.pde_separate_mul(eq, fun, sep)`

Helper function for searching multiplicative separable solutions.

Consider an equation of two independent variables x, y and a dependent variable w , we look for the product of two functions depending on different arguments:

$$w(x, y, z) = X(x) * u(y, z)$$

Examples

```
>>> from sympy import Function, Eq, pde_separate_mul, Derivative as D
>>> from sympy.abc import x, y
>>> u, X, Y = map(Function, 'uXY')
```

```
>>> eq = Eq(D(u(x, y), x, 2), D(u(x, y), y, 2))
>>> pde_separate_mul(eq, u(x, y), [X(x), Y(y)])
[Derivative(X(x), (x, 2))/X(x), Derivative(Y(y), (y, 2))/Y(y)]
```

pdsolve

`sympy.solvers.pde.pdsolve(eq, func=None, hint='default', dict=False, solvefun=None, **kwargs)`

Solves any (supported) kind of partial differential equation.

Usage

`pdsolve(eq, f(x,y), hint)` -> Solve partial differential equation `eq` for function `f(x,y)`, using method `hint`.

Details

eq can be any supported partial differential equation (see the pde docstring for supported methods). This can either be an Equality, or an expression, which is assumed to be equal to 0.

f(x,y) is a function of two variables whose derivatives in that variable make up the partial differential equation. In many cases it is not necessary to provide this; it will be autodetected (and an error raised if it could not be detected).

hint is the solving method that you want pdsolve to use. Use `classify_pde(eq, f(x,y))` to get all of the possible hints for a PDE. The default hint, 'default', will use whatever hint is returned first by `classify_pde()`. See Hints below for more options that you can use for hint.

solvefun is the convention used for arbitrary functions returned by the PDE solver. If not set by the user, it is set by default to be `F`.

Hints

Aside from the various solving methods, there are also some meta-hints that you can pass to `pdsolve()`:

“default”:

This uses whatever hint is returned first by `classify_pde()`. This is the default argument to `pdsolve()`.

“all”:

To make `pdsolve` apply all relevant classification hints, use `pdsolve(PDE, func, hint="all")`. This will return a dictionary of hint:solution terms. If a hint causes `pdsolve` to raise the `NotImplementedError`, value of that hint's key will be the exception object raised. The dictionary will also include some special keys:

- `order`: The order of the PDE. See also `ode_order()` in `deutils.py`
- `default`: The solution that would be returned by default. This is the one produced by the hint that appears first in the tuple returned by `classify_pde()`.

“all_Integral”:

This is the same as “all”, except if a hint also has a corresponding “_Integral” hint, it only returns the “_Integral” hint. This is useful if “all” causes `pdsolve()` to hang because of a difficult or impossible integral. This meta-hint will also be much faster than “all”, because `integrate()` is an expensive routine.

See also the `classify_pde()` docstring for more info on hints, and the pde docstring for a list of all supported hints.

Tips

- You can declare the derivative of an unknown function this way:

```
>>> from sympy import Function, Derivative
>>> from sympy.abc import x, y # x and y are the independent
    ↪ variables
>>> f = Function("f")(x, y) # f is a function of x and y
>>> # fx will be the partial derivative of f with respect to x
>>> fx = Derivative(f, x)
>>> # fy will be the partial derivative of f with respect to y
>>> fy = Derivative(f, y)
```

- See test_pde.py for many tests, which serves also as a set of examples for how to use pdsolve().
- pdsolve always returns an Equality class (except for the case when the hint is "all" or "all_Integral"). Note that it is not possible to get an explicit solution for $f(x, y)$ as in the case of ODE's
- Do `help(pde.pde_hintname)` to get help more information on a specific hint

Examples

```
>>> from sympy.solvers.pde import pdsolve
>>> from sympy import Function, Eq
>>> from sympy.abc import x, y
>>> f = Function('f')
>>> u = f(x, y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> eq = Eq(1 + (2*(ux/u)) + (3*(uy/u)), 0)
>>> pdsolve(eq)
Eq(f(x, y), F(3*x - 2*y)*exp(-2*x/13 - 3*y/13))
```

classify_pde

`sympy.solvers.pde.classify_pde(eq, func=None, dict=False, *, prep=True, **kwargs)`

Returns a tuple of possible pdsolve() classifications for a PDE.

The tuple is ordered so that first item is the classification that pdsolve() uses to solve the PDE by default. In general, classifications near the beginning of the list will produce better solutions faster than those near the end, though there are always exceptions. To make pdsolve use a different classification, use `pdsolve(PDE, func, hint=<classification>)`. See also the pdsolve() docstring for different meta-hints you can use.

If `dict` is true, `classify_pde()` will return a dictionary of hint:match expression terms. This is intended for internal use by pdsolve(). Note that because dictionaries are ordered arbitrarily, this will most likely not be in the same order as the tuple.

You can get help on different hints by doing `help(pde.pde_hintname)`, where `hintname` is the name of the hint without `"_Integral"`.

See `sympy.pde.allhints` or the `sympy.pde` docstring for a list of all supported hints that can be returned from `classify_pde`.

Examples

```
>>> from sympy.solvers.pde import classify_pde
>>> from sympy import Function, Eq
>>> from sympy.abc import x, y
>>> f = Function('f')
>>> u = f(x, y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> eq = Eq(1 + (2*(ux/u)) + (3*(uy/u)), 0)
>>> classify_pde(eq)
('1st_linear_constant_coeff_homogeneous',)
```

checkpdesol

`sympy.solvers.pde.checkpdesol(pde, sol, func=None, solve_for_func=True)`

Checks if the given solution satisfies the partial differential equation.

`pde` is the partial differential equation which can be given in the form of an equation or an expression. `sol` is the solution for which the `pde` is to be checked. This can also be given in an equation or an expression form. If the function is not provided, the helper function `_preprocess` from `deutils` is used to identify the function.

If a sequence of solutions is passed, the same sort of container will be used to return the result for each solution.

The following methods are currently being implemented to check if the solution satisfies the PDE:

1. Directly substitute the solution in the PDE and check. If the solution has not been solved for `f`, then it will solve for `f` provided `solve_for_func` has not been set to `False`.

If the solution satisfies the PDE, then a tuple `(True, 0)` is returned. Otherwise a tuple `(False, expr)` where `expr` is the value obtained after substituting the solution in the PDE. However if a known solution returns `False`, it may be due to the inability of `doit()` to simplify it to zero.

Examples

```
>>> from sympy import Function, symbols
>>> from sympy.solvers.pde import checkpdesol, pdsolve
>>> x, y = symbols('x y')
>>> f = Function('f')
>>> eq = 2*f(x,y) + 3*f(x,y).diff(x) + 4*f(x,y).diff(y)
>>> sol = pdsolve(eq)
>>> assert checkpdesol(eq, sol)[0]
>>> eq = x*f(x,y) + f(x,y).diff(x)
>>> checkpdesol(eq, sol)
(False, (x**F(4*x - 3*y) - 6*F(4*x - 3*y)/25 + 4*Subs(Derivative(F(_xi_1),
→ _xi_1), _xi_1, 4*x - 3*y))*exp(-6*x/25 - 8*y/25))
```

Hint Methods

These functions are meant for internal use. However they contain useful information on the various solving methods.

pde_1st_linear_constant_coeff_homogeneous

`sympy.solvers.pde.pde_1st_linear_constant_coeff_homogeneous(eq, func, order, match, solvefun)`

Solves a first order linear homogeneous partial differential equation with constant coefficients.

The general form of this partial differential equation is

$$a \frac{\partial f(x, y)}{\partial x} + b \frac{\partial f(x, y)}{\partial y} + cf(x, y) = 0$$

where a , b and c are constants.

The general solution is of the form:

$$f(x, y) = F(-ay + bx)e^{-\frac{c(ax+by)}{a^2+b^2}}$$

and can be found in SymPy with `pdsolve`:

```
>>> from sympy.solvers import pdsolve
>>> from sympy.abc import x, y, a, b, c
>>> from sympy import Function, pprint
>>> f = Function('f')
>>> u = f(x,y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> genform = a*ux + b*uy + c*u
>>> pprint(genform)
      d      d
a*--(f(x, y)) + b*--(f(x, y)) + c*f(x, y)
  dx      dy

>>> pprint(pdsolve(genform))
      -c*(a*x + b*y)
      -----
          2      2
         a  + b

f(x, y) = F(-a*y + b*x)*e
```

Examples

```
>>> from sympy import pdsolve
>>> from sympy import Function, pprint
>>> from sympy.abc import x,y
>>> f = Function('f')
>>> pdsolve(f(x,y) + f(x,y).diff(x) + f(x,y).diff(y))
Eq(f(x, y), F(x - y)*exp(-x/2 - y/2))
>>> pprint(pdsolve(f(x,y) + f(x,y).diff(x) + f(x,y).diff(y)))
      x      y
      -      -
      2      2
f(x, y) = F(x - y)*e
```

References

- Viktor Grigoryan, “Partial Differential Equations” Math 124A - Fall 2010, pp.7

pde_1st_linear_constant_coeff

`sympy.solvers.pde.pde_1st_linear_constant_coeff(eq, func, order, match, solvefun)`

Solves a first order linear partial differential equation with constant coefficients.

The general form of this partial differential equation is

$$a \frac{\partial f(x,y)}{\partial x} + b \frac{\partial f(x,y)}{\partial y} + cf(x,y) = G(x,y)$$

where a , b and c are constants and $G(x,y)$ can be an arbitrary function in x and y .

The general solution of the PDE is:

$$f(x,y) = \left[F(\eta) + \frac{1}{a^2 + b^2} \int^{ax+by} G\left(\frac{a\xi + b\eta}{a^2 + b^2}, \frac{-a\eta + b\xi}{a^2 + b^2}\right) e^{\frac{c\xi}{a^2 + b^2}} d\xi \right] e^{-\frac{c\xi}{a^2 + b^2}} \bigg|_{\substack{\eta = -ay + bx \\ \xi = ax + by}},$$

where $F(\eta)$ is an arbitrary single-valued function. The solution can be found in SymPy with `pdsolve`:

```
>>> from sympy.solvers import pdsolve
>>> from sympy.abc import x, y, a, b, c
>>> from sympy import Function, pprint
>>> f = Function('f')
>>> G = Function('G')
>>> u = f(x,y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> genform = a*ux + b*uy + c*u - G(x,y)
>>> pprint(genform)
      d      d
a*--(f(x, y)) + b*--(f(x, y)) + c*f(x, y) - G(x, y)
```

(continues on next page)

833

Examples

```
>>> from sympy.solvers.pde import pdsolve
>>> from sympy import Function, pprint, exp
>>> from sympy.abc import x,y
>>> f = Function('f')
>>> eq = -2*f(x,y).diff(x) + 4*f(x,y).diff(y) + 5*f(x,y) - exp(x + 3*y)
>>> pdsolve(eq)
Eq(f(x, y), (F(4*x + 2*y)*exp(x/2) + exp(x + 4*y)/15)*exp(-y))
```

References

- Viktor Grigoryan, “Partial Differential Equations” Math 124A - Fall 2010, pp.7

pde_1st_linear_variable_coeff

`sympy.solvers.pde.pde_1st_linear_variable_coeff(eq, func, order, match, solvefun)`

Solves a first order linear partial differential equation with variable coefficients. The general form of this partial differential equation is

$$a(x,y)\frac{\partial f(x,y)}{\partial x} + b(x,y)\frac{\partial f(x,y)}{\partial y} + c(x,y)f(x,y) = G(x,y)$$

where $a(x,y)$, $b(x,y)$, $c(x,y)$ and $G(x,y)$ are arbitrary functions in x and y . This PDE is converted into an ODE by making the following transformation:

1. ξ as x
2. η as the constant in the solution to the differential equation $\frac{dy}{dx} = -\frac{b}{a}$

Making the previous substitutions reduces it to the linear ODE

$$a(\xi,\eta)\frac{du}{d\xi} + c(\xi,\eta)u - G(\xi,\eta) = 0$$

which can be solved using `dsolve`.

```
>>> from sympy.abc import x, y
>>> from sympy import Function, pprint
>>> a, b, c, G, f= [Function(i) for i in ['a', 'b', 'c', 'G', 'f']]
>>> u = f(x,y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> genform = a(x, y)*u + b(x, y)*ux + c(x, y)*uy - G(x,y)
>>> pprint(genform)
      d      d
-G(x, y) + a(x, y)*f(x, y) + b(x, y)*--(f(x, y)) + c(x, y)*--(f(x, y))
      dx      dy
```

Examples

```
>>> from sympy.solvers.pde import pdsolve
>>> from sympy import Function, pprint
>>> from sympy.abc import x,y
>>> f = Function('f')
>>> eq = x*(u.diff(x)) - y*(u.diff(y)) + y**2*u - y**2
>>> pdsolve(eq)
Eq(f(x, y), F(x*y)*exp(y**2/2) + 1)
```

References

- Viktor Grigoryan, “Partial Differential Equations” Math 124A - Fall 2010, pp.7

Information on the pde module

This module contains `pdsolve()` and different helper functions that it uses. It is heavily inspired by the `ode` module and hence the basic infrastructure remains the same.

Functions in this module

These are the user functions in this module:

- `pdsolve()` - Solves PDE's
- `classify_pde()` - Classifies PDEs into possible hints for `dsolve()`.
- **`pde_separate()` - Separate variables in partial differential equation either by additive or multiplicative separation approach.**

These are the helper functions in this module:

- `pde_separate_add()` - Helper function for searching additive separable solutions.
- **`pde_separate_mul()` - Helper function for searching multiplicative separable solutions.**

Currently implemented solver methods

The following methods are implemented for solving partial differential equations. See the docstrings of the various `pde_hint()` functions for more information on each (run `help(pde)`):

- 1st order linear homogeneous partial differential equations with constant coefficients.
- 1st order linear general partial differential equations with constant coefficients.
- 1st order linear partial differential equations with variable coefficients.

Solvers

The *solvers* module in SymPy implements methods for solving equations.

Note: `solve()` (page 836) is an older more mature general function for solving many types of equations. `solve()` (page 836) has many options and uses different methods internally to determine what type of equations you pass it, so if you know what type of equation you are dealing with you may want to use the newer `solveset()` (page 858) which solves univariate equations, `linsolve()` (page 872) which solves system of linear equations, and `nonlinsolve()` (page 875) which solves systems of non linear equations.

Algebraic equations

Use `solve()` (page 836) to solve algebraic equations. We suppose all equations are equaled to 0, so solving `x**2 == 1` translates into the following code:

```
>>> from sympy.solvers import solve
>>> from sympy import Symbol
>>> x = Symbol('x')
>>> solve(x**2 - 1, x)
[-1, 1]
```

The first argument for `solve()` (page 836) is an equation (equaled to zero) and the second argument is the symbol that we want to solve the equation for.

`sympy.solvers.solvers.solve(f, *symbols, **flags)`

Algebraically solves equations and systems of equations.

Parameters

f :

- a single Expr or Poly that must be zero
- an Equality
- a Relational expression
- a Boolean
- iterable of one or more of the above

symbols : (object(s) to solve for) specified as

- none given (other non-numeric objects will be used)
- single symbol
- denested list of symbols (e.g., `solve(f, x, y)`)
- ordered iterable of symbols (e.g., `solve(f, [x, y])`)

flags :

dict=True (default is False)

Return list (perhaps empty) of solution mappings.

set=True (default is False)

Return list of symbols and set of tuple(s) of solution(s).

exclude=[] (default)

Do not try to solve for any of the free symbols in exclude; if expressions are given, the free symbols in them will be extracted automatically.

check=True (default)

If False, do not do any testing of solutions. This can be useful if you want to include solutions that make any denominator zero.

numerical=True (default)

Do a fast numerical check if f has only one symbol.

minimal=True (default is False)

A very fast, minimal testing.

warn=True (default is False)

Show a warning if `checksol()` could not conclude.

simplify=True (default)

Simplify all but polynomials of order 3 or greater before returning them and (if check is not False) use the general simplify function on the solutions and the expression obtained when they are substituted into the function which should be zero.

force=True (default is False)

Make positive all symbols without assumptions regarding sign.

rational=True (default)

Recast Floats as Rational; if this option is not used, the system containing Floats may fail to solve because of issues with polys. If `rational=None`, Floats will be recast as rationals but the answer will be recast as Floats. If the flag is False then nothing will be done to the Floats.

manual=True (default is False)

Do not use the polys/matrix method to solve a system of equations, solve them one at a time as you might “manually.”

implicit=True (default is False)

Allows solve to return a solution for a pattern in terms of other functions that contain that pattern; this is only needed if the pattern is inside of some invertible function like cos, exp, ect.

particular=True (default is False)

Instructs solve to try to find a particular solution to a linear system with as many zeros as possible; this is very expensive.

quick=True (default is False; particular must be True)

Selects a fast heuristic to find a solution with many zeros whereas a value of False uses the very slow method guaranteed to find the largest number of zeros possible.

cubics=True (default)

Return explicit solutions when cubic expressions are encountered. When False, quartics and quintics are disabled, too.

quartics=True (default)

Return explicit solutions when quartic expressions are encountered. When False, quintics are disabled, too.

quintics=True (default)

Return explicit solutions (if possible) when quintic expressions are encountered.

Explanation**Currently supported:**

- polynomial
- transcendental
- piecewise combinations of the above
- systems of linear and polynomial equations
- systems containing relational expressions

Examples

The output varies according to the input and can be seen by example:

```
>>> from sympy import solve, Poly, Eq, Function, exp
>>> from sympy.abc import x, y, z, a, b
>>> f = Function('f')
```

Boolean or univariate Relational:

```
>>> solve(x < 3)
(-oo < x) & (x < 3)
```

To always get a list of solution mappings, use flag dict=True:

```
>>> solve(x - 3, dict=True)
[{x: 3}]
>>> sol = solve([x - 3, y - 1], dict=True)
>>> sol
[{x: 3, y: 1}]
>>> sol[0][x]
3
>>> sol[0][y]
1
```

To get a list of *symbols* and set of solution(s) use flag set=True:

```
>>> solve([x**2 - 3, y - 1], set=True)
([x, y], {(-sqrt(3), 1), (sqrt(3), 1)})
```

Single expression and single symbol that is in the expression:

```
>>> solve(x - y, x)
[y]
>>> solve(x - 3, x)
[3]
```

(continues on next page)

(continued from previous page)

```
>>> solve(Eq(x, 3), x)
[3]
>>> solve(Poly(x - 3), x)
[3]
>>> solve(x**2 - y**2, x, set=True)
([x], {(-y,), (y,)})
>>> solve(x**4 - 1, x, set=True)
([x], {(-1,), (1,), (-I,), (I,)})
```

Single expression with no symbol that is in the expression:

```
>>> solve(3, x)
[]
>>> solve(x - 3, y)
[]
```

Single expression with no symbol given. In this case, all free *symbols* will be selected as potential *symbols* to solve for. If the equation is univariate then a list of solutions is returned; otherwise - as is the case when *symbols* are given as an iterable of length greater than 1 - a list of mappings will be returned:

```
>>> solve(x - 3)
[3]
>>> solve(x**2 - y**2)
[{x: -y}, {x: y}]
>>> solve(z**2*x**2 - z**2*y**2)
[{x: -y}, {x: y}, {z: 0}]
>>> solve(z**2*x - z**2*y**2)
[{x: y**2}, {z: 0}]
```

When an object other than a Symbol is given as a symbol, it is isolated algebraically and an implicit solution may be obtained. This is mostly provided as a convenience to save you from replacing the object with a Symbol and solving for that Symbol. It will only work if the specified object can be replaced with a Symbol using the subs method:

```
>>> solve(f(x) - x, f(x))
[x]
>>> solve(f(x).diff(x) - f(x) - x, f(x).diff(x))
[x + f(x)]
>>> solve(f(x).diff(x) - f(x) - x, f(x))
[-x + Derivative(f(x), x)]
>>> solve(x + exp(x)**2, exp(x), set=True)
([exp(x)], {(-sqrt(-x),), (sqrt(-x),)})
```

```
>>> from sympy import Indexed, IndexedBase, Tuple, sqrt
>>> A = IndexedBase('A')
>>> eqs = Tuple(A[1] + A[2] - 3, A[1] - A[2] + 1)
>>> solve(eqs, eqs.atoms(Indexed))
{A[1]: 1, A[2]: 2}
```

- To solve for a symbol implicitly, use `implicit=True`:

```
>>> solve(x + exp(x), x)
[-LambertW(1)]
>>> solve(x + exp(x), x, implicit=True)
[-exp(x)]
```

- It is possible to solve for anything that can be targeted with subs:

```
>>> solve(x + 2 + sqrt(3), x + 2)
[-sqrt(3)]
>>> solve((x + 2 + sqrt(3), x + 4 + y), y, x + 2)
{y: -2 + sqrt(3), x + 2: -sqrt(3)}
```

- Nothing heroic is done in this implicit solving so you may end up with a symbol still in the solution:

```
>>> eqs = (x*y + 3*y + sqrt(3), x + 4 + y)
>>> solve(eqs, y, x + 2)
{y: -sqrt(3)/(x + 3), x + 2: -2*x/(x + 3) - 6/(x + 3) + sqrt(3)/(x +
↪ 3)}
>>> solve(eqs, y*x, x)
{x: -y - 4, x*y: -3*y - sqrt(3)}
```

- If you attempt to solve for a number remember that the number you have obtained does not necessarily mean that the value is equivalent to the expression obtained:

```
>>> solve(sqrt(2) - 1, 1)
[sqrt(2)]
>>> solve(x - y + 1, 1) # /\ -1 is targeted, too
[x/(y - 1)]
>>> [_subs(z, -1) for _ in solve((x - y + 1).subs(-1, z), 1)]
[-x + y]
```

- To solve for a function within a derivative, use dsolve.

Single expression and more than one symbol:

- When there is a linear solution:

```
>>> solve(x - y**2, x, y)
[(y**2, y)]
>>> solve(x**2 - y, x, y)
[(x, x**2)]
>>> solve(x**2 - y, x, y, dict=True)
[{y: x**2}]
```

- When undetermined coefficients are identified:

- That are linear:

```
>>> solve((a + b)*x - b + 2, a, b)
{a: -2, b: 2}
```

- That are nonlinear:


```
>>> solve((a + b)*x - b**2 + 2, a, b, set=True)
([a, b], {(-sqrt(2), sqrt(2)), (sqrt(2), -sqrt(2))})
```

- If there is no linear solution, then the first successful attempt for a nonlinear solution will be returned:

```
>>> solve(x**2 - y**2, x, y, dict=True)
[{x: -y}, {x: y}]
>>> solve(x**2 - y**2/exp(x), x, y, dict=True)
[{x: 2*LambertW(-y/2)}, {x: 2*LambertW(y/2)}]
>>> solve(x**2 - y**2/exp(x), y, x)
[(-x*sqrt(exp(x)), x), (x*sqrt(exp(x)), x)]
```

Iterable of one or more of the above:

- Involving relationals or bools:

```
>>> solve([x < 3, x - 2])
Eq(x, 2)
>>> solve([x > 3, x - 2])
False
```

- When the system is linear:

- With a solution:

```
>>> solve([x - 3], x)
{x: 3}
>>> solve((x + 5*y - 2, -3*x + 6*y - 15), x, y)
{x: -3, y: 1}
>>> solve((x + 5*y - 2, -3*x + 6*y - 15), x, y, z)
{x: -3, y: 1}
>>> solve((x + 5*y - 2, -3*x + 6*y - z), z, x, y)
{x: 2 - 5*y, z: 21*y - 6}
```

- Without a solution:

```
>>> solve([x + 3, x - 3])
[]
```

- When the system is not linear:

```
>>> solve([x**2 + y - 2, y**2 - 4], x, y, set=True)
([x, y], {(-2, -2), (0, 2), (2, -2)})
```

- If no *symbols* are given, all free *symbols* will be selected and a list of mappings returned:

```
>>> solve([x - 2, x**2 + y])
[{x: 2, y: -4}]
>>> solve([x - 2, x**2 + f(x)], {f(x), x})
[{x: 2, f(x): -4}]
```

- If any equation does not depend on the symbol(s) given, it will be eliminated from the equation set and an answer may be given implicitly in terms of variables that were not of interest:

```
>>> solve([x - y, y - 3], x)
{x: y}
```

Additional Examples

`solve()` with `check=True` (default) will run through the symbol tags to eliminate unwanted solutions. If no assumptions are included, all possible solutions will be returned:

```
>>> from sympy import Symbol, solve
>>> x = Symbol("x")
>>> solve(x**2 - 1)
[-1, 1]
```

By using the positive tag, only one solution will be returned:

```
>>> pos = Symbol("pos", positive=True)
>>> solve(pos**2 - 1)
[1]
```

Assumptions are not checked when `solve()` input involves relationals or bools.

When the solutions are checked, those that make any denominator zero are automatically excluded. If you do not want to exclude such solutions, then use the `check=False` option:

```
>>> from sympy import sin, limit
>>> solve(sin(x)/x) # 0 is excluded
[pi]
```

If `check=False`, then a solution to the numerator being zero is found: $x = 0$. In this case, this is a spurious solution since $\sin(x)/x$ has the well known limit (without discontinuity) of 1 at $x = 0$:

```
>>> solve(sin(x)/x, check=False)
[0, pi]
```

In the following case, however, the limit exists and is equal to the value of $x = 0$ that is excluded when `check=True`:

```
>>> eq = x**2*(1/x - z**2/x)
>>> solve(eq, x)
[]
>>> solve(eq, x, check=False)
[0]
>>> limit(eq, x, 0, '-')
0
>>> limit(eq, x, 0, '+')
0
```

Disabling High-Order Explicit Solutions

When solving polynomial expressions, you might not want explicit solutions (which can be quite long). If the expression is univariate, `CRootOf` instances will be returned instead:

```
>>> solve(x**3 - x + 1)
[-1/((-1/2 - sqrt(3)*I/2)*(3*sqrt(69)/2 + 27/2)**(1/3)) -
```

(continues on next page)

(continued from previous page)

```
(-1/2 - sqrt(3)*I/2)*(3*sqrt(69)/2 + 27/2)**(1/3)/3,
-(-1/2 + sqrt(3)*I/2)*(3*sqrt(69)/2 + 27/2)**(1/3)/3 -
1/((-1/2 + sqrt(3)*I/2)*(3*sqrt(69)/2 + 27/2)**(1/3)),
-(3*sqrt(69)/2 + 27/2)**(1/3)/3 -
1/(3*sqrt(69)/2 + 27/2)**(1/3)]
>>> solve(x**3 - x + 1, cubics=False)
[CRootOf(x**3 - x + 1, 0),
 CRootOf(x**3 - x + 1, 1),
 CRootOf(x**3 - x + 1, 2)]
```

If the expression is multivariate, no solution might be returned:

```
>>> solve(x**3 - x + a, x, cubics=False)
[]
```

Sometimes solutions will be obtained even when a flag is False because the expression could be factored. In the following example, the equation can be factored as the product of a linear and a quadratic factor so explicit solutions (which did not require solving a cubic expression) are obtained:

```
>>> eq = x**3 + 3*x**2 + x - 1
>>> solve(eq, cubics=False)
[-1, -1 + sqrt(2), -sqrt(2) - 1]
```

Solving Equations Involving Radicals

Because of SymPy's use of the principle root, some solutions to radical equations will be missed unless `check=False`:

```
>>> from sympy import root
>>> eq = root(x**3 - 3*x**2, 3) + 1 - x
>>> solve(eq)
[]
>>> solve(eq, check=False)
[1/3]
```

In the above example, there is only a single solution to the equation. Other expressions will yield spurious roots which must be checked manually; roots which give a negative argument to odd-powered radicals will also need special checking:

```
>>> from sympy import real_root, S
>>> eq = root(x, 3) - root(x, 5) + S(1)/7
>>> solve(eq) # this gives 2 solutions but misses a 3rd
[CRootOf(7*x**5 - 7*x**3 + 1, 1)**15,
 CRootOf(7*x**5 - 7*x**3 + 1, 2)**15]
>>> sol = solve(eq, check=False)
>>> [abs(eq.subs(x,i).n(2)) for i in sol]
[0.48, 0.e-110, 0.e-110, 0.052, 0.052]
```

The first solution is negative so `real_root` must be used to see that it satisfies the expression:

```
>>> abs(real_root(eq.subs(x, sol[0])).n(2))
0.e-110
```

If the roots of the equation are not real then more care will be necessary to find the roots, especially for higher order equations. Consider the following expression:

```
>>> expr = root(x, 3) - root(x, 5)
```

We will construct a known value for this expression at $x = 3$ by selecting the 1-th root for each radical:

```
>>> expr1 = root(x, 3, 1) - root(x, 5, 1)
>>> v = expr1.subs(x, -3)
```

The solve function is unable to find any exact roots to this equation:

```
>>> eq = Eq(expr, v); eq1 = Eq(expr1, v)
>>> solve(eq, check=False), solve(eq1, check=False)
([], [])
```

The function `unrad`, however, can be used to get a form of the equation for which numerical roots can be found:

```
>>> from sympy.solvers.solvers import unrad
>>> from sympy import nroots
>>> e, (p, cov) = unrad(eq)
>>> pvals = nroots(e)
>>> inversion = solve(cov, x)[0]
>>> xvals = [inversion.subs(p, i) for i in pvals]
```

Although `eq` or `eq1` could have been used to find `xvals`, the solution can only be verified with `expr1`:

```
>>> z = expr - v
>>> [xi.n(chop=1e-9) for xi in xvals if abs(z.subs(x, xi).n()) < 1e-9]
[]
>>> z1 = expr1 - v
>>> [xi.n(chop=1e-9) for xi in xvals if abs(z1.subs(x, xi).n()) < 1e-9]
[-3.0]
```

See also:

[rsolve](#) (page 853)

For solving recurrence relationships

[dsolve](#) (page 755)

For solving differential equations

`sympy.solvers.solvers.solve_linear(lhs, rhs=0, symbols=[], exclude=[])`

Return a tuple derived from $f = \text{lhs} - \text{rhs}$ that is one of the following: $(0, 1)$, $(0, 0)$, $(\text{symbol}, \text{solution})$, (n, d) .

Explanation

$(0, 1)$ meaning that f is independent of the symbols in *symbols* that are not in *exclude*.

$(0, 0)$ meaning that there is no solution to the equation amongst the symbols given. If the first element of the tuple is not zero, then the function is guaranteed to be dependent on a symbol in *symbols*.

$(\text{symbol}, \text{solution})$ where *symbol* appears linearly in the numerator of f , is in *symbols* (if given), and is not in *exclude* (if given). No simplification is done to f other than a `mul=True` expansion, so the solution will correspond strictly to a unique solution.

(n, d) where n and d are the numerator and denominator of f when the numerator was not linear in any symbol of interest; n will never be a symbol unless a solution for that symbol was found (in which case the second element is the solution, not the denominator).

Examples

```
>>> from sympy import cancel, Pow
```

f is independent of the symbols in *symbols* that are not in *exclude*:

```
>>> from sympy import cos, sin, solve_linear
>>> from sympy.abc import x, y, z
>>> eq = y*cos(x)**2 + y*sin(x)**2 - y # = y*(1 - 1) = 0
>>> solve_linear(eq)
(0, 1)
>>> eq = cos(x)**2 + sin(x)**2 # = 1
>>> solve_linear(eq)
(0, 1)
>>> solve_linear(x, exclude=[x])
(0, 1)
```

The variable x appears as a linear variable in each of the following:

```
>>> solve_linear(x + y**2)
(x, -y**2)
>>> solve_linear(1/x - y**2)
(x, y**(-2))
```

When not linear in x or y then the numerator and denominator are returned:

```
>>> solve_linear(x**2/y**2 - 3)
(x**2 - 3*y**2, y**2)
```

If the numerator of the expression is a symbol, then $(0, 0)$ is returned if the solution for that symbol would have set any denominator to 0:

```
>>> eq = 1/(1/x - 2)
>>> eq.as_numer_denom()
(x, 1 - 2*x)
>>> solve_linear(eq)
(0, 0)
```

But automatic rewriting may cause a symbol in the denominator to appear in the numerator so a solution will be returned:

```
>>> (1/x)**-1
x
>>> solve_linear((1/x)**-1)
(x, 0)
```

Use an unevaluated expression to avoid this:

```
>>> solve_linear(Pow(1/x, -1, evaluate=False))
(0, 0)
```

If x is allowed to cancel in the following expression, then it appears to be linear in x , but this sort of cancellation is not done by `solve_linear` so the solution will always satisfy the original expression without causing a division by zero error.

```
>>> eq = x**2*(1/x - z**2/x)
>>> solve_linear(cancel(eq))
(x, 0)
>>> solve_linear(eq)
(x**2*(1 - z**2), x)
```

A list of symbols for which a solution is desired may be given:

```
>>> solve_linear(x + y + z, symbols=[y])
(y, -x - z)
```

A list of symbols to ignore may also be given:

```
>>> solve_linear(x + y + z, exclude=[x])
(y, -x - z)
```

(A solution for y is obtained because it is the first variable from the canonically sorted list of symbols that had a linear solution.)

`sympy.solvers.solvers.solve_linear_system(system, *symbols, **flags)`

Solve system of N linear equations with M variables, which means both under- and overdetermined systems are supported.

Explanation

The possible number of solutions is zero, one, or infinite. Respectively, this procedure will return `None` or a dictionary with solutions. In the case of underdetermined systems, all arbitrary parameters are skipped. This may cause a situation in which an empty dictionary is returned. In that case, all symbols can be assigned arbitrary values.

Input to this function is a $N \times M + 1$ matrix, which means it has to be in augmented form. If you prefer to enter N equations and M unknowns then use `solve(Neqs, *Msymbols)` instead. Note: a local copy of the matrix is made by this routine so the matrix that is passed will not be modified.

The algorithm used here is fraction-free Gaussian elimination, which results, after elimination, in an upper-triangular matrix. Then solutions are found using back-substitution. This approach is more efficient and compact than the Gauss-Jordan method.

Examples

```
>>> from sympy import Matrix, solve_linear_system
>>> from sympy.abc import x, y
```

Solve the following system:

```
x + 4 y == 2
-2 x + y == 14
```

```
>>> system = Matrix(( (1, 4, 2), (-2, 1, 14)))
>>> solve_linear_system(system, x, y)
{x: -6, y: 2}
```

A degenerate system returns an empty dictionary:

```
>>> system = Matrix(( (0,0,0), (0,0,0) ))
>>> solve_linear_system(system, x, y)
{}
```

`sympy.solvers.solvers.solve_linear_system_LU(matrix, syms)`

Solves the augmented matrix system using `LUsolve` and returns a dictionary in which solutions are keyed to the symbols of *syms* as ordered.

Explanation

The matrix must be invertible.

Examples

```
>>> from sympy import Matrix, solve_linear_system_LU
>>> from sympy.abc import x, y, z
```

```
>>> solve_linear_system_LU(Matrix([
... [1, 2, 0, 1],
... [3, 2, 2, 1],
... [2, 0, 0, 1]]), [x, y, z])
{x: 1/2, y: 1/4, z: -1/2}
```

See also:

[`LUsolve`](#) (page 1287)

`sympy.solvers.solvers.solve_undetermined_coeffs(equ, coeffs, sym, **flags)`

Solve equation of a type $p(x; a_1, \dots, a_k) = q(x)$ where both p and q are univariate polynomials that depend on k parameters.

Explanation

The result of this function is a dictionary with symbolic values of those parameters with respect to coefficients in q .

This function accepts both equations class instances and ordinary SymPy expressions. Specification of parameters and variables is obligatory for efficiency and simplicity reasons.

Examples

```
>>> from sympy import Eq, solve_undetermined_coeffs
>>> from sympy.abc import a, b, c, x
```

```
>>> solve_undetermined_coeffs(Eq(2*a*x + a+b, x), [a, b], x)
{a: 1/2, b: -1/2}
```

```
>>> solve_undetermined_coeffs(Eq(a*c*x + a+b, x), [a, b], x)
{a: 1/c, b: -1/c}
```

`sympy.solvers.solvers.nsolve(*args, dict=False, **kwargs)`

Solve a nonlinear equation system numerically: `nsolve(f, [args,] x0, modules=['mpmath'], **kwargs)`.

Explanation

f is a vector function of symbolic expressions representing the system. *args* are the variables. If there is only one variable, this argument can be omitted. x_0 is a starting vector close to a solution.

Use the `modules` keyword to specify which modules should be used to evaluate the function and the Jacobian matrix. Make sure to use a module that supports matrices. For more information on the syntax, please see the docstring of `lambdify`.

If the keyword arguments contain `dict=True` (default is `False`) `nsolve` will return a list (perhaps empty) of solution mappings. This might be especially useful if you want to use `nsolve` as a fallback to `solve` since using the `dict` argument for both methods produces return values of consistent type structure. Please note: to keep this consistent with `solve`, the solution will be returned in a list even though `nsolve` (currently at least) only finds one solution at a time.

Overdetermined systems are supported.

Examples

```
>>> from sympy import Symbol, nsolve
>>> import mpmath
>>> mpmath.mp.dps = 15
>>> x1 = Symbol('x1')
>>> x2 = Symbol('x2')
>>> f1 = 3 * x1**2 - 2 * x2**2 - 1
>>> f2 = x1**2 - 2 * x1 + x2**2 + 2 * x2 - 8
>>> print(nsolve((f1, f2), (x1, x2), (-1, 1)))
Matrix([[ -1.19287309935246], [1.27844411169911]])
```

For one-dimensional functions the syntax is simplified:

```
>>> from sympy import sin, nsolve
>>> from sympy.abc import x
>>> nsolve(sin(x), x, 2)
3.14159265358979
>>> nsolve(sin(x), 2)
3.14159265358979
```

To solve with higher precision than the default, use the `prec` argument:

```
>>> from sympy import cos
>>> nsolve(cos(x) - x, 1)
0.739085133215161
>>> nsolve(cos(x) - x, 1, prec=50)
0.73908513321516064165531208767387340401341175890076
>>> cos(_)
0.73908513321516064165531208767387340401341175890076
```

To solve for complex roots of real functions, a nonreal initial point must be specified:

```
>>> from sympy import I
>>> nsolve(x**2 + 2, I)
1.4142135623731*I
```

`mpmath.findroot` is used and you can find their more extensive documentation, especially concerning keyword parameters and available solvers. Note, however, that functions which are very steep near the root, the verification of the solution may fail. In this case you should use the flag `verify=False` and independently verify the solution.

```
>>> from sympy import cos, cosh
>>> f = cos(x)*cosh(x) - 1
>>> nsolve(f, 3.14*100)
Traceback (most recent call last):
...
ValueError: Could not find root within given tolerance. (1.39267e+230 >
↳ 2.1684e-19)
>>> ans = nsolve(f, 3.14*100, verify=False); ans
312.588469032184
>>> f.subs(x, ans).n(2)
2.1e+121
```

(continues on next page)

(continued from previous page)

```
>>> (f/f.diff(x)).subs(x, ans).n(2)
7.4e-15
```

One might safely skip the verification if bounds of the root are known and a bisection method is used:

```
>>> bounds = lambda i: (3.14*i, 3.14*(i + 1))
>>> nsolve(f, bounds(100), solver='bisect', verify=False)
315.730061685774
```

Alternatively, a function may be better behaved when the denominator is ignored. Since this is not always the case, however, the decision of what function to use is left to the discretion of the user.

```
>>> eq = x**2/(1 - x)/(1 - 2*x)**2 - 100
>>> nsolve(eq, 0.46)
Traceback (most recent call last):
...
ValueError: Could not find root within given tolerance. (10000 > 2.1684e-
→19)
Try another starting point or tweak arguments.
>>> nsolve(eq.as_numer_denom()[0], 0.46)
0.46792545969349058
```

`sympy.solvers.solvers.checksol(f, symbol, sol=None, **flags)`

Checks whether *sol* is a solution of equation $f == 0$.

Explanation

Input can be either a single symbol and corresponding value or a dictionary of symbols and values. When given as a dictionary and flag `simplify=True`, the values in the dictionary will be simplified. *f* can be a single equation or an iterable of equations. A solution must satisfy all equations in *f* to be considered valid; if a solution does not satisfy any equation, False is returned; if one or more checks are inconclusive (and none are False) then None is returned.

Examples

```
>>> from sympy import checksol, symbols
>>> x, y = symbols('x,y')
>>> checksol(x**4 - 1, x, 1)
True
>>> checksol(x**4 - 1, x, 0)
False
>>> checksol(x**2 + y**2 - 5**2, {x: 3, y: 4})
True
```

To check if an expression is zero using `checksol()`, pass it as *f* and send an empty dictionary for *symbol*:

```
>>> checksol(x**2 + x - x*(x + 1), {})
True
```

None is returned if `checksol()` could not conclude.

flags:

'numerical=True (default)'

do a fast numerical check if `f` has only one symbol.

'minimal=True (default is False)'

a very fast, minimal testing.

'warn=True (default is False)'

show a warning if `checksol()` could not conclude.

'simplify=True (default)'

simplify solution before substituting into function and simplify the function before trying specific simplifications

'force=True (default is False)'

make positive all symbols without assumptions regarding sign.

`sympy.solvers.solvers.unrad(eq, *syms, **flags)`

Remove radicals with symbolic arguments and return `(eq, cov)`, `None`, or raise an error.

Explanation

None is returned if there are no radicals to remove.

`NotImplementedError` is raised if there are radicals and they cannot be removed or if the relationship between the original symbols and the change of variable needed to rewrite the system as a polynomial cannot be solved.

Otherwise the tuple, `(eq, cov)`, is returned where:

eq, cov

`eq` is an equation without radicals (in the symbol(s) of interest) whose solutions are a superset of the solutions to the original expression. `eq` might be rewritten in terms of a new variable; the relationship to the original variables is given by `cov` which is a list containing `v` and `v**p - b` where `p` is the power needed to clear the radical and `b` is the radical now expressed as a polynomial in the symbols of interest. For example, for `sqrt(2 - x)` the tuple would be `(c, c**2 - 2 + x)`. The solutions of `eq` will contain solutions to the original equation (if there are any).

syms

An iterable of symbols which, if provided, will limit the focus of radical removal: only radicals with one or more of the symbols of interest will be cleared. All free symbols are used if `syms` is not set.

`flags` are used internally for communication during recursive calls. Two options are also recognized:

`take`, when defined, is interpreted as a single-argument function that returns `True` if a given `Pow` should be handled.

Radicals can be removed from an expression if:

- All bases of the radicals are the same; a change of variables is done in this case.

- If all radicals appear in one term of the expression.
- There are only four terms with `sqrt()` factors or there are less than four terms having `sqrt()` factors.
- There are only two terms with radicals.

Examples

```
>>> from sympy.solvers.solvers import unrad
>>> from sympy.abc import x
>>> from sympy import sqrt, Rational, root
```

```
>>> unrad(sqrt(x)*x**Rational(1, 3) + 2)
(x**5 - 64, [])
>>> unrad(sqrt(x) + root(x + 1, 3))
(-x**3 + x**2 + 2*x + 1, [])
>>> eq = sqrt(x) + root(x, 3) - 2
>>> unrad(eq)
(_p**3 + _p**2 - 2, [_p, _p**6 - x])
```

Ordinary Differential equations (ODEs)

See [ODE](#) (page 755).

Partial Differential Equations (PDEs)

See [PDE](#) (page 826).

Deutils (Utilities for solving ODE's and PDE's)

`sympy.solvers.deutils.ode_order(expr, func)`

Returns the order of a given differential equation with respect to *func*.

This function is implemented recursively.

Examples

```
>>> from sympy import Function
>>> from sympy.solvers.deutils import ode_order
>>> from sympy.abc import x
>>> f, g = map(Function, ['f', 'g'])
>>> ode_order(f(x).diff(x, 2) + f(x).diff(x)**2 +
... f(x).diff(x), f(x))
2
>>> ode_order(f(x).diff(x, 2) + g(x).diff(x, 3), f(x))
2
```

(continues on next page)

(continued from previous page)

```
>>> ode_order(f(x).diff(x, 2) + g(x).diff(x, 3), g(x))
3
```

Recurrence Equations

`sympy.solvers.recurr.rsolve(f, y, init=None)`

Solve univariate recurrence with rational coefficients.

Given k -th order linear recurrence $Ly = f$, or equivalently:

$$a_k(n)y(n+k) + a_{k-1}(n)y(n+k-1) + \cdots + a_0(n)y(n) = f(n)$$

where $a_i(n)$, for $i = 0, \dots, k$, are polynomials or rational functions in n , and f is a hypergeometric function or a sum of a fixed number of pairwise dissimilar hypergeometric terms in n , finds all solutions or returns None, if none were found.

Initial conditions can be given as a dictionary in two forms:

- (1) `{ n_0 : v_0, n_1 : v_1, ..., n_m : v_m }`
- (2) `{ y(n_0) : v_0, y(n_1) : v_1, ..., y(n_m) : v_m }`

or as a list `L` of values:

`L = [v_0, v_1, ..., v_m]`

where `L[i] = v_i`, for $i = 0, \dots, m$, maps to $y(n_i)$.

Examples

Lets consider the following recurrence:

$$(n-1)y(n+2) - (n^2 + 3n - 2)y(n+1) + 2n(n+1)y(n) = 0$$

```
>>> from sympy import Function, rsolve
>>> from sympy.abc import n
>>> y = Function('y')
```

```
>>> f = (n - 1)*y(n + 2) - (n**2 + 3*n - 2)*y(n + 1) + 2*n*(n + 1)*y(n)
```

```
>>> rsolve(f, y(n))
2**n*C0 + C1*factorial(n)
```

```
>>> rsolve(f, y(n), {y(0):0, y(1):3})
3*2**n - 3*factorial(n)
```

See also:

[`rsolve\_poly`](#) (page 853), [`rsolve\_ratio`](#) (page 854), [`rsolve\_hyper`](#) (page 855)

`sympy.solvers.recurr.rsolve_poly(coeffs, f, n, shift=0, **hints)`

Given linear recurrence operator L of order k with polynomial coefficients and inhomogeneous equation $Ly = f$, where f is a polynomial, we seek for all polynomial solutions over field K of characteristic zero.

The algorithm performs two basic steps:

- (1) Compute degree N of the general polynomial solution.
- (2) Find all polynomials of degree N or less of $Ly = f$.

There are two methods for computing the polynomial solutions. If the degree bound is relatively small, i.e. it's smaller than or equal to the order of the recurrence, then naive method of undetermined coefficients is being used. This gives system of algebraic equations with $N + 1$ unknowns.

In the other case, the algorithm performs transformation of the initial equation to an equivalent one, for which the system of algebraic equations has only r indeterminates. This method is quite sophisticated (in comparison with the naive one) and was invented together by Abramov, Bronstein and Petkovsek.

It is possible to generalize the algorithm implemented here to the case of linear q-difference and differential equations.

Lets say that we would like to compute m -th Bernoulli polynomial up to a constant. For this we can use $b(n+1) - b(n) = mn^{m-1}$ recurrence, which has solution $b(n) = B_m + C$. For example:

```
>>> from sympy import Symbol, rsolve_poly
>>> n = Symbol('n', integer=True)
```

```
>>> rsolve_poly([-1, 1], 4*n**3, n)
C0 + n**4 - 2*n**3 + n**2
```

References

[R797], [R798], [R799]

`sympy.solvers.recurr.rsolve_ratio(coeffs, f, n, **hints)`

Given linear recurrence operator L of order k with polynomial coefficients and inhomogeneous equation $Ly = f$, where f is a polynomial, we seek for all rational solutions over field K of characteristic zero.

This procedure accepts only polynomials, however if you are interested in solving recurrence with rational coefficients then use `rsolve` which will pre-process the given equation and run this procedure with polynomial arguments.

The algorithm performs two basic steps:

- (1) Compute polynomial $v(n)$ which can be used as universal denominator of any rational solution of equation $Ly = f$.
- (2) Construct new linear difference equation by substitution $y(n) = u(n)/v(n)$ and solve it for $u(n)$ finding all its polynomial solutions. Return `None` if none were found.

Algorithm implemented here is a revised version of the original Abramov's algorithm, developed in 1989. The new approach is much simpler to implement and has better overall efficiency. This method can be easily adapted to q-difference equations case.

Besides finding rational solutions alone, this functions is an important part of Hyper algorithm were it is used to find particular solution of inhomogeneous part of a recurrence.

Examples

```
>>> from sympy.abc import x
>>> from sympy.solvers.recurr import rsolve_ratio
>>> rsolve_ratio([-2*x**3 + x**2 + 2*x - 1, 2*x**3 + x**2 - 6*x,
... - 2*x**3 - 11*x**2 - 18*x - 9, 2*x**3 + 13*x**2 + 22*x + 8], 0, x)
C2*(2*x - 3)/(2*(x**2 - 1))
```

See also:

[`rsolve\_hyper`](#) (page 855)

References

[R800]

`sympy.solvers.recurr.rsolve_hyper(coeffs, f, n, **hints)`

Given linear recurrence operator L of order k with polynomial coefficients and inhomogeneous equation $Ly = f$ we seek for all hypergeometric solutions over field K of characteristic zero.

The inhomogeneous part can be either hypergeometric or a sum of a fixed number of pairwise dissimilar hypergeometric terms.

The algorithm performs three basic steps:

- (1) Group together similar hypergeometric terms in the inhomogeneous part of $Ly = f$, and find particular solution using Abramov's algorithm.
- (2) Compute generating set of L and find basis in it, so that all solutions are linearly independent.
- (3) Form final solution with the number of arbitrary constants equal to dimension of basis of L .

Term $a(n)$ is hypergeometric if it is annihilated by first order linear difference equations with polynomial coefficients or, in simpler words, if consecutive term ratio is a rational function.

The output of this procedure is a linear combination of fixed number of hypergeometric terms. However the underlying method can generate larger class of solutions - D'Alembertian terms.

Note also that this method not only computes the kernel of the inhomogeneous equation, but also reduces in to a basis so that solutions generated by this procedure are linearly independent

Examples

```
>>> from sympy.solvers import rsolve_hyper
>>> from sympy.abc import x
```

```
>>> rsolve_hyper([-1, -1, 1], 0, x)
C0*(1/2 - sqrt(5)/2)**x + C1*(1/2 + sqrt(5)/2)**x
```

```
>>> rsolve_hyper([-1, 1], 1 + x, x)
C0 + x*(x + 1)/2
```

References

[R801], [R802]

Systems of Polynomial Equations

`sympy.solvers.polysys.solve_poly_system(seq, *gens, strict=False, **args)`

Solve a system of polynomial equations.

Parameters

seq: a list/tuple/set

Listing all the equations that are needed to be solved

gens: generators

generators of the equations in seq for which we want the solutions

strict: a boolean (default is False)

if strict is True, `NotImplementedError` will be raised if the solution is known to be incomplete (which can occur if not all solutions are expressible in radicals)

args: Keyword arguments

Special options for solving the equations.

Returns

List[Tuple]

A List of tuples. Solutions for symbols that satisfy the equations listed in seq

Examples

```
>>> from sympy import solve_poly_system
>>> from sympy.abc import x, y
```

```
>>> solve_poly_system([x*y - 2*y, 2*y**2 - x**2], x, y)
[(0, 0), (2, -sqrt(2)), (2, sqrt(2))]
```

```
>>> solve_poly_system([x**5 - x + y**3, y**2 - 1], x, y, strict=True)
Traceback (most recent call last):
...
UnsolvableFactorError
```

`sympy.solvers.polysys.solve_triangulated(polys, *gens, **args)`

Solve a polynomial system using Gianni-Kalkbrenner algorithm.

The algorithm proceeds by computing one Groebner basis in the ground domain and then by iteratively computing polynomial factorizations in appropriately constructed algebraic extensions of the ground domain.

Parameters

polys: a list/tuple/set

Listing all the equations that are needed to be solved

gens: generators

generators of the equations in polys for which we want the solutions

args: Keyword arguments

Special options for solving the equations

Returns

List[Tuple]

A List of tuples. Solutions for symbols that satisfy the equations listed in polys

Examples

```
>>> from sympy import solve_triangulated
>>> from sympy.abc import x, y, z
```

```
>>> F = [x**2 + y + z - 1, x + y**2 + z - 1, x + y + z**2 - 1]
```

```
>>> solve_triangulated(F, x, y, z)
[(0, 0, 1), (0, 1, 0), (1, 0, 0)]
```

References

1. Patrizia Gianni, Teo Mora, Algebraic Solution of System of Polynomial Equations using Groebner Bases, AAEECC-5 on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, LNCS 356 247–257, 1989

Diophantine Equations (DEs)

See *Diophantine* (page 714)

Inequalities

See *Inequality Solvers* (page 750)

Solveset

This is the official documentation of the `solveset` module in solvers. It contains the frequently asked questions about our new module to solve equations.

What's wrong with solve():

SymPy already has a pretty powerful `solve` function. But it has some deficiencies. For example:

1. It doesn't have a consistent output for various types of solutions It needs to return a lot of types of solutions consistently:
 - Single solution : $x = 1$
 - Multiple solutions: $x^2 = 1$
 - No Solution: $x^2 + 1 = 0; x \in \mathbb{R}$
 - Interval of solution: $\lfloor x \rfloor = 0$
 - Infinitely many solutions: $\sin(x) = 0$
 - Multivariate functions with point solutions: $x^2 + y^2 = 0$
 - Multivariate functions with non-point solution: $x^2 + y^2 = 1$
 - System of equations: $x + y = 1$ and $x - y = 0$
 - Relational: $x > 0$
 - And the most important case: "We don't Know"
2. The input API has a lot of parameters and it can be difficult to use.
3. There are cases like finding the maxima and minima of function using critical points where it is important to know if it has returned all the solutions. `solve` does not guarantee this.

Why Solveset?

- `solveset` has an alternative consistent input and output interface: `solveset` returns a set object and a set object takes care of all types of output. For cases where it does not “know” all the solutions a `ConditionSet` with a partial solution is returned. For input it only takes the equation, the variables to solve for and the optional argument domain over which the equation is to be solved.
- `solveset` can return infinitely many solutions. For example solving for $\sin(x) = 0$ returns $\{2n\pi | n \in \mathbb{Z}\} \cup \{2n\pi + \pi | n \in \mathbb{Z}\}$, whereas `solve` only returns $[0, \pi]$.
- There is a clear code level and interface level separation between solvers for equations in the complex domain and the real domain. For example solving $e^x = 1$ when x is to be solved in the complex domain, returns the set of all solutions, that is $\{2ni\pi | n \in \mathbb{Z}\}$, whereas if x is to be solved in the real domain then only $\{0\}$ is returned.

Why do we use Sets as an output type?

SymPy has a well developed sets module, which can represent most of the set containers in mathematics such as:

- [FiniteSet](#) (page 1197)
Represents a finite set of discrete numbers.
- [Interval](#) (page 1194)
Represents a real interval as a set.
- [ProductSet](#) (page 1199)
Represents a Cartesian product of sets.
- [ImageSet](#) (page 1206)
Represents the image of a set under a mathematical function

```
>>> from sympy import ImageSet, S, Lambda
>>> from sympy.abc import x
>>> squares = ImageSet(Lambda(x, x**2), S.Naturals) # {x**2 for x in N}
>>> 4 in squares
True
```

- [ComplexRegion](#) (page 1209)
Represents the set of all complex numbers in a region in the Argand plane.
- [ConditionSet](#) (page 1215)
Represents the set of elements, which satisfies a given condition.

Also, the predefined set classes such as:

- [Naturals](#) (page 1203), $\mathbb{N}$
Represents the natural numbers (or counting numbers), which are all positive integers starting from 1.
- [Naturals0](#) (page 1204), $\mathbb{N}_0$
Represents the whole numbers, which are all the non-negative integers, inclusive of 0.

- *Integers* (page 1204), $\mathbb{Z}$
Represents all integers: positive, negative and zero.
- *Reals* (page 1205), $\mathbb{R}$
Represents the set of all real numbers.
- *Complexes* (page 1206), $\mathbb{C}$
Represents the set of all complex numbers.
- *EmptySet* (page 1202), $\emptyset$
Represents the empty set.

The above six sets are available as Singletons, like `S.Integers`.

It is capable of most of the set operations in mathematics:

- Union
- Intersection
- Complement
- SymmetricDifference

The main reason for using sets as output to solvers is that it can consistently represent many types of solutions. For the single variable case it can represent:

- No solution (by the empty set).
- Finitely many solutions (by `FiniteSet`).
- Infinitely many solutions, both countably and uncountably infinite solutions (using the `ImageSet` module).
- Interval
- There can also be bizarre solutions to equations like the set of rational numbers.

No other Python object (list, dictionary, generator, Python sets) provides the flexibility of mathematical sets which our sets module tries to emulate. The second reason to use sets is that they are close to the entities which mathematicians deal with and it makes it easier to reason about them. Set objects conform to Pythonic conventions when possible, i.e., `x in A` and `for i in A` both work when they can be computed. Another advantage of using objects closer to mathematical entities is that the user won't have to "learn" our representation and she can have her expectations transferred from her mathematical experience.

For the multivariate case we represent solutions as a set of points in a n -dimensional space and a point is represented by a `FiniteSet` of ordered tuples, which is a point in $\mathbb{R}^n$ or $\mathbb{C}^n$.

Please note that, the general `FiniteSet` is unordered, but a `FiniteSet` with a tuple as its only argument becomes ordered, since a tuple is ordered. So the order in the tuple is mapped to a pre-defined order of variables while returning solutions.

For example:

```
>>> from sympy import FiniteSet
>>> FiniteSet(1, 2, 3)    # Unordered
{1, 2, 3}
>>> FiniteSet((1, 2, 3)) # Ordered
{(1, 2, 3)}
```

Why not use dicts as output?

Dictionary are easy to deal with programmatically but mathematically they are not very precise and use of them can quickly lead to inconsistency and a lot of confusion. For example:

- There are a lot of cases where we don't know the complete solution and we may like to output a partial solution, consider the equation $fg = 0$. The solution of this equation is the union of the solution of the following two equations: $f = 0$, $g = 0$. Let's say that we are able to solve $f = 0$ but solving $g = 0$ isn't supported yet. In this case we cannot represent partial solution of the given equation $fg = 0$ using dicts. This problem is solved with sets using a `ConditionSet` object:

$sol_f \cup \{x | x \in \mathbb{R} \wedge g = 0\}$, where sol_f is the solution of the equation $f = 0$.

- Using a dict may lead to surprising results like:
 - `solve(Eq(x**2, 1), x) != solve(Eq(y**2, 1), y)`
 Mathematically, this doesn't make sense. Using `FiniteSet` here solves the problem.
- It also cannot represent solutions for equations like $|x| < 1$, which is a disk of radius 1 in the Argand Plane. This problem is solved using complex sets implemented as `ComplexRegion`.

Input API of solveset

`solveset` has simpler input API, unlike `solve`. It takes a maximum of three arguments:

`solveset(equation, variable=None, domain=S.Complexes)`

Equation(s)

The equation(s) to solve.

Variable(s)

The variable(s) for which the equation is to be solved.

Domain

The domain in which the equation is to be solved.

`solveset` removes the `flags` argument of `solve`, which had made the input API more complicated and output API inconsistent.

What is this domain argument about?

`Solveset` is designed to be independent of the assumptions on the variable being solved for and instead, uses the domain argument to decide the solver to dispatch the equation to, namely `solveset_real` or `solveset_complex`. It's unlike the old `solve` which considers the assumption on the variable.

```
>>> from sympy import solveset, S
>>> from sympy.abc import x
>>> solveset(x**2 + 1, x) # domain=S.Complexes is default
{-I, I}
>>> solveset(x**2 + 1, x, domain=S.Reals)
EmptySet
```

What are the general methods employed by solveset to solve an equation?

Solveset uses various methods to solve an equation, here is a brief overview of the methodology:

- The domain argument is first considered to know the domain in which the user is interested to get the solution.
- If the given function is a relational ($\geq$, $\leq$, $>$, $<$), and the domain is real, then `solve_univariate_inequality` and solutions are returned. Solving for complex solutions of inequalities, like $x^2 < 0$ is not yet supported.
- Based on the domain, the equation is dispatched to one of the two functions `solveset_real` or `solveset_complex`, which solves the given equation in the complex or real domain, respectively.
- If the given expression is a product of two or more functions, like say $gh = 0$, then the solution to the given equation is the Union of the solution of the equations $g = 0$ and $h = 0$, if and only if both g and h are finite for a finite input. So, the solution is built up recursively.
- If the function is trigonometric or hyperbolic, the function `_solve_real_trig` is called, which solves it by converting it to complex exponential form.
- The function is now checked if there is any instance of a `Piecewise` expression, if it is, then it's converted to explicit expression and set pairs and then solved recursively.
- The respective solver now tries to invert the equation using the routines `invert_real` and `invert_complex`. These routines are based on the concept of mathematical inverse (though not exactly). It reduces the real/complex valued equation $f(x) = y$ to a set of equations: $\{g(x) = h_1(y), g(x) = h_2(y), \dots, g(x) = h_n(y)\}$ where $g(x)$ is a simpler function than $f(x)$. There is some work needed to be done in this to find invert of more complex expressions.
- After the invert, the equations are checked for radical or Abs (Modulus), then the method `_solve_radical` tries to simplify the radical, by removing it using techniques like squaring, cubing etc, and `_solve_abs` solves nested Modulus by considering the positive and negative variants, iteratively.
- If none of the above method is successful, then methods of polynomial is used as follows:
 - The method to solve the rational function, `_solve_as_rational`, is called. Based on the domain, the respective poly solver `_solve_as_poly_real` or `_solve_as_poly_complex` is called to solve f as a polynomial.
 - The underlying method `_solve_as_poly` solves the equation using polynomial techniques if it's already a polynomial equation or, with a change of variables, can be made so.
- The final solution set returned by `solveset` is the intersection of the set of solutions found above and the input domain.

How do we manipulate and return an infinite solution?

- In the real domain, we use our ImageSet class in the sets module to return infinite solutions. ImageSet is an image of a set under a mathematical function. For example, to represent the solution of the equation $\sin(x) = 0$, we can use the ImageSet as:

```
>>> from sympy import ImageSet, Lambda, pi, S, Dummy, pprint
>>> n = Dummy('n')
>>> pprint(ImageSet(Lambda(n, 2*pi*n), S.Integers), use_unicode=True)
{2·n·π | n ∈ ℤ}
```

Where n is a dummy variable. It is basically the image of the set of integers under the function $2\pi n$.

- In the complex domain, we use complex sets, which are implemented as the ComplexRegion class in the sets module, to represent infinite solution in the Argand plane. For example to represent the solution of the equation $|z| = 1$, which is a unit circle, we can use the ComplexRegion as:

```
>>> from sympy import ComplexRegion, FiniteSet, Interval, pi, pprint
>>> pprint(ComplexRegion(FiniteSet(1)*Interval(0, 2*pi), polar=True),
→ use_unicode=True)
{r·(i·sin(θ) + cos(θ)) | r, θ ∈ {1} × [0, 2·π]}
```

Where the FiniteSet in the ProductSet is the range of the value of r , which is the radius of the circle and the Interval is the range of θ , the angle from the x axis representing a unit circle in the Argand plane.

Note: We also have non-polar form notation for representing solution in rectangular form. For example, to represent first two quadrants in the Argand plane, we can write the ComplexRegion as:

```
>>> from sympy import ComplexRegion, Interval, pi, oo, pprint
>>> pprint(ComplexRegion(Interval(-oo, oo)*Interval(0, oo)), use_
→ unicode=True)
{x + y·i | x, y ∈ (-∞, ∞) × [0, ∞)}
```

where the Intervals are the range of x and y for the set of complex numbers $x + iy$.

How does solveset ensure that it is not returning any wrong solution?

Solvers in a Computer Algebra System are based on heuristic algorithms, so it's usually very hard to ensure 100% percent correctness, in every possible case. However there are still a lot of cases where we can ensure correctness. Solveset tries to verify correctness wherever it can. For example:

Consider the equation $|x| = n$. A naive method to solve this equation would return $\{-n, n\}$ as its solution, which is not correct since $\{-n, n\}$ can be its solution if and only if n is positive. Solveset returns this information as well to ensure correctness.

```
>>> from sympy import symbols, S, pprint, solveset
>>> x, n = symbols('x, n')
>>> pprint(solveset(abs(x) - n, x, domain=S.Reals), use_unicode=True)
{x | x ∈ {-n, n} ∧ (n ∈ [0, ∞))}
```

Though, there still a lot of work needs to be done in this regard.

Search based solver and step-by-step solution

Note: This is under Development.

After the introduction of [ConditionSet](#) (page 1215), the solving of equations can be seen as set transformations. Here is an abstract view of the things we can do to solve equations.

- Apply various set transformations on the given set.
- Define a metric of the usability of solutions, or a notion of some solutions being better than others.
- Different transformations would be the nodes of a tree.
- Suitable searching techniques could be applied to get the best solution.

ConditionSet gives us the ability to represent unevaluated equations and inequalities in forms like $\{x|f(x) = 0; x \in S\}$ and $\{x|f(x) > 0; x \in S\}$ but a more powerful thing about ConditionSet is that it allows us to write the intermediate steps as set to set transformation. Some of the transformations are:

- Composition: $\{x|f(g(x)) = 0; x \in S\} \Rightarrow \{x|g(x) = y; x \in S, y \in \{z|f(z) = 0; z \in S\}\}$
- **Polynomial Solver:** $\{x|P(x) = 0; x \in S\} \Rightarrow \{x_1, x_2, \dots, x_n\} \cap S$,
where x_i are roots of $P(x)$.
- Invert solver: $\{x|f(x) = 0; x \in S\} \Rightarrow \{g(0)| \text{ all } g \text{ such that } f(g(x)) = x\}$
- **logcombine:** $\{x|\log(f(x)) + \log(g(x)); x \in S\}$
 $\Rightarrow \{x|\log(f(x).g(x)); x \in S\}$ if $f(x) > 0$ and $g(x) > 0 \Rightarrow \{x|\log(f(x)) + \log(g(x)); x \in S\}$ otherwise
- **product solve:** $\{x|f(x)g(x) = 0; x \in S\}$
 $\Rightarrow \{x|f(x) = 0; x \in S\} \cup \{x|g(x) = 0; x \in S\}$ given $f(x)$ and $g(x)$ are bounded.
 $\Rightarrow \{x|f(x)g(x) = 0; x \in S\}$, otherwise

Since the output type is same as the input type any composition of these transformations is also a valid transformation. And our aim is to find the right sequence of compositions (given the atoms) which transforms the given condition set to a set which is not a condition set i.e., FiniteSet, Interval, Set of Integers and their Union, Intersection, Complement or ImageSet. We can assign a cost function to each set, such that, the more desirable that form of set is to us, the less the value of the cost function. This way our problem is now reduced to finding the path from the initial ConditionSet to the lowest valued set on a graph where the atomic transformations forms the edges.

How do we deal with cases where only some of the solutions are known?

Creating a universal equation solver, which can solve each and every equation we encounter in mathematics is an ideal case for solvers in a Computer Algebra System. When cases which are not solved or can only be solved incompletely, a `ConditionSet` is used and acts as an unevaluated solveset object.

Note that, mathematically, finding a complete set of solutions for an equation is undecidable. See [Richardson's theorem](#).

`ConditionSet` is basically a Set of elements which satisfy a given condition. For example, to represent the solutions of the equation in the real domain:

$$(x^2 - 4)(\sin(x) + x)$$

We can represent it as:

$$\{-2, 2\} \cup \{x \mid x \in \mathbb{R} \wedge x + \sin(x) = 0\}$$

What is the plan for solve and solveset?

There are still a few things `solveset` can't do, which `solve` can, such as solving nonlinear multivariate & LambertW type equations. Hence, it's not yet a perfect replacement for `solve`. As the algorithms in `solveset` mature, `solveset` may be able to be used within `solve` to replace some of its algorithms.

How are symbolic parameters handled in solveset?

`Solveset` is in its initial phase of development, so the symbolic parameters aren't handled well for all the cases, but some work has been done in this regard to depict our ideology towards symbolic parameters. As an example, consider the solving of $|x| = n$ for real x , where n is a symbolic parameter. `Solveset` returns the value of x considering the domain of the symbolic parameter n as well:

$$([0, \infty) \cap \{n\}) \cup ((-\infty, 0] \cap \{-n\}).$$

This simply means n is the solution only when it belongs to the Interval $[0, \infty)$ and $-n$ is the solution only when $-n$ belongs to the Interval $(-\infty, 0]$.

There are other cases to address too, like solving $2^x + (a - 2)$ for x where a is a symbolic parameter. As of now, It returns the solution as an intersection with $\mathbb{R}$, which is trivial, as it doesn't reveal the domain of a in the solution.

Recently, we have also implemented a function to find the domain of the expression in a `FiniteSet` (Intersection with the interval) in which it is not-empty. It is a useful addition for dealing with symbolic parameters. For example:

```
>>> from sympy import Symbol, FiniteSet, Interval, not_empty_in, \
    sqrt, oo
>>> from sympy.abc import x
>>> not_empty_in(FiniteSet(x/2).intersect(Interval(0, 1)), x)
Interval(0, 2)
>>> not_empty_in(FiniteSet(x, x**2).intersect(Interval(1, 2)), x)
Union(Interval(1, 2), Interval(-sqrt(2), -1))
```

References

Solveset Module Reference

Use `solveset()` (page 866) to solve equations or expressions (assumed to be equal to 0) for a single variable. Solving an equation like $x^2 == 1$ can be done as follows:

```
>>> from sympy import solveset
>>> from sympy import Symbol, Eq
>>> x = Symbol('x')
>>> solveset(Eq(x**2, 1), x)
{-1, 1}
```

Or one may manually rewrite the equation as an expression equal to 0:

```
>>> solveset(x**2 - 1, x)
{-1, 1}
```

The first argument for `solveset()` (page 866) is an expression (equal to zero) or an equation and the second argument is the symbol that we want to solve the equation for.

`sympy.solvers.solveset.solveset(f, symbol=None, domain=Complexes)`

Solves a given inequality or equation with set as output

Parameters

f : Expr or a relational.

The target equation or inequality

symbol : Symbol

The variable for which the equation is solved

domain : Set

The domain over which the equation is solved

Returns

Set

A set of values for *symbol* for which *f* is True or is equal to zero. An [EmptySet](#) (page 1202) is returned if *f* is False or nonzero. A [ConditionSet](#) (page 1215) is returned as unsolved object if algorithms to evaluate complete solution are not yet implemented.

`solveset` claims to be complete in the solution set that it returns.

Raises

NotImplementedError

The algorithms to solve inequalities in complex domain are not yet implemented.

ValueError

The input is not valid.

RuntimeError

It is a bug, please report to the github issue tracker.

Notes

Python interprets 0 and 1 as False and True, respectively, but in this function they refer to solutions of an expression. So 0 and 1 return the domain and EmptySet, respectively, while True and False return the opposite (as they are assumed to be solutions of relational expressions).

Examples

```
>>> from sympy import exp, sin, Symbol, pprint, S, Eq
>>> from sympy.solvers.solveset import solveset, solveset_real
```

- The default domain is complex. Not specifying a domain will lead to the solving of the equation in the complex domain (and this is not affected by the assumptions on the symbol):

```
>>> x = Symbol('x')
>>> pprint(solveset(exp(x) - 1, x), use_unicode=False)
{2*n*I*pi | n in Integers}
```

```
>>> x = Symbol('x', real=True)
>>> pprint(solveset(exp(x) - 1, x), use_unicode=False)
{2*n*I*pi | n in Integers}
```

- If you want to use solveset to solve the equation in the real domain, provide a real domain. (Using solveset_real does this automatically.)

```
>>> R = S.Reals
>>> x = Symbol('x')
>>> solveset(exp(x) - 1, x, R)
{0}
>>> solveset_real(exp(x) - 1, x)
{0}
```

The solution is unaffected by assumptions on the symbol:

```
>>> p = Symbol('p', positive=True)
>>> pprint(solveset(p**2 - 4))
{-2, 2}
```

When a *ConditionSet* (page 1215) is returned, symbols with assumptions that would alter the set are replaced with more generic symbols:

```
>>> i = Symbol('i', imaginary=True)
>>> solveset(Eq(i**2 + i*sin(i), 1), i, domain=S.Reals)
ConditionSet(_R, Eq(_R**2 + _R*sin(_R) - 1, 0), Reals)
```

- Inequalities can be solved over the real domain only. Use of a complex domain leads to a `NotImplementedError`.

```
>>> solveset(exp(x) > 1, x, R)
Interval.open(0, oo)
```

See also:

`solveset_real` (page 868)
solver for real domain

`solveset_complex` (page 868)
solver for complex domain

`sympy.solvers.solveset.solveset_real(f, symbol)`

`sympy.solvers.solveset.solveset_complex(f, symbol)`

`sympy.solvers.solveset.invert_real(f_x, y, x)`

Inverts a real-valued function. Same as `invert_complex()` (page 868), but sets the domain to `S.Reals` before inverting.

`sympy.solvers.solveset.invert_complex(f_x, y, x, domain=Complexes)`

Reduce the complex valued equation $f(x) = y$ to a set of equations

$$\{g(x) = h_1(y), g(x) = h_2(y), \dots, g(x) = h_n(y)\}$$

where $g(x)$ is a simpler function than $f(x)$. The return value is a tuple $(g(x), \text{set}_h)$, where $g(x)$ is a function of x and set_h is the set of function $\{h_1(y), h_2(y), \dots, h_n(y)\}$. Here, y is not necessarily a symbol.

set_h contains the functions, along with the information about the domain in which they are valid, through set operations. For instance, if $y = |x| - n$ is inverted in the real domain, then set_h is not simply $\{-n, n\}$ as the nature of n is unknown; rather, it is:

`$$ left(left[0, inftyright) cap left{nright}right) cup`
`left(left(-infty, 0right] cap left{- nright}right)$$`

By default, the complex domain is used which means that inverting even seemingly simple functions like $\exp(x)$ will give very different results from those obtained in the real domain. (In the case of $\exp(x)$, the inversion via $\log$ is multi-valued in the complex domain, having infinitely many branches.)

If you are working with real values only (or you are not sure which function to use) you should probably set the domain to `S.Reals` (or use `invert_real` which does that automatically).

Examples

```
>>> from sympy.solvers.solveset import invert_complex, invert_real
>>> from sympy.abc import x, y
>>> from sympy import exp
```

When does $\exp(x) = y$?

```
>>> invert_complex(exp(x), y, x)
(x, ImageSet(Lambda(_n, I*(2*_n*pi + arg(y)) + log(Abs(y))), Integers))
>>> invert_real(exp(x), y, x)
(x, Intersection({log(y)}, Reals))
```

When does $\exp(x) = 1$?

```
>>> invert_complex(exp(x), 1, x)
(x, ImageSet(Lambda(_n, 2*_n*I*pi), Integers))
>>> invert_real(exp(x), 1, x)
(x, {0})
```

See also:

[`invert\_real`](#) (page 868), [`invert\_complex`](#) (page 868)

`sympy.solvers.solveset.domain_check(f, symbol, p)`

Returns False if point p is infinite or any subexpression of f is infinite or becomes so after replacing symbol with p . If none of these conditions is met then True will be returned.

Examples

```
>>> from sympy import Mul, oo
>>> from sympy.abc import x
>>> from sympy.solvers.solveset import domain_check
>>> g = 1/(1 + (1/(x + 1))**2)
>>> domain_check(g, x, -1)
False
>>> domain_check(x**2, x, 0)
True
>>> domain_check(1/x, x, oo)
False
```

- The function relies on the assumption that the original form of the equation has not been changed by automatic simplification.

```
>>> domain_check(x/x, x, 0) # x/x is automatically simplified to 1
True
```

- To deal with automatic evaluations use `evaluate=False`:

```
>>> domain_check(Mul(x, 1/x, evaluate=False), x, 0)
False
```

`sympy.solvers.solveset.solvify(f, symbol, domain)`

Solves an equation using solveset and returns the solution in accordance with the *solve* output API.

Returns

We classify the output based on the type of solution returned by *solveset*.

Raises

NotImplementedError

A ConditionSet is the input.

Solution | Output

FiniteSet | list

ImageSet, | list (if f is periodic) Union |

Union | list (with FiniteSet)

EmptySet | empty list

Others | None

Examples

```
>>> from sympy.solvers.solveset import solvify
>>> from sympy.abc import x
>>> from sympy import S, tan, sin, exp
>>> solvify(x**2 - 9, x, S.Reals)
[-3, 3]
>>> solvify(sin(x) - 1, x, S.Reals)
[pi/2]
>>> solvify(tan(x), x, S.Reals)
[0]
>>> solvify(exp(x) - 1, x, S.Complexes)
```

```
>>> solvify(exp(x) - 1, x, S.Reals)
[0]
```

linear_eq_to_matrix

`sympy.solvers.solveset.linear_eq_to_matrix(equations, *symbols)`

Converts a given System of Equations into Matrix form. Here *equations* must be a linear system of equations in *symbols*. Element $M[i, j]$ corresponds to the coefficient of the j th symbol in the i th equation.

The Matrix form corresponds to the augmented matrix form. For example:

$$4x + 2y + 3z = 1$$

$$3x + y + z = -6$$

$$2x + 4y + 9z = 2$$

This system will return A and b as:

```
$$ A = left[begin{array}{ccc}
4 & 2 & 3 \\ 3 & 1 & 1 \\ 2 & 4 & 9 \end{array}right] b = left[begin{array}{c} 1 \\ -6 \\ 2 \end{array}right] $$
```

The only simplification performed is to convert $\text{Eq}(a, b) \Rightarrow a - b$.

Raises

NonlinearError

The equations contain a nonlinear term.

ValueError

The symbols are not given or are not unique.

Examples

```
>>> from sympy import linear_eq_to_matrix, symbols
>>> c, x, y, z = symbols('c, x, y, z')
```

The coefficients (numerical or symbolic) of the symbols will be returned as matrices:

```
>>> eqns = [c*x + z - 1 - c, y + z, x - y]
>>> A, b = linear_eq_to_matrix(eqns, [x, y, z])
>>> A
Matrix([
[c, 0, 1],
[0, 1, 1],
[1, -1, 0]])
>>> b
Matrix([
[c + 1],
[0],
[0]])
```

This routine does not simplify expressions and will raise an error if nonlinearity is encountered:

```
>>> eqns = [
...     (x**2 - 3*x)/(x - 3) - 3,
...     y**2 - 3*y - y*(y - 4) + x - 4]
>>> linear_eq_to_matrix(eqns, [x, y])
Traceback (most recent call last):
...
NonlinearError:
The term (x**2 - 3*x)/(x - 3) is nonlinear in {x, y}
```

Simplifying these equations will discard the removable singularity in the first, reveal the linear structure of the second:

```
>>> [e.simplify() for e in eqns]
[x - 3, x + y - 4]
```

Any such simplification needed to eliminate nonlinear terms must be done before calling this routine.

linsolve

`sympy.solvers.solveset.linsolve(system, *symbols)`

Solve system of N linear equations with M variables; both underdetermined and overdetermined systems are supported. The possible number of solutions is zero, one or infinite. Zero solutions throws a `ValueError`, whereas infinite solutions are represented parametrically in terms of the given symbols. For unique solution a *FiniteSet* (page 1197) of ordered tuples is returned.

All standard input formats are supported: For the given set of equations, the respective input types are given below:

$$3x + 2y - z = 1$$

$$2x - 2y + 4z = -2$$

$$2x - y + 2z = 0$$

- Augmented matrix form, system given below:

\$\$ text{system} = left[{array}{cccc}

`3 & 2 & -1 & 1 \ 2 & -2 & 4 & -2 \ 2 & -1 & 2 & 0 end{array}right] $$`

```
system = Matrix([[3, 2, -1, 1], [2, -2, 4, -2], [2, -1, 2, 0]])
```

- List of equations form

```
system = [3x + 2y - z - 1, 2x - 2y + 4z + 2, 2x - y + 2z]
```

- Input A and b in matrix form (from $Ax = b$) are given as:

\$\$ A = left[begin{array}{ccc}

`3 & 2 & -1 \ 2 & -2 & 4 \ 2 & -1 & 2 end{array}right] b = left[begin{array}{c} 1 \ -2 \ 0 end{array}right] $$`

```
A = Matrix([[3, 2, -1], [2, -2, 4], [2, -1, 2]])
b = Matrix([[1], [-2], [0]])
system = (A, b)
```

Symbols can always be passed but are actually only needed when 1) a system of equations is being passed and 2) the system is passed as an underdetermined matrix and one wants to control the name of the free variables in the result. An error is raised if no symbols are used for case 1, but if no symbols are provided for case 2, internally generated symbols will be provided. When providing symbols for case 2, there should be at least as many symbols as there are columns in matrix A .

The algorithm used here is Gauss-Jordan elimination, which results, after elimination, in a row echelon form matrix.

Returns

A FiniteSet containing an ordered tuple of values for the unknowns for which the *system* has a solution. (Wrapping the tuple in FiniteSet is used to maintain a consistent output format throughout solveset.)

Returns EmptySet, if the linear system is inconsistent.

Raises**ValueError**

The input is not valid. The symbols are not given.

Examples

```
>>> from sympy import Matrix, linsolve, symbols
>>> x, y, z = symbols("x, y, z")
>>> A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 10]])
>>> b = Matrix([3, 6, 9])
>>> A
Matrix([
[1, 2, 3],
[4, 5, 6],
[7, 8, 10]])
>>> b
Matrix([
[3],
[6],
[9]])
>>> linsolve((A, b), [x, y, z])
{(-1, 2, 0)}
```

- Parametric Solution: In case the system is underdetermined, the function will return a parametric solution in terms of the given symbols. Those that are free will be returned unchanged. e.g. in the system below, z is returned as the solution for variable z ; it can take on any value.

```
>>> A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> b = Matrix([3, 6, 9])
>>> linsolve((A, b), x, y, z)
{(z - 1, 2 - 2*z, z)}
```

If no symbols are given, internally generated symbols will be used. The `tau0` in the third position indicates (as before) that the third variable - whatever it is named - can take on any value:

```
>>> linsolve((A, b))
{(tau0 - 1, 2 - 2*tau0, tau0)}
```

- List of equations as input

```
>>> Eqns = [3*x + 2*y - z - 1, 2*x - 2*y + 4*z + 2, - x + y/2 - z]
>>> linsolve(Eqns, x, y, z)
{(1, -2, -2)}
```

- Augmented matrix as input

```
>>> aug = Matrix([[2, 1, 3, 1], [2, 6, 8, 3], [6, 8, 18, 5]])
>>> aug
Matrix([
[2, 1, 3, 1],
[2, 6, 8, 3],
[6, 8, 18, 5]])
>>> linsolve(aug, x, y, z)
{(3/10, 2/5, 0)}
```

- Solve for symbolic coefficients

```
>>> a, b, c, d, e, f = symbols('a, b, c, d, e, f')
>>> eqns = [a*x + b*y - c, d*x + e*y - f]
>>> linsolve(eqns, x, y)
{((-b*f + c*e)/(a*e - b*d), (a*f - c*d)/(a*e - b*d))}
```

- A degenerate system returns solution as set of given symbols.

```
>>> system = Matrix([[0, 0, 0], [0, 0, 0], [0, 0, 0]])
>>> linsolve(system, x, y)
{(x, y)}
```

- For an empty system linsolve returns empty set

```
>>> linsolve([], x)
EmptySet
```

- An error is raised if, after expansion, any nonlinearity is detected:

```
>>> linsolve([x*(1/x - 1), (y - 1)**2 - y**2 + 1], x, y)
{(1, 1)}
>>> linsolve([x**2 - 1], x)
Traceback (most recent call last):
...
NonlinearError:
nonlinear term encountered: x**2
```

nonlinsolve

`sympy.solvers.solveset.nonlinsolve(system, *symbols)`

Solve system of N nonlinear equations with M variables, which means both under and overdetermined systems are supported. Positive dimensional system is also supported (A system with infinitely many solutions is said to be positive-dimensional). In a positive dimensional system the solution will be dependent on at least one symbol. Returns both real solution and complex solution (if they exist).

Parameters

system : list of equations

The target system of equations

symbols : list of Symbols

symbols should be given as a sequence eg. list

Returns

A [FiniteSet](#) (page 1197) of ordered tuple of values of *symbols* for which the *system*

has solution. Order of values in the tuple is same as symbols present in the parameter *symbols*.

Please note that general [FiniteSet](#) (page 1197) is unordered, the solution returned here is not simply a [FiniteSet](#) (page 1197) of solutions, rather it is a [FiniteSet](#) (page 1197) of ordered tuple, i.e. the first and only argument to [FiniteSet](#) (page 1197) is a tuple of solutions, which is ordered, and, hence, the returned solution is ordered.

Also note that solution could also have been returned as an ordered tuple, FiniteSet is just a wrapper `{}` around the tuple. It has no other significance except for the fact it is just used to maintain a consistent output format throughout the solveset.

For the given set of equations, the respective input types are given below:

$$xy - 1 = 0$$

$$4x^2 + y^2 - 5 = 0$$

```
system = [x*y - 1, 4*x**2 + y**2 - 5]
symbols = [x, y]
```

Raises

ValueError

The input is not valid. The symbols are not given.

AttributeError

The input symbols are not *Symbol* type.

Examples

```
>>> from sympy import symbols, nonlinsolve
>>> x, y, z = symbols('x, y, z', real=True)
>>> nonlinsolve([x*y - 1, 4*x**2 + y**2 - 5], [x, y])
{(-1, -1), (-1/2, -2), (1/2, 2), (1, 1)}
```

1. Positive dimensional system and complements:

```
>>> from sympy import pprint
>>> from sympy.polys.polytools import is_zero_dimensional
>>> a, b, c, d = symbols('a, b, c, d', extended_real=True)
>>> eq1 = a + b + c + d
>>> eq2 = a*b + b*c + c*d + d*a
>>> eq3 = a*b*c + b*c*d + c*d*a + d*a*b
>>> eq4 = a*b*c*d - 1
>>> system = [eq1, eq2, eq3, eq4]
>>> is_zero_dimensional(system)
False
>>> pprint(nonlinsolve(system, [a, b, c, d]), use_unicode=False)
-1      1      1      -1
{(-, -, -d, -, {d} \ {0}), (-, -d, -, {d} \ {0})}
  d      d      d      d
>>> nonlinsolve([(x+y)**2 - 4, x + y - 2], [x, y])
{(2 - y, y)}
```

2. If some of the equations are non-polynomial then *nonlinsolve* will call the substitution function and return real and complex solutions, if present.

```
>>> from sympy import exp, sin
>>> nonlinsolve([exp(x) - sin(y), y**2 - 4], [x, y])
{(ImageSet(Lambda(_n, I*(2*_n*pi + pi) + log(sin(2))), Integers), -2),
 (ImageSet(Lambda(_n, 2*_n*I*pi + log(sin(2))), Integers), 2)}
```

3. If system is non-linear polynomial and zero-dimensional then it returns both solution (real and complex solutions, if present) using *solve_poly_system()* (page 856):

```
>>> from sympy import sqrt
>>> nonlinsolve([x**2 - 2*y**2 - 2, x*y - 2], [x, y])
{(-2, -1), (2, 1), (-sqrt(2)*I, sqrt(2)*I), (sqrt(2)*I, -sqrt(2)*I)}
```

4. *nonlinsolve* can solve some linear (zero or positive dimensional) system (because it uses the *sympy.polys.polytools.groebner()* (page 2377) function to get the groebner basis and then uses the substitution function basis as the new *system*). But it is not recommended to solve linear system using *nonlinsolve*, because *linsolve()* (page 872) is better for general linear systems.

```
>>> nonlinsolve([x + 2*y - z - 3, x - y - 4*z + 9, y + z - 4], [x, y, z])
{(3*z - 5, 4 - z, z)}
```

5. System having polynomial equations and only real solution is solved using *solve_poly_system()* (page 856):

```

>>> e1 = sqrt(x**2 + y**2) - 10
>>> e2 = sqrt(y**2 + (-x + 10)**2) - 3
>>> nonlinsolve((e1, e2), (x, y))
{(191/20, -3*sqrt(391)/20), (191/20, 3*sqrt(391)/20)}
>>> nonlinsolve([x**2 + 2/y - 2, x + y - 3], [x, y])
{(1, 2), (1 - sqrt(5), 2 + sqrt(5)), (1 + sqrt(5), 2 - sqrt(5))}
>>> nonlinsolve([x**2 + 2/y - 2, x + y - 3], [y, x])
{(2, 1), (2 - sqrt(5), 1 + sqrt(5)), (2 + sqrt(5), 1 - sqrt(5))}

```

6. It is better to use symbols instead of trigonometric functions or [Function](#) (page 1050). For example, replace $\sin(x)$ with a symbol, replace $f(x)$ with a symbol and so on. Get a solution from `nonlinsolve` and then use `solveset()` (page 866) to get the value of x .

How Nonlinsolve Is Better Than Old Solver `_solve_system` :

1. A positive dimensional system solver: `nonlinsolve` can return solution for positive dimensional system. It finds the Groebner Basis of the positive dimensional system (calling it as basis) then we can start solving equation (having least number of variable first in the basis) using `solveset` and substituting that solved solutions into other equation (of basis) to get solution in terms of minimum variables. Here the important thing is how we are substituting the known values and in which equations.
2. Real and complex solutions: `nonlinsolve` returns both real and complex solution. If all the equations in the system are polynomial then using `solve_poly_system()` (page 856) both real and complex solution is returned. If all the equations in the system are not polynomial equation then goes to substitution method with this polynomial and non polynomial equation(s), to solve for unsolved variables. Here to solve for particular variable `solveset_real` and `solveset_complex` is used. For both real and complex solution `_solve_using_known_values` is used inside substitution (substitution will be called when any non-polynomial equation is present). If a solution is valid its general solution is added to the final result.
3. [Complement](#) (page 1200) and [Intersection](#) (page 1198) will be added: `nonlinsolve` maintains dict for complements and intersections. If `solveset` find complements or/and intersections with any interval or set during the execution of substitution function, then complement or/and intersection for that variable is added before returning final solution.

transolve

`sympy.solvers.solveset._transolve(f, symbol, domain)`

Function to solve transcendental equations. It is a helper to `solveset` and should be used internally. `_transolve` currently supports the following class of equations:

- Exponential equations
- Logarithmic equations

Parameters

f : Any transcendental equation that needs to be solved.

This needs to be an expression, which is assumed to be equal to 0.

symbol : The variable for which the equation is solved.

This needs to be of class Symbol.

domain : A set over which the equation is solved.

This needs to be of class Set.

Returns

Set

A set of values for symbol for which f is equal to zero. An EmptySet is returned if f does not have solutions in respective domain. A ConditionSet is returned as unsolved object if algorithms to evaluate complete solution are not yet implemented.

How To Use _transolve

_transolve should not be used as an independent function, because it assumes that the equation (f) and the symbol comes from solveset and might have undergone a few modification(s). To use _transolve as an independent function the equation (f) and the symbol should be passed as they would have been by solveset.

Examples

```
>>> from sympy.solvers.solveset import _transolve as transolve
>>> from sympy.solvers.solvers import _tsolve as tsolve
>>> from sympy import symbols, S, pprint
>>> x = symbols('x', real=True) # assumption added
>>> transolve(5**(x - 3) - 3**(2*x + 1), x, S.Reals)
{-(log(3) + 3*log(5))/(-log(5) + 2*log(3))}
```

How _transolve Works

_transolve uses two types of helper functions to solve equations of a particular class:

Identifying helpers: To determine whether a given equation belongs to a certain class of equation or not. Returns either True or False.

Solving helpers: Once an equation is identified, a corresponding helper either solves the equation or returns a form of the equation that solveset might better be able to handle.

- Philosophy behind the module

The purpose of _transolve is to take equations which are not already polynomial in their generator(s) and to either recast them as such through a valid transformation or to solve them outright. A pair of helper functions for each class of supported transcendental functions are employed for this purpose. One identifies the transcendental form of an equation and the other either solves it or recasts it into a tractable form that can be solved by solveset. For example, an equation in the form $ab^{f(x)} - cd^{g(x)} = 0$ can be transformed to $\log(a) + f(x)\log(b) - \log(c) - g(x)\log(d) = 0$ (under certain assumptions) and this can be solved with solveset if $f(x)$ and $g(x)$ are in polynomial form.

How `_transolve` Is Better Than `_tsolve`

1) Better output

`_transolve` provides expressions in a more simplified form.

Consider a simple exponential equation

```
>>> f = 3**(2*x) - 2**(x + 3)
>>> pprint(transolve(f, x, S.Reals), use_unicode=False)
-3*log(2)
{-----}
-2*log(3) + log(2)
>>> pprint(tsolve(f, x), use_unicode=False)
/ 3 \
| -----|
| log(2/9)|
[-log\2 /]
```

2) Extensible

The API of `_transolve` is designed such that it is easily extensible, i.e. the code that solves a given class of equations is encapsulated in a helper and not mixed in with the code of `_transolve` itself.

3) Modular

`_transolve` is designed to be modular i.e, for every class of equation a separate helper for identification and solving is implemented. This makes it easy to change or modify any of the method implemented directly in the helpers without interfering with the actual structure of the API.

4) Faster Computation

Solving equation via `_transolve` is much faster as compared to `_tsolve`. In `solve`, attempts are made computing every possibility to get the solutions. This series of attempts makes solving a bit slow. In `_transolve`, computation begins only after a particular type of equation is identified.

How To Add New Class Of Equations

Adding a new class of equation solver is a three-step procedure:

- Identify the type of the equations

Determine the type of the class of equations to which they belong: it could be of Add, Pow, etc. types. Separate internal functions are used for each type. Write identification and solving helpers and use them from within the routine for the given type of equation (after adding it, if necessary). Something like:

```
def add_type(lhs, rhs, x):
    ....
    if _is_exponential(lhs, x):
        new_eq = _solve_exponential(lhs, rhs, x)
    ....
```

(continues on next page)

(continued from previous page)

```
rhs, lhs = eq.as_independent(x)
if lhs.is_Add:
    result = add_type(lhs, rhs, x)
```

- Define the identification helper.
- Define the solving helper.

Apart from this, a few other things need to be taken care of while adding an equation solver:

- Naming conventions: Name of the identification helper should be as `_is_class` where class will be the name or abbreviation of the class of equation. The solving helper will be named as `_solve_class`. For example: for exponential equations it becomes `_is_exponential` and `_solve_expo`.
- The identifying helpers should take two input parameters, the equation to be checked and the variable for which a solution is being sought, while solving helpers would require an additional domain parameter.
- Be sure to consider corner cases.
- Add tests for each helper.
- Add a docstring to your helper that describes the method implemented. The documentation of the helpers should identify:
 - the purpose of the helper,
 - the method used to identify and solve the equation,
 - a proof of correctness
 - the return values of the helpers

`sympy.solvers.solveset._is_exponential(f, symbol)`

Return True if one or more terms contain `symbol` only in exponents, else False.

Parameters

f : Expr

The equation to be checked

symbol : Symbol

The variable in which the equation is checked

Examples

```
>>> from sympy import symbols, cos, exp
>>> from sympy.solvers.solveset import _is_exponential as check
>>> x, y = symbols('x y')
>>> check(y, y)
False
>>> check(x**y - 1, y)
True
>>> check(x**y*2**y - 1, y)
True
```

(continues on next page)

(continued from previous page)

```
>>> check(exp(x + 3) + 3**x, x)
True
>>> check(cos(2**x), x)
False
```

- Philosophy behind the helper

The function extracts each term of the equation and checks if it is of exponential form w.r.t symbol.

`sympy.solvers.solveset._solve_exponential(lhs, rhs, symbol, domain)`

Helper function for solving (supported) exponential equations.

Exponential equations are the sum of (currently) at most two terms with one or both of them having a power with a symbol-dependent exponent.

For example

$$5^{2x+3} - 5^{3x-1}$$

$$4^{5-9x} - e^{2-x}$$

Parameters

lhs, rhs : Expr

The exponential equation to be solved, $lhs = rhs$

symbol : Symbol

The variable in which the equation is solved

domain : Set

A set over which the equation is solved.

Returns

A set of solutions satisfying the given equation.

A ConditionSet if the equation is unsolvable or

if the assumptions are not properly defined, in that case

a different style of ConditionSet is returned having the

solution(s) of the equation with the desired assumptions.

Examples

```
>>> from sympy.solvers.solveset import _solve_exponential as solve_expo
>>> from sympy import symbols, S
>>> x = symbols('x', real=True)
>>> a, b = symbols('a b')
>>> solve_expo(2**x + 3**x - 5**x, 0, x, S.Reals) # not solvable
ConditionSet(x, Eq(2**x + 3**x - 5**x, 0), Reals)
>>> solve_expo(a**x - b**x, 0, x, S.Reals) # solvable but incorrect
↪assumptions
ConditionSet(x, (a > 0) & (b > 0), {0})
```

(continues on next page)

(continued from previous page)

```
>>> solve_expo(3**(2*x) - 2**(x + 3), 0, x, S.Reals)
{-3*log(2)/(-2*log(3) + log(2))}
>>> solve_expo(2**x - 4**x, 0, x, S.Reals)
{0}
```

- Proof of correctness of the method

The logarithm function is the inverse of the exponential function. The defining relation between exponentiation and logarithm is:

$$\log_b x = y \text{ if } b^y = x$$

Therefore if we are given an equation with exponent terms, we can convert every term to its corresponding logarithmic form. This is achieved by taking logarithms and expanding the equation using logarithmic identities so that it can easily be handled by `solveset`.

For example:

$$3^{2x} = 2^{x+3}$$

Taking log both sides will reduce the equation to

$$(2x)\log(3) = (x + 3)\log(2)$$

This form can be easily handed by `solveset`.

`sympy.solvers.solveset._solve_logarithm(lhs, rhs, symbol, domain)`

Helper to solve logarithmic equations which are reducible to a single instance of log.

Logarithmic equations are (currently) the equations that contains log terms which can be reduced to a single log term or a constant using various logarithmic identities.

For example:

$$\log(x) + \log(x - 4)$$

can be reduced to:

$$\log(x(x - 4))$$

Parameters

lhs, rhs : Expr

The logarithmic equation to be solved, $lhs = rhs$

symbol : Symbol

The variable in which the equation is solved

domain : Set

A set over which the equation is solved.

Returns

A set of solutions satisfying the given equation.

A ConditionSet if the equation is unsolvable.

Examples

```
>>> from sympy import symbols, log, S
>>> from sympy.solvers.solveset import _solve_logarithm as solve_log
>>> x = symbols('x')
>>> f = log(x - 3) + log(x + 3)
>>> solve_log(f, 0, x, S.Reals)
{-sqrt(10), sqrt(10)}
```

- Proof of correctness

A logarithm is another way to write exponent and is defined by

$$\log_b x = y \text{ if } b^y = x$$

When one side of the equation contains a single logarithm, the equation can be solved by rewriting the equation as an equivalent exponential equation as defined above. But if one side contains more than one logarithm, we need to use the properties of logarithm to condense it into a single logarithm.

Take for example

$$\log(2x) - 15 = 0$$

contains single logarithm, therefore we can directly rewrite it to exponential form as

$$x = \frac{e^{15}}{2}$$

But if the equation has more than one logarithm as

$$\log(x - 3) + \log(x + 3) = 0$$

we use logarithmic identities to convert it into a reduced form

Using,

$$\log(a) + \log(b) = \log(ab)$$

the equation becomes,

$$\log((x - 3)(x + 3))$$

This equation contains one logarithm and can be solved by rewriting to exponents.

`sympy.solvers.solveset._is_logarithmic(f, symbol)`

Return True if the equation is in the form $a \log(f(x)) + b \log(g(x)) + \dots + c$ else False.

Parameters

f : Expr

The equation to be checked

symbol : Symbol

The variable in which the equation is checked

Returns

True if the equation is logarithmic otherwise False.

Examples

```
>>> from sympy import symbols, tan, log
>>> from sympy.solvers.solveset import _is_logarithmic as check
>>> x, y = symbols('x y')
>>> check(log(x + 2) - log(x + 3), x)
True
>>> check(tan(log(2*x)), x)
False
>>> check(x*log(x), x)
False
>>> check(x + log(x), x)
False
>>> check(y + log(x), x)
True
```

- Philosophy behind the helper

The function extracts each term and checks whether it is logarithmic w.r.t symbol.

Diophantine Equations (DEs)

See [Diophantine](#) (page 714)

Inequalities

See [Inequality Solvers](#) (page 750)

Ordinary Differential equations (ODEs)

See [ODE](#) (page 755).

Partial Differential Equations (PDEs)

See [PDE](#) (page 826).

abc

This module exports all latin and greek letters as Symbols, so you can conveniently do

```
>>> from sympy.abc import x, y
```

instead of the slightly more clunky-looking

```
>>> from sympy import symbols
>>> x, y = symbols('x y')
```

Caveats

1. As of the time of writing this, the names O, S, I, N, E, and Q are colliding with names defined in SymPy. If you import them from both `sympy.abc` and `sympy`, the second import will “win”. This is an issue only for `*` imports, which should only be used for short-lived code such as interactive sessions and throwaway scripts that do not survive until the next SymPy upgrade, where `sympy` may contain a different set of names.

2. This module does not define symbol names on demand, i.e. `from sympy.abc import foo` will be reported as an error because `sympy.abc` does not contain the name `foo`. To get a symbol named `foo`, you still need to use `Symbol('foo')` or `symbols('foo')`. You can freely mix usage of `sympy.abc` and `Symbol/symbols`, though sticking with one and only one way to get the symbols does tend to make the code more readable.

The module also defines some special names to help detect which names clash with the default SymPy namespace.

`_clash1` defines all the single letter variables that clash with SymPy objects; `_clash2` defines the multi-letter clashing symbols; and `_clash` is the union of both. These can be passed for `locals` during sympification if one desires Symbols rather than the non-Symbol objects for those names.

Examples

```
>>> from sympy import S
>>> from sympy.abc import _clash1, _clash2, _clash
>>> S("Q & C", locals=_clash1)
C & Q
>>> S('pi(x)', locals=_clash2)
pi(x)
>>> S('pi(C, Q)', locals=_clash)
pi(C, Q)
```

Algebras

Introduction

The Algebras module for SymPy provides support for basic algebraic operations on Quaternions.

Quaternion Reference

This section lists the classes implemented by the Algebras module.

class `sympy.algebras.Quaternion(a=0, b=0, c=0, d=0, real_field=True)`

Provides basic quaternion operations. Quaternion objects can be instantiated as `Quaternion(a, b, c, d)` as in $(a + b*i + c*j + d*k)$.

Examples

```
>>> from sympy import Quaternion
>>> q = Quaternion(1, 2, 3, 4)
>>> q
1 + 2*i + 3*j + 4*k
```

Quaternions over complex fields can be defined as :

```
>>> from sympy import Quaternion
>>> from sympy import symbols, I
>>> x = symbols('x')
>>> q1 = Quaternion(x, x**3, x, x**2, real_field = False)
>>> q2 = Quaternion(3 + 4*I, 2 + 5*I, 0, 7 + 8*I, real_field = False)
>>> q1
x + x**3*i + x*j + x**2*k
>>> q2
(3 + 4*I) + (2 + 5*I)*i + 0*j + (7 + 8*I)*k
```

References

[R1], [R2]

add(*other*)

Adds quaternions.

Parameters

other : Quaternion

The quaternion to add to current (self) quaternion.

Returns

Quaternion

The resultant quaternion after adding self to other

Examples

```
>>> from sympy import Quaternion
>>> from sympy import symbols
>>> q1 = Quaternion(1, 2, 3, 4)
>>> q2 = Quaternion(5, 6, 7, 8)
>>> q1.add(q2)
6 + 8*i + 10*j + 12*k
>>> q1 + 5
6 + 2*i + 3*j + 4*k
>>> x = symbols('x', real = True)
>>> q1.add(x)
(x + 1) + 2*i + 3*j + 4*k
```

Quaternions over complex fields :

```
>>> from sympy import Quaternion
>>> from sympy import I
>>> q3 = Quaternion(3 + 4*I, 2 + 5*I, 0, 7 + 8*I, real_field = False)
>>> q3.add(2 + 3*I)
(5 + 7*I) + (2 + 5*I)*i + 0*j + (7 + 8*I)*k
```

angle()

Returns the angle of the quaternion measured in the real-axis plane.

Explanation

Given a quaternion $q = a + bi + cj + dk$ where a, b, c and d are real numbers, returns the angle of the quaternion given by

$$\text{angle} := \text{atan2}(\sqrt{b^2 + c^2 + d^2}, a)$$

Examples

```
>>> from sympy.algebras.quaternion import Quaternion
>>> q = Quaternion(1, 4, 4, 4)
>>> q.angle()
atan(4*sqrt(3))
```

arc_coplanar(*other*)

Returns True if the transformation arcs represented by the input quaternions happen in the same plane.

Parameters

other : a Quaternion

Returns

True : if the planes of the two quaternions are the same, apart from its orientation/sign.

False : if the planes of the two quaternions are not the same, apart from its orientation/sign.

None : if plane of either of the quaternion is unknown.

Explanation

Two quaternions are said to be coplanar (in this arc sense) when their axes are parallel. The plane of a quaternion is the one normal to its axis.

Examples

```
>>> from sympy.algebras.quaternion import Quaternion
>>> q1 = Quaternion(1, 4, 4, 4)
>>> q2 = Quaternion(3, 8, 8, 8)
>>> Quaternion.arc_coplanar(q1, q2)
True
```

```
>>> q1 = Quaternion(2, 8, 13, 12)
>>> Quaternion.arc_coplanar(q1, q2)
False
```

See also:

[*vector_coplanar*](#) (page 897), [*is_pure*](#) (page 891)

axis()

Returns the axis($\mathbf{Ax}(q)$) of the quaternion.

Explanation

Given a quaternion $q = a + bi + cj + dk$, returns $\mathbf{Ax}(q)$ i.e., the versor of the vector part of that quaternion equal to $\mathbf{U}[\mathbf{V}(q)]$. The axis is always an imaginary unit with square equal to $-1 + 0i + 0j + 0k$.

Examples

```
>>> from sympy.algebras.quaternion import Quaternion
>>> q = Quaternion(1, 1, 1, 1)
>>> q.axis()
0 + sqrt(3)/3*i + sqrt(3)/3*j + sqrt(3)/3*k
```

See also:

[*vector_part*](#) (page 898)

exp()

Returns the exponential of q (e^q).

Returns

Quaternion

Exponential of q (e^q).

Examples

```
>>> from sympy import Quaternion
>>> q = Quaternion(1, 2, 3, 4)
>>> q.exp()
E*cos(sqrt(29))
+ 2*sqrt(29)*E*sin(sqrt(29))/29*i
+ 3*sqrt(29)*E*sin(sqrt(29))/29*j
+ 4*sqrt(29)*E*sin(sqrt(29))/29*k
```

classmethod `from_axis_angle(vector, angle)`

Returns a rotation quaternion given the axis and the angle of rotation.

Parameters

vector : tuple of three numbers

The vector representation of the given axis.

angle : number

The angle by which axis is rotated (in radians).

Returns

Quaternion

The normalized rotation quaternion calculated from the given axis and the angle of rotation.

Examples

```
>>> from sympy import Quaternion
>>> from sympy import pi, sqrt
>>> q = Quaternion.from_axis_angle((sqrt(3)/3, sqrt(3)/3, sqrt(3)/3),
↳ 2*pi/3)
>>> q
1/2 + 1/2*i + 1/2*j + 1/2*k
```

classmethod `from_rotation_matrix(M)`

Returns the equivalent quaternion of a matrix. The quaternion will be normalized only if the matrix is special orthogonal (orthogonal and $\det(M) = 1$).

Parameters

M : Matrix

Input matrix to be converted to equivalent quaternion. M must be special orthogonal (orthogonal and $\det(M) = 1$) for the quaternion to be normalized.

Returns

Quaternion

The quaternion equivalent to given matrix.

Examples

```
>>> from sympy import Quaternion
>>> from sympy import Matrix, symbols, cos, sin, trigsimp
>>> x = symbols('x')
>>> M = Matrix([[cos(x), -sin(x), 0], [sin(x), cos(x), 0], [0, 0, 1]])
>>> q = trigsimp(Quaternion.from_rotation_matrix(M))
>>> q
sqrt(2)*sqrt(cos(x) + 1)/2 + 0*i + 0*j + sqrt(2) -
↳ 2*cos(x))*sign(sin(x))/2*k
```

`index_vector()`

Returns the index vector of the quaternion.

Returns

Quaternion: representing index vector of the provided quaternion.

Explanation

Index vector is given by $\mathbf{T}(q)$ multiplied by $\mathbf{Ax}(q)$ where $\mathbf{Ax}(q)$ is the axis of the quaternion q , and $\text{mod}(q)$ is the $\mathbf{T}(q)$ (magnitude) of the quaternion.

Examples

```
>>> from sympy.algebras.quaternion import Quaternion
>>> q = Quaternion(2, 4, 2, 4)
>>> q.index_vector()
0 + 4*sqrt(10)/3*i + 2*sqrt(10)/3*j + 4*sqrt(10)/3*k
```

See also:

[`axis`](#) (page 888), [`norm`](#) (page 892)

`integrate(*args)`

Computes integration of quaternion.

Returns

Quaternion

Integration of the quaternion(self) with the given variable.

Examples

Indefinite Integral of quaternion :

```
>>> from sympy import Quaternion
>>> from sympy.abc import x
>>> q = Quaternion(1, 2, 3, 4)
>>> q.integrate(x)
x + 2*x*i + 3*x*j + 4*x*k
```

Definite integral of quaternion :

```
>>> from sympy import Quaternion
>>> from sympy.abc import x
>>> q = Quaternion(1, 2, 3, 4)
>>> q.integrate(x, 1, 5)
4 + 8*i + 12*j + 16*k
```

inverse()

Returns the inverse of the quaternion.

is_pure()

Returns true if the quaternion is pure, false if the quaternion is not pure or returns none if it is unknown.

Explanation

A pure quaternion (also a vector quaternion) is a quaternion with scalar part equal to 0.

Examples

```
>>> from sympy.algebras.quaternion import Quaternion
>>> q = Quaternion(0, 8, 13, 12)
>>> q.is_pure()
True
```

See also:

[*scalar_part*](#) (page 895)

is_zero_quaternion()

Returns true if the quaternion is a zero quaternion or false if it is not a zero quaternion and None if the value is unknown.

Explanation

A zero quaternion is a quaternion with both scalar part and vector part equal to 0.

Examples

```
>>> from sympy.algebras.quaternion import Quaternion
>>> q = Quaternion(1, 0, 0, 0)
>>> q.is_zero_quaternion()
False
```

```
>>> q = Quaternion(0, 0, 0, 0)
>>> q.is_zero_quaternion()
True
```

See also:

[*scalar_part*](#) (page 895), [*vector_part*](#) (page 898)

tensor()

Returns the natural logarithm of the norm(magnitude) of the quaternion.

Examples

```
>>> from sympy.algebras.quaternion import Quaternion
>>> q = Quaternion(2, 4, 2, 4)
>>> q.tensor()
log(2*sqrt(10))
>>> q.norm()
2*sqrt(10)
```

See also:

[*norm*](#) (page 892)

mul(other)

Multiplies quaternions.

Parameters

other : Quaternion or symbol

The quaternion to multiply to current (self) quaternion.

Returns

Quaternion

The resultant quaternion after multiplying self with other

Examples

```
>>> from sympy import Quaternion
>>> from sympy import symbols
>>> q1 = Quaternion(1, 2, 3, 4)
>>> q2 = Quaternion(5, 6, 7, 8)
>>> q1.mul(q2)
(-60) + 12*i + 30*j + 24*k
>>> q1.mul(2)
2 + 4*i + 6*j + 8*k
>>> x = symbols('x', real = True)
>>> q1.mul(x)
x + 2*x*i + 3*x*j + 4*x*k
```

Quaternions over complex fields :

```
>>> from sympy import Quaternion
>>> from sympy import I
>>> q3 = Quaternion(3 + 4*I, 2 + 5*I, 0, 7 + 8*I, real_field = False)
>>> q3.mul(2 + 3*I)
(2 + 3*I)*(3 + 4*I) + (2 + 3*I)*(2 + 5*I)*i + 0*j + (2 + 3*I)*(7 +
↪ 8*I)*k
```

norm()

Returns the norm of the quaternion.

normalize()

Returns the normalized form of the quaternion.

orthogonal(*other*)

Returns the orthogonality of two quaternions.

Parameters

other : a Quaternion

Returns

True : if the two pure quaternions seen as 3D vectors are orthogonal.

False : if the two pure quaternions seen as 3D vectors are not orthogonal.

None : if the two pure quaternions seen as 3D vectors are orthogonal is unknown.

Explanation

Two pure quaternions are called orthogonal when their product is anti-commutative.

Examples

```
>>> from sympy.algebras.quaternion import Quaternion
>>> q = Quaternion(0, 4, 4, 4)
>>> q1 = Quaternion(0, 8, 8, 8)
>>> q.orthogonal(q1)
False
```

```
>>> q1 = Quaternion(0, 2, 2, 0)
>>> q = Quaternion(0, 2, -2, 0)
>>> q.orthogonal(q1)
True
```

parallel(*other*)

Returns True if the two pure quaternions seen as 3D vectors are parallel.

Parameters

other : a Quaternion

Returns

True : if the two pure quaternions seen as 3D vectors are parallel.

False : if the two pure quaternions seen as 3D vectors are not parallel.

None : if the two pure quaternions seen as 3D vectors are parallel is unknown.

Explanation

Two pure quaternions are called parallel when their vector product is commutative which implies that the quaternions seen as 3D vectors have same direction.

Examples

```
>>> from sympy.algebras.quaternion import Quaternion
>>> q = Quaternion(0, 4, 4, 4)
>>> q1 = Quaternion(0, 8, 8, 8)
>>> q.parallel(q1)
True
```

```
>>> q1 = Quaternion(0, 8, 13, 12)
>>> q.parallel(q1)
False
```

pow(*p*)

Finds the *pth* power of the quaternion.

Parameters

p : int

Power to be applied on quaternion.

Returns

Quaternion

Returns the *p*-th power of the current quaternion. Returns the inverse if *p* = -1.

Examples

```
>>> from sympy import Quaternion
>>> q = Quaternion(1, 2, 3, 4)
>>> q.pow(4)
668 + (-224)*i + (-336)*j + (-448)*k
```

pow_cos_sin(*p*)

Computes the *pth* power in the cos-sin form.

Parameters

p : int

Power to be applied on quaternion.

Returns

Quaternion

The *p*-th power in the cos-sin form.

Examples

```
>>> from sympy import Quaternion
>>> q = Quaternion(1, 2, 3, 4)
>>> q.pow_cos_sin(4)
900*cos(4*acos(sqrt(30)/30))
+ 1800*sqrt(29)*sin(4*acos(sqrt(30)/30))/29*i
+ 2700*sqrt(29)*sin(4*acos(sqrt(30)/30))/29*j
+ 3600*sqrt(29)*sin(4*acos(sqrt(30)/30))/29*k
```

static rotate_point(*pin*, *r*)

Returns the coordinates of the point *pin*(a 3 tuple) after rotation.

Parameters

pin : tuple

A 3-element tuple of coordinates of a point which needs to be rotated.

r : Quaternion or tuple

Axis and angle of rotation.

It's important to note that when *r* is a tuple, it must be of the form (axis, angle)

Returns

tuple

The coordinates of the point after rotation.

Examples

```
>>> from sympy import Quaternion
>>> from sympy import symbols, trigsimp, cos, sin
>>> x = symbols('x')
>>> q = Quaternion(cos(x/2), 0, 0, sin(x/2))
>>> trigsimp(Quaternion.rotate_point((1, 1, 1), q))
(sqrt(2)*cos(x + pi/4), sqrt(2)*sin(x + pi/4), 1)
>>> (axis, angle) = q.to_axis_angle()
>>> trigsimp(Quaternion.rotate_point((1, 1, 1), (axis, angle)))
(sqrt(2)*cos(x + pi/4), sqrt(2)*sin(x + pi/4), 1)
```

scalar_part()

Returns scalar part($\mathbf{S}(q)$) of the quaternion *q*.

Explanation

Given a quaternion $q = a + bi + cj + dk$, returns $\mathbf{S}(q) = a$.

Examples

```
>>> from sympy.algebras.quaternion import Quaternion
>>> q = Quaternion(4, 8, 13, 12)
>>> q.scalar_part()
4
```

to_axis_angle()

Returns the axis and angle of rotation of a quaternion

Returns

tuple

Tuple of (axis, angle)

Examples

```
>>> from sympy import Quaternion
>>> q = Quaternion(1, 1, 1, 1)
>>> (axis, angle) = q.to_axis_angle()
>>> axis
(sqrt(3)/3, sqrt(3)/3, sqrt(3)/3)
>>> angle
2*pi/3
```

to_rotation_matrix(v=None)

Returns the equivalent rotation transformation matrix of the quaternion which represents rotation about the origin if v is not passed.

Parameters

v : tuple or None

Default value: None

Returns

tuple

Returns the equivalent rotation transformation matrix of the quaternion which represents rotation about the origin if v is not passed.

Examples

```
>>> from sympy import Quaternion
>>> from sympy import symbols, trigsimp, cos, sin
>>> x = symbols('x')
>>> q = Quaternion(cos(x/2), 0, 0, sin(x/2))
>>> trigsimp(q.to_rotation_matrix((1, 1, 1)))
Matrix([
[cos(x), -sin(x), 0, sin(x) - cos(x) + 1],
[sin(x), cos(x), 0, -sin(x) - cos(x) + 1],
[0, 0, 1, 0],
[0, 0, 0, 1]])
```

classmethod `vector_coplanar(q1, q2, q3)`

Returns True if the axis of the pure quaternions seen as 3D vectors *q1*, *q2*, and *q3* are coplanar.

Parameters

q1 : a pure Quaternion.

q2 : a pure Quaternion.

q3 : a pure Quaternion.

Returns

True : if the axis of the pure quaternions seen as 3D vectors *q1*, *q2*, and *q3* are coplanar.

False : if the axis of the pure quaternions seen as 3D vectors *q1*, *q2*, and *q3* are not coplanar.

None : if the axis of the pure quaternions seen as 3D vectors *q1*, *q2*, and *q3* are coplanar is unknown.

Explanation

Three pure quaternions are vector coplanar if the quaternions seen as 3D vectors are coplanar.

Examples

```
>>> from sympy.algebras.quaternion import Quaternion
>>> q1 = Quaternion(0, 4, 4, 4)
>>> q2 = Quaternion(0, 8, 8, 8)
>>> q3 = Quaternion(0, 24, 24, 24)
>>> Quaternion.vector_coplanar(q1, q2, q3)
True
```

```
>>> q1 = Quaternion(0, 8, 16, 8)
>>> q2 = Quaternion(0, 8, 3, 12)
>>> Quaternion.vector_coplanar(q1, q2, q3)
False
```

See also:

[axis](#) (page 888), [is_pure](#) (page 891)

vector_part()

Returns vector part($\mathbf{V}(q)$) of the quaternion q .

Explanation

Given a quaternion $q = a + bi + cj + dk$, returns $\mathbf{V}(q) = bi + cj + dk$.

Examples

```
>>> from sympy.algebras.quaternion import Quaternion
>>> q = Quaternion(1, 1, 1, 1)
>>> q.vector_part()
0 + 1*i + 1*j + 1*k
```

```
>>> q = Quaternion(4, 8, 13, 12)
>>> q.vector_part()
0 + 8*i + 13*j + 12*k
```

Concrete**Hypergeometric terms**

The center stage, in recurrence solving and summations, play hypergeometric terms. Formally these are sequences annihilated by first order linear recurrence operators. In simple words if we are given term $a(n)$ then it is hypergeometric if its consecutive term ratio is a rational function in n .

To check if a sequence is of this type you can use the `is_hypergeometric` method which is available in Basic class. Here is simple example involving a polynomial:

```
>>> from sympy import *
>>> n, k = symbols('n,k')
>>> (n**2 + 1).is_hypergeometric(n)
True
```

Of course polynomials are hypergeometric but are there any more complicated sequences of this type? Here are some trivial examples:

```
>>> factorial(n).is_hypergeometric(n)
True
>>> binomial(n, k).is_hypergeometric(n)
True
>>> rf(n, k).is_hypergeometric(n)
True
>>> ff(n, k).is_hypergeometric(n)
True
```

(continues on next page)

(continued from previous page)

```
>>> gamma(n).is_hypergeometric(n)
True
>>> (2**n).is_hypergeometric(n)
True
```

We see that all species used in summations and other parts of concrete mathematics are hypergeometric. Note also that binomial coefficients and both rising and falling factorials are hypergeometric in both their arguments:

```
>>> binomial(n, k).is_hypergeometric(k)
True
>>> rf(n, k).is_hypergeometric(k)
True
>>> ff(n, k).is_hypergeometric(k)
True
```

To say more, all previously shown examples are valid for integer linear arguments:

```
>>> factorial(2*n).is_hypergeometric(n)
True
>>> binomial(3*n+1, k).is_hypergeometric(n)
True
>>> rf(n+1, k-1).is_hypergeometric(n)
True
>>> ff(n-1, k+1).is_hypergeometric(n)
True
>>> gamma(5*n).is_hypergeometric(n)
True
>>> (2**(n-7)).is_hypergeometric(n)
True
```

However nonlinear arguments make those sequences fail to be hypergeometric:

```
>>> factorial(n**2).is_hypergeometric(n)
False
>>> (2**(n**3 + 1)).is_hypergeometric(n)
False
```

If not only the knowledge of being hypergeometric or not is needed, you can use `hypersimp()` function. It will try to simplify combinatorial expression and if the term given is hypergeometric it will return a quotient of polynomials of minimal degree. Otherwise it will return `None` to say that sequence is not hypergeometric:

```
>>> hypersimp(factorial(2*n), n)
2*(n + 1)*(2*n + 1)
>>> hypersimp(factorial(n**2), n)
```

Concrete Class Reference

class `sympy.concrete.summations.Sum(function, *symbols, **assumptions)`

Represents unevaluated summation.

Explanation

Sum represents a finite or infinite series, with the first argument being the general form of terms in the series, and the second argument being (`dummy_variable`, `start`, `end`), with `dummy_variable` taking all integer values from `start` through `end`. In accordance with long-standing mathematical convention, the end term is included in the summation.

Finite Sums

For finite sums (and sums with symbolic limits assumed to be finite) we follow the summation convention described by Karr [1], especially definition 3 of section 1.4. The sum:

$$\sum_{m \leq i < n} f(i)$$

has *the obvious meaning* for $m < n$, namely:

$$\sum_{m \leq i < n} f(i) = f(m) + f(m+1) + \dots + f(n-2) + f(n-1)$$

with the upper limit value $f(n)$ excluded. The sum over an empty set is zero if and only if $m = n$:

$$\sum_{m \leq i < n} f(i) = 0 \quad \text{for} \quad m = n$$

Finally, for all other sums over empty sets we assume the following definition:

$$\sum_{m \leq i < n} f(i) = - \sum_{n \leq i < m} f(i) \quad \text{for} \quad m > n$$

It is important to note that Karr defines all sums with the upper limit being exclusive. This is in contrast to the usual mathematical notation, but does not affect the summation convention. Indeed we have:

$$\sum_{m \leq i < n} f(i) = \sum_{i=m}^{n-1} f(i)$$

where the difference in notation is intentional to emphasize the meaning, with limits typeset on the top being inclusive.

Examples

```
>>> from sympy.abc import i, k, m, n, x
>>> from sympy import Sum, factorial, oo, IndexedBase, Function
>>> Sum(k, (k, 1, m))
Sum(k, (k, 1, m))
>>> Sum(k, (k, 1, m)).doit()
m**2/2 + m/2
>>> Sum(k**2, (k, 1, m))
Sum(k**2, (k, 1, m))
>>> Sum(k**2, (k, 1, m)).doit()
m**3/3 + m**2/2 + m/6
>>> Sum(x**k, (k, 0, oo))
Sum(x**k, (k, 0, oo))
>>> Sum(x**k, (k, 0, oo)).doit()
Piecewise((1/(1 - x), Abs(x) < 1), (Sum(x**k, (k, 0, oo)), True))
>>> Sum(x**k/factorial(k), (k, 0, oo)).doit()
exp(x)
```

Here are examples to do summation with symbolic indices. You can use either Function or IndexedBase classes:

```
>>> f = Function('f')
>>> Sum(f(n), (n, 0, 3)).doit()
f(0) + f(1) + f(2) + f(3)
>>> Sum(f(n), (n, 0, oo)).doit()
Sum(f(n), (n, 0, oo))
>>> f = IndexedBase('f')
>>> Sum(f[n]**2, (n, 0, 3)).doit()
f[0]**2 + f[1]**2 + f[2]**2 + f[3]**2
```

An example showing that the symbolic result of a summation is still valid for seemingly nonsensical values of the limits. Then the Karr convention allows us to give a perfectly valid interpretation to those sums by interchanging the limits according to the above rules:

```
>>> S = Sum(i, (i, 1, n)).doit()
>>> S
n**2/2 + n/2
>>> S.subs(n, -4)
6
>>> Sum(i, (i, 1, -4)).doit()
6
>>> Sum(-i, (i, -3, 0)).doit()
6
```

An explicit example of the Karr summation convention:

```
>>> S1 = Sum(i**2, (i, m, m+n-1)).doit()
>>> S1
m**2*n + m*n**2 - m*n + n**3/3 - n**2/2 + n/6
>>> S2 = Sum(i**2, (i, m+n, m-1)).doit()
>>> S2
```

(continues on next page)

(continued from previous page)

```

-m**2*n - m*n**2 + m*n - n**3/3 + n**2/2 - n/6
>>> S1 + S2
0
>>> S3 = Sum(i, (i, m, m-1)).doit()
>>> S3
0

```

See also:

[summation](#) (page 915), [Product](#) (page 905), [sympy.concrete.products.product](#) (page 916)

References

[R87], [R88], [R89]

euler_maclaurin(*m=0, n=0, eps=0, eval_integral=True*)

Return an Euler-Maclaurin approximation of self, where *m* is the number of leading terms to sum directly and *n* is the number of terms in the tail.

With *m* = *n* = 0, this is simply the corresponding integral plus a first-order endpoint correction.

Returns (*s*, *e*) where *s* is the Euler-Maclaurin approximation and *e* is the estimated error (taken to be the magnitude of the first omitted term in the tail):

```

>>> from sympy.abc import k, a, b
>>> from sympy import Sum
>>> Sum(1/k, (k, 2, 5)).doit().evalf()
1.28333333333333
>>> s, e = Sum(1/k, (k, 2, 5)).euler_maclaurin()
>>> s
-log(2) + 7/20 + log(5)
>>> from sympy import sstr
>>> print(ssstr((s.evalf(), e.evalf()), full_prec=True))
(1.26629073187415, 0.0175000000000000)

```

The endpoints may be symbolic:

```

>>> s, e = Sum(1/k, (k, a, b)).euler_maclaurin()
>>> s
-log(a) + log(b) + 1/(2*b) + 1/(2*a)
>>> e
Abs(1/(12*b**2) - 1/(12*a**2))

```

If the function is a polynomial of degree at most $2n+1$, the Euler-Maclaurin formula becomes exact (and *e* = 0 is returned):

```

>>> Sum(k, (k, 2, b)).euler_maclaurin()
(b**2/2 + b/2 - 1, 0)
>>> Sum(k, (k, 2, b)).doit()
b**2/2 + b/2 - 1

```

With a nonzero eps specified, the summation is ended as soon as the remainder term is less than the epsilon.

eval_zeta_function(*f, limits*)

Check whether the function matches with the zeta function. If it matches, then return a *Piecewise* expression because zeta function does not converge unless $s > 1$ and $q > 0$

is_absolutely_convergent()

Checks for the absolute convergence of an infinite series.

Same as checking convergence of absolute value of `sequence_term` of an infinite series.

Examples

```
>>> from sympy import Sum, Symbol, oo
>>> n = Symbol('n', integer=True)
>>> Sum((-1)**n, (n, 1, oo)).is_absolutely_convergent()
False
>>> Sum((-1)**n/n**2, (n, 1, oo)).is_absolutely_convergent()
True
```

See also:

[*Sum.is_convergent*](#) (page 903)

References

[R90]

is_convergent()

Checks for the convergence of a Sum.

Explanation

We divide the study of convergence of infinite sums and products in two parts.

First Part: One part is the question whether all the terms are well defined, i.e., they are finite in a sum and also non-zero in a product. Zero is the analogy of (minus) infinity in products as $e^{-\infty} = 0$.

Second Part: The second part is the question of convergence after infinities, and zeros in products, have been omitted assuming that their number is finite. This means that we only consider the tail of the sum or product, starting from some point after which all terms are well defined.

For example, in a sum of the form:

$$\sum_{1 \leq i < \infty} \frac{1}{n^2 + an + b}$$

where a and b are numbers. The routine will return true, even if there are infinities in the term sequence (at most two). An analogous product would be:

$$\prod_{1 \leq i < \infty} e^{\frac{1}{n^2 + an + b}}$$

This is how convergence is interpreted. It is concerned with what happens at the limit. Finding the bad terms is another independent matter.

Note: It is responsibility of user to see that the sum or product is well defined.

There are various tests employed to check the convergence like divergence test, root test, integral test, alternating series test, comparison tests, Dirichlet tests. It returns true if Sum is convergent and false if divergent and NotImplementedError if it cannot be checked.

Examples

```
>>> from sympy import factorial, S, Sum, Symbol, oo
>>> n = Symbol('n', integer=True)
>>> Sum(n/(n - 1), (n, 4, 7)).is_convergent()
True
>>> Sum(n/(2*n + 1), (n, 1, oo)).is_convergent()
False
>>> Sum(factorial(n)/5**n, (n, 1, oo)).is_convergent()
False
>>> Sum(1/n**(S(6)/5), (n, 1, oo)).is_convergent()
True
```

See also:

[*Sum.is_absolutely_convergent*](#) (page 903), [*sympy.concrete.products.Product.is_convergent*](#) (page 908)

References

[R91]

reverse_order(*indices)

Reverse the order of a limit in a Sum.

Explanation

`reverse_order(self, *indices)` reverses some limits in the expression `self` which can be either a Sum or a Product. The selectors in the argument `indices` specify some indices whose limits get reversed. These selectors are either variable names or numerical indices counted starting from the inner-most limit tuple.

Examples

```
>>> from sympy import Sum
>>> from sympy.abc import x, y, a, b, c, d
```

```
>>> Sum(x, (x, 0, 3)).reverse_order(x)
Sum(-x, (x, 4, -1))
>>> Sum(x*y, (x, 1, 5), (y, 0, 6)).reverse_order(x, y)
Sum(x*y, (x, 6, 0), (y, 7, -1))
>>> Sum(x, (x, a, b)).reverse_order(x)
Sum(-x, (x, b + 1, a - 1))
>>> Sum(x, (x, a, b)).reverse_order(0)
Sum(-x, (x, b + 1, a - 1))
```

While one should prefer variable names when specifying which limits to reverse, the index counting notation comes in handy in case there are several symbols with the same name.

```
>>> S = Sum(x**2, (x, a, b), (x, c, d))
>>> S
Sum(x**2, (x, a, b), (x, c, d))
>>> S0 = S.reverse_order(0)
>>> S0
Sum(-x**2, (x, b + 1, a - 1), (x, c, d))
>>> S1 = S0.reverse_order(1)
>>> S1
Sum(x**2, (x, b + 1, a - 1), (x, d + 1, c - 1))
```

Of course we can mix both notations:

```
>>> Sum(x*y, (x, a, b), (y, 2, 5)).reverse_order(x, 1)
Sum(x*y, (x, b + 1, a - 1), (y, 6, 1))
>>> Sum(x*y, (x, a, b), (y, 2, 5)).reverse_order(y, x)
Sum(x*y, (x, b + 1, a - 1), (y, 6, 1))
```

See also:

[sympy.concrete.expr_with_intlimits.ExprWithIntLimits.index](#) (page 913),
[reorder_limit](#) (page 914), [sympy.concrete.expr_with_intlimits.ExprWithIntLimits.reorder](#) (page 914)

References

[R92]

class sympy.concrete.products.**Product**(*function*, **symbols*, ***assumptions*)
 Represents unevaluated products.

Explanation

`Product` represents a finite or infinite product, with the first argument being the general form of terms in the series, and the second argument being `(dummy_variable, start, end)`, with `dummy_variable` taking all integer values from `start` through `end`. In accordance with long-standing mathematical convention, the end term is included in the product.

Finite Products

For finite products (and products with symbolic limits assumed to be finite) we follow the analogue of the summation convention described by Karr [1], especially definition 3 of section 1.4. The product:

$$\prod_{m \leq i < n} f(i)$$

has the obvious meaning for $m < n$, namely:

$$\prod_{m \leq i < n} f(i) = f(m)f(m+1) \cdot \dots \cdot f(n-2)f(n-1)$$

with the upper limit value $f(n)$ excluded. The product over an empty set is one if and only if $m = n$:

$$\prod_{m \leq i < n} f(i) = 1 \quad \text{for } m = n$$

Finally, for all other products over empty sets we assume the following definition:

$$\prod_{m \leq i < n} f(i) = \frac{1}{\prod_{n \leq i < m} f(i)} \quad \text{for } m > n$$

It is important to note that above we define all products with the upper limit being exclusive. This is in contrast to the usual mathematical notation, but does not affect the product convention. Indeed we have:

$$\prod_{m \leq i < n} f(i) = \prod_{i=m}^{n-1} f(i)$$

where the difference in notation is intentional to emphasize the meaning, with limits typeset on the top being inclusive.

Examples

```
>>> from sympy.abc import a, b, i, k, m, n, x
>>> from sympy import Product, oo
>>> Product(k, (k, 1, m))
Product(k, (k, 1, m))
>>> Product(k, (k, 1, m)).doit()
factorial(m)
>>> Product(k**2, (k, 1, m))
Product(k**2, (k, 1, m))
>>> Product(k**2, (k, 1, m)).doit()
factorial(m)**2
```

Wallis' product for pi:

```
>>> W = Product(2*i/(2*i-1) * 2*i/(2*i+1), (i, 1, oo))
>>> W
Product(4*i**2/((2*i - 1)*(2*i + 1)), (i, 1, oo))
```

Direct computation currently fails:

```
>>> W.doit()
Product(4*i**2/((2*i - 1)*(2*i + 1)), (i, 1, oo))
```

But we can approach the infinite product by a limit of finite products:

```
>>> from sympy import limit
>>> W2 = Product(2*i/(2*i-1)*2*i/(2*i+1), (i, 1, n))
>>> W2
Product(4*i**2/((2*i - 1)*(2*i + 1)), (i, 1, n))
>>> W2e = W2.doit()
>>> W2e
4**n*factorial(n)**2/(2**(2*n)*RisingFactorial(1/2, n)*RisingFactorial(3/
→2, n))
>>> limit(W2e, n, oo)
pi/2
```

By the same formula we can compute $\sin(\pi/2)$:

```
>>> from sympy import combsimp, pi, gamma, simplify
>>> P = pi * x * Product(1 - x**2/k**2, (k, 1, n))
>>> P = P.subs(x, pi/2)
>>> P
pi**2*Product(1 - pi**2/(4*k**2), (k, 1, n))/2
>>> Pe = P.doit()
>>> Pe
pi**2*RisingFactorial(1 - pi/2, n)*RisingFactorial(1 + pi/2, n)/
→(2*factorial(n)**2)
>>> limit(Pe, n, oo).gammasimp()
sin(pi**2/2)
>>> Pe.rewrite(gamma)
(-1)**n*pi**2*gamma(pi/2)*gamma(n + 1 + pi/2)/(2*gamma(1 + pi/2)*gamma(-
→n + pi/2)*gamma(n + 1)**2)
```

Products with the lower limit being larger than the upper one:

```
>>> Product(1/i, (i, 6, 1)).doit()
120
>>> Product(i, (i, 2, 5)).doit()
120
```

The empty product:

```
>>> Product(i, (i, n, n-1)).doit()
1
```

An example showing that the symbolic result of a product is still valid for seemingly nonsensical values of the limits. Then the Karr convention allows us to give a perfectly

valid interpretation to those products by interchanging the limits according to the above rules:

```
>>> P = Product(2, (i, 10, n)).doit()
>>> P
2**(n - 9)
>>> P.subs(n, 5)
1/16
>>> Product(2, (i, 10, 5)).doit()
1/16
>>> 1/Product(2, (i, 6, 9)).doit()
1/16
```

An explicit example of the Karr summation convention applied to products:

```
>>> P1 = Product(x, (i, a, b)).doit()
>>> P1
x**(-a + b + 1)
>>> P2 = Product(x, (i, b+1, a-1)).doit()
>>> P2
x**(a - b - 1)
>>> simplify(P1 * P2)
1
```

And another one:

```
>>> P1 = Product(i, (i, b, a)).doit()
>>> P1
RisingFactorial(b, a - b + 1)
>>> P2 = Product(i, (i, a+1, b-1)).doit()
>>> P2
RisingFactorial(a + 1, -a + b - 1)
>>> P1 * P2
RisingFactorial(b, a - b + 1)*RisingFactorial(a + 1, -a + b - 1)
>>> combsimp(P1 * P2)
1
```

See also:

[Sum](#) (page 900), [summation](#) (page 915), [product](#) (page 916)

References

[R93], [R94], [R95]

`is_convergent()`

See docs of [Sum.is_convergent\(\)](#) (page 903) for explanation of convergence in SymPy.

Explanation

The infinite product:

$$\prod_{1 \leq i < \infty} f(i)$$

is defined by the sequence of partial products:

$$\prod_{i=1}^n f(i) = f(1)f(2)\cdots f(n)$$

as n increases without bound. The product converges to a non-zero value if and only if the sum:

$$\sum_{1 \leq i < \infty} \log f(n)$$

converges.

Examples

```
>>> from sympy import Product, Symbol, cos, pi, exp, oo
>>> n = Symbol('n', integer=True)
>>> Product(n/(n + 1), (n, 1, oo)).is_convergent()
False
>>> Product(1/n**2, (n, 1, oo)).is_convergent()
False
>>> Product(cos(pi/n), (n, 1, oo)).is_convergent()
True
>>> Product(exp(-n**2), (n, 1, oo)).is_convergent()
False
```

References

[R96]

reverse_order(*indices)

Reverse the order of a limit in a Product.

Explanation

`reverse_order(expr, *indices)` reverses some limits in the expression `expr` which can be either a `Sum` or a `Product`. The selectors in the argument `indices` specify some indices whose limits get reversed. These selectors are either variable names or numerical indices counted starting from the inner-most limit tuple.

Examples

```
>>> from sympy import gamma, Product, simplify, Sum
>>> from sympy.abc import x, y, a, b, c, d
>>> P = Product(x, (x, a, b))
>>> Pr = P.reverse_order(x)
>>> Pr
Product(1/x, (x, b + 1, a - 1))
>>> Pr = Pr.doit()
>>> Pr
1/RisingFactorial(b + 1, a - b - 1)
>>> simplify(Pr.rewrite(gamma))
Piecewise((gamma(b + 1)/gamma(a), b > -1), ((-1)**(-a + b + 1)
↳ 1)*gamma(1 - a)/gamma(-b), True))
>>> P = P.doit()
>>> P
RisingFactorial(a, -a + b + 1)
>>> simplify(P.rewrite(gamma))
Piecewise((gamma(b + 1)/gamma(a), a > 0), ((-1)**(-a + b + 1)*gamma(1
↳ - a)/gamma(-b), True))
```

While one should prefer variable names when specifying which limits to reverse, the index counting notation comes in handy in case there are several symbols with the same name.

```
>>> S = Sum(x*y, (x, a, b), (y, c, d))
>>> S
Sum(x*y, (x, a, b), (y, c, d))
>>> S0 = S.reverse_order(0)
>>> S0
Sum(-x*y, (x, b + 1, a - 1), (y, c, d))
>>> S1 = S0.reverse_order(1)
>>> S1
Sum(x*y, (x, b + 1, a - 1), (y, d + 1, c - 1))
```

Of course we can mix both notations:

```
>>> Sum(x*y, (x, a, b), (y, 2, 5)).reverse_order(x, 1)
Sum(x*y, (x, b + 1, a - 1), (y, 6, 1))
>>> Sum(x*y, (x, a, b), (y, 2, 5)).reverse_order(y, x)
Sum(x*y, (x, b + 1, a - 1), (y, 6, 1))
```

See also:

[*sympy.concrete.expr_with_intlimits.ExprWithIntLimits.index*](#) (page 913),
[*reorder_limit*](#) (page 914), [*sympy.concrete.expr_with_intlimits.ExprWithIntLimits.reorder*](#) (page 914)

References

[R97]

class `sympy.concrete.expr_with_intlimits.ExprWithIntLimits`(*function*, **symbols*, ***assumptions*)

Superclass for Product and Sum.

See also:

[`sympy.concrete.expr\_with\_limits.ExprWithLimits`](#) (page 606), [`sympy.concrete.products.Product`](#) (page 905), [`sympy.concrete.summations.Sum`](#) (page 900)

change_index(*var*, *trafo*, *newvar*=None)

Change index of a Sum or Product.

Perform a linear transformation $x \mapsto ax + b$ on the index variable x . For a the only values allowed are ± 1 . A new variable to be used after the change of index can also be specified.

Explanation

`change_index(expr, var, trafo, newvar=None)` where `var` specifies the index variable x to transform. The transformation `trafo` must be linear and given in terms of `var`. If the optional argument `newvar` is provided then `var` gets replaced by `newvar` in the final expression.

Examples

```
>>> from sympy import Sum, Product, simplify
>>> from sympy.abc import x, y, a, b, c, d, u, v, i, j, k, l
```

```
>>> S = Sum(x, (x, a, b))
>>> S.doit()
-a**2/2 + a/2 + b**2/2 + b/2
```

```
>>> Sn = S.change_index(x, x + 1, y)
>>> Sn
Sum(y - 1, (y, a + 1, b + 1))
>>> Sn.doit()
-a**2/2 + a/2 + b**2/2 + b/2
```

```
>>> Sn = S.change_index(x, -x, y)
>>> Sn
Sum(-y, (y, -b, -a))
>>> Sn.doit()
-a**2/2 + a/2 + b**2/2 + b/2
```

```
>>> Sn = S.change_index(x, x+u)
>>> Sn
Sum(-u + x, (x, a + u, b + u))
```

(continues on next page)

(continued from previous page)

```
>>> Sn.doit()
-a**2/2 - a*u + a/2 + b**2/2 + b*u + b/2 - u*(-a + b + 1) + u
>>> simplify(Sn.doit())
-a**2/2 + a/2 + b**2/2 + b/2
```

```
>>> Sn = S.change_index(x, -x - u, y)
>>> Sn
Sum(-u - y, (y, -b - u, -a - u))
>>> Sn.doit()
-a**2/2 - a*u + a/2 + b**2/2 + b*u + b/2 - u*(-a + b + 1) + u
>>> simplify(Sn.doit())
-a**2/2 + a/2 + b**2/2 + b/2
```

```
>>> P = Product(i*j**2, (i, a, b), (j, c, d))
>>> P
Product(i*j**2, (i, a, b), (j, c, d))
>>> P2 = P.change_index(i, i+3, k)
>>> P2
Product(j**2*(k - 3), (k, a + 3, b + 3), (j, c, d))
>>> P3 = P2.change_index(j, -j, l)
>>> P3
Product(l**2*(k - 3), (k, a + 3, b + 3), (l, -d, -c))
```

When dealing with symbols only, we can make a general linear transformation:

```
>>> Sn = S.change_index(x, u*x+v, y)
>>> Sn
Sum((-v + y)/u, (y, b*u + v, a*u + v))
>>> Sn.doit()
-v*(a*u - b*u + 1)/u + (a**2*u**2/2 + a*u*v + a*u/2 - b**2*u**2/2 -
↳ b*u*v + b*u/2 + v)/u
>>> simplify(Sn.doit())
a**2*u/2 + a/2 - b**2*u/2 + b/2
```

However, the last result can be inconsistent with usual summation where the index increment is always 1. This is obvious as we get back the original value only for u equal +1 or -1.

See also:

[sympy.concrete.expr_with_intlimits.ExprWithIntLimits.index](#) (page 913), [reorder_limit](#) (page 914), [sympy.concrete.expr_with_intlimits.ExprWithIntLimits.reorder](#) (page 914), [sympy.concrete.summations.Sum.reverse_order](#) (page 904), [sympy.concrete.products.Product.reverse_order](#) (page 909)

property `has_empty_sequence`

Returns True if the Sum or Product is computed for an empty sequence.

Examples

```
>>> from sympy import Sum, Product, Symbol
>>> m = Symbol('m')
>>> Sum(m, (m, 1, 0)).has_empty_sequence
True
```

```
>>> Sum(m, (m, 1, 1)).has_empty_sequence
False
```

```
>>> M = Symbol('M', integer=True, positive=True)
>>> Product(m, (m, 1, M)).has_empty_sequence
False
```

```
>>> Product(m, (m, 2, M)).has_empty_sequence
```

```
>>> Product(m, (m, M + 1, M)).has_empty_sequence
True
```

```
>>> N = Symbol('N', integer=True, positive=True)
>>> Sum(m, (m, N, M)).has_empty_sequence
```

```
>>> N = Symbol('N', integer=True, negative=True)
>>> Sum(m, (m, N, M)).has_empty_sequence
False
```

See also:

[*has_reversed_limits*](#) (page 607), [*has_finite_limits*](#) (page 607)

`index(x)`

Return the index of a dummy variable in the list of limits.

Explanation

`index(expr, x)` returns the index of the dummy variable `x` in the limits of `expr`. Note that we start counting with 0 at the inner-most limits tuple.

Examples

```
>>> from sympy.abc import x, y, a, b, c, d
>>> from sympy import Sum, Product
>>> Sum(x*y, (x, a, b), (y, c, d)).index(x)
0
>>> Sum(x*y, (x, a, b), (y, c, d)).index(y)
1
>>> Product(x*y, (x, a, b), (y, c, d)).index(x)
0
>>> Product(x*y, (x, a, b), (y, c, d)).index(y)
1
```

See also:

[reorder_limit](#) (page 914), [reorder](#) (page 914), [sympy.concrete.summations.Sum.reverse_order](#) (page 904), [sympy.concrete.products.Product.reverse_order](#) (page 909)

reorder(*arg)

Reorder limits in a expression containing a Sum or a Product.

Explanation

`expr.reorder(*arg)` reorders the limits in the expression `expr` according to the list of tuples given by `arg`. These tuples can contain numerical indices or index variable names or involve both.

Examples

```
>>> from sympy import Sum, Product
>>> from sympy.abc import x, y, z, a, b, c, d, e, f
```

```
>>> Sum(x*y, (x, a, b), (y, c, d)).reorder((x, y))
Sum(x*y, (y, c, d), (x, a, b))
```

```
>>> Sum(x*y*z, (x, a, b), (y, c, d), (z, e, f)).reorder((x, y), (x,
↪ z), (y, z))
Sum(x*y*z, (z, e, f), (y, c, d), (x, a, b))
```

```
>>> P = Product(x*y*z, (x, a, b), (y, c, d), (z, e, f))
>>> P.reorder((x, y), (x, z), (y, z))
Product(x*y*z, (z, e, f), (y, c, d), (x, a, b))
```

We can also select the index variables by counting them, starting with the inner-most one:

```
>>> Sum(x**2, (x, a, b), (x, c, d)).reorder((0, 1))
Sum(x**2, (x, c, d), (x, a, b))
```

And of course we can mix both schemes:

```
>>> Sum(x*y, (x, a, b), (y, c, d)).reorder((y, x))
Sum(x*y, (y, c, d), (x, a, b))
>>> Sum(x*y, (x, a, b), (y, c, d)).reorder((y, 0))
Sum(x*y, (y, c, d), (x, a, b))
```

See also:

[reorder_limit](#) (page 914), [index](#) (page 913), [sympy.concrete.summations.Sum.reverse_order](#) (page 904), [sympy.concrete.products.Product.reverse_order](#) (page 909)

reorder_limit(x, y)

Interchange two limit tuples of a Sum or Product expression.

Explanation

`expr.reorder_limit(x, y)` interchanges two limit tuples. The arguments `x` and `y` are integers corresponding to the index variables of the two limits which are to be interchanged. The expression `expr` has to be either a `Sum` or a `Product`.

Examples

```
>>> from sympy.abc import x, y, z, a, b, c, d, e, f
>>> from sympy import Sum, Product
```

```
>>> Sum(x*y*z, (x, a, b), (y, c, d), (z, e, f)).reorder_limit(0, 2)
Sum(x*y*z, (z, e, f), (y, c, d), (x, a, b))
>>> Sum(x**2, (x, a, b), (x, c, d)).reorder_limit(1, 0)
Sum(x**2, (x, c, d), (x, a, b))
```

```
>>> Product(x*y*z, (x, a, b), (y, c, d), (z, e, f)).reorder_limit(0, 2)
Product(x*y*z, (z, e, f), (y, c, d), (x, a, b))
```

See also:

[index](#) (page 913), [reorder](#) (page 914), [sympy.concrete.summations.Sum.reverse_order](#) (page 904), [sympy.concrete.products.Product.reverse_order](#) (page 909)

Concrete Functions Reference

`sympy.concrete.summations.summation(f, *symbols, **kwargs)`

Compute the summation of `f` with respect to `symbols`.

Explanation

The notation for `symbols` is similar to the notation used in `Integral`. `summation(f, (i, a, b))` computes the sum of `f` with respect to `i` from `a` to `b`, i.e.,

$$\text{summation}(f, (i, a, b)) = \sum_{i=a}^b f$$

If it cannot compute the sum, it returns an unevaluated `Sum` object. Repeated sums can be computed by introducing additional symbols tuples:

```
.. rubric:: Examples
```

```
>>> from sympy import summation, oo, symbols, log
>>> i, n, m = symbols('i n m', integer=True)
```

```
>>> summation(2*i - 1, (i, 1, n))
n**2
>>> summation(1/2**i, (i, 0, oo))
2
>>> summation(1/log(n)**n, (n, 2, oo))
Sum(log(n)**(-n), (n, 2, oo))
>>> summation(i, (i, 0, n), (n, 0, m))
m**3/6 + m**2/2 + m/3
```

```
>>> from sympy.abc import x
>>> from sympy import factorial
>>> summation(x**n/factorial(n), (n, 0, oo))
exp(x)
```

See also:

[Sum](#) (page 900), [Product](#) (page 905), [sympy.concrete.products.product](#) (page 916)

`sympy.concrete.products.product(*args, **kwargs)`

Compute the product.

Explanation

The notation for symbols is similar to the notation used in `Sum` or `Integral`. `product(f, (i, a, b))` computes the product of `f` with respect to `i` from `a` to `b`, i.e.,

$$\text{product}(f(n), (i, a, b)) = \prod_{i=a}^b f(n)$$

If it cannot compute the product, it returns an unevaluated `Product` object. Repeated products can be computed by introducing additional symbols tuples:

```
.. rubric:: Examples
```

```
>>> from sympy import product, symbols
>>> i, n, m, k = symbols('i n m k', integer=True)
```

```
>>> product(i, (i, 1, k))
factorial(k)
>>> product(m, (i, 1, k))
m**k
>>> product(i, (i, 1, k), (k, 1, n))
Product(factorial(k), (k, 1, n))
```

`sympy.concrete.gosper.gosper_normal(f, g, n, polys=True)`

Compute the Gosper's normal form of `f` and `g`.

Explanation

Given relatively prime univariate polynomials f and g , rewrite their quotient to a normal form defined as follows:

$$\frac{f(n)}{g(n)} = Z \cdot \frac{A(n)C(n+1)}{B(n)C(n)}$$

where Z is an arbitrary constant and A, B, C are monic polynomials in n with the following properties:

1. $\gcd(A(n), B(n+h)) = 1 \forall h \in \mathbb{N}$
2. $\gcd(B(n), C(n+1)) = 1$
3. $\gcd(A(n), C(n)) = 1$

This normal form, or rational factorization in other words, is a crucial step in Gosper's algorithm and in solving of difference equations. It can be also used to decide if two hypergeometric terms are similar or not.

This procedure will return a tuple containing elements of this factorization in the form $(Z*A, B, C)$.

Examples

```
>>> from sympy.concrete.gosper import gosper_normal
>>> from sympy.abc import n
```

```
>>> gosper_normal(4*n+5, 2*(4*n+1)*(2*n+3), n, polys=False)
(1/4, n + 3/2, n + 1/4)
```

`sympy.concrete.gosper.gosper_term( $f, n$ )`
 Compute Gosper's hypergeometric term for f .

Explanation

Suppose f is a hypergeometric term such that:

$$s_n = \sum_{k=0}^{n-1} f_k$$

and f_k does not depend on n . Returns a hypergeometric term g_n such that $g_{n+1} - g_n = f_n$.

Examples

```
>>> from sympy.concrete.gosper import gosper_term
>>> from sympy import factorial
>>> from sympy.abc import n
```

```
>>> gosper_term((4*n + 1)*factorial(n)/factorial(2*n + 1), n)
(-n - 1/2)/(n + 1/4)
```

`sympy.concrete.gosper.gosper_sum(f, k)`
Gosper's hypergeometric summation algorithm.

Explanation

Given a hypergeometric term f such that:

$$s_n = \sum_{k=0}^{n-1} f_k$$

and $f(n)$ does not depend on n , returns $g_n - g(0)$ where $g_{n+1} - g_n = f_n$, or `None` if s_n cannot be expressed in closed form as a sum of hypergeometric terms.

Examples

```
>>> from sympy.concrete.gosper import gosper_sum
>>> from sympy import factorial
>>> from sympy.abc import n, k
```

```
>>> f = (4*k + 1)*factorial(k)/factorial(2*k + 1)
>>> gosper_sum(f, (k, 0, n))
(-factorial(n) + 2*factorial(2*n + 1))/factorial(2*n + 1)
>>> _.subs(n, 2) == sum(f.subs(k, i) for i in [0, 1, 2])
True
>>> gosper_sum(f, (k, 3, n))
(-60*factorial(n) + factorial(2*n + 1))/(60*factorial(2*n + 1))
>>> _.subs(n, 5) == sum(f.subs(k, i) for i in [3, 4, 5])
True
```

References

[R98]

Core

sympify

sympify

`sympy.core.sympify.sympify(a, locals=None, convert_xor=True, strict=False, rational=False, evaluate=None)`

Converts an arbitrary expression to a type that can be used inside SymPy.

Parameters

a :

- any object defined in SymPy
- standard numeric Python types: `int`, `long`, `float`, `Decimal`

- strings (like "0.09", "2e-19" or 'sin(x)')
- booleans, including None (will leave None unchanged)
- dicts, lists, sets or tuples containing any of the above

convert_xor : bool, optional

If true, treats ^ as exponentiation. If False, treats ^ as XOR itself. Used only when input is a string.

locals : any object defined in SymPy, optional

In order to have strings be recognized it can be imported into a namespace dictionary and passed as locals.

strict : bool, optional

If the option strict is set to True, only the types for which an explicit conversion has been defined are converted. In the other cases, a SympifyError is raised.

rational : bool, optional

If True, converts floats into *Rational* (page 985). If False, it lets floats remain as it is. Used only when input is a string.

evaluate : bool, optional

If False, then arithmetic and operators will be converted into their SymPy equivalents. If True the expression will be evaluated and the result will be returned.

Explanation

It will convert Python ints into instances of *Integer* (page 987), floats into instances of *Float* (page 982), etc. It is also able to coerce symbolic expressions which inherit from *Basic* (page 927). This can be useful in cooperation with SAGE.

Warning: Note that this function uses `eval`, and thus shouldn't be used on unsanitized input.

If the argument is already a type that SymPy understands, it will do nothing but return that value. This can be used at the beginning of a function to ensure you are working with the correct type.

Examples

```
>>> from sympy import sympify
```

```
>>> sympify(2).is_integer
True
>>> sympify(2).is_real
True
```

```
>>> sympify(2.0).is_real
True
>>> sympify("2.0").is_real
True
>>> sympify("2e-45").is_real
True
```

If the expression could not be converted, a `SympifyError` is raised.

```
>>> sympify("x***2")
Traceback (most recent call last):
...
SympifyError: SympifyError: "could not parse 'x***2'"
```

Locals

The sympification happens with access to everything that is loaded by `from sympy import *`; anything used in a string that is not defined by that import will be converted to a symbol. In the following, the `bitcount` function is treated as a symbol and the `0` is interpreted as the `Order` (page 627) object (used with series) and it raises an error when used improperly:

```
>>> s = 'bitcount(42)'
>>> sympify(s)
bitcount(42)
>>> sympify("0(x)")
0(x)
>>> sympify("0 + 1")
Traceback (most recent call last):
...
TypeError: unbound method...
```

In order to have `bitcount` be recognized it can be imported into a namespace dictionary and passed as locals:

```
>>> ns = {}
>>> exec('from sympy.core.evalf import bitcount', ns)
>>> sympify(s, locals=ns)
6
```

In order to have the `0` interpreted as a `Symbol`, identify it as such in the namespace dictionary. This can be done in a variety of ways; all three of the following are possibilities:

```
>>> from sympy import Symbol
>>> ns["0"] = Symbol("0") # method 1
>>> exec('from sympy.abc import 0', ns) # method 2
>>> ns.update(dict(0=Symbol("0"))) # method 3
>>> sympify("0 + 1", locals=ns)
0 + 1
```

If you want *all* single-letter and Greek-letter variables to be symbols then you can use the clashing-symbols dictionaries that have been defined there as private variables: `_clash1`

(single-letter variables), `_clash2` (the multi-letter Greek names) or `_clash` (both single and multi-letter names that are defined in `abc`).

```
>>> from sympy.abc import _clash1
>>> set(_clash1)
{'E', 'I', 'N', 'O', 'Q', 'S'}
>>> sympify('I & Q', _clash1)
I & Q
```

Strict

If the option `strict` is set to `True`, only the types for which an explicit conversion has been defined are converted. In the other cases, a `SympifyError` is raised.

```
>>> print(sympify(None))
None
>>> sympify(None, strict=True)
Traceback (most recent call last):
...
SympifyError: SympifyError: None
```

Deprecated since version 1.6: `sympify(obj)` automatically falls back to `str(obj)` when all other conversion methods fail, but this is deprecated. `strict=True` will disable this deprecated behavior. See *The string fallback in `sympify()`* (page 176).

Evaluation

If the option `evaluate` is set to `False`, then arithmetic and operators will be converted into their SymPy equivalents and the `evaluate=False` option will be added. Nested `Add` or `Mul` will be denested first. This is done via an AST transformation that replaces operators with their SymPy equivalents, so if an operand redefines any of those operations, the redefined operators will not be used. If argument `a` is not a string, the mathematical expression is evaluated before being passed to `sympify`, so adding `evaluate=False` will still return the evaluated result of expression.

```
>>> sympify('2**2 / 3 + 5')
19/3
>>> sympify('2**2 / 3 + 5', evaluate=False)
2**2/3 + 5
>>> sympify('4/2+7', evaluate=True)
9
>>> sympify('4/2+7', evaluate=False)
4/2 + 7
>>> sympify(4/2+7, evaluate=False)
9.000000000000000
```

Extending

To extend sympify to convert custom objects (not derived from Basic), just define a `_sympy_` method to your class. You can do that even to classes that you do not own by subclassing or adding the method at runtime.

```
>>> from sympy import Matrix
>>> class MyList1(object):
...     def __iter__(self):
...         yield 1
...         yield 2
...         return
...     def __getitem__(self, i): return list(self)[i]
...     def _sympy_(self): return Matrix(self)
>>> sympify(MyList1())
Matrix([
[1],
[2]])
```

If you do not have control over the class definition you could also use the converter global dictionary. The key is the class and the value is a function that takes a single argument and returns the desired SymPy object, e.g. `converter[MyList] = lambda x: Matrix(x)`.

```
>>> class MyList2(object): # XXX Do not do this if you control the
↳class!
...     def __iter__(self): #      Use _sympy_!
...         yield 1
...         yield 2
...         return
...     def __getitem__(self, i): return list(self)[i]
>>> from sympy.core.sympify import converter
>>> converter[MyList2] = lambda x: Matrix(x)
>>> sympify(MyList2())
Matrix([
[1],
[2]])
```

Notes

The keywords `rational` and `convert_xor` are only used when the input is a string.

Convert_xor

```
>>> sympify('x^y', convert_xor=True)
x**y
>>> sympify('x^y', convert_xor=False)
x ^ y
```

Rational

```
>>> sympify('0.1', rational=False)
0.1
>>> sympify('0.1', rational=True)
1/10
```

Sometimes autosimplification during sympification results in expressions that are very different in structure than what was entered. Until such autosimplification is no longer done, the `kernS` function might be of some use. In the example below you can see how an expression reduces to -1 by autosimplification, but does not do so when `kernS` is used.

```
>>> from sympy.core.sympify import kernS
>>> from sympy.abc import x
>>> -2*(-(-x + 1/x)/(x*(x - 1/x)**2) - 1/(x*(x - 1/x))) - 1
-1
>>> s = '-2*(-(-x + 1/x)/(x*(x - 1/x)**2) - 1/(x*(x - 1/x))) - 1'
>>> sympify(s)
-1
>>> kernS(s)
-2*(-(-x + 1/x)/(x*(x - 1/x)**2) - 1/(x*(x - 1/x))) - 1
```

assumptions

This module contains the machinery handling assumptions. Do also consider the guide [Assumptions](#) (page 71).

All symbolic objects have assumption attributes that can be accessed via `.is_<assumption name>` attribute.

Assumptions determine certain properties of symbolic objects and can have 3 possible values: `True`, `False`, `None`. `True` is returned if the object has the property and `False` is returned if it does not or cannot (i.e. does not make sense):

```
>>> from sympy import I
>>> I.is_algebraic
True
>>> I.is_real
False
>>> I.is_prime
False
```

When the property cannot be determined (or when a method is not implemented) `None` will be returned. For example, a generic symbol, `x`, may or may not be positive so a value of `None` is returned for `x.is_positive`.

By default, all symbolic values are in the largest set in the given context without specifying the property. For example, a symbol that has a property being integer, is also real, complex, etc.

Here follows a list of possible assumption names:

commutative

object commutes with any other object with respect to multiplication operation. See¹².

complex

object can have only values from the set of complex numbers. See¹³.

imaginary

object value is a number that can be written as a real number multiplied by the imaginary unit I . See [R101]. Please note that 0 is not considered to be an imaginary number, see [issue #7649](#).

real

object can have only values from the set of real numbers.

extended_real

object can have only values from the set of real numbers, ∞ and $-\infty$.

integer

object can have only values from the set of integers.

odd**even**

object can have only values from the set of odd (even) integers [R100].

prime

object is a natural number greater than 1 that has no positive divisors other than 1 and itself. See [R104].

composite

object is a positive integer that has at least one positive divisor other than 1 or the number itself. See [R102].

zero

object has the value of 0.

nonzero

object is a real number that is not zero.

rational

object can have only values from the set of rationals.

algebraic

object can have only values from the set of algebraic numbers¹¹.

transcendental

object can have only values from the set of transcendental numbers¹⁰.

irrational

object value cannot be represented exactly by *Rational* (page 985), see [R103].

¹² https://en.wikipedia.org/wiki/Commutative_property

¹³ https://en.wikipedia.org/wiki/Complex_number

¹¹ https://en.wikipedia.org/wiki/Algebraic_number

¹⁰ https://en.wikipedia.org/wiki/Transcendental_number

finite**infinite**

object absolute value is bounded (arbitrarily large). See [\[R105\]](#), [\[R106\]](#), [\[R107\]](#).

negative**nonnegative**

object can have only negative (nonnegative) values [\[R99\]](#).

positive**nonpositive**

object can have only positive (nonpositive) values.

extended_negative**extended_nonnegative****extended_positive****extended_nonpositive****extended_nonzero**

as without the extended part, but also including infinity with corresponding sign, e.g., `extended_positive` includes `oo`

hermitian**antihermitian**

object belongs to the field of Hermitian (antihermitian) operators.

Examples

```
>>> from sympy import Symbol
>>> x = Symbol('x', real=True); x
x
>>> x.is_real
True
>>> x.is_complex
True
```

See Also

See also:

[sympy.core.numbers.ImaginaryUnit](#) (page 1000) [sympy.core.numbers.Zero](#) (page 995) [sympy.core.numbers.One](#) (page 995) [sympy.core.numbers.Infinity](#) (page 998) [sympy.core.numbers.NegativeInfinity](#) (page 998) [sympy.core.numbers.ComplexInfinity](#) (page 999)

Notes

The fully-resolved assumptions for any SymPy expression can be obtained as follows:

```
>>> from sympy.core.assumptions import assumptions
>>> x = Symbol('x', positive=True)
>>> assumptions(x + I)
{'commutative': True, 'complex': True, 'composite': False, 'even':
False, 'extended_negative': False, 'extended_nonnegative': False,
'extended_nonpositive': False, 'extended_nonzero': False,
'extended_positive': False, 'extended_real': False, 'finite': True,
'imaginary': False, 'infinite': False, 'integer': False, 'irrational':
False, 'negative': False, 'noninteger': False, 'nonnegative': False,
'nonpositive': False, 'nonzero': False, 'odd': False, 'positive':
False, 'prime': False, 'rational': False, 'real': False, 'zero':
False}
```

Developers Notes

The current (and possibly incomplete) values are stored in the `obj._assumptions` dictionary; queries to getter methods (with property decorators) or attributes of objects/classes will return values and update the dictionary.

```
>>> eq = x**2 + I
>>> eq._assumptions
{}
>>> eq.is_finite
True
>>> eq._assumptions
{'finite': True, 'infinite': False}
```

For a [Symbol](#) (page 976), there are two locations for assumptions that may be of interest. The `assumptions0` attribute gives the full set of assumptions derived from a given set of initial assumptions. The latter assumptions are stored as `Symbol._assumptions.generator`

```
>>> Symbol('x', prime=True, even=True)._assumptions.generator
{'even': True, 'prime': True}
```

The generator is not necessarily canonical nor is it filtered in any way: it records the assumptions used to instantiate a `Symbol` and (for storage purposes) represents a more compact representation of the assumptions needed to recreate the full set in `Symbol.assumptions0`.

References

cache

cacheit

`sympy.core.cache.__cacheit(maxsize)`

caching decorator.

important: the result of cached function must be *immutable*

Examples

```
>>> from sympy import cacheit
>>> @cacheit
... def f(a, b):
...     return a+b
```

```
>>> @cacheit
... def f(a, b): # noqa: F811
...     return [a, b] # <-- WRONG, returns mutable object
```

to force cacheit to check returned results mutability and consistency, set environment variable SYMPY_USE_CACHE to 'debug'

basic

Basic

`class sympy.core.basic.Basic(*args)`

Base class for all SymPy objects.

Notes And Conventions

- 1) Always use `.args`, when accessing parameters of some instance:

```
>>> from sympy import cot
>>> from sympy.abc import x, y
```

```
>>> cot(x).args
(x,)
```

```
>>> cot(x).args[0]
x
```

```
>>> (x*y).args
(x, y)
```

```
>>> (x*y).args[1]
y
```

2) Never use internal methods or variables (the ones prefixed with `_`):

```
>>> cot(x)._args      # do not use this, use cot(x).args instead
(x,)
```

3) By “SymPy object” we mean something that can be returned by `sympify`. But not all objects one encounters using SymPy are subclasses of `Basic`. For example, mutable objects are not:

```
>>> from sympy import Basic, Matrix, sympify
>>> A = Matrix([[1, 2], [3, 4]]).as_mutable()
>>> isinstance(A, Basic)
False
```

```
>>> B = sympify(A)
>>> isinstance(B, Basic)
True
```

property args: tuple[[sympy.core.basic.Basic](#) (page 927), ...]

Returns a tuple of arguments of ‘self’.

Examples

```
>>> from sympy import cot
>>> from sympy.abc import x, y
```

```
>>> cot(x).args
(x,)
```

```
>>> cot(x).args[0]
x
```

```
>>> (x*y).args
(x, y)
```

```
>>> (x*y).args[1]
y
```


Notes

Never use `self._args`, always use `self.args`. Only use `_args` in `__new__` when creating a new function. Do not override `.args()` from `Basic` (so that it is easy to change the interface in the future if needed).

as_content_primitive(*radical=False, clear=True*)

A stub to allow `Basic` args (like `Tuple`) to be skipped when computing the content and primitive components of an expression.

See also:

[`sympy.core.expr.Expr.as\_content\_primitive`](#) (page 951)

as_dummy()

Return the expression with any objects having structurally bound symbols replaced with unique, canonical symbols within the object in which they appear and having only the default assumption for commutativity being `True`. When applied to a symbol a new symbol having only the same commutativity will be returned.

Examples

```
>>> from sympy import Integral, Symbol
>>> from sympy.abc import x
>>> r = Symbol('r', real=True)
>>> Integral(r, (r, x)).as_dummy()
Integral(_0, (_0, x))
>>> _.variables[0].is_real is None
True
>>> r.as_dummy()
_r
```

Notes

Any object that has structurally bound variables should have a property, `bound_symbols` that returns those symbols appearing in the object.

property assumptions0

Return object *type* assumptions.

For example:

`Symbol('x', real=True)` `Symbol('x', integer=True)`

are different objects. In other words, besides Python type (`Symbol` in this case), the initial assumptions are also forming their typeinfo.

Examples

```
>>> from sympy import Symbol
>>> from sympy.abc import x
>>> x.assumptions0
{'commutative': True}
>>> x = Symbol("x", positive=True)
>>> x.assumptions0
{'commutative': True, 'complex': True, 'extended_negative': False,
 'extended_nonnegative': True, 'extended_nonpositive': False,
 'extended_nonzero': True, 'extended_positive': True, 'extended_real':
 True, 'finite': True, 'hermitian': True, 'imaginary': False,
 'infinite': False, 'negative': False, 'nonnegative': True,
 'nonpositive': False, 'nonzero': True, 'positive': True, 'real':
 True, 'zero': False}
```

`atoms(*types)`

Returns the atoms that form the current object.

By default, only objects that are truly atomic and cannot be divided into smaller pieces are returned: symbols, numbers, and number symbols like I and π . It is possible to request atoms of any type, however, as demonstrated below.

Examples

```
>>> from sympy import I, pi, sin
>>> from sympy.abc import x, y
>>> (1 + x + 2*sin(y + I*pi)).atoms()
{1, 2, I, pi, x, y}
```

If one or more types are given, the results will contain only those types of atoms.

```
>>> from sympy import Number, NumberSymbol, Symbol
>>> (1 + x + 2*sin(y + I*pi)).atoms(Symbol)
{x, y}
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(Number)
{1, 2}
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(Number, NumberSymbol)
{1, 2, pi}
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(Number, NumberSymbol, I)
{1, 2, I, pi}
```

Note that I (imaginary unit) and ∞ (complex infinity) are special types of number symbols and are not part of the `NumberSymbol` class.

The type can be given implicitly, too:

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(x) # x is a Symbol
{x, y}
```

Be careful to check your assumptions when using the implicit option since `S(1).is_Integer = True` but `type(S(1))` is `One`, a special type of SymPy atom, while `type(S(2))` is type `Integer` and will find all integers in an expression:

```
>>> from sympy import S
>>> (1 + x + 2*sin(y + I*pi)).atoms(S(1))
{1}
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(S(2))
{1, 2}
```

Finally, arguments to `atoms()` can select more than atomic atoms: any SymPy type (loaded in `core/_init_.py`) can be listed as an argument and those types of “atoms” as found in scanning the arguments of the expression recursively:

```
>>> from sympy import Function, Mul
>>> from sympy.core.function import AppliedUndef
>>> f = Function('f')
>>> (1 + f(x) + 2*sin(y + I*pi)).atoms(Function)
{f(x), sin(y + I*pi)}
>>> (1 + f(x) + 2*sin(y + I*pi)).atoms(AppliedUndef)
{f(x)}
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(Mul)
{I*pi, 2*sin(y + I*pi)}
```

property `canonical_variables`

Return a dictionary mapping any variable defined in `self.bound_symbols` to Symbols that do not clash with any free symbols in the expression.

Examples

```
>>> from sympy import Lambda
>>> from sympy.abc import x
>>> Lambda(x, 2*x).canonical_variables
{x: _0}
```

classmethod `class_key()`

Nice order of classes.

method `compare(other)`

Return -1, 0, 1 if the object is smaller, equal, or greater than other.

Not in the mathematical sense. If the object is of a different type from the “other” then their classes are ordered according to the `sorted_classes` list.

Examples

```
>>> from sympy.abc import x, y
>>> x.compare(y)
-1
>>> x.compare(x)
0
>>> y.compare(x)
1
```

count(*query*)

Count the number of matching subexpressions.

count_ops(*visual=None*)

wrapper for count_ops that returns the operation count.

doit(***hints*)

Evaluate objects that are not evaluated by default like limits, integrals, sums and products. All objects of this kind will be evaluated recursively, unless some species were excluded via 'hints' or unless the 'deep' hint was set to 'False'.

```
>>> from sympy import Integral
>>> from sympy.abc import x
```

```
>>> 2*Integral(x, x)
2*Integral(x, x)
```

```
>>> (2*Integral(x, x)).doit()
x**2
```

```
>>> (2*Integral(x, x)).doit(deep=False)
2*Integral(x, x)
```

dummy_eq(*other, symbol=None*)

Compare two expressions and handle dummy symbols.

Examples

```
>>> from sympy import Dummy
>>> from sympy.abc import x, y
```

```
>>> u = Dummy('u')
```

```
>>> (u**2 + 1).dummy_eq(x**2 + 1)
True
>>> (u**2 + 1) == (x**2 + 1)
False
```

```
>>> (u**2 + y).dummy_eq(x**2 + y, x)
True
>>> (u**2 + y).dummy_eq(x**2 + y, y)
False
```

find(*query*, *group=False*)

Find all subexpressions matching a query.

property free_symbols: `set[sympy.core.basic.Basic (page 927)]`

Return from the atoms of self those which are free symbols.

Not all free symbols are `Symbol`. Eg: `IndexedBase('t')[0].free_symbols`

For most expressions, all symbols are free symbols. For some classes this is not true. e.g. Integrals use Symbols for the dummy variables which are bound variables, so `Integral` has a method to return all symbols except those. `Derivative` keeps track of symbols with respect to which it will perform a derivative; those are bound variables, too, so it has its own `free_symbols` method.

Any other method that uses bound variables should implement a `free_symbols` method.

classmethod fromiter(*args*, ***assumptions*)

Create a new object from an iterable.

This is a convenience function that allows one to create objects from any iterable, without having to convert to a list or tuple first.

Examples

```
>>> from sympy import Tuple
>>> Tuple.fromiter(i for i in range(5))
(0, 1, 2, 3, 4)
```

property func

The top-level function in an expression.

The following should hold for all objects:

```
>> x == x.func(*x.args)
```

Examples

```
>>> from sympy.abc import x
>>> a = 2*x
>>> a.func
<class 'sympy.core.mul.Mul'>
>>> a.args
(2, x)
>>> a.func(*a.args)
2*x
>>> a == a.func(*a.args)
True
```

has(*patterns)

Test whether any subexpression matches any of the patterns.

Examples

```
>>> from sympy import sin
>>> from sympy.abc import x, y, z
>>> (x**2 + sin(x*y)).has(z)
False
>>> (x**2 + sin(x*y)).has(x, y, z)
True
>>> x.has(x)
True
```

Note `has` is a structural algorithm with no knowledge of mathematics. Consider the following half-open interval:

```
>>> from sympy import Interval
>>> i = Interval.Lopen(0, 5); i
Interval.Lopen(0, 5)
>>> i.args
(0, 5, True, False)
>>> i.has(4) # there is no "4" in the arguments
False
>>> i.has(0) # there *is* a "0" in the arguments
True
```

Instead, use `contains` to determine whether a number is in the interval or not:

```
>>> i.contains(4)
True
>>> i.contains(0)
False
```

Note that `expr.has(*patterns)` is exactly equivalent to `any(expr.has(p) for p in patterns)`. In particular, `False` is returned when the list of patterns is empty.

```
>>> x.has()
False
```

has_free(*patterns)

return `True` if self has object(s) `x` as a free expression else `False`.

Examples

```
>>> from sympy import Integral, Function
>>> from sympy.abc import x, y
>>> f = Function('f')
>>> g = Function('g')
>>> expr = Integral(f(x), (f(x), 1, g(y)))
>>> expr.free_symbols
{y}
>>> expr.has_free(g(y))
True
>>> expr.has_free(*(x, f(x)))
False
```

This works for subexpressions and types, too:

```
>>> expr.has_free(g)
True
>>> (x + y + 1).has_free(y + 1)
True
```

property is_comparable

Return True if self can be computed to a real number (or already is a real number) with precision, else False.

Examples

```
>>> from sympy import exp_polar, pi, I
>>> (I*exp_polar(I*pi/2)).is_comparable
True
>>> (I*exp_polar(I*pi*2)).is_comparable
False
```

A False result does not mean that *self* cannot be rewritten into a form that would be comparable. For example, the difference computed below is zero but without simplification it does not evaluate to a zero with precision:

```
>>> e = 2**pi*(1 + 2**pi)
>>> dif = e - e.expand()
>>> dif.is_comparable
False
>>> dif.n(2)._prec
1
```

match(pattern, old=False)

Pattern matching.

Wild symbols match all.

Return None when expression (self) does not match with pattern. Otherwise return a dictionary such that:

```
pattern.xreplace(self.match(pattern)) == self
```

Examples

```
>>> from sympy import Wild, Sum
>>> from sympy.abc import x, y
>>> p = Wild("p")
>>> q = Wild("q")
>>> r = Wild("r")
>>> e = (x+y)**(x+y)
>>> e.match(p**p)
{p_: x + y}
>>> e.match(p**q)
{p_: x + y, q_: x + y}
>>> e = (2*x)**2
>>> e.match(p*q**r)
{p_: 4, q_: x, r_: 2}
>>> (p*q**r).xreplace(e.match(p*q**r))
4*x**2
```

Structurally bound symbols are ignored during matching:

```
>>> Sum(x, (x, 1, 2)).match(Sum(y, (y, 1, p)))
{p_: 2}
```

But they can be identified if desired:

```
>>> Sum(x, (x, 1, 2)).match(Sum(q, (q, 1, p)))
{p_: 2, q_: x}
```

The old flag will give the old-style pattern matching where expressions and patterns are essentially solved to give the match. Both of the following give None unless `old=True`:

```
>>> (x - 2).match(p - x, old=True)
{p_: 2*x - 2}
>>> (2/x).match(p*x, old=True)
{p_: 2/x**2}
```

matches(*expr*, *repl_dict*=None, *old*=False)

Helper method for `match()` that looks for a match between Wild symbols in self and expressions in *expr*.

Examples

```
>>> from sympy import symbols, Wild, Basic
>>> a, b, c = symbols('a b c')
>>> x = Wild('x')
>>> Basic(a + x, x).matches(Basic(a + b, c)) is None
True
>>> Basic(a + x, x).matches(Basic(a + b + c, b + c))
{x_: b + c}
```

`rcall(*args)`

Apply on the argument recursively through the expression tree.

This method is used to simulate a common abuse of notation for operators. For instance, in SymPy the following will not work:

```
(x+Lambda(y, 2*y))(z) == x+2*z,
```

however, you can use:

```
>>> from sympy import Lambda
>>> from sympy.abc import x, y, z
>>> (x + Lambda(y, 2*y)).rcall(z)
x + 2*z
```

`refine(assumption=True)`

See the refine function in `sympy.assumptions`

`replace(query, value, map=False, simultaneous=True, exact=None)`

Replace matching subexpressions of self with value.

If `map = True` then also return the mapping `{old: new}` where `old` was a sub-expression found with `query` and `new` is the replacement value for it. If the expression itself does not match the query, then the returned value will be `self.xreplace(map)` otherwise it should be `self.subs(ordered(map.items()))`.

Traverses an expression tree and performs replacement of matching subexpressions from the bottom to the top of the tree. The default approach is to do the replacement in a simultaneous fashion so changes made are targeted only once. If this is not desired or causes problems, `simultaneous` can be set to `False`.

In addition, if an expression containing more than one Wild symbol is being used to match subexpressions and the `exact` flag is `None` it will be set to `True` so the match will only succeed if all non-zero values are received for each Wild that appears in the match pattern. Setting this to `False` accepts a match of 0; while setting it `True` accepts all matches that have a 0 in them. See example below for cautions.

The list of possible combinations of queries and replacement values is listed below:

Examples

Initial setup

```
>>> from sympy import log, sin, cos, tan, Wild, Mul, Add
>>> from sympy.abc import x, y
>>> f = log(sin(x)) + tan(sin(x**2))
```

1.1. type -> type

`obj.replace(type, newtype)`

When object of type `type` is found, replace it with the result of passing its argument(s) to `newtype`.

```
>>> f.replace(sin, cos)
log(cos(x)) + tan(cos(x**2))
>>> sin(x).replace(sin, cos, map=True)
(cos(x), {sin(x): cos(x)})
>>> (x*y).replace(Mul, Add)
x + y
```

1.2. type -> func

`obj.replace(type, func)`

When object of type `type` is found, apply `func` to its argument(s). `func` must be written to handle the number of arguments of `type`.

```
>>> f.replace(sin, lambda arg: sin(2*arg))
log(sin(2*x)) + tan(sin(2*x**2))
>>> (x*y).replace(Mul, lambda *args: sin(2*Mul(*args)))
sin(2*x*y)
```

2.1. pattern -> expr

`obj.replace(pattern(wild), expr(wild))`

Replace subexpressions matching `pattern` with the expression written in terms of the Wild symbols in `pattern`.

```
>>> a, b = map(Wild, 'ab')
>>> f.replace(sin(a), tan(a))
log(tan(x)) + tan(tan(x**2))
>>> f.replace(sin(a), tan(a/2))
log(tan(x/2)) + tan(tan(x**2/2))
>>> f.replace(sin(a), a)
log(x) + tan(x**2)
>>> (x*y).replace(a*x, a)
y
```

Matching is exact by default when more than one Wild symbol is used: matching fails unless the match gives non-zero values for all Wild symbols:

```
>>> (2*x + y).replace(a*x + b, b - a)
y - 2
>>> (2*x).replace(a*x + b, b - a)
2*x
```

When set to False, the results may be non-intuitive:

```
>>> (2*x).replace(a*x + b, b - a, exact=False)
2/x
```

2.2. pattern -> func

`obj.replace(pattern(wild), lambda wild: expr(wild))`

All behavior is the same as in 2.1 but now a function in terms of pattern variables is used rather than an expression:

```
>>> f.replace(sin(a), lambda a: sin(2*a))
log(sin(2*x)) + tan(sin(2*x**2))
```

3.1. func -> func

`obj.replace(filter, func)`

Replace subexpression `e` with `func(e)` if `filter(e)` is True.

```
>>> g = 2*sin(x**3)
>>> g.replace(lambda expr: expr.is_Number, lambda expr: expr**2)
4*sin(x**9)
```

The expression itself is also targeted by the query but is done in such a fashion that changes are not made twice.

```
>>> e = x*(x*y + 1)
>>> e.replace(lambda x: x.is_Mul, lambda x: 2*x)
2*x*(2*x*y + 1)
```

When matching a single symbol, `exact` will default to True, but this may or may not be the behavior that is desired:

Here, we want `exact = False`:

```
>>> from sympy import Function
>>> f = Function('f')
>>> e = f(1) + f(0)
>>> q = f(a), lambda a: f(a + 1)
>>> e.replace(*q, exact=False)
f(1) + f(2)
>>> e.replace(*q, exact=True)
f(0) + f(2)
```

But here, the nature of matching makes selecting the right setting tricky:

```
>>> e = x**(1 + y)
>>> (x**(1 + y)).replace(x**(1 + a), lambda a: x**(-a), exact=False)
x
>>> (x**(1 + y)).replace(x**(1 + a), lambda a: x**(-a), exact=True)
x**(-x - y + 1)
>>> (x**y).replace(x**(1 + a), lambda a: x**(-a), exact=False)
x
>>> (x**y).replace(x**(1 + a), lambda a: x**(-a), exact=True)
x**(1 - y)
```

It is probably better to use a different form of the query that describes the target expression more precisely:

```
>>> (1 + x**(1 + y)).replace(  
... lambda x: x.is_Pow and x.exp.is_Add and x.exp.args[0] == 1,  
... lambda x: x.base**(1 - (x.exp - 1)))  
...  
x**(1 - y) + 1
```

See also:

[subs](#) (page 941)

substitution of subexpressions as defined by the objects themselves.

[xreplace](#) (page 943)

exact node replacement in expr tree; also capable of using matching rules

rewrite(*args, deep=True, **hints)

Rewrite *self* using a defined rule.

Rewriting transforms an expression to another, which is mathematically equivalent but structurally different. For example you can rewrite trigonometric functions as complex exponentials or combinatorial functions as gamma function.

This method takes a *pattern* and a *rule* as positional arguments. *pattern* is optional parameter which defines the types of expressions that will be transformed. If it is not passed, all possible expressions will be rewritten. *rule* defines how the expression will be rewritten.

Parameters

args : *rule*, or *pattern* and *rule*.

- *pattern* is a type or an iterable of types.
- *rule* can be any object.

deep : bool, optional.

If True, subexpressions are recursively transformed. Default is True.

Examples

If *pattern* is unspecified, all possible expressions are transformed.

```
>>> from sympy import cos, sin, exp, I  
>>> from sympy.abc import x  
>>> expr = cos(x) + I*sin(x)  
>>> expr.rewrite(exp)  
exp(I*x)
```

Pattern can be a type or an iterable of types.

```
>>> expr.rewrite(sin, exp)  
exp(I*x)/2 + cos(x) - exp(-I*x)/2  
>>> expr.rewrite([cos,], exp)  
exp(I*x)/2 + I*sin(x) + exp(-I*x)/2
```

(continues on next page)

(continued from previous page)

```
>>> expr.rewrite([cos, sin], exp)
exp(I*x)
```

Rewriting behavior can be implemented by defining `_eval_rewrite()` method.

```
>>> from sympy import Expr, sqrt, pi
>>> class MySin(Expr):
...     def _eval_rewrite(self, rule, args, **hints):
...         x, = args
...         if rule == cos:
...             return cos(pi/2 - x, evaluate=False)
...         if rule == sqrt:
...             return sqrt(1 - cos(x)**2)
>>> MySin(MySin(x)).rewrite(cos)
cos(-cos(-x + pi/2) + pi/2)
>>> MySin(x).rewrite(sqrt)
sqrt(1 - cos(x)**2)
```

Defining `_eval_rewrite_as_...()` method is supported for backwards compatibility reason. This may be removed in the future and using it is discouraged.

```
>>> class MySin(Expr):
...     def _eval_rewrite_as_cos(self, *args, **hints):
...         x, = args
...         return cos(pi/2 - x, evaluate=False)
>>> MySin(x).rewrite(cos)
cos(-x + pi/2)
```

simplify(kwargs)**

See the `simplify` function in `sympy.simplify`

sort_key(order=None)

Return a sort key.

Examples

```
>>> from sympy import S, I
```

```
>>> sorted([S(1)/2, I, -I], key=lambda x: x.sort_key())
[1/2, -I, I]
```

```
>>> S("[x, 1/x, 1/x**2, x**2, x**(1/2), x**(1/4), x**(3/2)]")
[x, 1/x, x**(-2), x**2, sqrt(x), x**(1/4), x**(3/2)]
>>> sorted(_, key=lambda x: x.sort_key())
[x**(-2), 1/x, x**(1/4), sqrt(x), x, x**(3/2), x**2]
```

subs(*args, **kwargs)

Substitutes old for new in an expression after sympifying args.

args is either:

- two arguments, e.g. `foo.subs(old, new)`

- **one iterable argument, e.g. `foo.subs(iterable)`. The iterable may be**
 - **an iterable container with (old, new) pairs. In this case the** replacements are processed in the order given with successive patterns possibly affecting replacements already made.
 - **a dict or set whose key/value items correspond to old/new pairs.** In this case the old/new pairs will be sorted by op count and in case of a tie, by number of args and the default `_sort_key`. The resulting sorted list is then processed as an iterable container (see previous).

If the keyword `simultaneous` is `True`, the subexpressions will not be evaluated until all the substitutions have been made.

Examples

```
>>> from sympy import pi, exp, limit, oo
>>> from sympy.abc import x, y
>>> (1 + x*y).subs(x, pi)
pi*y + 1
>>> (1 + x*y).subs({x:pi, y:2})
1 + 2*pi
>>> (1 + x*y).subs([(x, pi), (y, 2)])
1 + 2*pi
>>> reps = [(y, x**2), (x, 2)]
>>> (x + y).subs(reps)
6
>>> (x + y).subs(reversed(reps))
x**2 + 2
```

```
>>> (x**2 + x**4).subs(x**2, y)
y**2 + y
```

To replace only the `x**2` but not the `x**4`, use `xreplace`:

```
>>> (x**2 + x**4).xreplace({x**2: y})
x**4 + y
```

To delay evaluation until all substitutions have been made, set the keyword `simultaneous` to `True`:

```
>>> (x/y).subs([(x, 0), (y, 0)])
0
>>> (x/y).subs([(x, 0), (y, 0)], simultaneous=True)
nan
```

This has the added feature of not allowing subsequent substitutions to affect those already made:

```
>>> ((x + y)/y).subs({x + y: y, y: x + y})
1
>>> ((x + y)/y).subs({x + y: y, y: x + y}, simultaneous=True)
y/(x + y)
```

```
>>> from sympy import sqrt, sin, cos
>>> from sympy.abc import a, b, c, d, e
```

Parameters**rule** : dict-like

Expresses a replacement rule

Returns**xreplace** : the result of the replacement**Examples**

```
>>> from sympy import symbols, pi, exp
>>> x, y, z = symbols('x y z')
>>> (1 + x*y).xreplace({x: pi})
pi*y + 1
>>> (1 + x*y).xreplace({x: pi, y: 2})
1 + 2*pi
```

Replacements occur only if an entire node in the expression tree is matched:

```
>>> (x*y + z).xreplace({x*y: pi})
z + pi
>>> (x*y*z).xreplace({x*y: pi})
x*y*z
>>> (2*x).xreplace({2*x: y, x: z})
y
>>> (2*2*x).xreplace({2*x: y, x: z})
4*z
>>> (x + y + 2).xreplace({x + y: 2})
x + y + 2
>>> (x + 2 + exp(x + 2)).xreplace({x + 2: y})
x + exp(y) + 2
```

xreplace does not differentiate between free and bound symbols. In the following, subs(x, y) would not change x since it is a bound symbol, but xreplace does:

```
>>> from sympy import Integral
>>> Integral(x, (x, 1, 2*x)).xreplace({x: y})
Integral(y, (y, 1, 2*y))
```

Trying to replace x with an expression raises an error:

```
>>> Integral(x, (x, 1, 2*x)).xreplace({x: 2*y})
ValueError: Invalid limits given: ((2*y, 1, 4*y),)
```

See also:**replace (page 937)**

replacement capable of doing wildcard-like matching, parsing of match, and conditional replacements

subs (page 941)

substitution of subexpressions as defined by the objects themselves.

Atom

class `sympy.core.basic.Atom(*args)`

A parent class for atomic things. An atom is an expression with no subexpressions.

Examples

Symbol, Number, Rational, Integer, ... But not: Add, Mul, Pow, ...

core

singleton

S

class `sympy.core.singleton.SingletonRegistry`

The registry for the singleton classes (accessible as S).

Explanation

This class serves as two separate things.

The first thing it is is the SingletonRegistry. Several classes in SymPy appear so often that they are singletonized, that is, using some metaprogramming they are made so that they can only be instantiated once (see the [sympy.core.singleton.Singleton](#) (page 946) class for details). For instance, every time you create `Integer(0)`, this will return the same instance, [sympy.core.numbers.Zero](#) (page 995). All singleton instances are attributes of the S object, so `Integer(0)` can also be accessed as `S.Zero`.

Singletonization offers two advantages: it saves memory, and it allows fast comparison. It saves memory because no matter how many times the singletonized objects appear in expressions in memory, they all point to the same single instance in memory. The fast comparison comes from the fact that you can use `is` to compare exact instances in Python (usually, you need to use `==` to compare things). `is` compares objects by memory address, and is very fast.

Examples

```
>>> from sympy import S, Integer
>>> a = Integer(0)
>>> a is S.Zero
True
```

For the most part, the fact that certain objects are singletonized is an implementation detail that users should not need to worry about. In SymPy library code, `is` comparison is often used for performance purposes. The primary advantage of S for end users is the convenient access to certain instances that are otherwise difficult to type, like `S.Half` (instead of `Rational(1, 2)`).

When using `is` comparison, make sure the argument is sympified. For instance,

```
>>> x = 0
>>> x is S.Zero
False
```

This problem is not an issue when using `==`, which is recommended for most use-cases:

```
>>> 0 == S.Zero
True
```

The second thing `S` is is a shortcut for `sympy.core.sympify.sympify()` (page 918). `sympy.core.sympify.sympify()` (page 918) is the function that converts Python objects such as `int(1)` into SymPy objects such as `Integer(1)`. It also converts the string form of an expression into a SymPy expression, like `sympify("x**2") -> Symbol("x")**2`. `S(1)` is the same thing as `sympify(1)` (basically, `S.__call__` has been defined to call `sympify`).

This is for convenience, since `S` is a single letter. It's mostly useful for defining rational numbers. Consider an expression like $x + 1/2$. If you enter this directly in Python, it will evaluate the $1/2$ and give `0.5`, because both arguments are ints (see also *Two Final Notes: ^ and /* (page 13)). However, in SymPy, you usually want the quotient of two integers to give an exact rational number. The way Python's evaluation works, at least one side of an operator needs to be a SymPy object for the SymPy evaluation to take over. You could write this as `x + Rational(1, 2)`, but this is a lot more typing. A shorter version is `x + S(1)/2`. Since `S(1)` returns `Integer(1)`, the division will return a `Rational` type, since it will call `Integer.__truediv__`, which knows how to return a `Rational`.

```
class sympy.core.singleton.Singleton(*args, **kwargs)
    Metaclass for singleton classes.
```

Explanation

A singleton class has only one instance which is returned every time the class is instantiated. Additionally, this instance can be accessed through the global registry object `S` as `S.<class_name>`.

Examples

```
>>> from sympy import S, Basic
>>> from sympy.core.singleton import Singleton
>>> class MySingleton(Basic, metaclass=Singleton):
...     pass
>>> Basic() is Basic()
False
>>> MySingleton() is MySingleton()
True
>>> S.MySingleton is MySingleton()
True
```

Notes

Instance creation is delayed until the first time the value is accessed. (SymPy versions before 1.0 would create the instance during class creation time, which would be prone to import cycles.)

This metaclass is a subclass of `ManagedProperties` because that is the metaclass of many classes that need to be Singletons (Python does not allow subclasses to have a different metaclass than the superclass, except the subclass may use a subclassed metaclass).

`expr`

`Expr`

class `sympy.core.expr.Expr(*args)`

Base class for algebraic expressions.

Explanation

Everything that requires arithmetic operations to be defined should subclass this class, instead of `Basic` (which should be used only for argument storage and expression manipulation, i.e. pattern matching, substitutions, etc).

If you want to override the comparisons of expressions: Should use `_eval_is_ge` for inequality, or `_eval_is_eq`, with multiple dispatch. `_eval_is_ge` return true if $x \geq y$, false if $x < y$, and `None` if the two types are not comparable or the comparison is indeterminate

See also:

[`sympy.core.basic.Basic`](#) (page 927)

apart(*x=None, **args*)

See the `apart` function in `sympy.polys`

args_cnc(*cset=False, warn=True, split_1=True*)

Return [commutative factors, non-commutative factors] of self.

Explanation

self is treated as a `Mul` and the ordering of the factors is maintained. If `cset` is `True` the commutative factors will be returned in a set. If there were repeated factors (as may happen with an unevaluated `Mul`) then an error will be raised unless it is explicitly suppressed by setting `warn` to `False`.

Note: -1 is always separated from a `Number` unless `split_1` is `False`.

Examples

```
>>> from sympy import symbols, oo
>>> A, B = symbols('A B', commutative=0)
>>> x, y = symbols('x y')
>>> (-2*x*y).args_cnc()
[[-1, 2, x, y], []]
>>> (-2.5*x).args_cnc()
[[-1, 2.5, x], []]
>>> (-2*x*A*B*y).args_cnc()
[[-1, 2, x, y], [A, B]]
>>> (-2*x*A*B*y).args_cnc(split_l=False)
[[-2, x, y], [A, B]]
>>> (-2*x*y).args_cnc(cset=True)
[{-1, 2, x, y}, []]
```

The arg is always treated as a Mul:

```
>>> (-2 + x + A).args_cnc()
[[], [x - 2 + A]]
>>> (-oo).args_cnc() # -oo is a singleton
[[-1, oo], []]
```

as_coeff_Add(*rational=False*) → tuple['Number', [sympy.core.expr.Expr](#) (page 947)]

Efficiently extract the coefficient of a summation.

as_coeff_Mul(*rational: bool = False*) → tuple['Number', [sympy.core.expr.Expr](#) (page 947)]

Efficiently extract the coefficient of a product.

as_coeff_add(**deps*) → tuple[[sympy.core.expr.Expr](#) (page 947),
tuple[[sympy.core.expr.Expr](#) (page 947), ...]]

Return the tuple (c, args) where self is written as an Add, a.

c should be a Rational added to any terms of the Add that are independent of deps.

args should be a tuple of all other terms of a; args is empty if self is a Number or if self is independent of deps (when given).

This should be used when you do not know if self is an Add or not but you want to treat self as an Add or if you want to process the individual arguments of the tail of self as an Add.

- if you know self is an Add and want only the head, use self.args[0];
- if you do not want to process the arguments of the tail but need the tail then use self.as_two_terms() which gives the head and tail.
- if you want to split self into an independent and dependent parts use self.as_independent(*deps)

```
>>> from sympy import S
>>> from sympy.abc import x, y
>>> (S(3)).as_coeff_add()
(3, ())
>>> (3 + x).as_coeff_add()
```

(continues on next page)

(continued from previous page)

```
(3, (x,))
>>> (3 + x + y).as_coeff_add(x)
(y + 3, (x,))
>>> (3 + y).as_coeff_add(x)
(y + 3, ())
```

as_coeff_exponent(*x*) → tuple[[sympy.core.expr.Expr](#) (page 947),
[sympy.core.expr.Expr](#) (page 947)]

$c*x**e \rightarrow c, e$ where *x* can be any symbolic expression.

as_coeff_mul(**deps*, ***kwargs*) → tuple[[sympy.core.expr.Expr](#) (page 947),
tuple[[sympy.core.expr.Expr](#) (page 947), ...]]

Return the tuple (*c*, *args*) where *self* is written as a Mul, *m*.

c should be a Rational multiplied by any factors of the Mul that are independent of *deps*.

args should be a tuple of all other factors of *m*; *args* is empty if *self* is a Number or if *self* is independent of *deps* (when given).

This should be used when you do not know if *self* is a Mul or not but you want to treat *self* as a Mul or if you want to process the individual arguments of the tail of *self* as a Mul.

- if you know *self* is a Mul and want only the head, use *self*.args[0];
- if you do not want to process the arguments of the tail but need the tail then use *self*.as_two_terms() which gives the head and tail;
- if you want to split *self* into an independent and dependent parts use *self*.as_independent(**deps*)

```
>>> from sympy import S
>>> from sympy.abc import x, y
>>> (S(3)).as_coeff_mul()
(3, ())
>>> (3*x*y).as_coeff_mul()
(3, (x, y))
>>> (3*x*y).as_coeff_mul(x)
(3*y, (x,))
>>> (3*y).as_coeff_mul(x)
(3*y, ())
```

as_coefficient(*expr*)

Extracts symbolic coefficient at the given expression. In other words, this functions separates 'self' into the product of 'expr' and 'expr'-free coefficient. If such separation is not possible it will return None.

Examples

```
>>> from sympy import E, pi, sin, I, Poly
>>> from sympy.abc import x
```

```
>>> E.as_coefficient(E)
1
>>> (2*E).as_coefficient(E)
2
>>> (2*sin(E)*E).as_coefficient(E)
```

Two terms have E in them so a sum is returned. (If one were desiring the coefficient of the term exactly matching E then the constant from the returned expression could be selected. Or, for greater precision, a method of Poly can be used to indicate the desired term from which the coefficient is desired.)

```
>>> (2*E + x*E).as_coefficient(E)
x + 2
>>> _.args[0] # just want the exact match
2
>>> p = Poly(2*E + x*E); p
Poly(x*E + 2*E, x, E, domain='ZZ')
>>> p.coeff_monomial(E)
2
>>> p.nth(0, 1)
2
```

Since the following cannot be written as a product containing E as a factor, None is returned. (If the coefficient 2*x is desired then the `coeff` method should be used.)

```
>>> (2*E*x + x).as_coefficient(E)
>>> (2*E*x + x).coeff(E)
2*x
```

```
>>> (E*(x + 1) + x).as_coefficient(E)
```

```
>>> (2*pi*I).as_coefficient(pi*I)
2
>>> (2*I).as_coefficient(pi*I)
```

See also:

`coeff` (page 958)

return sum of terms have a given factor

`as_coeff_Add` (page 948)

separate the additive constant from an expression

`as_coeff_Mul` (page 948)

separate the multiplicative constant from an expression

`as_independent` (page 952)

separate x-dependent terms/factors from others

`sympy.polys.polytools.Poly.coeff_monomial` (page 2384)

efficiently find the single coefficient of a monomial in Poly

`sympy.polys.polytools.Poly.nth` (page 2408)

like `coeff_monomial` but powers of monomial terms are used

`as_coefficients_dict(*syms)`

Return a dictionary mapping terms to their Rational coefficient. Since the dictionary is a defaultdict, inquiries about terms which were not present will return a coefficient of 0.

If symbols `syms` are provided, any multiplicative terms independent of them will be considered a coefficient and a regular dictionary of `syms`-dependent generators as keys and their corresponding coefficients as values will be returned.

Examples

```
>>> from sympy.abc import a, x, y
>>> (3*x + a*x + 4).as_coefficients_dict()
{1: 4, x: 3, a*x: 1}
>>> _[a]
0
>>> (3*a*x).as_coefficients_dict()
{a*x: 3}
>>> (3*a*x).as_coefficients_dict(x)
{x: 3*a}
>>> (3*a*x).as_coefficients_dict(y)
{1: 3*a*x}
```

`as_content_primitive(radical=False, clear=True)`

This method should recursively remove a Rational from all arguments and return that (content) and the new self (primitive). The content should always be positive and `Mul(*foo.as_content_primitive()) == foo`. The primitive need not be in canonical form and should try to preserve the underlying structure if possible (i.e. `expand_mul` should not be applied to self).

Examples

```
>>> from sympy import sqrt
>>> from sympy.abc import x, y, z
```

```
>>> eq = 2 + 2*x + 2*y*(3 + 3*y)
```

The `as_content_primitive` function is recursive and retains structure:

```
>>> eq.as_content_primitive()
(2, x + 3*y*(y + 1) + 1)
```

Integer powers will have Rationals extracted from the base:

```
>>> ((2 + 6*x)**2).as_content_primitive()
(4, (3*x + 1)**2)
>>> ((2 + 6*x)**(2*y)).as_content_primitive()
(1, (2*(3*x + 1))**(2*y))
```

Terms may end up joining once their `as_content_primitives` are added:

```
>>> ((5*(x*(1 + y)) + 2*x*(3 + 3*y))).as_content_primitive()
(11, x*(y + 1))
>>> ((3*(x*(1 + y)) + 2*x*(3 + 3*y))).as_content_primitive()
(9, x*(y + 1))
>>> ((3*(z*(1 + y)) + 2.0*x*(3 + 3*y))).as_content_primitive()
(1, 6.0*x*(y + 1) + 3*z*(y + 1))
>>> ((5*(x*(1 + y)) + 2*x*(3 + 3*y))**2).as_content_primitive()
(121, x**2*(y + 1)**2)
>>> ((x*(1 + y) + 0.4*x*(3 + 3*y))**2).as_content_primitive()
(1, 4.84*x**2*(y + 1)**2)
```

Radical content can also be factored out of the primitive:

```
>>> (2*sqrt(2) + 4*sqrt(10)).as_content_primitive(radical=True)
(2, sqrt(2)*(1 + 2*sqrt(5)))
```

If `clear=False` (default is `True`) then content will not be removed from an `Add` if it can be distributed to leave one or more terms with integer coefficients.

```
>>> (x/2 + y).as_content_primitive()
(1/2, x + 2*y)
>>> (x/2 + y).as_content_primitive(clear=False)
(1, x/2 + y)
```

`as_expr(*gens)`

Convert a polynomial to a SymPy expression.

Examples

```
>>> from sympy import sin
>>> from sympy.abc import x, y
```

```
>>> f = (x**2 + x*y).as_poly(x, y)
>>> f.as_expr()
x**2 + x*y
```

```
>>> sin(x).as_expr()
sin(x)
```

`as_independent(*deps, **hint)` → tuple[[sympy.core.expr.Expr](#) (page 947),
[sympy.core.expr.Expr](#) (page 947)]

A mostly naive separation of a `Mul` or `Add` into arguments that are not dependent on `deps`. To obtain as complete a separation of variables as possible, use a separation method first, e.g.:

- `separatevars()` to change Mul, Add and Pow (including exp) into Mul
- `.expand(mul=True)` to change Add or Mul into Add
- `.expand(log=True)` to change log expr into an Add

The only non-naive thing that is done here is to respect noncommutative ordering of variables and to always return (0, 0) for *self* of zero regardless of hints.

For nonzero *self*, the returned tuple (i, d) has the following interpretation:

- i will have no variable that appears in deps
- d will either have terms that contain variables that are in deps, or be equal to 0 (when *self* is an Add) or 1 (when *self* is a Mul)
- if *self* is an Add then $\text{self} = i + d$
- if *self* is a Mul then $\text{self} = i*d$
- otherwise (self, S.One) or (S.One, self) is returned.

To force the expression to be treated as an Add, use the hint `as_Add=True`

Examples

- self is an Add

```
>>> from sympy import sin, cos, exp
>>> from sympy.abc import x, y, z
```

```
>>> (x + x*y).as_independent(x)
(0, x*y + x)
>>> (x + x*y).as_independent(y)
(x, x*y)
>>> (2*x*sin(x) + y + x + z).as_independent(x)
(y + z, 2*x*sin(x) + x)
>>> (2*x*sin(x) + y + x + z).as_independent(x, y)
(z, 2*x*sin(x) + x + y)
```

- self is a Mul

```
>>> (x*sin(x)*cos(y)).as_independent(x)
(cos(y), x*sin(x))
```

non-commutative terms cannot always be separated out when *self* is a Mul

```
>>> from sympy import symbols
>>> n1, n2, n3 = symbols('n1 n2 n3', commutative=False)
>>> (n1 + n1*n2).as_independent(n2)
(n1, n1*n2)
>>> (n2*n1 + n1*n2).as_independent(n2)
(0, n1*n2 + n2*n1)
>>> (n1*n2*n3).as_independent(n1)
(1, n1*n2*n3)
>>> (n1*n2*n3).as_independent(n2)
(n1, n2*n3)
```

(continues on next page)

(continued from previous page)

```
>>> ((x-n1)*(x-y)).as_independent(x)
(1, (x - y)*(x - n1))
```

- self is anything else:

```
>>> (sin(x)).as_independent(x)
(1, sin(x))
>>> (sin(x)).as_independent(y)
(sin(x), 1)
>>> exp(x+y).as_independent(x)
(1, exp(x + y))
```

- force self to be treated as an Add:

```
>>> (3*x).as_independent(x, as_Add=True)
(0, 3*x)
```

- force self to be treated as a Mul:

```
>>> (3+x).as_independent(x, as_Add=False)
(1, x + 3)
>>> (-3+x).as_independent(x, as_Add=False)
(1, x - 3)
```

Note how the below differs from the above in making the constant on the dep term positive.

```
>>> (y*(-3+x)).as_independent(x)
(y, x - 3)
```

- use `.as_independent()` for true independence testing instead

of `.has()`. The former considers only symbols in the free symbols while the latter considers all symbols

```
>>> from sympy import Integral
>>> I = Integral(x, (x, 1, 2))
>>> I.has(x)
True
>>> x in I.free_symbols
False
>>> I.as_independent(x) == (I, 1)
True
>>> (I + x).as_independent(x) == (I, x)
True
```

Note: when trying to get independent terms, a separation method might need to be used first. In this case, it is important to keep track of what you send to this routine so you know how to interpret the returned values

```
>>> from sympy import separatevars, log
>>> separatevars(exp(x+y)).as_independent(x)
(exp(y), exp(x))
```

(continues on next page)

(continued from previous page)

```

>>> (x + x*y).as_independent(y)
(x, x*y)
>>> separatevars(x + x*y).as_independent(y)
(x, y + 1)
>>> (x*(1 + y)).as_independent(y)
(x, y + 1)
>>> (x*(1 + y)).expand(mul=True).as_independent(y)
(x, x*y)
>>> a, b=symbols('a b', positive=True)
>>> (log(a*b).expand(log=True)).as_independent(b)
(log(a), log(b))

```

See also:

[separatevars](#) (page 664), [expand](#) (page 962), [sympy.core.add.Add.as_two_terms](#) (page 1016), [sympy.core.mul.Mul.as_two_terms](#) (page 1011), [as_coeff_add](#) (page 948), [as_coeff_mul](#) (page 949)

as_leading_term(*symbols, logx=None, cdir=0)

Returns the leading (nonzero) term of the series expansion of self.

The `_eval_as_leading_term` routines are used to do this, and they must always return a non-zero value.

Examples

```

>>> from sympy.abc import x
>>> (1 + x + x**2).as_leading_term(x)
1
>>> (1/x**2 + x + x**2).as_leading_term(x)
x**(-2)

```

as_numer_denom()

expression -> a/b -> a, b

This is just a stub that should be defined by an object's class methods to get anything else.

See also:

[normal](#) (page 971)

return a/b instead of (a, b)

as_ordered_factors(order=None)

Return list of ordered factors (if Mul) else [self].

as_ordered_terms(order=None, data=False)

Transform an expression to an ordered list of terms.

Examples

```
>>> from sympy import sin, cos
>>> from sympy.abc import x
```

```
>>> (sin(x)**2*cos(x) + sin(x)**2 + 1).as_ordered_terms()
[sin(x)**2*cos(x), sin(x)**2, 1]
```

as_poly(*gens, **args)

Converts self to a polynomial or returns None.

Explanation

```
>>> from sympy import sin
>>> from sympy.abc import x, y
```

```
>>> print((x**2 + x*y).as_poly())
Poly(x**2 + x*y, x, y, domain='ZZ')
```

```
>>> print((x**2 + x*y).as_poly(x, y))
Poly(x**2 + x*y, x, y, domain='ZZ')
```

```
>>> print((x**2 + sin(y)).as_poly(x, y))
None
```

as_powers_dict()

Return self as a dictionary of factors with each factor being treated as a power. The keys are the bases of the factors and the values, the corresponding exponents. The resulting dictionary should be used with caution if the expression is a Mul and contains non-commutative factors since the order that they appeared will be lost in the dictionary.

See also:

***as_ordered_factors* (page 955)**

An alternative for noncommutative applications, returning an ordered list of factors.

***args_cnc* (page 947)**

Similar to `as_ordered_factors`, but guarantees separation of commutative and noncommutative factors.

as_real_imag(deep=True, **hints)

Performs complex expansion on 'self' and returns a tuple containing collected both real and imaginary parts. This method cannot be confused with `re()` and `im()` functions, which does not perform complex expansion at evaluation.

However it is possible to expand both `re()` and `im()` functions and get exactly the same results as with a single call to this function.

```
>>> from sympy import symbols, I
```

```
>>> x, y = symbols('x,y', real=True)
```

```
>>> (x + y*I).as_real_imag()
(x, y)
```

```
>>> from sympy.abc import z, w
```

```
>>> (z + w*I).as_real_imag()
(re(z) - im(w), re(w) + im(z))
```

as_terms()

Transform an expression to a list of terms.

aseries(x=None, n=6, bound=0, hir=False)

Asymptotic Series expansion of self. This is equivalent to `self.series(x, oo, n)`.

Parameters

self : Expression

The expression whose series is to be expanded.

x : Symbol

It is the variable of the expression to be calculated.

n : Value

The value used to represent the order in terms of x^{**n} , up to which the series is to be expanded.

hir : Boolean

Set this parameter to be True to produce hierarchical series. It stops the recursion at an early level and may provide nicer and more useful results.

bound : Value, Integer

Use the bound parameter to give limit on rewriting coefficients in its normalised form.

Returns

Expr

Asymptotic series expansion of the expression.

Examples

```
>>> from sympy import sin, exp
>>> from sympy.abc import x
```

```
>>> e = sin(1/x + exp(-x)) - sin(1/x)
```

```
>>> e.aseries(x)
(1/(24*x**4) - 1/(2*x**2) + 1 + O(x**(-6), (x, oo)))*exp(-x)
```

```
>>> e.aseries(x, n=3, hir=True)
-exp(-2*x)*sin(1/x)/2 + exp(-x)*cos(1/x) + 0(exp(-3*x), (x, oo))
```

```
>>> e = exp(exp(x)/(1 - 1/x))
```

```
>>> e.aseries(x)
exp(exp(x)/(1 - 1/x))
```

```
>>> e.aseries(x, bound=3)
exp(exp(x)/x**2)*exp(exp(x)/x)*exp(-exp(x)) + exp(x)/(1 - 1/x) -
exp(x)/x - exp(x)/x**2)*exp(exp(x))
```

For rational expressions this method may return original expression without the Order term. `>>> (1/x).aseries(x, n=8) 1/x`

Notes

This algorithm is directly induced from the limit computational algorithm provided by Gruntz. It majorly uses the `mrsv` and `rewrite` sub-routines. The overall idea of this algorithm is first to look for the most rapidly varying subexpression w of a given expression f and then expands f in a series in w . Then same thing is recursively done on the leading coefficient till we get constant coefficients.

If the most rapidly varying subexpression of a given expression f is f itself, the algorithm tries to find a normalised representation of the `mrsv` set and rewrites f using this normalised representation.

If the expansion contains an order term, it will be either $O(x^{(-n)})$ or $O(w^{(-n)})$ where w belongs to the most rapidly varying expression of `self`.

See also:

[Expr.aseries](#) (page 957)

See the docstring of this function for complete details of this wrapper.

References

[R108], [R109], [R110]

cancel(**gens*, ***args*)

See the `cancel` function in `sympy.polys`

coeff(*x*, *n=1*, *right=False*, *_first=True*)

Returns the coefficient from the term(s) containing x^n . If n is zero then all terms independent of x will be returned.

Explanation

When x is noncommutative, the coefficient to the left (default) or right of x can be returned. The keyword 'right' is ignored when x is commutative.

Examples

```
>>> from sympy import symbols
>>> from sympy.abc import x, y, z
```

You can select terms that have an explicit negative in front of them:

```
>>> (-x + 2*y).coeff(-1)
x
>>> (x - 2*y).coeff(-1)
2*y
```

You can select terms with no Rational coefficient:

```
>>> (x + 2*y).coeff(1)
x
>>> (3 + 2*x + 4*x**2).coeff(1)
0
```

You can select terms independent of x by making $n=0$; in this case `expr.as_independent(x)[0]` is returned (and 0 will be returned instead of None):

```
>>> (3 + 2*x + 4*x**2).coeff(x, 0)
3
>>> eq = ((x + 1)**3).expand() + 1
>>> eq
x**3 + 3*x**2 + 3*x + 2
>>> [eq.coeff(x, i) for i in reversed(range(4))]
[1, 3, 3, 2]
>>> eq -= 2
>>> [eq.coeff(x, i) for i in reversed(range(4))]
[1, 3, 3, 0]
```

You can select terms that have a numerical term in front of them:

```
>>> (-x - 2*y).coeff(2)
-y
>>> from sympy import sqrt
>>> (x + sqrt(2)*x).coeff(sqrt(2))
x
```

The matching is exact:

```
>>> (3 + 2*x + 4*x**2).coeff(x)
2
>>> (3 + 2*x + 4*x**2).coeff(x**2)
4
>>> (3 + 2*x + 4*x**2).coeff(x**3)
```

(continues on next page)

(continued from previous page)

```
0
>>> (z*(x + y)**2).coeff((x + y)**2)
z
>>> (z*(x + y)**2).coeff(x + y)
0
```

In addition, no factoring is done, so $1 + z*(1 + y)$ is not obtained from the following:

```
>>> (x + z*(x + x*y)).coeff(x)
1
```

If such factoring is desired, `factor_terms` can be used first:

```
>>> from sympy import factor_terms
>>> factor_terms(x + z*(x + x*y)).coeff(x)
z*(y + 1) + 1
```

```
>>> n, m, o = symbols('n m o', commutative=False)
>>> n.coeff(n)
1
>>> (3*n).coeff(n)
3
>>> (n*m + m*n*m).coeff(n) # = (1 + m)*n*m
1 + m
>>> (n*m + m*n*m).coeff(n, right=True) # = (1 + m)*n*m
m
```

If there is more than one possible coefficient 0 is returned:

```
>>> (n*m + m*n).coeff(n)
0
```

If there is only one possible coefficient, it is returned:

```
>>> (n*m + x*m*n).coeff(m*n)
x
>>> (n*m + x*m*n).coeff(m*n, right=1)
1
```

See also:

[**`as\_coefficient`**](#) (page 949)

separate the expression into a coefficient and factor

[**`as\_coeff\_Add`**](#) (page 948)

separate the additive constant from an expression

[**`as\_coeff\_Mul`**](#) (page 948)

separate the multiplicative constant from an expression

[**`as\_independent`**](#) (page 952)

separate x-dependent terms/factors from others

[**`sympy.polys.polytools.Poly.coeff\_monomial`**](#) (page 2384)

efficiently find the single coefficient of a monomial in Poly

***sympy.polys.polytools.Poly.nth* (page 2408)**

like `coeff_monomial` but powers of monomial terms are used

collect(*syms, func=None, evaluate=True, exact=False, distribute_order_term=True*)

See the `collect` function in `sympy.simplify`

combsimp()

See the `combsimp` function in `sympy.simplify`

compute_leading_term(*x, logx=None*)

`as_leading_term` is only allowed for results of `.series()` This is a wrapper to compute a series first.

conjugate()

Returns the complex conjugate of 'self'.

could_extract_minus_sign()

Return True if self has -1 as a leading factor or has more literal negative signs than positive signs in a sum, otherwise False.

Examples

```
>>> from sympy.abc import x, y
>>> e = x - y
>>> {i.could_extract_minus_sign() for i in (e, -e)}
{False, True}
```

Though the $y - x$ is considered like $-(x - y)$, since it is in a product without a leading factor of -1, the result is false below:

```
>>> (x*(y - x)).could_extract_minus_sign()
False
```

To put something in canonical form wrt to sign, use *signsimp*:

```
>>> from sympy import signsimp
>>> signsimp(x*(y - x))
-x*(x - y)
>>> _.could_extract_minus_sign()
True
```

count_ops(*visual=None*)

wrapper for `count_ops` that returns the operation count.

equals(*other, failing_expression=False*)

Return True if `self == other`, False if it does not, or None. If `failing_expression` is True then the expression which did not simplify to a 0 will be returned instead of None.

Explanation

If self is a Number (or complex number) that is not zero, then the result is False.

If self is a number and has not evaluated to zero, evalf will be used to test whether the expression evaluates to zero. If it does so and the result has significance (i.e. the precision is either -1, for a Rational result, or is greater than 1) then the evalf value will be used to return True or False.

expand(*deep=True, modulus=None, power_base=True, power_exp=True, mul=True, log=True, multinomial=True, basic=True, **hints*)

Expand an expression using hints.

See the docstring of the expand() function in sympy.core.function for more information.

property expr_free_symbols

Like free_symbols, but returns the free symbols only if they are contained in an expression node.

Examples

```
>>> from sympy.abc import x, y
>>> (x + y).expr_free_symbols
{x, y}
```

If the expression is contained in a non-expression object, do not return the free symbols. Compare:

```
>>> from sympy import Tuple
>>> t = Tuple(x + y)
>>> t.expr_free_symbols
set()
>>> t.free_symbols
{x, y}
```

extract_additively(c)

Return self - c if it's possible to subtract c from self and make all matching coefficients move towards zero, else return None.

Examples

```
>>> from sympy.abc import x, y
>>> e = 2*x + 3
>>> e.extract_additively(x + 1)
x + 2
>>> e.extract_additively(3*x)
None
>>> e.extract_additively(4)
None
>>> (y*(x + 1)).extract_additively(x + 1)
y
>>> ((x + 1)*(x + 2*y + 1) + 3).extract_additively(x + 1)
(x + 1)*(x + 2*y) + 3
```

See also:

[`extract\_multiplicatively`](#) (page 963), [`coeff`](#) (page 958), [`as\_coefficient`](#) (page 949)

extract_branch_factor(*allow_half=False*)

Try to write self as $\exp_{\text{polar}}(2\pi i n)z$ in a nice way. Return (z, n).

```
>>> from sympy import exp_polar, I, pi
>>> from sympy.abc import x, y
>>> exp_polar(I*pi).extract_branch_factor()
(exp_polar(I*pi), 0)
>>> exp_polar(2*I*pi).extract_branch_factor()
(1, 1)
>>> exp_polar(-pi*I).extract_branch_factor()
(exp_polar(I*pi), -1)
>>> exp_polar(3*pi*I + x).extract_branch_factor()
(exp_polar(x + I*pi), 1)
>>> (y*exp_polar(-5*pi*I)*exp_polar(3*pi*I + 2*pi*x)).extract_branch_
    factor()
(y*exp_polar(2*pi*x), -1)
>>> exp_polar(-I*pi/2).extract_branch_factor()
(exp_polar(-I*pi/2), 0)
```

If *allow_half* is True, also extract $\exp_{\text{polar}}(I\pi)$:

```
>>> exp_polar(I*pi).extract_branch_factor(allow_half=True)
(1, 1/2)
>>> exp_polar(2*I*pi).extract_branch_factor(allow_half=True)
(1, 1)
>>> exp_polar(3*I*pi).extract_branch_factor(allow_half=True)
(1, 3/2)
>>> exp_polar(-I*pi).extract_branch_factor(allow_half=True)
(1, -1/2)
```

extract_multiplicatively(*c*)

Return None if it's not possible to make self in the form $c * \text{something}$ in a nice way, i.e. preserving the properties of arguments of self.

Examples

```
>>> from sympy import symbols, Rational
```

```
>>> x, y = symbols('x,y', real=True)
```

```
>>> ((x*y)**3).extract_multiplicatively(x**2 * y)
x*y**2
```

```
>>> ((x*y)**3).extract_multiplicatively(x**4 * y)
```

```
>>> (2*x).extract_multiplicatively(2)
x
```

```
>>> (2*x).extract_multiplicatively(3)
```

```
>>> (Rational(1, 2)*x).extract_multiplicatively(3)
x/6
```

factor(*gens, **args)

See the `factor()` function in `sympy.polys.polytools`

fourier_series(limits=None)

Compute fourier sine/cosine series of self.

See the docstring of the `fourier_series()` (page 964) in `sympy.series.fourier` for more information.

fps(x=None, x0=0, dir=1, hyper=True, order=4, rational=True, full=False)

Compute formal power power series of self.

See the docstring of the `fps()` (page 964) function in `sympy.series.formal` for more information.

gammasimp()

See the `gammasimp` function in `sympy.simplify`

get0()

Returns the additive $O(\cdot)$ symbol if there is one, else `None`.

getn()

Returns the order of the expression.

Explanation

The order is determined either from the $O(\dots)$ term. If there is no $O(\dots)$ term, it returns `None`.

Examples

```
>>> from sympy import O
>>> from sympy.abc import x
>>> (1 + x + O(x**2)).getn()
2
>>> (1 + x).getn()
```

integrate(*args, **kwargs)

See the `integrate` function in `sympy.integrals`

invert(g, *gens, **args)

Return the multiplicative inverse of `self` mod `g` where `self` (and `g`) may be symbolic expressions).

See also:

`sympy.core.numbers.mod_inverse` (page 1004), `sympy.polys.polytools.invert` (page 2365)

is_algebraic_expr(*syms)

This tests whether a given expression is algebraic or not, in the given symbols, syms. When syms is not given, all free symbols will be used. The rational function does not have to be in expanded or in any kind of canonical form.

This function returns False for expressions that are “algebraic expressions” with symbolic exponents. This is a simple extension to the `is_rational_function`, including rational exponentiation.

Examples

```
>>> from sympy import Symbol, sqrt
>>> x = Symbol('x', real=True)
>>> sqrt(1 + x).is_rational_function()
False
>>> sqrt(1 + x).is_algebraic_expr()
True
```

This function does not attempt any nontrivial simplifications that may result in an expression that does not appear to be an algebraic expression to become one.

```
>>> from sympy import exp, factor
>>> a = sqrt(exp(x)**2 + 2*exp(x) + 1)/(exp(x) + 1)
>>> a.is_algebraic_expr(x)
False
>>> factor(a).is_algebraic_expr()
True
```

See also:

[*is_rational_function*](#) (page 969)

References

[R111]

is_constant(*wrt, **flags)

Return True if self is constant, False if not, or None if the constancy could not be determined conclusively.

Explanation

If an expression has no free symbols then it is a constant. If there are free symbols it is possible that the expression is a constant, perhaps (but not necessarily) zero. To test such expressions, a few strategies are tried:

1) numerical evaluation at two random points. If two such evaluations give two different values and the values have a precision greater than 1 then self is not constant. If the evaluations agree or could not be obtained with any precision, no decision is made. The numerical testing is done only if `wrt` is different than the free symbols.

2) differentiation with respect to variables in `wrt` (or all free symbols if omitted) to see if the expression is constant or not. This will not always lead to an expression

that is zero even though an expression is constant (see added test in test_expr.py). If all derivatives are zero then self is constant with respect to the given symbols.

3) finding out zeros of denominator expression with free_symbols. It will not be constant if there are zeros. It gives more negative answers for expression that are not constant.

If neither evaluation nor differentiation can prove the expression is constant, None is returned unless two numerical values happened to be the same and the flag failing_number is True - in that case the numerical value will be returned.

If flag simplify=False is passed, self will not be simplified; the default is True since self should be simplified before testing.

Examples

```
>>> from sympy import cos, sin, Sum, S, pi
>>> from sympy.abc import a, n, x, y
>>> x.is_constant()
False
>>> S(2).is_constant()
True
>>> Sum(x, (x, 1, 10)).is_constant()
True
>>> Sum(x, (x, 1, n)).is_constant()
False
>>> Sum(x, (x, 1, n)).is_constant(y)
True
>>> Sum(x, (x, 1, n)).is_constant(n)
False
>>> Sum(x, (x, 1, n)).is_constant(x)
True
>>> eq = a*cos(x)**2 + a*sin(x)**2 - a
>>> eq.is_constant()
True
>>> eq.subs({x: pi, a: 2}) == eq.subs({x: pi, a: 3}) == 0
True
```

```
>>> (0**x).is_constant()
False
>>> x.is_constant()
False
>>> (x**x).is_constant()
False
>>> one = cos(x)**2 + sin(x)**2
>>> one.is_constant()
True
>>> ((one - 1)**(x + 1)).is_constant() in (True, False) # could be 0
↳ or 1
True
```

is_meromorphic(x, a)

This tests whether an expression is meromorphic as a function of the given symbol x at the point a.

This method is intended as a quick test that will return None if no decision can be made without simplification or more detailed analysis.

Examples

```
>>> from sympy import zoo, log, sin, sqrt
>>> from sympy.abc import x
```

```
>>> f = 1/x**2 + 1 - 2*x**3
>>> f.is_meromorphic(x, 0)
True
>>> f.is_meromorphic(x, 1)
True
>>> f.is_meromorphic(x, zoo)
True
```

```
>>> g = x**log(3)
>>> g.is_meromorphic(x, 0)
False
>>> g.is_meromorphic(x, 1)
True
>>> g.is_meromorphic(x, zoo)
False
```

```
>>> h = sin(1/x)*x**2
>>> h.is_meromorphic(x, 0)
False
>>> h.is_meromorphic(x, 1)
True
>>> h.is_meromorphic(x, zoo)
True
```

Multivalued functions are considered meromorphic when their branches are meromorphic. Thus most functions are meromorphic everywhere except at essential singularities and branch points. In particular, they will be meromorphic also on branch cuts except at their endpoints.

```
>>> log(x).is_meromorphic(x, -1)
True
>>> log(x).is_meromorphic(x, 0)
False
>>> sqrt(x).is_meromorphic(x, -1)
True
>>> sqrt(x).is_meromorphic(x, 0)
False
```

property `is_number`

Returns True if self has no free symbols and no undefined functions (AppliedUndef, to be precise). It will be faster than `if not self.free_symbols`, however, since `is_number` will fail as soon as it hits a free symbol or undefined function.

Examples

```
>>> from sympy import Function, Integral, cos, sin, pi
>>> from sympy.abc import x
>>> f = Function('f')
```

```
>>> x.is_number
False
>>> f(1).is_number
False
>>> (2*x).is_number
False
>>> (2 + Integral(2, x)).is_number
False
>>> (2 + Integral(2, (x, 1, 2))).is_number
True
```

Not all numbers are Numbers in the SymPy sense:

```
>>> pi.is_number, pi.is_Number
(True, False)
```

If something is a number it should evaluate to a number with real and imaginary parts that are Numbers; the result may not be comparable, however, since the real and/or imaginary part of the result may not have precision.

```
>>> cos(1).is_number and cos(1).is_comparable
True
```

```
>>> z = cos(1)**2 + sin(1)**2 - 1
>>> z.is_number
True
>>> z.is_comparable
False
```

See also:

[*sympy.core.basic.Basic.is_comparable*](#) (page 935)

is_polynomial(*syms)

Return True if self is a polynomial in syms and False otherwise.

This checks if self is an exact polynomial in syms. This function returns False for expressions that are “polynomials” with symbolic exponents. Thus, you should be able to apply polynomial algorithms to expressions for which this returns True, and `Poly(expr, *syms)` should work if and only if `expr.is_polynomial(*syms)` returns True. The polynomial does not have to be in expanded form. If no symbols are given, all free symbols in the expression will be used.

This is not part of the assumptions system. You cannot do `Symbol('z', polynomial=True)`.

Examples

```
>>> from sympy import Symbol, Function
>>> x = Symbol('x')
>>> ((x**2 + 1)**4).is_polynomial(x)
True
>>> ((x**2 + 1)**4).is_polynomial()
True
>>> (2**x + 1).is_polynomial(x)
False
>>> (2**x + 1).is_polynomial(2**x)
True
>>> f = Function('f')
>>> (f(x) + 1).is_polynomial(x)
False
>>> (f(x) + 1).is_polynomial(f(x))
True
>>> (1/f(x) + 1).is_polynomial(f(x))
False
```

```
>>> n = Symbol('n', nonnegative=True, integer=True)
>>> (x**n + 1).is_polynomial(x)
False
```

This function does not attempt any nontrivial simplifications that may result in an expression that does not appear to be a polynomial to become one.

```
>>> from sympy import sqrt, factor, cancel
>>> y = Symbol('y', positive=True)
>>> a = sqrt(y**2 + 2*y + 1)
>>> a.is_polynomial(y)
False
>>> factor(a)
y + 1
>>> factor(a).is_polynomial(y)
True
```

```
>>> b = (y**2 + 2*y + 1)/(y + 1)
>>> b.is_polynomial(y)
False
>>> cancel(b)
y + 1
>>> cancel(b).is_polynomial(y)
True
```

See also `.is_rational_function()`

`is_rational_function(*syms)`

Test whether function is a ratio of two polynomials in the given symbols, syms. When syms is not given, all free symbols will be used. The rational function does not have to be in expanded or in any kind of canonical form.

This function returns False for expressions that are “rational functions” with symbolic exponents. Thus, you should be able to call `.as_numer_denom()` and apply

polynomial algorithms to the result for expressions for which this returns True.

This is not part of the assumptions system. You cannot do `Symbol('z', rational_function=True)`.

Examples

```
>>> from sympy import Symbol, sin
>>> from sympy.abc import x, y
```

```
>>> (x/y).is_rational_function()
True
```

```
>>> (x**2).is_rational_function()
True
```

```
>>> (x/sin(y)).is_rational_function(y)
False
```

```
>>> n = Symbol('n', integer=True)
>>> (x**n + 1).is_rational_function(x)
False
```

This function does not attempt any nontrivial simplifications that may result in an expression that does not appear to be a rational function to become one.

```
>>> from sympy import sqrt, factor
>>> y = Symbol('y', positive=True)
>>> a = sqrt(y**2 + 2*y + 1)/y
>>> a.is_rational_function(y)
False
>>> factor(a)
(y + 1)/y
>>> factor(a).is_rational_function(y)
True
```

See also `is_algebraic_expr()`.

leadterm(*x*, *logx=None*, *cdir=0*)

Returns the leading term $a*x^b$ as a tuple (a, b).

Examples

```
>>> from sympy.abc import x
>>> (1+x+x**2).leadterm(x)
(1, 0)
>>> (1/x**2+x+x**2).leadterm(x)
(1, -2)
```

limit(*x*, *xlim*, *dir='+'*)

Compute limit $x \rightarrow xlim$.

lseries(*x=None, x0=0, dir='+', logx=None, cdir=0*)

Wrapper for series yielding an iterator of the terms of the series.

Note: an infinite series will yield an infinite iterator. The following, for example, will never terminate. It will just keep printing terms of the $\sin(x)$ series:

```
for term in sin(x).lseries(x):
    print term
```

The advantage of `lseries()` over `nseries()` is that many times you are just interested in the next term in the series (i.e. the first term for example), but you do not know how many you should ask for in `nseries()` using the “n” parameter.

See also `nseries()`.

normal()

expression $\rightarrow a/b$

See also:

[**as_numer_denom** \(page 955\)](#)

return (a, b) instead of a/b

nseries(*x=None, x0=0, n=6, dir='+', logx=None, cdir=0*)

Wrapper to `_eval_nseries` if assumptions allow, else to `series`.

If *x* is given, *x0* is 0, *dir* = '+', and self has *x*, then `_eval_nseries` is called. This calculates “n” terms in the innermost expressions and then builds up the final series just by “cross-multiplying” everything out.

The optional *logx* parameter can be used to replace any $\log(x)$ in the returned series with a symbolic value to avoid evaluating $\log(x)$ at 0. A symbol to use in place of $\log(x)$ should be provided.

Advantage - it’s fast, because we do not have to determine how many terms we need to calculate in advance.

Disadvantage - you may end up with less terms than you may have expected, but the $O(x^n)$ term appended will always be correct and so the result, though perhaps shorter, will also be correct.

If any of those assumptions is not met, this is treated like a wrapper to `series` which will try harder to return the correct number of terms.

See also `lseries()`.

Examples

```
>>> from sympy import sin, log, Symbol
>>> from sympy.abc import x, y
>>> sin(x).nseries(x, 0, 6)
x - x**3/6 + x**5/120 + O(x**6)
>>> log(x+1).nseries(x, 0, 5)
x - x**2/2 + x**3/3 - x**4/4 + O(x**5)
```

Handling of the `logx` parameter — in the following example the expansion fails since `sin` does not have an asymptotic expansion at $-\infty$ (the limit of $\log(x)$ as x approaches 0):

```
>>> e = sin(log(x))
>>> e.nseries(x, 0, 6)
Traceback (most recent call last):
...
PoleError: ...
...
>>> logx = Symbol('logx')
>>> e.nseries(x, 0, 6, logx=logx)
sin(logx)
```

In the following example, the expansion works but only returns self unless the `logx` parameter is used:

```
>>> e = x**y
>>> e.nseries(x, 0, 2)
x**y
>>> e.nseries(x, 0, 2, logx=logx)
exp(logx*y)
```

nsimplify(*constants=()*, *tolerance=None*, *full=False*)

See the `nsimplify` function in `sympy.simplify`

powsimp(**args*, ***kwargs*)

See the `powsimp` function in `sympy.simplify`

primitive()

Return the positive Rational that can be extracted non-recursively from every term of self (i.e., self is treated like an Add). This is like the `as_coeff_Mul()` method but `primitive` always extracts a positive Rational (never a negative or a Float).

Examples

```
>>> from sympy.abc import x
>>> (3*(x + 1)**2).primitive()
(3, (x + 1)**2)
>>> a = (6*x + 2); a.primitive()
(2, 3*x + 1)
>>> b = (x/2 + 3); b.primitive()
(1/2, x + 6)
>>> (a*b).primitive() == (1, a*b)
True
```

radsimp(***kwargs*)

See the `radsimp` function in `sympy.simplify`

ratsimp()

See the `ratsimp` function in `sympy.simplify`

removeO()

Removes the additive $O(\dots)$ symbol if there is one

round($n=None$)

Return x rounded to the given decimal place.

If a complex number would results, apply round to the real and imaginary components of the number.

Examples

```
>>> from sympy import pi, E, I, S, Number
>>> pi.round()
3
>>> pi.round(2)
3.14
>>> (2*pi + E*I).round()
6 + 3*I
```

The round method has a chopping effect:

```
>>> (2*pi + I/10).round()
6
>>> (pi/10 + 2*I).round()
2*I
>>> (pi/10 + E*I).round(2)
0.31 + 2.72*I
```

Notes

The Python round function uses the SymPy round method so it will always return a SymPy number (not a Python float or int):

```
>>> isinstance(round(S(123), -2), Number)
True
```

separate($deep=False$, $force=False$)

See the separate function in `sympy.simplify`

series($x=None$, $x0=0$, $n=6$, $dir='+'$, $\log x=None$, $cdir=0$)

Series expansion of “self” around $x = x_0$ yielding either terms of the series one by one (the lazy series given when $n=None$), else all the terms at once when $n \neq None$.

Returns the series expansion of “self” around the point $x = x_0$ with respect to x up to $O((x - x_0)^{*n}, x, x_0)$ (default n is 6).

If $x=None$ and self is univariate, the univariate symbol will be supplied, otherwise an error will be raised.

Parameters

expr : Expression

The expression whose series is to be expanded.

x : Symbol

It is the variable of the expression to be calculated.

x0 : Value

The value around which x is calculated. Can be any value from $-\infty$ to ∞ .

n : Value

The value used to represent the order in terms of x^{**n} , up to which the series is to be expanded.

dir : String, optional

The series-expansion can be bi-directional. If `dir="+"`, then $(x \rightarrow x_0+)$. If `dir="-"`, then $(x \rightarrow x_0-)$. For infinite `x0` (∞ or $-\infty$), the `dir` argument is determined from the direction of the infinity (i.e., `dir="-"` for ∞).

logx : optional

It is used to replace any $\log(x)$ in the returned series with a symbolic value rather than evaluating the actual value.

cdir : optional

It stands for complex direction, and indicates the direction from which the expansion needs to be evaluated.

Returns

Expr : Expression

Series expansion of the expression about x_0

Raises

TypeError

If `"n"` and `"x0"` are infinity objects

PoleError

If `"x0"` is an infinity object

Examples

```
>>> from sympy import cos, exp, tan
>>> from sympy.abc import x, y
>>> cos(x).series()
1 - x**2/2 + x**4/24 + O(x**6)
>>> cos(x).series(n=4)
1 - x**2/2 + O(x**4)
>>> cos(x).series(x, x0=1, n=2)
cos(1) - (x - 1)*sin(1) + O((x - 1)**2, (x, 1))
>>> e = cos(x + exp(y))
>>> e.series(y, n=2)
cos(x + 1) - y*sin(x + 1) + O(y**2)
>>> e.series(x, n=2)
cos(exp(y)) - x*sin(exp(y)) + O(x**2)
```

If `n=None` then a generator of the series terms will be returned.

```
>>> term=cos(x).series(n=None)
>>> [next(term) for i in range(2)]
[1, -x**2/2]
```

For `dir=+` (default) the series is calculated from the right and for `dir=-` the series from the left. For smooth functions this flag will not alter the results.

```
>>> abs(x).series(dir="+")
x
>>> abs(x).series(dir="-")
-x
>>> f = tan(x)
>>> f.series(x, 2, 6, "+")
tan(2) + (1 + tan(2)**2)*(x - 2) + (x - 2)**2*(tan(2)**3 + tan(2)) +
(x - 2)**3*(1/3 + 4*tan(2)**2/3 + tan(2)**4) + (x - 2)**4*(tan(2)**5 +
5*tan(2)**3/3 + 2*tan(2)/3) + (x - 2)**5*(2/15 + 17*tan(2)**2/15 +
2*tan(2)**4 + tan(2)**6) + 0((x - 2)**6, (x, 2))
```

```
>>> f.series(x, 2, 3, "-")
tan(2) + (2 - x)*(-tan(2)**2 - 1) + (2 - x)**2*(tan(2)**3 + tan(2))
+ 0((x - 2)**3, (x, 2))
```

For rational expressions this method may return original expression without the Order term. `>>> (1/x).series(x, n=8)` `1/x`

taylor_term(*n*, *x*, **previous_terms*)

General method for the taylor term.

This method is slow, because it differentiates *n*-times. Subclasses can redefine it to make it faster by using the “previous_terms”.

together(**args*, ***kwargs*)

See the together function in `sympy.polys`

trigsimp(**args*)

See the trigsimp function in `sympy.simplify`

UnevaluatedExpr

class `sympy.core.expr.UnevaluatedExpr`(*arg*, ***kwargs*)

Expression that is not evaluated unless released.

Examples

```
>>> from sympy import UnevaluatedExpr
>>> from sympy.abc import x
>>> x*(1/x)
1
>>> x*UnevaluatedExpr(1/x)
x*1/x
```

AtomicExpr

class sympy.core.expr.AtomicExpr(*args)

A parent class for object which are both atoms and Exprs.

For example: Symbol, Number, Rational, Integer, ... But not: Add, Mul, Pow, ...

symbol

Symbol

class sympy.core.symbol.Symbol(name, **assumptions)

Assumptions:

commutative = True

You can override the default assumptions in the constructor.

Examples

```
>>> from sympy import symbols
>>> A,B = symbols('A,B', commutative = False)
>>> bool(A*B != B*A)
True
>>> bool(A*B*2 == 2*A*B) == True # multiplication by scalars is
↪commutative
True
```

Wild

class sympy.core.symbol.Wild(name, exclude=(), properties=(), **assumptions)

A Wild symbol matches anything, or anything without whatever is explicitly excluded.

Parameters

name : str

Name of the Wild instance.

exclude : iterable, optional

Instances in exclude will not be matched.

properties : iterable of functions, optional

Functions, each taking an expressions as input and returns a bool. All functions in properties need to return True in order for the Wild instance to match the expression.

Examples

```
>>> from sympy import Wild, WildFunction, cos, pi
>>> from sympy.abc import x, y, z
>>> a = Wild('a')
>>> x.match(a)
{a_: x}
>>> pi.match(a)
{a_: pi}
>>> (3*x**2).match(a*x)
{a_: 3*x}
>>> cos(x).match(a)
{a_: cos(x)}
>>> b = Wild('b', exclude=[x])
>>> (3*x**2).match(b*x)
>>> b.match(a)
{a_: b_}
>>> A = WildFunction('A')
>>> A.match(a)
{a_: A_}
```

Tips

When using Wild, be sure to use the exclude keyword to make the pattern more precise. Without the exclude pattern, you may get matches that are technically correct, but not what you wanted. For example, using the above without exclude:

```
>>> from sympy import symbols
>>> a, b = symbols('a b', cls=Wild)
>>> (2 + 3*y).match(a*x + b*y)
{a_: 2/x, b_: 3}
```

This is technically correct, because $(2/x)*x + 3*y == 2 + 3*y$, but you probably wanted it to not match at all. The issue is that you really did not want a and b to include x and y, and the exclude parameter lets you specify exactly this. With the exclude parameter, the pattern will not match.

```
>>> a = Wild('a', exclude=[x, y])
>>> b = Wild('b', exclude=[x, y])
>>> (2 + 3*y).match(a*x + b*y)
```

Exclude also helps remove ambiguity from matches.

```
>>> E = 2*x**3*y*z
>>> a, b = symbols('a b', cls=Wild)
>>> E.match(a*b)
{a_: 2*y*z, b_: x**3}
>>> a = Wild('a', exclude=[x, y])
>>> E.match(a*b)
{a_: z, b_: 2*x**3*y}
>>> a = Wild('a', exclude=[x, y, z])
```

(continues on next page)

(continued from previous page)

```
>>> E.match(a*b)
{a_: 2, b_: x**3*y*z}
```

Wild also accepts a properties parameter:

```
>>> a = Wild('a', properties=[lambda k: k.is_Integer])
>>> E.match(a*b)
{a_: 2, b_: x**3*y*z}
```

Dummy

class `sympy.core.symbol.Dummy`(*name=None, dummy_index=None, **assumptions*)

Dummy symbols are each unique, even if they have the same name:

Examples

```
>>> from sympy import Dummy
>>> Dummy("x") == Dummy("x")
False
```

If a name is not supplied then a string value of an internal count will be used. This is useful when a temporary variable is needed and the name of the variable used in the expression is not important.

```
>>> Dummy()
_Dummy_10
```

symbols

`sympy.core.symbol.symbols`(*names, *, cls=<class 'sympy.core.symbol.Symbol'>, **args*)
→ Any

Transform strings into instances of [Symbol](#) (page 976) class.

[symbols\(\)](#) (page 978) function returns a sequence of symbols with names taken from *names* argument, which can be a comma or whitespace delimited string, or a sequence of strings:

```
>>> from sympy import symbols, Function

>>> x, y, z = symbols('x,y,z')
>>> a, b, c = symbols('a b c')
```

The type of output is dependent on the properties of input arguments:

```
>>> symbols('x')
x
>>> symbols('x,')
(x,)
```

(continues on next page)

(continued from previous page)

```
>>> symbols('x,y')
(x, y)
>>> symbols(('a', 'b', 'c'))
(a, b, c)
>>> symbols(['a', 'b', 'c'])
[a, b, c]
>>> symbols({'a', 'b', 'c'})
{a, b, c}
```

If an iterable container is needed for a single symbol, set the `seq` argument to `True` or terminate the symbol name with a comma:

```
>>> symbols('x', seq=True)
(x,)
```

To reduce typing, range syntax is supported to create indexed symbols. Ranges are indicated by a colon and the type of range is determined by the character to the right of the colon. If the character is a digit then all contiguous digits to the left are taken as the nonnegative starting value (or 0 if there is no digit left of the colon) and all contiguous digits to the right are taken as 1 greater than the ending value:

```
>>> symbols('x:10')
(x0, x1, x2, x3, x4, x5, x6, x7, x8, x9)

>>> symbols('x5:10')
(x5, x6, x7, x8, x9)
>>> symbols('x5(:2)')
(x50, x51)

>>> symbols('x5:10,y:5')
(x5, x6, x7, x8, x9, y0, y1, y2, y3, y4)

>>> symbols(('x5:10', 'y:5'))
((x5, x6, x7, x8, x9), (y0, y1, y2, y3, y4))
```

If the character to the right of the colon is a letter, then the single letter to the left (or 'a' if there is none) is taken as the start and all characters in the lexicographic range *through* the letter to the right are used as the range:

```
>>> symbols('x:z')
(x, y, z)
>>> symbols('x:c') # null range
()
>>> symbols('x(:c)')
(xa, xb, xc)

>>> symbols(':c')
(a, b, c)

>>> symbols('a:d, x:z')
(a, b, c, d, x, y, z)
```

(continues on next page)

(continued from previous page)

```
>>> symbols(('a:d', 'x:z'))  
(a, b, c, d), (x, y, z))
```

Multiple ranges are supported; contiguous numerical ranges should be separated by parentheses to disambiguate the ending number of one range from the starting number of the next:

```
>>> symbols('x:2(1:3)')  
(x01, x02, x11, x12)  
>>> symbols(':3:2') # parsing is from left to right  
(00, 01, 10, 11, 20, 21)
```

Only one pair of parentheses surrounding ranges are removed, so to include parentheses around ranges, double them. And to include spaces, commas, or colons, escape them with a backslash:

```
>>> symbols('x((a:b))')  
(x(a), x(b))  
>>> symbols(r'x(:1\,:2)') # or r'x((:1)\,(:2))'  
(x(0,0), x(0,1))
```

All newly created symbols have assumptions set according to args:

```
>>> a = symbols('a', integer=True)  
>>> a.is_integer  
True  
  
>>> x, y, z = symbols('x,y,z', real=True)  
>>> x.is_real and y.is_real and z.is_real  
True
```

Despite its name, `symbols()` (page 978) can create symbol-like objects like instances of Function or Wild classes. To achieve this, set `cls` keyword argument to the desired type:

```
>>> symbols('f,g,h', cls=Function)  
(f, g, h)  
  
>>> type(_[0])  
<class 'sympy.core.function.UndefinedFunction'>
```

var

`sympy.core.symbol.var(names, **args)`

Create symbols and inject them into the global namespace.

Explanation

This calls `symbols()` (page 978) with the same arguments and puts the results into the *global* namespace. It's recommended not to use `var()` (page 980) in library code, where `symbols()` (page 978) has to be used:

```
.. rubric:: Examples
```

```
>>> from sympy import var
```

```
>>> var('x')
x
>>> x # noqa: F821
x
```

```
>>> var('a,ab,abc')
(a, ab, abc)
>>> abc # noqa: F821
abc
```

```
>>> var('x,y', real=True)
(x, y)
>>> x.is_real and y.is_real # noqa: F821
True
```

See `symbols()` (page 978) documentation for more details on what kinds of arguments can be passed to `var()` (page 980).

numbers

Number

```
class sympy.core.numbers.Number(*obj)
```

Represents atomic numbers in SymPy.

Explanation

Floating point numbers are represented by the `Float` class. Rational numbers (of any size) are represented by the `Rational` class. Integer numbers (of any size) are represented by the `Integer` class. `Float` and `Rational` are subclasses of `Number`; `Integer` is a subclass of `Rational`.

For example, $2/3$ is represented as `Rational(2, 3)` which is a different object from the floating point number obtained with Python division $2/3$. Even for numbers that are exactly represented in binary, there is a difference between how two forms, such as `Rational(1, 2)` and `Float(0.5)`, are used in SymPy. The rational form is to be preferred in symbolic computations.

Other kinds of numbers, such as algebraic numbers `sqrt(2)` or complex numbers $3 + 4i$, are not instances of `Number` class as they are not atomic.

See also:

[Float](#) (page 982), [Integer](#) (page 987), [Rational](#) (page 985)

as_coeff_Add(*rational=False*)

Efficiently extract the coefficient of a summation.

as_coeff_Mul(*rational=False*)

Efficiently extract the coefficient of a product.

cofactors(*other*)

Compute GCD and cofactors of *self* and *other*.

gcd(*other*)

Compute GCD of *self* and *other*.

lcm(*other*)

Compute LCM of *self* and *other*.

Float

class `sympy.core.numbers.Float`(*num*, *dps=None*, *precision=None*)

Represent a floating-point number of arbitrary precision.

Examples

```
>>> from sympy import Float
>>> Float(3.5)
3.500000000000000
>>> Float(3)
3.000000000000000
```

Creating Floats from strings (and Python int and long types) will give a minimum precision of 15 digits, but the precision will automatically increase to capture all digits entered.

```
>>> Float(1)
1.000000000000000
>>> Float(10**20)
100000000000000000000.
>>> Float('1e20')
100000000000000000000.
```

However, *floating-point* numbers (Python float types) retain only 15 digits of precision:

```
>>> Float(1e20)
1.000000000000000e+20
>>> Float(1.23456789123456789)
1.23456789123457
```

It may be preferable to enter high-precision decimal numbers as strings:

```
>>> Float('1.23456789123456789')
1.23456789123456789
```

The desired number of digits can also be specified:

```
>>> Float('1e-3', 3)
0.00100
>>> Float(100, 4)
100.0
```

Float can automatically count significant figures if a null string is sent for the precision; spaces or underscores are also allowed. (Auto-counting is only allowed for strings, ints and longs).

```
>>> Float('123 456 789.123_456', '')
123456789.123456
>>> Float('12e-3', '')
0.012
>>> Float(3, '')
3.
```

If a number is written in scientific notation, only the digits before the exponent are considered significant if a decimal appears, otherwise the “e” signifies only how to move the decimal:

```
>>> Float('60.e2', '') # 2 digits significant
6.0e+3
>>> Float('60e2', '') # 4 digits significant
6000.
>>> Float('600e-2', '') # 3 digits significant
6.00
```

Notes

Floats are inexact by their nature unless their value is a binary-exact value.

```
>>> approx, exact = Float(.1, 1), Float(.125, 1)
```

For calculation purposes, evalf needs to be able to change the precision but this will not increase the accuracy of the inexact value. The following is the most accurate 5-digit approximation of a value of 0.1 that had only 1 digit of precision:

```
>>> approx.evalf(5)
0.099609
```

By contrast, 0.125 is exact in binary (as it is in base 10) and so it can be passed to Float or evalf to obtain an arbitrary precision with matching accuracy:

```
>>> Float(exact, 5)
0.12500
>>> exact.evalf(20)
0.12500000000000000000
```

Trying to make a high-precision Float from a float is not disallowed, but one must keep in mind that the *underlying float* (not the apparent decimal value) is being obtained with high precision. For example, 0.3 does not have a finite binary representation. The closest rational is the fraction $5404319552844595/2^{54}$. So if you try to obtain a Float of 0.3 to 20 digits of precision you will not see the same thing as 0.3 followed by 19 zeros:

```
>>> Float(0.3, 20)
0.29999999999999998890
```

If you want a 20-digit value of the decimal 0.3 (not the floating point approximation of 0.3) you should send the 0.3 as a string. The underlying representation is still binary but a higher precision than Python's float is used:

```
>>> Float('0.3', 20)
0.30000000000000000000
```

Although you can increase the precision of an existing Float using Float it will not increase the accuracy - the underlying value is not changed:

```
>>> def show(f): # binary rep of Float
...     from sympy import Mul, Pow
...     s, m, e, b = f._mpf_
...     v = Mul(int(m), Pow(2, int(e), evaluate=False), evaluate=False)
...     print('%s at prec=%s' % (v, f._prec))
...
>>> t = Float('0.3', 3)
>>> show(t)
4915/2**14 at prec=13
>>> show(Float(t, 20)) # higher prec, not higher accuracy
4915/2**14 at prec=70
>>> show(Float(t, 2)) # lower prec
307/2**10 at prec=10
```

The same thing happens when evalf is used on a Float:

```
>>> show(t.evalf(20))
4915/2**14 at prec=70
>>> show(t.evalf(2))
307/2**10 at prec=10
```

Finally, Floats can be instantiated with an mpf tuple (n, c, p) to produce the number $(-1)^n c 2^p$:

```
>>> n, c, p = 1, 5, 0
>>> (-1)**n*c*2**p
-5
>>> Float((1, 5, 0))
-5.0000000000000000
```

An actual mpf tuple also contains the number of bits in c as the last element of the tuple:

```
>>> _._mpf_
(1, 5, 0, 3)
```


This is not needed for instantiation and is not the same thing as the precision. The mpf tuple and the precision are two separate quantities that Float tracks.

In SymPy, a Float is a number that can be computed with arbitrary precision. Although floating point 'inf' and 'nan' are not such numbers, Float can create these numbers:

```
>>> Float('-inf')
-oo
>>> _.is_Float
False
```

Rational

class sympy.core.numbers.**Rational**(*p*, *q=None*, *gcd=None*)

Represents rational numbers (p/q) of any size.

Examples

```
>>> from sympy import Rational, nsimplify, S, pi
>>> Rational(1, 2)
1/2
```

Rational is unprejudiced in accepting input. If a float is passed, the underlying value of the binary representation will be returned:

```
>>> Rational(.5)
1/2
>>> Rational(.2)
3602879701896397/18014398509481984
```

If the simpler representation of the float is desired then consider limiting the denominator to the desired value or convert the float to a string (which is roughly equivalent to limiting the denominator to 10^{12}):

```
>>> Rational(str(.2))
1/5
>>> Rational(.2).limit_denominator(10**12)
1/5
```

An arbitrarily precise Rational is obtained when a string literal is passed:

```
>>> Rational("1.23")
123/100
>>> Rational('1e-2')
1/100
>>> Rational(".1")
1/10
>>> Rational('1e-2/3.2')
1/320
```

The conversion of other types of strings can be handled by the `sympify()` function, and conversion of floats to expressions or simple fractions can be handled with `nsimplify`:

```
>>> S('.[3]') # repeating digits in brackets
1/3
>>> S('3**2/10') # general expressions
9/10
>>> nsimplify(.3) # numbers that have a simple form
3/10
```

But if the input does not reduce to a literal Rational, an error will be raised:

```
>>> Rational(pi)
Traceback (most recent call last):
...
TypeError: invalid input: pi
```

Low-level

Access numerator and denominator as `.p` and `.q`:

```
>>> r = Rational(3, 4)
>>> r
3/4
>>> r.p
3
>>> r.q
4
```

Note that `p` and `q` return integers (not SymPy Integers) so some care is needed when using them in expressions:

```
>>> r.p/r.q
0.75
```

If an unevaluated Rational is desired, `gcd=1` can be passed and this will keep common divisors of the numerator and denominator from being eliminated. It is not possible, however, to leave a negative value in the denominator.

```
>>> Rational(2, 4, gcd=1)
2/4
>>> Rational(2, -4, gcd=1).q
4
```

See also:

[`sympy.core.sympify.sympify`](#) (page 918), [`sympy.simplify.simplify.nsimplify`](#) (page 667)

as_coeff_Add(*rational=False*)

Efficiently extract the coefficient of a summation.

as_coeff_Mul(*rational=False*)

Efficiently extract the coefficient of a product.

as_content_primitive(*radical=False, clear=True*)

Return the tuple (R, self/R) where R is the positive Rational extracted from self.

Examples

```
>>> from sympy import S
>>> (S(-3)/2).as_content_primitive()
(3/2, -1)
```

See docstring of `Expr.as_content_primitive` for more examples.

factors(*limit=None, use_trial=True, use_rho=False, use_pm1=False, verbose=False, visual=False*)

A wrapper to `factorint` which return factors of self that are smaller than limit (or cheap to compute). Special methods of factoring are disabled by default so that only trial division is used.

limit_denominator(*max_denominator=1000000*)

Closest Rational to self with denominator at most `max_denominator`.

Examples

```
>>> from sympy import Rational
>>> Rational('3.141592653589793').limit_denominator(10)
22/7
>>> Rational('3.141592653589793').limit_denominator(100)
311/99
```

Integer

class `sympy.core.numbers.Integer(i)`

Represents integer numbers of any size.

Examples

```
>>> from sympy import Integer
>>> Integer(3)
3
```

If a float or a rational is passed to `Integer`, the fractional part will be discarded; the effect is of rounding toward zero.

```
>>> Integer(3.8)
3
>>> Integer(-3.8)
-3
```

A string is acceptable input if it can be parsed as an integer:

```
>>> Integer("9" * 20)
99999999999999999999
```

It is rarely needed to explicitly instantiate an Integer, because Python integers are automatically converted to Integer when they are used in SymPy expressions.

AlgebraicNumber

class sympy.core.numbers.AlgebraicNumber(*expr*, *coeffs*=None, *alias*=None, ***args*)

Class for representing algebraic numbers in SymPy.

Symbolically, an instance of this class represents an element $\alpha \in \mathbb{Q}(\theta) \hookrightarrow \mathbb{C}$. That is, the algebraic number α is represented as an element of a particular number field $\mathbb{Q}(\theta)$, with a particular embedding of this field into the complex numbers.

Formally, the primitive element θ is given by two data points: (1) its minimal polynomial (which defines $\mathbb{Q}(\theta)$), and (2) a particular complex number that is a root of this polynomial (which defines the embedding $\mathbb{Q}(\theta) \hookrightarrow \mathbb{C}$). Finally, the algebraic number α which we represent is then given by the coefficients of a polynomial in θ .

static **__new__**(*cls*, *expr*, *coeffs*=None, *alias*=None, ***args*)

Construct a new algebraic number α belonging to a number field $k = \mathbb{Q}(\theta)$.

There are four instance attributes to be determined:

Attribute	Type/Meaning
root	Expr (page 947) for θ as a complex number
minpoly	Poly (page 2378), the minimal polynomial of θ
rep	DMP (page 2561) giving α as poly in θ
alias	Symbol (page 976) for θ , or None

See Parameters section for how they are determined.

Parameters

expr : [Expr](#) (page 947), or pair (m, r)

There are three distinct modes of construction, depending on what is passed as *expr*.

(1) *expr* is an [AlgebraicNumber](#) (page 988): In this case we begin by copying all four instance attributes from *expr*. If *coeffs* were also given, we compose the two coeff polynomials (see below). If an *alias* was given, it overrides.

(2) *expr* is any other type of [Expr](#) (page 947): Then root will equal *expr*. Therefore it must express an algebraic quantity, and we will compute its minpoly.

(3) *expr* is an ordered pair (m, r) giving the minpoly *m*, and a root *r* thereof, which together define θ . In this case *m* may be either a univariate [Poly](#) (page 2378) or any [Expr](#) (page 947) which represents the same, while *r* must be some [Expr](#) (page 947) representing a complex number that is a root of *m*, including both explicit expressions in radicals, and instances of [ComplexRootOf](#) (page 2431) or [AlgebraicNumber](#) (page 988).

coeffs : list, [ANP](#) (page 2568), None, optional (default=None)

This defines rep, giving the algebraic number α as a polynomial in θ .

If a list, the elements should be integers or rational numbers. If an [ANP](#) (page 2568), we take its coefficients (using its `to_list()` (page 2569) method). If `None`, then the list of coefficients defaults to `[1, 0]`, meaning that $\alpha = \theta$ is the primitive element of the field.

If `expr` was an [AlgebraicNumber](#) (page 988), let $g(x)$ be its rep polynomial, and let $f(x)$ be the polynomial defined by `coeffs`. Then `self.rep` will represent the composition $(f \circ g)(x)$.

alias : str, [Symbol](#) (page 976), `None`, optional (default=`None`)

This is a way to provide a name for the primitive element. We described several ways in which the `expr` argument can define the value of the primitive element, but none of these methods gave it a name. Here, for example, `alias` could be set as `Symbol('theta')`, in order to make this symbol appear when α is printed, or rendered as a polynomial, using the `as_poly()` (page 991) method.

Examples

Recall that we are constructing an algebraic number as a field element $\alpha \in \mathbb{Q}(\theta)$.

```
>>> from sympy import AlgebraicNumber, sqrt, CRootOf, S
>>> from sympy.abc import x
```

Example (1): $\alpha = \theta = \sqrt{2}$

```
>>> a1 = AlgebraicNumber(sqrt(2))
>>> a1.minpoly_of_element().as_expr(x)
x**2 - 2
>>> a1.evalf(10)
1.414213562
```

Example (2): $\alpha = 3\sqrt{2} - 5$, $\theta = \sqrt{2}$. We can either build on the last example:

```
>>> a2 = AlgebraicNumber(a1, [3, -5])
>>> a2.as_expr()
-5 + 3*sqrt(2)
```

or start from scratch:

```
>>> a2 = AlgebraicNumber(sqrt(2), [3, -5])
>>> a2.as_expr()
-5 + 3*sqrt(2)
```

Example (3): $\alpha = 6\sqrt{2} - 11$, $\theta = \sqrt{2}$. Again we can build on the previous example, and we see that the coeff polys are composed:

```
>>> a3 = AlgebraicNumber(a2, [2, -1])
>>> a3.as_expr()
-11 + 6*sqrt(2)
```

reflecting the fact that $(2x - 1) \circ (3x - 5) = 6x - 11$.

Example (4): $\alpha = \sqrt{2}$, $\theta = \sqrt{2} + \sqrt{3}$. The easiest way is to use the `to_number_field()` (page 2714) function:

```

>>> from sympy import to_number_field
>>> a4 = to_number_field(sqrt(2), sqrt(2) + sqrt(3))
>>> a4.minpoly_of_element().as_expr(x)
x**2 - 2
>>> a4.to_root()
sqrt(2)
>>> a4.primitive_element()
sqrt(2) + sqrt(3)
>>> a4.coeffs()
[1/2, 0, -9/2, 0]

```

but if you already knew the right coefficients, you could construct it directly:

```

>>> a4 = AlgebraicNumber(sqrt(2) + sqrt(3), [S(1)/2, 0, S(-9)/2, 0])
>>> a4.to_root()
sqrt(2)
>>> a4.primitive_element()
sqrt(2) + sqrt(3)

```

Example (5): Construct the Golden Ratio as an element of the 5th cyclotomic field, supposing we already know its coefficients. This time we introduce the alias ζ for the primitive element of the field:

```

>>> from sympy import cyclotomic_poly
>>> from sympy.abc import zeta
>>> a5 = AlgebraicNumber(CRootOf(cyclotomic_poly(5), -1),
...                       [-1, -1, 0, 0], alias=zeta)
>>> a5.as_poly().as_expr()
-zeta**3 - zeta**2
>>> a5.evalf()
1.61803398874989

```

(The index -1 to `CRootOf` selects the complex root with the largest real and imaginary parts, which in this case is $e^{2i\pi/5}$. See [ComplexRootOf](#) (page 2431).)

Example (6): Building on the last example, construct the number $2\phi \in \mathbb{Q}(\phi)$, where ϕ is the Golden Ratio:

```

>>> from sympy.abc import phi
>>> a6 = AlgebraicNumber(a5.to_root(), coeffs=[2, 0], alias=phi)
>>> a6.as_poly().as_expr()
2*phi
>>> a6.primitive_element().evalf()
1.61803398874989

```

Note that we needed to use `a5.to_root()`, since passing `a5` as the first argument would have constructed the number 2ϕ as an element of the field $\mathbb{Q}(\zeta)$:

```

>>> a6_wrong = AlgebraicNumber(a5, coeffs=[2, 0])
>>> a6_wrong.as_poly().as_expr()
-2*zeta**3 - 2*zeta**2
>>> a6_wrong.primitive_element().evalf()
0.309016994374947 + 0.951056516295154*I

```

as_expr(*x=None*)

Create a Basic expression from self.

as_poly(*x=None*)

Create a Poly instance from self.

coeffs()

Returns all SymPy coefficients of an algebraic number.

field_element(*coeffs*)

Form another element of the same number field.

Parameters

coeffs : list, [ANP](#) (page 2568)

Like the *coeffs* arg to the class [constructor](#) (page 988), defines the new element as a polynomial in the primitive element.

If a list, the elements should be integers or rational numbers. If an [ANP](#) (page 2568), we take its coefficients (using its [to_list\(\)](#) (page 2569) method).

Explanation

If we represent $\alpha \in \mathbb{Q}(\theta)$, form another element $\beta \in \mathbb{Q}(\theta)$ of the same number field.

Examples

```
>>> from sympy import AlgebraicNumber, sqrt
>>> a = AlgebraicNumber(sqrt(5), [-1, 1])
>>> b = a.field_element([3, 2])
>>> print(a)
1 - sqrt(5)
>>> print(b)
2 + 3*sqrt(5)
>>> print(b.primitive_element() == a.primitive_element())
True
```

See also:

[AlgebraicNumber.__new__](#) (page 988)

property is_aliased

Returns True if alias was set.

property is_primitive_element

Say whether this algebraic number $\alpha \in \mathbb{Q}(\theta)$ is equal to the primitive element θ for its field.

minpoly_of_element()

Compute the minimal polynomial for this algebraic number.

Explanation

Recall that we represent an element $\alpha \in \mathbb{Q}(\theta)$. Our instance attribute `self.minpoly` is the minimal polynomial for our primitive element θ . This method computes the minimal polynomial for α .

`native_coeffs()`

Returns all native coefficients of an algebraic number.

`primitive_element()`

Get the primitive element θ for the number field $\mathbb{Q}(\theta)$ to which this algebraic number α belongs.

Returns

AlgebraicNumber

`to_algebraic_integer()`

Convert `self` to an algebraic integer.

`to_primitive_element(radicals=True)`

Convert `self` to an [AlgebraicNumber](#) (page 988) instance that is equal to its own primitive element.

Parameters

radicals : boolean, optional (default=True)

If True, then we will try to return an [AlgebraicNumber](#) (page 988) whose root is an expression in radicals. If that is not possible (or if *radicals* is False), root will be a [ComplexRootOf](#) (page 2431).

Returns

AlgebraicNumber

Explanation

If we represent $\alpha \in \mathbb{Q}(\theta)$, $\alpha \neq \theta$, construct a new [AlgebraicNumber](#) (page 988) that represents $\alpha \in \mathbb{Q}(\alpha)$.

Examples

```
>>> from sympy import sqrt, to_number_field
>>> from sympy.abc import x
>>> a = to_number_field(sqrt(2), sqrt(2) + sqrt(3))
```

The [AlgebraicNumber](#) (page 988) `a` represents the number $\sqrt{2}$ in the field $\mathbb{Q}(\sqrt{2} + \sqrt{3})$. Rendering `a` as a polynomial,

```
>>> a.as_poly().as_expr(x)
x**3/2 - 9*x/2
```

reflects the fact that $\sqrt{2} = \theta^3/2 - 9\theta/2$, where $\theta = \sqrt{2} + \sqrt{3}$.

`a` is not equal to its own primitive element. Its `minpoly`


```
>>> a.minpoly.as_poly().as_expr(x)
x**4 - 10*x**2 + 1
```

is that of θ .

Converting to a primitive element,

```
>>> a_prim = a.to_primitive_element()
>>> a_prim.minpoly.as_poly().as_expr(x)
x**2 - 2
```

we obtain an *AlgebraicNumber* (page 988) whose minpoly is that of the number itself.

See also:

is_primitive_element (page 991)

to_root(*radicals=True*, *minpoly=None*)

Convert to an *Expr* (page 947) that is not an *AlgebraicNumber* (page 988), specifically, either a *ComplexRootOf* (page 2431), or, optionally and where possible, an expression in radicals.

Parameters

radicals : boolean, optional (default=True)

If True, then we will try to return the root as an expression in radicals. If that is not possible, we will return a *ComplexRootOf* (page 2431).

minpoly : *Poly* (page 2378)

If the minimal polynomial for *self* has been pre-computed, it can be passed in order to save time.

NumberSymbol

class sympy.core.numbers.NumberSymbol

approximation(*number_cls*)

Return an interval with *number_cls* endpoints that contains the value of NumberSymbol. If not implemented, then return None.

RealNumber

sympy.core.numbers.RealNumber

alias of *Float* (page 982)

igcd

`sympy.core.numbers.igcd(*args)`

Computes nonnegative integer greatest common divisor.

Explanation

The algorithm is based on the well known Euclid's algorithm [R112]. To improve speed, `igcd()` has its own caching mechanism.

Examples

```
>>> from sympy import igcd
>>> igcd(2, 4)
2
>>> igcd(5, 10, 15)
5
```

References

[R112]

ilcm

`sympy.core.numbers.ilcm(*args)`

Computes integer least common multiple.

Examples

```
>>> from sympy import ilcm
>>> ilcm(5, 10)
10
>>> ilcm(7, 3)
21
>>> ilcm(5, 10, 15)
30
```

seterr

`sympy.core.numbers.seterr(divide=False)`

Should SymPy raise an exception on 0/0 or return a nan?

divide == True raise an exception divide == False ... return nan

Zero

class `sympy.core.numbers.Zero`

The number zero.

Zero is a singleton, and can be accessed by `S.Zero`

Examples

```
>>> from sympy import S, Integer
>>> Integer(0) is S.Zero
True
>>> 1/S.Zero
zoo
```

References

[R113]

One

class `sympy.core.numbers.One`

The number one.

One is a singleton, and can be accessed by `S.One`.

Examples

```
>>> from sympy import S, Integer
>>> Integer(1) is S.One
True
```

References

[R114]

NegativeOne

class `sympy.core.numbers.NegativeOne`

The number negative one.

NegativeOne is a singleton, and can be accessed by `S.NegativeOne`.

Examples

```
>>> from sympy import S, Integer
>>> Integer(-1) is S.NegativeOne
True
```

See also:

[One](#) (page 995)

References

[R115]

Half

class `sympy.core.numbers.Half`

The rational number 1/2.

Half is a singleton, and can be accessed by `S.Half`.

Examples

```
>>> from sympy import S, Rational
>>> Rational(1, 2) is S.Half
True
```

References

[R116]

NaN

class sympy.core.numbers.NaN

Not a Number.

Explanation

This serves as a place holder for numeric values that are indeterminate. Most operations on NaN, produce another NaN. Most indeterminate forms, such as $0/0$ or $\infty - \infty$ produce NaN. Two exceptions are $0^{**}0$ and $\infty^{**}0$, which all produce 1 (this is consistent with Python's float).

NaN is loosely related to floating point nan, which is defined in the IEEE 754 floating point standard, and corresponds to the Python `float('nan')`. Differences are noted below.

NaN is mathematically not equal to anything else, even NaN itself. This explains the initially counter-intuitive results with `Eq` and `==` in the examples below.

NaN is not comparable so inequalities raise a `TypeError`. This is in contrast with floating point nan where all inequalities are false.

NaN is a singleton, and can be accessed by `S.NaN`, or can be imported as `nan`.

Examples

```

>>> from sympy import nan, S, oo, Eq
>>> nan is S.NaN
True
>>> oo - oo
nan
>>> nan + 1
nan
>>> Eq(nan, nan)    # mathematical equality
False
>>> nan == nan      # structural equality
True

```

References

[R117]

Infinity

class sympy.core.numbers.Infinity

Positive infinite quantity.

Explanation

In real analysis the symbol ∞ denotes an unbounded limit: $x \rightarrow \infty$ means that x grows without bound.

Infinity is often used not only to define a limit but as a value in the affinely extended real number system. Points labeled $+\infty$ and $-\infty$ can be added to the topological space of the real numbers, producing the two-point compactification of the real numbers. Adding algebraic properties to this gives us the extended real numbers.

Infinity is a singleton, and can be accessed by `S.Infinity`, or can be imported as `oo`.

Examples

```
>>> from sympy import oo, exp, limit, Symbol
>>> 1 + oo
oo
>>> 42/oo
0
>>> x = Symbol('x')
>>> limit(exp(x), x, oo)
oo
```

See also:

[*NegativeInfinity*](#) (page 998), [*NaN*](#) (page 997)

References

[R118]

NegativeInfinity

class sympy.core.numbers.NegativeInfinity

Negative infinite quantity.

NegativeInfinity is a singleton, and can be accessed by `S.NegativeInfinity`.

See also:

[*Infinity*](#) (page 998)

ComplexInfinity

class sympy.core.numbers.ComplexInfinity

Complex infinity.

Explanation

In complex analysis the symbol ∞ , called “complex infinity”, represents a quantity with infinite magnitude, but undetermined complex phase.

ComplexInfinity is a singleton, and can be accessed by `S.ComplexInfinity`, or can be imported as `zoo`.

Examples

```
>>> from sympy import zoo
>>> zoo + 42
zoo
>>> 42/zoo
0
>>> zoo + zoo
nan
>>> zoo*zoo
zoo
```

See also:

[*Infinity*](#) (page 998)

Exp1

class sympy.core.numbers.Exp1

The e constant.

Explanation

The transcendental number $e = 2.718281828\dots$ is the base of the natural logarithm and of the exponential function, $e = \exp(1)$. Sometimes called Euler’s number or Napier’s constant.

Exp1 is a singleton, and can be accessed by `S.Exp1`, or can be imported as `E`.

Examples

```
>>> from sympy import exp, log, E
>>> E is exp(1)
True
>>> log(E)
1
```

References

[R119]

ImaginaryUnit

class sympy.core.numbers.ImaginaryUnit

The imaginary unit, $i = \sqrt{-1}$.

I is a singleton, and can be accessed by S.I, or can be imported as I.

Examples

```
>>> from sympy import I, sqrt
>>> sqrt(-1)
I
>>> I*I
-1
>>> 1/I
-I
```

References

[R120]

Pi

class sympy.core.numbers.Pi

The π constant.

Explanation

The transcendental number $\pi = 3.141592654\dots$ represents the ratio of a circle's circumference to its diameter, the area of the unit circle, the half-period of trigonometric functions, and many other things in mathematics.

Pi is a singleton, and can be accessed by `S.Pi`, or can be imported as `pi`.

Examples

```
>>> from sympy import S, pi, oo, sin, exp, integrate, Symbol
>>> S.Pi
pi
>>> pi > 3
True
>>> pi.is_irrational
True
>>> x = Symbol('x')
>>> sin(x + 2*pi)
sin(x)
>>> integrate(exp(-x**2), (x, -oo, oo))
sqrt(pi)
```

References

[R121]

EulerGamma

class `sympy.core.numbers.EulerGamma`

The Euler-Mascheroni constant.

Explanation

$\gamma = 0.5772157\dots$ (also called Euler's constant) is a mathematical constant recurring in analysis and number theory. It is defined as the limiting difference between the harmonic series and the natural logarithm:

$$\gamma = \lim_{n \rightarrow \infty} \left(\sum_{k=1}^n \frac{1}{k} - \ln n \right)$$

EulerGamma is a singleton, and can be accessed by `S.EulerGamma`.

Examples

```
>>> from sympy import S
>>> S.EulerGamma.is_irrational
>>> S.EulerGamma > 0
True
>>> S.EulerGamma > 1
False
```

References

[R122]

Catalan

class sympy.core.numbers.Catalan

Catalan's constant.

Explanation

$G = 0.91596559\dots$ is given by the infinite series

$$G = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)^2}$$

Catalan is a singleton, and can be accessed by `S.Catalan`.

Examples

```
>>> from sympy import S
>>> S.Catalan.is_irrational
>>> S.Catalan > 0
True
>>> S.Catalan > 1
False
```

References

[R123]

GoldenRatio

class sympy.core.numbers.GoldenRatio

The golden ratio, ϕ .

Explanation

$\phi = \frac{1+\sqrt{5}}{2}$ is an algebraic number. Two quantities are in the golden ratio if their ratio is the same as the ratio of their sum to the larger of the two quantities, i.e. their maximum.

GoldenRatio is a singleton, and can be accessed by `S.GoldenRatio`.

Examples

```
>>> from sympy import S
>>> S.GoldenRatio > 1
True
>>> S.GoldenRatio.expand(func=True)
1/2 + sqrt(5)/2
>>> S.GoldenRatio.is_irrational
True
```

References

[R124]

TribonacciConstant

class sympy.core.numbers.TribonacciConstant

The tribonacci constant.

Explanation

The tribonacci numbers are like the Fibonacci numbers, but instead of starting with two predetermined terms, the sequence starts with three predetermined terms and each term afterwards is the sum of the preceding three terms.

The tribonacci constant is the ratio toward which adjacent tribonacci numbers tend. It is a root of the polynomial $x^3 - x^2 - x - 1 = 0$, and also satisfies the equation $x + x^{-3} = 2$.

TribonacciConstant is a singleton, and can be accessed by `S.TribonacciConstant`.

Examples

```
>>> from sympy import S
>>> S.TribonacciConstant > 1
True
>>> S.TribonacciConstant.expand(func=True)
1/3 + (19 - 3*sqrt(33))**(1/3)/3 + (3*sqrt(33) + 19)**(1/3)/3
>>> S.TribonacciConstant.is_irrational
True
>>> S.TribonacciConstant.n(20)
1.8392867552141611326
```

References

[R125]

mod_inverse

`sympy.core.numbers.mod_inverse(a, m)`

Return the number c such that, $a \times c = 1 \pmod{m}$ where c has the same sign as m . If no such value exists, a `ValueError` is raised.

Examples

```
>>> from sympy import mod_inverse, S
```

Suppose we wish to find multiplicative inverse x of 3 modulo 11. This is the same as finding x such that $3x = 1 \pmod{11}$. One value of x that satisfies this congruence is 4. Because $3 \times 4 = 12$ and $12 = 1 \pmod{11}$. This is the value returned by `mod_inverse`:

```
>>> mod_inverse(3, 11)
4
>>> mod_inverse(-3, 11)
7
```

When there is a common factor between the numerators of a and m the inverse does not exist:

```
>>> mod_inverse(2, 4)
Traceback (most recent call last):
...
ValueError: inverse of 2 mod 4 does not exist
```

```
>>> mod_inverse(S(2)/7, S(5)/2)
7/2
```

References

[R126], [R127]

power

Pow

class sympy.core.power.Pow(*b, e, evaluate=None*)

Defines the expression x^y as “x raised to a power y”

Deprecated since version 1.7: Using arguments that aren’t subclasses of [Expr](#) (page 947) in core operators ([Mul](#) (page 1009), [Add](#) (page 1013), and [Pow](#) (page 1005)) is deprecated. See [Core operators no longer accept non-Expr args](#) (page 174) for details.

Singleton definitions involving (0, 1, -1, oo, -oo, I, -I):

expr	value	reason
z^{**0}	1	Although arguments over 0^{**0} exist, see [2].
z^{**1}	z	
$(-\infty)^{**(-1)}$	0	
$(-1)^{**-1}$	-1	
$S.Zero^{**-1}$	∞	This is not strictly true, as 0^{**-1} may be undefined, but is convenient in some contexts where the base is assumed to be positive.
1^{**-1}	1	
∞^{**-1}	0	
$0^{**\infty}$	0	Because for all complex numbers z near 0, $z^{**\infty} \rightarrow 0$.
$0^{**-\infty}$	∞	This is not strictly true, as $0^{**\infty}$ may be oscillating between positive and negative values or rotating in the complex plane. It is convenient, however, when the base is positive.
$1^{**\infty}$ $1^{**-\infty}$	nan	Because there are various cases where $\lim(x(t),t)=1$, $\lim(y(t),t)=\infty$ (or $-\infty$), but $\lim(x(t)^{**}y(t), t) \neq 1$. See [3].
$b^{**\infty}$	nan	Because b^{**z} has no limit as $z \rightarrow \infty$
$(-1)^{**\infty}$ $(-1)^{**(-\infty)}$	nan	Because of oscillations in the limit.
$\infty^{**\infty}$	∞	
$\infty^{**-\infty}$	0	
$(-\infty)^{**\infty}$ $(-\infty)^{**-\infty}$	nan	
∞^{**I} ($-\infty)^{**I}$	nan	∞^{**e} could probably be best thought of as the limit of x^{**e} for real x as x tends to ∞ . If e is I , then the limit does not exist and nan is used to indicate that.
$\infty^{**(1+I)}$ ($-\infty)^{**(1+I)}$	∞	If the real part of e is positive, then the limit of $\text{abs}(x^{**e})$ is ∞ . So the limit value is ∞ .
$\infty^{*(-1+I)}$ $-\infty^{*(-1+I)}$	0	If the real part of e is negative, then the limit is 0.

Because symbolic computations are more flexible than floating point calculations and we prefer to never return an incorrect answer, we choose not to conform to all IEEE 754 conventions. This helps us avoid extra test-case code in the calculation of limits.

See also:

[*sympy.core.numbers.Infinity*](#) (page 998), [*sympy.core.numbers.NegativeInfinity*](#) (page 998), [*sympy.core.numbers.NaN*](#) (page 997)

References

[R128], [R129], [R130]

`as_base_exp()`

Return base and exp of self.

Explanation

If base is `1/Integer`, then return `Integer, -exp`. If this extra processing is not needed, the base and exp properties will give the raw arguments

Examples

```
>>> from sympy import Pow, S
>>> p = Pow(S.Half, 2, evaluate=False)
>>> p.as_base_exp()
(2, -2)
>>> p.args
(1/2, 2)
```

`as_content_primitive(radical=False, clear=True)`

Return the tuple (R, self/R) where R is the positive Rational extracted from self.

Examples

```
>>> from sympy import sqrt
>>> sqrt(4 + 4*sqrt(2)).as_content_primitive()
(2, sqrt(1 + sqrt(2)))
>>> sqrt(3 + 3*sqrt(2)).as_content_primitive()
(1, sqrt(3)*sqrt(1 + sqrt(2)))
```

```
>>> from sympy import expand_power_base, powsimp, Mul
>>> from sympy.abc import x, y
```

```
>>> ((2*x + 2)**2).as_content_primitive()
(4, (x + 1)**2)
>>> (4**((1 + y)/2)).as_content_primitive()
(2, 4**(y/2))
>>> (3**((1 + y)/2)).as_content_primitive()
(1, 3**((y + 1)/2))
>>> (3**((5 + y)/2)).as_content_primitive()
(9, 3**((y + 1)/2))
>>> eq = 3**(2 + 2*x)
>>> powsimp(eq) == eq
True
>>> eq.as_content_primitive()
(9, 3**(2*x))
```

(continues on next page)

(continued from previous page)

```
>>> powsimp(Mul(*_))
3**(2*x + 2)
```

```
>>> eq = (2 + 2*x)**y
>>> s = expand_power_base(eq); s.is_Mul, s
(False, (2*x + 2)**y)
>>> eq.as_content_primitive()
(1, (2*(x + 1))**y)
>>> s = expand_power_base(_[1]); s.is_Mul, s
(True, 2**y*(x + 1)**y)
```

See docstring of `Expr.as_content_primitive` for more examples.

integer_nthroot

`sympy.core.power.integer_nthroot(y, n)`

Return a tuple containing $x = \text{floor}(y^{1/n})$ and a boolean indicating whether the result is exact (that is, whether $x^n == y$).

Examples

```
>>> from sympy import integer_nthroot
>>> integer_nthroot(16, 2)
(4, True)
>>> integer_nthroot(26, 2)
(5, False)
```

To simply determine if a number is a perfect square, the `is_square` function should be used:

```
>>> from sympy.ntheory.primetest import is_square
>>> is_square(26)
False
```

See also:

[`sympy.ntheory.primetest.is\_square`](#) (page 1514), [`integer\_log`](#) (page 1008)

integer_log

`sympy.core.power.integer_log(y, x)`

Returns (e, bool) where e is the largest nonnegative integer such that $|y| \geq |x^e|$ and `bool` is True if $y = x^e$.

Examples

```
>>> from sympy import integer_log
>>> integer_log(125, 5)
(3, True)
>>> integer_log(17, 9)
(1, False)
>>> integer_log(4, -2)
(2, True)
>>> integer_log(-125, -5)
(3, True)
```

See also:

[integer_nthroot](#) (page 1008), [sympy.ntheory.primetest.is_square](#) (page 1514), [sympy.ntheory.factor_.multiplicity](#) (page 1488), [sympy.ntheory.factor_.perfect_power](#) (page 1488)

mul

Mul

class sympy.core.mul.Mul(*args, evaluate=None, _sympify=True)

Expression representing multiplication operation for algebraic field.

Deprecated since version 1.7: Using arguments that aren't subclasses of [Expr](#) (page 947) in core operators ([Mul](#) (page 1009), [Add](#) (page 1013), and [Pow](#) (page 1005)) is deprecated. See [Core operators no longer accept non-Expr args](#) (page 174) for details.

Every argument of `Mul()` must be `Expr`. Infix operator `*` on most scalar objects in SymPy calls this class.

Another use of `Mul()` is to represent the structure of abstract multiplication so that its arguments can be substituted to return different class. Refer to examples section for this.

`Mul()` evaluates the argument unless `evaluate=False` is passed. The evaluation logic includes:

1. **Flattening**
`Mul(x, Mul(y, z)) -> Mul(x, y, z)`
2. **Identity removing**
`Mul(x, 1, y) -> Mul(x, y)`
3. **Exponent collecting by .as_base_exp()**
`Mul(x, x**2) -> Pow(x, 3)`
4. **Term sorting**
`Mul(y, x, 2) -> Mul(2, x, y)`

Since multiplication can be vector space operation, arguments may have the different [sympy.core.kind.Kind\(\)](#) (page 1073). Kind of the resulting object is automatically inferred.

Examples

```
>>> from sympy import Mul
>>> from sympy.abc import x, y
>>> Mul(x, 1)
x
>>> Mul(x, x)
x**2
```

If `evaluate=False` is passed, result is not evaluated.

```
>>> Mul(1, 2, evaluate=False)
1*2
>>> Mul(x, x, evaluate=False)
x*x
```

`Mul()` also represents the general structure of multiplication operation.

```
>>> from sympy import MatrixSymbol
>>> A = MatrixSymbol('A', 2,2)
>>> expr = Mul(x,y).subs({y:A})
>>> expr
x*A
>>> type(expr)
<class 'sympy.matrices.expressions.matmul.MatMul'>
```

See also:

[*MatMul*](#) (page 1373)

as_coeff_Mul(*rational=False*)

Efficiently extract the coefficient of a product.

as_content_primitive(*radical=False, clear=True*)

Return the tuple (R, self/R) where R is the positive Rational extracted from self.

Examples

```
>>> from sympy import sqrt
>>> (-3*sqrt(2)*(2 - 2*sqrt(2))).as_content_primitive()
(6, -sqrt(2)*(1 - sqrt(2)))
```

See docstring of `Expr.as_content_primitive` for more examples.

as_ordered_factors(*order=None*)

Transform an expression into an ordered list of factors.

Examples

```
>>> from sympy import sin, cos
>>> from sympy.abc import x, y
```

```
>>> (2*x*y*sin(x)*cos(x)).as_ordered_factors()
[2, x, y, sin(x), cos(x)]
```

`as_two_terms()`

Return head and tail of self.

This is the most efficient way to get the head and tail of an expression.

- if you want only the head, use `self.args[0]`;
- if you want to process the arguments of the tail then use `self.as_coef_mul()` which gives the head and a tuple containing the arguments of the tail when treated as a Mul.
- if you want the coefficient when self is treated as an Add then use `self.as_coeff_add()[0]`

Examples

```
>>> from sympy.abc import x, y
>>> (3*x*y).as_two_terms()
(3, x*y)
```

`classmethod flatten(seq)`

Return commutative, noncommutative and order arguments by combining related terms.

Notes

- In an expression like `a*b*c`, Python process this through SymPy as `Mul(Mul(a, b), c)`. This can have undesirable consequences.
 - Sometimes terms are not combined as one would like: {c.f. <https://github.com/sympy/sympy/issues/4596>}

```
>>> from sympy import Mul, sqrt
>>> from sympy.abc import x, y, z
>>> 2*(x + 1) # this is the 2-arg Mul behavior
2*x + 2
>>> y*(x + 1)*2
2*y*(x + 1)
>>> 2*(x + 1)*y # 2-arg result will be obtained first
y*(2*x + 2)
>>> Mul(2, x + 1, y) # all 3 args simultaneously processed
2*y*(x + 1)
>>> 2*((x + 1)*y) # parentheses can control this behavior
2*y*(x + 1)
```

Powers with compound bases may not find a single base to combine with unless all arguments are processed at once. Post-processing may be necessary in such cases. {c.f. <https://github.com/sympy/sympy/issues/5728>}

```
>>> a = sqrt(x*sqrt(y))
>>> a**3
(x*sqrt(y))**(3/2)
>>> Mul(a,a,a)
(x*sqrt(y))**(3/2)
>>> a*a*a
x*sqrt(y)*sqrt(x*sqrt(y))
>>> _.subs(a.base, z).subs(z, a.base)
(x*sqrt(y))**(3/2)
```

- If more than two terms are being multiplied then all the previous terms will be re-processed for each new argument. So if each of a , b and c were *Mul* (page 1009) expression, then $a*b*c$ (or building up the product with $=$) will process all the arguments of a and b twice: once when $a*b$ is computed and again when c is multiplied.

Using *Mul*(a , b , c) will process all arguments once.

- The results of *Mul* are cached according to arguments, so *flatten* will only be called once for *Mul*(a , b , c). If you can structure a calculation so the arguments are most likely to be repeats then this can save time in computing the answer. For example, say you had a *Mul*, M , that you wished to divide by $d[i]$ and multiply by $n[i]$ and you suspect there are many repeats in n . It would be better to compute $M*n[i]/d[i]$ rather than $M/d[i]*n[i]$ since every time $n[i]$ is a repeat, the product, $M*n[i]$ will be returned without flattening – the cached value will be returned. If you divide by the $d[i]$ first (and those are more unique than the $n[i]$) then that will create a new *Mul*, $M/d[i]$ the args of which will be traversed again when it is multiplied by $n[i]$.

{c.f. <https://github.com/sympy/sympy/issues/5706>}

This consideration is moot if the cache is turned off.

Nb

The validity of the above notes depends on the implementation details of `Mul` and `flatten` which may change at any time. Therefore, you should only consider them when your code is highly performance sensitive.

Removal of 1 from the sequence is already handled by `AssocOp.__new__`.

prod

`sympy.core.mul.prod(a, start=1)`

Return product of elements of a. Start with int 1 so if only ints are included then an int result is returned.

Examples

```
>>> from sympy import prod, S
>>> prod(range(3))
0
>>> type(_) is int
True
>>> prod([S(2), 3])
6
>>> _.is_Integer
True
```

You can start the product at something other than 1:

```
>>> prod([1, 2], 3)
6
```

add**Add**

class `sympy.core.add.Add(*args, evaluate=None, _sympify=True)`

Expression representing addition operation for algebraic group.

Deprecated since version 1.7: Using arguments that aren't subclasses of [Expr](#) (page 947) in core operators ([Mul](#) (page 1009), [Add](#) (page 1013), and [Pow](#) (page 1005)) is deprecated. See [Core operators no longer accept non-Expr args](#) (page 174) for details.

Every argument of `Add()` must be `Expr`. Infix operator `+` on most scalar objects in SymPy calls this class.

Another use of `Add()` is to represent the structure of abstract addition so that its arguments can be substituted to return different class. Refer to examples section for this.

`Add()` evaluates the argument unless `evaluate=False` is passed. The evaluation logic includes:

1. Flattening

`Add(x, Add(y, z)) -> Add(x, y, z)`

2. Identity removing

`Add(x, 0, y) -> Add(x, y)`

3. Coefficient collecting by `.as_coeff_Mul()`

`Add(x, 2*x) -> Mul(3, x)`

4. Term sorting

`Add(y, x, 2) -> Add(2, x, y)`

If no argument is passed, identity element 0 is returned. If single element is passed, that element is returned.

Note that `Add(*args)` is more efficient than `sum(args)` because it flattens the arguments. `sum(a, b, c, ...)` recursively adds the arguments as `a + (b + (c + ...))`, which has quadratic complexity. On the other hand, `Add(a, b, c, d)` does not assume nested structure, making the complexity linear.

Since addition is group operation, every argument should have the same [`sympy.core.kind.Kind\(\)`](#) (page 1073).

Examples

```
>>> from sympy import Add, I
>>> from sympy.abc import x, y
>>> Add(x, 1)
x + 1
>>> Add(x, x)
2*x
>>> 2*x**2 + 3*x + I*y + 2*y + 2*x/5 + 1.0*y + 1
2*x**2 + 17*x/5 + 3.0*y + I*y + 1
```

If `evaluate=False` is passed, result is not evaluated.

```
>>> Add(1, 2, evaluate=False)
1 + 2
>>> Add(x, x, evaluate=False)
x + x
```

`Add()` also represents the general structure of addition operation.

```
>>> from sympy import MatrixSymbol
>>> A,B = MatrixSymbol('A', 2,2), MatrixSymbol('B', 2,2)
>>> expr = Add(x,y).subs({x:A, y:B})
>>> expr
A + B
>>> type(expr)
<class 'sympy.matrices.expressions.matadd.MatAdd'>
```

Note that the printers do not display in args order.

```
>>> Add(x, 1)
x + 1
```

(continues on next page)

(continued from previous page)

```
>>> Add(x, 1).args
(1, x)
```

See also:[MatAdd](#) (page 1372)**as_coeff_Add**(*rational=False, deps=None*)

Efficiently extract the coefficient of a summation.

as_coeff_add(**deps*)

Returns a tuple (coeff, args) where self is treated as an Add and coeff is the Number term and args is a tuple of all other terms.

Examples

```
>>> from sympy.abc import x
>>> (7 + 3*x).as_coeff_add()
(7, (3*x,))
>>> (7*x).as_coeff_add()
(0, (7*x,))
```

as_content_primitive(*radical=False, clear=True*)

Return the tuple (R, self/R) where R is the positive Rational extracted from self. If radical is True (default is False) then common radicals will be removed and included as a factor of the primitive expression.

Examples

```
>>> from sympy import sqrt
>>> (3 + 3*sqrt(2)).as_content_primitive()
(3, 1 + sqrt(2))
```

Radical content can also be factored out of the primitive:

```
>>> (2*sqrt(2) + 4*sqrt(10)).as_content_primitive(radical=True)
(2, sqrt(2)*(1 + 2*sqrt(5)))
```

See docstring of Expr.as_content_primitive for more examples.

as_numer_denom()

Decomposes an expression to its numerator part and its denominator part.

Examples

```
>>> from sympy.abc import x, y, z
>>> (x*y/z).as_numer_denom()
(x*y, z)
>>> (x*(y + 1)/y**7).as_numer_denom()
(x*(y + 1), y**7)
```

See also:

[*sympy.core.expr.Expr.as_numer_denom*](#) (page 955)

as_real_imag(*deep=True, **hints*)

returns a tuple representing a complex number

Examples

```
>>> from sympy import I
>>> (7 + 9*I).as_real_imag()
(7, 9)
>>> ((1 + I)/(1 - I)).as_real_imag()
(0, 1)
>>> ((1 + 2*I)*(1 + 3*I)).as_real_imag()
(-5, 5)
```

as_two_terms()

Return head and tail of self.

This is the most efficient way to get the head and tail of an expression.

- if you want only the head, use `self.args[0]`;
- if you want to process the arguments of the tail then use `self.as_coef_add()` which gives the head and a tuple containing the arguments of the tail when treated as an Add.
- if you want the coefficient when self is treated as a Mul then use `self.as_coeff_mul()[0]`

```
>>> from sympy.abc import x, y
>>> (3*x - 2*y + 5).as_two_terms()
(5, 3*x - 2*y)
```

classmethod class_key()

Nice order of classes

extract_leading_order(*symbols, point=None*)

Returns the leading term and its order.

Examples

```
>>> from sympy.abc import x
>>> (x + 1 + 1/x**5).extract_leading_order(x)
((x**(-5), 0(x**(-5))),)
>>> (1 + x).extract_leading_order(x)
((1, 0(1)),)
>>> (x + x**2).extract_leading_order(x)
((x, 0(x)),)
```

classmethod `flatten(seq)`

Takes the sequence “seq” of nested Adds and returns a flatten list.

Returns: (commutative_part, noncommutative_part, order_symbols)

Applies associativity, all terms are commutable with respect to addition.

NB: the removal of 0 is already handled by AssocOp.__new__

See also:

[*sympy.core.mul.Mul.flatten*](#) (page 1011)

primitive()

Return (R, self/R) where R` is the Rational GCD of self`.

R is collected only from the leading coefficient of each term.

Examples

```
>>> from sympy.abc import x, y
```

```
>>> (2*x + 4*y).primitive()
(2, x + 2*y)
```

```
>>> (2*x/3 + 4*y/9).primitive()
(2/9, 3*x + 2*y)
```

```
>>> (2*x/3 + 4.2*y).primitive()
(1/3, 2*x + 12.6*y)
```

No subprocessing of term factors is performed:

```
>>> ((2 + 2*x)*x + 2).primitive()
(1, x*(2*x + 2) + 2)
```

Recursive processing can be done with the `as_content_primitive()` method:

```
>>> ((2 + 2*x)*x + 2).as_content_primitive()
(2, x*(x + 1) + 1)
```

See also: `primitive()` function in `polytools.py`

mod

Mod

class sympy.core.mod.**Mod**(*p*, *q*)

Represents a modulo operation on symbolic expressions.

Parameters

p : Expr

Dividend.

q : Expr

Divisor.

Notes

The convention used is the same as Python's: the remainder always has the same sign as the divisor.

Examples

```
>>> from sympy.abc import x, y
>>> x**2 % y
Mod(x**2, y)
>>> _.subs({x: 5, y: 6})
1
```

relational

Rel

class sympy.core.relational.**Relational**(*lhs*, *rhs*, *rop*=None, ***assumptions*)

Base class for all relation types.

Parameters

rop : str or None

Indicates what subclass to instantiate. Valid values can be found in the keys of Relational.ValidRelationOperator.

Explanation

Subclasses of Relational should generally be instantiated directly, but Relational can be instantiated with a valid rop value to dispatch to the appropriate subclass.

Examples

```
>>> from sympy import Rel
>>> from sympy.abc import x, y
>>> Rel(y, x + x**2, '==')
Eq(y, x**2 + x)
```

A relation's type can be defined upon creation using rop. The relation type of an existing expression can be obtained using its rel_op property. Here is a table of all the relation types, along with their rop and rel_op values:

Relation	rop	rel_op
Equality	== or eq or None	==
Unequality	!= or ne	!=
GreaterThan	>= or ge	>=
LessThan	<= or le	<=
StrictGreaterThan	> or gt	>
StrictLessThan	< or lt	<

For example, setting rop to == produces an Equality relation, Eq(). So does setting rop to eq, or leaving rop unspecified. That is, the first three Rel() below all produce the same result. Using a rop from a different row in the table produces a different relation type. For example, the fourth Rel() below using lt for rop produces a StrictLessThan inequality:

```
>>> from sympy import Rel
>>> from sympy.abc import x, y
>>> Rel(y, x + x**2, '==')
Eq(y, x**2 + x)
>>> Rel(y, x + x**2, 'eq')
Eq(y, x**2 + x)
>>> Rel(y, x + x**2)
Eq(y, x**2 + x)
>>> Rel(y, x + x**2, 'lt')
y < x**2 + x
```

To obtain the relation type of an existing expression, get its rel_op property. For example, rel_op is == for the Equality relation above, and < for the strict less than inequality above:

```
>>> from sympy import Rel
>>> from sympy.abc import x, y
>>> my_equality = Rel(y, x + x**2, '==')
>>> my_equality.rel_op
'=='
>>> my_inequality = Rel(y, x + x**2, 'lt')
```

(continues on next page)

(continued from previous page)

```
>>> my_inequality.rel_op
'<'
```

property canonical

Return a canonical form of the relational by putting a number on the rhs, canonically removing a sign or else ordering the args canonically. No other simplification is attempted.

Examples

```
>>> from sympy.abc import x, y
>>> x < 2
x < 2
>>> _.reversed.canonical
x < 2
>>> (-y < x).canonical
x > -y
>>> (-y > x).canonical
x < -y
>>> (-y < -x).canonical
x < y
```

The canonicalization is recursively applied:

```
>>> from sympy import Eq
>>> Eq(x < y, y > x).canonical
True
```

equals(*other*, *failing_expression*=False)

Return True if the sides of the relationship are mathematically identical and the type of relationship is the same. If *failing_expression* is True, return the expression whose truth value was unknown.

property lhs

The left-hand side of the relation.

property negated

Return the negated relationship.

Examples

```
>>> from sympy import Eq
>>> from sympy.abc import x
>>> Eq(x, 1)
Eq(x, 1)
>>> _.negated
Ne(x, 1)
>>> x < 1
x < 1
```

(continues on next page)

(continued from previous page)

```
>>> _.negated
x >= 1
```

Notes

This works more or less identical to `~/Not`. The difference is that `negated` returns the relationship even if `evaluate=False`. Hence, this is useful in code when checking for e.g. negated relations to existing ones as it will not be affected by the *evaluate* flag.

property reversed

Return the relationship with sides reversed.

Examples

```
>>> from sympy import Eq
>>> from sympy.abc import x
>>> Eq(x, 1)
Eq(x, 1)
>>> _.reversed
Eq(1, x)
>>> x < 1
x < 1
>>> _.reversed
1 > x
```

property reversedsign

Return the relationship with signs reversed.

Examples

```
>>> from sympy import Eq
>>> from sympy.abc import x
>>> Eq(x, 1)
Eq(x, 1)
>>> _.reversedsign
Eq(-x, -1)
>>> x < 1
x < 1
>>> _.reversedsign
-x > -1
```

property rhs

The right-hand side of the relation.

property strict

return the strict version of the inequality or self

Examples

```
>>> from sympy.abc import x
>>> (x <= 1).strict
x < 1
>>> _.strict
x < 1
```

property weak

return the non-strict version of the inequality or self

Examples

```
>>> from sympy.abc import x
>>> (x < 1).weak
x <= 1
>>> _.weak
x <= 1
```

`sympy.core.relational.Re1`

alias of *Relational* (page 1018)

Eq

`sympy.core.relational.Eq`

alias of *Equality* (page 1023)

Ne

`sympy.core.relational.Ne`

alias of *Unequality* (page 1031)

Lt

`sympy.core.relational.Lt`

alias of *StrictLessThan* (page 1035)

Le

`sympy.core.relational.Le`

alias of *LessThan* (page 1028)

Gt

`sympy.core.relational.Gt`

alias of *StrictGreaterThan* (page 1032)

Ge

`sympy.core.relational.Ge`

alias of *GreaterThan* (page 1024)

Equality

class `sympy.core.relational.Equality(lhs, rhs=None, **options)`

An equal relation between two objects.

Explanation

Represents that two objects are equal. If they can be easily shown to be definitively equal (or unequal), this will reduce to True (or False). Otherwise, the relation is maintained as an unevaluated Equality object. Use the `simplify` function on this object for more nontrivial evaluation of the equality relation.

As usual, the keyword argument `evaluate=False` can be used to prevent any evaluation.

Examples

```

>>> from sympy import Eq, simplify, exp, cos
>>> from sympy.abc import x, y
>>> Eq(y, x + x**2)
Eq(y, x**2 + x)
>>> Eq(2, 5)
False
>>> Eq(2, 5, evaluate=False)
Eq(2, 5)
>>> _.doit()
False
>>> Eq(exp(x), exp(x).rewrite(cos))
Eq(exp(x), sinh(x) + cosh(x))
>>> simplify(_)
True

```

Notes

Python treats 1 and True (and 0 and False) as being equal; SymPy does not. And integer will always compare as unequal to a Boolean:

```
>>> Eq(True, 1), True == 1
(False, True)
```

This class is not the same as the == operator. The == operator tests for exact structural equality between two expressions; this class compares expressions mathematically.

If either object defines an `_eval_Eq` method, it can be used in place of the default algorithm. If `lhs._eval_Eq(rhs)` or `rhs._eval_Eq(lhs)` returns anything other than None, that return value will be substituted for the Equality. If None is returned by `_eval_Eq`, an Equality object will be created as usual.

Since this object is already an expression, it does not respond to the method `as_expr` if one tries to create $x - y$ from `Eq(x, y)`. This can be done with the `rewrite(Add)` method.

Deprecated since version 1.5: `Eq(expr)` with a single argument is a shorthand for `Eq(expr, 0)`, but this behavior is deprecated and will be removed in a future version of SymPy.

See also:

[*`sympy.logic.boolalg.Equivalent`*](#) (page 1171)

for representing equality between two boolean expressions

`as_poly(*gens, **kwargs)`

Returns lhs-rhs as a Poly

Examples

```
>>> from sympy import Eq
>>> from sympy.abc import x
>>> Eq(x**2, 1).as_poly(x)
Poly(x**2 - 1, x, domain='ZZ')
```

`integrate(*args, **kwargs)`

See the `integrate` function in `sympy.integrals`

GreaterThan

`class sympy.core.relational.GreaterThan(lhs, rhs, **options)`

Class representations of inequalities.

Explanation

The `*Than` classes represent inequal relationships, where the left-hand side is generally bigger or smaller than the right-hand side. For example, the `GreaterThan` class represents an inequal relationship where the left-hand side is at least as big as the right side, if not bigger. In mathematical notation:

$$\text{lhs} \geq \text{rhs}$$

In total, there are four `*Than` classes, to represent the four inequalities:

Class Name	Symbol
<code>GreaterThan</code>	$\geq$
<code>LessThan</code>	$\leq$
<code>StrictGreaterThan</code>	$>$
<code>StrictLessThan</code>	$<$

All classes take two arguments, `lhs` and `rhs`.

Signature Example	Math Equivalent
<code>GreaterThan(lhs, rhs)</code>	$\text{lhs} \geq \text{rhs}$
<code>LessThan(lhs, rhs)</code>	$\text{lhs} \leq \text{rhs}$
<code>StrictGreaterThan(lhs, rhs)</code>	$\text{lhs} > \text{rhs}$
<code>StrictLessThan(lhs, rhs)</code>	$\text{lhs} < \text{rhs}$

In addition to the normal `.lhs` and `.rhs` of `Relations`, `*Than` inequality objects also have the `.lts` and `.gts` properties, which represent the “less than side” and “greater than side” of the operator. Use of `.lts` and `.gts` in an algorithm rather than `.lhs` and `.rhs` as an assumption of inequality direction will make more explicit the intent of a certain section of code, and will make it similarly more robust to client code changes:

```
>>> from sympy import GreaterThan, StrictGreaterThan
>>> from sympy import LessThan, StrictLessThan
>>> from sympy import And, Ge, Gt, Le, Lt, Rel, S
>>> from sympy.abc import x, y, z
>>> from sympy.core.relational import Relational
```

```
>>> e = GreaterThan(x, 1)
>>> e
x >= 1
>>> '%s >= %s is the same as %s <= %s' % (e.gts, e.lts, e.lts, e.gts)
'x >= 1 is the same as 1 <= x'
```

Examples

One generally does not instantiate these classes directly, but uses various convenience methods:

```
>>> for f in [Ge, Gt, Le, Lt]: # convenience wrappers
...     print(f(x, 2))
x >= 2
x > 2
x <= 2
x < 2
```

Another option is to use the Python inequality operators ($\geq$, $>$, $\leq$, $<$) directly. Their main advantage over the `Ge`, `Gt`, `Le`, and `Lt` counterparts, is that one can write a more “mathematical looking” statement rather than littering the math with oddball function calls. However there are certain (minor) caveats of which to be aware (search for ‘gotcha’, below).

```
>>> x >= 2
x >= 2
>>> _ == Ge(x, 2)
True
```

However, it is also perfectly valid to instantiate a `*Than` class less succinctly and less conveniently:

```
>>> Rel(x, 1, ">")
x > 1
>>> Relational(x, 1, ">")
x > 1
```

```
>>> StrictGreaterThan(x, 1)
x > 1
>>> GreaterThan(x, 1)
x >= 1
>>> LessThan(x, 1)
x <= 1
>>> StrictLessThan(x, 1)
x < 1
```

Notes

There are a couple of “gotchas” to be aware of when using Python’s operators.

The first is that what you write is not always what you get:

```
>>> 1 < x
x > 1
```

Due to the order that Python parses a statement, it may not immediately find two objects comparable. When `1 < x` is evaluated, Python recognizes that the number 1 is a native number and that `x` is *not*. Because a native Python number does not know how to compare itself with a SymPy object Python will try

the reflective operation, $x > 1$ and that is the form that gets evaluated, hence returned.

If the order of the statement is important (for visual output to the console, perhaps), one can work around this annoyance in a couple ways:

(1) “sympify” the literal before comparison

```
>>> S(1) < x
1 < x
```

(2) use one of the wrappers or less succinct methods described above

```
>>> Lt(1, x)
1 < x
>>> Relational(1, x, "<")
1 < x
```

The second gotcha involves writing equality tests between relationals when one or both sides of the test involve a literal relational:

```
>>> e = x < 1; e
x < 1
>>> e == e # neither side is a literal
True
>>> e == x < 1 # expecting True, too
False
>>> e != x < 1 # expecting False
x < 1
>>> x < 1 != x < 1 # expecting False or the same thing as before
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
```

The solution for this case is to wrap literal relationals in parentheses:

```
>>> e == (x < 1)
True
>>> e != (x < 1)
False
>>> (x < 1) != (x < 1)
False
```

The third gotcha involves chained inequalities not involving `==` or `!=`. Occasionally, one may be tempted to write:

```
>>> e = x < y < z
Traceback (most recent call last):
...
TypeError: symbolic boolean expression has no truth value.
```

Due to an implementation detail or decision of Python [R131], there is no way for SymPy to create a chained inequality with that syntax so one must use `And`:

```
>>> e = And(x < y, y < z)
>>> type( e )
And
>>> e
(x < y) & (y < z)
```

Although this can also be done with the ‘&’ operator, it cannot be done with the ‘and’ operator:

```
>>> (x < y) & (y < z)
(x < y) & (y < z)
>>> (x < y) and (y < z)
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
```

LessThan

class sympy.core.relational.**LessThan**(lhs, rhs, **options)

Class representations of inequalities.

Explanation

The *Than classes represent inequal relationships, where the left-hand side is generally bigger or smaller than the right-hand side. For example, the GreaterThan class represents an inequal relationship where the left-hand side is at least as big as the right side, if not bigger. In mathematical notation:

$\text{lhs} \geq \text{rhs}$

In total, there are four *Than classes, to represent the four inequalities:

Class Name	Symbol
GreaterThan	$\geq$
LessThan	$\leq$
StrictGreaterThan	$>$
StrictLessThan	$<$

All classes take two arguments, lhs and rhs.

Signature Example	Math Equivalent
GreaterThan(lhs, rhs)	$\text{lhs} \geq \text{rhs}$
LessThan(lhs, rhs)	$\text{lhs} \leq \text{rhs}$
StrictGreaterThan(lhs, rhs)	$\text{lhs} > \text{rhs}$
StrictLessThan(lhs, rhs)	$\text{lhs} < \text{rhs}$

In addition to the normal .lhs and .rhs of Relations, *Than inequality objects also have the .lts and .gts properties, which represent the “less than side” and “greater than side” of the operator. Use of .lts and .gts in an algorithm rather than .lhs and .rhs as an assumption of inequality direction will make more explicit the intent of a certain section of code, and will make it similarly more robust to client code changes:

```
>>> from sympy import GreaterThan, StrictGreaterThan
>>> from sympy import LessThan, StrictLessThan
>>> from sympy import And, Ge, Gt, Le, Lt, Rel, S
>>> from sympy.abc import x, y, z
>>> from sympy.core.relational import Relational
```

```
>>> e = GreaterThan(x, 1)
>>> e
x >= 1
>>> '%s >= %s is the same as %s <= %s' % (e.gts, e.lts, e.lts, e.gts)
'x >= 1 is the same as 1 <= x'
```

Examples

One generally does not instantiate these classes directly, but uses various convenience methods:

```
>>> for f in [Ge, Gt, Le, Lt]: # convenience wrappers
...     print(f(x, 2))
x >= 2
x > 2
x <= 2
x < 2
```

Another option is to use the Python inequality operators ($\geq$, $>$, $\leq$, $<$) directly. Their main advantage over the `Ge`, `Gt`, `Le`, and `Lt` counterparts, is that one can write a more “mathematical looking” statement rather than littering the math with oddball function calls. However there are certain (minor) caveats of which to be aware (search for ‘gotcha’, below).

```
>>> x >= 2
x >= 2
>>> _ == Ge(x, 2)
True
```

However, it is also perfectly valid to instantiate a `*Than` class less succinctly and less conveniently:

```
>>> Rel(x, 1, ">")
x > 1
>>> Relational(x, 1, ">")
x > 1
```

```
>>> StrictGreaterThan(x, 1)
x > 1
>>> GreaterThan(x, 1)
x >= 1
>>> LessThan(x, 1)
x <= 1
>>> StrictLessThan(x, 1)
x < 1
```

Notes

There are a couple of “gotchas” to be aware of when using Python’s operators.

The first is that what you write is not always what you get:

```
>>> 1 < x
x > 1
```

Due to the order that Python parses a statement, it may not immediately find two objects comparable. When `1 < x` is evaluated, Python recognizes that the number 1 is a native number and that `x` is *not*. Because a native Python number does not know how to compare itself with a SymPy object Python will try the reflective operation, `x > 1` and that is the form that gets evaluated, hence returned.

If the order of the statement is important (for visual output to the console, perhaps), one can work around this annoyance in a couple ways:

(1) “sympify” the literal before comparison

```
>>> S(1) < x
1 < x
```

(2) use one of the wrappers or less succinct methods described above

```
>>> Lt(1, x)
1 < x
>>> Relational(1, x, "<")
1 < x
```

The second gotcha involves writing equality tests between relationals when one or both sides of the test involve a literal relational:

```
>>> e = x < 1; e
x < 1
>>> e == e # neither side is a literal
True
>>> e == x < 1 # expecting True, too
False
>>> e != x < 1 # expecting False
x < 1
>>> x < 1 != x < 1 # expecting False or the same thing as before
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
```

The solution for this case is to wrap literal relationals in parentheses:

```
>>> e == (x < 1)
True
>>> e != (x < 1)
False
>>> (x < 1) != (x < 1)
False
```

The third gotcha involves chained inequalities not involving `==` or `!=`. Occasionally, one may be tempted to write:

```
>>> e = x < y < z
Traceback (most recent call last):
...
TypeError: symbolic boolean expression has no truth value.
```

Due to an implementation detail or decision of Python [R132], there is no way for SymPy to create a chained inequality with that syntax so one must use `And`:

```
>>> e = And(x < y, y < z)
>>> type( e )
And
>>> e
(x < y) & (y < z)
```

Although this can also be done with the `'&'` operator, it cannot be done with the `'and'` operator:

```
>>> (x < y) & (y < z)
(x < y) & (y < z)
>>> (x < y) and (y < z)
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
```

Unequality

class `sympy.core.relational.Unequality(lhs, rhs, **options)`

An unequal relation between two objects.

Explanation

Represents that two objects are not equal. If they can be shown to be definitively equal, this will reduce to `False`; if definitively unequal, this will reduce to `True`. Otherwise, the relation is maintained as an `Unequality` object.

Examples

```
>>> from sympy import Ne
>>> from sympy.abc import x, y
>>> Ne(y, x+x**2)
Ne(y, x**2 + x)
```

Notes

This class is not the same as the `!=` operator. The `!=` operator tests for exact structural equality between two expressions; this class compares expressions mathematically.

This class is effectively the inverse of `Equality`. As such, it uses the same algorithms, including any available `evalEq` methods.

See also:

[`Equality`](#) (page 1023)

StrictGreaterThan

class `sympy.core.relational.StrictGreaterThan(lhs, rhs, **options)`

Class representations of inequalities.

Explanation

The `*Than` classes represent inequal relationships, where the left-hand side is generally bigger or smaller than the right-hand side. For example, the `GreaterThan` class represents an inequal relationship where the left-hand side is at least as big as the right side, if not bigger. In mathematical notation:

$\text{lhs} \geq \text{rhs}$

In total, there are four `*Than` classes, to represent the four inequalities:

Class Name	Symbol
<code>GreaterThan</code>	$\geq$
<code>LessThan</code>	$\leq$
<code>StrictGreaterThan</code>	$>$
<code>StrictLessThan</code>	$<$

All classes take two arguments, `lhs` and `rhs`.

Signature Example	Math Equivalent
<code>GreaterThan(lhs, rhs)</code>	$\text{lhs} \geq \text{rhs}$
<code>LessThan(lhs, rhs)</code>	$\text{lhs} \leq \text{rhs}$
<code>StrictGreaterThan(lhs, rhs)</code>	$\text{lhs} > \text{rhs}$
<code>StrictLessThan(lhs, rhs)</code>	$\text{lhs} < \text{rhs}$

In addition to the normal `.lhs` and `.rhs` of `Relations`, `*Than` inequality objects also have the `.lts` and `.gts` properties, which represent the “less than side” and “greater than side” of the operator. Use of `.lts` and `.gts` in an algorithm rather than `.lhs` and `.rhs` as an assumption of inequality direction will make more explicit the intent of a certain section of code, and will make it similarly more robust to client code changes:

```
>>> from sympy import GreaterThan, StrictGreaterThan
>>> from sympy import LessThan, StrictLessThan
>>> from sympy import And, Ge, Gt, Le, Lt, Rel, S
>>> from sympy.abc import x, y, z
>>> from sympy.core.relational import Relational
```



```
>>> e = GreaterThan(x, 1)
>>> e
x >= 1
>>> '%s >= %s is the same as %s <= %s' % (e.gts, e.lts, e.lts, e.gts)
'x >= 1 is the same as 1 <= x'
```

Examples

One generally does not instantiate these classes directly, but uses various convenience methods:

```
>>> for f in [Ge, Gt, Le, Lt]: # convenience wrappers
...     print(f(x, 2))
x >= 2
x > 2
x <= 2
x < 2
```

Another option is to use the Python inequality operators ($\geq$, $>$, $\leq$, $<$) directly. Their main advantage over the `Ge`, `Gt`, `Le`, and `Lt` counterparts, is that one can write a more “mathematical looking” statement rather than littering the math with oddball function calls. However there are certain (minor) caveats of which to be aware (search for ‘gotcha’, below).

```
>>> x >= 2
x >= 2
>>> _ == Ge(x, 2)
True
```

However, it is also perfectly valid to instantiate a `*Than` class less succinctly and less conveniently:

```
>>> Rel(x, 1, ">")
x > 1
>>> Relational(x, 1, ">")
x > 1
```

```
>>> StrictGreaterThan(x, 1)
x > 1
>>> GreaterThan(x, 1)
x >= 1
>>> LessThan(x, 1)
x <= 1
>>> StrictLessThan(x, 1)
x < 1
```

Notes

There are a couple of “gotchas” to be aware of when using Python’s operators.

The first is that what you write is not always what you get:

```
>>> 1 < x
x > 1
```

Due to the order that Python parses a statement, it may not immediately find two objects comparable. When `1 < x` is evaluated, Python recognizes that the number 1 is a native number and that `x` is *not*. Because a native Python number does not know how to compare itself with a SymPy object Python will try the reflective operation, `x > 1` and that is the form that gets evaluated, hence returned.

If the order of the statement is important (for visual output to the console, perhaps), one can work around this annoyance in a couple ways:

(1) “sympify” the literal before comparison

```
>>> S(1) < x
1 < x
```

(2) use one of the wrappers or less succinct methods described above

```
>>> Lt(1, x)
1 < x
>>> Relational(1, x, "<")
1 < x
```

The second gotcha involves writing equality tests between relationals when one or both sides of the test involve a literal relational:

```
>>> e = x < 1; e
x < 1
>>> e == e # neither side is a literal
True
>>> e == x < 1 # expecting True, too
False
>>> e != x < 1 # expecting False
x < 1
>>> x < 1 != x < 1 # expecting False or the same thing as before
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
```

The solution for this case is to wrap literal relationals in parentheses:

```
>>> e == (x < 1)
True
>>> e != (x < 1)
False
>>> (x < 1) != (x < 1)
False
```

The third gotcha involves chained inequalities not involving `==` or `!=`. Occasionally, one may be tempted to write:

```
>>> e = x < y < z
Traceback (most recent call last):
...
TypeError: symbolic boolean expression has no truth value.
```

Due to an implementation detail or decision of Python [R133], there is no way for SymPy to create a chained inequality with that syntax so one must use `And`:

```
>>> e = And(x < y, y < z)
>>> type( e )
And
>>> e
(x < y) & (y < z)
```

Although this can also be done with the `'&'` operator, it cannot be done with the `'and'` operator:

```
>>> (x < y) & (y < z)
(x < y) & (y < z)
>>> (x < y) and (y < z)
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
```

StrictLessThan

class `sympy.core.relational.StrictLessThan(lhs, rhs, **options)`

Class representations of inequalities.

Explanation

The `*Than` classes represent unequal relationships, where the left-hand side is generally bigger or smaller than the right-hand side. For example, the `GreaterThan` class represents an unequal relationship where the left-hand side is at least as big as the right side, if not bigger. In mathematical notation:

$$\text{lhs} \geq \text{rhs}$$

In total, there are four `*Than` classes, to represent the four inequalities:

Class Name	Symbol
<code>GreaterThan</code>	<code>>=</code>
<code>LessThan</code>	<code><=</code>
<code>StrictGreaterThan</code>	<code>></code>
<code>StrictLessThan</code>	<code><</code>

All classes take two arguments, `lhs` and `rhs`.

Signature	Example	Math Equivalent
GreaterThan	(lhs, rhs)	$\text{lhs} \geq \text{rhs}$
LessThan	(lhs, rhs)	$\text{lhs} \leq \text{rhs}$
StrictGreaterThan	(lhs, rhs)	$\text{lhs} > \text{rhs}$
StrictLessThan	(lhs, rhs)	$\text{lhs} < \text{rhs}$

In addition to the normal `.lhs` and `.rhs` of Relations, `*Than` inequality objects also have the `.lts` and `.gts` properties, which represent the “less than side” and “greater than side” of the operator. Use of `.lts` and `.gts` in an algorithm rather than `.lhs` and `.rhs` as an assumption of inequality direction will make more explicit the intent of a certain section of code, and will make it similarly more robust to client code changes:

```
>>> from sympy import GreaterThan, StrictGreaterThan
>>> from sympy import LessThan, StrictLessThan
>>> from sympy import And, Ge, Gt, Le, Lt, Rel, S
>>> from sympy.abc import x, y, z
>>> from sympy.core.relational import Relational
```

```
>>> e = GreaterThan(x, 1)
>>> e
x >= 1
>>> '%s >= %s is the same as %s <= %s' % (e.gts, e.lts, e.lts, e.gts)
'x >= 1 is the same as 1 <= x'
```

Examples

One generally does not instantiate these classes directly, but uses various convenience methods:

```
>>> for f in [Ge, Gt, Le, Lt]: # convenience wrappers
...     print(f(x, 2))
x >= 2
x > 2
x <= 2
x < 2
```

Another option is to use the Python inequality operators (`>=`, `>`, `<=`, `<`) directly. Their main advantage over the `Ge`, `Gt`, `Le`, and `Lt` counterparts, is that one can write a more “mathematical looking” statement rather than littering the math with oddball function calls. However there are certain (minor) caveats of which to be aware (search for ‘gotcha’, below).

```
>>> x >= 2
x >= 2
>>> _ == Ge(x, 2)
True
```

However, it is also perfectly valid to instantiate a `*Than` class less succinctly and less conveniently:

```
>>> Rel(x, 1, ">")
x > 1
>>> Relational(x, 1, ">")
x > 1
```

```
>>> StrictGreaterThan(x, 1)
x > 1
>>> GreaterThan(x, 1)
x >= 1
>>> LessThan(x, 1)
x <= 1
>>> StrictLessThan(x, 1)
x < 1
```

Notes

There are a couple of “gotchas” to be aware of when using Python’s operators.

The first is that what you write is not always what you get:

```
>>> 1 < x
x > 1
```

Due to the order that Python parses a statement, it may not immediately find two objects comparable. When `1 < x` is evaluated, Python recognizes that the number 1 is a native number and that `x` is *not*. Because a native Python number does not know how to compare itself with a SymPy object Python will try the reflective operation, `x > 1` and that is the form that gets evaluated, hence returned.

If the order of the statement is important (for visual output to the console, perhaps), one can work around this annoyance in a couple ways:

(1) “sympify” the literal before comparison

```
>>> S(1) < x
1 < x
```

(2) use one of the wrappers or less succinct methods described above

```
>>> Lt(1, x)
1 < x
>>> Relational(1, x, "<")
1 < x
```

The second gotcha involves writing equality tests between relationals when one or both sides of the test involve a literal relational:

```
>>> e = x < 1; e
x < 1
>>> e == e # neither side is a literal
True
>>> e == x < 1 # expecting True, too
```

(continues on next page)

(continued from previous page)

```
False
>>> e != x < 1 # expecting False
x < 1
>>> x < 1 != x < 1 # expecting False or the same thing as before
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
```

The solution for this case is to wrap literal relationals in parentheses:

```
>>> e == (x < 1)
True
>>> e != (x < 1)
False
>>> (x < 1) != (x < 1)
False
```

The third gotcha involves chained inequalities not involving `==` or `!=`. Occasionally, one may be tempted to write:

```
>>> e = x < y < z
Traceback (most recent call last):
...
TypeError: symbolic boolean expression has no truth value.
```

Due to an implementation detail or decision of Python [R134], there is no way for SymPy to create a chained inequality with that syntax so one must use `And`:

```
>>> e = And(x < y, y < z)
>>> type( e )
And
>>> e
(x < y) & (y < z)
```

Although this can also be done with the `'&'` operator, it cannot be done with the `'and'` operator:

```
>>> (x < y) & (y < z)
(x < y) & (y < z)
>>> (x < y) and (y < z)
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
```

multidimensional

vectorize

class `sympy.core.multidimensional.vectorize(*mdargs)`

Generalizes a function taking scalars to accept multidimensional arguments.

Examples

```
>>> from sympy import vectorize, diff, sin, symbols, Function
>>> x, y, z = symbols('x y z')
>>> f, g, h = list(map(Function, 'fgh'))
```

```
>>> @vectorize(0)
... def vsin(x):
...     return sin(x)
```

```
>>> vsin([1, x, y])
[sin(1), sin(x), sin(y)]
```

```
>>> @vectorize(0, 1)
... def vdiff(f, y):
...     return diff(f, y)
```

```
>>> vdiff([f(x, y, z), g(x, y, z), h(x, y, z)], [x, y, z])
[[Derivative(f(x, y, z), x), Derivative(f(x, y, z), y), Derivative(f(x,
→y, z), z)], [Derivative(g(x, y, z), x), Derivative(g(x, y, z), y),
→Derivative(g(x, y, z), z)], [Derivative(h(x, y, z), x), Derivative(h(x,
→y, z), y), Derivative(h(x, y, z), z)]]
```

function

Lambda

class `sympy.core.function.Lambda(signature, expr)`

`Lambda(x, expr)` represents a lambda function similar to Python's 'lambda x: expr'. A function of several variables is written as `Lambda((x, y, ...), expr)`.

Examples

A simple example:

```
>>> from sympy import Lambda
>>> from sympy.abc import x
>>> f = Lambda(x, x**2)
>>> f(4)
16
```

For multivariate functions, use:

```
>>> from sympy.abc import y, z, t
>>> f2 = Lambda((x, y, z, t), x + y**z + t**z)
>>> f2(1, 2, 3, 4)
73
```

It is also possible to unpack tuple arguments:

```
>>> f = Lambda(((x, y), z), x + y + z)
>>> f((1, 2), 3)
6
```

A handy shortcut for lots of arguments:

```
>>> p = x, y, z
>>> f = Lambda(p, x + y*z)
>>> f(*p)
x + y*z
```

property bound_symbols

The variables used in the internal representation of the function

property expr

The return value of the function

property is_identity

Return True if this Lambda is an identity function.

property signature

The expected form of the arguments to be unpacked into variables

property variables

The variables used in the internal representation of the function

WildFunction

```
class sympy.core.function.WildFunction(*args)
```

A WildFunction function matches any function (with its arguments).

Examples

```
>>> from sympy import WildFunction, Function, cos
>>> from sympy.abc import x, y
>>> F = WildFunction('F')
>>> f = Function('f')
>>> F.nargs
Naturals0
>>> x.match(F)
>>> F.match(F)
{F_: F_}
>>> f(x).match(F)
{F_: f(x)}
>>> cos(x).match(F)
{F_: cos(x)}
>>> f(x, y).match(F)
{F_: f(x, y)}
```

To match functions with a given number of arguments, set nargs to the desired value at instantiation:

```
>>> F = WildFunction('F', nargs=2)
>>> F.nargs
{2}
>>> f(x).match(F)
>>> f(x, y).match(F)
{F_: f(x, y)}
```

To match functions with a range of arguments, set nargs to a tuple containing the desired number of arguments, e.g. if nargs = (1, 2) then functions with 1 or 2 arguments will be matched.

```
>>> F = WildFunction('F', nargs=(1, 2))
>>> F.nargs
{1, 2}
>>> f(x).match(F)
{F_: f(x)}
>>> f(x, y).match(F)
{F_: f(x, y)}
>>> f(x, y, 1).match(F)
```

Derivative

class sympy.core.function.Derivative(*expr*, **variables*, ***kwargs*)

Carries out differentiation of the given expression with respect to symbols.

Examples

```
>>> from sympy import Derivative, Function, symbols, Subs
>>> from sympy.abc import x, y
>>> f, g = symbols('f g', cls=Function)
```

```
>>> Derivative(x**2, x, evaluate=True)
2*x
```

Denesting of derivatives retains the ordering of variables:

```
>>> Derivative(Derivative(f(x, y), y), x)
Derivative(f(x, y), y, x)
```

Contiguously identical symbols are merged into a tuple giving the symbol and the count:

```
>>> Derivative(f(x), x, x, y, x)
Derivative(f(x), (x, 2), y, x)
```

If the derivative cannot be performed, and evaluate is True, the order of the variables of differentiation will be made canonical:

```
>>> Derivative(f(x, y), y, x, evaluate=True)
Derivative(f(x, y), x, y)
```

Derivatives with respect to undefined functions can be calculated:

```
>>> Derivative(f(x)**2, f(x), evaluate=True)
2*f(x)
```

Such derivatives will show up when the chain rule is used to evaluate a derivative:

```
>>> f(g(x)).diff(x)
Derivative(f(g(x)), g(x))*Derivative(g(x), x)
```

Substitution is used to represent derivatives of functions with arguments that are not symbols or functions:

```
>>> f(2*x + 3).diff(x) == 2*Subs(f(y).diff(y), y, 2*x + 3)
True
```

Notes

Simplification of high-order derivatives:

Because there can be a significant amount of simplification that can be done when multiple differentiations are performed, results will be automatically simplified in a fairly conservative fashion unless the keyword `simplify` is set to `False`.

```
>>> from sympy import sqrt, diff, Function, symbols
>>> from sympy.abc import x, y, z
>>> f, g = symbols('f,g', cls=Function)
```

```
>>> e = sqrt((x + 1)**2 + x)
>>> diff(e, (x, 5), simplify=False).count_ops()
136
>>> diff(e, (x, 5)).count_ops()
30
```

Ordering of variables:

If `evaluate` is set to `True` and the expression cannot be evaluated, the list of differentiation symbols will be sorted, that is, the expression is assumed to have continuous derivatives up to the order asked.

Derivative wrt non-Symbols:

For the most part, one may not differentiate wrt non-symbols. For example, we do not allow differentiation wrt $x * y$ because there are multiple ways of structurally defining where $x*y$ appears in an expression: a very strict definition would make $(x*y*z).diff(x*y) == 0$. Derivatives wrt defined functions (like $\cos(x)$) are not allowed, either:

```
>>> (x*y*z).diff(x*y)
Traceback (most recent call last):
...
ValueError: Can't calculate derivative wrt x*y.
```

To make it easier to work with variational calculus, however, derivatives wrt `AppliedUnDef` and `Derivatives` are allowed. For example, in the Euler-Lagrange method one may write $F(t, u, v)$ where $u = f(t)$ and $v = f'(t)$. These variables can be written explicitly as functions of time:

```
>>> from sympy.abc import t
>>> F = Function('F')
>>> U = f(t)
>>> V = U.diff(t)
```

The derivative wrt $f(t)$ can be obtained directly:

```
>>> direct = F(t, U, V).diff(U)
```

When differentiation wrt a non-Symbol is attempted, the non-Symbol is temporarily converted to a Symbol while the differentiation is performed and the same answer is obtained:

```
>>> indirect = F(t, U, V).subs(U, x).diff(x).subs(x, U)
>>> assert direct == indirect
```

The implication of this non-symbol replacement is that all functions are treated as independent of other functions and the symbols are independent of the functions that contain them:

```
>>> x.diff(f(x))
0
>>> g(x).diff(f(x))
0
```

It also means that derivatives are assumed to depend only on the variables of differentiation, not on anything contained within the expression being differentiated:

```
>>> F = f(x)
>>> Fx = F.diff(x)
>>> Fx.diff(F)  # derivative depends on x, not F
0
>>> Fxx = Fx.diff(x)
>>> Fxx.diff(Fx)  # derivative depends on x, not Fx
0
```

The last example can be made explicit by showing the replacement of Fx in Fxx with y :

```
>>> Fxx.subs(Fx, y)
Derivative(y, x)
```

Since that in itself will evaluate to zero, differentiating wrt Fx will also be zero:

```
>>> _.doit()
0
```

Replacing undefined functions with concrete expressions

One must be careful to replace undefined functions with expressions that contain variables consistent with the function definition and the variables of differentiation or else inconsistent result will be obtained. Consider the following example:

```
>>> eq = f(x)*g(y)
>>> eq.subs(f(x), x*y).diff(x, y).doit()
y*Derivative(g(y), y) + g(y)
>>> eq.diff(x, y).subs(f(x), x*y).doit()
y*Derivative(g(y), y)
```

The results differ because $f(x)$ was replaced with an expression that involved both variables of differentiation. In the abstract case, differentiation of $f(x)$ by y is 0; in the concrete case, the presence of y made that derivative nonvanishing and produced the extra $g(y)$ term.

Defining differentiation for an object

An object must define `._eval_derivative(symbol)` method that returns the differentiation result. This function only needs to consider the non-trivial case where `expr` contains `symbol` and it should call the `diff()` method internally (not `._eval_derivative`); `Derivative` should be the only one to call `._eval_derivative`.

Any class can allow derivatives to be taken with respect to itself (while indicating its scalar nature). See the docstring of `Expr._diff_wrt`.

See also:[`\_sort\_variable\_count`](#) (page 1045)**property `_diff_wrt`**

An expression may be differentiated wrt a Derivative if it is in elementary form.

Examples

```
>>> from sympy import Function, Derivative, cos
>>> from sympy.abc import x
>>> f = Function('f')
```

```
>>> Derivative(f(x), x)._diff_wrt
True
>>> Derivative(cos(x), x)._diff_wrt
False
>>> Derivative(x + 1, x)._diff_wrt
False
```

A Derivative might be an unevaluated form of what will not be a valid variable of differentiation if evaluated. For example,

```
>>> Derivative(f(f(x)), x).doit()
Derivative(f(x), x)*Derivative(f(f(x)), f(x))
```

Such an expression will present the same ambiguities as arise when dealing with any other product, like $2*x$, so `_diff_wrt` is False:

```
>>> Derivative(f(f(x)), x)._diff_wrt
False
```

classmethod `_sort_variable_count(vc)`

Sort (variable, count) pairs into canonical order while retaining order of variables that do not commute during differentiation:

- symbols and functions commute with each other
- derivatives commute with each other
- a derivative does not commute with anything it contains
- any other object is not allowed to commute if it has free symbols in common with another object

Examples

```
>>> from sympy import Derivative, Function, symbols
>>> vsort = Derivative._sort_variable_count
>>> x, y, z = symbols('x y z')
>>> f, g, h = symbols('f g h', cls=Function)
```

Contiguous items are collapsed into one pair:

```
>>> vsort([(x, 1), (x, 1)])
[(x, 2)]
>>> vsort([(y, 1), (f(x), 1), (y, 1), (f(x), 1)])
[(y, 2), (f(x), 2)]
```

Ordering is canonical.

```
>>> def vsort0(*v):
...     # docstring helper to
...     # change vi -> (vi, 0), sort, and return vi vals
...     return [i[0] for i in vsort([(i, 0) for i in v])]
```

```
>>> vsort0(y, x)
[x, y]
>>> vsort0(g(y), g(x), f(y))
[f(y), g(x), g(y)]
```

Symbols are sorted as far to the left as possible but never move to the left of a derivative having the same symbol in its variables; the same applies to AppliedUndef which are always sorted after Symbols:

```
>>> dfx = f(x).diff(x)
>>> assert vsort0(dfx, y) == [y, dfx]
>>> assert vsort0(dfx, x) == [dfx, x]
```

as_finite_difference(points=1, x0=None, wrt=None)

Expresses a Derivative instance as a finite difference.

Parameters

points : sequence or coefficient, optional

If sequence: discrete values (length $\geq$ order+1) of the independent variable used for generating the finite difference weights. If it is a coefficient, it will be used as the step-size for generating an equidistant sequence of length order+1 centered around x0. Default: 1 (step-size 1)

x0 : number or Symbol, optional

the value of the independent variable (wrt) at which the derivative is to be approximated. Default: same as wrt.

wrt : Symbol, optional

“with respect to” the variable for which the (partial) derivative is to be approximated for. If not provided it is required that the derivative is ordinary. Default: None.

Examples

```
>>> from sympy import symbols, Function, exp, sqrt, Symbol
>>> x, h = symbols('x h')
>>> f = Function('f')
>>> f(x).diff(x).as_finite_difference()
-f(x - 1/2) + f(x + 1/2)
```

The default step size and number of points are 1 and order + 1 respectively. We can change the step size by passing a symbol as a parameter:

```
>>> f(x).diff(x).as_finite_difference(h)
-f(-h/2 + x)/h + f(h/2 + x)/h
```

We can also specify the discretized values to be used in a sequence:

```
>>> f(x).diff(x).as_finite_difference([x, x+h, x+2*h])
-3*f(x)/(2*h) + 2*f(h + x)/h - f(2*h + x)/(2*h)
```

The algorithm is not restricted to use equidistant spacing, nor do we need to make the approximation around x_0 , but we can get an expression estimating the derivative at an offset:

```
>>> e, sq2 = exp(1), sqrt(2)
>>> xl = [x-h, x+h, x+e*h]
>>> f(x).diff(x, 1).as_finite_difference(xl, x+h*sq2)
2*h*((h + sqrt(2)*h)/(2*h) - (-sqrt(2)*h + h)/(2*h))*f(E*h + x)/...
```

To approximate Derivative around x_0 using a non-equidistant spacing step, the algorithm supports assignment of undefined functions to points:

```
>>> dx = Function('dx')
>>> f(x).diff(x).as_finite_difference(points=dx(x), x0=x-h)
-f(-h + x - dx(-h + x)/2)/dx(-h + x) + f(-h + x + dx(-h + x)/2)/dx(-h
↪ + x)
```

Partial derivatives are also supported:

```
>>> y = Symbol('y')
>>> d2fdxdy=f(x,y).diff(x,y)
>>> d2fdxdy.as_finite_difference(wrt=x)
-Derivative(f(x - 1/2, y), y) + Derivative(f(x + 1/2, y), y)
```

We can apply `as_finite_difference` to Derivative instances in compound expressions using `replace`:

```
>>> (1 + 42*f(x).diff(x)).replace(lambda arg: arg.is_Derivative,
... lambda arg: arg.as_finite_difference())
42*(-f(x - 1/2) + f(x + 1/2)) + 1
```

See also:

[`sympy.calculus.finite\_diff.apply\_finite\_diff`](#) (page 240), [`sympy.calculus.finite\_diff.differentiate\_finite`](#) (page 241), [`sympy.calculus.finite\_diff.finite\_diff\_weights`](#) (page 242)

doit_numerically(*z0*)

Evaluate the derivative at *z* numerically.

When we can represent derivatives at a point, this should be folded into the normal evalf. For now, we need a special method.

diff

`sympy.core.function.diff(f, *symbols, **kwargs)`

Differentiate *f* with respect to *symbols*.

Explanation

This is just a wrapper to unify `.diff()` and the Derivative class; its interface is similar to that of `integrate()`. You can use the same shortcuts for multiple variables as with Derivative. For example, `diff(f(x), x, x, x)` and `diff(f(x), x, 3)` both return the third derivative of *f*(*x*).

You can pass `evaluate=False` to get an unevaluated Derivative class. Note that if there are 0 symbols (such as `diff(f(x), x, 0)`), then the result will be the function (the zeroth derivative), even if `evaluate=False`.

Examples

```
>>> from sympy import sin, cos, Function, diff
>>> from sympy.abc import x, y
>>> f = Function('f')
```

```
>>> diff(sin(x), x)
cos(x)
>>> diff(f(x), x, x, x)
Derivative(f(x), (x, 3))
>>> diff(f(x), x, 3)
Derivative(f(x), (x, 3))
>>> diff(sin(x)*cos(y), x, 2, y, 2)
sin(x)*cos(y)
```

```
>>> type(diff(sin(x), x))
cos
>>> type(diff(sin(x), x, evaluate=False))
<class 'sympy.core.function.Derivative'>
>>> type(diff(sin(x), x, 0))
sin
>>> type(diff(sin(x), x, 0, evaluate=False))
sin
```

```
>>> diff(sin(x))
cos(x)
>>> diff(sin(x*y))
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: specify differentiation variables to differentiate sin(x*y)
```

Note that `diff(sin(x))` syntax is meant only for convenience in interactive sessions and should be avoided in library code.

See also:

[Derivative](#) (page 1042)

`idiff` (page 2201)

computes the derivative implicitly

References

[R135]

FunctionClass

class `sympy.core.function.FunctionClass(*args, **kwargs)`

Base class for function classes. `FunctionClass` is a subclass of `type`.

Use `Function('<function name>' [, signature ])` to create undefined function classes.

property nargs

Return a set of the allowed number of arguments for the function.

Examples

```
>>> from sympy import Function
>>> f = Function('f')
```

If the function can take any number of arguments, the set of whole numbers is returned:

```
>>> Function('f').nargs
Naturals0
```

If the function was initialized to accept one or more arguments, a corresponding set will be returned:

```
>>> Function('f', nargs=1).nargs
{1}
>>> Function('f', nargs=(2, 1)).nargs
{1, 2}
```

The undefined function, after application, also has the `nargs` attribute; the actual number of arguments is always available by checking the `args` attribute:

```
>>> f = Function('f')
>>> f(1).nargs
Naturals0
>>> len(f(1).args)
1
```

Function

class `sympy.core.function.Function(*args)`

Base class for applied mathematical functions.

It also serves as a constructor for undefined function classes.

See the [Writing Custom Functions](#) (page 102) guide for details on how to subclass `Function` and what methods can be defined.

Examples

Undefined Functions

To create an undefined function, pass a string of the function name to `Function`.

```
>>> from sympy import Function, Symbol
>>> x = Symbol('x')
>>> f = Function('f')
>>> g = Function('g')(x)
>>> f
f
>>> f(x)
f(x)
>>> g
g(x)
>>> f(x).diff(x)
Derivative(f(x), x)
>>> g.diff(x)
Derivative(g(x), x)
```

Assumptions can be passed to `Function` the same as with a [Symbol](#) (page 976). Alternatively, you can use a `Symbol` with assumptions for the function name and the function will inherit the name and assumptions associated with the `Symbol`:

```
>>> f_real = Function('f', real=True)
>>> f_real(x).is_real
True
>>> f_real_inherit = Function(Symbol('f', real=True))
>>> f_real_inherit(x).is_real
True
```

Note that assumptions on a function are unrelated to the assumptions on the variables it is called on. If you want to add a relationship, subclass `Function` and define custom assumptions handler methods. See the [Assumptions](#) (page 111) section of the [Writing Custom Functions](#) (page 102) guide for more details.

Custom Function Subclasses

The [Writing Custom Functions](#) (page 102) guide has several [Complete Examples](#) (page 122) of how to subclass `Function` to create a custom function.

as_base_exp()

Returns the method as the 2-tuple (base, exponent).

fdiff(*argindex*=1)

Returns the first derivative of the function.

classmethod is_singular(*a*)

Tests whether the argument is an essential singularity or a branch point, or the functions is non-holomorphic.

Note: Not all functions are the same

SymPy defines many functions (like `cos` and `factorial`). It also allows the user to create generic functions which act as argument holders. Such functions are created just like symbols:

```
>>> from sympy import Function, cos
>>> from sympy.abc import x
>>> f = Function('f')
>>> f(2) + f(x)
f(2) + f(x)
```

If you want to see which functions appear in an expression you can use the `atoms` method:

```
>>> e = (f(x) + cos(x) + 2)
>>> e.atoms(Function)
{f(x), cos(x)}
```

If you just want the function you defined, not SymPy functions, the thing to search for is `AppliedUndef`:

```
>>> from sympy.core.function import AppliedUndef
>>> e.atoms(AppliedUndef)
{f(x)}
```

Subs

class `sympy.core.function.Subs(expr, variables, point, **assumptions)`

Represents unevaluated substitutions of an expression.

`Subs(expr, x, x0)` represents the expression resulting from substituting *x* with *x0* in *expr*.

Parameters

expr : Expr

An expression.

x : tuple, variable

A variable or list of distinct variables.

x0 : tuple or list of tuples

A point or list of evaluation points corresponding to those variables.

Notes

In order to allow expressions to combine before `doit` is done, a representation of the Subs expression is used internally to make expressions that are superficially different compare the same:

```
>>> a, b = Subs(x, x, 0), Subs(y, y, 0)
>>> a + b
2*Subs(x, x, 0)
```

This can lead to unexpected consequences when using methods like `has` that are cached:

```
>>> s = Subs(x, x, 0)
>>> s.has(x), s.has(y)
(True, False)
>>> ss = s.subs(x, y)
>>> ss.has(x), ss.has(y)
(True, False)
>>> s, ss
(Subs(x, x, 0), Subs(y, y, 0))
```

Examples

```
>>> from sympy import Subs, Function, sin, cos
>>> from sympy.abc import x, y, z
>>> f = Function('f')
```

Subs are created when a particular substitution cannot be made. The `x` in the derivative cannot be replaced with `0` because `0` is not a valid variables of differentiation:

```
>>> f(x).diff(x).subs(x, 0)
Subs(Derivative(f(x), x), x, 0)
```

Once `f` is known, the derivative and evaluation at `0` can be done:

```
>>> _.subs(f, sin).doit() == sin(x).diff(x).subs(x, 0) == cos(0)
True
```

Subs can also be created directly with one or more variables:

```
>>> Subs(f(x)*sin(y) + z, (x, y), (0, 1))
Subs(z + f(x)*sin(y), (x, y), (0, 1))
>>> _.doit()
z + f(0)*sin(1)
```

property `bound_symbols`

The variables to be evaluated

property expr

The expression on which the substitution operates

property point

The values for which the variables are to be substituted

property variables

The variables to be evaluated

expand

```
sympy.core.function.expand(e, deep=True, modulus=None, power_base=True,  
                             power_exp=True, mul=True, log=True, multinomial=True,  
                             basic=True, **hints)
```

Expand an expression using methods given as hints.

Explanation

Hints evaluated unless explicitly set to False are: `basic`, `log`, `multinomial`, `mul`, `power_base`, and `power_exp`. The following hints are supported but not applied unless set to True: `complex`, `func`, and `trig`. In addition, the following meta-hints are supported by some or all of the other hints: `frac`, `numer`, `denom`, `modulus`, and `force`. `deep` is supported by all hints. Additionally, subclasses of `Expr` may define their own hints or meta-hints.

The `basic` hint is used for any special rewriting of an object that should be done automatically (along with the other hints like `mul`) when `expand` is called. This is a catch-all hint to handle any sort of expansion that may not be described by the existing hint names. To use this hint an object should override the `_eval_expand_basic` method. Objects may also define their own `expand` methods, which are not run by default. See the API section below.

If `deep` is set to True (the default), things like arguments of functions are recursively expanded. Use `deep=False` to only expand on the top level.

If the `force` hint is used, assumptions about variables will be ignored in making the expansion.

Hints

These hints are run by default

Mul

Distributes multiplication over addition:

```
>>> from sympy import cos, exp, sin
>>> from sympy.abc import x, y, z
>>> (y*(x + z)).expand(mul=True)
x*y + y*z
```

Multinomial

Expand $(x + y + \dots)^n$ where n is a positive integer.

```
>>> ((x + y + z)**2).expand(multinomial=True)
x**2 + 2*x*y + 2*x*z + y**2 + 2*y*z + z**2
```

Power_exp

Expand addition in exponents into multiplied bases.

```
>>> exp(x + y).expand(power_exp=True)
exp(x)*exp(y)
>>> (2**(x + y)).expand(power_exp=True)
2**x*2**y
```

Power_base

Split powers of multiplied bases.

This only happens by default if assumptions allow, or if the `force` meta-hint is used:

```
>>> ((x*y)**z).expand(power_base=True)
(x*y)**z
>>> ((x*y)**z).expand(power_base=True, force=True)
x**z*y**z
>>> ((2*y)**z).expand(power_base=True)
2**z*y**z
```

Note that in some cases where this expansion always holds, SymPy performs it automatically:

```
>>> (x*y)**2
x**2*y**2
```

Log

Pull out power of an argument as a coefficient and split logs products into sums of logs.

Note that these only work if the arguments of the log function have the proper assumptions-the arguments must be positive and the exponents must be real-or else the force hint must be True:

```
>>> from sympy import log, symbols
>>> log(x**2*y).expand(log=True)
log(x**2*y)
>>> log(x**2*y).expand(log=True, force=True)
2*log(x) + log(y)
>>> x, y = symbols('x,y', positive=True)
>>> log(x**2*y).expand(log=True)
2*log(x) + log(y)
```

Basic

This hint is intended primarily as a way for custom subclasses to enable expansion by default.

These hints are not run by default:

Complex

Split an expression into real and imaginary parts.

```
>>> x, y = symbols('x,y')
>>> (x + y).expand(complex=True)
re(x) + re(y) + I*im(x) + I*im(y)
>>> cos(x).expand(complex=True)
-I*sin(re(x))*sinh(im(x)) + cos(re(x))*cosh(im(x))
```

Note that this is just a wrapper around `as_real_imag()`. Most objects that wish to redefine `_eval_expand_complex()` should consider redefining `as_real_imag()` instead.

Func

Expand other functions.

```
>>> from sympy import gamma
>>> gamma(x + 1).expand(func=True)
x*gamma(x)
```

Trig

Do trigonometric expansions.

```
>>> cos(x + y).expand(trig=True)
-sin(x)*sin(y) + cos(x)*cos(y)
>>> sin(2*x).expand(trig=True)
2*sin(x)*cos(x)
```

Note that the forms of $\sin(n*x)$ and $\cos(n*x)$ in terms of $\sin(x)$ and $\cos(x)$ are not unique, due to the identity $\sin^2(x) + \cos^2(x) = 1$. The current implementation uses the form obtained from Chebyshev polynomials, but this may change. See [this MathWorld article](#) for more information.

Notes

- You can shut off unwanted methods:

```
>>> (exp(x + y)*(x + y)).expand()
x*exp(x)*exp(y) + y*exp(x)*exp(y)
>>> (exp(x + y)*(x + y)).expand(power_exp=False)
x*exp(x + y) + y*exp(x + y)
>>> (exp(x + y)*(x + y)).expand(mul=False)
(x + y)*exp(x)*exp(y)
```

- Use `deep=False` to only expand on the top level:

```
>>> exp(x + exp(x + y)).expand()
exp(x)*exp(exp(x)*exp(y))
>>> exp(x + exp(x + y)).expand(deep=False)
exp(x)*exp(exp(x + y))
```

- Hints are applied in an arbitrary, but consistent order (in the current implementation, they are applied in alphabetical order, except `multinomial` comes before `mul`, but this may change). Because of this, some hints may prevent expansion by other hints if they are applied first. For example, `mul` may distribute multiplications and prevent `log` and `power_base` from expanding them. Also, if `mul` is applied before `multinomial`, the expression might not be fully distributed. The solution is to use the various `expand_hint` helper functions or to use `hint=False` to this function to finely control which hints are applied. Here are some examples:

```
>>> from sympy import expand, expand_mul, expand_power_base
>>> x, y, z = symbols('x,y,z', positive=True)

>>> expand(log(x*(y + z)))
log(x) + log(y + z)
```

Here, we see that `log` was applied before `mul`. To get the `mul` expanded form, either of the following will work:

```
>>> expand_mul(log(x*(y + z)))
log(x*y + x*z)
```

(continues on next page)

(continued from previous page)

```
>>> expand(log(x*(y + z)), log=False)
log(x*y + x*z)
```

A similar thing can happen with the `power_base` hint:

```
>>> expand((x*(y + z))**x)
(x*y + x*z)**x
```

To get the `power_base` expanded form, either of the following will work:

```
>>> expand((x*(y + z))**x, mul=False)
x**x*(y + z)**x
>>> expand_power_base((x*(y + z))**x)
x**x*(y + z)**x

>>> expand((x + y)*y/x)
y + y**2/x
```

The parts of a rational expression can be targeted:

```
>>> expand((x + y)*y/x/(x + 1), frac=True)
(x*y + y**2)/(x**2 + x)
>>> expand((x + y)*y/x/(x + 1), numer=True)
(x*y + y**2)/(x*(x + 1))
>>> expand((x + y)*y/x/(x + 1), denom=True)
y*(x + y)/(x**2 + x)
```

- The modulus meta-hint can be used to reduce the coefficients of an expression post-expansion:

```
>>> expand((3*x + 1)**2)
9*x**2 + 6*x + 1
>>> expand((3*x + 1)**2, modulus=5)
4*x**2 + x + 1
```

- Either `expand()` the function or `.expand()` the method can be used. Both are equivalent:

```
>>> expand((x + 1)**2)
x**2 + 2*x + 1
>>> ((x + 1)**2).expand()
x**2 + 2*x + 1
```

Api

Objects can define their own expand hints by defining `_eval_expand_hint()`. The function should take the form:

```
def _eval_expand_hint(self, **hints):  
    # Only apply the method to the top-level expression  
    ...
```

See also the example below. Objects should define `_eval_expand_hint()` methods only if hint applies to that specific object. The generic `_eval_expand_hint()` method defined in `Expr` will handle the no-op case.

Each hint should be responsible for expanding that hint only. Furthermore, the expansion should be applied to the top-level expression only. `expand()` takes care of the recursion that happens when `deep=True`.

You should only call `_eval_expand_hint()` methods directly if you are 100% sure that the object has the method, as otherwise you are liable to get unexpected `AttributeError`'s. Note, again, that you do not need to recursively apply the hint to args of your object: this is handled automatically by `expand()`. `_eval_expand_hint()` should generally not be used at all outside of an `_eval_expand_hint()` method. If you want to apply a specific expansion from within another method, use the public `expand()` function, method, or `expand_hint()` functions.

In order for `expand` to work, objects must be rebuildable by their args, i.e., `obj.func(*obj.args) == obj` must hold.

Expand methods are passed `**hints` so that expand hints may use 'metahints'-hints that control how different expand methods are applied. For example, the `force=True` hint described above that causes `expand(log=True)` to ignore assumptions is such a metahint. The deep meta-hint is handled exclusively by `expand()` and is not passed to `_eval_expand_hint()` methods.

Note that expansion hints should generally be methods that perform some kind of 'expansion'. For hints that simply rewrite an expression, use the `.rewrite()` API.

Examples

```
>>> from sympy import Expr, sympify  
>>> class MyClass(Expr):  
...     def __new__(cls, *args):  
...         args = sympify(args)  
...         return Expr.__new__(cls, *args)  
...  
...     def _eval_expand_double(self, *, force=False, **hints):  
...         '''  
...         Doubles the args of MyClass.  
...  
...         If there more than four args, doubling is not performed,  
...         unless force=True is also used (False by default).  
...         '''  
...         if not force and len(self.args) > 4:  
...             return self
```

(continues on next page)

(continued from previous page)

```

...         return self.func(*(self.args + self.args))
...
>>> a = MyClass(1, 2, MyClass(3, 4))
>>> a
MyClass(1, 2, MyClass(3, 4))
>>> a.expand(double=True)
MyClass(1, 2, MyClass(3, 4, 3, 4), 1, 2, MyClass(3, 4, 3, 4))
>>> a.expand(double=True, deep=False)
MyClass(1, 2, MyClass(3, 4), 1, 2, MyClass(3, 4))

```

```

>>> b = MyClass(1, 2, 3, 4, 5)
>>> b.expand(double=True)
MyClass(1, 2, 3, 4, 5)
>>> b.expand(double=True, force=True)
MyClass(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)

```

See also:

[expand_log](#) (page 1061), [expand_mul](#) (page 1061), [expand_multinomial](#) (page 1062), [expand_complex](#) (page 1062), [expand_trig](#) (page 1062), [expand_power_base](#) (page 1063), [expand_power_exp](#) (page 1063), [expand_func](#) (page 1061), [sympy.simplify.hyperexpand.hyperexpand](#) (page 687)

PoleError

class sympy.core.function.PoleError

count_ops

sympy.core.function.count_ops(*expr*, *visual=False*)

Return a representation (integer or expression) of the operations in *expr*.

Parameters

expr : Expr

If *expr* is an iterable, the sum of the op counts of the items will be returned.

visual : bool, optional

If False (default) then the sum of the coefficients of the visual expression will be returned. If True then the number of each type of operation is shown with the core class types (or their virtual equivalent) multiplied by the number of times they occur.

Examples

```
>>> from sympy.abc import a, b, x, y
>>> from sympy import sin, count_ops
```

Although there is not a SUB object, minus signs are interpreted as either negations or subtractions:

```
>>> (x - y).count_ops(visual=True)
SUB
>>> (-x).count_ops(visual=True)
NEG
```

Here, there are two Adds and a Pow:

```
>>> (1 + a + b**2).count_ops(visual=True)
2*ADD + POW
```

In the following, an Add, Mul, Pow and two functions:

```
>>> (sin(x)*x + sin(x)**2).count_ops(visual=True)
ADD + MUL + POW + 2*SIN
```

for a total of 5:

```
>>> (sin(x)*x + sin(x)**2).count_ops(visual=False)
5
```

Note that “what you type” is not always what you get. The expression $1/x/y$ is translated by sympy into $1/(x*y)$ so it gives a DIV and MUL rather than two DIVs:

```
>>> (1/x/y).count_ops(visual=True)
DIV + MUL
```

The visual option can be used to demonstrate the difference in operations for expressions in different forms. Here, the Horner representation is compared with the expanded form of a polynomial:

```
>>> eq=x*(1 + x*(2 + x*(3 + x)))
>>> count_ops(eq.expand(), visual=True) - count_ops(eq, visual=True)
-MUL + 3*POW
```

The count_ops function also handles iterables:

```
>>> count_ops([x, sin(x), None, True, x + 2], visual=False)
2
>>> count_ops([x, sin(x), None, True, x + 2], visual=True)
ADD + SIN
>>> count_ops({x: sin(x), x + 2: y + 1}, visual=True)
2*ADD + SIN
```

expand_mul

`sympy.core.function.expand_mul(expr, deep=True)`

Wrapper around `expand` that only uses the `mul` hint. See the `expand` docstring for more information.

Examples

```
>>> from sympy import symbols, expand_mul, exp, log
>>> x, y = symbols('x,y', positive=True)
>>> expand_mul(exp(x+y)*(x+y)*log(x*y**2))
x*exp(x + y)*log(x*y**2) + y*exp(x + y)*log(x*y**2)
```

expand_log

`sympy.core.function.expand_log(expr, deep=True, force=False, factor=False)`

Wrapper around `expand` that only uses the `log` hint. See the `expand` docstring for more information.

Examples

```
>>> from sympy import symbols, expand_log, exp, log
>>> x, y = symbols('x,y', positive=True)
>>> expand_log(exp(x+y)*(x+y)*log(x*y**2))
(x + y)*(log(x) + 2*log(y))*exp(x + y)
```

expand_func

`sympy.core.function.expand_func(expr, deep=True)`

Wrapper around `expand` that only uses the `func` hint. See the `expand` docstring for more information.

Examples

```
>>> from sympy import expand_func, gamma
>>> from sympy.abc import x
>>> expand_func(gamma(x + 2))
x*(x + 1)*gamma(x)
```

expand_trig

`sympy.core.function.expand_trig(expr, deep=True)`

Wrapper around `expand` that only uses the trig hint. See the `expand` docstring for more information.

Examples

```
>>> from sympy import expand_trig, sin
>>> from sympy.abc import x, y
>>> expand_trig(sin(x+y)*(x+y))
(x + y)*(sin(x)*cos(y) + sin(y)*cos(x))
```

expand_complex

`sympy.core.function.expand_complex(expr, deep=True)`

Wrapper around `expand` that only uses the complex hint. See the `expand` docstring for more information.

Examples

```
>>> from sympy import expand_complex, exp, sqrt, I
>>> from sympy.abc import z
>>> expand_complex(exp(z))
I*exp(re(z))*sin(im(z)) + exp(re(z))*cos(im(z))
>>> expand_complex(sqrt(I))
sqrt(2)/2 + sqrt(2)*I/2
```

See also:

[`sympy.core.expr.Expr.as\_real\_imag`](#) (page 956)

expand_multinomial

`sympy.core.function.expand_multinomial(expr, deep=True)`

Wrapper around `expand` that only uses the multinomial hint. See the `expand` docstring for more information.

Examples

```
>>> from sympy import symbols, expand_multinomial, exp
>>> x, y = symbols('x y', positive=True)
>>> expand_multinomial((x + exp(x + 1))**2)
x**2 + 2*x*exp(x + 1) + exp(2*x + 2)
```

expand_power_exp

`sympy.core.function.expand_power_exp(expr, deep=True)`
 Wrapper around `expand` that only uses the `power_exp` hint.
 See the `expand` docstring for more information.

Examples

```
>>> from sympy import expand_power_exp
>>> from sympy.abc import x, y
>>> expand_power_exp(x**(y + 2))
x**2*x**y
```

expand_power_base

`sympy.core.function.expand_power_base(expr, deep=True, force=False)`
 Wrapper around `expand` that only uses the `power_base` hint.

A wrapper to `expand(power_base=True)` which separates a power with a base that is a `Mul` into a product of powers, without performing any other expansions, provided that assumptions about the power's base and exponent allow.

`deep=False` (default is `True`) will only apply to the top-level expression.

`force=True` (default is `False`) will cause the expansion to ignore assumptions about the base and exponent. When `False`, the expansion will only happen if the base is non-negative or the exponent is an integer.

```
>>> from sympy.abc import x, y, z
>>> from sympy import expand_power_base, sin, cos, exp
```

```
>>> (x*y)**2
x**2*y**2
```

```
>>> (2*x)**y
(2*x)**y
>>> expand_power_base(_)
2**y*x**y
```

```
>>> expand_power_base((x*y)**z)
(x*y)**z
>>> expand_power_base((x*y)**z, force=True)
x**z*y**z
>>> expand_power_base(sin((x*y)**z), deep=False)
sin((x*y)**z)
>>> expand_power_base(sin((x*y)**z), force=True)
sin(x**z*y**z)
```

```
>>> expand_power_base((2*sin(x))**y + (2*cos(x))**y)
2**y*sin(x)**y + 2**y*cos(x)**y
```

```
>>> expand_power_base((2*exp(y))**x)
2**x*exp(y)**x
```

```
>>> expand_power_base((2*cos(x))**y)
2**y*cos(x)**y
```

Notice that sums are left untouched. If this is not the desired behavior, apply full `expand()` to the expression:

```
>>> expand_power_base(((x+y)*z)**2)
z**2*(x + y)**2
>>> (((x+y)*z)**2).expand()
x**2*z**2 + 2*x*y*z**2 + y**2*z**2
```

```
>>> expand_power_base((2*y)**(1+z))
2**(z + 1)*y**(z + 1)
>>> ((2*y)**(1+z)).expand()
2*2**z*y*y**z
```

See also:

[*expand*](#) (page 1053)

nfloat

`sympy.core.function.nfloat(expr, n=15, exponent=False, dkeys=False)`

Make all Rationals in `expr` Floats except those in exponents (unless the `exponents` flag is set to `True`) and those in undefined functions. When processing dictionaries, do not modify the keys unless `dkeys=True`.

Examples

```
>>> from sympy import nfloat, cos, pi, sqrt
>>> from sympy.abc import x, y
>>> nfloat(x**4 + x/2 + cos(pi/3) + 1 + sqrt(y))
x**4 + 0.5*x + sqrt(y) + 1.5
>>> nfloat(x**4 + sqrt(y), exponent=True)
x**4.0 + y**0.5
```

Container types are not modified:

```
>>> type(nfloat((1, 2))) is tuple
True
```

evalf

EvalfMixin

class sympy.core.evalf.**EvalfMixin**

Mixin class adding evalf capability.

evalf(*n=15*, *subs=None*, *maxn=100*, *chop=False*, *strict=False*, *quad=None*, *verbose=False*)

Evaluate the given formula to an accuracy of *n* digits.

Parameters

subs : dict, optional

Substitute numerical values for symbols, e.g. `subs={x:3, y:1+pi}`.
The substitutions must be given as a dictionary.

maxn : int, optional

Allow a maximum temporary working precision of *maxn* digits.

chop : bool or number, optional

Specifies how to replace tiny real or imaginary parts in subresults by exact zeros.

When True the chop value defaults to standard precision.

Otherwise the chop value is used to determine the magnitude of “small” for purposes of chopping.

```
>>> from sympy import N
>>> x = 1e-4
>>> N(x, chop=True)
0.00010000000000000000
>>> N(x, chop=1e-5)
0.00010000000000000000
>>> N(x, chop=1e-4)
0
```

strict : bool, optional

Raise `PrecisionExhausted` if any subresult fails to evaluate to full accuracy, given the available `maxprec`.

quad : str, optional

Choose algorithm for numerical quadrature. By default, `tanh-sinh` quadrature is used. For oscillatory integrals on an infinite interval, try `quad='osc'`.

verbose : bool, optional

Print debug information.

Notes

When Floats are naively substituted into an expression, precision errors may adversely affect the result. For example, adding `1e16` (a Float) to `1` will truncate to `1e16`; if `1e16` is then subtracted, the result will be `0`. That is exactly what happens in the following:

```
>>> from sympy.abc import x, y, z
>>> values = {x: 1e16, y: 1, z: 1e16}
>>> (x + y - z).subs(values)
0
```

Using the `subs` argument for `evalf` is the accurate way to evaluate such an expression:

```
>>> (x + y - z).evalf(subs=values)
1.0000000000000000
```

`n` (`n=15`, `subs=None`, `maxn=100`, `chop=False`, `strict=False`, `quad=None`, `verbose=False`)

Evaluate the given formula to an accuracy of n digits.

Parameters

subs : dict, optional

Substitute numerical values for symbols, e.g. `subs={x:3, y:1+pi}`. The substitutions must be given as a dictionary.

maxn : int, optional

Allow a maximum temporary working precision of `maxn` digits.

chop : bool or number, optional

Specifies how to replace tiny real or imaginary parts in subresults by exact zeros.

When `True` the `chop` value defaults to standard precision.

Otherwise the `chop` value is used to determine the magnitude of “small” for purposes of chopping.

```
>>> from sympy import N
>>> x = 1e-4
>>> N(x, chop=True)
```

(continues on next page)

(continued from previous page)

```
0.00010000000000000000
>>> N(x, chop=1e-5)
0.00010000000000000000
>>> N(x, chop=1e-4)
0
```

strict : bool, optional

Raise PrecisionExhausted if any subresult fails to evaluate to full accuracy, given the available maxprec.

quad : str, optional

Choose algorithm for numerical quadrature. By default, tanh-sinh quadrature is used. For oscillatory integrals on an infinite interval, try quad='osc'.

verbose : bool, optional

Print debug information.

Notes

When Floats are naively substituted into an expression, precision errors may adversely affect the result. For example, adding $1e16$ (a Float) to 1 will truncate to $1e16$; if $1e16$ is then subtracted, the result will be 0. That is exactly what happens in the following:

```
>>> from sympy.abc import x, y, z
>>> values = {x: 1e16, y: 1, z: 1e16}
>>> (x + y - z).subs(values)
0
```

Using the subs argument for evalf is the accurate way to evaluate such an expression:

```
>>> (x + y - z).evalf(subs=values)
1.0000000000000000
```

PrecisionExhausted

class sympy.core.evalf.PrecisionExhausted

N

`sympy.core.evalf.N(x, n=15, **options)`

Calls `x.evalf(n, **options)`.

Explanations

Both `.n()` and `N()` are equivalent to `.evalf()`; use the one that you like better. See also the docstring of `.evalf()` for information on the options.

Examples

```
>>> from sympy import Sum, oo, N
>>> from sympy.abc import k
>>> Sum(1/k**k, (k, 1, oo))
Sum(k**(-k), (k, 1, oo))
>>> N(_, 4)
1.291
```

containers

Tuple

class `sympy.core.containers.Tuple(*args, **kwargs)`

Wrapper around the builtin tuple object.

Parameters

sympify : bool

If False, `sympify` is not called on args. This can be used for speedups for very large tuples where the elements are known to already be SymPy objects.

Explanation

The `Tuple` is a subclass of `Basic`, so that it works well in the SymPy framework. The wrapped tuple is available as `self.args`, but you can also access elements or slices with `[:]` syntax.

Examples

```
>>> from sympy import Tuple, symbols
>>> a, b, c, d = symbols('a b c d')
>>> Tuple(a, b, c)[1:]
(b, c)
>>> Tuple(a, b, c).subs(a, d)
(d, b, c)
```

index(*value*, *start=None*, *stop=None*)

Searches and returns the first index of the value.

property kind

The kind of a Tuple instance.

The kind of a Tuple is always of [TupleKind](#) (page 1069) but parametrised by the number of elements and the kind of each element.

Examples

```
>>> from sympy import Tuple, Matrix
>>> Tuple(1, 2).kind
TupleKind(NumberKind, NumberKind)
>>> Tuple(Matrix([1, 2]), 1).kind
TupleKind(MatrixKind(NumberKind), NumberKind)
>>> Tuple(1, 2).kind.element_kind
(NumberKind, NumberKind)
```

See also:

[sympy.matrices.common.MatrixKind](#) (page 1360), [sympy.core.kind.NumberKind](#) (page 1074)

tuple_count(*value*)

T.count(value) -> integer - return number of occurrences of value

TupleKind

class sympy.core.containers.**TupleKind**(*args)

TupleKind is a subclass of Kind, which is used to define Kind of Tuple.

Parameters of TupleKind will be kinds of all the arguments in Tuples, for example

Parameters

args : tuple(element_kind)

element_kind is kind of element. args is tuple of kinds of element

Examples

```
>>> from sympy import Tuple
>>> Tuple(1, 2).kind
TupleKind(NumberKind, NumberKind)
>>> Tuple(1, 2).kind.element_kind
(NumberKind, NumberKind)
```

See also:

[*sympy.core.kind.NumberKind*](#) (page 1074), [*MatrixKind*](#) (page 1360), [*sympy.sets.sets.SetKind*](#) (page 1217)

Dict

class sympy.core.containers.Dict(*args)

Wrapper around the builtin dict object

Explanation

The Dict is a subclass of Basic, so that it works well in the SymPy framework. Because it is immutable, it may be included in sets, but its values must all be given at instantiation and cannot be changed afterwards. Otherwise it behaves identically to the Python dict.

Examples

```
>>> from sympy import Dict, Symbol
```

```
>>> D = Dict({1: 'one', 2: 'two'})
>>> for key in D:
...     if key == 1:
...         print('%s %s' % (key, D[key]))
1 one
```

The args are sympified so the 1 and 2 are Integers and the values are Symbols. Queries automatically sympify args so the following work:

```
>>> 1 in D
True
>>> D.has(Symbol('one')) # searches keys and values
True
>>> 'one' in D # not in the keys
False
>>> D[1]
one
```

get(key, default=None)

Returns the value for key if the key is in the dictionary.

items()

Returns a set-like object providing a view on dict's items.

keys()

Returns the list of the dict's keys.

values()

Returns the list of the dict's values.

exprtools

gcd_terms

`sympy.core.exprtools.gcd_terms(terms, isprimitive=False, clear=True, fraction=True)`

Compute the GCD of terms and put them together.

Parameters**terms** : Expr

Can be an expression or a non-Basic sequence of expressions which will be handled as though they are terms from a sum.

isprimitive : bool, optionalIf isprimitive is True the `_gcd_terms` will not run the primitive method on the terms.**clear** : bool, optional

It controls the removal of integers from the denominator of an Add expression. When True (default), all numerical denominator will be cleared; when False the denominators will be cleared only if all terms had numerical denominators other than 1.

fraction : bool, optional

When True (default), will put the expression over a common denominator.

Examples

```
>>> from sympy import gcd_terms
>>> from sympy.abc import x, y
```

```
>>> gcd_terms((x + 1)**2*y + (x + 1)*y**2)
y*(x + 1)*(x + y + 1)
>>> gcd_terms(x/2 + 1)
(x + 2)/2
>>> gcd_terms(x/2 + 1, clear=False)
x/2 + 1
>>> gcd_terms(x/2 + y/2, clear=False)
(x + y)/2
>>> gcd_terms(x/2 + 1/x)
(x**2 + 2)/(2*x)
```

(continues on next page)

(continued from previous page)

```
>>> gcd_terms(x/2 + 1/x, fraction=False)
(x + 2/x)/2
>>> gcd_terms(x/2 + 1/x, fraction=False, clear=False)
x/2 + 1/x

>>> gcd_terms(x/2/y + 1/x/y)
(x**2 + 2)/(2*x*y)
>>> gcd_terms(x/2/y + 1/x/y, clear=False)
(x**2/2 + 1)/(x*y)
>>> gcd_terms(x/2/y + 1/x/y, clear=False, fraction=False)
(x/2 + 1/x)/y
```

The `clear` flag was ignored in this case because the returned expression was a rational expression, not a simple sum.

See also:

[`factor\_terms`](#) (page 1072), [`sympy.polys.polytools.terms\_gcd`](#) (page 2366)

factor_terms

`sympy.core.exprtools.factor_terms(expr, radical=False, clear=False, fraction=False, sign=True)`

Remove common factors from terms in all arguments without changing the underlying structure of the `expr`. No expansion or simplification (and no processing of non-commutatives) is performed.

Parameters

radical: bool, optional

If `radical=True` then a radical common to all terms will be factored out of any Add sub-expressions of the `expr`.

clear : bool, optional

If `clear=False` (default) then coefficients will not be separated from a single Add if they can be distributed to leave one or more terms with integer coefficients.

fraction : bool, optional

If `fraction=True` (default is False) then a common denominator will be constructed for the expression.

sign : bool, optional

If `sign=True` (default) then even if the only factor in common is a -1, it will be factored out of the expression.

Examples

```
>>> from sympy import factor_terms, Symbol
>>> from sympy.abc import x, y
>>> factor_terms(x + x*(2 + 4*y)**3)
x*(8*(2*y + 1)**3 + 1)
>>> A = Symbol('A', commutative=False)
>>> factor_terms(x*A + x*A + x*y*A)
x*(y*A + 2*A)
```

When `clear` is `False`, a rational will only be factored out of an `Add` expression if all terms of the `Add` have coefficients that are fractions:

```
>>> factor_terms(x/2 + 1, clear=False)
x/2 + 1
>>> factor_terms(x/2 + 1, clear=True)
(x + 2)/2
```

If a `-1` is all that can be factored out, to *not* factor it out, the flag `sign` must be `False`:

```
>>> factor_terms(-x - y)
-(x + y)
>>> factor_terms(-x - y, sign=False)
-x - y
>>> factor_terms(-2*x - 2*y, sign=False)
-2*(x + y)
```

See also:

[`gcd\_terms`](#) (page 1071), [`sympy.polys.polytools.terms\_gcd`](#) (page 2366)

kind

Kind

class `sympy.core.kind.Kind(*args)`

Base class for kinds.

Kind of the object represents the mathematical classification that the entity falls into. It is expected that functions and classes recognize and filter the argument by its kind.

Kind of every object must be carefully selected so that it shows the intention of design. Expressions may have different kind according to the kind of its arguments. For example, arguments of `Add` must have common kind since addition is group operator, and the resulting `Add()` has the same kind.

For the performance, each kind is as broad as possible and is not based on set theory. For example, `NumberKind` includes not only complex number but expression containing `S.Infinity` or `S.NaN` which are not strictly number.

Kind may have arguments as parameter. For example, `MatrixKind()` may be constructed with one element which represents the kind of its elements.

Kind behaves in singleton-like fashion. Same signature will return the same object.

NumberKind

`sympy.core.kind.NumberKind`

alias of `NumberKind`

UndefinedKind

`sympy.core.kind.UndefinedKind`

alias of `UndefinedKind`

BooleanKind

`sympy.core.kind.BooleanKind`

alias of `BooleanKind`

Sorting

default_sort_key

`sympy.core.sorting.default_sort_key(item, order=None)`

Return a key that can be used for sorting.

The key has the structure:

`(class_key, (len(args), args), exponent.sort_key(), coefficient)`

This key is supplied by the `sort_key` routine of Basic objects when `item` is a Basic object or an object (other than a string) that sympifies to a Basic object. Otherwise, this function produces the key.

The `order` argument is passed along to the `sort_key` routine and is used to determine how the terms *within* an expression are ordered. (See examples below) `order` options are: 'lex', 'grlex', 'grevlex', and reversed values of the same (e.g. 'rev-lex'). The default order value is `None` (which translates to 'lex').

Examples

```
>>> from sympy import S, I, default_sort_key, sin, cos, sqrt
>>> from sympy.core.function import UndefinedFunction
>>> from sympy.abc import x
```

The following are equivalent ways of getting the key for an object:

```
>>> x.sort_key() == default_sort_key(x)
True
```

Here are some examples of the key that is produced:

```

>>> default_sort_key(UndefinedFunction('f'))
((0, 0, 'UndefinedFunction'), (1, ('f',)), ((1, 0, 'Number'),
(0, ()), ()), 1), 1)
>>> default_sort_key('1')
((0, 0, 'str'), (1, ('1',)), ((1, 0, 'Number'), (0, ()), ()), 1), 1)
>>> default_sort_key(S.One)
((1, 0, 'Number'), (0, ()), ()), 1)
>>> default_sort_key(2)
((1, 0, 'Number'), (0, ()), ()), 2)

```

While `sort_key` is a method only defined for SymPy objects, `default_sort_key` will accept anything as an argument so it is more robust as a sorting key. For the following, using `key= lambda i: i.sort_key()` would fail because 2 does not have a `sort_key` method; that's why `default_sort_key` is used. Note, that it also handles sympification of non-string items like ints:

```

>>> a = [2, I, -I]
>>> sorted(a, key=default_sort_key)
[2, -I, I]

```

The returned key can be used anywhere that a key can be specified for a function, e.g. `sort`, `min`, `max`, etc...:

```

>>> a.sort(key=default_sort_key); a[0]
2
>>> min(a, key=default_sort_key)
2

```

Note

The key returned is useful for getting items into a canonical order that will be the same across platforms. It is not directly useful for sorting lists of expressions:

```

>>> a, b = x, 1/x

```

Since `a` has only 1 term, its value of `sort_key` is unaffected by order:

```

>>> a.sort_key() == a.sort_key('rev-lex')
True

```

If `a` and `b` are combined then the key will differ because there are terms that can be ordered:

```

>>> eq = a + b
>>> eq.sort_key() == eq.sort_key('rev-lex')
False
>>> eq.as_ordered_terms()
[x, 1/x]
>>> eq.as_ordered_terms('rev-lex')
[1/x, x]

```

But since the keys for each of these terms are independent of order's value, they do not sort differently when they appear separately in a list:

```
>>> sorted(eq.args, key=default_sort_key)
[1/x, x]
>>> sorted(eq.args, key=lambda i: default_sort_key(i, order='rev-lex'))
[1/x, x]
```

The order of terms obtained when using these keys is the order that would be obtained if those terms were *factors* in a product.

Although it is useful for quickly putting expressions in canonical order, it does not sort expressions based on their complexity defined by the number of operations, power of variables and others:

```
>>> sorted([sin(x)*cos(x), sin(x)], key=default_sort_key)
[sin(x)*cos(x), sin(x)]
>>> sorted([x, x**2, sqrt(x), x**3], key=default_sort_key)
[sqrt(x), x, x**2, x**3]
```

See also:

[ordered](#) (page 1076), [sympy.core.expr.Expr.as_ordered_factors](#) (page 955), [sympy.core.expr.Expr.as_ordered_terms](#) (page 955)

ordered

`sympy.core.sorting.ordered(seq, keys=None, default=True, warn=False)`

Return an iterator of the seq where keys are used to break ties in a conservative fashion: if, after applying a key, there are no ties then no other keys will be computed.

Two default keys will be applied if 1) keys are not provided or 2) the given keys do not resolve all ties (but only if default is True). The two keys are `_nodes` (which places smaller expressions before large) and `default_sort_key` which (if the `sort_key` for an object is defined properly) should resolve any ties.

If `warn` is True then an error will be raised if there were no keys remaining to break ties. This can be used if it was expected that there should be no ties between items that are not identical.

Examples

```
>>> from sympy import ordered, count_ops
>>> from sympy.abc import x, y
```

The `count_ops` is not sufficient to break ties in this list and the first two items appear in their original order (i.e. the sorting is stable):

```
>>> list(ordered([y + 2, x + 2, x**2 + y + 3],
...             count_ops, default=False, warn=False))
...
[y + 2, x + 2, x**2 + y + 3]
```

The `default_sort_key` allows the tie to be broken:

```
>>> list(ordered([y + 2, x + 2, x**2 + y + 3]))
...
[x + 2, y + 2, x**2 + y + 3]
```

Here, sequences are sorted by length, then sum:

```
>>> seq, keys = [[[1, 2, 1], [0, 3, 1], [1, 1, 3], [2], [1]], [
...     lambda x: len(x),
...     lambda x: sum(x)]]
...
>>> list(ordered(seq, keys, default=False, warn=False))
[[1], [2], [1, 2, 1], [0, 3, 1], [1, 1, 3]]
```

If warn is True, an error will be raised if there were not enough keys to break ties:

```
>>> list(ordered(seq, keys, default=False, warn=True))
Traceback (most recent call last):
...
ValueError: not enough keys to break ties
```

Notes

The decorated sort is one of the fastest ways to sort a sequence for which special item comparison is desired: the sequence is decorated, sorted on the basis of the decoration (e.g. making all letters lower case) and then undecorated. If one wants to break ties for items that have the same decorated value, a second key can be used. But if the second key is expensive to compute then it is inefficient to decorate all items with both keys: only those items having identical first key values need to be decorated. This function applies keys successively only when needed to break ties. By yielding an iterator, use of the tie-breaker is delayed as long as possible.

This function is best used in cases when use of the first key is expected to be a good hashing function; if there are no unique hashes from application of a key, then that key should not have been used. The exception, however, is that even if there are many collisions, if the first group is small and one does not need to process all items in the list then time will not be wasted sorting what one was not interested in. For example, if one were looking for the minimum in a list and there were several criteria used to define the sort order, then this function would be good at returning that quickly if the first group of candidates is small relative to the number of items being processed.

Random

When you need to use random numbers in SymPy library code, import from here so there is only one generator working for SymPy. Imports from here should behave the same as if they were being imported from Python's random module. But only the routines currently used in SymPy are included here. To use others import rng and access the method directly. For example, to capture the current state of the generator use `rng.getstate()`.

There is intentionally no Random to import from here. If you want to control the state of the generator, import `seed` and call it with or without an argument to set the state.

Examples

```
>>> from sympy.core.random import random, seed
>>> assert random() < 1
>>> seed(1); a = random()
>>> b = random()
>>> seed(1); c = random()
>>> assert a == c
>>> assert a != b # remote possibility this will fail
```

random_complex_number

`sympy.core.random.random_complex_number(a=2, b=-1, c=3, d=1, rational=False, tolerance=None)`

Return a random complex number.

To reduce chance of hitting branch cuts or anything, we guarantee $b \leq \text{Im } z \leq d$, $a \leq \text{Re } z \leq c$

When `rational` is `True`, a rational approximation to a random number is obtained within specified tolerance, if any.

verify_numerically

`sympy.core.random.verify_numerically(f, g, z=None, tol=1e-06, a=2, b=-1, c=3, d=1)`

Test numerically that `f` and `g` agree when evaluated in the argument `z`.

If `z` is `None`, all symbols will be tested. This routine does not test whether there are Floats present with precision higher than 15 digits so if there are, your results may not be what you expect due to round-off errors.

Examples

```
>>> from sympy import sin, cos
>>> from sympy.abc import x
>>> from sympy.core.random import verify_numerically as tn
>>> tn(sin(x)**2 + cos(x)**2, 1, x)
True
```

test_derivative_numerically

`sympy.core.random.test_derivative_numerically(f, z, tol=1e-06, a=2, b=-1, c=3, d=1)`

Test numerically that the symbolically computed derivative of `f` with respect to `z` is correct.

This routine does not test whether there are Floats present with precision higher than 15 digits so if there are, your results may not be what you expect due to round-off errors.

Examples

```
>>> from sympy import sin
>>> from sympy.abc import x
>>> from sympy.core.random import test_derivative_numerically as td
>>> td(sin(x), x)
True
```

_randrange

`sympy.core.random._randrange(seed=None)`

Return a randrange generator.

seed can be

- None - return randomly seeded generator
- int - return a generator seeded with the int
- list - the values to be returned will be taken from the list in the order given; the provided list is not modified.

Examples

```
>>> from sympy.core.random import _randrange
>>> rr = _randrange()
>>> rr(1000)
999
>>> rr = _randrange(3)
>>> rr(1000)
238
>>> rr = _randrange([0, 5, 1, 3, 4])
>>> rr(3), rr(3)
(0, 1)
```

_randint

`sympy.core.random._randint(seed=None)`

Return a randint generator.

seed can be

- None - return randomly seeded generator
- int - return a generator seeded with the int
- list - the values to be returned will be taken from the list in the order given; the provided list is not modified.

Examples

```
>>> from sympy.core.random import _randint
>>> ri = _randint()
>>> ri(1, 1000)
999
>>> ri = _randint(3)
>>> ri(1, 1000)
238
>>> ri = _randint([0, 5, 1, 2, 4])
>>> ri(1, 3), ri(1, 3)
(1, 2)
```

Traversal

bottom_up

`sympy.core.traversal.bottom_up(rv, F, atoms=False, nonbasic=False)`

Apply *F* to all expressions in an expression tree from the bottom up. If *atoms* is *True*, apply *F* even if there are no args; if *nonbasic* is *True*, try to apply *F* to non-Basic objects.

postorder_traversal

`sympy.core.traversal.postorder_traversal(node, keys=None)`

Do a postorder traversal of a tree.

This generator recursively yields nodes that it has visited in a postorder fashion. That is, it descends through the tree depth-first to yield all of a node's children's postorder traversal before yielding the node itself.

Parameters

node : SymPy expression

The expression to traverse.

keys : (default *None*) sort key(s)

The key(s) used to sort args of Basic objects. When *None*, args of Basic objects are processed in arbitrary order. If key is defined, it will be passed along to `ordered()` as the only key(s) to use to sort the arguments; if key is simply *True* then the default keys of `ordered` will be used (node count and `default_sort_key`).

Yields

subtree : SymPy expression

All of the subtrees in the tree.

Examples

```
>>> from sympy import postorder_traversal
>>> from sympy.abc import w, x, y, z
```

The nodes are returned in the order that they are encountered unless key is given; simply passing key=True will guarantee that the traversal is unique.

```
>>> list(postorder_traversal(w + (x + y)*z))
[z, y, x, x + y, z*(x + y), w, w + z*(x + y)]
>>> list(postorder_traversal(w + (x + y)*z, keys=True))
[w, z, x, y, x + y, z*(x + y), w + z*(x + y)]
```

preorder_traversal

`sympy.core.traversal.preorder_traversal(node, keys=None)`

Do a pre-order traversal of a tree.

This iterator recursively yields nodes that it has visited in a pre-order fashion. That is, it yields the current node then descends through the tree breadth-first to yield all of a node's children's pre-order traversal.

For an expression, the order of the traversal depends on the order of `.args`, which in many cases can be arbitrary.

Parameters

node : SymPy expression

The expression to traverse.

keys : (default None) sort key(s)

The key(s) used to sort args of Basic objects. When None, args of Basic objects are processed in arbitrary order. If key is defined, it will be passed along to `ordered()` as the only key(s) to use to sort the arguments; if key is simply True then the default keys of `ordered` will be used.

Yields

subtree : SymPy expression

All of the subtrees in the tree.

Examples

```
>>> from sympy import preorder_traversal, symbols
>>> x, y, z = symbols('x y z')
```

The nodes are returned in the order that they are encountered unless key is given; simply passing key=True will guarantee that the traversal is unique.

```
>>> list(preorder_traversal((x + y)*z, keys=None))
[z*(x + y), z, x + y, y, x]
```

(continues on next page)

(continued from previous page)

```
>>> list(preorder_traversal((x + y)*z, keys=True))
[z*(x + y), z, x + y, x, y]
```

use

`sympy.core.traversal.use(expr, func, level=0, args=(), kwargs={})`

Use func to transform expr at the given level.

Examples

```
>>> from sympy import use, expand
>>> from sympy.abc import x, y
```

```
>>> f = (x + y)**2*x + 1
```

```
>>> use(f, expand, level=2)
x*(x**2 + 2*x*y + y**2) + 1
>>> expand(f)
x**3 + 2*x**2*y + x*y**2 + 1
```

walk

`sympy.core.traversal.walk(e, *target)`

Iterate through the args that are the given types (target) and return a list of the args that were traversed; arguments that are not of the specified types are not traversed.

Examples

```
>>> from sympy.core.traversal import walk
>>> from sympy import Min, Max
>>> from sympy.abc import x, y, z
>>> list(walk(Min(x, Max(y, Min(1, z))), Min))
[Min(x, Max(y, Min(1, z)))]
>>> list(walk(Min(x, Max(y, Min(1, z))), Min, Max))
[Min(x, Max(y, Min(1, z))), Max(y, Min(1, z)), Min(1, z)]
```

See also:

[*bottom_up*](#) (page 1080)

Discrete

The `discrete` module in SymPy implements methods to compute discrete transforms and convolutions of finite sequences.

This module contains functions which operate on discrete sequences.

Transforms - `fft`, `ifft`, `ntt`, `intt`, `fwht`, `ifwht`,
`mobius_transform`, `inverse_mobius_transform`

Convolutions - `convolution`, `convolution_fft`, `convolution_ntt`,
`convolution_fwht`, `convolution_subset`, `covering_product`, `intersecting_product`

Since the discrete transforms can be used to reduce the computational complexity of the discrete convolutions, the `convolutions` module makes use of the `transforms` module for efficient computation (notable for long input sequences).

Transforms

This section lists the methods which implement the basic transforms for discrete sequences.

Fast Fourier Transform

`sympy.discrete.transforms.fft(seq, dps=None)`

Performs the Discrete Fourier Transform (**DFT**) in the complex domain.

The sequence is automatically padded to the right with zeros, as the *radix-2 FFT* requires the number of sample points to be a power of 2.

This method should be used with default arguments only for short sequences as the complexity of expressions increases with the size of the sequence.

Parameters

seq : iterable

The sequence on which **DFT** is to be applied.

dps : Integer

Specifies the number of decimal digits for precision.

Examples

```
>>> from sympy import fft, ifft
```

```
>>> fft([1, 2, 3, 4])
[10, -2 - 2*I, -2, -2 + 2*I]
>>> ifft(_)
[1, 2, 3, 4]
```

```
>>> ifft([1, 2, 3, 4])
[5/2, -1/2 + I/2, -1/2, -1/2 - I/2]
>>> fft(_)
[1, 2, 3, 4]
```

```
>>> ifft([1, 7, 3, 4], dps=15)
[3.75, -0.5 - 0.75*I, -1.75, -0.5 + 0.75*I]
>>> fft(_)
[1.0, 7.0, 3.0, 4.0]
```

References

[R164], [R165]

`sympy.discrete.transforms.ifft(seq, dps=None)`

Performs the Discrete Fourier Transform (**DFT**) in the complex domain.

The sequence is automatically padded to the right with zeros, as the *radix-2 FFT* requires the number of sample points to be a power of 2.

This method should be used with default arguments only for short sequences as the complexity of expressions increases with the size of the sequence.

Parameters

seq : iterable

The sequence on which **DFT** is to be applied.

dps : Integer

Specifies the number of decimal digits for precision.

Examples

```
>>> from sympy import fft, ifft
```

```
>>> fft([1, 2, 3, 4])
[10, -2 - 2*I, -2, -2 + 2*I]
>>> ifft(_)
[1, 2, 3, 4]
```

```
>>> ifft([1, 2, 3, 4])
[5/2, -1/2 + I/2, -1/2, -1/2 - I/2]
>>> fft(_)
[1, 2, 3, 4]
```

```
>>> ifft([1, 7, 3, 4], dps=15)
[3.75, -0.5 - 0.75*I, -1.75, -0.5 + 0.75*I]
>>> fft(_)
[1.0, 7.0, 3.0, 4.0]
```

References

[R166], [R167]

Number Theoretic Transform

`sympy.discrete.transforms.ntt(seq, prime)`

Performs the Number Theoretic Transform (**NTT**), which specializes the Discrete Fourier Transform (**DFT**) over quotient ring Z/pZ for prime p instead of complex numbers C .

The sequence is automatically padded to the right with zeros, as the *radix-2 NTT* requires the number of sample points to be a power of 2.

Parameters

seq : iterable

The sequence on which **DFT** is to be applied.

prime : Integer

Prime modulus of the form $(m2^k + 1)$ to be used for performing **NTT** on the sequence.

Examples

```
>>> from sympy import ntt, intt
>>> ntt([1, 2, 3, 4], prime=3*2**8 + 1)
[10, 643, 767, 122]
>>> intt(_, 3*2**8 + 1)
[1, 2, 3, 4]
>>> intt([1, 2, 3, 4], prime=3*2**8 + 1)
[387, 415, 384, 353]
>>> ntt(_, prime=3*2**8 + 1)
[1, 2, 3, 4]
```

References

[R168], [R169], [R170]

`sympy.discrete.transforms.intt(seq, prime)`

Performs the Number Theoretic Transform (**NTT**), which specializes the Discrete Fourier Transform (**DFT**) over quotient ring Z/pZ for prime p instead of complex numbers C .

The sequence is automatically padded to the right with zeros, as the *radix-2 NTT* requires the number of sample points to be a power of 2.

Parameters

seq : iterable

The sequence on which **DFT** is to be applied.

prime : Integer

Prime modulus of the form $(m2^k + 1)$ to be used for performing **NTT** on the sequence.

Examples

```
>>> from sympy import ntt, intt
>>> ntt([1, 2, 3, 4], prime=3*2**8 + 1)
[10, 643, 767, 122]
>>> intt(_, 3*2**8 + 1)
[1, 2, 3, 4]
>>> intt([1, 2, 3, 4], prime=3*2**8 + 1)
[387, 415, 384, 353]
>>> ntt(_, prime=3*2**8 + 1)
[1, 2, 3, 4]
```

References

[R171], [R172], [R173]

Fast Walsh Hadamard Transform

`sympy.discrete.transforms.fwht(seq)`

Performs the Walsh Hadamard Transform (**WHT**), and uses Hadamard ordering for the sequence.

The sequence is automatically padded to the right with zeros, as the *radix-2 FWHT* requires the number of sample points to be a power of 2.

Parameters

seq : iterable

The sequence on which WHT is to be applied.

Examples

```
>>> from sympy import fwht, ifwht
>>> fwht([4, 2, 2, 0, 0, 2, -2, 0])
[8, 0, 8, 0, 8, 8, 0, 0]
>>> ifwht(_)
[4, 2, 2, 0, 0, 2, -2, 0]
```

```
>>> ifwht([19, -1, 11, -9, -7, 13, -15, 5])
[2, 0, 4, 0, 3, 10, 0, 0]
>>> fwht(_)
[19, -1, 11, -9, -7, 13, -15, 5]
```

References

[R174], [R175]

`sympy.discrete.transforms.ifwht(seq)`

Performs the Walsh Hadamard Transform (**WHT**), and uses Hadamard ordering for the sequence.

The sequence is automatically padded to the right with zeros, as the *radix-2 FWHT* requires the number of sample points to be a power of 2.

Parameters

seq : iterable

The sequence on which WHT is to be applied.

Examples

```
>>> from sympy import fwht, ifwht
>>> fwht([4, 2, 2, 0, 0, 2, -2, 0])
[8, 0, 8, 0, 8, 8, 0, 0]
>>> ifwht(_)
[4, 2, 2, 0, 0, 2, -2, 0]
```

```
>>> ifwht([19, -1, 11, -9, -7, 13, -15, 5])
[2, 0, 4, 0, 3, 10, 0, 0]
>>> fwht(_)
[19, -1, 11, -9, -7, 13, -15, 5]
```

References

[R176], [R177]

Möbius Transform

`sympy.discrete.transforms.mobius_transform(seq, subset=True)`

Performs the Möbius Transform for subset lattice with indices of sequence as bitmasks.

The indices of each argument, considered as bit strings, correspond to subsets of a finite set.

The sequence is automatically padded to the right with zeros, as the definition of subset/superset based on bitmasks (indices) requires the size of sequence to be a power of 2.

Parameters

seq : iterable

The sequence on which Möbius Transform is to be applied.

subset : bool

Specifies if Möbius Transform is applied by enumerating subsets or supersets of the given set.

Examples

```
>>> from sympy import symbols
>>> from sympy import mobius_transform, inverse_mobius_transform
>>> x, y, z = symbols('x y z')
```

```
>>> mobius_transform([x, y, z])
[x, x + y, x + z, x + y + z]
>>> inverse_mobius_transform(_)
[x, y, z, 0]
```

```
>>> mobius_transform([x, y, z], subset=False)
[x + y + z, y, z, 0]
>>> inverse_mobius_transform(_, subset=False)
[x, y, z, 0]
```

```
>>> mobius_transform([1, 2, 3, 4])
[1, 3, 4, 10]
>>> inverse_mobius_transform(_)
[1, 2, 3, 4]
>>> mobius_transform([1, 2, 3, 4], subset=False)
[10, 6, 7, 4]
>>> inverse_mobius_transform(_, subset=False)
[1, 2, 3, 4]
```

References

[R178], [R179], [R180]

`sympy.discrete.transforms.inverse_mobius_transform(seq, subset=True)`

Performs the Mobius Transform for subset lattice with indices of sequence as bitmasks.

The indices of each argument, considered as bit strings, correspond to subsets of a finite set.

The sequence is automatically padded to the right with zeros, as the definition of subset/superset based on bitmasks (indices) requires the size of sequence to be a power of 2.

Parameters

seq : iterable

The sequence on which Mobius Transform is to be applied.

subset : bool

Specifies if Mobius Transform is applied by enumerating subsets or supersets of the given set.

Examples

```
>>> from sympy import symbols
>>> from sympy import mobius_transform, inverse_mobius_transform
>>> x, y, z = symbols('x y z')
```

```
>>> mobius_transform([x, y, z])
[x, x + y, x + z, x + y + z]
>>> inverse_mobius_transform(_)
[x, y, z, 0]
```

```
>>> mobius_transform([x, y, z], subset=False)
[x + y + z, y, z, 0]
>>> inverse_mobius_transform(_, subset=False)
[x, y, z, 0]
```

```
>>> mobius_transform([1, 2, 3, 4])
[1, 3, 4, 10]
>>> inverse_mobius_transform(_)
[1, 2, 3, 4]
>>> mobius_transform([1, 2, 3, 4], subset=False)
[10, 6, 7, 4]
>>> inverse_mobius_transform(_, subset=False)
[1, 2, 3, 4]
```

References

[R181], [R182], [R183]

Convolutions

This section lists the methods which implement the basic convolutions for discrete sequences.

Convolution

This is a general method for calculating the convolution of discrete sequences, which internally calls one of the methods `convolution_fft`, `convolution_ntt`, `convolution_fwht`, or `convolution_subset`.

`sympy.discrete.convolutions.convolution(a, b, cycle=0, dps=None, prime=None, dyadic=None, subset=None)`

Performs convolution by determining the type of desired convolution using hints.

Exactly one of `dps`, `prime`, `dyadic`, `subset` arguments should be specified explicitly for identifying the type of convolution, and the argument `cycle` can be specified optionally.

For the default arguments, linear convolution is performed using **FFT**.

Parameters

a, **b** : iterables

The sequences for which convolution is performed.

cycle : Integer

Specifies the length for doing cyclic convolution.

dps : Integer

Specifies the number of decimal digits for precision for performing **FFT** on the sequence.

prime : Integer

Prime modulus of the form $(m2^k + 1)$ to be used for performing **NTT** on the sequence.

dyadic : bool

Identifies the convolution type as dyadic (*bitwise-XOR*) convolution, which is performed using **FWHT**.

subset : bool

Identifies the convolution type as subset convolution.

Examples

```
>>> from sympy import convolution, symbols, S, I
>>> u, v, w, x, y, z = symbols('u v w x y z')
```

```
>>> convolution([1 + 2*I, 4 + 3*I], [S(5)/4, 6], dps=3)
[1.25 + 2.5*I, 11.0 + 15.8*I, 24.0 + 18.0*I]
>>> convolution([1, 2, 3], [4, 5, 6], cycle=3)
[31, 31, 28]
```

```
>>> convolution([111, 777], [888, 444], prime=19*2**10 + 1)
[1283, 19351, 14219]
>>> convolution([111, 777], [888, 444], prime=19*2**10 + 1, cycle=2)
[15502, 19351]
```

```
>>> convolution([u, v], [x, y, z], dyadic=True)
[u*x + v*y, u*y + v*x, u*z, v*z]
>>> convolution([u, v], [x, y, z], dyadic=True, cycle=2)
[u*x + u*z + v*y, u*y + v*x + v*z]
```

```
>>> convolution([u, v, w], [x, y, z], subset=True)
[u*x, u*y + v*x, u*z + w*x, v*z + w*y]
>>> convolution([u, v, w], [x, y, z], subset=True, cycle=3)
[u*x + v*z + w*y, u*y + v*x, u*z + w*x]
```

Convolution using Fast Fourier Transform

`sympy.discrete.convolutions.convolution_fft(a, b, dps=None)`

Performs linear convolution using Fast Fourier Transform.

Parameters

a, b : iterables

The sequences for which convolution is performed.

dps : Integer

Specifies the number of decimal digits for precision.

Examples

```
>>> from sympy import S, I
>>> from sympy.discrete.convolutions import convolution_fft
```

```
>>> convolution_fft([2, 3], [4, 5])
[8, 22, 15]
>>> convolution_fft([2, 5], [6, 7, 3])
[12, 44, 41, 15]
>>> convolution_fft([1 + 2*I, 4 + 3*I], [S(5)/4, 6])
[5/4 + 5*I/2, 11 + 63*I/4, 24 + 18*I]
```

References

[R184], [R185]

Convolution using Number Theoretic Transform

`sympy.discrete.convolutions.convolution_ntt(a, b, prime)`

Performs linear convolution using Number Theoretic Transform.

Parameters

a, b : iterables

The sequences for which convolution is performed.

prime : Integer

Prime modulus of the form $(m2^k + 1)$ to be used for performing **NTT** on the sequence.

Examples

```
>>> from sympy.discrete.convolutions import convolution_ntt
>>> convolution_ntt([2, 3], [4, 5], prime=19*2**10 + 1)
[8, 22, 15]
>>> convolution_ntt([2, 5], [6, 7, 3], prime=19*2**10 + 1)
[12, 44, 41, 15]
>>> convolution_ntt([333, 555], [222, 666], prime=19*2**10 + 1)
[15555, 14219, 19404]
```

References

[R186], [R187]

Convolution using Fast Walsh Hadamard Transform

`sympy.discrete.convolutions.convolution_fwht(a, b)`

Performs dyadic (*bitwise-XOR*) convolution using Fast Walsh Hadamard Transform.

The convolution is automatically padded to the right with zeros, as the *radix-2 FWHT* requires the number of sample points to be a power of 2.

Parameters

a, b : iterables

The sequences for which convolution is performed.

Examples

```
>>> from sympy import symbols, S, I
>>> from sympy.discrete.convolutions import convolution_fwht
```

```
>>> u, v, x, y = symbols('u v x y')
>>> convolution_fwht([u, v], [x, y])
[u*x + v*y, u*y + v*x]
```

```
>>> convolution_fwht([2, 3], [4, 5])
[23, 22]
>>> convolution_fwht([2, 5 + 4*I, 7], [6*I, 7, 3 + 4*I])
[56 + 68*I, -10 + 30*I, 6 + 50*I, 48 + 32*I]
```

```
>>> convolution_fwht([S(33)/7, S(55)/6, S(7)/4], [S(2)/3, 5])
[2057/42, 1870/63, 7/6, 35/4]
```

References

[R188], [R189]

Subset Convolution

`sympy.discrete.convolutions.convolution_subset(a, b)`

Performs Subset Convolution of given sequences.

The indices of each argument, considered as bit strings, correspond to subsets of a finite set.

The sequence is automatically padded to the right with zeros, as the definition of subset based on bitmasks (indices) requires the size of sequence to be a power of 2.

Parameters

a, b : iterables

The sequences for which convolution is performed.

Examples

```
>>> from sympy import symbols, S
>>> from sympy.discrete.convolutions import convolution_subset
>>> u, v, x, y, z = symbols('u v x y z')
```

```
>>> convolution_subset([u, v], [x, y])
[u*x, u*y + v*x]
>>> convolution_subset([u, v, x], [y, z])
[u*y, u*z + v*y, x*y, x*z]
```

```
>>> convolution_subset([1, S(2)/3], [3, 4])
[3, 6]
>>> convolution_subset([1, 3, S(5)/7], [7])
[7, 21, 5, 0]
```

References

[R190]

Covering Product

`sympy.discrete.convolutions.covering_product(a, b)`

Returns the covering product of given sequences.

The indices of each argument, considered as bit strings, correspond to subsets of a finite set.

The covering product of given sequences is a sequence which contains the sum of products of the elements of the given sequences grouped by the *bitwise-OR* of the corresponding indices.

The sequence is automatically padded to the right with zeros, as the definition of subset based on bitmasks (indices) requires the size of sequence to be a power of 2.

Parameters

a, b : iterables

The sequences for which covering product is to be obtained.

Examples

```
>>> from sympy import symbols, S, I, covering_product
>>> u, v, x, y, z = symbols('u v x y z')
```

```
>>> covering_product([u, v], [x, y])
[u*x, u*y + v*x + v*y]
>>> covering_product([u, v, x], [y, z])
[u*y, u*z + v*y + v*z, x*y, x*z]
```

```
>>> covering_product([1, S(2)/3], [3, 4 + 5*I])
[3, 26/3 + 25*I/3]
>>> covering_product([1, 3, S(5)/7], [7, 8])
[7, 53, 5, 40/7]
```

References

[R191]

Intersecting Product

`sympy.discrete.convolutions.intersecting_product(a, b)`

Returns the intersecting product of given sequences.

The indices of each argument, considered as bit strings, correspond to subsets of a finite set.

The intersecting product of given sequences is the sequence which contains the sum of products of the elements of the given sequences grouped by the *bitwise-AND* of the corresponding indices.

The sequence is automatically padded to the right with zeros, as the definition of subset based on bitmasks (indices) requires the size of sequence to be a power of 2.

Parameters

a, b : iterables

The sequences for which intersecting product is to be obtained.

Examples

```
>>> from sympy import symbols, S, I, intersecting_product
>>> u, v, x, y, z = symbols('u v x y z')
```

```
>>> intersecting_product([u, v], [x, y])
[u*x + u*y + v*x, v*y]
>>> intersecting_product([u, v, x], [y, z])
[u*y + u*z + v*y + x*y + x*z, v*z, 0, 0]
```

```
>>> intersecting_product([1, S(2)/3], [3, 4 + 5*I])
[9 + 5*I, 8/3 + 10*I/3]
>>> intersecting_product([1, 3, S(5)/7], [7, 8])
[327/7, 24, 0, 0]
```

References

[R192]

Numerical Evaluation

Basics

Exact SymPy expressions can be converted to floating-point approximations (decimal numbers) using either the `.evalf()` method or the `N()` function. `N(expr, <args>)` is equivalent to `sympify(expr).evalf(<args>)`.

```
>>> from sympy import *
>>> N(sqrt(2)*pi)
4.44288293815837
>>> (sqrt(2)*pi).evalf()
4.44288293815837
```

By default, numerical evaluation is performed to an accuracy of 15 decimal digits. You can optionally pass a desired accuracy (which should be a positive integer) as an argument to `evalf` or `N`:

```
>>> N(sqrt(2)*pi, 5)
4.4429
>>> N(sqrt(2)*pi, 50)
4.4428829381583662470158809900606936986146216893757
```

Complex numbers are supported:

```
>>> N(1/(pi + I), 20)
0.28902548222223624241 - 0.091999668350375232456*I
```

If the expression contains symbols or for some other reason cannot be evaluated numerically, calling `.evalf()` or `N()` returns the original expression, or in some cases a partially evaluated expression. For example, when the expression is a polynomial in expanded form, the coefficients are evaluated:

```
>>> x = Symbol('x')
>>> (pi*x**2 + x/3).evalf()
3.14159265358979*x**2 + 0.333333333333333*x
```

You can also use the standard Python functions `float()`, `complex()` to convert SymPy expressions to regular Python numbers:

```
>>> float(pi)
3.1415926535...
>>> complex(pi+E*I)
(3.1415926535...+2.7182818284...j)
```

If these functions are used, failure to evaluate the expression to an explicit number (for example if the expression contains symbols) will raise an exception.

There is essentially no upper precision limit. The following command, for example, computes the first 100,000 digits of π/e :

```
>>> N(pi/E, 100000)
...
```

This shows digits 999,951 through 1,000,000 of pi:

```
>>> str(N(pi, 10**6))[-50:]
'95678796130331164628399634646042209010610577945815'
```

High-precision calculations can be slow. It is recommended (but entirely optional) to install gmpy (<https://code.google.com/p/gmpy/>), which will significantly speed up computations such as the one above.

Floating-point numbers

Floating-point numbers in SymPy are instances of the class `Float`. A `Float` can be created with a custom precision as second argument:

```
>>> Float(0.1)
0.10000000000000000
>>> Float(0.1, 10)
0.1000000000
>>> Float(0.125, 30)
0.125000000000000000000000000000000000000000000000000
>>> Float(0.1, 30)
0.100000000000000000000000000000000000000000000000000
```

As the last example shows, some Python floats are only accurate to about 15 digits as inputs, while others (those that have a denominator that is a power of 2, like $0.125 = 1/8$) are exact. To create a `Float` from a high-precision decimal number, it is better to pass a string, `Rational`, or `evalf` a `Rational`:

[illegible]

(continues on next page)

(continued from previous page)

```
>>> Rational(1, 10).evalf(30)
0.1000000000000000000000000000000000
```

The precision of a number determines 1) the precision to use when performing arithmetic with the number, and 2) the number of digits to display when printing the number. When two numbers with different precision are used together in an arithmetic operation, the higher of the precisions is used for the result. The product of 0.1 ± 0.001 and 3.1415 ± 0.0001 has an uncertainty of about 0.003 and yet 5 digits of precision are shown.

```
>>> Float(0.1, 3)*Float(3.1415, 5)
0.31417
```

So the displayed precision should not be used as a model of error propagation or significance arithmetic; rather, this scheme is employed to ensure stability of numerical algorithms.

`N` and `evalf` can be used to change the precision of existing floating-point numbers:

```
>>> N(3.5)
3.5000000000000000
>>> N(3.5, 5)
3.5000
>>> N(3.5, 30)
3.5000000000000000000000000000000000
```

Accuracy and error handling

When the input to `N` or `evalf` is a complicated expression, numerical error propagation becomes a concern. As an example, consider the 100'th Fibonacci number and the excellent (but not exact) approximation $\varphi^{100}/\sqrt{5}$ where φ is the golden ratio. With ordinary floating-point arithmetic, subtracting these numbers from each other erroneously results in a complete cancellation:

```
>>> a, b = GoldenRatio**1000/sqrt(5), fibonacci(1000)
>>> float(a)
4.34665576869e+208
>>> float(b)
4.34665576869e+208
>>> float(a) - float(b)
0.0
```

`N` and `evalf` keep track of errors and automatically increase the precision used internally in order to obtain a correct result:

```
>>> N(fibonacci(100) - GoldenRatio**100/sqrt(5))
-5.64613129282185e-22
```

Unfortunately, numerical evaluation cannot tell an expression that is exactly zero apart from one that is merely very small. The working precision is therefore capped, by default to around 100 digits. If we try with the 1000'th Fibonacci number, the following happens:

```
>>> N(fibonacci(1000) - (GoldenRatio)**1000/sqrt(5))
0.e+85
```

The lack of digits in the returned number indicates that `N` failed to achieve full accuracy. The result indicates that the magnitude of the expression is something less than 10^{84} , but that is not a particularly good answer. To force a higher working precision, the `maxn` keyword argument can be used:

```
>>> N(fibonacci(1000) - (GoldenRatio)**1000/sqrt(5), maxn=500)
-4.60123853010113e-210
```

Normally, `maxn` can be set very high (thousands of digits), but be aware that this may cause significant slowdown in extreme cases. Alternatively, the `strict=True` option can be set to force an exception instead of silently returning a value with less than the requested accuracy:

```
>>> N(fibonacci(1000) - (GoldenRatio)**1000/sqrt(5), strict=True)
Traceback (most recent call last):
...
PrecisionExhausted: Failed to distinguish the expression:

-sqrt(5)*GoldenRatio**1000/5 +
↳4346655768693745643568852767504062580256466051737178040248172908953655541794905189040387

from zero. Try simplifying the input, using chop=True, or providing a higher
↳maxn for evalf
```

If we add a term so that the Fibonacci approximation becomes exact (the full form of Binet's formula), we get an expression that is exactly zero, but `N` does not know this:

```
>>> f = fibonacci(100) - (GoldenRatio**100 - (GoldenRatio-1)**100)/sqrt(5)
>>> N(f)
0.e-104
>>> N(f, maxn=1000)
0.e-1336
```

In situations where such cancellations are known to occur, the `chop` options is useful. This basically replaces very small numbers in the real or imaginary portions of a number with exact zeros:

```
>>> N(f, chop=True)
0
>>> N(3 + I*f, chop=True)
3.000000000000000
```

In situations where you wish to remove meaningless digits, re-evaluation or the use of the `round` method are useful:

```
>>> Float('.1', '')*Float('.12345', '')
0.012297
>>> ans = _
>>> N(ans, 1)
0.01
>>> ans.round(2)
0.01
```

If you are dealing with a numeric expression that contains no floats, it can be evaluated to arbitrary precision. To round the result relative to a given decimal, the `round` method is useful:

```
>>> v = 10*pi + cos(1)
>>> N(v)
31.9562288417661
>>> v.round(3)
31.956
```

Sums and integrals

Sums (in particular, infinite series) and integrals can be used like regular closed-form expressions, and support arbitrary-precision evaluation:

```
>>> var('n x')
(n, x)
>>> Sum(1/n**n, (n, 1, oo)).evalf()
1.29128599706266
>>> Integral(x**(-x), (x, 0, 1)).evalf()
1.29128599706266
>>> Sum(1/n**n, (n, 1, oo)).evalf(50)
1.2912859970626635404072825905956005414986193682745
>>> Integral(x**(-x), (x, 0, 1)).evalf(50)
1.2912859970626635404072825905956005414986193682745
>>> (Integral(exp(-x**2), (x, -oo, oo)) ** 2).evalf(30)
3.14159265358979323846264338328
```

By default, the tanh-sinh quadrature algorithm is used to evaluate integrals. This algorithm is very efficient and robust for smooth integrands (and even integrals with endpoint singularities), but may struggle with integrals that are highly oscillatory or have mid-interval discontinuities. In many cases, `evalf`/`N` will correctly estimate the error. With the following integral, the result is accurate but only good to four digits:

```
>>> f = abs(sin(x))
>>> Integral(abs(sin(x)), (x, 0, 4)).evalf()
2.346
```

It is better to split this integral into two pieces:

```
>>> (Integral(f, (x, 0, pi)) + Integral(f, (x, pi, 4))).evalf()
2.34635637913639
```

A similar example is the following oscillatory integral:

```
>>> Integral(sin(x)/x**2, (x, 1, oo)).evalf(maxn=20)
0.5
```

It can be dealt with much more efficiently by telling `evalf` or `N` to use an oscillatory quadrature algorithm:

```
>>> Integral(sin(x)/x**2, (x, 1, oo)).evalf(quad='osc')
0.504067061906928
>>> Integral(sin(x)/x**2, (x, 1, oo)).evalf(20, quad='osc')
0.50406706190692837199
```

Oscillatory quadrature requires an integrand containing a factor $\cos(ax+b)$ or $\sin(ax+b)$. Note that many other oscillatory integrals can be transformed to this form with a change of variables:

```
>>> init_printing(use_unicode=False, wrap_line=False)
>>> intgr1 = Integral(sin(1/x), (x, 0, 1)).transform(x, 1/x)
>>> intgr1
oo
/
|
| sin(x)
| ----- dx
|      2
|     x
|
/
1
>>> N(intgr1, quad='osc')
0.504067061906928
```

Infinite series use direct summation if the series converges quickly enough. Otherwise, extrapolation methods (generally the Euler-Maclaurin formula but also Richardson extrapolation) are used to speed up convergence. This allows high-precision evaluation of slowly convergent series:

```
>>> var('k')
k
>>> Sum(1/k**2, (k, 1, oo)).evalf()
1.64493406684823
>>> zeta(2).evalf()
1.64493406684823
>>> Sum(1/k-log(1+1/k), (k, 1, oo)).evalf()
0.577215664901533
>>> Sum(1/k-log(1+1/k), (k, 1, oo)).evalf(50)
0.57721566490153286060651209008240243104215933593992
>>> EulerGamma.evalf(50)
0.57721566490153286060651209008240243104215933593992
```

The Euler-Maclaurin formula is also used for finite series, allowing them to be approximated quickly without evaluating all terms:

```
>>> Sum(1/k, (k, 10000000, 20000000)).evalf()
0.693147255559946
```

Note that `evalf` makes some assumptions that are not always optimal. For fine-tuned control over numerical summation, it might be worthwhile to manually use the method `Sum.euler_maclaurin`.

Special optimizations are used for rational hypergeometric series (where the term is a product of polynomials, powers, factorials, binomial coefficients and the like). `N/evalf` sum series of this type very rapidly to high precision. For example, this Ramanujan formula for π can be summed to 10,000 digits in a fraction of a second with a simple command:

```
>>> f = factorial
>>> n = Symbol('n', integer=True)
```

(continues on next page)

(continued from previous page)

```

>>> R = 9801/sqrt(8)/Sum(f(4*n)*(1103+26390*n)/f(n)**4/396**(4*n),
...                      (n, 0, oo))
>>> N(R, 10000)
3.
↪ 141592653589793238462643383279502884197169399375105820974944592307816406286208
99862803482534211706798214808651328230664709384460955058223172535940812848111745
02841027019385211055596446229489549303819644288109756659334461284756482337867831
...

```

Numerical simplification

The function `nsimplify` attempts to find a formula that is numerically equal to the given input. This feature can be used to guess an exact formula for an approximate floating-point input, or to guess a simpler formula for a complicated symbolic input. The algorithm used by `nsimplify` is capable of identifying simple fractions, simple algebraic expressions, linear combinations of given constants, and certain elementary functional transformations of any of the preceding.

Optionally, `nsimplify` can be passed a list of constants to include (e.g. `pi`) and a minimum numerical tolerance. Here are some elementary examples:

```

>>> nsimplify(0.1)
1/10
>>> nsimplify(6.28, [pi], tolerance=0.01)
2*pi
>>> nsimplify(pi, tolerance=0.01)
22/7
>>> nsimplify(pi, tolerance=0.001)
355
---
113
>>> nsimplify(0.33333, tolerance=1e-4)
1/3
>>> nsimplify(2.0**(1/3.), tolerance=0.001)
635
---
504
>>> nsimplify(2.0**(1/3.), tolerance=0.001, full=True)
3
√ 2

```

Here are several more advanced examples:

```

>>> nsimplify(Float('0.13019886629986772369127970337',30), [pi, E])
1
-----
5*pi
---- + 2*e
7
>>> nsimplify(cos(atan('1/3'))))
_____

```

(continues on next page)

(continued from previous page)

```

3*\ / 10
-----
      10
>>> nsimplify(4/(1+sqrt(5)), [GoldenRatio])
-2 + 2*GoldenRatio
>>> nsimplify(2 + exp(2*atan('1/4')*I))
49      8*I
-- + ---
17      17
>>> nsimplify((1/(exp(3*pi*I/5)+1)))
1      / \ 5      1
- - I* /  ----- + -
2      \ /      10      4
>>> nsimplify(I**I, [pi])
-pi
----
 2
e
>>> n = Symbol('n')
>>> nsimplify(Sum(1/n**2, (n, 1, oo)), [pi])
2
pi
---
 6
>>> nsimplify(gamma('1/4')*gamma('3/4'), [pi])
\ / 2 *pi

```

Numeric Computation

Symbolic computer algebra systems like SymPy facilitate the construction and manipulation of mathematical expressions. Unfortunately when it comes time to evaluate these expressions on numerical data, symbolic systems often have poor performance.

Fortunately SymPy offers a number of easy-to-use hooks into other numeric systems, allowing you to create mathematical expressions in SymPy and then ship them off to the numeric system of your choice. This page documents many of the options available including the math library, the popular array computing package numpy, code generation in Fortran or C, and the use of the array compiler Aesara.

Subs/evalf

Subs is the slowest but simplest option. It runs at SymPy speeds. The `.subs(...).evalf()` method can substitute a numeric value for a symbolic one and then evaluate the result within SymPy.

```
>>> from sympy import *
>>> from sympy.abc import x
>>> expr = sin(x)/x
>>> expr.evalf(subs={x: 3.14})
0.000507214304613640
```

This method is slow. You should use this method production only if performance is not an issue. You can expect `.subs` to take tens of microseconds. It can be useful while prototyping or if you just want to see a value once.

Lambdify

The `lambdify` function translates SymPy expressions into Python functions, leveraging a variety of numerical libraries. It is used as follows:

```
>>> from sympy import *
>>> from sympy.abc import x
>>> expr = sin(x)/x
>>> f = lambdify(x, expr)
>>> f(3.14)
0.000507214304614
```

Here `lambdify` makes a function that computes $f(x) = \sin(x)/x$. By default `lambdify` relies on implementations in the `math` standard library. This numerical evaluation takes on the order of hundreds of nanoseconds, roughly two orders of magnitude faster than the `.subs` method. This is the speed difference between SymPy and raw Python.

`Lambdify` can leverage a variety of numerical backends. By default it uses the `math` library. However it also supports `mpmath` and most notably, `numpy`. Using the `numpy` library gives the generated function access to powerful vectorized ufuncs that are backed by compiled C code.

```
>>> from sympy import *
>>> from sympy.abc import x
>>> expr = sin(x)/x
>>> f = lambdify(x, expr, "numpy")
```

```
>>> import numpy
>>> data = numpy.linspace(1, 10, 10000)
>>> f(data)
[ 0.84147098  0.84119981  0.84092844 ... -0.05426074 -0.05433146
 -0.05440211]
```

If you have array-based data this can confer a considerable speedup, on the order of 10 nanoseconds per element. Unfortunately `numpy` incurs some start-up time and introduces an overhead of a few microseconds.

CuPy is a NumPy-compatible array library that mainly runs on CUDA, but has increasing support for other GPU manufacturers. It can in many cases be used as a drop-in replacement

for numpy.

```
>>> f = lambdify(x, expr, "cupy")
>>> import cupy as cp
>>> data = cp.linspace(1, 10, 10000)
>>> y = f(data) # perform the computation
>>> cp.asnumpy(y) # explicitly copy from GPU to CPU / numpy array
[ 0.84147098  0.84119981  0.84092844 ... -0.05426074 -0.05433146
 -0.05440211]
```

JAX is a similar alternative to CuPy that provides GPU and TPU acceleration via just-in-time compilation to XLA. It too, can in some cases, be used as a drop-in replacement for numpy.

```
>>> f = lambdify(x, expr, "jax")
>>> import jax.numpy as jnp
>>> data = jnp.linspace(1, 10, 10000)
>>> y = f(data) # perform the computation
>>> numpy.asarray(y) # explicitly copy to CPU / numpy array
array([ 0.84147096,  0.8411998 ,  0.84092844, ..., -0.05426079,
        -0.05433151, -0.05440211], dtype=float32)
```

uFuncify

The `autowrap` module contains methods that help in efficient computation.

- [autowrap](#) (page 1114) method for compiling code generated by the [codegen](#) (page 1108) module, and wrap the binary for use in python.
- [binary_function](#) (page 1116) method automates the steps needed to autowrap the SymPy expression and attaching it to a Function object with `implemented_function()`.
- [ufuncify](#) (page 1116) generates a binary function that supports broadcasting on numpy arrays using different backends that are faster as compared to `subs/evalf` and `lambdify`.

The API reference of all the above is listed here: [sympy.utilities.autowrap\(\)](#) (page 2039).

Aesara

SymPy has a strong connection with [Aesara](#), a mathematical array compiler. SymPy expressions can be easily translated to Aesara graphs and then compiled using the Aesara compiler chain.

```
>>> from sympy import *
>>> from sympy.abc import x
>>> expr = sin(x)/x
```

```
>>> from sympy.printing.aesaracode import aesara_function
>>> f = aesara_function([x], [expr])
```

If array broadcasting or types are desired then Aesara requires this extra information


```
>>> f = aesara_function([x], [expr], dims={x: 1}, dtypes={x: 'float64'})
```

Aesara has a more sophisticated code generation system than SymPy's C/Fortran code printers. Among other things it handles common sub-expressions and compilation onto the GPU. Aesara also supports SymPy Matrix and Matrix Expression objects.

So Which Should I Use?

The options here were listed in order from slowest and least dependencies to fastest and most dependencies. For example, if you have Aesara installed then that will often be the best choice. If you don't have Aesara but do have f2py then you should use ufuncify. If you have been comfortable using lambdify with the numpy module, but have a GPU, CuPy and JAX can provide substantial speedups with little effort.

Tool	Speed	Qualities	Dependencies
subs/evalf	50us	Simple	None
lambdify	1us	Scalar functions	math
lambdify-numpy	10ns	Vector functions	numpy
ufuncify	10ns	Complex vector expressions	f2py, Cython
lambdify-cupy	10ns	Vector functions on GPUs	cupy
lambdify-jax	10ns	Vector functions on CPUs, GPUs and TPUs	jax
Aesara	10ns	Many outputs, CSE, GPUs	Aesara

Term Rewriting

Term rewriting is a very general class of functionalities which are used to convert expressions of one type in terms of expressions of different kind. For example expanding, combining and converting expressions apply to term rewriting, and also simplification routines can be included here. Currently SymPy has several functions and basic built-in methods for performing various types of rewriting.

Expanding

The simplest rewrite rule is expanding expressions into a `_sparse_` form. Expanding has several flavors and include expanding complex valued expressions, arithmetic expand of products and powers but also expanding functions in terms of more general functions is possible. Below are listed all currently available expand rules.

Expanding of arithmetic expressions involving products and powers:

```
>>> from sympy import *
>>> x, y, z = symbols('x,y,z')
>>> ((x + y)*(x - y)).expand(basic=True)
x**2 - y**2
>>> ((x + y + z)**2).expand(basic=True)
x**2 + 2*x*y + 2*x*z + y**2 + 2*y*z + z**2
```

Arithmetic expand is done by default in `expand()` so the keyword `basic` can be omitted. However you can set `basic=False` to avoid this type of expand if you use rules described below. This give complete control on what is done with the expression.

Another type of expand rule is expanding complex valued expressions and putting them into a normal form. For this `complex` keyword is used. Note that it will always perform arithmetic expand to obtain the desired normal form:

```
>>> (x + I*y).expand(complex=True)
re(x) + I*re(y) + I*im(x) - im(y)
```

```
>>> sin(x + I*y).expand(complex=True)
sin(re(x) - im(y))*cosh(re(y) + im(x)) + I*cos(re(x) - im(y))*sinh(re(y) +
↪im(x))
```

Note also that the same behavior can be obtained by using `as_real_imag()` method. However it will return a tuple containing the real part in the first place and the imaginary part in the other. This can be also done in a two step process by using `collect` function:

```
>>> (x + I*y).as_real_imag()
(re(x) - im(y), re(y) + im(x))
```

```
>>> collect((x + I*y).expand(complex=True), I, evaluate=False)
{1: re(x) - im(y), I: re(y) + im(x)}
```

There is also possibility for expanding expressions in terms of expressions of different kind. This is very general type of expanding and usually you would use `rewrite()` to do specific type of rewrite:

```
>>> GoldenRatio.expand(func=True)
1/2 + sqrt(5)/2
```

Common Subexpression Detection and Collection

Before evaluating a large expression, it is often useful to identify common subexpressions, collect them and evaluate them at once. This is implemented in the `cse` function. Examples:

```
>>> from sympy import cse, sqrt, sin, pprint
>>> from sympy.abc import x

>>> pprint(cse(sqrt(sin(x))), use_unicode=True)
(
  (
    [], [
       $\sqrt{\sin(x)}$ 
    ]
  )
)

>>> pprint(cse(sqrt(sin(x)+5)*sqrt(sin(x)+4)), use_unicode=True)
(
  (
    [(x0, sin(x))], [
       $\sqrt{x_0 + 4} \cdot \sqrt{x_0 + 5}$ 
    ]
  )
)

>>> pprint(cse(sqrt(sin(x+1) + 5 + cos(y))*sqrt(sin(x+1) + 4 + cos(y))),
...         use_unicode=True)
(
  (
    [(x0, sin(x + 1) + cos(y))], [
       $\sqrt{x_0 + 4} \cdot \sqrt{x_0 + 5}$ 
    ]
  )
)

>>> pprint(cse((x-y)*(z-y) + sqrt((x-y)*(z-y))), use_unicode=True)
(
  (
    [(x0, (x - y)*(-y + z))], [
       $\sqrt{x_0 + x_0}$ 
    ]
  )
)
```

Optimizations to be performed before and after common subexpressions elimination can be passed in the `optimizations` optional argument. A set of predefined basic optimizations can be applied by passing optimizations='basic':`

```
>>> pprint(cse((x-y)*(z-y) + sqrt((x-y)*(z-y)), optimizations='basic'),
...         use_unicode=True)
(
  ([(x0, -(x - y)·(y - z))], [√ x0 + x0])
)
```

However, these optimizations can be very slow for large expressions. Moreover, if speed is a concern, one can pass the option `order='none'`. Order of terms will then be dependent on hashing algorithm implementation, but speed will be greatly improved.

More information:

```
sympy.simplify.cse_main.cse(exprs, symbols=None, optimizations=None,
                           postprocess=None, order='canonical', ignore=(),
                           list=True)
```

Perform common subexpression elimination on an expression.

Parameters

exprs : list of SymPy expressions, or a single SymPy expression

The expressions to reduce.

symbols : infinite iterator yielding unique Symbols

The symbols used to label the common subexpressions which are pulled out. The `numbered_symbols` generator is useful. The default is a stream of symbols of the form “x0”, “x1”, etc. This must be an infinite iterator.

optimizations : list of (callable, callable) pairs

The (preprocessor, postprocessor) pairs of external optimization functions. Optionally ‘basic’ can be passed for a set of predefined basic optimizations. Such ‘basic’ optimizations were used by default in old implementation, however they can be really slow on larger expressions. Now, no pre or post optimizations are made by default.

postprocess : a function which accepts the two return values of `cse` and

returns the desired form of output from `cse`, e.g. if you want the replacements reversed the function might be the following lambda:
`lambda r, e: return reversed(r), e`

order : string, ‘none’ or ‘canonical’

The order by which Mul and Add arguments are processed. If set to ‘canonical’, arguments will be canonically ordered. If set to ‘none’, ordering will be faster but dependent on expressions hashes, thus machine dependent and variable. For large expressions where speed is a concern, use the setting `order='none'`.

ignore : iterable of Symbols

Substitutions containing any Symbol from `ignore` will be ignored.

list : bool, (default True)

Returns expression in list or else with same type as input (when False).

Returns

replacements : list of (Symbol, expression) pairs

All of the common subexpressions that were replaced. Subexpressions earlier in this list might show up in subexpressions later in this list.

reduced_exprs : list of SymPy expressions

The reduced expressions with all of the replacements above.

Examples

```
>>> from sympy import cse, SparseMatrix
>>> from sympy.abc import x, y, z, w
>>> cse(((w + x + y + z)*(w + y + z))/(w + x)**3)
([(x0, y + z), (x1, w + x)], [(w + x0)*(x0 + x1)/x1**3])
```

List of expressions with recursive substitutions:

```
>>> m = SparseMatrix([x + y, x + y + z])
>>> cse([(x+y)**2, x + y + z, y + z, x + z + y, m])
([(x0, x + y), (x1, x0 + z)], [x0**2, x1, y + z, x1, Matrix([
[x0],
[x1]])])
```

Note: the type and mutability of input matrices is retained.

```
>>> isinstance(_[1][-1], SparseMatrix)
True
```

The user may disallow substitutions containing certain symbols:

```
>>> cse([y**2*(x + 1), 3*y**2*(x + 1)], ignore=(y,))
([(x0, x + 1)], [x0*y**2, 3*x0*y**2])
```

The default return value for the reduced expression(s) is a list, even if there is only one expression. The *list* flag preserves the type of the input in the output:

```
>>> cse(x)
([], [x])
>>> cse(x, list=False)
([], x)
```

5.8.2 Code Generation

Contents

Several submodules in SymPy allow one to generate directly compilable and executable code in a variety of different programming languages from SymPy expressions. In addition, there are functions that generate Python importable objects that can evaluate SymPy expressions very efficiently.

We will start with a brief introduction to the components that make up the code generation capabilities of SymPy.

Introduction

There are four main levels of abstractions:

```
expression
|
code printers
|
code generators
|
autowrap
```

`sympy.utilities.autowrap` (page 2039) uses codegen, and codegen uses the code printers. `sympy.utilities.autowrap` (page 2039) does everything: it lets you go from SymPy expression to numerical function in the same Python process in one step. Codegen is actual code generation, i.e., to compile and use later, or to include in some larger project.

The code printers translate the SymPy objects into actual code, like `abs(x) -> fabs(x)` (for C).

The code printers don't print optimal code in many cases. An example of this is powers in C. `x**2` prints as `pow(x, 2)` instead of `x*x`. Other optimizations (like mathematical simplifications) should happen before the code printers.

Currently, `sympy.simplify.cse_main.cse()` (page 685) is not applied automatically anywhere in this chain. It ideally happens at the codegen level, or somewhere above it.

We will iterate through the levels below.

The following three lines will be used to setup each example:

```
>>> from sympy import *
>>> init_printing(use_unicode=True)
>>> from sympy.abc import a, e, k, n, r, t, x, y, z, T, Z
>>> from sympy.abc import beta, omega, tau
>>> f, g = symbols('f, g', cls=Function)
```

Code printers (sympy.printing)

This is where the meat of code generation is; the translation of SymPy actually more like a lightweight version of codegen for Python, and Python (`sympy.printing.pycode.pycode()` (page 2176)), and `sympy.printing.lambdarepr.lambdarepr()` (page 2170), which supports many libraries (like NumPy), and Aesara (`sympy.printing.aesaracode.aesara_function()` (page 2167)). The code printers are special cases of the other prints in SymPy (str printer, pretty printer, etc.).

An important distinction is that the code printer has to deal with assignments (using the `sympy.codegen.ast.Assignment` (page 1126) object). This serves as building blocks for the code printers and hence the codegen module. An example that shows the use of Assignment in C code:

```
>>> from sympy.codegen.ast import Assignment
>>> print(ccode(Assignment(x, y + 1)))
x = y + 1;
```

Here is another simple example of printing a C version of a SymPy expression:

```
>>> expr = (Rational(-1, 2) * Z * k * (e**2) / r)
>>> expr
      2
-Z·e ·k
-----
 2·r
>>> ccode(expr)
-1.0/2.0*Z*pow(e, 2)*k/r
>>> from sympy.codegen.ast import real, float80
>>> ccode(expr, assign_to="E", type_aliases={real: float80})
E = -1.0L/2.0L*Z*powl(e, 2)*k/r;
```

To generate code with some math functions provided by e.g. the C99 standard we need to import functions from [sympy.codegen.cfunctions](#) (page 1143):

```
>>> from sympy.codegen.cfunctions import expm1
>>> ccode(expm1(x), standard='C99')
expm1(x)
```

Piecewise expressions are converted into conditionals. If an `assign_to` variable is provided an if statement is created, otherwise the ternary operator is used. Note that if the Piecewise lacks a default term, represented by `(expr, True)` then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything. A use case for Piecewise:

```
>>> expr = Piecewise((x + 1, x > 0), (x, True))
>>> print(fcode(expr, tau))
    if (x > 0) then
        tau = x + 1
    else
        tau = x
    end if
```

The various printers also tend to support Indexed objects well. With `contract=True` these expressions will be turned into loops, whereas `contract=False` will just print the assignment expression that should be looped over:

```
>>> len_y = 5
>>> mat_1 = IndexedBase('mat_1', shape=(len_y,))
>>> mat_2 = IndexedBase('mat_2', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y-1,))
>>> i = Idx('i', len_y-1)
>>> eq = Eq(Dy[i], (mat_1[i+1] - mat_1[i]) / (mat_2[i+1] - mat_2[i]))
>>> print(jscode(eq.rhs, assign_to=eq.lhs, contract=False))
Dy[i] = (mat_1[i + 1] - mat_1[i])/(mat_2[i + 1] - mat_2[i]);
>>> Res = IndexedBase('Res', shape=(len_y,))
>>> j = Idx('j', len_y)
>>> eq = Eq(Res[j], mat_1[j]*mat_2[j])
>>> print(jscode(eq.rhs, assign_to=eq.lhs, contract=True))
for (var j=0; j<5; j++){
    Res[j] = 0;
}
for (var j=0; j<5; j++){
    for (var j=0; j<5; j++){
```

(continues on next page)

(continued from previous page)

```

        Res[j] = Res[j] + mat_1[j]*mat_2[j];
    }
}
>>> print(jscode(eq.rhs, assign_to=eq.lhs, contract=False))
Res[j] = mat_1[j]*mat_2[j];

```

Custom printing can be defined for certain types by passing a dictionary of “type”: “function” to the `user_functions` kwarg. Alternatively, the dictionary value can be a list of tuples i.e., [(argument_test, cfunction_string)]. This can be used to call a custom Octave function:

```

>>> custom_functions = {
...     "f": "existing_octave_fcn",
...     "g": [(lambda x: x.is_Matrix, "my_mat_fcn"),
...           (lambda x: not x.is_Matrix, "my_fcn")]
... }
>>> mat = Matrix([[1, x]])
>>> octave_code(f(x) + g(x) + g(mat), user_functions=custom_functions)
existing_octave_fcn(x) + my_fcn(x) + my_mat_fcn([1 x])

```

An example of Mathematica code printer:

```

>>> x_ = Function('x')
>>> expr = x_(n*T) * sin((t - n*T) / T)
>>> expr = expr / ((-T*n + t) / T)
>>> expr

```

$$\frac{T \cdot x(T \cdot n) \cdot \sin\left(\frac{-T \cdot n + t}{T}\right)}{-T \cdot n + t}$$

```

>>> expr = summation(expr, (n, -1, 1))
>>> mathematica_code(expr)
T*(x[-T]*Sin[(T + t)/T]/(T + t) + x[T]*Sin[(-T + t)/T]/(-T + t) + x[0]*Sin[t/T]/t)

```

We can go through a common expression in different languages we support and see how it works:

```

>>> k, g1, g2, r, I, S = symbols("k, gamma_1, gamma_2, r, I, S")
>>> expr = k * g1 * g2 / (r**3)
>>> expr = expr * 2 * I * S * (3 * (cos(beta))**2 - 1) / 2
>>> expr

```

$$\frac{I \cdot S \cdot \gamma_1 \cdot \gamma_2 \cdot k \cdot \left(3 \cdot \cos^2(\beta) - 1\right)}{r^3}$$

```

>>> print(jscode(expr, assign_to="H_is"))
H_is = I*S*gamma_1*gamma_2*k*(3*Math.pow(Math.cos(beta), 2) - 1)/Math.pow(r, 3);
>>> print(ccode(expr, assign_to="H_is", standard='C89'))

```

(continues on next page)

(continued from previous page)

```
H_is = I*S*gamma_1*gamma_2*k*(3*pow(cos(beta), 2) - 1)/pow(r, 3);
>>> print(fcode(expr, assign_to="H_is"))
      H_is = I*S*gamma_1*gamma_2*k*(3*cos(beta)**2 - 1)/r**3
>>> print(julia_code(expr, assign_to="H_is"))
H_is = I .* S .* gamma_1 .* gamma_2 .* k .* (3 * cos(beta) .^ 2 - 1) ./ r .^ 3
>>> print(octave_code(expr, assign_to="H_is"))
H_is = I.*S.*gamma_1.*gamma_2.*k.*(3*cos(beta).^2 - 1)./r.^3;
>>> print(rust_code(expr, assign_to="H_is"))
H_is = I*S*gamma_1*gamma_2*k*(3*beta.cos().powi(2) - 1)/r.powi(3);
>>> print(mathematica_code(expr))
I*S*gamma_1*gamma_2*k*(3*Cos[beta]^2 - 1)/r^3
```

Codegen (sympy.utilities.codegen)

This module deals with creating compilable code from SymPy expressions. This is lower level than `autowrap`, as it doesn't actually attempt to compile the code, but higher level than the printers, as it generates compilable files (including header files), rather than just code snippets.

The user friendly functions, here, are `codegen` and `make_routine`. `codegen` takes a list of (variable, expression) pairs and a language (C, F95, and Octave/Matlab are supported). It returns, as strings, a code file and a header file (for relevant languages). The variables are created as functions that return the value of the expression as output.

Note: The `codegen` callable is not in the `sympy` namespace automatically, to use it you must first import `codegen` from `sympy.utilities.codegen`

For instance:

```
>>> from sympy.utilities.codegen import codegen
>>> length, breadth, height = symbols('length, breadth, height')
>>> [(c_name, c_code), (h_name, c_header)] = \
... codegen(('volume', length*breadth*height), "C99", "test",
...         header=False, empty=False)
>>> print(c_name)
test.c
>>> print(c_code)
#include "test.h"
#include <math.h>
double volume(double breadth, double height, double length) {
    double volume_result;
    volume_result = breadth*height*length;
    return volume_result;
}
>>> print(h_name)
test.h
>>> print(c_header)
#ifndef PROJECT__TEST__H
#define PROJECT__TEST__H
```

(continues on next page)

(continued from previous page)

```
double volume(double breadth, double height, double length);
#endif
```

Various flags to codegen let you modify things. The project name for preprocessor instructions can be varied using `project`. Variables listed as global variables in `arg global_vars` will not show up as function arguments.

`language` is a case-insensitive string that indicates the source code language. Currently, C, F95 and Octave are supported. Octave generates code compatible with both Octave and Matlab.

`header` when True, a header is written on top of each source file. `empty` when True, empty lines are used to structure the code. With `argument_sequence` a sequence of arguments for the routine can be defined in a preferred order.

`prefix` defines a prefix for the names of the files that contain the source code. If omitted, the name of the first `name_expr` tuple is used.

`to_files` when True, the code will be written to one or more files with the given prefix.

Here is an example:

```
>>> [(f_name, f_code), header] = codegen(("volume", length*breadth*height),
...     "F95", header=False, empty=False, argument_sequence=(breadth, length),
...     global_vars=(height,))
>>> print(f_code)
REAL*8 function volume(breadth, length)
implicit none
REAL*8, intent(in) :: breadth
REAL*8, intent(in) :: length
volume = breadth*height*length
end function
```

The method `make_routine` creates a `Routine` object, which represents an evaluation routine for a set of expressions. This is only good for internal use by the `CodeGen` objects, as an intermediate representation from SymPy expression to generated code. It is not recommended to make a `Routine` object yourself. You should instead use `make_routine` method. `make_routine` in turn calls the `routine` method of the `CodeGen` object depending upon the language of choice. This creates the internal objects representing assignments and so on, and creates the `Routine` class with them.

The various codegen objects such as `Routine` and `Variable` aren't SymPy objects (they don't subclass from `Basic`).

For example:

```
>>> from sympy.utilities.codegen import make_routine
>>> from sympy.physics.hydrogen import R_nl
>>> expr = R_nl(3, y, x, 6)
>>> routine = make_routine('my_routine', expr)
>>> [arg.result_var for arg in routine.results]
[result5142341681397719428]
>>> [arg.expr for arg in routine.results]
[
    y / (2 - y)! - 2·x
]
```

(continues on next page)

(continued from previous page)

```

| 4·√6·(4·x) · √ ————— · e ————— · assoc_laguerre(2 - y, 2·y + 1, 4·x) |
|                                     (y + 3)!                                     |
|                                     3                                     |
|_____|
>>> [arg.name for arg in routine.arguments]
[x, y]

```

Another more complicated example with a mixture of specified and automatically-assigned names. Also has Matrix output:

```

>>> routine = make_routine('fcn', [x*y, Eq(a, 1), Eq(r, x + r), Matrix([[x, 2]])])
>>> [arg.result_var for arg in routine.results]
[result_5397460570204848505]
>>> [arg.expr for arg in routine.results]
[x·y]
>>> [arg.name for arg in routine.arguments]
[x, y, a, r, out_8598435338387848786]

```

We can examine the various arguments more closely:

```

>>> from sympy.utilities.codegen import (InputArgument, OutputArgument,
...                                     InOutArgument)
>>> [a.name for a in routine.arguments if isinstance(a, InputArgument)]
[x, y]

>>> [a.name for a in routine.arguments if isinstance(a, OutputArgument)]
[a, out_8598435338387848786]
>>> [a.expr for a in routine.arguments if isinstance(a, OutputArgument)]
[1, [x 2]]

>>> [a.name for a in routine.arguments if isinstance(a, InOutArgument)]
[r]
>>> [a.expr for a in routine.arguments if isinstance(a, InOutArgument)]
[r + x]

```

The full API reference can be viewed [here](#) (page 2045).

Autowrap

Autowrap automatically generates code, writes it to disk, compiles it, and imports it into the current session. Main functions of this module are `autowrap`, `binary_function`, and `ufuncify`.

It also automatically converts expressions containing Indexed objects into summations. The classes `IndexedBase`, `Indexed` and `Idx` represent a matrix element $M[i, j]$. See [Tensor](#) (page 1387) for more on this.

`autowrap` creates a wrapper using `f2py` or `Cython` and creates a numerical function.

Note: The autowrap callable is not in the sympy namespace automatically, to use it you must

first import autowrap from sympy.utilities.autowrap

The callable returned from autowrap() is a binary Python function, not a SymPy object. For example:

```
>>> from sympy.utilities.autowrap import autowrap
>>> expr = ((x - y + z)**(13)).expand()
>>> binary_func = autowrap(expr)
>>> binary_func(1, 4, 2)
-1.0
```

The various flags available with autowrap() help to modify the services provided by the method. The argument tempdir tells autowrap to compile the code in a specific directory, and leave the files intact when finished. For instance:

```
>>> from sympy.utilities.autowrap import autowrap
>>> from sympy.physics.qho_1d import psi_n
>>> x_ = IndexedBase('x')
>>> y_ = IndexedBase('y')
>>> m = symbols('m', integer=True)
>>> i = Idx('i', m)
>>> qho = autowrap(Eq(y_[i], psi_n(0, x_[i], m, omega)), tempdir='/tmp')
```

Checking the Fortran source code in the directory specified reveals this:

```
subroutine autofunc(m, omega, x, y)
implicit none
INTEGER*4, intent(in) :: m
REAL*8, intent(in) :: omega
REAL*8, intent(in), dimension(1:m) :: x
REAL*8, intent(out), dimension(1:m) :: y
INTEGER*4 :: i

REAL*8, parameter :: hbar = 1.05457162d-34
REAL*8, parameter :: pi = 3.14159265358979d0
do i = 1, m
    y(i) = (m*omega)**(1.0d0/4.0d0)*exp(-4.74126166983329d+33*m*omega*x(i &
        )**2)/(hbar**(1.0d0/4.0d0)*pi**(1.0d0/4.0d0))
end do

end subroutine
```

Using the argument args along with it changes argument sequence:

```
>>> eq = Eq(y_[i], psi_n(0, x_[i], m, omega))
>>> qho = autowrap(eq, tempdir='/tmp', args=[y, x, m, omega])
```

yields:

```
subroutine autofunc(y, x, m, omega)
implicit none
INTEGER*4, intent(in) :: m
REAL*8, intent(in) :: omega
REAL*8, intent(out), dimension(1:m) :: y
```

(continues on next page)

(continued from previous page)

```

REAL*8, intent(in), dimension(1:m) :: x
INTEGER*4 :: i

REAL*8, parameter :: hbar = 1.05457162d-34
REAL*8, parameter :: pi = 3.14159265358979d0
do i = 1, m
    y(i) = (m*omega)**(1.0d0/4.0d0)*exp(-4.74126166983329d+33*m*omega*x(i &
        )**2)/(hbar**(1.0d0/4.0d0)*pi**(1.0d0/4.0d0))
end do

end subroutine

```

The argument `verbose` is boolean, optional and if `True`, `autowrap` will not mute the command line backends. This can be helpful for debugging.

The argument `language` and `backend` are used to change defaults: Fortran and f2py to C and Cython. The argument `helpers` is used to define auxiliary expressions needed for the main expression. If the main expression needs to call a specialized function it should be put in the `helpers` iterable. `autowrap` will then make sure that the compiled main expression can link to the helper routine. Items should be tuples with (`<function_name>`, `<sympy_expression>`, `<arguments>`). It is mandatory to supply an argument sequence to helper routines.

Another method available at the `autowrap` level is `binary_function`. It returns a sympy function. The advantage is that we can have very fast functions as compared to SymPy speeds. This is because we will be using compiled functions with SymPy attributes and methods. An illustration:

```

>>> from sympy.utilities.autowrap import binary_function
>>> from sympy.physics.hydrogen import R_nl
>>> psi_nl = R_nl(1, 0, a, r)
>>> f = binary_function('f', psi_nl)
>>> f(a, r).evalf(3, subs={a: 1, r: 2})
0.766

```

While NumPy operations are very efficient for vectorized data but they sometimes incur unnecessary costs when chained together. Consider the following operation

```

>>> x = get_numpy_array(...)
>>> y = sin(x) / x

```

The operators `sin` and `/` call routines that execute tight for loops in C. The resulting computation looks something like this

```

for(int i = 0; i < n; i++)
{
    temp[i] = sin(x[i]);
}
for(int i = 0; i < n; i++)
{
    y[i] = temp[i] / x[i];
}

```

This is slightly sub-optimal because

1. We allocate an extra `temp` array

2. We walk over x memory twice when once would have been sufficient

A better solution would fuse both element-wise operations into a single for loop

```
for(int i = i; i < n; i++)
{
    y[i] = sin(x[i]) / x[i];
}
```

Statically compiled projects like NumPy are unable to take advantage of such optimizations. Fortunately, SymPy is able to generate efficient low-level C or Fortran code. It can then depend on projects like Cython or f2py to compile and reconnect that code back up to Python. Fortunately this process is well automated and a SymPy user wishing to make use of this code generation should call the `ufuncify` function.

`ufuncify` is the third method available with Autowrap module. It basically implies ‘Universal functions’ and follows an ideology set by NumPy. The main point of `ufuncify` as compared to `autowrap` is that it allows arrays as arguments and can operate in an element-by-element fashion. The core operation done element-wise is in accordance to NumPy’s array broadcasting rules. See [this](#) for more.

```
>>> from sympy import *
>>> from sympy.abc import x
>>> expr = sin(x)/x
```

```
>>> from sympy.utilities.autowrap import ufuncify
>>> f = ufuncify([x], expr)
```

This function `f` consumes and returns a NumPy array. Generally `ufuncify` performs at least as well as `lambdify`. If the expression is complicated then `ufuncify` often significantly outperforms the NumPy backed solution. Jensen has a good [blog post](#) on this topic.

Let us see an example for some quantitative analysis:

```
>>> from sympy.physics.hydrogen import R_nl
>>> expr = R_nl(3, 1, x, 6)
>>> expr
```

$$\frac{8 \cdot x \cdot (4 - 4 \cdot x) \cdot e^{-2 \cdot x}}{3}$$

The `lambdify` function translates SymPy expressions into Python functions, leveraging a variety of numerical libraries. By default `lambdify` relies on implementations in the `math` standard library. Naturally, Raw Python is faster than SymPy. However it also supports `mpmath` and most notably, `numpy`. Using the NumPy library gives the generated function access to powerful vectorized ufuncs that are backed by compiled C code.

Let us compare the speeds:

```
>>> from sympy.utilities.autowrap import ufuncify
>>> from sympy.utilities.lambdify import lambdify
>>> fn_numpy = lambdify(x, expr, 'numpy')
>>> fn_fortran = ufuncify([x], expr, backend='f2py')
>>> from numpy import linspace
>>> xx = linspace(0, 1, 5)
```

(continues on next page)

(continued from previous page)

```
>>> fn_numpy(xx)
[ 0.          1.21306132  0.98101184  0.44626032  0.          ]
>>> fn_fortran(xx)
[ 0.          1.21306132  0.98101184  0.44626032  0.          ]
>>> import timeit
>>> timeit.timeit('fn_numpy(xx)', 'from __main__ import fn_numpy, xx',
↳number=10000)
0.18891601900395472
>>> timeit.timeit('fn_fortran(xx)', 'from __main__ import fn_fortran, xx',
↳number=10000)
0.004707066000264604
```

The options available with `ufuncify` are more or less the same as those available with `autowrap`.

There are other facilities available with SymPy to do efficient numeric computation. See [this](#) (page 1102) page for a comparison among them.

Classes and functions for rewriting expressions (`sympy.codegen.rewriting`)

Classes and functions useful for rewriting expressions for optimized code generation. Some languages (or standards thereof), e.g. C99, offer specialized math functions for better performance and/or precision.

Using the `optimize` function in this module, together with a collection of rules (represented as instances of `Optimization`), one can rewrite the expressions for this purpose:

```
>>> from sympy import Symbol, exp, log
>>> from sympy.codegen.rewriting import optimize, optims_c99
>>> x = Symbol('x')
>>> optimize(3*exp(2*x) - 3, optims_c99)
3*expm1(2*x)
>>> optimize(exp(2*x) - 1 - exp(-33), optims_c99)
expm1(2*x) - exp(-33)
>>> optimize(log(3*x + 3), optims_c99)
log1p(x) + log(3)
>>> optimize(log(2*x + 3), optims_c99)
log(2*x + 3)
```

The `optims_c99` imported above is tuple containing the following instances (which may be imported from `sympy.codegen.rewriting`):

- `expm1_opt`
- `log1p_opt`
- `exp2_opt`
- `log2_opt`
- `log2const_opt`

class `sympy.codegen.rewriting.FuncMinusOneOptim`(*func*, *func_m_1*,
opportunistic=True)

Specialization of `ReplaceOptim` for functions evaluating “ $f(x) - 1$ ”.

Parameters**func :**

The function which is subtracted by one.

func_m_1 :The specialized function evaluating $\text{func}(x) - 1$.**opportunistic :** bool

When True, apply the transformation as long as the magnitude of the remaining number terms decreases. When False, only apply the transformation if it completely eliminates the number term.

Explanation

Numerical functions which go toward one as x go toward zero is often best implemented by a dedicated function in order to avoid catastrophic cancellation. One such example is `expm1(x)` in the C standard library which evaluates $\exp(x) - 1$. Such functions preserves many more significant digits when its argument is much smaller than one, compared to subtracting one afterwards.

Examples

```
>>> from sympy import symbols, exp
>>> from sympy.codegen.rewriting import FuncMinusOneOptim
>>> from sympy.codegen.cfunctions import expm1
>>> x, y = symbols('x y')
>>> expm1_opt = FuncMinusOneOptim(exp, expm1)
>>> expm1_opt(exp(x) + 2*exp(5*y) - 3)
expm1(x) + 2*expm1(5*y)
```

replace_in_Add(*e*)passed as second argument to `Basic.replace(...)`**class** `sympy.codegen.rewriting.Optimization`(*cost_function=None, priority=1*)

Abstract base class for rewriting optimization.

Subclasses should implement `__call__` taking an expression as argument.**Parameters****cost_function :** callable returning number**priority :** number**class** `sympy.codegen.rewriting.ReplaceOptim`(*query, value, **kwargs*)Rewriting optimization calling `replace` on expressions.**Parameters****query :**First argument passed to `replace`.**value :**Second argument passed to `replace`.

Explanation

The instance can be used as a function on expressions for which it will apply the `replace` method (see `sympy.core.basic.Basic.replace()` (page 937)).

Examples

```
>>> from sympy import Symbol
>>> from sympy.codegen.rewriting import ReplaceOptim
>>> from sympy.codegen.cfunctions import exp2
>>> x = Symbol('x')
>>> exp2_opt = ReplaceOptim(lambda p: p.is_Pow and p.base == 2,
... lambda p: exp2(p.exp))
>>> exp2_opt(2**x)
exp2(x)
```

```
sympy.codegen.rewriting.create_expand_pow_optimization(limit, *,
                                                         base_req=<function
                                                         <lambda>>>)
```

Creates an instance of `ReplaceOptim` (page 1119) for expanding Pow.

Parameters

limit : int

The highest power which is expanded into multiplication.

base_req : function returning bool

Requirement on base for expansion to happen, default is to return the `is_symbol` attribute of the base.

Explanation

The requirements for expansions are that the base needs to be a symbol and the exponent needs to be an Integer (and be less than or equal to limit).

Examples

```
>>> from sympy import Symbol, sin
>>> from sympy.codegen.rewriting import create_expand_pow_optimization
>>> x = Symbol('x')
>>> expand_opt = create_expand_pow_optimization(3)
>>> expand_opt(x**5 + x**3)
x**5 + x*x*x
>>> expand_opt(x**5 + x**3 + sin(x)**3)
x**5 + sin(x)**3 + x*x*x
>>> opt2 = create_expand_pow_optimization(3, base_req=lambda b: not b.is_
->Function)
>>> opt2((x+1)**2 + sin(x)**2)
sin(x)**2 + (x + 1)*(x + 1)
```


`sympy.codegen.rewriting.optimize(expr, optimizations)`

Apply optimizations to an expression.

Parameters

expr : expression

optimizations : iterable of Optimization instances

The optimizations will be sorted with respect to priority (highest first).

Examples

```
>>> from sympy import log, Symbol
>>> from sympy.codegen.rewriting import optims_c99, optimize
>>> x = Symbol('x')
>>> optimize(log(x+3)/log(2) + log(x**2 + 1), optims_c99)
log1p(x**2) + log2(x + 3)
```

Additional AST nodes for operations on matrices. The nodes in this module are meant to represent optimization of matrix expressions within codegen's target languages that cannot be represented by SymPy expressions.

As an example, we can use `sympy.codegen.rewriting.optimize()` (page 1120) and the `matinv_opt` optimization provided in `sympy.codegen.rewriting` (page 1118) to transform matrix multiplication under certain assumptions:

```
>>> from sympy import symbols, MatrixSymbol
>>> n = symbols('n', integer=True)
>>> A = MatrixSymbol('A', n, n)
>>> x = MatrixSymbol('x', n, 1)
>>> expr = A**(-1) * x
>>> from sympy import assuming, Q
>>> from sympy.codegen.rewriting import matinv_opt, optimize
>>> with assuming(Q.fullrank(A)):
...     optimize(expr, [matinv_opt])
MatrixSolve(A, vector=x)
```

class `sympy.codegen.matrix_nodes.MatrixSolve(*args, **kwargs)`

Represents an operation to solve a linear matrix equation.

Parameters

matrix : MatrixSymbol

Matrix representing the coefficients of variables in the linear equation. This matrix must be square and full-rank (i.e. all columns must be linearly independent) for the solving operation to be valid.

vector : MatrixSymbol

One-column matrix representing the solutions to the equations represented in matrix.

Examples

```
>>> from sympy import symbols, MatrixSymbol
>>> from sympy.codegen.matrix_nodes import MatrixSolve
>>> n = symbols('n', integer=True)
>>> A = MatrixSymbol('A', n, n)
>>> x = MatrixSymbol('x', n, 1)
>>> from sympy.printing.numpy import NumPyPrinter
>>> NumPyPrinter().doprint(MatrixSolve(A, x))
'numpy.linalg.solve(A, x)'
>>> from sympy import octave_code
>>> octave_code(MatrixSolve(A, x))
'A \ x'
```

Tools for simplifying expressions using approximations (sympy.codegen.approximations)

```
class sympy.codegen.approximations.SeriesApprox(bounds, reltol, max_order=4,
                                                n_point_checks=4, **kwargs)
```

Approximates functions by expanding them as a series.

Parameters

bounds : dict

Mapping expressions to length 2 tuple of bounds (low, high).

reltol : number

Threshold for when to ignore a term. Taken relative to the largest lower bound among bounds.

max_order : int

Largest order to include in series expansion

n_point_checks : int (even)

The validity of an expansion (with respect to reltol) is checked at discrete points (linearly spaced over the bounds of the variable). The number of points used in this numerical check is given by this number.

Examples

```
>>> from sympy import sin, pi
>>> from sympy.abc import x, y
>>> from sympy.codegen.rewriting import optimize
>>> from sympy.codegen.approximations import SeriesApprox
>>> bounds = {x: (-.1, .1), y: (pi-1, pi+1)}
>>> series_approx2 = SeriesApprox(bounds, reltol=1e-2)
>>> series_approx3 = SeriesApprox(bounds, reltol=1e-3)
>>> series_approx8 = SeriesApprox(bounds, reltol=1e-8)
>>> expr = sin(x)*sin(y)
```

(continues on next page)

(continued from previous page)

```
>>> optimize(expr, [series_approx2])
x*(-y + (y - pi)**3/6 + pi)
>>> optimize(expr, [series_approx3])
(-x**3/6 + x)*sin(y)
>>> optimize(expr, [series_approx8])
sin(x)*sin(y)
```

class `sympy.codegen.approximations.SumApprox(bounds, reltol, **kwargs)`

Approximates sum by neglecting small terms.

Parameters

bounds : dict

Mapping expressions to length 2 tuple of bounds (low, high).

reltol : number

Threshold for when to ignore a term. Taken relative to the largest lower bound among bounds.

Explanation

If terms are expressions which can be determined to be monotonic, then bounds for those expressions are added.

Examples

```
>>> from sympy import exp
>>> from sympy.abc import x, y, z
>>> from sympy.codegen.rewriting import optimize
>>> from sympy.codegen.approximations import SumApprox
>>> bounds = {x: (-1, 1), y: (1000, 2000), z: (-10, 3)}
>>> sum_approx3 = SumApprox(bounds, reltol=1e-3)
>>> sum_approx2 = SumApprox(bounds, reltol=1e-2)
>>> sum_approx1 = SumApprox(bounds, reltol=1e-1)
>>> expr = 3*(x + y + exp(z))
>>> optimize(expr, [sum_approx3])
3*(x + y + exp(z))
>>> optimize(expr, [sum_approx2])
3*y + 3*exp(z)
>>> optimize(expr, [sum_approx1])
3*y
```

Classes for abstract syntax trees (sympy.codegen.ast)

Types used to represent a full function/module as an Abstract Syntax Tree.

Most types are small, and are merely used as tokens in the AST. A tree diagram has been included below to illustrate the relationships between the AST types.

AST Type Tree

```

*Basic*
|
|_ CodegenAST
|   |
|   |_ ---> AssignmentBase
|   |   |
|   |   |_ ---> Assignment
|   |   |_ ---> AugmentedAssignment
|   |       |
|   |       |_ ---> AddAugmentedAssignment
|   |       |_ ---> SubAugmentedAssignment
|   |       |_ ---> MulAugmentedAssignment
|   |       |_ ---> DivAugmentedAssignment
|   |       |_ ---> ModAugmentedAssignment
|   |
|   |_ ---> CodeBlock
|   |
|   |_ ---> Token
|       |
|       |_ ---> Attribute
|       |_ ---> For
|       |_ ---> String
|       |   |
|       |   |_ ---> QuotedString
|       |   |_ ---> Comment
|       |_ ---> Type
|       |   |
|       |   |_ ---> IntBaseType
|       |       |
|       |       |_ ---> _SizedIntType
|       |           |
|       |           |_ ---> SignedIntType
|       |           |_ ---> UnsignedIntType
|       |   |_ ---> FloatBaseType
|       |       |
|       |       |_ ---> FloatType
|       |       |_ ---> ComplexBaseType
|       |           |
|       |           |_ ---> ComplexType
|       |_ ---> Node
|       |   |
|       |   |_ ---> Variable
|       |       |
|       |       |_ ---> Pointer
|       |   |_ ---> FunctionPrototype
|       |       |
|       |       |_ ---> FunctionDefinition
|       |_ ---> Element
|       |_ ---> Declaration
|       |_ ---> While
|       |_ ---> Scope
|       |_ ---> Stream
|       |_ ---> Print

```

(continues on next page)

(continued from previous page)

```

|--->FunctionCall
|--->BreakToken
|--->ContinueToken
|--->NoneToken
|--->Return

```

Predefined types

A number of Type instances are provided in the `sympy.codegen.ast` module for convenience. Perhaps the two most common ones for code-generation (of numeric codes) are `float32` and `float64` (known as single and double precision respectively). There are also precision generic versions of Types (for which the codeprinters selects the underlying data type at time of printing): `real`, `integer`, `complex_`, `bool_`.

The other Type instances defined are:

- `intc`: Integer type used by C's "int".
- `intp`: Integer type used by C's "unsigned".
- `int8`, `int16`, `int32`, `int64`: n-bit integers.
- `uint8`, `uint16`, `uint32`, `uint64`: n-bit unsigned integers.
- `float80`: known as "extended precision" on modern x86/amd64 hardware.
- `complex64`: Complex number represented by two `float32` numbers
- `complex128`: Complex number represented by two `float64` numbers

Using the nodes

It is possible to construct simple algorithms using the AST nodes. Let's construct a loop applying Newton's method:

```

>>> from sympy import symbols, cos
>>> from sympy.codegen.ast import While, Assignment, aug_assign, Print
>>> t, dx, x = symbols('tol delta val')
>>> expr = cos(x) - x**3
>>> whl = While(abs(dx) > t, [
...     Assignment(dx, -expr/expr.diff(x)),
...     aug_assign(x, '+', dx),
...     Print([x])
... ])
>>> from sympy import pycode
>>> py_str = pycode(whl)
>>> print(py_str)
while (abs(delta) > tol):
    delta = (val**3 - math.cos(val))/(-3*val**2 - math.sin(val))
    val += delta
    print(val)
>>> import math
>>> tol, val, delta = 1e-5, 0.5, float('inf')

```

(continues on next page)

(continued from previous page)

```
>>> exec(py_str)
1.1121416371
0.909672693737
0.867263818209
0.865477135298
0.865474033111
>>> print('%3.1g' % (math.cos(val) - val**3))
-3e-11
```

If we want to generate Fortran code for the same while loop we simple call fcode:

```
>>> from sympy import fcode
>>> print(fcode(whl, standard=2003, source_format='free'))
do while (abs(delta) > tol)
    delta = (val**3 - cos(val))/(-3*val**2 - sin(val))
    val = val + delta
    print *, val
end do
```

There is a function constructing a loop (or a complete function) like this in [sympy.codegen.algorithms](#) (page 1157).

class sympy.codegen.ast.Assignment(*lhs*, *rhs*)

Represents variable assignment for code generation.

Parameters

lhs : Expr

SymPy object representing the lhs of the expression. These should be singular objects, such as one would use in writing code. Notable types include Symbol, MatrixSymbol, MatrixElement, and Indexed. Types that subclass these types are also supported.

rhs : Expr

SymPy object representing the rhs of the expression. This can be any type, provided its shape corresponds to that of the lhs. For example, a Matrix type can be assigned to MatrixSymbol, but not to Symbol, as the dimensions will not align.

Examples

```
>>> from sympy import symbols, MatrixSymbol, Matrix
>>> from sympy.codegen.ast import Assignment
>>> x, y, z = symbols('x, y, z')
>>> Assignment(x, y)
Assignment(x, y)
>>> Assignment(x, 0)
Assignment(x, 0)
>>> A = MatrixSymbol('A', 1, 3)
>>> mat = Matrix([x, y, z]).T
>>> Assignment(A, mat)
Assignment(A, Matrix([[x, y, z]]))
```

(continues on next page)

(continued from previous page)

```
>>> Assignment(A[0, 1], x)
Assignment(A[0, 1], x)
```

class `sympy.codegen.ast.AssignmentBase(lhs, rhs)`
 Abstract base class for Assignment and AugmentedAssignment.

Attributes:

op
 [str] Symbol for assignment operator, e.g. "=", "+=", etc.

class `sympy.codegen.ast.Attribute(possibly parametrized)`
 For use with `sympy.codegen.ast.Node` (page 1134) (which takes instances of Attribute as attrs).

Parameters

name : str
parameters : Tuple

Examples

```
>>> from sympy.codegen.ast import Attribute
>>> volatile = Attribute('volatile')
>>> volatile
volatile
>>> print(repr(volatile))
Attribute(String('volatile'))
>>> a = Attribute('foo', [1, 2, 3])
>>> a
foo(1, 2, 3)
>>> a.parameters == (1, 2, 3)
True
```

class `sympy.codegen.ast.AugmentedAssignment(lhs, rhs)`
 Base class for augmented assignments.

Attributes:

binop
 [str] Symbol for binary operation being applied in the assignment, such as "+", "*", etc.

class `sympy.codegen.ast.BreakToken(*args, **kwargs)`
 Represents 'break' in C/Python ('exit' in Fortran).
 Use the premade instance `break_` or instantiate manually.

Examples

```
>>> from sympy import ccode, fcode
>>> from sympy.codegen.ast import break_
>>> ccode(break_)
'break'
>>> fcode(break_, source_format='free')
'exit'
```

class `sympy.codegen.ast.CodeBlock(*args)`

Represents a block of code.

Explanation

For now only assignments are supported. This restriction will be lifted in the future.

Useful attributes on this object are:

left_hand_sides:

Tuple of left-hand sides of assignments, in order.

right_hand_sides:

Tuple of right-hand sides of assignments, in order.

free_symbols: **Free symbols of the expressions in the right-hand sides**

which do not appear in the left-hand side of an assignment.

Useful methods on this object are:

topological_sort:

Class method. Return a CodeBlock with assignments sorted so that variables are assigned before they are used.

cse:

Return a new CodeBlock with common subexpressions eliminated and pulled out as assignments.

Examples

```
>>> from sympy import symbols, ccode
>>> from sympy.codegen.ast import CodeBlock, Assignment
>>> x, y = symbols('x y')
>>> c = CodeBlock(Assignment(x, 1), Assignment(y, x + 1))
>>> print(ccode(c))
x = 1;
y = x + 1;
```

cse(*symbols=None, optimizations=None, postprocess=None, order='canonical'*)

Return a new code block with common subexpressions eliminated.

Explanation

See the docstring of `sympy.simplify.cse_main.cse()` (page 685) for more information.

Examples

```
>>> from sympy import symbols, sin
>>> from sympy.codegen.ast import CodeBlock, Assignment
>>> x, y, z = symbols('x y z')
```

```
>>> c = CodeBlock(
...     Assignment(x, 1),
...     Assignment(y, sin(x) + 1),
...     Assignment(z, sin(x) - 1),
... )
...
>>> c.cse()
CodeBlock(
    Assignment(x, 1),
    Assignment(x0, sin(x)),
    Assignment(y, x0 + 1),
    Assignment(z, x0 - 1)
)
```

`classmethod topological_sort(assignments)`

Return a CodeBlock with topologically sorted assignments so that variables are assigned before they are used.

Examples

```
>>> from sympy import symbols
>>> from sympy.codegen.ast import CodeBlock, Assignment
>>> x, y, z = symbols('x y z')
```

```
>>> assignments = [
...     Assignment(x, y + z),
...     Assignment(y, z + 1),
...     Assignment(z, 2),
... ]
>>> CodeBlock.topological_sort(assignments)
CodeBlock(
    Assignment(z, 2),
    Assignment(y, z + 1),
    Assignment(x, y + z)
)
```

`class sympy.codegen.ast.Comment(*args, **kwargs)`

Represents a comment.

class sympy.codegen.ast.**ComplexType**(*args, **kwargs)

Represents a complex floating point number.

class sympy.codegen.ast.**ContinueToken**(*args, **kwargs)

Represents 'continue' in C/Python ('cycle' in Fortran)

Use the premade instance `continue_` or instantiate manually.

Examples

```
>>> from sympy import ccode, fcode
>>> from sympy.codegen.ast import continue_
>>> ccode(continue_)
'continue'
>>> fcode(continue_, source_format='free')
'cycle'
```

class sympy.codegen.ast.**Declaration**(*args, **kwargs)

Represents a variable declaration

Parameters

variable : Variable

Examples

```
>>> from sympy.codegen.ast import Declaration, NoneToken, untyped
>>> z = Declaration('z')
>>> z.variable.type == untyped
True
>>> # value is special NoneToken() which must be tested with == operator
>>> z.variable.value is None # won't work
False
>>> z.variable.value == None # not PEP-8 compliant
True
>>> z.variable.value == NoneToken() # OK
True
```

class sympy.codegen.ast.**Element**(*args, **kwargs)

Element in (a possibly N-dimensional) array.

Examples

```
>>> from sympy.codegen.ast import Element
>>> elem = Element('x', 'ijk')
>>> elem.symbol.name == 'x'
True
>>> elem.indices
(i, j, k)
>>> from sympy import ccode
>>> ccode(elem)
```

(continues on next page)

(continued from previous page)

```
'x[i][j][k]'
>>> ccode(Element('x', 'ijk', strides='lmn', offset='o'))
'x[i*l + j*m + k*n + o]'
```

class `sympy.codegen.ast.FloatBaseType(*args, **kwargs)`

Represents a floating point number type.

cast_nocheck

alias of `Float` (page 982)

class `sympy.codegen.ast.FloatType(*args, **kwargs)`

Represents a floating point type with fixed bit width.

Base 2 & one sign bit is assumed.

Parameters

name : str

Name of the type.

nbits : integer

Number of bits used (storage).

nmant : integer

Number of bits used to represent the mantissa.

nexp : integer

Number of bits used to represent the mantissa.

Examples

```
>>> from sympy import S
>>> from sympy.codegen.ast import FloatType
>>> half_precision = FloatType('f16', nbits=16, nmant=10, nexp=5)
>>> half_precision.max
65504
>>> half_precision.tiny == S(2)**-14
True
>>> half_precision.eps == S(2)**-10
True
>>> half_precision.dig == 3
True
>>> half_precision.decimal_dig == 5
True
>>> half_precision.cast_check(1.0)
1.0
>>> half_precision.cast_check(1e5)
Traceback (most recent call last):
...
ValueError: Maximum value for data type smaller than new value.
```

cast_nocheck(*value*)

Casts without checking if out of bounds or subnormal.

property decimal_dig

Number of digits needed to store & load without loss.

Explanation

Number of decimal digits needed to guarantee that two consecutive conversions (float -> text -> float) to be idempotent. This is useful when one do not want to loose precision due to rounding errors when storing a floating point value as text.

property dig

Number of decimal digits that are guaranteed to be preserved in text.

When converting text -> float -> text, you are guaranteed that at least dig number of digits are preserved with respect to rounding or overflow.

property eps

Difference between 1.0 and the next representable value.

property max

Maximum value representable.

property max_exponent

The largest positive number n , such that $2^{(n-1)}$ is a representable finite value.

property min_exponent

The lowest negative number n , such that $2^{(n-1)}$ is a valid normalized number.

property tiny

The minimum positive normalized value.

class sympy.codegen.ast.**For**(*args, **kwargs)

Represents a 'for-loop' in the code.

Expressions are of the form:

"for target in iter:
body..."

Parameters

target : symbol

iter : iterable body : CodeBlock or iterable

! When passed an iterable it is used to instantiate a CodeBlock.

Examples

```
>>> from sympy import symbols, Range
>>> from sympy.codegen.ast import aug_assign, For
>>> x, i, j, k = symbols('x i j k')
>>> for_i = For(i, Range(10), [aug_assign(x, '+', i*j*k)])
>>> for_i
For(i, iterable=Range(0, 10, 1), body=CodeBlock(
    AddAugmentedAssignment(x, i*j*k)
))
>>> for_ji = For(j, Range(7), [for_i])
>>> for_ji
For(j, iterable=Range(0, 7, 1), body=CodeBlock(
    For(i, iterable=Range(0, 10, 1), body=CodeBlock(
        AddAugmentedAssignment(x, i*j*k)
    ))
))
>>> for_kji = For(k, Range(5), [for_ji])
>>> for_kji
For(k, iterable=Range(0, 5, 1), body=CodeBlock(
    For(j, iterable=Range(0, 7, 1), body=CodeBlock(
        For(i, iterable=Range(0, 10, 1), body=CodeBlock(
            AddAugmentedAssignment(x, i*j*k)
        ))
    ))
))
>>>
```

class sympy.codegen.ast.**FunctionCall**(*args, **kwargs)

Represents a call to a function in the code.

Parameters

name : str

function_args : Tuple

Examples

```
>>> from sympy.codegen.ast import FunctionCall
>>> from sympy import pycode
>>> fcall = FunctionCall('foo', 'bar baz'.split())
>>> print(pycode(fcall))
foo(bar, baz)
```

class sympy.codegen.ast.**FunctionDefinition**(*args, **kwargs)

Represents a function definition in the code.

Parameters

return_type : Type

name : str

parameters: iterable of Variable instances

body : CodeBlock or iterable

attrs : iterable of Attribute instances

Examples

```
>>> from sympy import ccode, symbols
>>> from sympy.codegen.ast import real, FunctionPrototype
>>> x, y = symbols('x y', real=True)
>>> fp = FunctionPrototype(real, 'foo', [x, y])
>>> ccode(fp)
'double foo(double x, double y)'
>>> from sympy.codegen.ast import FunctionDefinition, Return
>>> body = [Return(x*y)]
>>> fd = FunctionDefinition.from_FunctionPrototype(fp, body)
>>> print(ccode(fd))
double foo(double x, double y){
    return x*y;
}
```

class sympy.codegen.ast.FunctionPrototype(*args, **kwargs)

Represents a function prototype

Allows the user to generate forward declaration in e.g. C/C++.

Parameters

return_type : Type

name : str

parameters: iterable of Variable instances

attrs : iterable of Attribute instances

Examples

```
>>> from sympy import ccode, symbols
>>> from sympy.codegen.ast import real, FunctionPrototype
>>> x, y = symbols('x y', real=True)
>>> fp = FunctionPrototype(real, 'foo', [x, y])
>>> ccode(fp)
'double foo(double x, double y)'
```

class sympy.codegen.ast.IntBaseType(*args, **kwargs)

Integer base type, contains no size information.

class sympy.codegen.ast.Node(*args, **kwargs)

Subclass of Token, carrying the attribute 'attrs' (Tuple)

Examples

```
>>> from sympy.codegen.ast import Node, value_const, pointer_const
>>> n1 = Node([value_const])
>>> n1.attr_params('value_const') # get the parameters of attribute (by
↳name)
()
>>> from sympy.codegen.fnodes import dimension
>>> n2 = Node([value_const, dimension(5, 3)])
>>> n2.attr_params(value_const) # get the parameters of attribute (by
↳Attribute instance)
()
>>> n2.attr_params('dimension') # get the parameters of attribute (by
↳name)
(5, 3)
>>> n2.attr_params(pointer_const) is None
True
```

attr_params(*looking_for*)

Returns the parameters of the Attribute with name *looking_for* in *self.attrs*

class sympy.codegen.ast.**NoneToken**(*args, **kwargs)

The AST equivalence of Python's NoneType

The corresponding instance of Python's None is none.

Examples

```
>>> from sympy.codegen.ast import none, Variable
>>> from sympy import pycode
>>> print(pycode(Variable('x').as_Declaration(value=none)))
x = None
```

class sympy.codegen.ast.**Pointer**(*args, **kwargs)

Represents a pointer. See Variable.

Examples

Can create instances of Element:

```
>>> from sympy import Symbol
>>> from sympy.codegen.ast import Pointer
>>> i = Symbol('i', integer=True)
>>> p = Pointer('x')
>>> p[i+1]
Element(x, indices=(i + 1,))
```

class sympy.codegen.ast.**Print**(*args, **kwargs)

Represents print command in the code.

Parameters

formatstring : str

***args** : Basic instances (or convertible to such through sympify)

Examples

```
>>> from sympy.codegen.ast import Print
>>> from sympy import pycode
>>> print(pycode(Print('x y'.split(), "coordinate: %12.5g %12.5g")))
print("coordinate: %12.5g %12.5g" % (x, y))
```

class sympy.codegen.ast.**QuotedString**(*args, **kwargs)
Represents a string which should be printed with quotes.

class sympy.codegen.ast.**Return**(*args, **kwargs)
Represents a return command in the code.

Parameters

return : Basic

Examples

```
>>> from sympy.codegen.ast import Return
>>> from sympy.printing.pycode import pycode
>>> from sympy import Symbol
>>> x = Symbol('x')
>>> print(pycode(Return(x)))
return x
```

class sympy.codegen.ast.**Scope**(*args, **kwargs)
Represents a scope in the code.

Parameters

body : CodeBlock or iterable

When passed an iterable it is used to instantiate a CodeBlock.

class sympy.codegen.ast.**SignedIntType**(*args, **kwargs)
Represents a signed integer type.

class sympy.codegen.ast.**Stream**(*args, **kwargs)
Represents a stream.

There are two predefined Stream instances stdout & stderr.

Parameters

name : str

Examples

```
>>> from sympy import pycode, Symbol
>>> from sympy.codegen.ast import Print, stderr, QuotedString
>>> print(pycode(Print(['x'], file=stderr)))
print(x, file=sys.stderr)
>>> x = Symbol('x')
>>> print(pycode(Print([QuotedString('x')], file=stderr))) # print_
↳literally "x"
print("x", file=sys.stderr)
```

class sympy.codegen.ast.**String**(*args, **kwargs)

SymPy object representing a string.

Atomic object which is not an expression (as opposed to Symbol).

Parameters

text : str

Examples

```
>>> from sympy.codegen.ast import String
>>> f = String('foo')
>>> f
foo
>>> str(f)
'foo'
>>> f.text
'foo'
>>> print(repr(f))
String('foo')
```

class sympy.codegen.ast.**Token**(*args, **kwargs)

Base class for the AST types.

Explanation

Defining fields are set in `_fields`. Attributes (defined in `_fields`) are only allowed to contain instances of Basic (unless atomic, see `String`). The arguments to `__new__()` correspond to the attributes in the order defined in `_fields`. The `defaults` class attribute is a dictionary mapping attribute names to their default values.

Subclasses should not need to override the `__new__()` method. They may define a class or static method named `_construct_<attr>` for each attribute to process the value passed to `__new__()`. Attributes listed in the class attribute `not_in_args` are not passed to `Basic` (page 927).

kwargs (*exclude=()*, *apply=None*)

Get instance's attributes as dict of keyword arguments.

Parameters

exclude : collection of str

Collection of keywords to exclude.

apply : callable, optional

Function to apply to all values.

class sympy.codegen.ast.Type(*args, **kwargs)

Represents a type.

Parameters

name : str

Name of the type, e.g. object, int16, float16 (where the latter two would use the Type sub-classes IntType and FloatType respectively). If a Type instance is given, the said instance is returned.

Explanation

The naming is a super-set of NumPy naming. Type has a classmethod `from_expr` which offer type deduction. It also has a method `cast_check` which casts the argument to its type, possibly raising an exception if rounding error is not within tolerances, or if the value is not representable by the underlying data type (e.g. unsigned integers).

Examples

```
>>> from sympy.codegen.ast import Type
>>> t = Type.from_expr(42)
>>> t
integer
>>> print(repr(t))
IntBaseType(String('integer'))
>>> from sympy.codegen.ast import uint8
>>> uint8.cast_check(-1)
Traceback (most recent call last):
...
ValueError: Minimum value for data type bigger than new value.
>>> from sympy.codegen.ast import float32
>>> v6 = 0.123456
>>> float32.cast_check(v6)
0.123456
>>> v10 = 12345.67894
>>> float32.cast_check(v10)
Traceback (most recent call last):
...
ValueError: Casting gives a significantly different value.
>>> boost_mp50 = Type('boost::multiprecision::cpp_dec_float_50')
>>> from sympy import cxxcode
>>> from sympy.codegen.ast import Declaration, Variable
>>> cxxcode(Declaration(Variable('x', type=boost_mp50)))
'boost::multiprecision::cpp_dec_float_50 x'
```

References

[R37]

cast_check(*value*, *rtol*=None, *atol*=0, *precision_targets*=None)

Casts a value to the data type of the instance.

Parameters

value : number

rtol : floating point number

Relative tolerance. (will be deduced if not given).

atol : floating point number

Absolute tolerance (in addition to *rtol*).

type_aliases : dict

Maps substitutions for Type, e.g. {integer: int64, real: float32}

Examples

```
>>> from sympy.codegen.ast import integer, float32, int8
>>> integer.cast_check(3.0) == 3
True
>>> float32.cast_check(1e-40)
Traceback (most recent call last):
...
ValueError: Minimum value for data type bigger than new value.
>>> int8.cast_check(256)
Traceback (most recent call last):
...
ValueError: Maximum value for data type smaller than new value.
>>> v10 = 12345.67894
>>> float32.cast_check(v10)
Traceback (most recent call last):
...
ValueError: Casting gives a significantly different value.
>>> from sympy.codegen.ast import float64
>>> float64.cast_check(v10)
12345.67894
>>> from sympy import Float
>>> v18 = Float('0.123456789012345646')
>>> float64.cast_check(v18)
Traceback (most recent call last):
...
ValueError: Casting gives a significantly different value.
>>> from sympy.codegen.ast import float80
>>> float80.cast_check(v18)
0.123456789012345649
```

classmethod from_expr(*expr*)

Deduces type from an expression or a Symbol.

Parameters

expr : number or SymPy object

The type will be deduced from type or properties.

Raises

ValueError when type deduction fails.

Examples

```
>>> from sympy.codegen.ast import Type, integer, complex_
>>> Type.from_expr(2) == integer
True
>>> from sympy import Symbol
>>> Type.from_expr(Symbol('z', complex=True)) == complex_
True
>>> Type.from_expr(sum)
Traceback (most recent call last):
...
ValueError: Could not deduce type from expr.
```

class sympy.codegen.ast.**UnsignedIntType**(*args, **kwargs)

Represents an unsigned integer type.

class sympy.codegen.ast.**Variable**(*args, **kwargs)

Represents a variable.

Parameters

symbol : Symbol

type : Type (optional)

Type of the variable.

attrs : iterable of Attribute instances

Will be stored as a Tuple.

Examples

```
>>> from sympy import Symbol
>>> from sympy.codegen.ast import Variable, float32, integer
>>> x = Symbol('x')
>>> v = Variable(x, type=float32)
>>> v.attrs
()
>>> v == Variable('x')
False
>>> v == Variable('x', type=float32)
True
>>> v
Variable(x, type=float32)
```

One may also construct a Variable instance with the type deduced from assumptions about the symbol using the deduced classmethod:

```

>>> i = Symbol('i', integer=True)
>>> v = Variable.deduced(i)
>>> v.type == integer
True
>>> v == Variable('i')
False
>>> from sympy.codegen.ast import value_const
>>> value_const in v.attrs
False
>>> w = Variable('w', attrs=[value_const])
>>> w
Variable(w, attrs=(value_const,))
>>> value_const in w.attrs
True
>>> w.as_Declaration(value=42)
Declaration(Variable(w, value=42, attrs=(value_const,)))

```

as_Declaration(**kwargs)

Convenience method for creating a Declaration instance.

Explanation

If the variable of the Declaration need to wrap a modified variable keyword arguments may be passed (overriding e.g. the value of the Variable instance).

Examples

```

>>> from sympy.codegen.ast import Variable, NoneToken
>>> x = Variable('x')
>>> decl1 = x.as_Declaration()
>>> # value is special NoneToken() which must be tested with ==
>>> ↪ operator
>>> decl1.variable.value is None # won't work
False
>>> decl1.variable.value == None # not PEP-8 compliant
True
>>> decl1.variable.value == NoneToken() # OK
True
>>> decl2 = x.as_Declaration(value=42.0)
>>> decl2.variable.value == 42
True

```

classmethod deduced(symbol, value=None, attrs=(), cast_check=True)

Alt. constructor with type deduction from Type.from_expr.

Deduces type primarily from symbol, secondarily from value.

Parameters

symbol : Symbol

value : expr

(optional) value of the variable.

attrs : iterable of Attribute instances

cast_check : bool

Whether to apply `Type.cast_check` on value.

Examples

```
>>> from sympy import Symbol
>>> from sympy.codegen.ast import Variable, complex_
>>> n = Symbol('n', integer=True)
>>> str(Variable.deduced(n).type)
'integer'
>>> x = Symbol('x', real=True)
>>> v = Variable.deduced(x)
>>> v.type
real
>>> z = Symbol('z', complex=True)
>>> Variable.deduced(z).type == complex_
True
```

class `sympy.codegen.ast.While(*args, **kwargs)`

Represents a ‘for-loop’ in the code.

Expressions are of the form:

“while condition:
body...”

Parameters

condition : expression convertible to Boolean

body : CodeBlock or iterable

When passed an iterable it is used to instantiate a CodeBlock.

Examples

```
>>> from sympy import symbols, Gt, Abs
>>> from sympy.codegen import aug_assign, Assignment, While
>>> x, dx = symbols('x dx')
>>> expr = 1 - x**2
>>> whl = While(Gt(Abs(dx), 1e-9), [
...     Assignment(dx, -expr/expr.diff(x)),
...     aug_assign(x, '+', dx)
... ])
```

`sympy.codegen.ast.aug_assign(lhs, op, rhs)`

Create ‘lhs op= rhs’.

Parameters

lhs : Expr

SymPy object representing the lhs of the expression. These should be singular objects, such as one would use in writing code. Notable types include `Symbol`, `MatrixSymbol`, `MatrixElement`, and `Indexed`. Types that subclass these types are also supported.

op : str

Operator (+, -, /, *, %).

rhs : Expr

SymPy object representing the rhs of the expression. This can be any type, provided its shape corresponds to that of the lhs. For example, a `Matrix` type can be assigned to `MatrixSymbol`, but not to `Symbol`, as the dimensions will not align.

Explanation

Represents augmented variable assignment for code generation. This is a convenience function. You can also use the `AugmentedAssignment` classes directly, like `AddAugmentedAssignment(x, y)`.

Examples

```
>>> from sympy import symbols
>>> from sympy.codegen.ast import aug_assign
>>> x, y = symbols('x, y')
>>> aug_assign(x, '+', y)
AddAugmentedAssignment(x, y)
```

Special C math functions (`sympy.codegen.cfunctions`)

This module contains SymPy functions matching corresponding to special math functions in the C standard library (since C99, also available in C++11).

The functions defined in this module allows the user to express functions such as `expm1` as a SymPy function for symbolic manipulation.

class `sympy.codegen.cfunctions.Cbrt(*args)`

Represents the cube root function.

Explanation

The reason why one would use `Cbrt(x)` over `cbert(x)` is that the latter is internally represented as `Pow(x, Rational(1, 3))` which may not be what one wants when doing code-generation.

Examples

```
>>> from sympy.abc import x
>>> from sympy.codegen.cfunctions import Cbrt
>>> Cbrt(x)
Cbrt(x)
>>> Cbrt(x).diff(x)
1/(3*x**(2/3))
```

See also:

[Sqrt](#) (page 1144)

fdiff(*argindex*=1)

Returns the first derivative of this function.

class sympy.codegen.cfunctions.**Sqrt**(*args)

Represents the square root function.

Explanation

The reason why one would use `Sqrt(x)` over `sqrt(x)` is that the latter is internally represented as `Pow(x, S.Half)` which may not be what one wants when doing code-generation.

Examples

```
>>> from sympy.abc import x
>>> from sympy.codegen.cfunctions import Sqrt
>>> Sqrt(x)
Sqrt(x)
>>> Sqrt(x).diff(x)
1/(2*sqrt(x))
```

See also:

[Cbrt](#) (page 1143)

fdiff(*argindex*=1)

Returns the first derivative of this function.

class sympy.codegen.cfunctions.**exp2**(*arg*)

Represents the exponential function with base two.

Explanation

The benefit of using `exp2(x)` over `2**x` is that the latter is not as efficient under finite precision arithmetic.

Examples

```
>>> from sympy.abc import x
>>> from sympy.codegen.cfunctions import exp2
>>> exp2(2).evalf() == 4
True
>>> exp2(x).diff(x)
log(2)*exp2(x)
```

See also:

[`log2`](#) (page 1147)

`fdiff(argindex=1)`

Returns the first derivative of this function.

`class sympy.codegen.cfunctions.expml(arg)`

Represents the exponential function minus one.

Explanation

The benefit of using `expml(x)` over `exp(x) - 1` is that the latter is prone to cancellation under finite precision arithmetic when x is close to zero.

Examples

```
>>> from sympy.abc import x
>>> from sympy.codegen.cfunctions import expml
>>> '%.0e' % expml(1e-99).evalf()
'1e-99'
>>> from math import exp
>>> exp(1e-99) - 1
0.0
>>> expml(x).diff(x)
exp(x)
```

See also:

[`log1p`](#) (page 1147)

`fdiff(argindex=1)`

Returns the first derivative of this function.

`class sympy.codegen.cfunctions.fma(*args)`

Represents “fused multiply add”.

Explanation

The benefit of using `fma(x, y, z)` over `x*y + z` is that, under finite precision arithmetic, the former is supported by special instructions on some CPUs.

Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.codegen.cfunctions import fma
>>> fma(x, y, z).diff(x)
y
```

fdiff(*argindex*=1)

Returns the first derivative of this function.

class sympy.codegen.cfunctions.**hypot**(*args)

Represents the hypotenuse function.

Explanation

The hypotenuse function is provided by e.g. the math library in the C99 standard, hence one may want to represent the function symbolically when doing code-generation.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy.codegen.cfunctions import hypot
>>> hypot(3, 4).evalf() == 5
True
>>> hypot(x, y)
hypot(x, y)
>>> hypot(x, y).diff(x)
x/hypot(x, y)
```

fdiff(*argindex*=1)

Returns the first derivative of this function.

class sympy.codegen.cfunctions.**log10**(arg)

Represents the logarithm function with base ten.

Examples

```
>>> from sympy.abc import x
>>> from sympy.codegen.cfunctions import log10
>>> log10(100).evalf() == 2
True
>>> log10(x).diff(x)
1/(x*log(10))
```

See also:

[log2](#) (page 1147)

fdiff(*argindex*=1)

Returns the first derivative of this function.

class sympy.codegen.cfunctions.**log1p**(*arg*)

Represents the natural logarithm of a number plus one.

Explanation

The benefit of using `log1p(x)` over `log(x + 1)` is that the latter is prone to cancellation under finite precision arithmetic when `x` is close to zero.

Examples

```
>>> from sympy.abc import x
>>> from sympy.codegen.cfunctions import log1p
>>> from sympy import expand_log
>>> '%.0e' % expand_log(log1p(1e-99)).evalf()
'1e-99'
>>> from math import log
>>> log(1 + 1e-99)
0.0
>>> log1p(x).diff(x)
1/(x + 1)
```

See also:

[expm1](#) (page 1145)

fdiff(*argindex*=1)

Returns the first derivative of this function.

class sympy.codegen.cfunctions.**log2**(*arg*)

Represents the logarithm function with base two.

Explanation

The benefit of using $\log_2(x)$ over $\log(x)/\log(2)$ is that the latter is not as efficient under finite precision arithmetic.

Examples

```
>>> from sympy.abc import x
>>> from sympy.codegen.cfunctions import log2
>>> log2(4).evalf() == 2
True
>>> log2(x).diff(x)
1/(x*log(2))
```

See also:

[*exp2*](#) (page 1144), [*log10*](#) (page 1146)

fdiff(*argindex*=1)

Returns the first derivative of this function.

C specific AST nodes (`sympy.codegen.cnodes`)

AST nodes specific to the C family of languages

class `sympy.codegen.cnodes.CommaOperator(*args)`

Represents the comma operator in C

class `sympy.codegen.cnodes.Label(*args, **kwargs)`

Label for use with e.g. goto statement.

Examples

```
>>> from sympy import ccode, Symbol
>>> from sympy.codegen.cnodes import Label, PreIncrement
>>> print(ccode(Label('foo')))
foo:
>>> print(ccode(Label('bar', [PreIncrement(Symbol('a'))])))
bar:
++(a);
```

class `sympy.codegen.cnodes.PostDecrement(*args)`

Represents the post-decrement operator

class `sympy.codegen.cnodes.PostIncrement(*args)`

Represents the post-increment operator

class `sympy.codegen.cnodes.PreDecrement(*args)`

Represents the pre-decrement operator

Examples

```
>>> from sympy.abc import x
>>> from sympy.codegen.cnodes import PreDecrement
>>> from sympy import ccode
>>> ccode(PreDecrement(x))
'--(x)'
```

class sympy.codegen.cnodes.**PreIncrement**(*args)

Represents the pre-increment operator

sympy.codegen.cnodes.**alignof**(arg)

Generate of FunctionCall instance for calling 'alignof'

class sympy.codegen.cnodes.**goto**(*args, **kwargs)

Represents goto in C

sympy.codegen.cnodes.**sizeof**(arg)

Generate of FunctionCall instance for calling 'sizeof'

Examples

```
>>> from sympy.codegen.ast import real
>>> from sympy.codegen.cnodes import sizeof
>>> from sympy import ccode
>>> ccode(sizeof(real))
'sizeof(double)'
```

class sympy.codegen.cnodes.**struct**(*args, **kwargs)

Represents a struct in C

class sympy.codegen.cnodes.**union**(*args, **kwargs)

Represents a union in C

C++ specific AST nodes (sympy.codegen.cxxnodes)

AST nodes specific to C++.

class sympy.codegen.cxxnodes.**using**(*args, **kwargs)

Represents a 'using' statement in C++

Fortran specific AST nodes (sympy.codegen.fnodes)

AST nodes specific to Fortran.

The functions defined in this module allows the user to express functions such as dsig as a SymPy function for symbolic manipulation.

class sympy.codegen.fnodes.**ArrayConstructor**(*args, **kwargs)

Represents an array constructor.

Examples

```
>>> from sympy import fcode
>>> from sympy.codegen.fnodes import ArrayConstructor
>>> ac = ArrayConstructor([1, 2, 3])
>>> fcode(ac, standard=95, source_format='free')
'(/1, 2, 3/)'
>>> fcode(ac, standard=2003, source_format='free')
'[1, 2, 3]'
```

class sympy.codegen.fnodes.**Do**(*args, **kwargs)
Represents a Do loop in in Fortran.

Examples

```
>>> from sympy import fcode, symbols
>>> from sympy.codegen.ast import aug_assign, Print
>>> from sympy.codegen.fnodes import Do
>>> i, n = symbols('i n', integer=True)
>>> r = symbols('r', real=True)
>>> body = [aug_assign(r, '+', 1/i), Print([i, r])]
>>> do1 = Do(body, i, 1, n)
>>> print(fcode(do1, source_format='free'))
do i = 1, n
  r = r + 1d0/i
  print *, i, r
end do
>>> do2 = Do(body, i, 1, n, 2)
>>> print(fcode(do2, source_format='free'))
do i = 1, n, 2
  r = r + 1d0/i
  print *, i, r
end do
```

class sympy.codegen.fnodes.**Extent**(*args)
Represents a dimension extent.

Examples

```
>>> from sympy.codegen.fnodes import Extent
>>> e = Extent(-3, 3) # -3, -2, -1, 0, 1, 2, 3
>>> from sympy import fcode
>>> fcode(e, source_format='free')
'-3:3'
>>> from sympy.codegen.ast import Variable, real
>>> from sympy.codegen.fnodes import dimension, intent_out
>>> dim = dimension(e, e)
>>> arr = Variable('x', real, attrs=[dim, intent_out])
>>> fcode(arr.as_Declaration(), source_format='free', standard=2003)
'real*8, dimension(-3:3, -3:3), intent(out) :: x'
```

class `sympy.codegen.fnodes.FortranReturn(*args, **kwargs)`

AST node explicitly mapped to a fortran “return”.

Explanation

Because a return statement in fortran is different from C, and in order to aid reuse of our codegen ASTs the ordinary `.codegen.ast.Return` is interpreted as assignment to the result variable of the function. If one for some reason needs to generate a fortran RETURN statement, this node should be used.

Examples

```
>>> from sympy.codegen.fnodes import FortranReturn
>>> from sympy import fcode
>>> fcode(FortranReturn('x'))
'    return x'
```

class `sympy.codegen.fnodes.GoTo(*args, **kwargs)`

Represents a goto statement in Fortran

Examples

```
>>> from sympy.codegen.fnodes import GoTo
>>> go = GoTo([10, 20, 30], 'i')
>>> from sympy import fcode
>>> fcode(go, source_format='free')
'go to (10, 20, 30), i'
```

class `sympy.codegen.fnodes.ImpliedDoLoop(*args, **kwargs)`

Represents an implied do loop in Fortran.

Examples

```
>>> from sympy import Symbol, fcode
>>> from sympy.codegen.fnodes import ImpliedDoLoop, ArrayConstructor
>>> i = Symbol('i', integer=True)
>>> idl = ImpliedDoLoop(i**3, i, -3, 3, 2) # -27, -1, 1, 27
>>> ac = ArrayConstructor([-28, idl, 28]) # -28, -27, -1, 1, 27, 28
>>> fcode(ac, standard=2003, source_format='free')
'[-28, (i**3, i = -3, 3, 2), 28]'
```

class `sympy.codegen.fnodes.Module(*args, **kwargs)`

Represents a module in Fortran.

Examples

```
>>> from sympy.codegen.fnodes import Module
>>> from sympy import fcode
>>> print(fcode(Module('signallib', ['implicit none'], []), source_
→format='free'))
module signallib
implicit none

contains

end module
```

class sympy.codegen.fnodes.**Program**(*args, **kwargs)
Represents a ‘program’ block in Fortran.

Examples

```
>>> from sympy.codegen.ast import Print
>>> from sympy.codegen.fnodes import Program
>>> prog = Program('myprogram', [Print([42])])
>>> from sympy import fcode
>>> print(fcode(prog, source_format='free'))
program myprogram
  print *, 42
end program
```

class sympy.codegen.fnodes.**Subroutine**(*args, **kwargs)
Represents a subroutine in Fortran.

Examples

```
>>> from sympy import fcode, symbols
>>> from sympy.codegen.ast import Print
>>> from sympy.codegen.fnodes import Subroutine
>>> x, y = symbols('x y', real=True)
>>> sub = Subroutine('mysub', [x, y], [Print([x**2 + y**2, x*y])])
>>> print(fcode(sub, source_format='free', standard=2003))
subroutine mysub(x, y)
real*8 :: x
real*8 :: y
print *, x**2 + y**2, x*y
end subroutine
```

class sympy.codegen.fnodes.**SubroutineCall**(*args, **kwargs)
Represents a call to a subroutine in Fortran.

Examples

```
>>> from sympy.codegen.fnodes import SubroutineCall
>>> from sympy import fcode
>>> fcode(SubroutineCall('mysub', 'x y'.split()))
'      call mysub(x, y)'
```

`sympy.codegen.fnodes.allocated(array)`

Creates an AST node for a function call to Fortran’s “allocated(...)”

Examples

```
>>> from sympy import fcode
>>> from sympy.codegen.fnodes import allocated
>>> alloc = allocated('x')
>>> fcode(alloc, source_format='free')
'allocated(x)'
```

`sympy.codegen.fnodes.array(symbol, dim, intent=None, *, attrs=(), value=None, type=None)`

Convenience function for creating a Variable instance for a Fortran array.

Parameters

symbol : symbol

dim : Attribute or iterable

If dim is an Attribute it need to have the name ‘dimension’. If it is not an Attribute, then it is passed to [dimension\(\)](#) (page 1154) as *dim

intent : str

One of: ‘in’, ‘out’, ‘inout’ or None

****kwargs**:

Keyword arguments for Variable (‘type’ & ‘value’)

Examples

```
>>> from sympy import fcode
>>> from sympy.codegen.ast import integer, real
>>> from sympy.codegen.fnodes import array
>>> arr = array('a', '*', 'in', type=integer)
>>> print(fcode(arr.as_Declaration(), source_format='free',
↳ standard=2003))
integer*4, dimension(*), intent(in) :: a
>>> x = array('x', [3, ':', ':'], intent='out', type=real)
>>> print(fcode(x.as_Declaration(value=1), source_format='free',
↳ standard=2003))
real*8, dimension(3, :, :), intent(out) :: x = 1
```

`sympy.codegen.fnodes.bind_C(name=None)`

Creates an Attribute `bind_C` with a name.

Parameters

name : str

Examples

```
>>> from sympy import fcode, Symbol
>>> from sympy.codegen.ast import FunctionDefinition, real, Return
>>> from sympy.codegen.fnodes import array, sum_, bind_C
>>> a = Symbol('a', real=True)
>>> s = Symbol('s', integer=True)
>>> arr = array(a, dim=[s], intent='in')
>>> body = [Return((sum_(a**2)/s)**.5)]
>>> fd = FunctionDefinition(real, 'rms', [arr, s], body, attrs=[bind_C(
    ↪ 'rms')])
>>> print(fcode(fd, source_format='free', standard=2003))
real*8 function rms(a, s) bind(C, name="rms")
real*8, dimension(s), intent(in) :: a
integer*4 :: s
rms = sqrt(sum(a**2)/s)
end function
```

class `sympy.codegen.fnodes.cmplx(*args)`

Fortran complex conversion function.

`sympy.codegen.fnodes.dimension(*args)`

Creates a 'dimension' Attribute with (up to 7) extents.

Examples

```
>>> from sympy import fcode
>>> from sympy.codegen.fnodes import dimension, intent_in
>>> dim = dimension('2', ':') # 2 rows, runtime determined number of
    ↪ columns
>>> from sympy.codegen.ast import Variable, integer
>>> arr = Variable('a', integer, attrs=[dim, intent_in])
>>> fcode(arr.as_Declaration(), source_format='free', standard=2003)
'integer*4, dimension(2, :), intent(in) :: a'
```

class `sympy.codegen.fnodes.dsign(*args)`

Fortran sign intrinsic for double precision arguments.

class `sympy.codegen.fnodes.isign(*args)`

Fortran sign intrinsic for integer arguments.

class `sympy.codegen.fnodes.kind(*args)`

Fortran kind function.

`sympy.codegen.fnodes.lbound(array, dim=None, kind=None)`

Creates an AST node for a function call to Fortran's "lbound(...)"

Parameters**array** : Symbol or String**dim** : expr**kind** : expr**Examples**

```
>>> from sympy import fcode
>>> from sympy.codegen.fnodes import lbound
>>> lb = lbound('arr', dim=2)
>>> fcode(lb, source_format='free')
'lbound(arr, 2)'
```

class sympy.codegen.fnodes.**literal_dp**(num, dps=None, precision=None)
Fortran double precision real literal

class sympy.codegen.fnodes.**literal_sp**(num, dps=None, precision=None)
Fortran single precision real literal

class sympy.codegen.fnodes.**merge**(*args)
Fortran merge function

sympy.codegen.fnodes.**reshape**(source, shape, pad=None, order=None)
Creates an AST node for a function call to Fortran's "reshape(...)"

Parameters**source** : Symbol or String**shape** : ArrayExpr

sympy.codegen.fnodes.**shape**(source, kind=None)
Creates an AST node for a function call to Fortran's "shape(...)"

Parameters**source** : Symbol or String**kind** : expr**Examples**

```
>>> from sympy import fcode
>>> from sympy.codegen.fnodes import shape
>>> shp = shape('x')
>>> fcode(shp, source_format='free')
'shape(x)'
```

sympy.codegen.fnodes.**size**(array, dim=None, kind=None)
Creates an AST node for a function call to Fortran's "size(...)"

Examples

```
>>> from sympy import fcode, Symbol
>>> from sympy.codegen.ast import FunctionDefinition, real, Return
>>> from sympy.codegen.fnodes import array, sum_, size
>>> a = Symbol('a', real=True)
>>> body = [Return((sum_(a**2)/size(a))**.5)]
>>> arr = array(a, dim=[':'], intent='in')
>>> fd = FunctionDefinition(real, 'rms', [arr], body)
>>> print(fcode(fd, source_format='free', standard=2003))
real*8 function rms(a)
real*8, dimension(:), intent(in) :: a
rms = sqrt(sum(a**2)*ld0/size(a))
end function
```

class sympy.codegen.fnodes.use(*args, **kwargs)

Represents a use statement in Fortran.

Examples

```
>>> from sympy.codegen.fnodes import use
>>> from sympy import fcode
>>> fcode(use('signallib'), source_format='free')
'use signallib'
>>> fcode(use('signallib', [('metric', 'snr')]), source_format='free')
'use signallib, metric => snr'
>>> fcode(use('signallib', only=['snr', 'convolution2d']), source_format=
→ 'free')
'use signallib, only: snr, convolution2d'
```

class sympy.codegen.fnodes.use_rename(*args, **kwargs)

Represents a renaming in a use statement in Fortran.

Examples

```
>>> from sympy.codegen.fnodes import use_rename, use
>>> from sympy import fcode
>>> ren = use_rename("thingy", "convolution2d")
>>> print(fcode(ren, source_format='free'))
thingy => convolution2d
>>> full = use('signallib', only=['snr', ren])
>>> print(fcode(full, source_format='free'))
use signallib, only: snr, thingy => convolution2d
```

Algorithms (sympy.codegen.algorithms)

`sympy.codegen.algorithms.newtons_method`(*expr*, *wrt*, *atol*=1e-12, *delta*=None, *debug*=False, *itermax*=None, *counter*=None)

Generates an AST for Newton-Raphson method (a root-finding algorithm).

Parameters

expr : expression

wrt : Symbol

With respect to, i.e. what is the variable.

atol : number or expr

Absolute tolerance (stopping criterion)

delta : Symbol

Will be a Dummy if None.

debug : bool

Whether to print convergence information during iterations

itermax : number or expr

Maximum number of iterations.

counter : Symbol

Will be a Dummy if None.

Explanation

Returns an abstract syntax tree (AST) based on `sympy.codegen.ast` for Newton's method of root-finding.

Examples

```
>>> from sympy import symbols, cos
>>> from sympy.codegen.ast import Assignment
>>> from sympy.codegen.algorithms import newtons_method
>>> x, dx, atol = symbols('x dx atol')
>>> expr = cos(x) - x**3
>>> algo = newtons_method(expr, x, atol, dx)
>>> algo.has(Assignment(dx, -expr/expr.diff(x)))
True
```

References

[R38]

`sympy.codegen.algorithms.newtons_method_function(expr, wrt, params=None, func_name='newton', attrs=(), *, delta=None, **kwargs)`

Generates an AST for a function implementing the Newton-Raphson method.

Parameters

expr : expression

wrt : Symbol

With respect to, i.e. what is the variable

params : iterable of symbols

Symbols appearing in `expr` that are taken as constants during the iterations (these will be accepted as parameters to the generated function).

func_name : str

Name of the generated function.

attrs : Tuple

Attribute instances passed as `attrs` to `FunctionDefinition`.

****kwargs** :

Keyword arguments passed to `sympy.codegen.algorithms.newtons_method()` (page 1157).

Examples

```
>>> from sympy import symbols, cos
>>> from sympy.codegen.algorithms import newtons_method_function
>>> from sympy.codegen.pyutils import render_as_module
>>> x = symbols('x')
>>> expr = cos(x) - x**3
>>> func = newtons_method_function(expr, x)
>>> py_mod = render_as_module(func) # source code as string
>>> namespace = {}
>>> exec(py_mod, namespace, namespace)
>>> res = eval('newton(0.5)', namespace)
>>> abs(res - 0.865474033102) < 1e-12
True
```

See also:

`sympy.codegen.algorithms.newtons_method` (page 1157)

Python utilities (`sympy.codegen.pyutils`)

`sympy.codegen.pyutils.render_as_module(content, standard='python3')`

Renders Python code as a module (with the required imports).

Parameters

standard :

See the parameter standard in [sympy.printing.pycode.pycode\(\)](#) (page 2176)

C utilities (`sympy.codegen.cutils`)

`sympy.codegen.cutils.render_as_source_file(content, Printer=<class 'sympy.printing.c.C99CodePrinter'>, settings=None)`

Renders a C source file (with required `#include` statements)

Fortran utilities (`sympy.codegen.futils`)

`sympy.codegen.futils.render_as_module(definitions, name, declarations=(), printer_settings=None)`

Creates a Module instance and renders it as a string.

This generates Fortran source code for a module with the correct use statements.

Parameters

definitions : iterable

Passed to [sympy.codegen.fnodes.Module](#) (page 1151).

name : str

Passed to [sympy.codegen.fnodes.Module](#) (page 1151).

declarations : iterable

Passed to [sympy.codegen.fnodes.Module](#) (page 1151). It will be extended with use statements, 'implicit none' and public list generated from definitions.

printer_settings : dict

Passed to `FCodePrinter` (default: `{'standard': 2003, 'source_format': 'free'}`).

5.8.3 Logic

Contents

Logic

Introduction

The logic module for SymPy allows to form and manipulate logic expressions using symbolic and Boolean values.

Forming logical expressions

You can build Boolean expressions with the standard python operators & ([And](#) (page 1166)), | ([Or](#) (page 1167)), ~ ([Not](#) (page 1167)):

```
>>> from sympy import *
>>> x, y = symbols('x,y')
>>> y | (x & y)
y | (x & y)
>>> x | y
x | y
>>> ~x
~x
```

You can also form implications with >> and <<:

```
>>> x >> y
Implies(x, y)
>>> x << y
Implies(y, x)
```

Like most types in SymPy, Boolean expressions inherit from [Basic](#) (page 927):

```
>>> (y & x).subs({x: True, y: True})
True
>>> (x | y).atoms()
{x, y}
```

The logic module also includes the following functions to derive boolean expressions from their truth tables:

`sympy.logic.boolalg.SOPform(variables, minterms, dontcares=None)`

The SOPform function uses `simplified_pairs` and a redundant group- eliminating algorithm to convert the list of all input combos that generate '1' (the minterms) into the smallest sum-of-products form.

The variables must be given as the first argument.

Return a logical [Or](#) (page 1167) function (i.e., the “sum of products” or “SOP” form) that gives the desired outcome. If there are inputs that can be ignored, pass them as a list, too.

The result will be one of the (perhaps many) functions that satisfy the conditions.

Examples

```
>>> from sympy.logic import SOPform
>>> from sympy import symbols
>>> w, x, y, z = symbols('w x y z')
>>> minterms = [[0, 0, 0, 1], [0, 0, 1, 1],
...             [0, 1, 1, 1], [1, 0, 1, 1], [1, 1, 1, 1]]
>>> dontcares = [[0, 0, 0, 0], [0, 0, 1, 0], [0, 1, 0, 1]]
>>> SOPform([w, x, y, z], minterms, dontcares)
(y & z) | (~w & ~x)
```

The terms can also be represented as integers:

```
>>> minterms = [1, 3, 7, 11, 15]
>>> dontcares = [0, 2, 5]
>>> SOPform([w, x, y, z], minterms, dontcares)
(y & z) | (~w & ~x)
```

They can also be specified using dicts, which does not have to be fully specified:

```
>>> minterms = [{w: 0, x: 1}, {y: 1, z: 1, x: 0}]
>>> SOPform([w, x, y, z], minterms)
(x & ~w) | (y & z & ~x)
```

Or a combination:

```
>>> minterms = [4, 7, 11, [1, 1, 1, 1]]
>>> dontcares = [{w: 0, x: 0, y: 0}, 5]
>>> SOPform([w, x, y, z], minterms, dontcares)
(w & y & z) | (~w & ~y) | (x & z & ~w)
```

References

[R557]

`sympy.logic.boolalg.POSform(variables, minterms, dontcares=None)`

The POSform function uses `simplified_pairs` and a redundant-group eliminating algorithm to convert the list of all input combinations that generate '1' (the minterms) into the smallest product-of-sums form.

The variables must be given as the first argument.

Return a logical [And](#) (page 1166) function (i.e., the “product of sums” or “POS” form) that gives the desired outcome. If there are inputs that can be ignored, pass them as a list, too.

The result will be one of the (perhaps many) functions that satisfy the conditions.

Examples

```
>>> from sympy.logic import POSform
>>> from sympy import symbols
>>> w, x, y, z = symbols('w x y z')
>>> minterms = [[0, 0, 0, 1], [0, 0, 1, 1], [0, 1, 1, 1],
...             [1, 0, 1, 1], [1, 1, 1, 1]]
>>> dontcares = [[0, 0, 0, 0], [0, 0, 1, 0], [0, 1, 0, 1]]
>>> POSform([w, x, y, z], minterms, dontcares)
z & (y | ~w)
```

The terms can also be represented as integers:

```
>>> minterms = [1, 3, 7, 11, 15]
>>> dontcares = [0, 2, 5]
>>> POSform([w, x, y, z], minterms, dontcares)
z & (y | ~w)
```

They can also be specified using dicts, which does not have to be fully specified:

```
>>> minterms = [{w: 0, x: 1}, {y: 1, z: 1, x: 0}]
>>> POSform([w, x, y, z], minterms)
(x | y) & (x | z) & (~w | ~x)
```

Or a combination:

```
>>> minterms = [4, 7, 11, [1, 1, 1, 1]]
>>> dontcares = [{w : 0, x : 0, y: 0}, 5]
>>> POSform([w, x, y, z], minterms, dontcares)
(w | x) & (y | ~w) & (z | ~y)
```

References

[R558]

`sympy.logic.boolalg.ANFform(variables, truthvalues)`

The ANFform function converts the list of truth values to Algebraic Normal Form (ANF).

The variables must be given as the first argument.

Return True, False, logical [And](#) (page 1166) function (i.e., the “Zhegalkin monomial”) or logical [Xor](#) (page 1168) function (i.e., the “Zhegalkin polynomial”). When True and False are represented by 1 and 0, respectively, then [And](#) (page 1166) is multiplication and [Xor](#) (page 1168) is addition.

Formally a “Zhegalkin monomial” is the product (logical And) of a finite set of distinct variables, including the empty set whose product is denoted 1 (True). A “Zhegalkin polynomial” is the sum (logical Xor) of a set of Zhegalkin monomials, with the empty set denoted by 0 (False).

Parameters

variables : list of variables

truthvalues : list of 1’s and 0’s (result column of truth table)

Examples

```
>>> from sympy.logic.boolalg import ANFform
>>> from sympy.abc import x, y
>>> ANFform([x], [1, 0])
x ^ True
>>> ANFform([x, y], [0, 1, 1, 1])
x ^ y ^ (x & y)
```

References

[R559]

Boolean functions

class sympy.logic.boolalg.**Boolean**(*args)

A Boolean object is an object for which logic operations make sense.

as_set()

Rewrites Boolean expression in terms of real sets.

Examples

```
>>> from sympy import Symbol, Eq, Or, And
>>> x = Symbol('x', real=True)
>>> Eq(x, 0).as_set()
{0}
>>> (x > 0).as_set()
Interval.open(0, oo)
>>> And(-2 < x, x < 2).as_set()
Interval.open(-2, 2)
>>> Or(x < -2, 2 < x).as_set()
Union(Interval.open(-oo, -2), Interval.open(2, oo))
```

equals(other)

Returns True if the given formulas have the same truth table. For two formulas to be equal they must have the same literals.

Examples

```
>>> from sympy.abc import A, B, C
>>> from sympy import And, Or, Not
>>> (A >> B).equals(~B >> ~A)
True
>>> Not(And(A, B, C)).equals(And(Not(A), Not(B), Not(C)))
False
>>> Not(And(A, Not(A))).equals(Or(B, Not(B)))
False
```

class sympy.logic.boolalg.BooleanTrue

SymPy version of True, a singleton that can be accessed via `S.true`.

This is the SymPy version of True, for use in the logic module. The primary advantage of using `true` instead of `True` is that shorthand Boolean operations like `~` and `>>` will work as expected on this class, whereas with `True` they act bitwise on 1. Functions in the logic module will return this class when they evaluate to `true`.

Notes

There is liable to be some confusion as to when `True` should be used and when `S.true` should be used in various contexts throughout SymPy. An important thing to remember is that `sympify(True)` returns `S.true`. This means that for the most part, you can just use `True` and it will automatically be converted to `S.true` when necessary, similar to how you can generally use 1 instead of `S.One`.

The rule of thumb is:

“If the boolean in question can be replaced by an arbitrary symbolic Boolean, like `Or(x, y)` or `x > 1`, use `S.true`. Otherwise, use `True`”

In other words, use `S.true` only on those contexts where the boolean is being used as a symbolic representation of truth. For example, if the object ends up in the `.args` of any expression, then it must necessarily be `S.true` instead of `True`, as elements of `.args` must be `Basic`. On the other hand, `==` is not a symbolic operation in SymPy, since it always returns `True` or `False`, and does so in terms of structural equality rather than mathematical, so it should return `True`. The assumptions system should use `True` and `False`. Aside from not satisfying the above rule of thumb, the assumptions system uses a three-valued logic (`True`, `False`, `None`), whereas `S.true` and `S.false` represent a two-valued logic. When in doubt, use `True`.

“`S.true == True` is `True`.”

While “`S.true is True`” is `False`, “`S.true == True`” is `True`, so if there is any doubt over whether a function or expression will return `S.true` or `True`, just use `==` instead of `is` to do the comparison, and it will work in either case. Finally, for boolean flags, it’s better to just use `if x` instead of `if x is True`. To quote PEP 8:

Do not compare boolean values to `True` or `False` using `==`.

- Yes: `if greeting:`
- No: `if greeting == True:`
- Worse: `if greeting is True:`

Examples

```
>>> from sympy import sympify, true, false, Or
>>> sympify(True)
True
>>> _ is True, _ is true
(False, True)
```

```
>>> Or(true, false)
True
>>> _ is true
True
```

Python operators give a boolean result for true but a bitwise result for True

```
>>> ~true, ~True
(False, -2)
>>> true >> true, True >> True
(True, 0)
```

Python operators give a boolean result for true but a bitwise result for True

```
>>> ~true, ~True
(False, -2)
>>> true >> true, True >> True
(True, 0)
```

See also:

[*sympy.logic.boolalg.BooleanFalse*](#) (page 1165)

as_set()

Rewrite logic operators and relationals in terms of real sets.

Examples

```
>>> from sympy import true
>>> true.as_set()
UniversalSet
```

class `sympy.logic.boolalg.BooleanFalse`

SymPy version of False, a singleton that can be accessed via `S.false`.

This is the SymPy version of False, for use in the logic module. The primary advantage of using `false` instead of `False` is that shorthand Boolean operations like `~` and `>>` will work as expected on this class, whereas with `False` they act bitwise on 0. Functions in the logic module will return this class when they evaluate to false.

Notes

See the notes section in [*sympy.logic.boolalg.BooleanTrue*](#) (page 1163)

Examples

```
>>> from sympy import sympify, true, false, Or
>>> sympify(False)
False
>>> _ is False, _ is false
(False, True)
```

```
>>> Or(true, false)
True
>>> _ is true
True
```

Python operators give a boolean result for false but a bitwise result for False

```
>>> ~false, ~False
(True, -1)
>>> false >> false, False >> False
(True, 0)
```

See also:

[*sympy.logic.boolalg.BooleanTrue*](#) (page 1163)

`as_set()`

Rewrite logic operators and relationals in terms of real sets.

Examples

```
>>> from sympy import false
>>> false.as_set()
EmptySet
```

class `sympy.logic.boolalg.And(*args)`

Logical AND function.

It evaluates its arguments in order, returning false immediately when an argument is false and true if they are all true.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy import And
>>> x & y
x & y
```

Notes

The `&` operator is provided as a convenience, but note that its use here is different from its normal use in Python, which is bitwise and. Hence, `And(a, b)` and `a & b` will produce different results if `a` and `b` are integers.

```
>>> And(x, y).subs(x, 1)
y
```

class `sympy.logic.boolalg.Or(*args)`

Logical OR function

It evaluates its arguments in order, returning true immediately when an argument is true, and false if they are all false.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy import Or
>>> x | y
x | y
```

Notes

The `|` operator is provided as a convenience, but note that its use here is different from its normal use in Python, which is bitwise or. Hence, `Or(a, b)` and `a | b` will return different things if `a` and `b` are integers.

```
>>> Or(x, y).subs(x, 0)
y
```

class `sympy.logic.boolalg.Not(arg)`

Logical Not function (negation)

Returns true if the statement is false or False. Returns false if the statement is true or True.

Examples

```
>>> from sympy import Not, And, Or
>>> from sympy.abc import x, A, B
>>> Not(True)
False
>>> Not(False)
True
>>> Not(And(True, False))
True
>>> Not(Or(True, False))
False
>>> Not(And(And(True, x), Or(x, False)))
```

(continues on next page)

(continued from previous page)

```
~X
>>> ~X
~X
>>> Not(And(Or(A, B), Or(~A, ~B)))
~((A | B) & (~A | ~B))
```

Notes

- The `~` operator is provided as a convenience, but note that its use here is different from its normal use in Python, which is bitwise not. In particular, `~a` and `Not(a)` will be different if `a` is an integer. Furthermore, since bools in Python subclass from `int`, `~True` is the same as `~1` which is `-2`, which has a boolean value of `True`. To avoid this issue, use the SymPy boolean types `true` and `false`.

```
>>> from sympy import true
>>> ~True
-2
>>> ~true
False
```

class `sympy.logic.boolalg.Xor(*args)`

Logical XOR (exclusive OR) function.

Returns `True` if an odd number of the arguments are `True` and the rest are `False`.

Returns `False` if an even number of the arguments are `True` and the rest are `False`.

Examples

```
>>> from sympy.logic.boolalg import Xor
>>> from sympy import symbols
>>> x, y = symbols('x y')
>>> Xor(True, False)
True
>>> Xor(True, True)
False
>>> Xor(True, False, True, True, False)
True
>>> Xor(True, False, True, False)
False
>>> x ^ y
x ^ y
```


Notes

The $\wedge$ operator is provided as a convenience, but note that its use here is different from its normal use in Python, which is bitwise xor. In particular, $a \wedge b$ and `Xor(a, b)` will be different if a and b are integers.

```
>>> Xor(x, y).subs(y, 0)
x
```

class `sympy.logic.boolalg.Nand(*args)`

Logical NAND function.

It evaluates its arguments in order, giving True immediately if any of them are False, and False if they are all True.

Returns True if any of the arguments are False Returns False if all arguments are True

Examples

```
>>> from sympy.logic.boolalg import Nand
>>> from sympy import symbols
>>> x, y = symbols('x y')
>>> Nand(False, True)
True
>>> Nand(True, True)
False
>>> Nand(x, y)
~(x & y)
```

class `sympy.logic.boolalg.Nor(*args)`

Logical NOR function.

It evaluates its arguments in order, giving False immediately if any of them are True, and True if they are all False.

Returns False if any argument is True Returns True if all arguments are False

Examples

```
>>> from sympy.logic.boolalg import Nor
>>> from sympy import symbols
>>> x, y = symbols('x y')
```

```
>>> Nor(True, False)
False
>>> Nor(True, True)
False
>>> Nor(False, True)
False
>>> Nor(False, False)
True
```

(continues on next page)

(continued from previous page)

```
>>> Nor(x, y)
~(x | y)
```

class sympy.logic.boolalg.Xnor(*args)

Logical XNOR function.

Returns False if an odd number of the arguments are True and the rest are False.

Returns True if an even number of the arguments are True and the rest are False.

Examples

```
>>> from sympy.logic.boolalg import Xnor
>>> from sympy import symbols
>>> x, y = symbols('x y')
>>> Xnor(True, False)
False
>>> Xnor(True, True)
True
>>> Xnor(True, False, True, True, False)
False
>>> Xnor(True, False, True, False)
True
```

class sympy.logic.boolalg.Implies(*args)

Logical implication.

A implies B is equivalent to if A then B. Mathematically, it is written as $A \Rightarrow B$ and is equivalent to $\neg A \vee B$ or $\neg A \mid B$.

Accepts two Boolean arguments; A and B. Returns False if A is True and B is False Returns True otherwise.

Examples

```
>>> from sympy.logic.boolalg import Implies
>>> from sympy import symbols
>>> x, y = symbols('x y')
```

```
>>> Implies(True, False)
False
>>> Implies(False, False)
True
>>> Implies(True, True)
True
>>> Implies(False, True)
True
>>> x >> y
Implies(x, y)
>>> y << x
Implies(x, y)
```

Notes

The `>>` and `<<` operators are provided as a convenience, but note that their use here is different from their normal use in Python, which is bit shifts. Hence, `Implies(a, b)` and `a >> b` will return different things if `a` and `b` are integers. In particular, since Python considers `True` and `False` to be integers, `True >> True` will be the same as `1 >> 1`, i.e., 0, which has a truth value of `False`. To avoid this issue, use the SymPy objects `true` and `false`.

```
>>> from sympy import true, false
>>> True >> False
1
>>> true >> false
False
```

class `sympy.logic.boolalg.Equivalent(*args)`

Equivalence relation.

`Equivalent(A, B)` is `True` iff `A` and `B` are both `True` or both `False`.

Returns `True` if all of the arguments are logically equivalent. Returns `False` otherwise.

For two arguments, this is equivalent to [*Xnor*](#) (page 1170).

Examples

```
>>> from sympy.logic.boolalg import Equivalent, And
>>> from sympy.abc import x
>>> Equivalent(False, False, False)
True
>>> Equivalent(True, False, False)
False
>>> Equivalent(x, And(x, True))
True
```

class `sympy.logic.boolalg.ITE(*args)`

If-then-else clause.

`ITE(A, B, C)` evaluates and returns the result of `B` if `A` is `true` else it returns the result of `C`. All args must be Booleans.

From a logic gate perspective, `ITE` corresponds to a 2-to-1 multiplexer, where `A` is the select signal.

Examples

```
>>> from sympy.logic.boolalg import ITE, And, Xor, Or
>>> from sympy.abc import x, y, z
>>> ITE(True, False, True)
False
>>> ITE(Or(True, False), And(True, True), Xor(True, True))
True
```

(continues on next page)

(continued from previous page)

```
>>> ITE(x, y, z)
ITE(x, y, z)
>>> ITE(True, x, y)
x
>>> ITE(False, x, y)
y
>>> ITE(x, y, y)
y
```

Trying to use non-Boolean args will generate a `TypeError`:

```
>>> ITE(True, [], ())
Traceback (most recent call last):
...
TypeError: expecting bool, Boolean or ITE, not `[]`
```

class `sympy.logic.boolalg.Exclusive(*args)`

True if only one or no argument is true.

`Exclusive(A, B, C)` is equivalent to $\sim(A \ \& \ B) \ \& \ \sim(A \ \& \ C) \ \& \ \sim(B \ \& \ C)$.

For two arguments, this is equivalent to [Xor](#) (page 1168).

Examples

```
>>> from sympy.logic.boolalg import Exclusive
>>> Exclusive(False, False, False)
True
>>> Exclusive(False, True, False)
True
>>> Exclusive(False, True, True)
False
```

The following functions can be used to handle Algebraic, Conjunctive, Disjunctive, and Negated Normal forms:

`sympy.logic.boolalg.to_anf(expr, deep=True)`

Converts `expr` to Algebraic Normal Form (ANF).

ANF is a canonical normal form, which means that two equivalent formulas will convert to the same ANF.

A logical expression is in ANF if it has the form

$$1 \oplus a \oplus b \oplus ab \oplus abc$$

i.e. it can be:

- purely true,
- purely false,
- conjunction of variables,
- exclusive disjunction.

The exclusive disjunction can only contain true, variables or conjunction of variables. No negations are permitted.

If `deep` is `False`, arguments of the boolean expression are considered variables, i.e. only the top-level expression is converted to ANF.

Examples

```
>>> from sympy.logic.boolalg import And, Or, Not, Implies, Equivalent
>>> from sympy.logic.boolalg import to_anf
>>> from sympy.abc import A, B, C
>>> to_anf(Not(A))
A ^ True
>>> to_anf(And(Or(A, B), Not(C)))
A ^ B ^ (A & B) ^ (A & C) ^ (B & C) ^ (A & B & C)
>>> to_anf(Implies(Not(A), Equivalent(B, C)), deep=False)
True ^ ~A ^ (~A & (Equivalent(B, C)))
```

`sympy.logic.boolalg.to_cnf(expr, simplify=False, force=False)`

Convert a propositional logical sentence `expr` to conjunctive normal form: $((A \mid \sim B \mid \dots) \& (B \mid C \mid \dots) \& \dots)$. If `simplify` is `True`, `expr` is evaluated to its simplest CNF form using the Quine-McCluskey algorithm; this may take a long time. If there are more than 8 variables the `force` flag must be set to `True` to simplify (default is `False`).

Examples

```
>>> from sympy.logic.boolalg import to_cnf
>>> from sympy.abc import A, B, D
>>> to_cnf(~(A | B) | D)
(D | ~A) & (D | ~B)
>>> to_cnf((A | B) & (A | ~A), True)
A | B
```

`sympy.logic.boolalg.to_dnf(expr, simplify=False, force=False)`

Convert a propositional logical sentence `expr` to disjunctive normal form: $((A \& \sim B \& \dots) \mid (B \& C \& \dots) \mid \dots)$. If `simplify` is `True`, `expr` is evaluated to its simplest DNF form using the Quine-McCluskey algorithm; this may take a long time. If there are more than 8 variables, the `force` flag must be set to `True` to simplify (default is `False`).

Examples

```
>>> from sympy.logic.boolalg import to_dnf
>>> from sympy.abc import A, B, C
>>> to_dnf(B & (A | C))
(A & B) | (B & C)
>>> to_dnf((A & B) | (A & ~B) | (B & C) | (~B & C), True)
A | C
```

`sympy.logic.boolalg.to_nnf(expr, simplify=True)`

Converts `expr` to Negation Normal Form (NNF).

A logical expression is in NNF if it contains only [And](#) (page 1166), [Or](#) (page 1167) and [Not](#) (page 1167), and [Not](#) (page 1167) is applied only to literals. If `simplify` is `True`, the result contains no redundant clauses.

Examples

```
>>> from sympy.abc import A, B, C, D
>>> from sympy.logic.boolalg import Not, Equivalent, to_nnf
>>> to_nnf(Not((~A & ~B) | (C & D)))
(A | B) & (~C | ~D)
>>> to_nnf(Equivalent(A >> B, B >> A))
(A | ~B | (A & ~B)) & (B | ~A | (B & ~A))
```

`sympy.logic.boolalg.is_anf(expr)`

Checks if `expr` is in Algebraic Normal Form (ANF).

A logical expression is in ANF if it has the form

$$1 \oplus a \oplus b \oplus ab \oplus abc$$

i.e. it is purely true, purely false, conjunction of variables or exclusive disjunction. The exclusive disjunction can only contain true, variables or conjunction of variables. No negations are permitted.

Examples

```
>>> from sympy.logic.boolalg import And, Not, Xor, true, is_anf
>>> from sympy.abc import A, B, C
>>> is_anf(true)
True
>>> is_anf(A)
True
>>> is_anf(And(A, B, C))
True
>>> is_anf(Xor(A, Not(B)))
False
```

`sympy.logic.boolalg.is_cnf(expr)`

Test whether or not an expression is in conjunctive normal form.

Examples

```
>>> from sympy.logic.boolalg import is_cnf
>>> from sympy.abc import A, B, C
>>> is_cnf(A | B | C)
True
>>> is_cnf(A & B & C)
True
>>> is_cnf((A & B) | C)
False
```

`sympy.logic.boolalg.is_dnf(expr)`

Test whether or not an expression is in disjunctive normal form.

Examples

```
>>> from sympy.logic.boolalg import is_dnf
>>> from sympy.abc import A, B, C
>>> is_dnf(A | B | C)
True
>>> is_dnf(A & B & C)
True
>>> is_dnf((A & B) | C)
True
>>> is_dnf(A & (B | C))
False
```

`sympy.logic.boolalg.is_nnf(expr, simplified=True)`

Checks if `expr` is in Negation Normal Form (NNF).

A logical expression is in NNF if it contains only *And* (page 1166), *Or* (page 1167) and *Not* (page 1167), and *Not* (page 1167) is applied only to literals. If `simplified` is `True`, checks if result contains no redundant clauses.

Examples

```
>>> from sympy.abc import A, B, C
>>> from sympy.logic.boolalg import Not, is_nnf
>>> is_nnf(A & B | ~C)
True
>>> is_nnf((A | ~A) & (B | C))
False
>>> is_nnf((A | ~A) & (B | C), False)
True
>>> is_nnf(Not(A & B) | C)
False
>>> is_nnf((A >> B) & (B >> A))
False
```

`sympy.logic.boolalg.gateinputcount(expr)`

Return the total number of inputs for the logic gates realizing the Boolean expression.

Returns

int

Number of gate inputs

Note

Not all Boolean functions count as gate here, only those that are considered to be standard gates. These are: [And](#) (page 1166), [Or](#) (page 1167), [Xor](#) (page 1168), [Not](#) (page 1167), and [ITE](#) (page 1171) (multiplexer). [Nand](#) (page 1169), [Nor](#) (page 1169), and [Xnor](#) (page 1170) will be evaluated to `Not(And())` etc.

Examples

```
>>> from sympy.logic import And, Or, Nand, Not, gateinputcount
>>> from sympy.abc import x, y, z
>>> expr = And(x, y)
>>> gateinputcount(expr)
2
>>> gateinputcount(Or(expr, z))
4
```

Note that `Nand` is automatically evaluated to `Not(And())` so

```
>>> gateinputcount(Nand(x, y, z))
4
>>> gateinputcount(Not(And(x, y, z)))
4
```

Although this can be avoided by using `evaluate=False`

```
>>> gateinputcount(Nand(x, y, z, evaluate=False))
3
```

Also note that a comparison will count as a Boolean variable:

```
>>> gateinputcount(And(x > z, y >= 2))
2
```

As will a symbol: `>>> gateinputcount(x)` 0

Simplification and equivalence-testing

`sympy.logic.boolalg.simplify_logic(expr, form=None, deep=True, force=False)`

This function simplifies a boolean function to its simplified version in SOP or POS form. The return type is an [Or](#) (page 1167) or [And](#) (page 1166) object in SymPy.

Parameters

expr : Boolean expression

form : string ('cnf' or 'dnf') or None (default).

If 'cnf' or 'dnf', the simplest expression in the corresponding normal form is returned; if None, the answer is returned according to the form with fewest args (in CNF by default).

deep : bool (default True)

Indicates whether to recursively simplify any non-boolean functions contained within the input.

force : bool (default False)

As the simplifications require exponential time in the number of variables, there is by default a limit on expressions with 8 variables. When the expression has more than 8 variables only symbolical simplification (controlled by deep) is made. By setting force to True, this limit is removed. Be aware that this can lead to very long simplification times.

Examples

```
>>> from sympy.logic import simplify_logic
>>> from sympy.abc import x, y, z
>>> from sympy import S
>>> b = (~x & ~y & ~z) | (~x & ~y & z)
>>> simplify_logic(b)
~x & ~y
```

```
>>> S(b)
(z & ~x & ~y) | (~x & ~y & ~z)
>>> simplify_logic(_)
~x & ~y
```

SymPy's `simplify()` (page 661) function can also be used to simplify logic expressions to their simplest forms.

`sympy.logic.boolalg.bool_map(bool1, bool2)`

Return the simplified version of *bool1*, and the mapping of variables that makes the two expressions *bool1* and *bool2* represent the same logical behaviour for some correspondence between the variables of each. If more than one mappings of this sort exist, one of them is returned.

For example, `And(x, y)` is logically equivalent to `And(a, b)` for the mapping `{x: a, y: b}` or `{x: b, y: a}`. If no such mapping exists, return False.

Examples

```
>>> from sympy import SOPform, bool_map, Or, And, Not, Xor
>>> from sympy.abc import w, x, y, z, a, b, c, d
>>> function1 = SOPform([x, z, y], [[1, 0, 1], [0, 0, 1]])
>>> function2 = SOPform([a, b, c], [[1, 0, 1], [1, 0, 0]])
>>> bool_map(function1, function2)
(y & ~z, {y: a, z: b})
```

The results are not necessarily unique, but they are canonical. Here, (w, z) could be (a, d) or (d, a) :

```
>>> eq = Or(And(Not(y), w), And(Not(y), z), And(x, y))
>>> eq2 = Or(And(Not(c), a), And(Not(c), d), And(b, c))
>>> bool_map(eq, eq2)
((x & y) | (w & ~y) | (z & ~y), {w: a, x: b, y: c, z: d})
>>> eq = And(Xor(a, b), c, And(c,d))
>>> bool_map(eq, eq.subs(c, x))
(c & d & (a | b) & (~a | ~b), {a: a, b: b, c: d, d: x})
```

Manipulating expressions

The following functions can be used to manipulate Boolean expressions:

`sympy.logic.boolalg.distribute_and_over_or(expr)`

Given a sentence *expr* consisting of conjunctions and disjunctions of literals, return an equivalent sentence in CNF.

Examples

```
>>> from sympy.logic.boolalg import distribute_and_over_or, And, Or, Not
>>> from sympy.abc import A, B, C
>>> distribute_and_over_or(Or(A, And(Not(B), Not(C))))
(A | ~B) & (A | ~C)
```

`sympy.logic.boolalg.distribute_or_over_and(expr)`

Given a sentence *expr* consisting of conjunctions and disjunctions of literals, return an equivalent sentence in DNF.

Note that the output is NOT simplified.

Examples

```
>>> from sympy.logic.boolalg import distribute_or_over_and, And, Or, Not
>>> from sympy.abc import A, B, C
>>> distribute_or_over_and(And(Or(Not(A), B), C))
(B & C) | (C & ~A)
```

`sympy.logic.boolalg.distribute_xor_over_and(expr)`

Given a sentence *expr* consisting of conjunction and exclusive disjunctions of literals, return an equivalent exclusive disjunction.

Note that the output is NOT simplified.

Examples

```
>>> from sympy.logic.boolalg import distribute_xor_over_and, And, Xor, Not
>>> from sympy.abc import A, B, C
>>> distribute_xor_over_and(And(Xor(Not(A), B), C))
(B & C) ^ (C & ~A)
```

`sympy.logic.boolalg.eliminate_implications(expr)`

Change *Implies* (page 1170) and *Equivalent* (page 1171) into *And* (page 1166), *Or* (page 1167), and *Not* (page 1167). That is, return an expression that is equivalent to `expr`, but has only `&`, `|`, and `~` as logical operators.

Examples

```
>>> from sympy.logic.boolalg import Implies, Equivalent,
>>> eliminate_implications
>>> from sympy.abc import A, B, C
>>> eliminate_implications(Implies(A, B))
B | ~A
>>> eliminate_implications(Equivalent(A, B))
(A | ~B) & (B | ~A)
>>> eliminate_implications(Equivalent(A, B, C))
(A | ~C) & (B | ~A) & (C | ~B)
```

Truth tables and related functions

It is possible to create a truth table for a Boolean function with

`sympy.logic.boolalg.truth_table(expr, variables, input=True)`

Return a generator of all possible configurations of the input variables, and the result of the boolean expression for those values.

Parameters

expr : Boolean expression

variables : list of variables

input : bool (default True)

Indicates whether to return the input combinations.

Examples

```
>>> from sympy.logic.boolalg import truth_table
>>> from sympy.abc import x,y
>>> table = truth_table(x >> y, [x, y])
>>> for t in table:
...     print('{0} -> {1}'.format(*t))
[0, 0] -> True
[0, 1] -> True
[1, 0] -> False
[1, 1] -> True
```

```
>>> table = truth_table(x | y, [x, y])
>>> list(table)
[[0, 0], False), ([0, 1], True), ([1, 0], True), ([1, 1], True)]
```

If input is False, `truth_table` returns only a list of truth values. In this case, the corresponding input values of variables can be deduced from the index of a given output.

```
>>> from sympy.utilities.iterables import ibin
>>> vars = [y, x]
>>> values = truth_table(x >> y, vars, input=False)
>>> values = list(values)
>>> values
[True, False, True, True]
```

```
>>> for i, value in enumerate(values):
...     print('{0} -> {1}'.format(list(zip(
...         vars, ibin(i, len(vars))))), value))
[(y, 0), (x, 0)] -> True
[(y, 0), (x, 1)] -> False
[(y, 1), (x, 0)] -> True
[(y, 1), (x, 1)] -> True
```

For mapping between integer representations of truth table positions, lists of zeros and ones and symbols, the following functions can be used:

`sympy.logic.boolalg.integer_to_term(n, bits=None, str=False)`

Return a list of length `bits` corresponding to the binary value of `n` with small bits to the right (last). If `bits` is omitted, the length will be the number required to represent `n`. If the bits are desired in reversed order, use the `[::-1]` slice of the returned list.

If a sequence of all bits-length lists starting from `[0, 0, ..., 0]` through `[1, 1, ..., 1]` are desired, pass a non-integer for `bits`, e.g. `'all'`.

If the bit *string* is desired pass `str=True`.

Examples

```
>>> from sympy.utilities.iterables import ibin
>>> ibin(2)
[1, 0]
>>> ibin(2, 4)
[0, 0, 1, 0]
```

If all lists corresponding to 0 to $2^n - 1$, pass a non-integer for bits:

```
>>> bits = 2
>>> for i in ibin(2, 'all'):
...     print(i)
(0, 0)
(0, 1)
(1, 0)
(1, 1)
```

If a bit string is desired of a given length, use `str=True`:

```
>>> n = 123
>>> bits = 10
>>> ibin(n, bits, str=True)
'0001111011'
>>> ibin(n, bits, str=True)[::-1] # small bits left
'1101111000'
>>> list(ibin(3, 'all', str=True))
['000', '001', '010', '011', '100', '101', '110', '111']
```

`sympy.logic.boolalg.term_to_integer(term)`

Return an integer corresponding to the base-2 digits given by *term*.

Parameters

term : a string or list of ones and zeros

Examples

```
>>> from sympy.logic.boolalg import term_to_integer
>>> term_to_integer([1, 0, 0])
4
>>> term_to_integer('100')
4
```

`sympy.logic.boolalg.bool_maxterm(k, variables)`

Return the *k*-th maxterm.

Each maxterm is assigned an index based on the opposite conventional binary encoding used for minterms. The maxterm convention assigns the value 0 to the direct form and 1 to the complemented form.

Parameters

k : int or list of 1's and 0's (complementation pattern)

variables : list of variables

Examples

```
>>> from sympy.logic.boolalg import bool_maxterm
>>> from sympy.abc import x, y, z
>>> bool_maxterm([1, 0, 1], [x, y, z])
y | ~x | ~z
>>> bool_maxterm(6, [x, y, z])
z | ~x | ~y
```

References

[R560]

`sympy.logic.boolalg.bool_minterm(k, variables)`

Return the *k*-th minterm.

Minterms are numbered by a binary encoding of the complementation pattern of the variables. This convention assigns the value 1 to the direct form and 0 to the complemented form.

Parameters

k : int or list of 1's and 0's (complementation patter)

variables : list of variables

Examples

```
>>> from sympy.logic.boolalg import bool_minterm
>>> from sympy.abc import x, y, z
>>> bool_minterm([1, 0, 1], [x, y, z])
x & z & ~y
>>> bool_minterm(6, [x, y, z])
x & y & ~z
```

References

[R561]

`sympy.logic.boolalg.bool_monomial(k, variables)`

Return the *k*-th monomial.

Monomials are numbered by a binary encoding of the presence and absences of the variables. This convention assigns the value 1 to the presence of variable and 0 to the absence of variable.

Each boolean function can be uniquely represented by a Zhegalkin Polynomial (Algebraic Normal Form). The Zhegalkin Polynomial of the boolean function with n variables can contain up to 2^n monomials. We can enumerate all the monomials. Each monomial is fully specified by the presence or absence of each variable.

For example, boolean function with four variables (*a*, *b*, *c*, *d*) can contain up to $2^4 = 16$ monomials. The 13-th monomial is the product *a* & *b* & *d*, because 13 in binary is 1, 1, 0, 1.

Parameters**k** : int or list of 1's and 0's**variables** : list of variables**Examples**

```
>>> from sympy.logic.boolalg import bool_monomial
>>> from sympy.abc import x, y, z
>>> bool_monomial([1, 0, 1], [x, y, z])
x & z
>>> bool_monomial(6, [x, y, z])
x & y
```

`sympy.logic.boolalg.anf_coeffs(truthvalues)`

Convert a list of truth values of some boolean expression to the list of coefficients of the polynomial mod 2 (exclusive disjunction) representing the boolean expression in ANF (i.e., the “Zhegalkin polynomial”).

There are 2^n possible Zhegalkin monomials in n variables, since each monomial is fully specified by the presence or absence of each variable.

We can enumerate all the monomials. For example, boolean function with four variables (a , b , c , d) can contain up to $2^4 = 16$ monomials. The 13-th monomial is the product $a \& b \& d$, because 13 in binary is 1, 1, 0, 1.

A given monomial's presence or absence in a polynomial corresponds to that monomial's coefficient being 1 or 0 respectively.

Examples

```
>>> from sympy.logic.boolalg import anf_coeffs, bool_monomial, Xor
>>> from sympy.abc import a, b, c
>>> truthvalues = [0, 1, 1, 0, 0, 1, 0, 1]
>>> coeffs = anf_coeffs(truthvalues)
>>> coeffs
[0, 1, 1, 0, 0, 0, 1, 0]
>>> polynomial = Xor(*[
...     bool_monomial(k, [a, b, c])
...     for k, coeff in enumerate(coeffs) if coeff == 1
... ])
>>> polynomial
b ^ c ^ (a & b)
```

`sympy.logic.boolalg.to_int_repr(clauses, symbols)`

Takes clauses in CNF format and puts them into an integer representation.

Examples

```
>>> from sympy.logic.boolalg import to_int_repr
>>> from sympy.abc import x, y
>>> to_int_repr([x | y, y], [x, y]) == [{1, 2}, {2}]
True
```

Inference

This module implements some inference routines in propositional logic.

The function `satisfiable` will test that a given Boolean expression is satisfiable, that is, you can assign values to the variables to make the sentence True.

For example, the expression $x \ \& \ \sim x$ is not satisfiable, since there are no values for x that make this sentence True. On the other hand, $(x \mid y) \ \& \ (x \mid \sim y) \ \& \ (\sim x \mid y)$ is satisfiable with both x and y being True.

```
>>> from sympy.logic.inference import satisfiable
>>> from sympy import Symbol
>>> x = Symbol('x')
>>> y = Symbol('y')
>>> satisfiable(x & ~x)
False
>>> satisfiable((x | y) & (x | ~y) & (~x | y))
{x: True, y: True}
```

As you see, when a sentence is satisfiable, it returns a model that makes that sentence True. If it is not satisfiable it will return False.

`sympy.logic.inference.satisfiable(expr, algorithm=None, all_models=False, minimal=False)`

Check satisfiability of a propositional sentence. Returns a model when it succeeds. Returns `{true: true}` for trivially true expressions.

On setting `all_models` to True, if given `expr` is satisfiable then returns a generator of models. However, if `expr` is unsatisfiable then returns a generator containing the single element False.

Examples

```
>>> from sympy.abc import A, B
>>> from sympy.logic.inference import satisfiable
>>> satisfiable(A & ~B)
{A: True, B: False}
>>> satisfiable(A & ~A)
False
>>> satisfiable(True)
{True: True}
>>> next(satisfiable(A & ~A, all_models=True))
False
```

(continues on next page)

(continued from previous page)

```

>>> models = satisfiable((A >> B) & B, all_models=True)
>>> next(models)
{A: False, B: True}
>>> next(models)
{A: True, B: True}
>>> def use_models(models):
...     for model in models:
...         if model:
...             # Do something with the model.
...             print(model)
...         else:
...             # Given expr is unsatisfiable.
...             print("UNSAT")
>>> use_models(satisfiable(A >> ~A, all_models=True))
{A: False}
>>> use_models(satisfiable(A ^ A, all_models=True))
UNSAT

```

Sets

Basic Sets

Set

class sympy.sets.sets.**Set**(*args)

The base class for any kind of set.

Explanation

This is not meant to be used directly as a container of items. It does not behave like the builtin set; see [FiniteSet](#) (page 1197) for that.

Real intervals are represented by the [Interval](#) (page 1194) class and unions of sets by the [Union](#) (page 1197) class. The empty set is represented by the [EmptySet](#) (page 1202) class and available as a singleton as `S.EmptySet`.

property boundary

The boundary or frontier of a set.

Explanation

A point x is on the boundary of a set S if

1. x is in the closure of S . I.e. Every neighborhood of x contains a point in S .
2. x is not in the interior of S . I.e. There does not exist an open set centered on x contained entirely within S .

There are the points on the outer rim of S . If S is open then these points need not actually be contained within S .

For example, the boundary of an interval is its start and end points. This is true regardless of whether or not the interval is open.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 1).boundary
{0, 1}
>>> Interval(0, 1, True, False).boundary
{0, 1}
```

property closure

Property method which returns the closure of a set. The closure is defined as the union of the set itself and its boundary.

Examples

```
>>> from sympy import S, Interval
>>> S.Reals.closure
Reals
>>> Interval(0, 1).closure
Interval(0, 1)
```

complement(*universe*)

The complement of 'self' w.r.t the given universe.

Examples

```
>>> from sympy import Interval, S
>>> Interval(0, 1).complement(S.Reals)
Union(Interval.open(-oo, 0), Interval.open(1, oo))
```

```
>>> Interval(0, 1).complement(S.UniversalSet)
Complement(UniversalSet, Interval(0, 1))
```

contains(*other*)

Returns a SymPy value indicating whether *other* is contained in *self*: *true* if it is, *false* if it is not, else an unevaluated *Contains* expression (or, as in the case of *ConditionSet* and a union of *FiniteSet*/Intervals, an expression indicating the conditions for containment).

Examples

```
>>> from sympy import Interval, S
>>> from sympy.abc import x
```

```
>>> Interval(0, 1).contains(0.5)
True
```

As a shortcut it is possible to use the `in` operator, but that will raise an error unless an affirmative true or false is not obtained.

```
>>> Interval(0, 1).contains(x)
(0 <= x) & (x <= 1)
>>> x in Interval(0, 1)
Traceback (most recent call last):
...
TypeError: did not evaluate to a bool: None
```

The result of `'in'` is a bool, not a SymPy value

```
>>> 1 in Interval(0, 2)
True
>>> _ is S.true
False
```

property `inf`

The infimum of `self`.

Examples

```
>>> from sympy import Interval, Union
>>> Interval(0, 1).inf
0
>>> Union(Interval(0, 1), Interval(2, 3)).inf
0
```

property `interior`

Property method which returns the interior of a set. The interior of a set `S` consists all points of `S` that do not belong to the boundary of `S`.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 1).interior
Interval.open(0, 1)
>>> Interval(0, 1).boundary.interior
EmptySet
```

`intersect(other)`

Returns the intersection of `'self'` and `'other'`.

Examples

```
>>> from sympy import Interval
```

```
>>> Interval(1, 3).intersect(Interval(1, 2))
Interval(1, 2)
```

```
>>> from sympy import imageset, Lambda, symbols, S
>>> n, m = symbols('n m')
>>> a = imageset(Lambda(n, 2*n), S.Integers)
>>> a.intersect(imageset(Lambda(m, 2*m + 1), S.Integers))
EmptySet
```

intersection(*other*)

Alias for [*intersect\(\)*](#) (page 1187)

property is_closed

A property method to check whether a set is closed.

Explanation

A set is closed if its complement is an open set. The closedness of a subset of the reals is determined with respect to $\mathbb{R}$ and its standard topology.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 1).is_closed
True
```

is_disjoint(*other*)

Returns True if self and other are disjoint.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 2).is_disjoint(Interval(1, 2))
False
>>> Interval(0, 2).is_disjoint(Interval(3, 4))
True
```

References

[R744]

property `is_open`

Property method to check whether a set is open.

Explanation

A set is open if and only if it has an empty intersection with its boundary. In particular, a subset A of the reals is open if and only if each one of its points is contained in an open interval that is a subset of A .

Examples

```
>>> from sympy import S
>>> S.Reals.is_open
True
>>> S.Rationals.is_open
False
```

`is_proper_subset(other)`

Returns True if self is a proper subset of other.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 0.5).is_proper_subset(Interval(0, 1))
True
>>> Interval(0, 1).is_proper_subset(Interval(0, 1))
False
```

`is_proper_superset(other)`

Returns True if self is a proper superset of other.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 1).is_proper_superset(Interval(0, 0.5))
True
>>> Interval(0, 1).is_proper_superset(Interval(0, 1))
False
```

`is_subset(other)`

Returns True if self is a subset of other.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 0.5).is_subset(Interval(0, 1))
True
>>> Interval(0, 1).is_subset(Interval(0, 1, left_open=True))
False
```

is_superset(*other*)

Returns True if self is a superset of other.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 0.5).is_superset(Interval(0, 1))
False
>>> Interval(0, 1).is_superset(Interval(0, 1, left_open=True))
True
```

isdisjoint(*other*)

Alias for [*is_disjoint\(\)*](#) (page 1188)

issubset(*other*)

Alias for [*is_subset\(\)*](#) (page 1189)

issuperset(*other*)

Alias for [*is_superset\(\)*](#) (page 1190)

property kind

The kind of a Set

Explanation

Any [*Set*](#) (page 1185) will have kind [*SetKind*](#) (page 1217) which is parametrised by the kind of the elements of the set. For example most sets are sets of numbers and will have kind `SetKind(NumberKind)`. If elements of sets are different in kind than their kind will be `SetKind(UndefinedKind)`. See [*sympy.core.kind.Kind*](#) (page 1073) for an explanation of the kind system.

Examples

```
>>> from sympy import Interval, Matrix, FiniteSet, EmptySet,
↪ ProductSet, PowerSet
```

```
>>> FiniteSet(Matrix([1, 2])).kind
SetKind(MatrixKind(NumberKind))
```

```
>>> Interval(1, 2).kind
SetKind(NumberKind)
```

```
>>> EmptySet.kind
SetKind()
```

A [*sympy.sets.powerset.PowerSet*](#) (page 1213) is a set of sets:

```
>>> PowerSet({1, 2, 3}).kind
SetKind(SetKind(NumberKind))
```

A [*ProductSet*](#) (page 1199) represents the set of tuples of elements of other sets. Its kind is [*sympy.core.containers.TupleKind*](#) (page 1069) parametrised by the kinds of the elements of those sets:

```
>>> p = ProductSet(FiniteSet(1, 2), FiniteSet(3, 4))
>>> list(p)
[(1, 3), (2, 3), (1, 4), (2, 4)]
>>> p.kind
SetKind(TupleKind(NumberKind, NumberKind))
```

When all elements of the set do not have same kind, the kind will be returned as `SetKind(UndefinedKind)`:

```
>>> FiniteSet(0, Matrix([1, 2])).kind
SetKind(UndefinedKind)
```

The kind of the elements of a set are given by the `element_kind` attribute of `SetKind`:

```
>>> Interval(1, 2).kind.element_kind
NumberKind
```

See also:

[*NumberKind*](#) (page 1074), [*sympy.core.kind.UndefinedKind*](#) (page 1074), [*sympy.core.containers.TupleKind*](#) (page 1069), [*MatrixKind*](#) (page 1360), [*sympy.matrices.expressions.sets.MatrixSet*](#) (page 1379), [*sympy.sets.conditionset.ConditionSet*](#) (page 1215), [*Rationals*](#) (page 1203), [*Naturals*](#) (page 1203), [*Integers*](#) (page 1204), [*sympy.sets.fancysets.ImageSet*](#) (page 1206), [*sympy.sets.fancysets.Range*](#) (page 1207), [*sympy.sets.fancysets.ComplexRegion*](#) (page 1209), [*sympy.sets.powerset.PowerSet*](#) (page 1213), [*sympy.sets.sets.ProductSet*](#) (page 1199), [*sympy.sets.sets.Interval*](#) (page 1194), [*sympy.sets.sets.Union*](#) (page 1197), [*sympy.sets.sets.Intersection*](#) (page 1198), [*sympy.sets.sets.Complement*](#) (page 1200), [*sympy.sets.sets.EmptySet*](#) (page 1202), [*sympy.sets.sets.UniversalSet*](#) (page 1202), [*sympy.sets.sets.FiniteSet*](#) (page 1197), [*sympy.sets.sets.SymmetricDifference*](#) (page 1201), [*sympy.sets.sets.DisjointUnion*](#) (page 1201)

property measure

The (Lebesgue) measure of self.

Examples

```
>>> from sympy import Interval, Union
>>> Interval(0, 1).measure
1
>>> Union(Interval(0, 1), Interval(2, 3)).measure
2
```

powerset()

Find the Power set of self.

Examples

```
>>> from sympy import EmptySet, FiniteSet, Interval
```

A power set of an empty set:

```
>>> A = EmptySet
>>> A.powerset()
{EmptySet}
```

A power set of a finite set:

```
>>> A = FiniteSet(1, 2)
>>> a, b, c = FiniteSet(1), FiniteSet(2), FiniteSet(1, 2)
>>> A.powerset() == FiniteSet(a, b, c, EmptySet)
True
```

A power set of an interval:

```
>>> Interval(1, 2).powerset()
PowerSet(Interval(1, 2))
```

References

[R745]

property sup

The supremum of self.

Examples

```
>>> from sympy import Interval, Union
>>> Interval(0, 1).sup
1
>>> Union(Interval(0, 1), Interval(2, 3)).sup
3
```

symmetric_difference(*other*)

Returns symmetric difference of self and other.

Examples

```
>>> from sympy import Interval, S
>>> Interval(1, 3).symmetric_difference(S.Reals)
Union(Interval.open(-oo, 1), Interval.open(3, oo))
>>> Interval(1, 10).symmetric_difference(S.Reals)
Union(Interval.open(-oo, 1), Interval.open(10, oo))
```

```
>>> from sympy import S, EmptySet
>>> S.Reals.symmetric_difference(EmptySet)
Reals
```

References

[R746]

union(*other*)

Returns the union of self and other.

Examples

As a shortcut it is possible to use the + operator:

```
>>> from sympy import Interval, FiniteSet
>>> Interval(0, 1).union(Interval(2, 3))
Union(Interval(0, 1), Interval(2, 3))
>>> Interval(0, 1) + Interval(2, 3)
Union(Interval(0, 1), Interval(2, 3))
>>> Interval(1, 2, True, True) + FiniteSet(2, 3)
Union({3}, Interval.Lopen(1, 2))
```

Similarly it is possible to use the - operator for set differences:

```
>>> Interval(0, 2) - Interval(0, 1)
Interval.Lopen(1, 2)
>>> Interval(1, 3) - FiniteSet(2)
Union(Interval.Ropen(1, 2), Interval.Lopen(2, 3))
```

sympy.sets.sets.imageset(*args)

Return an image of the set under transformation f.

Explanation

If this function cannot compute the image, it returns an unevaluated ImageSet object.

$$\{f(x) \mid x \in \text{self}\}$$

Examples

```
>>> from sympy import S, Interval, imageset, sin, Lambda
>>> from sympy.abc import x
```

```
>>> imageset(x, 2*x, Interval(0, 2))
Interval(0, 4)
```

```
>>> imageset(lambda x: 2*x, Interval(0, 2))
Interval(0, 4)
```

```
>>> imageset(Lambda(x, sin(x)), Interval(-2, 1))
ImageSet(Lambda(x, sin(x)), Interval(-2, 1))
```

```
>>> imageset(sin, Interval(-2, 1))
ImageSet(Lambda(x, sin(x)), Interval(-2, 1))
>>> imageset(lambda y: x + y, Interval(-2, 1))
ImageSet(Lambda(y, x + y), Interval(-2, 1))
```

Expressions applied to the set of Integers are simplified to show as few negatives as possible and linear expressions are converted to a canonical form. If this is not desirable then the unevaluated ImageSet should be used.

```
>>> imageset(x, -2*x + 5, S.Integers)
ImageSet(Lambda(x, 2*x + 1), Integers)
```

See also:

[*sympy.sets.fancysets.ImageSet*](#) (page 1206)

Elementary Sets

Interval

class `sympy.sets.sets.Interval`(*start*, *end*, *left_open=False*, *right_open=False*)

Represents a real interval as a Set.

Usage:

Returns an interval with end points *start* and *end*.

For *left_open=True* (default *left_open* is *False*) the interval will be open on the left. Similarly, for *right_open=True* the interval will be open on the right.

Examples

```
>>> from sympy import Symbol, Interval
>>> Interval(0, 1)
Interval(0, 1)
>>> Interval.Ropen(0, 1)
Interval.Ropen(0, 1)
>>> Interval.Ropen(0, 1)
Interval.Ropen(0, 1)
>>> Interval.Lopen(0, 1)
Interval.Lopen(0, 1)
>>> Interval.open(0, 1)
Interval.open(0, 1)
```

```
>>> a = Symbol('a', real=True)
>>> Interval(0, a)
Interval(0, a)
```

Notes

- Only real end points are supported
- $\text{Interval}(a, b)$ with $a > b$ will return the empty set
- Use the `evalf()` method to turn an `Interval` into an mpmath `mpi` interval instance

References

[R747]

classmethod `Lopen( $a, b$ )`

Return an interval not including the left boundary.

classmethod `Ropen( $a, b$ )`

Return an interval not including the right boundary.

as_relational(x)

Rewrite an interval in terms of inequalities and logic operators.

property `end`

The right end point of the interval.

This property takes the same value as the `sup` property.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 1).end
1
```

property `is_left_unbounded`

Return True if the left endpoint is negative infinity.

property `is_right_unbounded`

Return True if the right endpoint is positive infinity.

property `left_open`

True if interval is left-open.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 1, left_open=True).left_open
True
>>> Interval(0, 1, left_open=False).left_open
False
```

classmethod `open(a, b)`

Return an interval including neither boundary.

property `right_open`

True if interval is right-open.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 1, right_open=True).right_open
True
>>> Interval(0, 1, right_open=False).right_open
False
```

property `start`

The left end point of the interval.

This property takes the same value as the `inf` property.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 1).start
0
```

FiniteSet

class sympy.sets.sets.**FiniteSet**(*args, **kwargs)

Represents a finite set of SymPy expressions.

Examples

```
>>> from sympy import FiniteSet, Symbol, Interval, Naturals0
>>> FiniteSet(1, 2, 3, 4)
{1, 2, 3, 4}
>>> 3 in FiniteSet(1, 2, 3, 4)
True
>>> FiniteSet(1, (1, 2), Symbol('x'))
{1, x, (1, 2)}
>>> FiniteSet(Interval(1, 2), Naturals0, {1, 2})
FiniteSet({1, 2}, Interval(1, 2), Naturals0)
>>> members = [1, 2, 3, 4]
>>> f = FiniteSet(*members)
>>> f
{1, 2, 3, 4}
>>> f - FiniteSet(2)
{1, 3, 4}
>>> f + FiniteSet(2, 5)
{1, 2, 3, 4, 5}
```

References

[R748]

as_relational(symbol)

Rewrite a FiniteSet in terms of equalities and logic operators.

Compound Sets

Union

class sympy.sets.sets.**Union**(*args, **kwargs)

Represents a union of sets as a [Set](#) (page 1185).

Examples

```
>>> from sympy import Union, Interval
>>> Union(Interval(1, 2), Interval(3, 4))
Union(Interval(1, 2), Interval(3, 4))
```

The Union constructor will always try to merge overlapping intervals, if possible. For example:

```
>>> Union(Interval(1, 2), Interval(2, 3))
Interval(1, 3)
```

See also:

[*Intersection*](#) (page 1198)

References

[R749]

as_relational(*symbol*)

Rewrite a Union in terms of equalities and logic operators.

Intersection

class sympy.sets.sets.**Intersection**(*args, **kwargs)

Represents an intersection of sets as a [*Set*](#) (page 1185).

Examples

```
>>> from sympy import Intersection, Interval
>>> Intersection(Interval(1, 3), Interval(2, 4))
Interval(2, 3)
```

We often use the `.intersect` method

```
>>> Interval(1,3).intersect(Interval(2,4))
Interval(2, 3)
```

See also:

[*Union*](#) (page 1197)

References

[R750]

as_relational(*symbol*)

Rewrite an Intersection in terms of equalities and logic operators

ProductSet

class sympy.sets.sets.**ProductSet**(*sets, **assumptions)

Represents a Cartesian Product of Sets.

Explanation

Returns a Cartesian product given several sets as either an iterable or individual arguments.

Can use * operator on any sets for convenient shorthand.

Examples

```
>>> from sympy import Interval, FiniteSet, ProductSet
>>> I = Interval(0, 5); S = FiniteSet(1, 2, 3)
>>> ProductSet(I, S)
ProductSet(Interval(0, 5), {1, 2, 3})
```

```
>>> (2, 2) in ProductSet(I, S)
True
```

```
>>> Interval(0, 1) * Interval(0, 1) # The unit square
ProductSet(Interval(0, 1), Interval(0, 1))
```

```
>>> coin = FiniteSet('H', 'T')
>>> set(coin**2)
{(H, H), (H, T), (T, H), (T, T)}
```

The Cartesian product is not commutative or associative e.g.:

```
>>> I*S == S*I
False
>>> (I*I)*I == I*(I*I)
False
```

Notes

- Passes most operations down to the argument sets

References

[R751]

property `is_iterable`

A property method which tests whether a set is iterable or not. Returns True if set is iterable, otherwise returns False.

Examples

```
>>> from sympy import FiniteSet, Interval
>>> I = Interval(0, 1)
>>> A = FiniteSet(1, 2, 3, 4, 5)
>>> I.is_iterable
False
>>> A.is_iterable
True
```

Complement

class `sympy.sets.sets.Complement(a, b, evaluate=True)`

Represents the set difference or relative complement of a set with another set.

$$A - B = \{x \in A \mid x \notin B\}$$

Examples

```
>>> from sympy import Complement, FiniteSet
>>> Complement(FiniteSet(0, 1, 2), FiniteSet(1))
{0, 2}
```

See also:

[*Intersection*](#) (page 1198), [*Union*](#) (page 1197)

References

[R752]

as_relational(*symbol*)

Rewrite a complement in terms of equalities and logic operators

static reduce(*A, B*)

Simplify a [Complement](#) (page 1200).

SymmetricDifference

class sympy.sets.sets.**SymmetricDifference**(*a, b, evaluate=True*)

Represents the set of elements which are in either of the sets and not in their intersection.

Examples

```
>>> from sympy import SymmetricDifference, FiniteSet
>>> SymmetricDifference(FiniteSet(1, 2, 3), FiniteSet(3, 4, 5))
{1, 2, 4, 5}
```

See also:

[Complement](#) (page 1200), [Union](#) (page 1197)

References

[R753]

as_relational(*symbol*)

Rewrite a symmetric_difference in terms of equalities and logic operators

DisjointUnion

class sympy.sets.sets.**DisjointUnion**(**sets*)

Represents the disjoint union (also known as the external disjoint union) of a finite number of sets.

Examples

```
>>> from sympy import DisjointUnion, FiniteSet, Interval, Union, Symbol
>>> A = FiniteSet(1, 2, 3)
>>> B = Interval(0, 5)
>>> DisjointUnion(A, B)
DisjointUnion({1, 2, 3}, Interval(0, 5))
>>> DisjointUnion(A, B).rewrite(Union)
Union(ProductSet({1, 2, 3}, {0}), ProductSet(Interval(0, 5), {1}))
```

(continues on next page)

(continued from previous page)

```
>>> C = FiniteSet(Symbol('x'), Symbol('y'), Symbol('z'))
>>> DisjointUnion(C, C)
DisjointUnion({x, y, z}, {x, y, z})
>>> DisjointUnion(C, C).rewrite(Union)
ProductSet({x, y, z}, {0, 1})
```

References

https://en.wikipedia.org/wiki/Disjoint_union

Singleton Sets

EmptySet

class sympy.sets.sets.EmptySet

Represents the empty set. The empty set is available as a singleton as `S.EmptySet`.

Examples

```
>>> from sympy import S, Interval
>>> S.EmptySet
EmptySet
```

```
>>> Interval(1, 2).intersect(S.EmptySet)
EmptySet
```

See also:

[*UniversalSet*](#) (page 1202)

References

[R754]

UniversalSet

class sympy.sets.sets.UniversalSet

Represents the set of all things. The universal set is available as a singleton as `S.UniversalSet`.

Examples

```
>>> from sympy import S, Interval
>>> S.UniversalSet
UniversalSet
```

```
>>> Interval(1, 2).intersect(S.UniversalSet)
Interval(1, 2)
```

See also:

[*EmptySet*](#) (page 1202)

References

[R755]

Special Sets

Rationals

class sympy.sets.fancysets.**Rationals**

Represents the rational numbers. This set is also available as the singleton `S.Rationals`.

Examples

```
>>> from sympy import S
>>> S.Half in S.Rationals
True
>>> iterable = iter(S.Rationals)
>>> [next(iterable) for i in range(12)]
[0, 1, -1, 1/2, 2, -1/2, -2, 1/3, 3, -1/3, -3, 2/3]
```

Naturals

class sympy.sets.fancysets.**Naturals**

Represents the natural numbers (or counting numbers) which are all positive integers starting from 1. This set is also available as the singleton `S.Naturals`.

Examples

```
>>> from sympy import S, Interval, pprint
>>> 5 in S.Naturals
True
>>> iterable = iter(S.Naturals)
>>> next(iterable)
1
>>> next(iterable)
2
>>> next(iterable)
3
>>> pprint(S.Naturals.intersect(Interval(0, 10)))
{1, 2, ..., 10}
```

See also:

Naturals0 (page 1204)

non-negative integers (i.e. includes 0, too)

Integers (page 1204)

also includes negative integers

Naturals0

class sympy.sets.fancysets.Naturals0

Represents the whole numbers which are all the non-negative integers, inclusive of zero.

See also:

Naturals (page 1203)

positive integers; does not include 0

Integers (page 1204)

also includes the negative integers

Integers

class sympy.sets.fancysets.Integers

Represents all integers: positive, negative and zero. This set is also available as the singleton `S.Integers`.

Examples

```
>>> from sympy import S, Interval, pprint
>>> 5 in S.Naturals
True
>>> iterable = iter(S.Integers)
>>> next(iterable)
0
>>> next(iterable)
1
>>> next(iterable)
-1
>>> next(iterable)
2
```

```
>>> pprint(S.Integers.intersect(Interval(-4, 4)))
{-4, -3, ..., 4}
```

See also:

[*Naturals0*](#) (page 1204)
non-negative integers

[*Integers*](#) (page 1204)
positive and negative integers and zero

Reals

`class sympy.sets.fancysets.Reals`

Represents all real numbers from negative infinity to positive infinity, including all integer, rational and irrational numbers. This set is also available as the singleton `S.Reals`.

Examples

```
>>> from sympy import S, Rational, pi, I
>>> 5 in S.Reals
True
>>> Rational(-1, 2) in S.Reals
True
>>> pi in S.Reals
True
>>> 3*I in S.Reals
False
>>> S.Reals.contains(pi)
True
```

See also:

[*ComplexRegion*](#) (page 1209)

Complexes

class sympy.sets.fancysets.Complexes

The Set of all complex numbers

Examples

```
>>> from sympy import S, I
>>> S.Complexes
Complexes
>>> 1 + I in S.Complexes
True
```

See also:

[Reals](#) (page 1205), [ComplexRegion](#) (page 1209)

ImageSet

class sympy.sets.fancysets.ImageSet(*flambda, *sets*)

Image of a set under a mathematical function. The transformation must be given as a Lambda function which has as many arguments as the elements of the set upon which it operates, e.g. 1 argument when acting on the set of integers or 2 arguments when acting on a complex region.

This function is not normally called directly, but is called from `imageset`.

Examples

```
>>> from sympy import Symbol, S, pi, Dummy, Lambda
>>> from sympy import FiniteSet, ImageSet, Interval
```

```
>>> x = Symbol('x')
>>> N = S.Naturals
>>> squares = ImageSet(Lambda(x, x**2), N) # {x**2 for x in N}
>>> 4 in squares
True
>>> 5 in squares
False
```

```
>>> FiniteSet(0, 1, 2, 3, 4, 5, 6, 7, 9, 10).intersect(squares)
{1, 4, 9}
```

```
>>> square_iterable = iter(squares)
>>> for i in range(4):
...     next(square_iterable)
1
4
```

(continues on next page)

(continued from previous page)

```
9
16
```

If you want to get value for $x = 2$, $1/2$ etc. (Please check whether the x value is in `base_set` or not before passing it as args)

```
>>> squares.lamda(2)
4
>>> squares.lamda(S(1)/2)
1/4
```

```
>>> n = Dummy('n')
>>> solutions = ImageSet(Lambda(n, n*pi), S.Integers) # solutions of
↳ sin(x) = 0
>>> dom = Interval(-1, 1)
>>> dom.intersect(solutions)
{0}
```

See also:

[`sympy.sets.sets.imageset`](#) (page 1193)

Range

class `sympy.sets.fancysets.Range(*args)`

Represents a range of integers. Can be called as `Range(stop)`, `Range(start, stop)`, or `Range(start, stop, step)`; when `step` is not given it defaults to 1.

`Range(stop)` is the same as `Range(0, stop, 1)` and the stop value (just as for Python ranges) is not included in the Range values.

```
>>> from sympy import Range
>>> list(Range(3))
[0, 1, 2]
```

The step can also be negative:

```
>>> list(Range(10, 0, -2))
[10, 8, 6, 4, 2]
```

The stop value is made canonical so equivalent ranges always have the same args:

```
>>> Range(0, 10, 3)
Range(0, 12, 3)
```

Infinite ranges are allowed. ∞ and $-\infty$ are never included in the set (Range is always a subset of Integers). If the starting point is infinite, then the final value is `stop - step`. To iterate such a range, it needs to be reversed:

```
>>> from sympy import oo
>>> r = Range(-oo, 1)
>>> r[-1]
```

(continues on next page)

(continued from previous page)

```
0
>>> next(iter(r))
Traceback (most recent call last):
...
TypeError: Cannot iterate over Range with infinite start
>>> next(iter(r.reversed))
0
```

Although Range is a Set (and supports the normal set operations) it maintains the order of the elements and can be used in contexts where range would be used.

```
>>> from sympy import Interval
>>> Range(0, 10, 2).intersect(Interval(3, 7))
Range(4, 8, 2)
>>> list(_)
[4, 6]
```

Although slicing of a Range will always return a Range - possibly empty - an empty set will be returned from any intersection that is empty:

```
>>> Range(3)[:0]
Range(0, 0, 1)
>>> Range(3).intersect(Interval(4, oo))
EmptySet
>>> Range(3).intersect(Range(4, oo))
EmptySet
```

Range will accept symbolic arguments but has very limited support for doing anything other than displaying the Range:

```
>>> from sympy import Symbol, pprint
>>> from sympy.abc import i, j, k
>>> Range(i, j, k).start
i
>>> Range(i, j, k).inf
Traceback (most recent call last):
...
ValueError: invalid method for symbolic range
```

Better success will be had when using integer symbols:

```
>>> n = Symbol('n', integer=True)
>>> r = Range(n, n + 20, 3)
>>> r.inf
n
>>> pprint(r)
{n, n + 3, ..., n + 18}
```

as_relational(x)

Rewrite a Range in terms of equalities and logic operators.

property reversed

Return an equivalent Range in the opposite order.

Examples

```
>>> from sympy import Range
>>> Range(10).reversed
Range(9, -1, -1)
```

ComplexRegion

class sympy.sets.fancysets.**ComplexRegion**(sets, polar=False)

Represents the Set of all Complex Numbers. It can represent a region of Complex Plane in both the standard forms Polar and Rectangular coordinates.

- Polar Form Input is in the form of the ProductSet or Union of ProductSets of the intervals of r and theta, and use the flag polar=True.

$$Z = \{z \in \mathbb{C} \mid z = r \times (\cos(\theta) + I \sin(\theta)), r \in [r], \theta \in [\text{theta}]\}$$

- Rectangular Form Input is in the form of the ProductSet or Union of ProductSets of interval of x and y, the real and imaginary parts of the Complex numbers in a plane. Default input type is in rectangular form.

$$Z = \{z \in \mathbb{C} \mid z = x + Iy, x \in [\text{re}(z)], y \in [\text{im}(z)]\}$$

Examples

```
>>> from sympy import ComplexRegion, Interval, S, I, Union
>>> a = Interval(2, 3)
>>> b = Interval(4, 6)
>>> c1 = ComplexRegion(a*b) # Rectangular Form
>>> c1
CartesianComplexRegion(ProductSet(Interval(2, 3), Interval(4, 6)))
```

- c1 represents the rectangular region in complex plane surrounded by the coordinates (2, 4), (3, 4), (3, 6) and (2, 6), of the four vertices.

```
>>> c = Interval(1, 8)
>>> c2 = ComplexRegion(Union(a*b, b*c))
>>> c2
CartesianComplexRegion(Union(ProductSet(Interval(2, 3), Interval(4, 6)),
↳ ProductSet(Interval(4, 6), Interval(1, 8))))
```

- c2 represents the Union of two rectangular regions in complex plane. One of them surrounded by the coordinates of c1 and other surrounded by the coordinates (4, 1), (6, 1), (6, 8) and (4, 8).

```
>>> 2.5 + 4.5*I in c1
True
>>> 2.5 + 6.5*I in c1
False
```

```
>>> r = Interval(0, 1)
>>> theta = Interval(0, 2*S.Pi)
>>> c2 = ComplexRegion(r*theta, polar=True) # Polar Form
>>> c2 # unit Disk
PolarComplexRegion(ProductSet(Interval(0, 1), Interval.Ropen(0, 2*pi)))
```

- c2 represents the region in complex plane inside the Unit Disk centered at the origin.

```
>>> 0.5 + 0.5*I in c2
True
>>> 1 + 2*I in c2
False
```

```
>>> unit_disk = ComplexRegion(Interval(0, 1)*Interval(0, 2*S.Pi),
    ↪ polar=True)
>>> upper_half_unit_disk = ComplexRegion(Interval(0, 1)*Interval(0, S.
    ↪ Pi), polar=True)
>>> intersection = unit_disk.intersect(upper_half_unit_disk)
>>> intersection
PolarComplexRegion(ProductSet(Interval(0, 1), Interval(0, pi)))
>>> intersection == upper_half_unit_disk
True
```

See also:

[CartesianComplexRegion](#) (page 1212), [PolarComplexRegion](#) (page 1212), [Complexes](#) (page 1206)

property a_interval

Return the union of intervals of x when, self is in rectangular form, or the union of intervals of r when self is in polar form.

Examples

```
>>> from sympy import Interval, ComplexRegion, Union
>>> a = Interval(2, 3)
>>> b = Interval(4, 5)
>>> c = Interval(1, 7)
>>> C1 = ComplexRegion(a*b)
>>> C1.a_interval
Interval(2, 3)
>>> C2 = ComplexRegion(Union(a*b, b*c))
>>> C2.a_interval
Union(Interval(2, 3), Interval(4, 5))
```

property b_interval

Return the union of intervals of y when, self is in rectangular form, or the union of intervals of θ when self is in polar form.

Examples

```
>>> from sympy import Interval, ComplexRegion, Union
>>> a = Interval(2, 3)
>>> b = Interval(4, 5)
>>> c = Interval(1, 7)
>>> C1 = ComplexRegion(a*b)
>>> C1.b_interval
Interval(4, 5)
>>> C2 = ComplexRegion(Union(a*b, b*c))
>>> C2.b_interval
Interval(1, 7)
```

classmethod `from_real(sets)`

Converts given subset of real numbers to a complex region.

Examples

```
>>> from sympy import Interval, ComplexRegion
>>> unit = Interval(0,1)
>>> ComplexRegion.from_real(unit)
CartesianComplexRegion(ProductSet(Interval(0, 1), {0}))
```

property `psets`

Return a tuple of sets (ProductSets) input of the self.

Examples

```
>>> from sympy import Interval, ComplexRegion, Union
>>> a = Interval(2, 3)
>>> b = Interval(4, 5)
>>> c = Interval(1, 7)
>>> C1 = ComplexRegion(a*b)
>>> C1.psets
(ProductSet(Interval(2, 3), Interval(4, 5)),)
>>> C2 = ComplexRegion(Union(a*b, b*c))
>>> C2.psets
(ProductSet(Interval(2, 3), Interval(4, 5)), ProductSet(Interval(4, 5), Interval(1, 7)))
```

property `sets`

Return raw input sets to the self.

Examples

```
>>> from sympy import Interval, ComplexRegion, Union
>>> a = Interval(2, 3)
>>> b = Interval(4, 5)
>>> c = Interval(1, 7)
>>> C1 = ComplexRegion(a*b)
>>> C1.sets
ProductSet(Interval(2, 3), Interval(4, 5))
>>> C2 = ComplexRegion(Union(a*b, b*c))
>>> C2.sets
Union(ProductSet(Interval(2, 3), Interval(4, 5)),
      ProductSet(Interval(4, 5), Interval(1, 7)))
```

class sympy.sets.fancysets.**CartesianComplexRegion**(sets)

Set representing a square region of the complex plane.

$$Z = \{z \in \mathbb{C} \mid z = x + Iy, x \in [\operatorname{re}(z)], y \in [\operatorname{im}(z)]\}$$

Examples

```
>>> from sympy import ComplexRegion, I, Interval
>>> region = ComplexRegion(Interval(1, 3) * Interval(4, 6))
>>> 2 + 5*I in region
True
>>> 5*I in region
False
```

See also:

[ComplexRegion](#) (page 1209), [PolarComplexRegion](#) (page 1212), [Complexes](#) (page 1206)

class sympy.sets.fancysets.**PolarComplexRegion**(sets)

Set representing a polar region of the complex plane.

$$Z = \{z \in \mathbb{C} \mid z = r \times (\cos(\theta) + I \sin(\theta)), r \in [r], \theta \in [\theta]\}$$

Examples

```
>>> from sympy import ComplexRegion, Interval, oo, pi, I
>>> rset = Interval(0, oo)
>>> thetaset = Interval(0, pi)
>>> upper_half_plane = ComplexRegion(rset * thetaset, polar=True)
>>> 1 + I in upper_half_plane
True
>>> 1 - I in upper_half_plane
False
```

See also:

[ComplexRegion](#) (page 1209), [CartesianComplexRegion](#) (page 1212), [Complexes](#) (page 1206)

`sympy.sets.fancysets.normalize_theta_set(theta)`

Normalize a Real Set θ in the interval $[0, 2\pi)$. It returns a normalized value of θ in the Set. For Interval, a maximum of one cycle $[0, 2\pi]$, is returned i.e. for θ equal to $[0, 10\pi]$, returned normalized value would be $[0, 2\pi)$. As of now intervals with end points as non-multiples of π is not supported.

Raises

NotImplementedError

The algorithms for Normalizing θ Set are not yet implemented.

ValueError

The input is not valid, i.e. the input is not a real set.

RuntimeError

It is a bug, please report to the github issue tracker.

Examples

```
>>> from sympy.sets.fancysets import normalize_theta_set
>>> from sympy import Interval, FiniteSet, pi
>>> normalize_theta_set(Interval(9*pi/2, 5*pi))
Interval(pi/2, pi)
>>> normalize_theta_set(Interval(-3*pi/2, pi/2))
Interval.Ropen(0, 2*pi)
>>> normalize_theta_set(Interval(-pi/2, pi/2))
Union(Interval(0, pi/2), Interval.Ropen(3*pi/2, 2*pi))
>>> normalize_theta_set(Interval(-4*pi, 3*pi))
Interval.Ropen(0, 2*pi)
>>> normalize_theta_set(Interval(-3*pi/2, -pi/2))
Interval(pi/2, 3*pi/2)
>>> normalize_theta_set(FiniteSet(0, pi, 3*pi))
{0, pi}
```

Power sets

PowerSet

class `sympy.sets.powerset.PowerSet(arg, evaluate=None)`

A symbolic object representing a power set.

Parameters

arg : Set

The set to take power of.

evaluate : bool

The flag to control evaluation.

If the evaluation is disabled for finite sets, it can take advantage of using subset test as a membership test.

Notes

Power set $\mathcal{P}(S)$ is defined as a set containing all the subsets of S .

If the set S is a finite set, its power set would have $2^{|S|}$ elements, where $|S|$ denotes the cardinality of S .

Examples

```
>>> from sympy import PowerSet, S, FiniteSet
```

A power set of a finite set:

```
>>> PowerSet(FiniteSet(1, 2, 3))
PowerSet({1, 2, 3})
```

A power set of an empty set:

```
>>> PowerSet(S.EmptySet)
PowerSet(EmptySet)
>>> PowerSet(PowerSet(S.EmptySet))
PowerSet(PowerSet(EmptySet))
```

A power set of an infinite set:

```
>>> PowerSet(S.Reals)
PowerSet(Reals)
```

Evaluating the power set of a finite set to its explicit form:

```
>>> PowerSet(FiniteSet(1, 2, 3)).rewrite(FiniteSet)
FiniteSet(EmptySet, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3})
```

References

[R756], [R757]

Iteration over sets

For set unions, $\{a, b\} \cup \{x, y\}$ can be treated as $\{a, b, x, y\}$ for iteration regardless of the distinctiveness of the elements, however, for set intersections, assuming that $\{a, b\} \cap \{x, y\}$ is $\emptyset$ or $\{a, b\}$ would not always be valid, since some of a , b , x or y may or may not be the elements of the intersection.

Iterating over the elements of a set involving intersection, complement, or symmetric difference yields (possibly duplicate) elements of the set provided that all elements are known to be the elements of the set. If any element cannot be determined to be a member of a set then the iteration gives `TypeError`. This happens in the same cases where `x in y` would give an error.

There are some reasons to implement like this, even if it breaks the consistency with how the python set iterator works. We keep in mind that sympy set comprehension like

`FiniteSet(*s)` from an existing sympy set could be a common usage. And this approach would make `FiniteSet(*s)` to be consistent with any symbolic set processing methods like `FiniteSet(*simplify(s))`.

Condition Sets

ConditionSet

class `sympy.sets.conditionset.ConditionSet(sym, condition, base_set=UniversalSet)`

Set of elements which satisfies a given condition.

$$\{x \mid \text{condition}(x) = \text{True}, x \in S\}$$

Examples

```
>>> from sympy import Symbol, S, ConditionSet, pi, Eq, sin, Interval
>>> from sympy.abc import x, y, z
```

```
>>> sin_sols = ConditionSet(x, Eq(sin(x), 0), Interval(0, 2*pi))
>>> 2*pi in sin_sols
True
>>> pi/2 in sin_sols
False
>>> 3*pi in sin_sols
False
>>> 5 in ConditionSet(x, x**2 > 4, S.Reals)
True
```

If the value is not in the base set, the result is false:

```
>>> 5 in ConditionSet(x, x**2 > 4, Interval(2, 4))
False
```

Notes

Symbols with assumptions should be avoided or else the condition may evaluate without consideration of the set:

```
>>> n = Symbol('n', negative=True)
>>> cond = (n > 0); cond
False
>>> ConditionSet(n, cond, S.Integers)
EmptySet
```

Only free symbols can be changed by using `subs`:

```
>>> c = ConditionSet(x, x < 1, {x, z})
>>> c.subs(x, y)
ConditionSet(x, x < 1, {y, z})
```

To check if π is in c use:

```
>>> pi in c
False
```

If no base set is specified, the universal set is implied:

```
>>> ConditionSet(x, x < 1).base_set
UniversalSet
```

Only symbols or symbol-like expressions can be used:

```
>>> ConditionSet(x + 1, x + 1 < 1, S.Integers)
Traceback (most recent call last):
...
ValueError: non-symbol dummy not recognized in condition
```

When the base set is a `ConditionSet`, the symbols will be unified if possible with preference for the outermost symbols:

```
>>> ConditionSet(x, x < y, ConditionSet(z, z + y < 2, S.Integers))
ConditionSet(x, (x < y) & (x + y < 2), Integers)
```

Relations on sets

class `sympy.sets.conditionset.Contains(x, s)`
Asserts that x is an element of the set S .

Examples

```
>>> from sympy import Symbol, Integer, S, Contains
>>> Contains(Integer(2), S.Integers)
True
>>> Contains(Integer(-2), S.Naturals)
False
>>> i = Symbol('i', integer=True)
>>> Contains(i, S.Naturals)
Contains(i, Naturals)
```

References

[R758]

SetKind

class `sympy.sets.conditionset.SetKind(element_kind=None)`

SetKind is kind for all Sets

Every instance of Set will have kind SetKind parametrised by the kind of the elements of the Set. The kind of the elements might be NumberKind, or TupleKind or something else. When not all elements have the same kind then the kind of the elements will be given as UndefinedKind.

Parameters

element_kind: Kind (optional)

The kind of the elements of the set. In a well defined set all elements will have the same kind. Otherwise the kind should [sympy.core.kind.UndefinedKind](#) (page 1074). The `element_kind` argument is optional but should only be omitted in the case of EmptySet whose kind is simply SetKind()

Examples

```
>>> from sympy import Interval
>>> Interval(1, 2).kind
SetKind(NumberKind)
>>> Interval(1,2).kind.element_kind
NumberKind
```

See also:

[sympy.core.kind.NumberKind](#) (page 1074), [sympy.matrices.common.MatrixKind](#) (page 1360), [sympy.core.containers.TupleKind](#) (page 1069)

5.8.4 Matrices

Contents

Matrices

A module that handles matrices.

Includes functions for fast creating matrices like zero, one/eye, random matrix, etc.

Contents:

Matrices (linear algebra)

Creating Matrices

The linear algebra module is designed to be as simple as possible. First, we import and declare our first Matrix object:

```
>>> from sympy.interactive.printing import init_printing
>>> init_printing(use_unicode=False, wrap_line=False)
>>> from sympy.matrices import Matrix, eye, zeros, ones, diag, GramSchmidt
>>> M = Matrix([[1,0,0], [0,0,0]]); M
[1  0  0]
[
 ]
[0  0  0]
>>> Matrix([M, (0, 0, -1)])
[1  0  0 ]
[
 ]
[0  0  0 ]
[
 ]
[0  0 -1]
>>> Matrix([[1, 2, 3]])
[1 2 3]
>>> Matrix([1, 2, 3])
[1]
[ ]
[2]
[ ]
[3]
```

In addition to creating a matrix from a list of appropriately-sized lists and/or matrices, SymPy also supports more advanced methods of matrix creation including a single list of values and dimension inputs:

```
>>> Matrix(2, 3, [1, 2, 3, 4, 5, 6])
[1  2  3]
[
 ]
[4  5  6]
```

More interesting (and useful), is the ability to use a 2-variable function (or `lambda`) to create a matrix. Here we create an indicator function which is 1 on the diagonal and then use it to make the identity matrix:

```
>>> def f(i,j):
...     if i == j:
...         return 1
...     else:
...         return 0
...
>>> Matrix(4, 4, f)
[1  0  0  0]
[
 ]
[0  1  0  0]
[
 ]
```

(continues on next page)

(continued from previous page)

```
[0  0  1  0]
[      ]
[0  0  0  1]
```

Finally let's use `lambda` to create a 1-line matrix with 1's in the even permutation entries:

```
>>> Matrix(3, 4, lambda i,j: 1 - (i+j) % 2)
[1  0  1  0]
[      ]
[0  1  0  1]
[      ]
[1  0  1  0]
```

There are also a couple of special constructors for quick matrix construction: `eye` is the identity matrix, `zeros` and `ones` for matrices of all zeros and ones, respectively, and `diag` to put matrices or elements along the diagonal:

```
>>> eye(4)
[1  0  0  0]
[      ]
[0  1  0  0]
[      ]
[0  0  1  0]
[      ]
[0  0  0  1]
>>> zeros(2)
[0  0]
[      ]
[0  0]
>>> zeros(2, 5)
[0  0  0  0  0]
[      ]
[0  0  0  0  0]
>>> ones(3)
[1  1  1]
[      ]
[1  1  1]
[      ]
[1  1  1]
>>> ones(1, 3)
[1  1  1]
>>> diag(1, Matrix([[1, 2], [3, 4]]))
[1  0  0]
[      ]
[0  1  2]
[      ]
[0  3  4]
```

Basic Manipulation

While learning to work with matrices, let's choose one where the entries are readily identifiable. One useful thing to know is that while matrices are 2-dimensional, the storage is not and so it is allowable - though one should be careful - to access the entries as if they were a 1-d list.

```
>>> M = Matrix(2, 3, [1, 2, 3, 4, 5, 6])
>>> M[4]
5
```

Now, the more standard entry access is a pair of indices which will always return the value at the corresponding row and column of the matrix:

```
>>> M[1, 2]
6
>>> M[0, 0]
1
>>> M[1, 1]
5
```

Since this is Python we're also able to slice submatrices; slices always give a matrix in return, even if the dimension is 1 x 1:

```
>>> M[0:2, 0:2]
[1 2]
[ ]
[4 5]
>>> M[2:2, 2]
[]
>>> M[:, 2]
[3]
[ ]
[6]
>>> M[:1, 2]
[3]
```

In the second example above notice that the slice 2:2 gives an empty range. Note also (in keeping with 0-based indexing of Python) the first row/column is 0.

You cannot access rows or columns that are not present unless they are in a slice:

```
>>> M[:, 10] # the 10-th column (not there)
Traceback (most recent call last):
...
IndexError: Index out of range: a[[0, 10]]
>>> M[:, 10:11] # the 10-th column (if there)
[]
>>> M[:, :10] # all columns up to the 10-th
[1 2 3]
[ ]
[4 5 6]
```

Slicing an empty matrix works as long as you use a slice for the coordinate that has no size:

```
>>> Matrix(0, 3, [])[:, 1]
[]
```

Slicing gives a copy of what is sliced, so modifications of one object do not affect the other:

```
>>> M2 = M[:, :]
>>> M2[0, 0] = 100
>>> M[0, 0] == 100
False
```

Notice that changing M2 didn't change M. Since we can slice, we can also assign entries:

```
>>> M = Matrix(([1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]))
>>> M
[1  2  3  4 ]
[      ]
[5  6  7  8 ]
[      ]
[9  10 11 12]
[      ]
[13 14 15 16]
>>> M[2,2] = M[0,3] = 0
>>> M
[1  2  3  0 ]
[      ]
[5  6  7  8 ]
[      ]
[9  10 0  12]
[      ]
[13 14 15 16]
```

as well as assign slices:

```
>>> M = Matrix(([1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]))
>>> M[2:,2:] = Matrix(2,2,lambda i,j: 0)
>>> M
[1  2  3  4]
[      ]
[5  6  7  8]
[      ]
[9  10 0  0]
[      ]
[13 14 0  0]
```

All the standard arithmetic operations are supported:

```
>>> M = Matrix(([1,2,3],[4,5,6],[7,8,9]))
>>> M - M
[0  0  0]
[      ]
[0  0  0]
[      ]
[0  0  0]
```

(continues on next page)

(continued from previous page)

```

>>> M + M
[2  4  6 ]
[   ]
[8 10 12]
[   ]
[14 16 18]
>>> M * M
[30  36  42 ]
[   ]
[66  81  96 ]
[   ]
[102 126 150]
>>> M2 = Matrix(3,1,[1,5,0])
>>> M*M2
[11]
[  ]
[29]
[  ]
[47]
>>> M**2
[30  36  42 ]
[   ]
[66  81  96 ]
[   ]
[102 126 150]

```

As well as some useful vector operations:

```

>>> M.row_del(0)
>>> M
[4 5 6]
[   ]
[7 8 9]
>>> M.col_del(1)
>>> M
[4 6]
[   ]
[7 9]
>>> v1 = Matrix([1,2,3])
>>> v2 = Matrix([4,5,6])
>>> v3 = v1.cross(v2)
>>> v1.dot(v2)
32
>>> v2.dot(v3)
0
>>> v1.dot(v3)
0

```

Recall that the `row_del()` and `col_del()` operations don't return a value - they simply change the matrix object. We can also “glue” together matrices of the appropriate size:

```

>>> M1 = eye(3)

```

(continues on next page)

(continued from previous page)

```

>>> M2 = zeros(3, 4)
>>> M1.row_join(M2)
[1  0  0  0  0  0  0]
[
]
[0  1  0  0  0  0  0]
[
]
[0  0  1  0  0  0  0]
>>> M3 = zeros(4, 3)
>>> M1.col_join(M3)
[1  0  0]
[
]
[0  1  0]
[
]
[0  0  1]
[
]
[0  0  0]
[
]
[0  0  0]
[
]
[0  0  0]
[
]
[0  0  0]
[
]

```

Operations on entries

We are not restricted to having multiplication between two matrices:

```

>>> M = eye(3)
>>> 2*M
[2  0  0]
[
]
[0  2  0]
[
]
[0  0  2]
>>> 3*M
[3  0  0]
[
]
[0  3  0]
[
]
[0  0  3]

```

but we can also apply functions to our matrix entries using `applyfunc()`. Here we'll declare a function that double any input number. Then we apply it to the 3x3 identity matrix:

```

>>> f = lambda x: 2*x
>>> eye(3).applyfunc(f)
[2  0  0]
[
]
[0  2  0]
[
]
[0  0  2]

```

If you want to extract a common factor from a matrix you can do so by applying `gcd` to the data of the matrix:

```
>>> from sympy.abc import x, y
>>> from sympy import gcd
>>> m = Matrix([[x, y], [1, x*y]]).inv('ADJ'); m
[ x*y      -y      ]
[-----]
[ 2      2      ]
[x *y - y  x *y - y]
[      ]
[ -1      x      ]
[-----]
[ 2      2      ]
[x *y - y  x *y - y]
>>> gcd(tuple(_))
1
-----
2
x *y - y
>>> m/_
[x*y  -y]
[      ]
[-1   x ]
```

One more useful matrix-wide entry application function is the substitution function. Let's declare a matrix with symbolic entries then substitute a value. Remember we can substitute anything - even another symbol!:

```
>>> from sympy import Symbol
>>> x = Symbol('x')
>>> M = eye(3) * x
>>> M
[x  0  0]
[  ]
[0  x  0]
[  ]
[0  0  x]
>>> M.subs(x, 4)
[4  0  0]
[  ]
[0  4  0]
[  ]
[0  0  4]
>>> y = Symbol('y')
>>> M.subs(x, y)
[y  0  0]
[  ]
[0  y  0]
[  ]
[0  0  y]
```


Linear algebra

Now that we have the basics out of the way, let's see what we can do with the actual matrices. Of course, one of the first things that comes to mind is the determinant:

```
>>> M = Matrix(( [1, 2, 3], [3, 6, 2], [2, 0, 1] ))
>>> M.det()
-28
>>> M2 = eye(3)
>>> M2.det()
1
>>> M3 = Matrix(( [1, 0, 0], [1, 0, 0], [1, 0, 0] ))
>>> M3.det()
0
```

Another common operation is the inverse: In SymPy, this is computed by Gaussian elimination by default (for dense matrices) but we can specify it be done by *LU* decomposition as well:

```
>>> M2.inv()
[1  0  0]
[  0  1  0]
[0  0  1]
>>> M2.inv(method="LU")
[1  0  0]
[  0  1  0]
[0  0  1]
>>> M.inv(method="LU")
[-3/14  1/14  1/2 ]
[  0  0  0 ]
[-1/28  5/28  -1/4]
[  0  0  0 ]
[ 3/7   -1/7   0 ]
>>> M * M.inv(method="LU")
[1  0  0]
[  0  1  0]
[0  0  1]
```

We can perform a *QR* factorization which is handy for solving systems:

```
>>> A = Matrix([[1,1,1],[1,1,3],[2,3,4]])
>>> Q, R = A.QRdecomposition()
>>> Q
[  \sqrt{6}   -\sqrt{3}   -\sqrt{2} ]
[  -----  -----  ----- ]
[   6         3         2       ]
[  ]
```

(continues on next page)

(continued from previous page)

```

[ \sqrt{6}  -\sqrt{3}  \sqrt{2} ]
[ -----  -----  ----- ]
[ 6        3        2        ]
[ ]
[ ]
[ \sqrt{6}  \sqrt{3}  ]
[ -----  -----  0 ]
[ 3        3        ]
>>> R
[ ]
[ 4*\sqrt{6} ]
[ \sqrt{6}  -----  2*\sqrt{6} ]
[ 3 ]
[ ]
[ ]
[ \sqrt{3} ]
[ 0  -----  0 ]
[ 3 ]
[ ]
[ ]
[ 0  0  \sqrt{2} ]
>>> Q*R
[1  1  1]
[ ]
[1  1  3]
[ ]
[2  3  4]

```

In addition to the solvers in the `solver.py` file, we can solve the system $Ax=b$ by passing the `b` vector to the matrix `A`'s `LUsolve` function. Here we'll cheat a little choose `A` and `x` then multiply to get `b`. Then we can solve for `x` and check that it's correct:

```

>>> A = Matrix([ [2, 3, 5], [3, 6, 2], [8, 3, 6] ])
>>> x = Matrix(3,1,[3,7,5])
>>> b = A*x
>>> soln = A.LUsolve(b)
>>> soln
[3]
[ ]
[7]
[ ]
[5]

```

There's also a nice Gram-Schmidt orthogonalizer which will take a set of vectors and orthogonalize them with respect to another. There is an optional argument which specifies whether or not the output should also be normalized, it defaults to `False`. Let's take some vectors and orthogonalize them - one normalized and one not:

```

>>> L = [Matrix([2,3,5]), Matrix([3,6,2]), Matrix([8,3,6])]
>>> out1 = GramSchmidt(L)
>>> out2 = GramSchmidt(L, True)

```

Let's take a look at the vectors:

```
>>> for i in out1:
...     print(i)
...
Matrix([[2], [3], [5]])
Matrix([[23/19], [63/19], [-47/19]])
Matrix([[1692/353], [-1551/706], [-423/706]])
>>> for i in out2:
...     print(i)
...
Matrix([[sqrt(38)/19], [3*sqrt(38)/38], [5*sqrt(38)/38]])
Matrix([[23*sqrt(6707)/6707], [63*sqrt(6707)/6707], [-47*sqrt(6707)/6707]])
Matrix([[12*sqrt(706)/353], [-11*sqrt(706)/706], [-3*sqrt(706)/706]])
```

We can spot-check their orthogonality with `dot()` and their normality with `norm()`:

```
>>> out1[0].dot(out1[1])
0
>>> out1[0].dot(out1[2])
0
>>> out1[1].dot(out1[2])
0
>>> out2[0].norm()
1
>>> out2[1].norm()
1
>>> out2[2].norm()
1
```

So there is quite a bit that can be done with the module including eigenvalues, eigenvectors, nullspace calculation, cofactor expansion tools, and so on. From here one might want to look over the `matrices.py` file for all functionality.

MatrixDeterminant Class Reference

class `sympy.matrices.matrices.MatrixDeterminant`

Provides basic matrix determinant operations. Should not be instantiated directly. See `determinant.py` for their implementations.

adjugate(*method*='berkowitz')

Returns the adjugate, or classical adjoint, of a matrix. That is, the transpose of the matrix of cofactors.

<https://en.wikipedia.org/wiki/Adjugate>

Parameters

method : string, optional

Method to use to find the cofactors, can be "bareiss", "berkowitz" or "lu".

Examples

```
>>> from sympy import Matrix
>>> M = Matrix([[1, 2], [3, 4]])
>>> M.adjugate()
Matrix([
 [ 4, -2],
 [-3,  1]])
```

See also:

[`cofactor\_matrix`](#) (page 1229), [`sympy.matrices.common.MatrixCommon.transpose`](#) (page 1355)

charpoly(*x*='lambda', *simplify*=<function _simplify>)

Computes characteristic polynomial $\det(xI - M)$ where I is the identity matrix.

A `PurePoly` is returned, so using different variables for x does not affect the comparison or the polynomials:

Parameters

x : string, optional

Name for the “lambda” variable, defaults to “lambda”.

simplify : function, optional

Simplification function to use on the characteristic polynomial calculated. Defaults to `simplify`.

Examples

```
>>> from sympy import Matrix
>>> from sympy.abc import x, y
>>> M = Matrix([[1, 3], [2, 0]])
>>> M.charpoly()
PurePoly(lambda**2 - lambda - 6, lambda, domain='ZZ')
>>> M.charpoly(x) == M.charpoly(y)
True
>>> M.charpoly(x) == M.charpoly(y)
True
```

Specifying x is optional; a symbol named `lambda` is used by default (which looks good when pretty-printed in unicode):

```
>>> M.charpoly().as_expr()
lambda**2 - lambda - 6
```

And if x clashes with an existing symbol, underscores will be prepended to the name to make it unique:

```
>>> M = Matrix([[1, 2], [x, 0]])
>>> M.charpoly(x).as_expr()
_x**2 - _x - 2*x
```

Whether you pass a symbol or not, the generator can be obtained with the `gen` attribute since it may not be the same as the symbol that was passed:

```
>>> M.charpoly(x).gen
_x
>>> M.charpoly(x).gen == x
False
```

Notes

The Samuelson-Berkowitz algorithm is used to compute the characteristic polynomial efficiently and without any division operations. Thus the characteristic polynomial over any commutative ring without zero divisors can be computed.

If the determinant $\det(xI - M)$ can be found out easily as in the case of an upper or a lower triangular matrix, then instead of Samuelson-Berkowitz algorithm, eigenvalues are computed and the characteristic polynomial with their help.

See also:

[`det`](#) (page 1230)

cofactor(*i, j, method='berkowitz'*)

Calculate the cofactor of an element.

Parameters

method : string, optional

Method to use to find the cofactors, can be “bareiss”, “berkowitz” or “lu”.

Examples

```
>>> from sympy import Matrix
>>> M = Matrix([[1, 2], [3, 4]])
>>> M.cofactor(0, 1)
-3
```

See also:

[`cofactor\_matrix`](#) (page 1229), [`minor`](#) (page 1231), [`minor\_submatrix`](#) (page 1231)

cofactor_matrix(*method='berkowitz'*)

Return a matrix containing the cofactor of each element.

Parameters

method : string, optional

Method to use to find the cofactors, can be “bareiss”, “berkowitz” or “lu”.

Examples

```
>>> from sympy import Matrix
>>> M = Matrix([[1, 2], [3, 4]])
>>> M.cofactor_matrix()
Matrix([
 [ 4, -3],
 [-2,  1]])
```

See also:

[`cofactor`](#) (page 1229), [`minor`](#) (page 1231), [`minor\_submatrix`](#) (page 1231)

det (*method*='bareiss', *iszerofunc*=None)

Computes the determinant of a matrix if *M* is a concrete matrix object otherwise return an expressions Determinant(*M*) if *M* is a MatrixSymbol or other expression.

Parameters

method : string, optional

Specifies the algorithm used for computing the matrix determinant.

If the matrix is at most 3x3, a hard-coded formula is used and the specified method is ignored. Otherwise, it defaults to 'bareiss'.

Also, if the matrix is an upper or a lower triangular matrix, determinant is computed by simple multiplication of diagonal elements, and the specified method is ignored.

If it is set to 'domain-ge', then Gaussian elimination method will be used via using DomainMatrix.

If it is set to 'bareiss', Bareiss' fraction-free algorithm will be used.

If it is set to 'berkowitz', Berkowitz' algorithm will be used.

Otherwise, if it is set to 'lu', LU decomposition will be used.

Note: For backward compatibility, legacy keys like "bareis" and "det_lu" can still be used to indicate the corresponding methods. And the keys are also case-insensitive for now. However, it is suggested to use the precise keys for specifying the method.

iszerofunc : FunctionType or None, optional

If it is set to None, it will be defaulted to `_iszero` if the method is set to 'bareiss', and `_is_zero_after_expand_mul` if the method is set to 'lu'.

It can also accept any user-specified zero testing function, if it is formatted as a function which accepts a single symbolic argument and returns True if it is tested as zero and False if it tested as non-zero, and also None if it is undecidable.

Returns

det : Basic

Result of determinant.

Raises**ValueError**

If unrecognized keys are given for method or iszerofunc.

NonSquareMatrixError

If attempted to calculate determinant from a non-square matrix.

Examples

```
>>> from sympy import Matrix, eye, det
>>> I3 = eye(3)
>>> det(I3)
1
>>> M = Matrix([[1, 2], [3, 4]])
>>> det(M)
-2
>>> det(M) == M.det()
True
>>> M.det(method="domain-ge")
-2
```

minor(*i*, *j*, method='berkowitz')

Return the (*i*,*j*) minor of *M*. That is, return the determinant of the matrix obtained by deleting the *i*'th row and *j*'th column from "*M*".

Parameters

i, j : int

The row and column to exclude to obtain the submatrix.

method : string, optional

Method to use to find the determinant of the submatrix, can be "bareiss", "berkowitz" or "lu".

Examples

```
>>> from sympy import Matrix
>>> M = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> M.minor(1, 1)
-12
```

See also:

[*minor_submatrix*](#) (page 1231), [*cofactor*](#) (page 1229), [*det*](#) (page 1230)

minor_submatrix(*i*, *j*)

Return the submatrix obtained by removing the *i*'th row and *j*'th column from "*M*" (works with Pythonic negative indices).

Parameters

i, j : int

The row and column to exclude to obtain the submatrix.

Examples

```
>>> from sympy import Matrix
>>> M = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> M.minor_submatrix(1, 1)
Matrix([
[1, 3],
[7, 9]])
```

See also:

[*minor*](#) (page 1231), [*cofactor*](#) (page 1229)

`per()`

Returns the permanent of a matrix. Unlike determinant, permanent is defined for both square and non-square matrices.

For an $m \times n$ matrix, with m less than or equal to n , it is given as the sum over the permutations s of size less than or equal to m on $[1, 2, \dots, n]$ of the product from $i = 1$ to m of $M[i, s[i]]$. Taking the transpose will not affect the value of the permanent.

In the case of a square matrix, this is the same as the permutation definition of the determinant, but it does not take the sign of the permutation into account. Computing the permanent with this definition is quite inefficient, so here the Ryser formula is used.

Examples

```
>>> from sympy import Matrix
>>> M = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> M.per()
450
>>> M = Matrix([1, 5, 7])
>>> M.per()
13
```

References

[R566], [R567], [R568], [R569]

MatrixReductions Class Reference

`class sympy.matrices.matrices.MatrixReductions`

Provides basic matrix row/column operations. Should not be instantiated directly. See `reductions.py` for some of their implementations.

`echelon_form`(*iszerofunc*=<function `_iszero`>, *simplify*=False, *with_pivots*=False)

Returns a matrix row-equivalent to M that is in echelon form. Note that echelon form of a matrix is *not* unique, however, properties like the row space and the null space are preserved.

Examples

```
>>> from sympy import Matrix
>>> M = Matrix([[1, 2], [3, 4]])
>>> M.echelon_form()
Matrix([
[1,  2],
[0, -2]])
```

elementary_col_op(*op*='n->kn', *col*=None, *k*=None, *col1*=None, *col2*=None)

Performs the elementary column operation *op*.

op may be one of

- "n->kn" (column n goes to k*n)
- "n<->m" (swap column n and column m)
- "n->n+km" (column n goes to column n + k*column m)

Parameters

op : string; the elementary row operation
col : the column to apply the column operation
k : the multiple to apply in the column operation
col1 : one column of a column swap
col2 : second column of a column swap or column "m" in the column operation
 "n->n+km"

elementary_row_op(*op*='n->kn', *row*=None, *k*=None, *row1*=None, *row2*=None)

Performs the elementary row operation *op*.

op may be one of

- "n->kn" (row n goes to k*n)
- "n<->m" (swap row n and row m)
- "n->n+km" (row n goes to row n + k*row m)

Parameters

op : string; the elementary row operation
row : the row to apply the row operation
k : the multiple to apply in the row operation
row1 : one row of a row swap
row2 : second row of a row swap or row "m" in the row operation
 "n->n+km"

property is_echelon

Returns *True* if the matrix is in echelon form. That is, all rows of zeros are at the bottom, and below each leading non-zero in a row are exclusively zeros.

rank(iszerofunc=<function _iszero>, simplify=False)

Returns the rank of a matrix.

Examples

```
>>> from sympy import Matrix
>>> from sympy.abc import x
>>> m = Matrix([[1, 2], [x, 1 - 1/x]])
>>> m.rank()
2
>>> n = Matrix(3, 3, range(1, 10))
>>> n.rank()
2
```

rref(iszerofunc=<function _iszero>, simplify=False, pivots=True, normalize_last=True)

Return reduced row-echelon form of matrix and indices of pivot vars.

Parameters

iszerofunc : Function

A function used for detecting whether an element can act as a pivot. `lambda x: x.is_zero` is used by default.

simplify : Function

A function used to simplify elements when looking for a pivot. By default SymPy's `simplify` is used.

pivots : True or False

If True, a tuple containing the row-reduced matrix and a tuple of pivot columns is returned. If False just the row-reduced matrix is returned.

normalize_last : True or False

If True, no pivots are normalized to 1 until after all entries above and below each pivot are zeroed. This means the row reduction algorithm is fraction free until the very last step. If False, the naive row reduction procedure is used where each pivot is normalized to be 1 before row operations are used to zero above and below the pivot.

Examples

```
>>> from sympy import Matrix
>>> from sympy.abc import x
>>> m = Matrix([[1, 2], [x, 1 - 1/x]])
>>> m.rref()
(Matrix([
[1, 0],
[0, 1]]), (0, 1))
>>> rref_matrix, rref_pivots = m.rref()
>>> rref_matrix
```

(continues on next page)

(continued from previous page)

```
Matrix([
[1, 0],
[0, 1]])
>>> rref_pivots
(0, 1)
```

`iszerofunc` can correct rounding errors in matrices with float values. In the following example, calling `rref()` leads to floating point errors, incorrectly row reducing the matrix. `iszerofunc= lambda x: abs(x)<1e-9` sets sufficiently small numbers to zero, avoiding this error.

```
>>> m = Matrix([[0.9, -0.1, -0.2, 0], [-0.8, 0.9, -0.4, 0], [-0.1, -0.
↪8, 0.6, 0]])
>>> m.rref()
(Matrix([
[1, 0, 0, 0],
[0, 1, 0, 0],
[0, 0, 1, 0]]), (0, 1, 2))
>>> m.rref(iszerofunc=lambda x:abs(x)<1e-9)
(Matrix([
[1, 0, -0.301369863013699, 0],
[0, 1, -0.712328767123288, 0],
[0, 0, 0, 0]]), (0, 1))
```

Notes

The default value of `normalize_last=True` can provide significant speedup to row reduction, especially on matrices with symbols. However, if you depend on the form row reduction algorithm leaves entries of the matrix, set `normalize_last=False`

MatrixSubspaces Class Reference

`class sympy.matrices.matrices.MatrixSubspaces`

Provides methods relating to the fundamental subspaces of a matrix. Should not be instantiated directly. See `subspaces.py` for their implementations.

`columnspace(simplify=False)`

Returns a list of vectors (Matrix objects) that span columnspace of M

Examples

```
>>> from sympy import Matrix
>>> M = Matrix(3, 3, [1, 3, 0, -2, -6, 0, 3, 9, 6])
>>> M
Matrix([
[ 1,  3, 0],
[-2, -6, 0],
[ 3,  9, 6]])
```

(continues on next page)

(continued from previous page)

```
>>> M.columnspace()
[Matrix([
[ 1],
[-2],
[ 3]]), Matrix([
[0],
[0],
[6]])]
```

See also:

[nullspace](#) (page 1236), [rowspace](#) (page 1237)

nullspace(*simplify=False, iszerofunc=<function _iszero>*)

Returns list of vectors (Matrix objects) that span nullspace of M

Examples

```
>>> from sympy import Matrix
>>> M = Matrix(3, 3, [1, 3, 0, -2, -6, 0, 3, 9, 6])
>>> M
Matrix([
[ 1,  3, 0],
[-2, -6, 0],
[ 3,  9, 6]])
>>> M.nullspace()
[Matrix([
[-3],
[ 1],
[ 0]])]
```

See also:

[columnspace](#) (page 1235), [rowspace](#) (page 1237)

classmethod orthogonalize(*vecs, **kwargs)

Apply the Gram-Schmidt orthogonalization procedure to vectors supplied in vecs.

Parameters**vecs**

vectors to be made orthogonal

normalize : bool

If True, return an orthonormal basis.

rankcheck : bool

If True, the computation does not stop when encountering linearly dependent vectors.

If False, it will raise `ValueError` when any zero or linearly dependent vectors are found.

Returns

list

List of orthogonal (or orthonormal) basis vectors.

Examples

```
>>> from sympy import I, Matrix
>>> v = [Matrix([1, I]), Matrix([1, -I])]
>>> Matrix.orthogonalize(*v)
[Matrix([
[1],
[I]]), Matrix([
[ 1],
[-I]])]
```

See also:

[*MatrixBase.QRdecomposition*](#) (page 1287)

References

[R570]

rowspace(*simplify=False*)

Returns a list of vectors that span the row space of M.

Examples

```
>>> from sympy import Matrix
>>> M = Matrix(3, 3, [1, 3, 0, -2, -6, 0, 3, 9, 6])
>>> M
Matrix([
[ 1,  3, 0],
[-2, -6, 0],
[ 3,  9, 6]])
>>> M.rowspace()
[Matrix([[1, 3, 0]]), Matrix([[0, 0, 6]])]
```

MatrixEigen Class Reference

class sympy.matrices.matrices.**MatrixEigen**

Provides basic matrix eigenvalue/vector operations. Should not be instantiated directly. See `eigen.py` for their implementations.

bidiagonal_decomposition(*upper=True*)

Returns $(U, B, V.H)$ for

$$A = UBV^H$$

where A is the input matrix, and B is its Bidiagonalized form

Note: Bidiagonal Computation can hang for symbolic matrices.

Parameters

upper : bool. Whether to do upper bidiagonalization or lower.
True for upper and False for lower.

References

[R571], [R572]

bidiagonalize(*upper=True*)

Returns B , the Bidiagonalized form of the input matrix.

Note: Bidiagonal Computation can hang for symbolic matrices.

Parameters

upper : bool. Whether to do upper bidiagonalization or lower.
True for upper and False for lower.

References

[R573], [R574]

diagonalize(*reals_only=False, sort=False, normalize=False*)

Return (P, D) , where D is diagonal and

$$D = P^{-1} * M * P$$

where M is current matrix.

Parameters

reals_only : bool. Whether to throw an error if complex numbers are need

to diagonalize. (Default: False)

sort : bool. Sort the eigenvalues along the diagonal. (Default: False)

normalize : bool. If True, normalize the columns of P . (Default: False)

Examples

```
>>> from sympy import Matrix
>>> M = Matrix(3, 3, [1, 2, 0, 0, 3, 0, 2, -4, 2])
>>> M
Matrix([
[1, 2, 0],
[0, 3, 0],
[2, -4, 2]])
>>> (P, D) = M.diagonalize()
>>> D
Matrix([
[1, 0, 0],
[0, 2, 0],
```

(continues on next page)

(continued from previous page)

```
[0, 0, 3]])
>>> P
Matrix([
[-1, 0, -1],
[ 0, 0, -1],
[ 2, 1,  2]])
>>> P.inv() * M * P
Matrix([
[1, 0, 0],
[0, 2, 0],
[0, 0, 3]])
```

See also:

[*is_diagonal*](#) (page 1337), [*is_diagonalizable*](#) (page 1241)

eigenvals(*error_when_incomplete=True, **flags*)

Compute eigenvalues of the matrix.

Parameters

error_when_incomplete : bool, optional

If it is set to True, it will raise an error if not all eigenvalues are computed. This is caused by roots not returning a full list of eigenvalues.

simplify : bool or function, optional

If it is set to True, it attempts to return the most simplified form of expressions returned by applying default simplification method in every routine.

If it is set to False, it will skip simplification in this particular routine to save computation resources.

If a function is passed to, it will attempt to apply the particular function as simplification method.

rational : bool, optional

If it is set to True, every floating point numbers would be replaced with rationals before computation. It can solve some issues of roots routine not working well with floats.

multiple : bool, optional

If it is set to True, the result will be in the form of a list.

If it is set to False, the result will be in the form of a dictionary.

Returns

eigs : list or dict

Eigenvalues of a matrix. The return format would be specified by the key multiple.

Raises

MatrixError

If not enough roots had got computed.

NonSquareMatrixError

If attempted to compute eigenvalues from a non-square matrix.

Examples

```
>>> from sympy import Matrix
>>> M = Matrix(3, 3, [0, 1, 1, 1, 0, 0, 1, 1, 1])
>>> M.eigenvals()
{-1: 1, 0: 1, 2: 1}
```

Notes

Eigenvalues of a matrix A can be computed by solving a matrix equation $\det(A - \lambda I) = 0$

It's not always possible to return radical solutions for eigenvalues for matrices larger than 4,4 shape due to Abel-Ruffini theorem.

If there is no radical solution is found for the eigenvalue, it may return eigenvalues in the form of [`sympy.polys.rootoftools.ComplexRootOf`](#) (page 2431).

See also:

[`MatrixDeterminant.charpoly`](#) (page 1228), [`eigenvecs`](#) (page 1240)

eigenvecs(*error_when_incomplete=True*, *iszerofunc=<function _iszero>*, ***flags*)

Compute eigenvectors of the matrix.

Parameters

error_when_incomplete : bool, optional

Raise an error when not all eigenvalues are computed. This is caused by roots not returning a full list of eigenvalues.

iszerofunc : function, optional

Specifies a zero testing function to be used in `rref`.

Default value is `_iszero`, which uses SymPy's naive and fast default assumption handler.

It can also accept any user-specified zero testing function, if it is formatted as a function which accepts a single symbolic argument and returns `True` if it is tested as zero and `False` if it is tested as non-zero, and `None` if it is undecidable.

simplify : bool or function, optional

If `True`, `as_content_primitive()` will be used to tidy up normalization artifacts.

It will also be used by the `nullspace` routine.

chop : bool or positive number, optional

If the matrix contains any Floats, they will be changed to Rationals for computation purposes, but the answers will be returned after being evaluated with `evalf`. The `chop` flag is passed to `evalf`. When `chop=True` a default precision will be used; a number will be interpreted as the desired level of precision.

Returns

ret : [(eigenval, multiplicity, eigenspace), ...]

A ragged list containing tuples of data obtained by `eigenvals` and `nullspace`.

`eigenspace` is a list containing the eigenvector for each eigenvalue.

`eigenvector` is a vector in the form of a `Matrix`. e.g. a vector of length 3 is returned as `Matrix([a_1, a_2, a_3])`.

Raises

NotImplementedError

If failed to compute nullspace.

Examples

```
>>> from sympy import Matrix
>>> M = Matrix(3, 3, [0, 1, 1, 1, 0, 0, 1, 1, 1])
>>> M.eigenvecs()
[(-1, 1, [Matrix([
[-1],
[ 1],
[ 0]])]), (0, 1, [Matrix([
[ 0],
[-1],
[ 1]])]), (2, 1, [Matrix([
[2/3],
[1/3],
[ 1]])])]
```

See also:

[`eigenvals`](#) (page 1239), [`MatrixSubspaces.nullspace`](#) (page 1236)

is_diagonalizable(*reals_only=False*, ***kwargs*)

Returns True if a matrix is diagonalizable.

Parameters

reals_only : bool, optional

If True, it tests whether the matrix can be diagonalized to contain only real numbers on the diagonal.

If False, it tests whether the matrix can be diagonalized at all, even with numbers that may not be real.

Examples

Example of a diagonalizable matrix:

```
>>> from sympy import Matrix
>>> M = Matrix([[1, 2, 0], [0, 3, 0], [2, -4, 2]])
>>> M.is_diagonalizable()
True
```

Example of a non-diagonalizable matrix:

```
>>> M = Matrix([[0, 1], [0, 0]])
>>> M.is_diagonalizable()
False
```

Example of a matrix that is diagonalized in terms of non-real entries:

```
>>> M = Matrix([[0, 1], [-1, 0]])
>>> M.is_diagonalizable(reals_only=False)
True
>>> M.is_diagonalizable(reals_only=True)
False
```

See also:

[*is_diagonal*](#) (page 1337), [*diagonalize*](#) (page 1238)

property `is_indefinite`

Finds out the definiteness of a matrix.

Explanation

A square real matrix A is:

- A positive definite matrix if $x^T A x > 0$ for all non-zero real vectors x .
- A positive semidefinite matrix if $x^T A x \geq 0$ for all non-zero real vectors x .
- A negative definite matrix if $x^T A x < 0$ for all non-zero real vectors x .
- A negative semidefinite matrix if $x^T A x \leq 0$ for all non-zero real vectors x .
- An indefinite matrix if there exists non-zero real vectors x, y with $x^T A x > 0 > y^T A y$.

A square complex matrix A is:

- A positive definite matrix if $\operatorname{re}(x^H A x) > 0$ for all non-zero complex vectors x .
- A positive semidefinite matrix if $\operatorname{re}(x^H A x) \geq 0$ for all non-zero complex vectors x .
- A negative definite matrix if $\operatorname{re}(x^H A x) < 0$ for all non-zero complex vectors x .
- A negative semidefinite matrix if $\operatorname{re}(x^H A x) \leq 0$ for all non-zero complex vectors x .
- An indefinite matrix if there exists non-zero complex vectors x, y with $\operatorname{re}(x^H A x) > 0 > \operatorname{re}(y^H A y)$.

A matrix need not be symmetric or hermitian to be positive definite.

- A real non-symmetric matrix is positive definite if and only if $\frac{A+A^T}{2}$ is positive definite.
- A complex non-hermitian matrix is positive definite if and only if $\frac{A+A^H}{2}$ is positive definite.

And this extension can apply for all the definitions above.

However, for complex cases, you can restrict the definition of $\text{re}(x^H Ax) > 0$ to $x^H Ax > 0$ and require the matrix to be hermitian. But we do not present this restriction for computation because you can check `M.is_hermitian` independently with this and use the same procedure.

Examples

An example of symmetric positive definite matrix:

```
>>> from sympy import Matrix, symbols
>>> from sympy.plotting import plot3d
>>> a, b = symbols('a b')
>>> x = Matrix([a, b])
```

```
>>> A = Matrix([[1, 0], [0, 1]])
>>> A.is_positive_definite
True
>>> A.is_positive_semidefinite
True
```

```
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```

An example of symmetric positive semidefinite matrix:

```
>>> A = Matrix([[1, -1], [-1, 1]])
>>> A.is_positive_definite
False
>>> A.is_positive_semidefinite
True
```

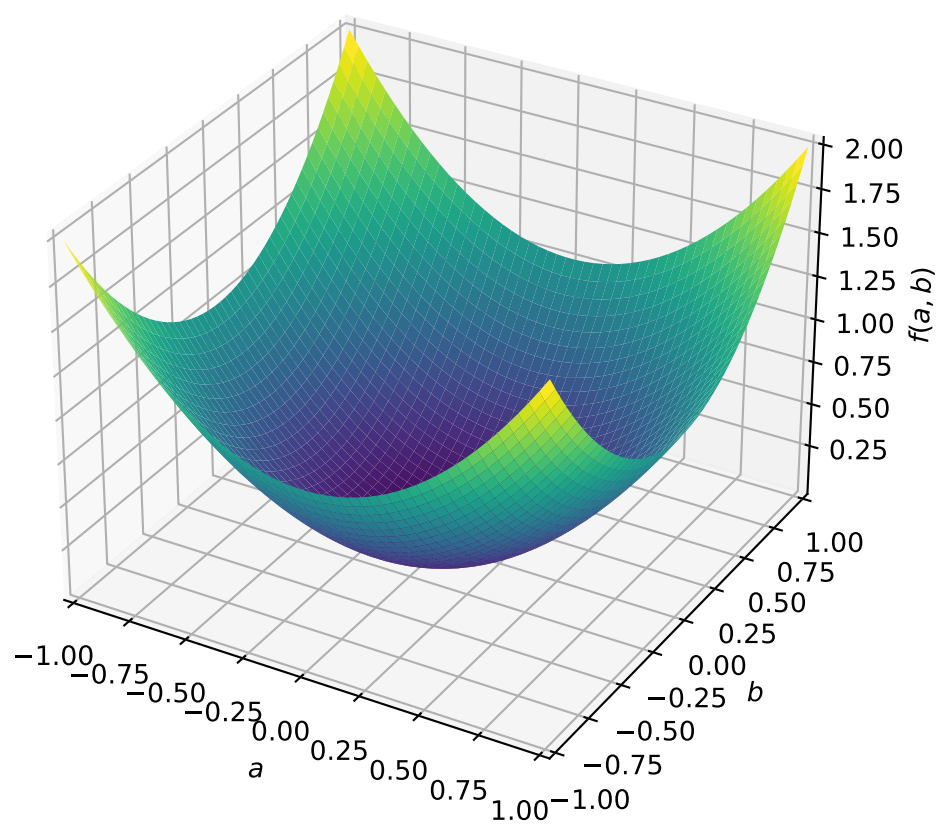
```
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```

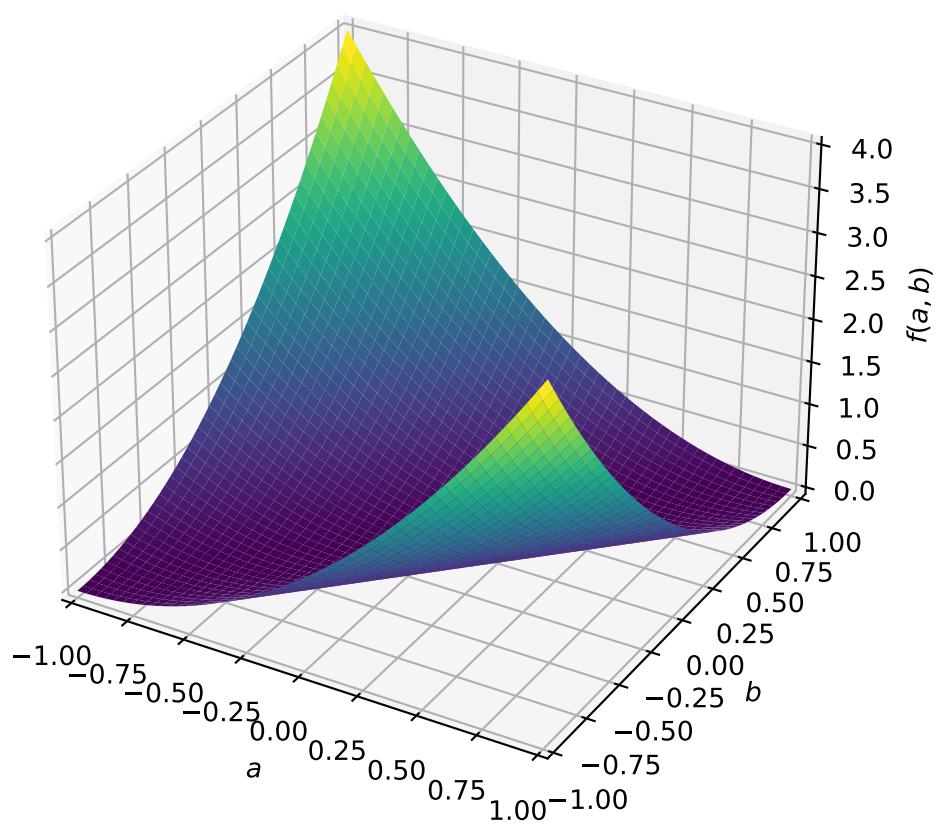
An example of symmetric negative definite matrix:

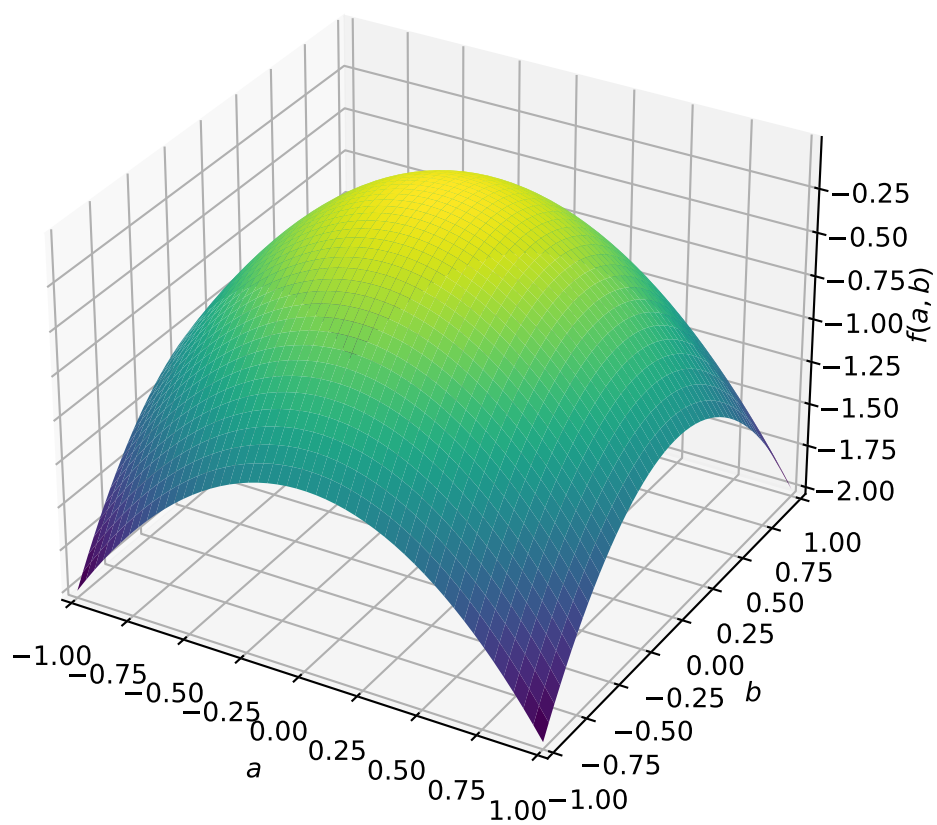
```
>>> A = Matrix([[-1, 0], [0, -1]])
>>> A.is_negative_definite
True
>>> A.is_negative_semidefinite
True
>>> A.is_indefinite
False
```

```
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```

An example of symmetric indefinite matrix:

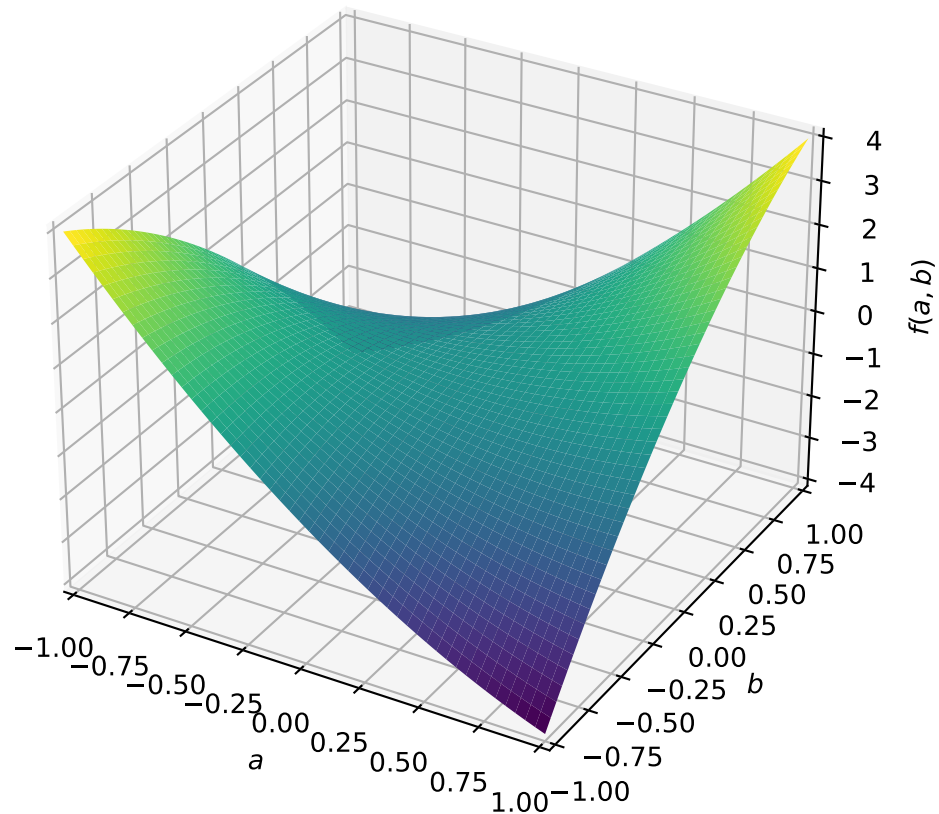






```
>>> A = Matrix([[1, 2], [2, -1]])  
>>> A.is_indefinite  
True
```

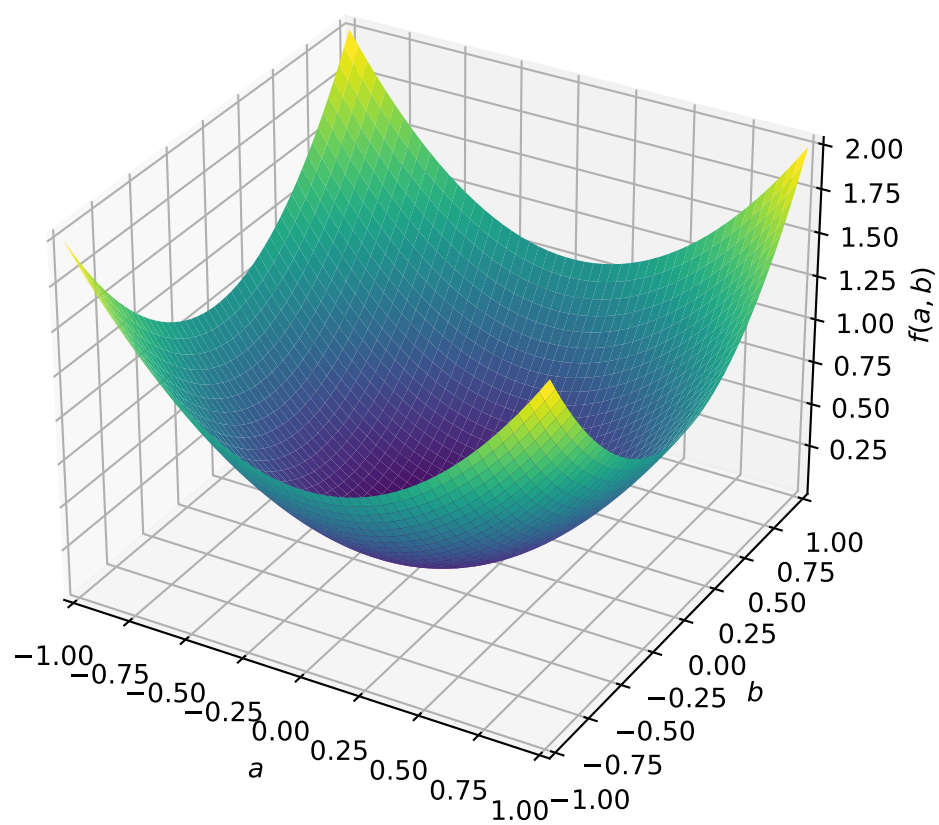
```
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```



An example of non-symmetric positive definite matrix.

```
>>> A = Matrix([[1, 2], [-2, 1]])  
>>> A.is_positive_definite  
True  
>>> A.is_positive_semidefinite  
True
```

```
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```



Notes

Although some people trivialize the definition of positive definite matrices only for symmetric or hermitian matrices, this restriction is not correct because it does not classify all instances of positive definite matrices from the definition $x^T Ax > 0$ or $\operatorname{re}(x^H Ax) > 0$.

For instance, `Matrix([[1, 2], [-2, 1]])` presented in the example above is an example of real positive definite matrix that is not symmetric.

However, since the following formula holds true;

$$\operatorname{re}(x^H Ax) > 0 \iff \operatorname{re}\left(x^H \frac{A + A^H}{2} x\right) > 0$$

We can classify all positive definite matrices that may or may not be symmetric or hermitian by transforming the matrix to $\frac{A+A^T}{2}$ or $\frac{A+A^H}{2}$ (which is guaranteed to be always real symmetric or complex hermitian) and we can defer most of the studies to symmetric or hermitian positive definite matrices.

But it is a different problem for the existence of Cholesky decomposition. Because even though a non symmetric or a non hermitian matrix can be positive definite, Cholesky or LDL decomposition does not exist because the decompositions require the matrix to be symmetric or hermitian.

References

[R575], [R576], [R577]

property `is_negative_definite`

Finds out the definiteness of a matrix.

Explanation

A square real matrix A is:

- A positive definite matrix if $x^T Ax > 0$ for all non-zero real vectors x .
- A positive semidefinite matrix if $x^T Ax \geq 0$ for all non-zero real vectors x .
- A negative definite matrix if $x^T Ax < 0$ for all non-zero real vectors x .
- A negative semidefinite matrix if $x^T Ax \leq 0$ for all non-zero real vectors x .
- An indefinite matrix if there exists non-zero real vectors x, y with $x^T Ax > 0 > y^T Ay$.

A square complex matrix A is:

- A positive definite matrix if $\operatorname{re}(x^H Ax) > 0$ for all non-zero complex vectors x .
- A positive semidefinite matrix if $\operatorname{re}(x^H Ax) \geq 0$ for all non-zero complex vectors x .
- A negative definite matrix if $\operatorname{re}(x^H Ax) < 0$ for all non-zero complex vectors x .
- A negative semidefinite matrix if $\operatorname{re}(x^H Ax) \leq 0$ for all non-zero complex vectors x .
- An indefinite matrix if there exists non-zero complex vectors x, y with $\operatorname{re}(x^H Ax) > 0 > \operatorname{re}(y^H Ay)$.

A matrix need not be symmetric or hermitian to be positive definite.

- A real non-symmetric matrix is positive definite if and only if $\frac{A+A^T}{2}$ is positive definite.
- A complex non-hermitian matrix is positive definite if and only if $\frac{A+A^H}{2}$ is positive definite.

And this extension can apply for all the definitions above.

However, for complex cases, you can restrict the definition of $\text{re}(x^H Ax) > 0$ to $x^H Ax > 0$ and require the matrix to be hermitian. But we do not present this restriction for computation because you can check `M.is_hermitian` independently with this and use the same procedure.

Examples

An example of symmetric positive definite matrix:

```
>>> from sympy import Matrix, symbols
>>> from sympy.plotting import plot3d
>>> a, b = symbols('a b')
>>> x = Matrix([a, b])
```

```
>>> A = Matrix([[1, 0], [0, 1]])
>>> A.is_positive_definite
True
>>> A.is_positive_semidefinite
True
```

```
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```

An example of symmetric positive semidefinite matrix:

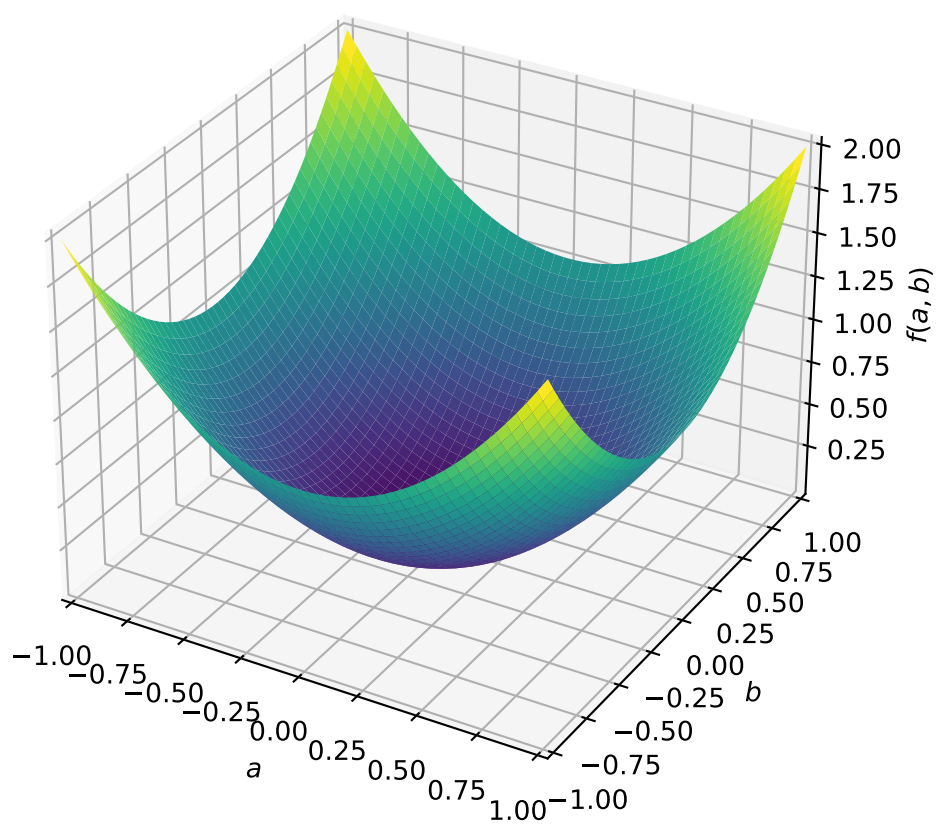
```
>>> A = Matrix([[1, -1], [-1, 1]])
>>> A.is_positive_definite
False
>>> A.is_positive_semidefinite
True
```

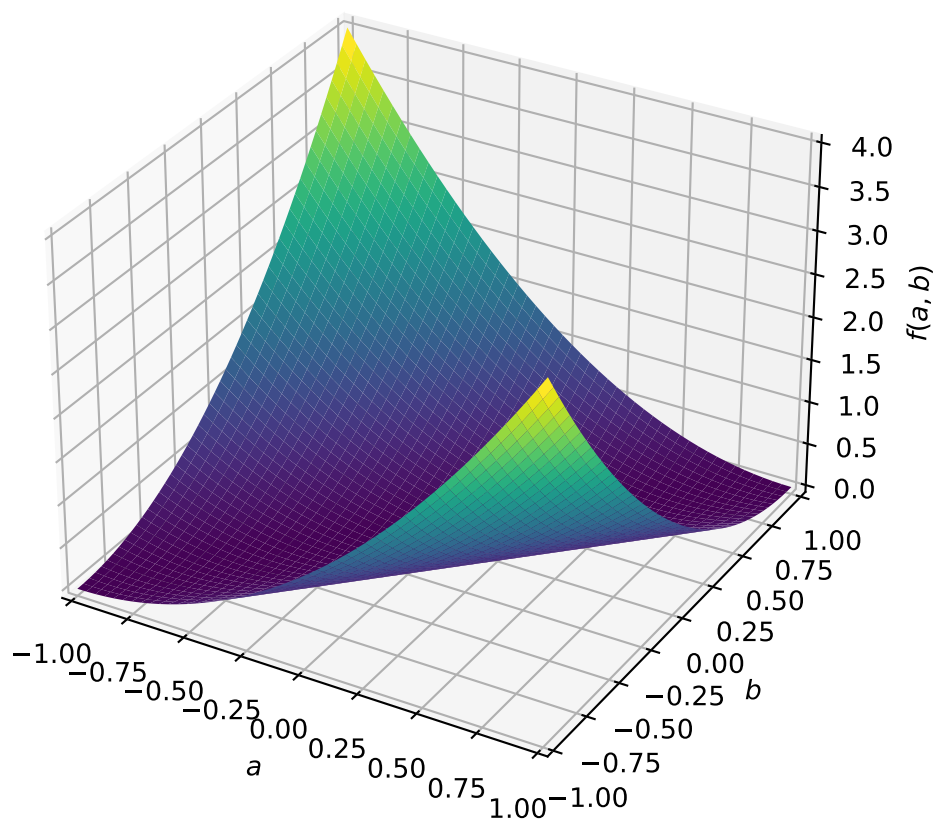
```
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```

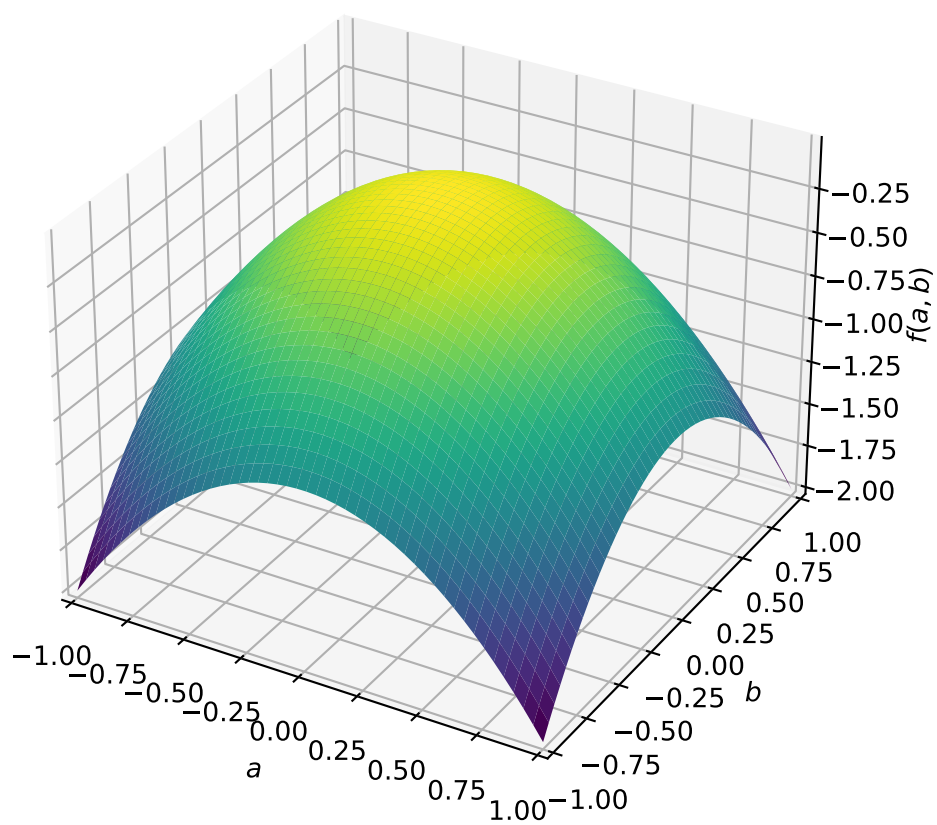
An example of symmetric negative definite matrix:

```
>>> A = Matrix([[-1, 0], [0, -1]])
>>> A.is_negative_definite
True
>>> A.is_negative_semidefinite
True
>>> A.is_indefinite
False
```

```
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```



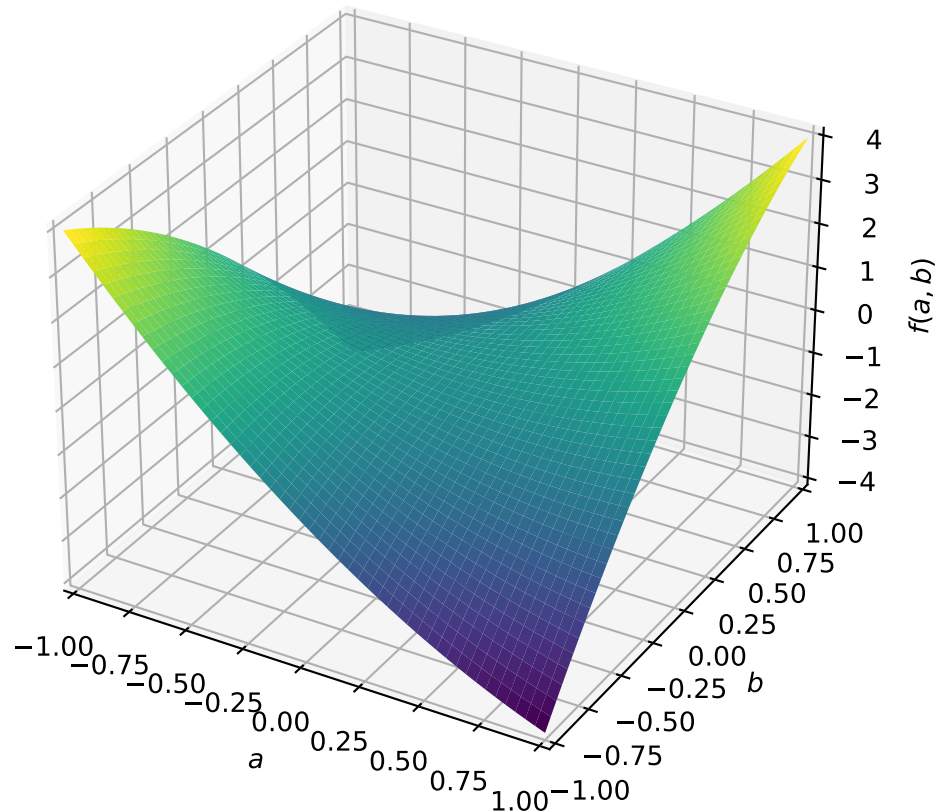




An example of symmetric indefinite matrix:

```
>>> A = Matrix([[1, 2], [2, -1]])
>>> A.is_indefinite
True
```

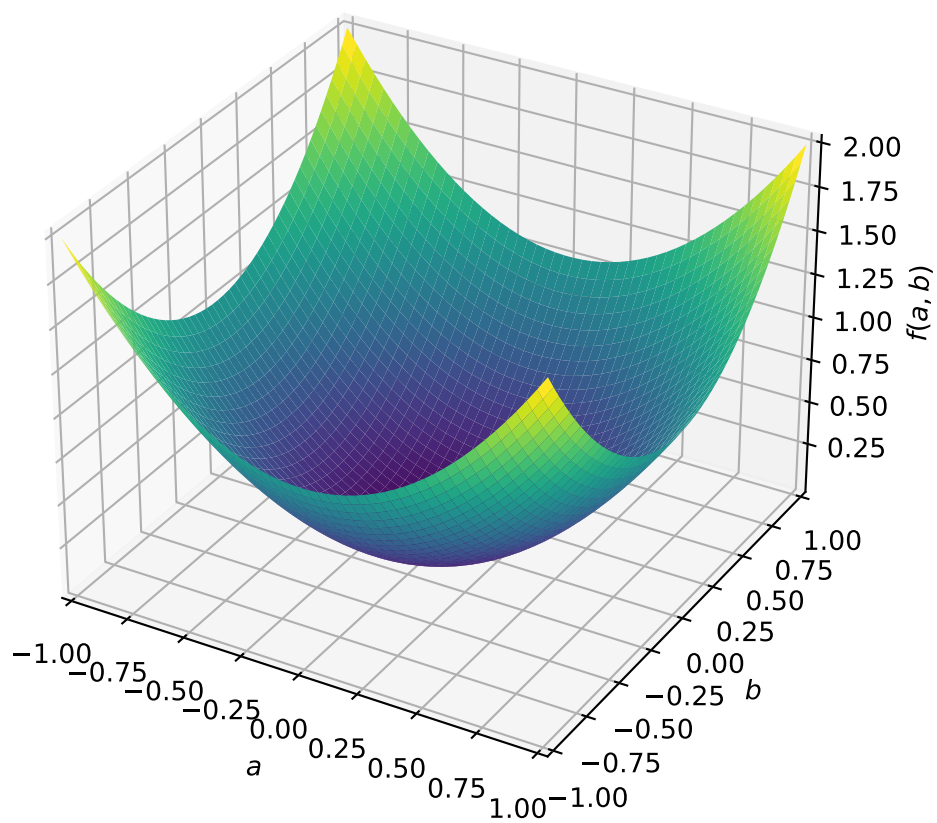
```
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```



An example of non-symmetric positive definite matrix.

```
>>> A = Matrix([[1, 2], [-2, 1]])
>>> A.is_positive_definite
True
>>> A.is_positive_semidefinite
True
```

```
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```



Notes

Although some people trivialize the definition of positive definite matrices only for symmetric or hermitian matrices, this restriction is not correct because it does not classify all instances of positive definite matrices from the definition $x^T Ax > 0$ or $\text{re}(x^H Ax) > 0$.

For instance, `Matrix([[1, 2], [-2, 1]])` presented in the example above is an example of real positive definite matrix that is not symmetric.

However, since the following formula holds true;

$$\text{re}(x^H Ax) > 0 \iff \text{re}(x^H \frac{A + A^H}{2} x) > 0$$

We can classify all positive definite matrices that may or may not be symmetric or hermitian by transforming the matrix to $\frac{A+A^T}{2}$ or $\frac{A+A^H}{2}$ (which is guaranteed to be always real symmetric or complex hermitian) and we can defer most of the studies to symmetric or hermitian positive definite matrices.

But it is a different problem for the existence of Cholesky decomposition. Because even though a non symmetric or a non hermitian matrix can be positive definite, Cholesky or LDL decomposition does not exist because the decompositions require the matrix to be symmetric or hermitian.

References

[R578], [R579], [R580]

property is_negative_semidefinite

Finds out the definiteness of a matrix.

Explanation

A square real matrix A is:

- A positive definite matrix if $x^T Ax > 0$ for all non-zero real vectors x .
- A positive semidefinite matrix if $x^T Ax \geq 0$ for all non-zero real vectors x .
- A negative definite matrix if $x^T Ax < 0$ for all non-zero real vectors x .
- A negative semidefinite matrix if $x^T Ax \leq 0$ for all non-zero real vectors x .
- An indefinite matrix if there exists non-zero real vectors x, y with $x^T Ax > 0 > y^T Ay$.

A square complex matrix A is:

- A positive definite matrix if $\text{re}(x^H Ax) > 0$ for all non-zero complex vectors x .
- A positive semidefinite matrix if $\text{re}(x^H Ax) \geq 0$ for all non-zero complex vectors x .
- A negative definite matrix if $\text{re}(x^H Ax) < 0$ for all non-zero complex vectors x .
- A negative semidefinite matrix if $\text{re}(x^H Ax) \leq 0$ for all non-zero complex vectors x .
- An indefinite matrix if there exists non-zero complex vectors x, y with $\text{re}(x^H Ax) > 0 > \text{re}(y^H Ay)$.

A matrix need not be symmetric or hermitian to be positive definite.

- A real non-symmetric matrix is positive definite if and only if $\frac{A+A^T}{2}$ is positive definite.
- A complex non-hermitian matrix is positive definite if and only if $\frac{A+A^H}{2}$ is positive definite.

And this extension can apply for all the definitions above.

However, for complex cases, you can restrict the definition of $\text{re}(x^H Ax) > 0$ to $x^H Ax > 0$ and require the matrix to be hermitian. But we do not present this restriction for computation because you can check `M.is_hermitian` independently with this and use the same procedure.

Examples

An example of symmetric positive definite matrix:

```
>>> from sympy import Matrix, symbols
>>> from sympy.plotting import plot3d
>>> a, b = symbols('a b')
>>> x = Matrix([a, b])
```

```
>>> A = Matrix([[1, 0], [0, 1]])
>>> A.is_positive_definite
True
>>> A.is_positive_semidefinite
True
```

```
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```

An example of symmetric positive semidefinite matrix:

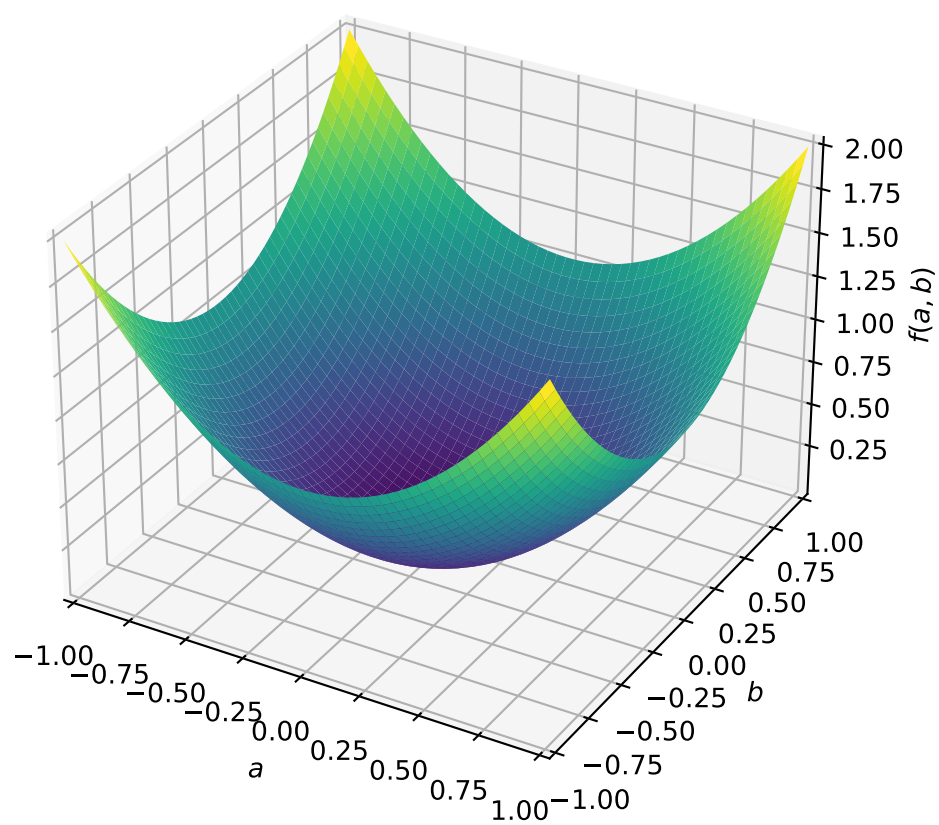
```
>>> A = Matrix([[1, -1], [-1, 1]])
>>> A.is_positive_definite
False
>>> A.is_positive_semidefinite
True
```

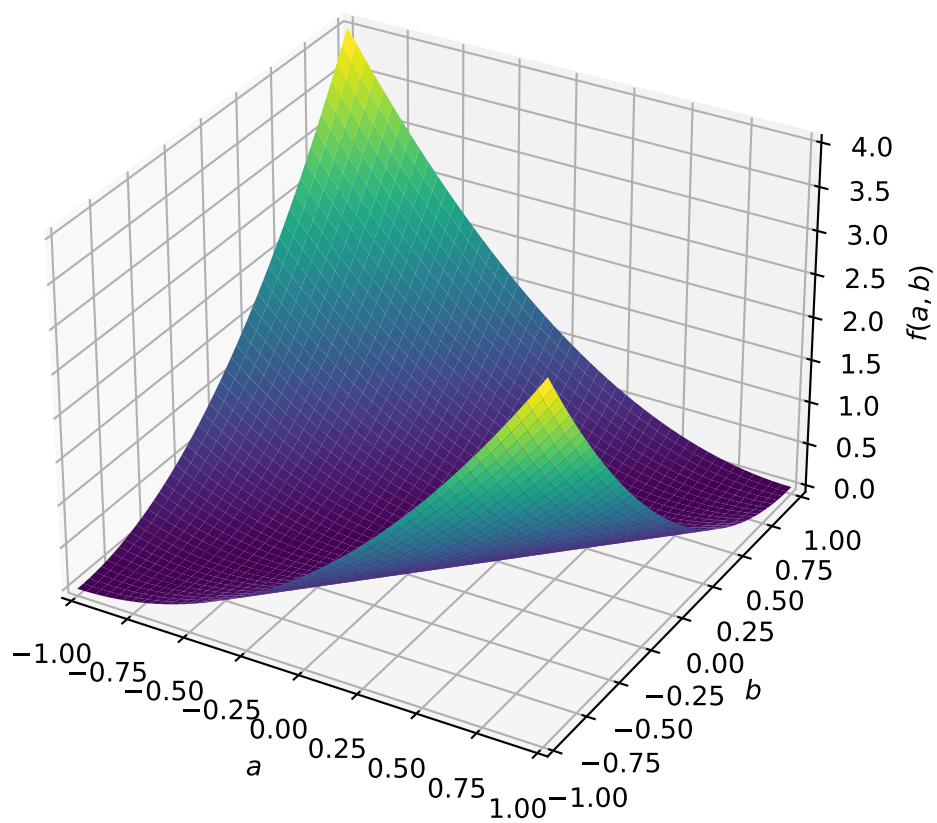
```
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```

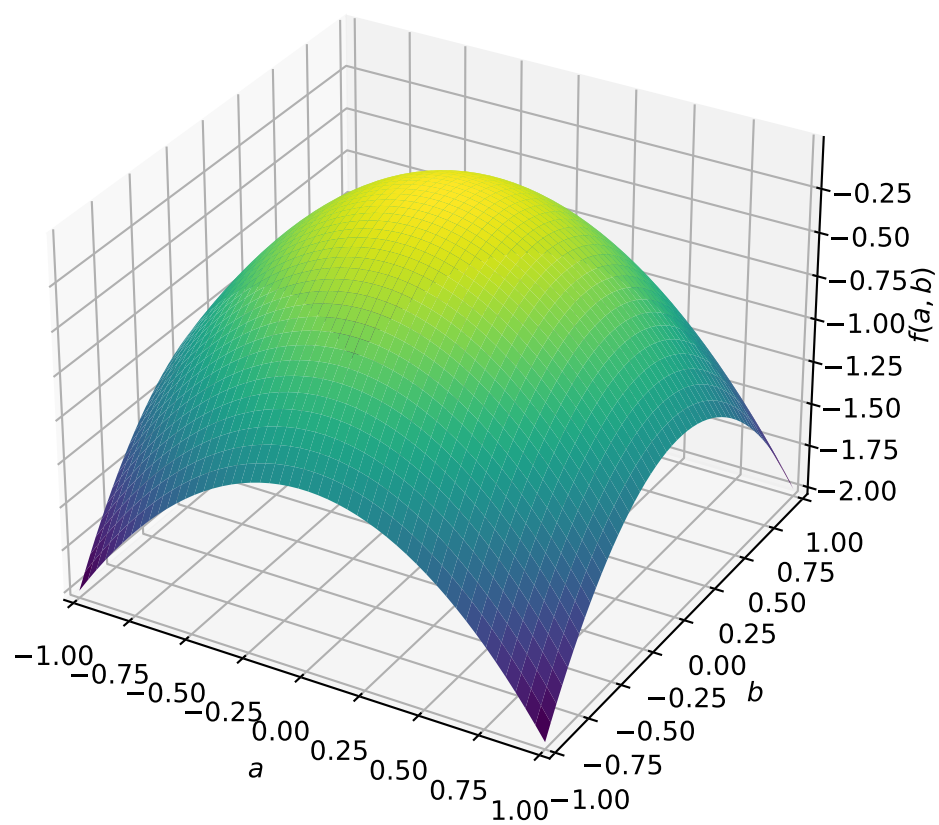
An example of symmetric negative definite matrix:

```
>>> A = Matrix([[-1, 0], [0, -1]])
>>> A.is_negative_definite
True
>>> A.is_negative_semidefinite
True
>>> A.is_indefinite
False
```

```
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```



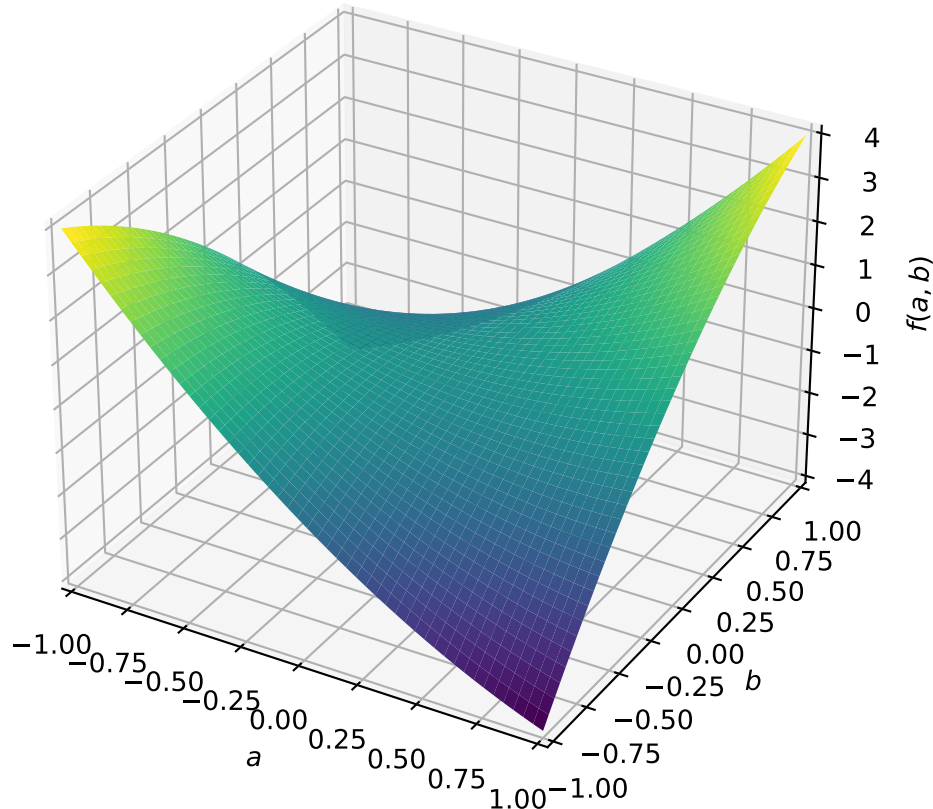




An example of symmetric indefinite matrix:

```
>>> A = Matrix([[1, 2], [2, -1]])
>>> A.is_indefinite
True
```

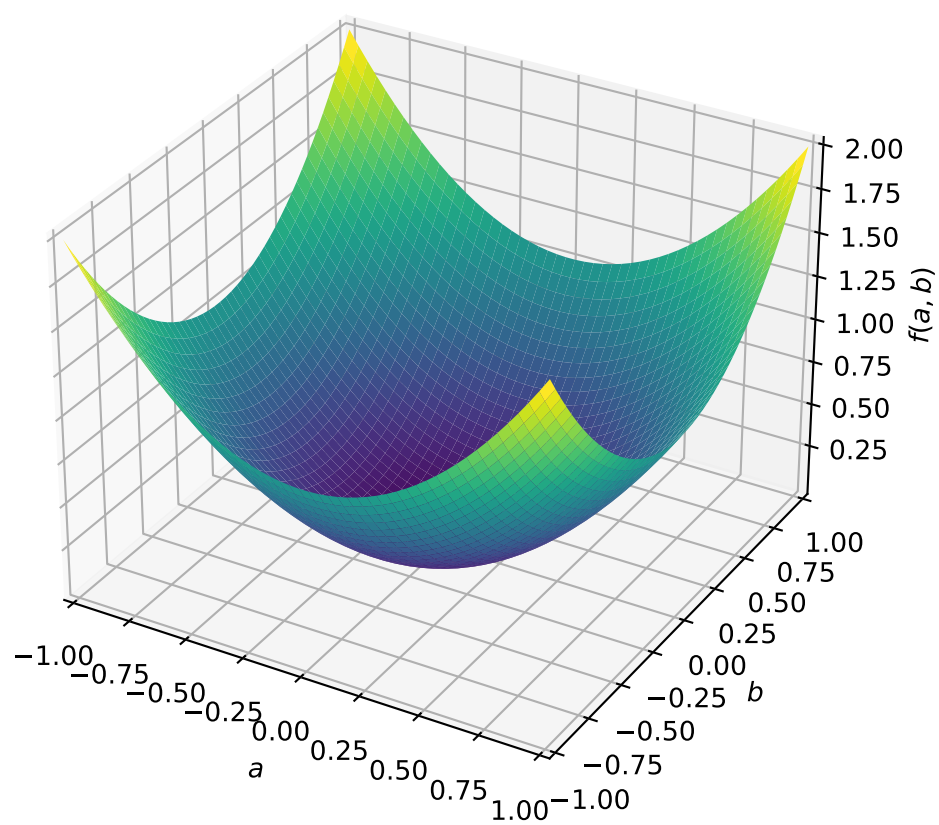
```
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```



An example of non-symmetric positive definite matrix.

```
>>> A = Matrix([[1, 2], [-2, 1]])
>>> A.is_positive_definite
True
>>> A.is_positive_semidefinite
True
```

```
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```



Notes

Although some people trivialize the definition of positive definite matrices only for symmetric or hermitian matrices, this restriction is not correct because it does not classify all instances of positive definite matrices from the definition $x^T Ax > 0$ or $\text{re}(x^H Ax) > 0$.

For instance, `Matrix([[1, 2], [-2, 1]])` presented in the example above is an example of real positive definite matrix that is not symmetric.

However, since the following formula holds true;

$$\text{re}(x^H Ax) > 0 \iff \text{re}\left(x^H \frac{A + A^H}{2} x\right) > 0$$

We can classify all positive definite matrices that may or may not be symmetric or hermitian by transforming the matrix to $\frac{A+A^T}{2}$ or $\frac{A+A^H}{2}$ (which is guaranteed to be always real symmetric or complex hermitian) and we can defer most of the studies to symmetric or hermitian positive definite matrices.

But it is a different problem for the existence of Cholesky decomposition. Because even though a non symmetric or a non hermitian matrix can be positive definite, Cholesky or LDL decomposition does not exist because the decompositions require the matrix to be symmetric or hermitian.

References

[R581], [R582], [R583]

property is_positive_definite

Finds out the definiteness of a matrix.

Explanation

A square real matrix A is:

- A positive definite matrix if $x^T Ax > 0$ for all non-zero real vectors x .
- A positive semidefinite matrix if $x^T Ax \geq 0$ for all non-zero real vectors x .
- A negative definite matrix if $x^T Ax < 0$ for all non-zero real vectors x .
- A negative semidefinite matrix if $x^T Ax \leq 0$ for all non-zero real vectors x .
- An indefinite matrix if there exists non-zero real vectors x, y with $x^T Ax > 0 > y^T Ay$.

A square complex matrix A is:

- A positive definite matrix if $\text{re}(x^H Ax) > 0$ for all non-zero complex vectors x .
- A positive semidefinite matrix if $\text{re}(x^H Ax) \geq 0$ for all non-zero complex vectors x .
- A negative definite matrix if $\text{re}(x^H Ax) < 0$ for all non-zero complex vectors x .
- A negative semidefinite matrix if $\text{re}(x^H Ax) \leq 0$ for all non-zero complex vectors x .
- An indefinite matrix if there exists non-zero complex vectors x, y with $\text{re}(x^H Ax) > 0 > \text{re}(y^H Ay)$.

A matrix need not be symmetric or hermitian to be positive definite.

- A real non-symmetric matrix is positive definite if and only if $\frac{A+A^T}{2}$ is positive definite.
- A complex non-hermitian matrix is positive definite if and only if $\frac{A+A^H}{2}$ is positive definite.

And this extension can apply for all the definitions above.

However, for complex cases, you can restrict the definition of $\text{re}(x^H Ax) > 0$ to $x^H Ax > 0$ and require the matrix to be hermitian. But we do not present this restriction for computation because you can check `M.is_hermitian` independently with this and use the same procedure.

Examples

An example of symmetric positive definite matrix:

```
>>> from sympy import Matrix, symbols
>>> from sympy.plotting import plot3d
>>> a, b = symbols('a b')
>>> x = Matrix([a, b])
```

```
>>> A = Matrix([[1, 0], [0, 1]])
>>> A.is_positive_definite
True
>>> A.is_positive_semidefinite
True
```

```
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```

An example of symmetric positive semidefinite matrix:

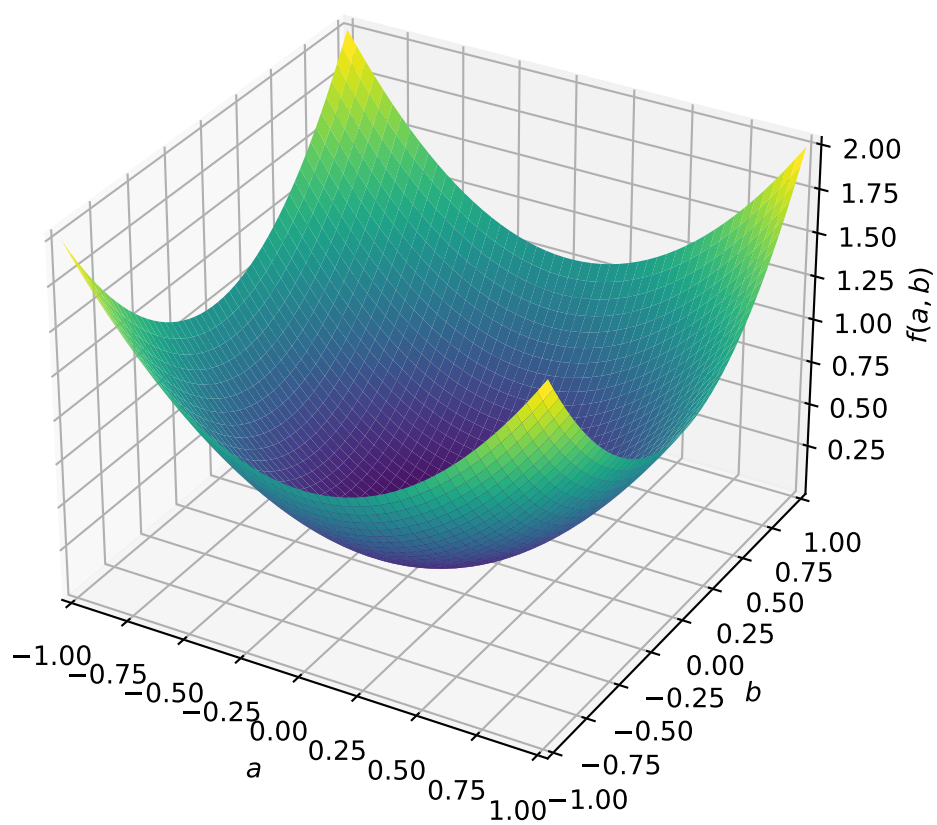
```
>>> A = Matrix([[1, -1], [-1, 1]])
>>> A.is_positive_definite
False
>>> A.is_positive_semidefinite
True
```

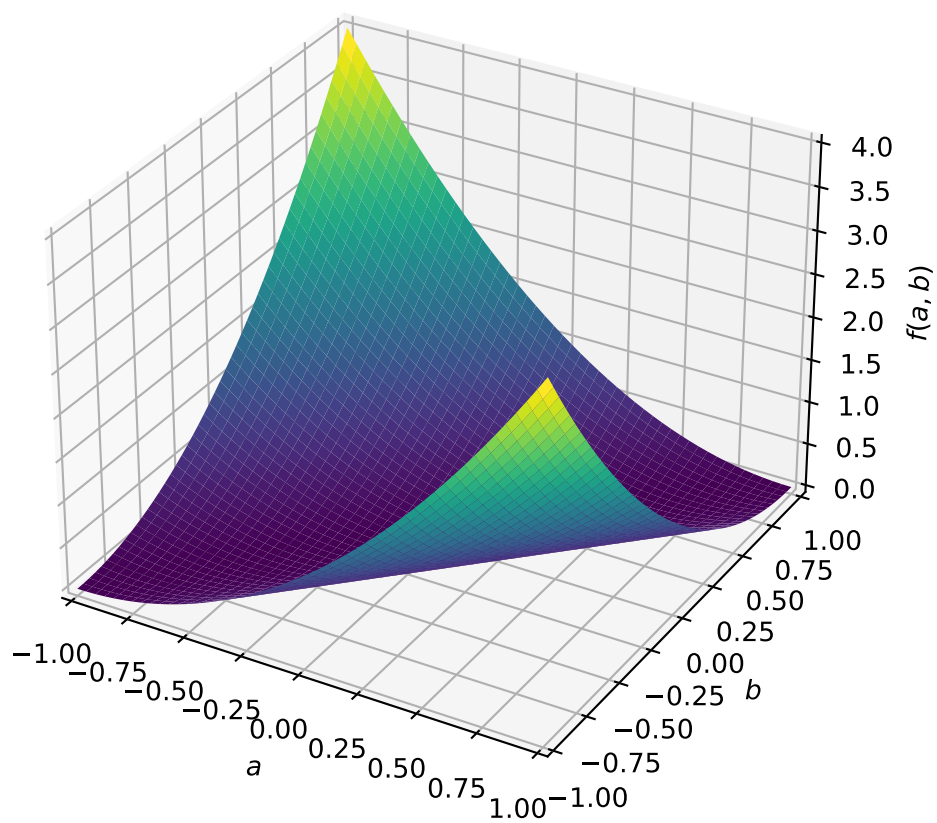
```
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```

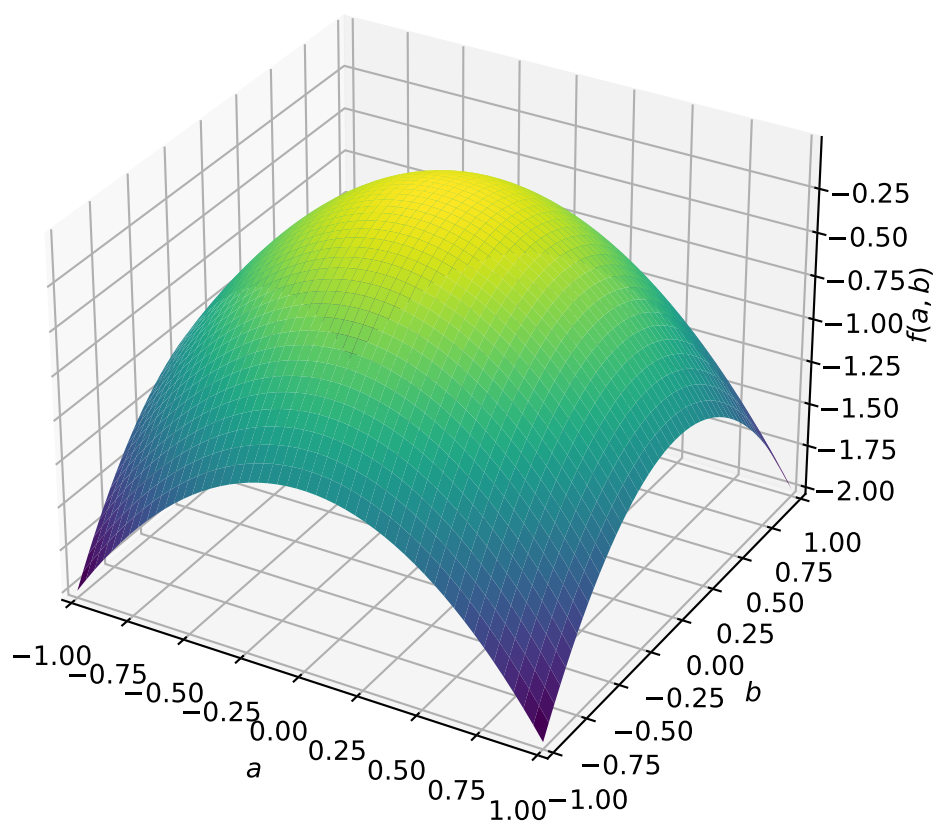
An example of symmetric negative definite matrix:

```
>>> A = Matrix([[-1, 0], [0, -1]])
>>> A.is_negative_definite
True
>>> A.is_negative_semidefinite
True
>>> A.is_indefinite
False
```

```
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```

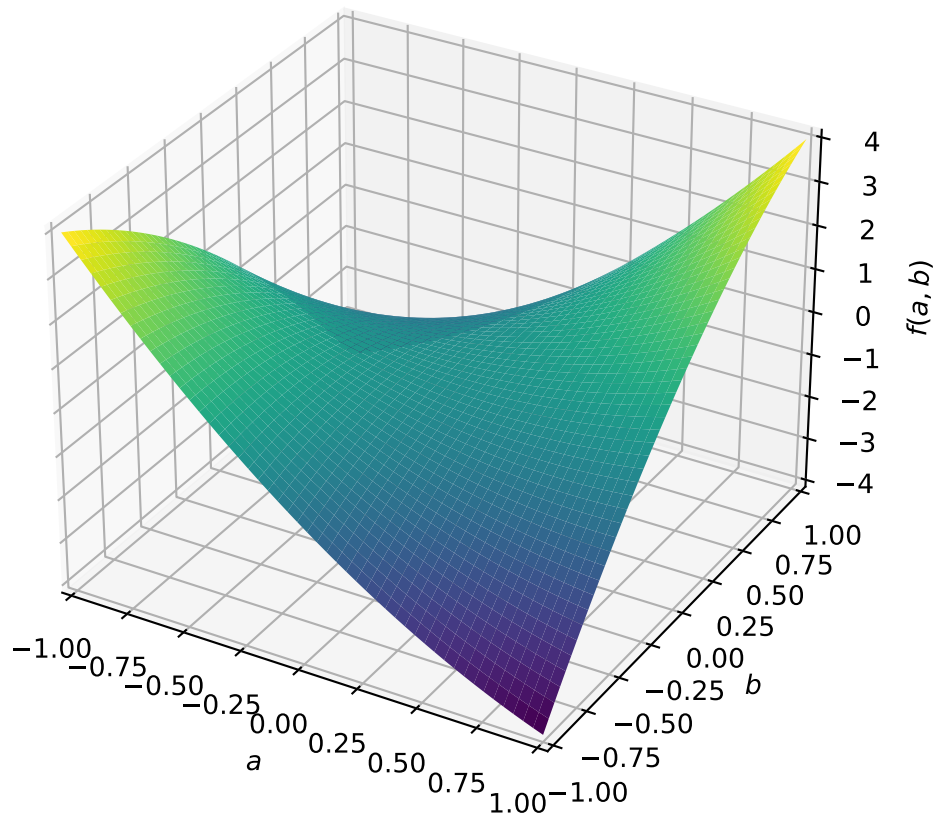




An example of symmetric indefinite matrix:

```
>>> A = Matrix([[1, 2], [2, -1]])
>>> A.is_indefinite
True
```

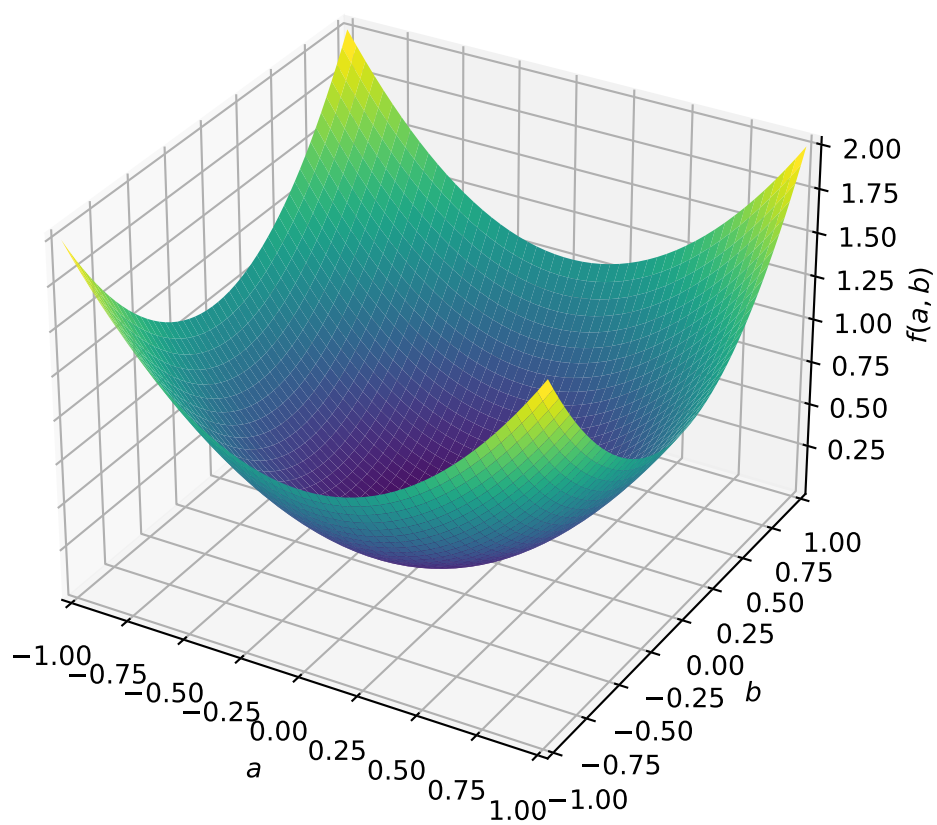
```
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```



An example of non-symmetric positive definite matrix.

```
>>> A = Matrix([[1, 2], [-2, 1]])
>>> A.is_positive_definite
True
>>> A.is_positive_semidefinite
True
```

```
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```



Notes

Although some people trivialize the definition of positive definite matrices only for symmetric or hermitian matrices, this restriction is not correct because it does not classify all instances of positive definite matrices from the definition $x^T Ax > 0$ or $\text{re}(x^H Ax) > 0$.

For instance, `Matrix([[1, 2], [-2, 1]])` presented in the example above is an example of real positive definite matrix that is not symmetric.

However, since the following formula holds true;

$$\text{re}(x^H Ax) > 0 \iff \text{re}(x^H \frac{A + A^H}{2} x) > 0$$

We can classify all positive definite matrices that may or may not be symmetric or hermitian by transforming the matrix to $\frac{A+A^T}{2}$ or $\frac{A+A^H}{2}$ (which is guaranteed to be always real symmetric or complex hermitian) and we can defer most of the studies to symmetric or hermitian positive definite matrices.

But it is a different problem for the existence of Cholesky decomposition. Because even though a non symmetric or a non hermitian matrix can be positive definite, Cholesky or LDL decomposition does not exist because the decompositions require the matrix to be symmetric or hermitian.

References

[R584], [R585], [R586]

property is_positive_semidefinite

Finds out the definiteness of a matrix.

Explanation

A square real matrix A is:

- A positive definite matrix if $x^T Ax > 0$ for all non-zero real vectors x .
- A positive semidefinite matrix if $x^T Ax \geq 0$ for all non-zero real vectors x .
- A negative definite matrix if $x^T Ax < 0$ for all non-zero real vectors x .
- A negative semidefinite matrix if $x^T Ax \leq 0$ for all non-zero real vectors x .
- An indefinite matrix if there exists non-zero real vectors x, y with $x^T Ax > 0 > y^T Ay$.

A square complex matrix A is:

- A positive definite matrix if $\text{re}(x^H Ax) > 0$ for all non-zero complex vectors x .
- A positive semidefinite matrix if $\text{re}(x^H Ax) \geq 0$ for all non-zero complex vectors x .
- A negative definite matrix if $\text{re}(x^H Ax) < 0$ for all non-zero complex vectors x .
- A negative semidefinite matrix if $\text{re}(x^H Ax) \leq 0$ for all non-zero complex vectors x .
- An indefinite matrix if there exists non-zero complex vectors x, y with $\text{re}(x^H Ax) > 0 > \text{re}(y^H Ay)$.

A matrix need not be symmetric or hermitian to be positive definite.

- A real non-symmetric matrix is positive definite if and only if $\frac{A+A^T}{2}$ is positive definite.
- A complex non-hermitian matrix is positive definite if and only if $\frac{A+A^H}{2}$ is positive definite.

And this extension can apply for all the definitions above.

However, for complex cases, you can restrict the definition of $\text{re}(x^H Ax) > 0$ to $x^H Ax > 0$ and require the matrix to be hermitian. But we do not present this restriction for computation because you can check `M.is_hermitian` independently with this and use the same procedure.

Examples

An example of symmetric positive definite matrix:

```
>>> from sympy import Matrix, symbols
>>> from sympy.plotting import plot3d
>>> a, b = symbols('a b')
>>> x = Matrix([a, b])
```

```
>>> A = Matrix([[1, 0], [0, 1]])
>>> A.is_positive_definite
True
>>> A.is_positive_semidefinite
True
```

```
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```

An example of symmetric positive semidefinite matrix:

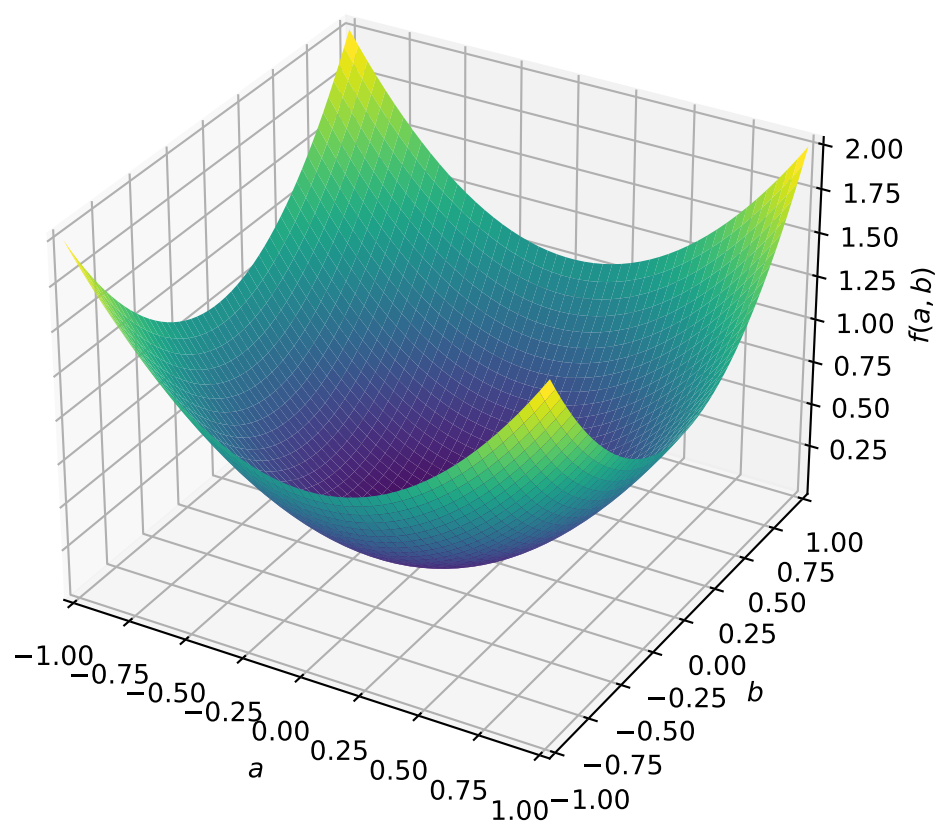
```
>>> A = Matrix([[1, -1], [-1, 1]])
>>> A.is_positive_definite
False
>>> A.is_positive_semidefinite
True
```

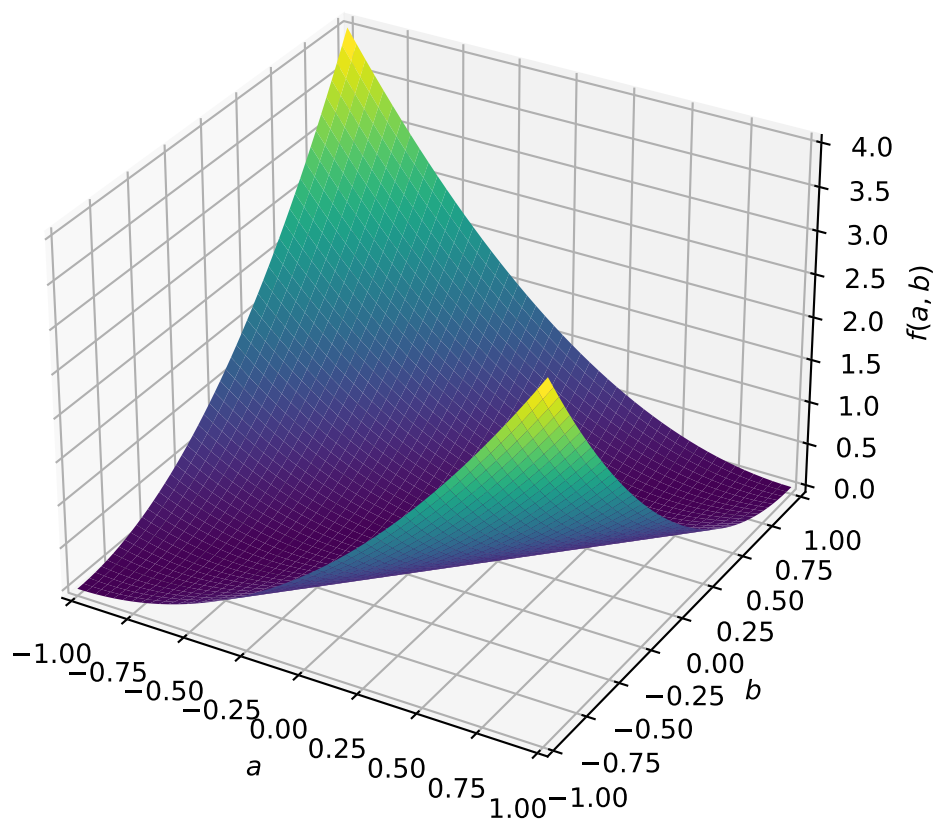
```
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```

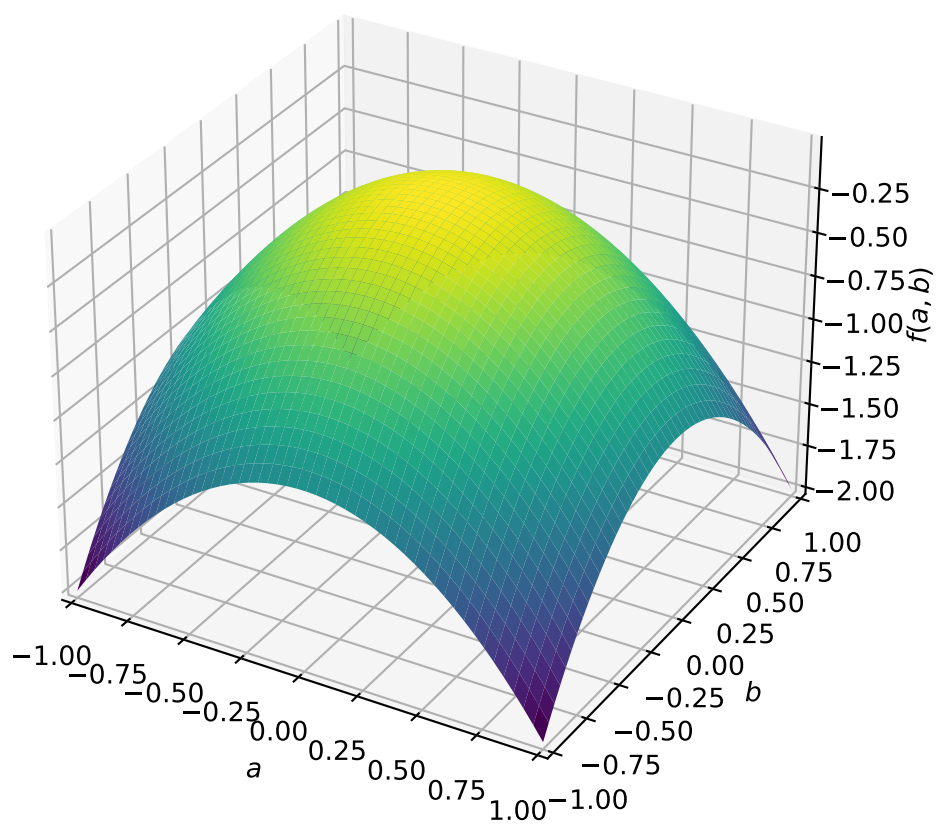
An example of symmetric negative definite matrix:

```
>>> A = Matrix([[-1, 0], [0, -1]])
>>> A.is_negative_definite
True
>>> A.is_negative_semidefinite
True
>>> A.is_indefinite
False
```

```
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```



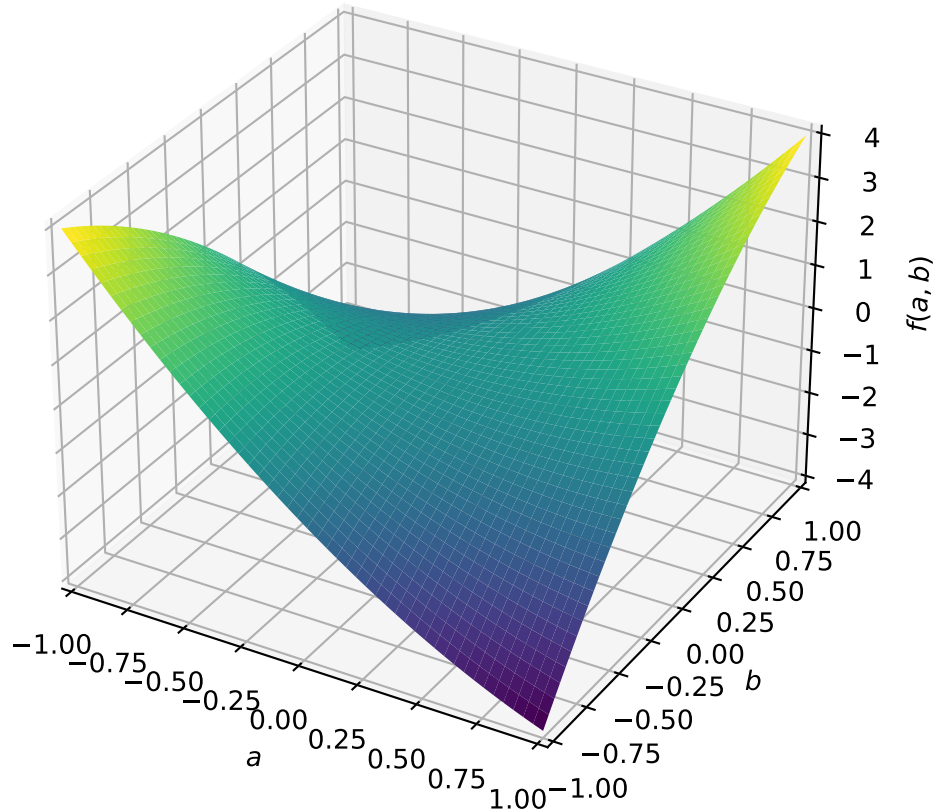




An example of symmetric indefinite matrix:

```
>>> A = Matrix([[1, 2], [2, -1]])
>>> A.is_indefinite
True
```

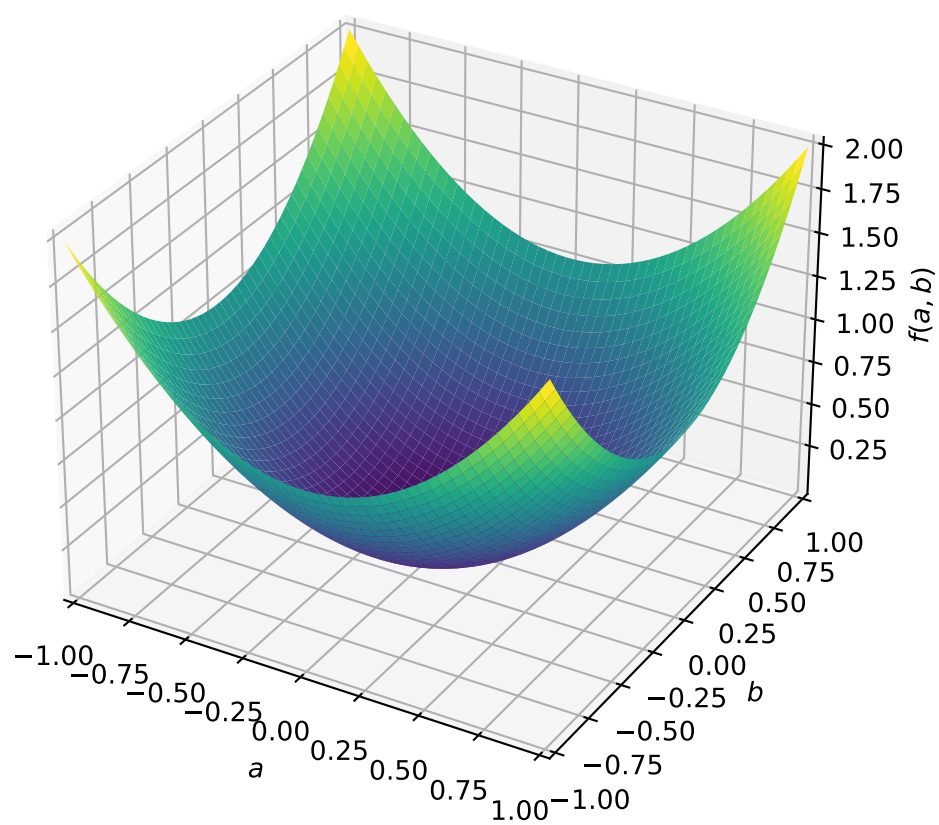
```
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```



An example of non-symmetric positive definite matrix.

```
>>> A = Matrix([[1, 2], [-2, 1]])
>>> A.is_positive_definite
True
>>> A.is_positive_semidefinite
True
```

```
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```



Notes

Although some people trivialize the definition of positive definite matrices only for symmetric or hermitian matrices, this restriction is not correct because it does not classify all instances of positive definite matrices from the definition $x^T A x > 0$ or $\operatorname{re}(x^H A x) > 0$.

For instance, `Matrix([[1, 2], [-2, 1]])` presented in the example above is an example of real positive definite matrix that is not symmetric.

However, since the following formula holds true;

$$\operatorname{re}(x^H A x) > 0 \iff \operatorname{re}\left(x^H \frac{A + A^H}{2} x\right) > 0$$

We can classify all positive definite matrices that may or may not be symmetric or hermitian by transforming the matrix to $\frac{A+A^T}{2}$ or $\frac{A+A^H}{2}$ (which is guaranteed to be always real symmetric or complex hermitian) and we can defer most of the studies to symmetric or hermitian positive definite matrices.

But it is a different problem for the existence of Cholesky decomposition. Because even though a non symmetric or a non hermitian matrix can be positive definite, Cholesky or LDL decomposition does not exist because the decompositions require the matrix to be symmetric or hermitian.

References

[R587], [R588], [R589]

jordan_form(*calc_transform=True*, **kwargs)

Return (*P*, *J*) where *J* is a Jordan block matrix and *P* is a matrix such that $M = PJP^{-1}$

Parameters

calc_transform : bool

If False, then only *J* is returned.

chop : bool

All matrices are converted to exact types when computing eigenvalues and eigenvectors. As a result, there may be approximation errors. If chop==True, these errors will be truncated.

Examples

```
>>> from sympy import Matrix
>>> M = Matrix([[ 6,  5, -2, -3], [-3, -1,  3,  3], [ 2,  1, -2, -3],
               [-1,  1,  5,  5]])
>>> P, J = M.jordan_form()
>>> J
Matrix([
[2, 1, 0, 0],
[0, 2, 0, 0],
[0, 0, 2, 1],
[0, 0, 0, 2]])
```

See also:[*jordan_block*](#) (page 1345)**left_eigenvectors(**flags)**

Returns left eigenvectors and eigenvalues.

This function returns the list of triples (eigenval, multiplicity, basis) for the left eigenvectors. Options are the same as for `eigenvectors()`, i.e. the `**flags` arguments gets passed directly to `eigenvectors()`.

Examples

```
>>> from sympy import Matrix
>>> M = Matrix([[0, 1, 1], [1, 0, 0], [1, 1, 1]])
>>> M.eigenvectors()
[(-1, 1, [Matrix([
[-1],
[ 1],
[ 0]])]), (0, 1, [Matrix([
[ 0],
[-1],
[ 1]])]), (2, 1, [Matrix([
[2/3],
[1/3],
[ 1]])])]
>>> M.left_eigenvectors()
[(-1, 1, [Matrix([[-2, 1, 1]])]), (0, 1, [Matrix([[-1, -1, 1]])]), (2,
1, [Matrix([[1, 1, 1]])])]
```

singular_values()

Compute the singular values of a Matrix

Examples

```
>>> from sympy import Matrix, Symbol
>>> x = Symbol('x', real=True)
>>> M = Matrix([[0, 1, 0], [0, x, 0], [-1, 0, 0]])
>>> M.singular_values()
[sqrt(x**2 + 1), 1, 0]
```

See also:[*condition_number*](#) (page 1293)

MatrixCalculus Class Reference

class sympy.matrices.matrices.**MatrixCalculus**

Provides calculus-related matrix operations.

diff(*args, **kwargs)

Calculate the derivative of each element in the matrix. args will be passed to the integrate function.

Examples

```
>>> from sympy import Matrix
>>> from sympy.abc import x, y
>>> M = Matrix([[x, y], [1, 0]])
>>> M.diff(x)
Matrix([
[1, 0],
[0, 0]])
```

See also:

[integrate](#) (page 1279), [limit](#) (page 1280)

integrate(*args, **kwargs)

Integrate each element of the matrix. args will be passed to the integrate function.

Examples

```
>>> from sympy import Matrix
>>> from sympy.abc import x, y
>>> M = Matrix([[x, y], [1, 0]])
>>> M.integrate((x, ))
Matrix([
[x**2/2, x*y],
[      x,  0]])
>>> M.integrate((x, 0, 2))
Matrix([
[2, 2*y],
[2,  0]])
```

See also:

[limit](#) (page 1280), [diff](#) (page 1279)

jacobian(X)

Calculates the Jacobian matrix (derivative of a vector-valued function).

Parameters

self: vector of expressions representing functions $f_i(x_1, \dots, x_n)$.

X: set of x_i 's in order, it can be a list or a Matrix

Both `self` and X can be a row or a column matrix in any order (i.e., `jacobian()` should always work).

Examples

```
>>> from sympy import sin, cos, Matrix
>>> from sympy.abc import rho, phi
>>> X = Matrix([rho*cos(phi), rho*sin(phi), rho**2])
>>> Y = Matrix([rho, phi])
>>> X.jacobian(Y)
Matrix([
[cos(phi), -rho*sin(phi)],
[sin(phi),  rho*cos(phi)],
[ 2*rho,      0]])
>>> X = Matrix([rho*cos(phi), rho*sin(phi)])
>>> X.jacobian(Y)
Matrix([
[cos(phi), -rho*sin(phi)],
[sin(phi),  rho*cos(phi)]])
```

See also:

[*hessian*](#) (page 1320), [*wronskian*](#) (page 1321)

`limit(*args)`

Calculate the limit of each element in the matrix. `args` will be passed to the `limit` function.

Examples

```
>>> from sympy import Matrix
>>> from sympy.abc import x, y
>>> M = Matrix([[x, y], [1, 0]])
>>> M.limit(x, 2)
Matrix([
[2, y],
[1, 0]])
```

See also:

[*integrate*](#) (page 1279), [*diff*](#) (page 1279)

MatrixBase Class Reference

class `sympy.matrices.matrices.MatrixBase`

Base class for matrix objects.

property `D`

Return Dirac conjugate (if `self.rows == 4`).

Examples

```
>>> from sympy import Matrix, I, eye
>>> m = Matrix((0, 1 + I, 2, 3))
>>> m.D
Matrix([[0, 1 - I, -2, -3]])
>>> m = (eye(4) + I*eye(4))
>>> m[0, 3] = 2
>>> m.D
Matrix([
[1 - I,      0,      0,      0],
[      0, 1 - I,      0,      0],
[      0,      0, -1 + I,      0],
[      2,      0,      0, -1 + I]])
```

If the matrix does not have 4 rows an `AttributeError` will be raised because this property is only defined for matrices with 4 rows.

```
>>> Matrix(eye(2)).D
Traceback (most recent call last):
...
AttributeError: Matrix has no attribute D.
```

See also:

[*sympy.matrices.common.MatrixCommon.conjugate* \(page 1330\)](#)

By-element conjugation

[*sympy.matrices.common.MatrixCommon.H* \(page 1327\)](#)

Hermite conjugation

LDLdecomposition(*hermitian=True*)

Returns the LDL Decomposition (L , D) of matrix A , such that $L * D * L.H == A$ if hermitian flag is `True`, or $L * D * L.T == A$ if hermitian is `False`. This method eliminates the use of square root. Further this ensures that all the diagonal entries of L are 1. A must be a Hermitian positive-definite matrix if hermitian is `True`, or a symmetric matrix otherwise.

Examples

```
>>> from sympy import Matrix, eye
>>> A = Matrix(((25, 15, -5), (15, 18, 0), (-5, 0, 11)))
>>> L, D = A.LDLdecomposition()
>>> L
Matrix([
[ 1,  0, 0],
[ 3/5, 1, 0],
[-1/5, 1/3, 1]])
>>> D
Matrix([
[25, 0, 0],
[ 0, 9, 0],
```

(continues on next page)

(continued from previous page)

```
[ 0, 0, 9]])
>>> L * D * L.T * A.inv() == eye(A.rows)
True
```

The matrix can have complex entries:

```
>>> from sympy import I
>>> A = Matrix(((9, 3*I), (-3*I, 5)))
>>> L, D = A.LDLdecomposition()
>>> L
Matrix([
[ 1, 0],
[-I/3, 1]])
>>> D
Matrix([
[9, 0],
[0, 4]])
>>> L*D*L.H == A
True
```

See also:

[*sympy.matrices.dense.DenseMatrix.cholesky*](#) (page 1362), [*sympy.matrices.matrices.MatrixBase.LUdecomposition*](#) (page 1282), [*QRdecomposition*](#) (page 1287)

LDLsolve(*rhs*)

Solves $Ax = B$ using LDL decomposition, for a general square and non-singular matrix.

For a non-square matrix with rows > cols, the least squares solution is returned.

Examples

```
>>> from sympy import Matrix, eye
>>> A = eye(2)*2
>>> B = Matrix([[1, 2], [3, 4]])
>>> A.LDLsolve(B) == B/2
True
```

See also:

[*sympy.matrices.dense.DenseMatrix.LDLdecomposition*](#) (page 1361), [*sympy.matrices.dense.DenseMatrix.lower_triangular_solve*](#) (page 1363), [*sympy.matrices.dense.DenseMatrix.upper_triangular_solve*](#) (page 1363), [*gauss_jordan_solve*](#) (page 1297), [*cholesky_solve*](#) (page 1292), [*diagonal_solve*](#) (page 1295), [*LUsolve*](#) (page 1287), [*QRsolve*](#) (page 1291), [*pinv_solve*](#) (page 1306)

LUdecomposition(*iszerofunc*=<function *iszero*>, *simpfunc*=None, *rankcheck*=False)

Returns (L, U, perm) where L is a lower triangular matrix with unit diagonal, U is an upper triangular matrix, and perm is a list of row swap index pairs. If A is the original matrix, then $A = (L*U).permuteBkwd(perm)$, and the row permutation matrix P such that $PA = LU$ can be computed by $P = eye(A.rows).permuteFwd(perm)$.

See documentation for `LUCombined` for details about the keyword argument `rankcheck`, `iszerofunc`, and `simpfunc`.

Parameters

rankcheck : bool, optional

Determines if this function should detect the rank deficiency of the matrix and should raise a `ValueError`.

iszerofunc : function, optional

A function which determines if a given expression is zero.

The function should be a callable that takes a single SymPy expression and returns a 3-valued boolean value `True`, `False`, or `None`.

It is internally used by the pivot searching algorithm. See the notes section for a more information about the pivot searching algorithm.

simpfunc : function or `None`, optional

A function that simplifies the input.

If this is specified as a function, this function should be a callable that takes a single SymPy expression and returns an another SymPy expression that is algebraically equivalent.

If `None`, it indicates that the pivot search algorithm should not attempt to simplify any candidate pivots.

It is internally used by the pivot searching algorithm. See the notes section for a more information about the pivot searching algorithm.

Examples

```
>>> from sympy import Matrix
>>> a = Matrix([[4, 3], [6, 3]])
>>> L, U, _ = a.LUdecomposition()
>>> L
Matrix([
[ 1, 0],
[3/2, 1]])
>>> U
Matrix([
[4, 3],
[0, -3/2]])
```

See also:

[`sympy.matrices.dense.DenseMatrix.cholesky`](#) (page 1362), [`sympy.matrices.dense.DenseMatrix.LDLdecomposition`](#) (page 1361), [`QRdecomposition`](#) (page 1287), [`LUdecomposition\_Simple`](#) (page 1284), [`LUdecompositionFF`](#) (page 1283), [`LUsolve`](#) (page 1287)

`LUdecompositionFF()`

Compute a fraction-free LU decomposition.

Returns 4 matrices `P`, `L`, `D`, `U` such that $PA = L D^{-1} U$. If the elements of the matrix belong to some integral domain `I`, then all elements of `L`, `D` and `U` are guaranteed to belong to `I`.

See also:

[`sympy.matrices.matrices.MatrixBase.LUdecomposition`](#) (page 1282),
[`LUdecomposition\_Simple`](#) (page 1284), [`LUsolve`](#) (page 1287)

References

[R590]

LUdecomposition_Simple(*iszerofunc*=<function *_iszero*>, *simpfunc*=None,
rankcheck=False)

Compute the PLU decomposition of the matrix.

Parameters

rankcheck : bool, optional

Determines if this function should detect the rank deficiency of the matrix and should raise a `ValueError`.

iszerofunc : function, optional

A function which determines if a given expression is zero.

The function should be a callable that takes a single SymPy expression and returns a 3-valued boolean value `True`, `False`, or `None`.

It is internally used by the pivot searching algorithm. See the notes section for a more information about the pivot searching algorithm.

simpfunc : function or None, optional

A function that simplifies the input.

If this is specified as a function, this function should be a callable that takes a single SymPy expression and returns an another SymPy expression that is algebraically equivalent.

If `None`, it indicates that the pivot search algorithm should not attempt to simplify any candidate pivots.

It is internally used by the pivot searching algorithm. See the notes section for a more information about the pivot searching algorithm.

Returns

(lu, row_swaps) : (Matrix, list)

If the original matrix is a m, n matrix:

lu is a m, n matrix, which contains result of the decomposition in a compressed form. See the notes section to see how the matrix is compressed.

row_swaps is a m -element list where each element is a pair of row exchange indices.

$A = (L*U).permute\_backward(perm)$, and the row permutation matrix P from the formula $PA = LU$ can be computed by $P=eye(A.row).permute\_forward(perm)$.

Raises

ValueError

Raised if rankcheck=True and the matrix is found to be rank deficient during the computation.

Notes

About the PLU decomposition:

PLU decomposition is a generalization of a LU decomposition which can be extended for rank-deficient matrices.

It can further be generalized for non-square matrices, and this is the notation that SymPy is using.

PLU decomposition is a decomposition of a m, n matrix A in the form of $PA = LU$ where

- L is a m, m **lower triangular matrix with unit diagonal** entries.
- U is a m, n upper triangular matrix.
- P is a m, m permutation matrix.

So, for a square matrix, the decomposition would look like:

$$L = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ L_{1,0} & 1 & 0 & \cdots & 0 \\ L_{2,0} & L_{2,1} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L_{n-1,0} & L_{n-1,1} & L_{n-1,2} & \cdots & 1 \end{bmatrix}$$

$$U = \begin{bmatrix} U_{0,0} & U_{0,1} & U_{0,2} & \cdots & U_{0,n-1} \\ 0 & U_{1,1} & U_{1,2} & \cdots & U_{1,n-1} \\ 0 & 0 & U_{2,2} & \cdots & U_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & U_{n-1,n-1} \end{bmatrix}$$

And for a matrix with more rows than the columns, the decomposition would look like:

$$L = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ L_{1,0} & 1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ L_{2,0} & L_{2,1} & 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ L_{n-1,0} & L_{n-1,1} & L_{n-1,2} & \cdots & 1 & 0 & \cdots & 0 \\ L_{n,0} & L_{n,1} & L_{n,2} & \cdots & L_{n,n-1} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ L_{m-1,0} & L_{m-1,1} & L_{m-1,2} & \cdots & L_{m-1,n-1} & 0 & \cdots & 1 \end{bmatrix}$$

$$U = \begin{bmatrix} U_{0,0} & U_{0,1} & U_{0,2} & \cdots & U_{0,n-1} \\ 0 & U_{1,1} & U_{1,2} & \cdots & U_{1,n-1} \\ 0 & 0 & U_{2,2} & \cdots & U_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & U_{n-1,n-1} \\ 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 \end{bmatrix}$$

Finally, for a matrix with more columns than the rows, the decomposition would look like:

$$L = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ L_{1,0} & 1 & 0 & \cdots & 0 \\ L_{2,0} & L_{2,1} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L_{m-1,0} & L_{m-1,1} & L_{m-1,2} & \cdots & 1 \end{bmatrix}$$

$$U = \begin{bmatrix} U_{0,0} & U_{0,1} & U_{0,2} & \cdots & U_{0,m-1} & \cdots & U_{0,n-1} \\ 0 & U_{1,1} & U_{1,2} & \cdots & U_{1,m-1} & \cdots & U_{1,n-1} \\ 0 & 0 & U_{2,2} & \cdots & U_{2,m-1} & \cdots & U_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \cdots & \vdots \\ 0 & 0 & 0 & \cdots & U_{m-1,m-1} & \cdots & U_{m-1,n-1} \end{bmatrix}$$

About the compressed LU storage:

The results of the decomposition are often stored in compressed forms rather than returning L and U matrices individually.

It may be less intuitive, but it is commonly used for a lot of numeric libraries because of the efficiency.

The storage matrix is defined as following for this specific method:

- **The subdiagonal elements of L are stored in the subdiagonal portion of LU , that is $LU_{i,j} = L_{i,j}$ whenever $i > j$.**
- **The elements on the diagonal of L are all 1, and are not explicitly stored.**
- **U is stored in the upper triangular portion of LU , that is $LU_{i,j} = U_{i,j}$ whenever $i \leq j$.**
- **For a case of $m > n$, the right side of the L matrix is trivial to store.**
- **For a case of $m < n$, the below side of the U matrix is trivial to store.**

So, for a square matrix, the compressed output matrix would be:

$$LU = \begin{bmatrix} U_{0,0} & U_{0,1} & U_{0,2} & \cdots & U_{0,n-1} \\ L_{1,0} & U_{1,1} & U_{1,2} & \cdots & U_{1,n-1} \\ L_{2,0} & L_{2,1} & U_{2,2} & \cdots & U_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L_{n-1,0} & L_{n-1,1} & L_{n-1,2} & \cdots & U_{n-1,n-1} \end{bmatrix}$$

For a matrix with more rows than the columns, the compressed output matrix would be:

$$LU = \begin{bmatrix} U_{0,0} & U_{0,1} & U_{0,2} & \cdots & U_{0,n-1} \\ L_{1,0} & U_{1,1} & U_{1,2} & \cdots & U_{1,n-1} \\ L_{2,0} & L_{2,1} & U_{2,2} & \cdots & U_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L_{n-1,0} & L_{n-1,1} & L_{n-1,2} & \cdots & U_{n-1,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L_{m-1,0} & L_{m-1,1} & L_{m-1,2} & \cdots & L_{m-1,n-1} \end{bmatrix}$$

For a matrix with more columns than the rows, the compressed output matrix would be:

$$LU = \begin{bmatrix} U_{0,0} & U_{0,1} & U_{0,2} & \cdots & U_{0,m-1} & \cdots & U_{0,n-1} \\ L_{1,0} & U_{1,1} & U_{1,2} & \cdots & U_{1,m-1} & \cdots & U_{1,n-1} \\ L_{2,0} & L_{2,1} & U_{2,2} & \cdots & U_{2,m-1} & \cdots & U_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \cdots & \vdots \\ L_{m-1,0} & L_{m-1,1} & L_{m-1,2} & \cdots & U_{m-1,m-1} & \cdots & U_{m-1,n-1} \end{bmatrix}$$

About the pivot searching algorithm:

When a matrix contains symbolic entries, the pivot search algorithm differs from the case where every entry can be categorized as zero or nonzero. The algorithm searches column by column through the submatrix whose top left entry coincides with the pivot position. If it exists, the pivot is the first entry in the current search column that `iszerofunc` guarantees is nonzero. If no such candidate exists, then each candidate pivot is simplified if `simpfunc` is not `None`. The search is repeated, with the difference that a candidate may be the pivot if `iszerofunc()` cannot guarantee that it is nonzero. In the second search the pivot is the first candidate that `iszerofunc` can guarantee is nonzero. If no such candidate exists, then the pivot is the first candidate for which `iszerofunc` returns `None`. If no such candidate exists, then the search is repeated in the next column to the right. The pivot search algorithm differs from the one in `rref()`, which relies on `_find_reasonable_pivot()`. Future versions of `LUdecomposition_simple()` may use `_find_reasonable_pivot()`.

See also:

[sympy.matrices.matrices.MatrixBase.LUdecomposition](#) (page 1282),
[LUdecompositionFF](#) (page 1283), [LUsolve](#) (page 1287)

LUsolve(*rhs*, *iszerofunc*=<function *_iszero*>)

Solve the linear system $Ax = rhs$ for x where $A = M$.

This is for symbolic matrices, for real or complex ones use `mpmath.lu_solve` or `mpmath.qr_solve`.

See also:

[sympy.matrices.dense.DenseMatrix.lower_triangular_solve](#) (page 1363),
[sympy.matrices.dense.DenseMatrix.upper_triangular_solve](#) (page 1363),
[gauss_jordan_solve](#) (page 1297), [cholesky_solve](#) (page 1292), [diagonal_solve](#)
 (page 1295), [LDLsolve](#) (page 1282), [QRsolve](#) (page 1291), [pinv_solve](#) (page 1306),
[LUdecomposition](#) (page 1282)

QRdecomposition()

Returns a QR decomposition.

Explanation

A QR decomposition is a decomposition in the form $A = QR$ where

- Q is a column orthogonal matrix.
- R is a upper triangular (trapezoidal) matrix.

A column orthogonal matrix satisfies $\mathbb{I} = Q^H Q$ while a full orthogonal matrix satisfies relation $\mathbb{I} = Q Q^H = Q^H Q$ where I is an identity matrix with matching dimensions.

For matrices which are not square or are rank-deficient, it is sufficient to return a column orthogonal matrix because augmenting them may introduce redundant computations. And an another advantage of this is that you can easily inspect the matrix rank by counting the number of columns of Q .

If you want to augment the results to return a full orthogonal decomposition, you should use the following procedures.

- Augment the Q matrix with columns that are orthogonal to every other columns and make it square.
- Augument the R matrix with zero rows to make it have the same shape as the original matrix.

The procedure will be illustrated in the examples section.

Examples

A full rank matrix example:

```
>>> from sympy import Matrix
>>> A = Matrix([[12, -51, 4], [6, 167, -68], [-4, 24, -41]])
>>> Q, R = A.QRdecomposition()
>>> Q
Matrix([
 [ 6/7, -69/175, -58/175],
 [ 3/7, 158/175,  6/175],
 [-2/7,  6/35, -33/35]])
>>> R
Matrix([
 [14, 21, -14],
 [ 0, 175, -70],
 [ 0,  0, 35]])
```

If the matrix is square and full rank, the Q matrix becomes orthogonal in both directions, and needs no augmentation.

```
>>> Q * Q.H
Matrix([
 [1, 0, 0],
 [0, 1, 0],
 [0, 0, 1]])
>>> Q.H * Q
Matrix([
 [1, 0, 0],
```

(continues on next page)

(continued from previous page)

```
[0, 1, 0],
[0, 0, 1]])
```

```
>>> A == Q*R
True
```

A rank deficient matrix example:

```
>>> A = Matrix([[12, -51, 0], [6, 167, 0], [-4, 24, 0]])
>>> Q, R = A.QRdecomposition()
>>> Q
Matrix([
 [ 6/7, -69/175],
 [ 3/7, 158/175],
 [-2/7,   6/35]])
>>> R
Matrix([
 [14, 21, 0],
 [ 0, 175, 0]])
```

QRdecomposition might return a matrix Q that is rectangular. In this case the orthogonality condition might be satisfied as $\mathbb{I} = Q.H*Q$ but not in the reversed product $\mathbb{I} = Q * Q.H$.

```
>>> Q.H * Q
Matrix([
 [1, 0],
 [0, 1]])
>>> Q * Q.H
Matrix([
 [27261/30625, 348/30625, -1914/6125],
 [ 348/30625, 30589/30625, 198/6125],
 [-1914/6125, 198/6125, 136/1225]])
```

If you want to augment the results to be a full orthogonal decomposition, you should augment Q with an another orthogonal column.

You are able to append an arbitrary standard basis that are linearly independent to every other columns and you can run the Gram-Schmidt process to make them augmented as orthogonal basis.

```
>>> Q_aug = Q.row_join(Matrix([0, 0, 1]))
>>> Q_aug = Q_aug.QRdecomposition()[0]
>>> Q_aug
Matrix([
 [ 6/7, -69/175, 58/175],
 [ 3/7, 158/175, -6/175],
 [-2/7,   6/35, 33/35]])
>>> Q_aug.H * Q_aug
Matrix([
 [1, 0, 0],
 [0, 1, 0],
 [0, 0, 1]])
```

(continues on next page)

(continued from previous page)

```
>>> Q_aug * Q_aug.H
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
```

Augmenting the R matrix with zero row is straightforward.

```
>>> R_aug = R.col_join(Matrix([[0, 0, 0]]))
>>> R_aug
Matrix([
[14, 21, 0],
[ 0, 175, 0],
[ 0,  0, 0]])
>>> Q_aug * R_aug == A
True
```

A zero matrix example:

```
>>> from sympy import Matrix
>>> A = Matrix.zeros(3, 4)
>>> Q, R = A.QRdecomposition()
```

They may return matrices with zero rows and columns.

```
>>> Q
Matrix(3, 0, [])
>>> R
Matrix(0, 4, [])
>>> Q*R
Matrix([
[0, 0, 0, 0],
[0, 0, 0, 0],
[0, 0, 0, 0]])
```

As the same augmentation rule described above, Q can be augmented with columns of an identity matrix and R can be augmented with rows of a zero matrix.

```
>>> Q_aug = Q.row_join(Matrix.eye(3))
>>> R_aug = R.col_join(Matrix.zeros(3, 4))
>>> Q_aug * Q_aug.T
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> R_aug
Matrix([
[0, 0, 0, 0],
[0, 0, 0, 0],
[0, 0, 0, 0]])
>>> Q_aug * R_aug == A
True
```

See also:

[*sympy.matrices.dense.DenseMatrix.cholesky*](#) (page 1362), [*sympy.matrices.dense.DenseMatrix.LDLdecomposition*](#) (page 1361), [*sympy.matrices.matrices.MatrixBase.LUdecomposition*](#) (page 1282), [*QRsolve*](#) (page 1291)

QRsolve(b)

Solve the linear system $Ax = b$.

M is the matrix A, the method argument is the vector b. The method returns the solution vector x. If b is a matrix, the system is solved for each column of b and the return value is a matrix of the same shape as b.

This method is slower (approximately by a factor of 2) but more stable for floating-point arithmetic than the LUsolve method. However, LUsolve usually uses an exact arithmetic, so you do not need to use QRsolve.

This is mainly for educational purposes and symbolic matrices, for real (or complex) matrices use `mpmath.qr_solve`.

See also:

[*sympy.matrices.dense.DenseMatrix.lower_triangular_solve*](#) (page 1363), [*sympy.matrices.dense.DenseMatrix.upper_triangular_solve*](#) (page 1363), [*gauss_jordan_solve*](#) (page 1297), [*cholesky_solve*](#) (page 1292), [*diagonal_solve*](#) (page 1295), [*LDLsolve*](#) (page 1282), [*LUsolve*](#) (page 1287), [*pinv_solve*](#) (page 1306), [*QRdecomposition*](#) (page 1287)

add(b)

Return self + b

analytic_func(f, x)

Computes $f(A)$ where A is a Square Matrix and f is an analytic function.

Parameters

f : Expr

Analytic Function

x : Symbol

parameter of f

Examples

```
>>> from sympy import Symbol, Matrix, S, log
```

```
>>> x = Symbol('x')
>>> m = Matrix([[S(5)/4, S(3)/4], [S(3)/4, S(5)/4]])
>>> f = log(x)
>>> m.analytic_func(f, x)
Matrix([
[      0, log(2)],
[log(2),      0]])
```

cholesky(*hermitian=True*)

Returns the Cholesky-type decomposition L of a matrix A such that $L * L.H == A$ if hermitian flag is True, or $L * L.T == A$ if hermitian is False.

A must be a Hermitian positive-definite matrix if hermitian is True, or a symmetric matrix if it is False.

Examples

```
>>> from sympy import Matrix
>>> A = Matrix(((25, 15, -5), (15, 18, 0), (-5, 0, 11)))
>>> A.cholesky()
Matrix([
[ 5, 0, 0],
[ 3, 3, 0],
[-1, 1, 3]])
>>> A.cholesky() * A.cholesky().T
Matrix([
[25, 15, -5],
[15, 18,  0],
[-5,  0, 11]])
```

The matrix can have complex entries:

```
>>> from sympy import I
>>> A = Matrix(((9, 3*I), (-3*I, 5)))
>>> A.cholesky()
Matrix([
[ 3, 0],
[-I, 2]])
>>> A.cholesky() * A.cholesky().H
Matrix([
[ 9, 3*I],
[-3*I,  5]])
```

Non-hermitian Cholesky-type decomposition may be useful when the matrix is not positive-definite.

```
>>> A = Matrix([[1, 2], [2, 1]])
>>> L = A.cholesky(hermitian=False)
>>> L
Matrix([
[1,      0],
[2, sqrt(3)*I]])
>>> L*L.T == A
True
```

See also:

<i>sympy.matrices.dense.DenseMatrix.LDLdecomposition</i>	(page 1361),
<i>sympy.matrices.matrices.MatrixBase.LUdecomposition</i>	(page 1282),
<i>QRdecomposition</i>	(page 1287)

cholesky_solve(*rhs*)

Solves $Ax = B$ using Cholesky decomposition, for a general square non-singular matrix. For a non-square matrix with rows > cols, the least squares solution is returned.

See also:

[sympy.matrices.dense.DenseMatrix.lower_triangular_solve](#) (page 1363), [sympy.matrices.dense.DenseMatrix.upper_triangular_solve](#) (page 1363), [gauss_jordan_solve](#) (page 1297), [diagonal_solve](#) (page 1295), [LDLsolve](#) (page 1282), [LUsolve](#) (page 1287), [QRsolve](#) (page 1291), [pinv_solve](#) (page 1306)

condition_number()

Returns the condition number of a matrix.

This is the maximum singular value divided by the minimum singular value

Examples

```
>>> from sympy import Matrix, S
>>> A = Matrix([[1, 0, 0], [0, 10, 0], [0, 0, S.One/10]])
>>> A.condition_number()
100
```

See also:

[singular_values](#) (page 1278)

connected_components()

Returns the list of connected vertices of the graph when a square matrix is viewed as a weighted graph.

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([
...     [66, 0, 0, 68, 0, 0, 0, 0, 67],
...     [0, 55, 0, 0, 0, 0, 54, 53, 0],
...     [0, 0, 0, 0, 1, 2, 0, 0, 0],
...     [86, 0, 0, 88, 0, 0, 0, 0, 87],
...     [0, 0, 10, 0, 11, 12, 0, 0, 0],
...     [0, 0, 20, 0, 21, 22, 0, 0, 0],
...     [0, 45, 0, 0, 0, 0, 44, 43, 0],
...     [0, 35, 0, 0, 0, 0, 34, 33, 0],
...     [76, 0, 0, 78, 0, 0, 0, 0, 77]])
>>> A.connected_components()
[[0, 3, 8], [1, 6, 7], [2, 4, 5]]
```

Notes

Even if any symbolic elements of the matrix can be indeterminate to be zero mathematically, this only takes the account of the structural aspect of the matrix, so they will be considered to be nonzero.

`connected_components_decomposition()`

Decomposes a square matrix into block diagonal form only using the permutations.

Returns

P, B : `PermutationMatrix`, `BlockDiagMatrix`

P is a permutation matrix for the similarity transform as in the explanation. And B is the block diagonal matrix of the result of the permutation.

If you would like to get the diagonal blocks from the `BlockDiagMatrix`, see [`get\_diag\_blocks\(\)`](#) (page 1385).

Explanation

The decomposition is in a form of $A = P^{-1}BP$ where P is a permutation matrix and B is a block diagonal matrix.

Examples

```
>>> from sympy import Matrix, pprint
>>> A = Matrix([
...     [66, 0, 0, 68, 0, 0, 0, 0, 67],
...     [0, 55, 0, 0, 0, 0, 54, 53, 0],
...     [0, 0, 0, 0, 1, 2, 0, 0, 0],
...     [86, 0, 0, 88, 0, 0, 0, 0, 87],
...     [0, 0, 10, 0, 11, 12, 0, 0, 0],
...     [0, 0, 20, 0, 21, 22, 0, 0, 0],
...     [0, 45, 0, 0, 0, 0, 44, 43, 0],
...     [0, 35, 0, 0, 0, 0, 34, 33, 0],
...     [76, 0, 0, 78, 0, 0, 0, 0, 77]])
```

```
>>> P, B = A.connected_components_decomposition()
>>> pprint(P)
PermutationMatrix((1 3)(2 8 5 7 4 6))
>>> pprint(B)
[[66  68  67]
 [
 [86  88  87]      0      0
 [
 [[76  78  77]
 [
 [          [55  54  53]
 [          [
 [      0    [45  44  43]      0
 [          [
```

(continues on next page)

(continued from previous page)

```
[
      [35  34  33]
      ]
[
      [0  1  2 ]
      [
      [10 11 12]]
      [
      [20 21 22]]
```

```
>>> P = P.as_explicit()
>>> B = B.as_explicit()
>>> P.T*B*P == A
True
```

Notes

This problem corresponds to the finding of the connected components of a graph, when a matrix is viewed as a weighted graph.

copy()

Returns the copy of a matrix.

Examples

```
>>> from sympy import Matrix
>>> A = Matrix(2, 2, [1, 2, 3, 4])
>>> A.copy()
Matrix([
[1, 2],
[3, 4]])
```

cross(b)

Return the cross product of `self` and `b` relaxing the condition of compatible dimensions: if each has 3 elements, a matrix of the same type and shape as `self` will be returned. If `b` has the same shape as `self` then common identities for the cross product (like $a \times b = -b \times a$) will hold.

Parameters

b : 3x1 or 1x3 Matrix

See also:

[`dot`](#) (page 1296), [`multiply`](#) (page 1347), [`multiply\_elementwise`](#) (page 1347)

diagonal_solve(rhs)

Solves $Ax = B$ efficiently, where A is a diagonal Matrix, with non-zero diagonal entries.

Examples

```
>>> from sympy import Matrix, eye
>>> A = eye(2)*2
>>> B = Matrix([[1, 2], [3, 4]])
>>> A.diagonal_solve(B) == B/2
True
```

See also:

[*sympy.matrices.dense.DenseMatrix.lower_triangular_solve*](#) (page 1363), [*sympy.matrices.dense.DenseMatrix.upper_triangular_solve*](#) (page 1363), [*gauss_jordan_solve*](#) (page 1297), [*cholesky_solve*](#) (page 1292), [*LDLsolve*](#) (page 1282), [*LUsolve*](#) (page 1287), [*QRsolve*](#) (page 1291), [*pinv_solve*](#) (page 1306)

dot(*b*, *hermitian*=None, *conjugate_convention*=None)

Return the dot or inner product of two vectors of equal length. Here *self* must be a Matrix of size 1 x n or n x 1, and *b* must be either a matrix of size 1 x n, n x 1, or a list/tuple of length n. A scalar is returned.

By default, dot does not conjugate *self* or *b*, even if there are complex entries. Set *hermitian*=True (and optionally a *conjugate_convention*) to compute the hermitian inner product.

Possible kwargs are *hermitian* and *conjugate_convention*.

If *conjugate_convention* is "left", "math" or "maths", the conjugate of the first vector (*self*) is used. If "right" or "physics" is specified, the conjugate of the second vector *b* is used.

Examples

```
>>> from sympy import Matrix
>>> M = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> v = Matrix([1, 1, 1])
>>> M.row(0).dot(v)
6
>>> M.col(0).dot(v)
12
>>> v = [3, 2, 1]
>>> M.row(0).dot(v)
10
```

```
>>> from sympy import I
>>> q = Matrix([1*I, 1*I, 1*I])
>>> q.dot(q, hermitian=False)
-3
```

```
>>> q.dot(q, hermitian=True)
3
```

```
>>> q1 = Matrix([1, 1, 1*I])
>>> q.dot(q1, hermitian=True, conjugate_convention="maths")
```

(continues on next page)

(continued from previous page)

```

1 - 2*I
>>> q.dot(q1, hermitian=True, conjugate_convention="physics")
1 + 2*I

```

See also:

[cross](#) (page 1295), [multiply](#) (page 1347), [multiply_elementwise](#) (page 1347)

dual()

Returns the dual of a matrix, which is:

$(1/2) * \text{levicivita}(i, j, k, l) * M(k, l)$ summed over indices k and l

Since the levicivita method is anti_symmetric for any pairwise exchange of indices, the dual of a symmetric matrix is the zero matrix. Strictly speaking the dual defined here assumes that the 'matrix' M is a contravariant anti_symmetric second rank tensor, so that the dual is a covariant second rank tensor.

exp()

Return the exponential of a square matrix

Examples

```
>>> from sympy import Symbol, Matrix
```

```

>>> t = Symbol('t')
>>> m = Matrix([[0, 1], [-1, 0]]) * t
>>> m.exp()
Matrix([
[ exp(I*t)/2 + exp(-I*t)/2, -I*exp(I*t)/2 + I*exp(-I*t)/2],
[ I*exp(I*t)/2 - I*exp(-I*t)/2, exp(I*t)/2 + exp(-I*t)/2]])

```

gauss_jordan_solve(B , freevar=False)

Solves $Ax = B$ using Gauss Jordan elimination.

There may be zero, one, or infinite solutions. If one solution exists, it will be returned. If infinite solutions exist, it will be returned parametrically. If no solutions exist, It will throw ValueError.

Parameters

B : Matrix

The right hand side of the equation to be solved for. Must have the same number of rows as matrix A.

freevar : boolean, optional

Flag, when set to *True* will return the indices of the free variables in the solutions (column Matrix), for a system that is undetermined (e.g. A has more columns than rows), for which infinite solutions are possible, in terms of arbitrary values of free variables. Default *False*.

Returns

x : Matrix

The matrix that will satisfy $Ax = B$. Will have as many rows as matrix A has columns, and as many columns as matrix B.

params : Matrix

If the system is underdetermined (e.g. A has more columns than rows), infinite solutions are possible, in terms of arbitrary parameters. These arbitrary parameters are returned as params Matrix.

free_var_index : List, optional

If the system is underdetermined (e.g. A has more columns than rows), infinite solutions are possible, in terms of arbitrary values of free variables. Then the indices of the free variables in the solutions (column Matrix) are returned by free_var_index, if the flag *freevar* is set to *True*.

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([[1, 2, 1, 1], [1, 2, 2, -1], [2, 4, 0, 6]])
>>> B = Matrix([7, 12, 4])
>>> sol, params = A.gauss_jordan_solve(B)
>>> sol
Matrix([
[-2*tau0 - 3*tau1 + 2],
[          tau0],
[        2*tau1 + 5],
[          tau1]])
>>> params
Matrix([
[tau0],
[tau1]])
>>> taus_zeroes = { tau:0 for tau in params }
>>> sol_unique = sol.xreplace(taus_zeroes)
>>> sol_unique
Matrix([
[2],
[0],
[5],
[0]])
```

```
>>> A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 10]])
>>> B = Matrix([3, 6, 9])
>>> sol, params = A.gauss_jordan_solve(B)
>>> sol
Matrix([
[-1],
[ 2],
[ 0]])
>>> params
Matrix(0, 1, [])
```

```
>>> A = Matrix([[2, -7], [-1, 4]])
>>> B = Matrix([[-21, 3], [12, -2]])
>>> sol, params = A.gauss_jordan_solve(B)
>>> sol
Matrix([
[0, -2],
[3, -1]])
>>> params
Matrix(0, 2, [])
```

```
>>> from sympy import Matrix
>>> A = Matrix([[1, 2, 1, 1], [1, 2, 2, -1], [2, 4, 0, 6]])
>>> B = Matrix([7, 12, 4])
>>> sol, params, freevars = A.gauss_jordan_solve(B, freevar=True)
>>> sol
Matrix([
[-2*tau0 - 3*tau1 + 2],
[tau0],
[2*tau1 + 5],
[tau1]])
>>> params
Matrix([
[tau0],
[tau1]])
>>> freevars
[1, 3]
```

See also:

[sympy.matrices.dense.DenseMatrix.lower_triangular_solve](#) (page 1363),
[sympy.matrices.dense.DenseMatrix.upper_triangular_solve](#) (page 1363),
[cholesky_solve](#) (page 1292), [diagonal_solve](#) (page 1295), [LDLsolve](#) (page 1282),
[LUsolve](#) (page 1287), [QRsolve](#) (page 1291), [pinv](#) (page 1305)

References

[R591]

inv(*method=None*, *iszerofunc=<function _iszero>*, *try_block_diag=False*)

Return the inverse of a matrix using the method indicated. Default for dense matrices is Gauss elimination, default for sparse matrices is LDL.

Parameters

method : ('GE', 'LU', 'ADJ', 'CH', 'LDL')

iszerofunc : function, optional

Zero-testing function to use.

try_block_diag : bool, optional

If True then will try to form block diagonal matrices using the method `get_diag_blocks()`, invert these individually, and then reconstruct the full inverse matrix.

Raises
ValueError

If the determinant of the matrix is zero.

Examples

```
>>> from sympy import SparseMatrix, Matrix
>>> A = SparseMatrix([
... [ 2, -1,  0],
... [-1,  2, -1],
... [ 0,  0,  2]])
>>> A.inv('CH')
Matrix([
[2/3, 1/3, 1/6],
[1/3, 2/3, 1/3],
[ 0,   0, 1/2]])
>>> A.inv(method='LDL') # use of 'method=' is optional
Matrix([
[2/3, 1/3, 1/6],
[1/3, 2/3, 1/3],
[ 0,   0, 1/2]])
>>> A *
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> A = Matrix(A)
>>> A.inv('CH')
Matrix([
[2/3, 1/3, 1/6],
[1/3, 2/3, 1/3],
[ 0,   0, 1/2]])
>>> A.inv('ADJ') == A.inv('GE') == A.inv('LU') == A.inv('CH') == A.
inv('LDL') == A.inv('QR')
True
```

Notes

According to the method keyword, it calls the appropriate method:

GE inverse_GE(); default for dense matrices LU inverse_LU() ADJ ...
inverse_ADJ() CH ... inverse_CH() LDL ... inverse_LDL(); default for sparse
matrices QR ... inverse_QR()

Note, the GE and LU methods may require the matrix to be simplified before it is inverted in order to properly detect zeros during pivoting. In difficult cases a custom zero detection function can be provided by setting the `iszerofunc` argument to a function that should return True if its argument is zero. The ADJ routine computes the determinant and uses that to detect singular matrices in addition to testing for zeros on the diagonal.

See also:

[inverse_ADJ](#) (page 1301), [inverse_GE](#) (page 1301), [inverse_LU](#) (page 1302), [inverse_CH](#) (page 1301), [inverse_LDL](#) (page 1301)

inv_mod(*m*)

Returns the inverse of the matrix K (mod m), if it exists.

Method to find the matrix inverse of K (mod m) implemented in this function:

- Compute $\text{adj}(K) = \text{cof}(K)^t$, the adjoint matrix of K .
- Compute $r = 1/\det(K)$ (mod m).
- $K^{-1} = r \cdot \text{adj}(K)$ (mod m).

Examples

```
>>> from sympy import Matrix
>>> A = Matrix(2, 2, [1, 2, 3, 4])
>>> A.inv_mod(5)
Matrix([
[3, 1],
[4, 2]])
>>> A.inv_mod(3)
Matrix([
[1, 1],
[0, 1]])
```

inverse_ADJ(*iszerofunc*=<function _iszero>)

Calculates the inverse using the adjugate matrix and a determinant.

See also:

[inv](#) (page 1299), [inverse_GE](#) (page 1301), [inverse_LU](#) (page 1302), [inverse_CH](#) (page 1301), [inverse_LDL](#) (page 1301)

inverse_BLOCK(*iszerofunc*=<function _iszero>)

Calculates the inverse using BLOCKWISE inversion.

See also:

[inv](#) (page 1299), [inverse_ADJ](#) (page 1301), [inverse_GE](#) (page 1301), [inverse_CH](#) (page 1301), [inverse_LDL](#) (page 1301)

inverse_CH(*iszerofunc*=<function _iszero>)

Calculates the inverse using cholesky decomposition.

See also:

[inv](#) (page 1299), [inverse_ADJ](#) (page 1301), [inverse_GE](#) (page 1301), [inverse_LU](#) (page 1302), [inverse_LDL](#) (page 1301)

inverse_GE(*iszerofunc*=<function _iszero>)

Calculates the inverse using Gaussian elimination.

See also:

[inv](#) (page 1299), [inverse_ADJ](#) (page 1301), [inverse_LU](#) (page 1302), [inverse_CH](#) (page 1301), [inverse_LDL](#) (page 1301)

inverse_LDL(*iszerofunc*=<function *_iszero*>)

Calculates the inverse using LDL decomposition.

See also:

[*inv*](#) (page 1299), [*inverse_ADJ*](#) (page 1301), [*inverse_GE*](#) (page 1301), [*inverse_LU*](#) (page 1302), [*inverse_CH*](#) (page 1301)

inverse_LU(*iszerofunc*=<function *_iszero*>)

Calculates the inverse using LU decomposition.

See also:

[*inv*](#) (page 1299), [*inverse_ADJ*](#) (page 1301), [*inverse_GE*](#) (page 1301), [*inverse_CH*](#) (page 1301), [*inverse_LDL*](#) (page 1301)

inverse_QR(*iszerofunc*=<function *_iszero*>)

Calculates the inverse using QR decomposition.

See also:

[*inv*](#) (page 1299), [*inverse_ADJ*](#) (page 1301), [*inverse_GE*](#) (page 1301), [*inverse_CH*](#) (page 1301), [*inverse_LDL*](#) (page 1301)

classmethod irregular(*ntop*, **matrices*, ***kwargs*)

Return a matrix filled by the given matrices which are listed in order of appearance from left to right, top to bottom as they first appear in the matrix. They must fill the matrix completely.

Examples

```
>>> from sympy import ones, Matrix
>>> Matrix.irregular(3, ones(2,1), ones(3,3)*2, ones(2,2)*3,
...   ones(1,1)*4, ones(2,2)*5, ones(1,2)*6, ones(1,2)*7)
Matrix([
  [1, 2, 2, 2, 3, 3],
  [1, 2, 2, 2, 3, 3],
  [4, 2, 2, 2, 5, 5],
  [6, 6, 7, 7, 5, 5]])
```

is_nilpotent()

Checks if a matrix is nilpotent.

A matrix *B* is nilpotent if for some integer *k*, *B****k* is a zero matrix.

Examples

```
>>> from sympy import Matrix
>>> a = Matrix([[0, 0, 0], [1, 0, 0], [1, 1, 0]])
>>> a.is_nilpotent()
True
```

```
>>> a = Matrix([[1, 0, 1], [1, 0, 0], [1, 1, 0]])
>>> a.is_nilpotent()
False
```

key2bounds(*keys*)

Converts a key with potentially mixed types of keys (integer and slice) into a tuple of ranges and raises an error if any index is out of `self`'s range.

See also:

[`key2ij`](#) (page 1303)

key2ij(*key*)

Converts key into canonical form, converting integers or indexable items into valid integers for `self`'s range or returning slices unchanged.

See also:

[`key2bounds`](#) (page 1302)

log(*simplify*=<function cancel>)

Return the logarithm of a square matrix

Parameters

simplify : function, bool

The function to simplify the result with.

Default is `cancel`, which is effective to reduce the expression growing for taking reciprocals and inverses for symbolic matrices.

Examples

```
>>> from sympy import S, Matrix
```

Examples for positive-definite matrices:

```
>>> m = Matrix([[1, 1], [0, 1]])
>>> m.log()
Matrix([
[0, 1],
[0, 0]])
```

```
>>> m = Matrix([[S(5)/4, S(3)/4], [S(3)/4, S(5)/4]])
>>> m.log()
Matrix([
[0, log(2)],
[log(2), 0]])
```

Examples for non positive-definite matrices:

```
>>> m = Matrix([[S(3)/4, S(5)/4], [S(5)/4, S(3)/4]])
>>> m.log()
Matrix([
[ I*pi/2, log(2) - I*pi/2],
[log(2) - I*pi/2, I*pi/2]])
```

```
>>> m = Matrix(
...     [[0, 0, 0, 1],
```

(continues on next page)

(continued from previous page)

```

...      [0, 0, 1, 0],
...      [0, 1, 0, 0],
...      [1, 0, 0, 0]])
>>> m.log()
Matrix([
[ I*pi/2,      0,      0, -I*pi/2],
[      0,  I*pi/2, -I*pi/2,      0],
[      0, -I*pi/2,  I*pi/2,      0],
[-I*pi/2,      0,      0,  I*pi/2]])

```

lower_triangular_solve(*rhs*)

Solves $Ax = B$, where A is a lower triangular matrix.

See also:

[upper_triangular_solve](#) (page 1318), [gauss_jordan_solve](#) (page 1297), [cholesky_solve](#) (page 1292), [diagonal_solve](#) (page 1295), [LDLsolve](#) (page 1282), [LUsolve](#) (page 1287), [QRsolve](#) (page 1291), [pinv_solve](#) (page 1306)

norm(*ord=None*)

Return the Norm of a Matrix or Vector. In the simplest case this is the geometric size of the vector Other norms can be specified by the *ord* parameter

ord	norm for matrices	norm for vectors
None	Frobenius norm	2-norm
'fro'	Frobenius norm	• does not exist
inf	maximum row sum	max(abs(x))
-inf	-	min(abs(x))
1	maximum column sum	as below
-1	-	as below
2	2-norm (largest sing. value)	as below
-2	smallest singular value	as below
other	• does not exist	sum(abs(x)**ord)**(1./ord)

Examples

```

>>> from sympy import Matrix, Symbol, trigsimp, cos, sin, oo
>>> x = Symbol('x', real=True)
>>> v = Matrix([cos(x), sin(x)])
>>> trigsimp( v.norm() )
1
>>> v.norm(10)
(sin(x)**10 + cos(x)**10)**(1/10)
>>> A = Matrix([[1, 1], [1, 1]])
>>> A.norm(1) # maximum sum of absolute values of A is 2

```

(continues on next page)

(continued from previous page)

```

2
>>> A.norm(2) # Spectral norm (max of |Ax|/|x| under 2-vector-norm)
2
>>> A.norm(-2) # Inverse spectral norm (smallest singular value)
0
>>> A.norm() # Frobenius Norm
2
>>> A.norm(oo) # Infinity Norm
2
>>> Matrix([1, -2]).norm(oo)
2
>>> Matrix([-1, 2]).norm(-oo)
1

```

See also:[*normalized*](#) (page 1305)**normalized**(iszerofunc=<function _iszero>)

Return the normalized version of self.

Parameters**iszerofunc** : Function, optional

A function to determine whether self is a zero vector. The default `_iszero` tests to see if each element is exactly zero.

Returns

Matrix

Normalized vector form of self. It has the same length as a unit vector. However, a zero vector will be returned for a vector with norm 0.

Raises**ShapeError**

If the matrix is not in a vector form.

See also:[*norm*](#) (page 1304)**pinv**(method='RD')

Calculate the Moore-Penrose pseudoinverse of the matrix.

The Moore-Penrose pseudoinverse exists and is unique for any matrix. If the matrix is invertible, the pseudoinverse is the same as the inverse.

Parameters**method** : String, optional

Specifies the method for computing the pseudoinverse.

If 'RD', Rank-Decomposition will be used.

If 'ED', Diagonalization will be used.

Examples

Computing pseudoinverse by rank decomposition :

```
>>> from sympy import Matrix
>>> A = Matrix([[1, 2, 3], [4, 5, 6]])
>>> A.pinv()
Matrix([
[-17/18,  4/9],
[ -1/9,  1/9],
[ 13/18, -2/9]])
```

Computing pseudoinverse by diagonalization :

```
>>> B = A.pinv(method='ED')
>>> B.simplify()
>>> B
Matrix([
[-17/18,  4/9],
[ -1/9,  1/9],
[ 13/18, -2/9]])
```

See also:

[*inv*](#) (page 1299), [*pinv_solve*](#) (page 1306)

References

[R592]

`pinv_solve(B, arbitrary_matrix=None)`

Solve $Ax = B$ using the Moore-Penrose pseudoinverse.

There may be zero, one, or infinite solutions. If one solution exists, it will be returned. If infinite solutions exist, one will be returned based on the value of `arbitrary_matrix`. If no solutions exist, the least-squares solution is returned.

Parameters

B : Matrix

The right hand side of the equation to be solved for. Must have the same number of rows as matrix A.

arbitrary_matrix : Matrix

If the system is underdetermined (e.g. A has more columns than rows), infinite solutions are possible, in terms of an arbitrary matrix. This parameter may be set to a specific matrix to use for that purpose; if so, it must be the same shape as x, with as many rows as matrix A has columns, and as many columns as matrix B. If left as None, an appropriate matrix containing dummy symbols in the form of `wn_m` will be used, with n and m being row and column position of each symbol.

Returns

x : Matrix

The matrix that will satisfy $Ax = B$. Will have as many rows as matrix A has columns, and as many columns as matrix B.

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([[1, 2, 3], [4, 5, 6]])
>>> B = Matrix([7, 8])
>>> A.pinv_solve(B)
Matrix([
[ _w0_0/6 - _w1_0/3 + _w2_0/6 - 55/18],
[ -_w0_0/3 + 2*_w1_0/3 - _w2_0/3 + 1/9],
[ _w0_0/6 - _w1_0/3 + _w2_0/6 + 59/18]])
>>> A.pinv_solve(B, arbitrary_matrix=Matrix([0, 0, 0]))
Matrix([
[ -55/18],
[  1/9],
[ 59/18]])
```

Notes

This may return either exact solutions or least squares solutions. To determine which, check `A * A.pinv() * B == B`. It will be True if exact solutions exist, and False if only a least-squares solution exists. Be aware that the left hand side of that equation may need to be simplified to correctly compare to the right hand side.

See also:

[sympy.matrices.dense.DenseMatrix.lower_triangular_solve](#) (page 1363),
[sympy.matrices.dense.DenseMatrix.upper_triangular_solve](#) (page 1363),
[gauss_jordan_solve](#) (page 1297), [cholesky_solve](#) (page 1292), [diagonal_solve](#)
 (page 1295), [LDLsolve](#) (page 1282), [LUsolve](#) (page 1287), [QRsolve](#) (page 1291),
[pinv](#) (page 1305)

References

[R593]

`print_nonzero(symb='X')`

Shows location of non-zero entries for fast shape lookup.

Examples

```
>>> from sympy import Matrix, eye
>>> m = Matrix(2, 3, lambda i, j: i*3+j)
>>> m
Matrix([
[0, 1, 2],
[3, 4, 5]])
>>> m.print_nonzero()
[ XX]
[XXX]
>>> m = eye(4)
```

(continues on next page)

(continued from previous page)

```
>>> m.print_nonzero("x")
[x ]
[ x ]
[  x ]
[   x]
```

project(v)

Return the projection of self onto the line containing v.

Examples

```
>>> from sympy import Matrix, S, sqrt
>>> V = Matrix([sqrt(3)/2, S.Half])
>>> x = Matrix([[1, 0]])
>>> V.project(x)
Matrix([[sqrt(3)/2, 0]])
>>> V.project(-x)
Matrix([[sqrt(3)/2, 0]])
```

rank_decomposition(iszerofunc=<function _iszero>, simplify=False)

Returns a pair of matrices (C , F) with matching rank such that $A = CF$.

Parameters

iszerofunc : Function, optional

A function used for detecting whether an element can act as a pivot.
lambda x: x.is_zero is used by default.

simplify : Bool or Function, optional

A function used to simplify elements when looking for a pivot. By default SymPy's simplify is used.

Returns

(C, F) : Matrices

C and F are full-rank matrices with rank as same as A , whose product gives A .

See Notes for additional mathematical details.

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([
...     [1, 3, 1, 4],
...     [2, 7, 3, 9],
...     [1, 5, 3, 1],
...     [1, 2, 0, 8]
... ])
>>> C, F = A.rank_decomposition()
>>> C
```

(continues on next page)

(continued from previous page)

```

Matrix([
[1, 3, 4],
[2, 7, 9],
[1, 5, 1],
[1, 2, 8]])
>>> F
Matrix([
[1, 0, -2, 0],
[0, 1, 1, 0],
[0, 0, 0, 1]])
>>> C * F == A
True

```

Notes

Obtaining F , an RREF of A , is equivalent to creating a product

$$E_n E_{n-1} \dots E_1 A = F$$

where $E_n, E_{n-1}, \dots, E_1$ are the elimination matrices or permutation matrices equivalent to each row-reduction step.

The inverse of the same product of elimination matrices gives C :

$$C = (E_n E_{n-1} \dots E_1)^{-1}$$

It is not necessary, however, to actually compute the inverse: the columns of C are those from the original matrix with the same column indices as the indices of the pivot columns of F .

See also:

[`sympy.matrices.matrices.MatrixReductions.rref`](#) (page 1234)

References

[R594], [R595]

`singular_value_decomposition()`

Returns a Condensed Singular Value decomposition.

Explanation

A Singular Value decomposition is a decomposition in the form $A = U\Sigma V$ where

- U, V are column orthogonal matrix.
- Σ is a diagonal matrix, where the main diagonal contains singular values of matrix A .

A column orthogonal matrix satisfies $\mathbb{I} = U^H U$ while a full orthogonal matrix satisfies relation $\mathbb{I} = U U^H = U^H U$ where $\mathbb{I}$ is an identity matrix with matching dimensions.

For matrices which are not square or are rank-deficient, it is sufficient to return a column orthogonal matrix because augmenting them may introduce redundant computations. In condensed Singular Value Decomposition we only return column orthogonal matrices because of this reason

If you want to augment the results to return a full orthogonal decomposition, you should use the following procedures.

- Augment the U, V matrices with columns that are orthogonal to every other columns and make it square.
- Augment the Σ matrix with zero rows to make it have the same shape as the original matrix.

The procedure will be illustrated in the examples section.

Examples

we take a full rank matrix first:

```
>>> from sympy import Matrix
>>> A = Matrix([[1, 2],[2,1]])
>>> U, S, V = A.singular_value_decomposition()
>>> U
Matrix([
[ sqrt(2)/2, sqrt(2)/2],
[-sqrt(2)/2, sqrt(2)/2]])
>>> S
Matrix([
[1, 0],
[0, 3]])
>>> V
Matrix([
[-sqrt(2)/2, sqrt(2)/2],
[ sqrt(2)/2, sqrt(2)/2]])
```

If a matrix is square and full rank both U, V are orthogonal in both directions

```
>>> U * U.H
Matrix([
[1, 0],
[0, 1]])
>>> U.H * U
Matrix([
[1, 0],
[0, 1]])
```

```
>>> V * V.H
Matrix([
[1, 0],
[0, 1]])
>>> V.H * V
```

(continues on next page)

(continued from previous page)

```
Matrix([
[1, 0],
[0, 1]])
>>> A == U * S * V.H
True
```

```
>>> C = Matrix([
...     [1, 0, 0, 0, 2],
...     [0, 0, 3, 0, 0],
...     [0, 0, 0, 0, 0],
...     [0, 2, 0, 0, 0],
... ])
>>> U, S, V = C.singular_value_decomposition()
```

```
>>> V.H * V
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> V * V.H
Matrix([
[1/5, 0, 0, 0, 2/5],
[ 0, 1, 0, 0,  0],
[ 0, 0, 1, 0,  0],
[ 0, 0, 0, 0,  0],
[2/5, 0, 0, 0, 4/5]])
```

If you want to augment the results to be a full orthogonal decomposition, you should augment V with an another orthogonal column.

You are able to append an arbitrary standard basis that are linearly independent to every other columns and you can run the Gram-Schmidt process to make them augmented as orthogonal basis.

```
>>> V_aug = V.row_join(Matrix([[0,0,0,0,1],
... [0,0,0,1,0]]).H)
>>> V_aug = V_aug.QRdecomposition()[0]
>>> V_aug
Matrix([
[0, sqrt(5)/5, 0, -2*sqrt(5)/5, 0],
[1, 0, 0, 0, 0],
[0, 0, 1, 0, 0],
[0, 0, 0, 0, 1],
[0, 2*sqrt(5)/5, 0, sqrt(5)/5, 0]])
>>> V_aug.H * V_aug
Matrix([
[1, 0, 0, 0, 0],
[0, 1, 0, 0, 0],
[0, 0, 1, 0, 0],
[0, 0, 0, 1, 0],
[0, 0, 0, 0, 1]])
>>> V_aug * V_aug.H
```

(continues on next page)

(continued from previous page)

```
Matrix([
[1, 0, 0, 0, 0],
[0, 1, 0, 0, 0],
[0, 0, 1, 0, 0],
[0, 0, 0, 1, 0],
[0, 0, 0, 0, 1]])
```

Similarly we augment U

```
>>> U_aug = U.row_join(Matrix([0,0,1,0]))
>>> U_aug = U_aug.QRdecomposition()[0]
>>> U_aug
Matrix([
[0, 1, 0, 0],
[0, 0, 1, 0],
[0, 0, 0, 1],
[1, 0, 0, 0]])
```

```
>>> U_aug.H * U_aug
Matrix([
[1, 0, 0, 0],
[0, 1, 0, 0],
[0, 0, 1, 0],
[0, 0, 0, 1]])
>>> U_aug * U_aug.H
Matrix([
[1, 0, 0, 0],
[0, 1, 0, 0],
[0, 0, 1, 0],
[0, 0, 0, 1]])
```

We add 2 zero columns and one row to S

```
>>> S_aug = S.col_join(Matrix([[0,0,0]]))
>>> S_aug = S_aug.row_join(Matrix([[0,0,0,0],
... [0,0,0,0]]).H)
>>> S_aug
Matrix([
[2,      0, 0, 0, 0],
[0, sqrt(5), 0, 0, 0],
[0,      0, 3, 0, 0],
[0,      0, 0, 0, 0]])
```

```
>>> U_aug * S_aug * V_aug.H == C
True
```

solve(*rhs*, *method*='GJ')

Solves linear equation where the unique solution exists.

Parameters

rhs : Matrix

Vector representing the right hand side of the linear equation.

method : string, optional

If set to 'GJ' or 'GE', the Gauss-Jordan elimination will be used, which is implemented in the routine `gauss_jordan_solve`.

If set to 'LU', `LUsolve` routine will be used.

If set to 'QR', `QRsolve` routine will be used.

If set to 'PINV', `pinv_solve` routine will be used.

It also supports the methods available for special linear systems

For positive definite systems:

If set to 'CH', `cholesky_solve` routine will be used.

If set to 'LDL', `LDLsolve` routine will be used.

To use a different method and to compute the solution via the inverse, use a method defined in the `.inv()` docstring.

Returns

solutions : Matrix

Vector representing the solution.

Raises

ValueError

If there is not a unique solution then a `ValueError` will be raised.

If `M` is not square, a `ValueError` and a different routine for solving the system will be suggested.

`solve_least_squares(rhs, method='CH')`

Return the least-square fit to the data.

Parameters

rhs : Matrix

Vector representing the right hand side of the linear equation.

method : string or boolean, optional

If set to 'CH', `cholesky_solve` routine will be used.

If set to 'LDL', `LDLsolve` routine will be used.

If set to 'QR', `QRsolve` routine will be used.

If set to 'PINV', `pinv_solve` routine will be used.

Otherwise, the conjugate of `M` will be used to create a system of equations that is passed to `solve` along with the hint defined by `method`.

Returns

solutions : Matrix

Vector representing the solution.

Examples

```
>>> from sympy import Matrix, ones
>>> A = Matrix([1, 2, 3])
>>> B = Matrix([2, 3, 4])
>>> S = Matrix(A.row_join(B))
>>> S
Matrix([
[1, 2],
[2, 3],
[3, 4]])
```

If each line of S represent coefficients of $Ax + By$ and x and y are $[2, 3]$ then $S*xy$ is:

```
>>> r = S*Matrix([2, 3]); r
Matrix([
[ 8],
[13],
[18]])
```

But let's add 1 to the middle value and then solve for the least-squares value of xy :

```
>>> xy = S.solve_least_squares(Matrix([8, 14, 18])); xy
Matrix([
[ 5/3],
[10/3]])
```

The error is given by $S*xy - r$:

```
>>> S*xy - r
Matrix([
[1/3],
[1/3],
[1/3]])
>>> _.norm().n(2)
0.58
```

If a different xy is used, the norm will be higher:

```
>>> xy += ones(2, 1)/10
>>> (S*xy - r).norm().n(2)
1.5
```

strongly_connected_components()

Returns the list of strongly connected vertices of the graph when a square matrix is viewed as a weighted graph.

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([
...     [44, 0, 0, 0, 43, 0, 45, 0, 0],
...     [0, 66, 62, 61, 0, 68, 0, 60, 67],
...     [0, 0, 22, 21, 0, 0, 0, 20, 0],
...     [0, 0, 12, 11, 0, 0, 0, 10, 0],
...     [34, 0, 0, 0, 33, 0, 35, 0, 0],
...     [0, 86, 82, 81, 0, 88, 0, 80, 87],
...     [54, 0, 0, 0, 53, 0, 55, 0, 0],
...     [0, 0, 2, 1, 0, 0, 0, 0, 0],
...     [0, 76, 72, 71, 0, 78, 0, 70, 77]])
>>> A.strongly_connected_components()
[[0, 4, 6], [2, 3, 7], [1, 5, 8]]
```

`strongly_connected_components_decomposition(lower=True)`

Decomposes a square matrix into block triangular form only using the permutations.

Parameters

lower : bool

Makes B lower block triangular when True. Otherwise, makes B upper block triangular.

Returns

P, B : PermutationMatrix, BlockMatrix

P is a permutation matrix for the similarity transform as in the explanation. And B is the block triangular matrix of the result of the permutation.

Explanation

The decomposition is in a form of $A = P^{-1}BP$ where P is a permutation matrix and B is a block diagonal matrix.

Examples

```
>>> from sympy import Matrix, pprint
>>> A = Matrix([
...     [44, 0, 0, 0, 43, 0, 45, 0, 0],
...     [0, 66, 62, 61, 0, 68, 0, 60, 67],
...     [0, 0, 22, 21, 0, 0, 0, 20, 0],
...     [0, 0, 12, 11, 0, 0, 0, 10, 0],
...     [34, 0, 0, 0, 33, 0, 35, 0, 0],
...     [0, 86, 82, 81, 0, 88, 0, 80, 87],
...     [54, 0, 0, 0, 53, 0, 55, 0, 0],
...     [0, 0, 2, 1, 0, 0, 0, 0, 0],
...     [0, 76, 72, 71, 0, 78, 0, 70, 77]])
```

A lower block triangular decomposition:

```

>>> P, B = A.strongly_connected_components_decomposition()
>>> pprint(P)
PermutationMatrix((8)(1 4 3 2 6)(5 7))
>>> pprint(B)
[[44 43 45] [0 0 0] [0 0 0] ]
[[         ] [         ] [         ] ]
[[34 33 35] [0 0 0] [0 0 0] ]
[[         ] [         ] [         ] ]
[[54 53 55] [0 0 0] [0 0 0] ]
[[         ] [         ] [         ] ]
[[0 0 0] [22 21 20] [0 0 0] ]
[[         ] [         ] [         ] ]
[[0 0 0] [12 11 10] [0 0 0] ]
[[         ] [         ] [         ] ]
[[0 0 0] [2 1 0] [0 0 0] ]
[[         ] [         ] [         ] ]
[[0 0 0] [62 61 60] [66 68 67]]
[[         ] [         ] [         ] ]
[[0 0 0] [82 81 80] [86 88 87]]
[[         ] [         ] [         ] ]
[[0 0 0] [72 71 70] [76 78 77]]

```

```

>>> P = P.as_explicit()
>>> B = B.as_explicit()
>>> P.T * B * P == A
True

```

An upper block triangular decomposition:

```

>>> P, B = A.strongly_connected_components_decomposition(lower=False)
>>> pprint(P)
PermutationMatrix((0 1 5 7 4 3 2 8 6))
>>> pprint(B)
[[66 68 67] [62 61 60] [0 0 0] ]
[[         ] [         ] [         ] ]
[[86 88 87] [82 81 80] [0 0 0] ]
[[         ] [         ] [         ] ]
[[76 78 77] [72 71 70] [0 0 0] ]
[[         ] [         ] [         ] ]
[[0 0 0] [22 21 20] [0 0 0] ]
[[         ] [         ] [         ] ]
[[0 0 0] [12 11 10] [0 0 0] ]
[[         ] [         ] [         ] ]
[[0 0 0] [2 1 0] [0 0 0] ]
[[         ] [         ] [         ] ]
[[0 0 0] [0 0 0] [44 43 45]]
[[         ] [         ] [         ] ]
[[0 0 0] [0 0 0] [34 33 35]]
[[         ] [         ] [         ] ]
[[0 0 0] [0 0 0] [54 53 55]]

```

```

>>> P = P.as_explicit()

```

(continues on next page)

(continued from previous page)

```
>>> B = B.as_explicit()
>>> P.T * B * P == A
True
```

table(printer, rowstart='[', rowend=']', rowsep='\n', colsep=', ', align='right')

String form of Matrix as a table.

printer is the printer to use for on the elements (generally something like StrPrinter())

rowstart is the string used to start each row (by default '[').

rowend is the string used to end each row (by default ']').

rowsep is the string used to separate rows (by default a newline).

colsep is the string used to separate columns (by default ', ').

align defines how the elements are aligned. Must be one of 'left', 'right', or 'center'. You can also use '<', '>', and '^' to mean the same thing, respectively.

This is used by the string printer for Matrix.

Examples

```
>>> from sympy import Matrix, StrPrinter
>>> M = Matrix([[1, 2], [-33, 4]])
>>> printer = StrPrinter()
>>> M.table(printer)
'[ 1, 2]\n[-33, 4]'
>>> print(M.table(printer))
[ 1, 2]
[-33, 4]
>>> print(M.table(printer, rowsep=',\n'))
[ 1, 2],
[-33, 4]
>>> print('[%s]' % M.table(printer, rowsep=',\n'))
[[ 1, 2],
[-33, 4]]
>>> print(M.table(printer, colsep=' '))
[ 1 2]
[-33 4]
>>> print(M.table(printer, align='center'))
[ 1 , 2]
[-33, 4]
>>> print(M.table(printer, rowstart='{', rowend='}'))
{ 1, 2}
{-33, 4}
```

upper_hessenberg_decomposition()

Converts a matrix into Hessenberg matrix H

Returns 2 matrices H, P s.t. $PHP^T = A$, where H is an upper hessenberg matrix and P is an orthogonal matrix

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([
...     [1,2,3],
...     [-3,5,6],
...     [4,-8,9],
... ])
>>> H, P = A.upper_hessenberg_decomposition()
>>> H
Matrix([
[1, 6/5, 17/5],
[5, 213/25, -134/25],
[0, 216/25, 137/25]])
>>> P
Matrix([
[1, 0, 0],
[0, -3/5, 4/5],
[0, 4/5, 3/5]])
>>> P * H * P.H == A
True
```

References

`upper_triangular_solve(rhs)`

Solves $Ax = B$, where A is an upper triangular matrix.

See also:

[*lower_triangular_solve*](#) (page 1304), [*gauss_jordan_solve*](#) (page 1297), [*cholesky_solve*](#) (page 1292), [*diagonal_solve*](#) (page 1295), [*LDLsolve*](#) (page 1282), [*LUsolve*](#) (page 1287), [*QRsolve*](#) (page 1291), [*pinv_solve*](#) (page 1306)

Matrix Exceptions Reference

class sympy.matrices.matrices.**MatrixError**

class sympy.matrices.matrices.**ShapeError**

Wrong matrix shape

class sympy.matrices.matrices.**NonSquareMatrixError**

Matrix Functions Reference

`sympy.matrices.dense.matrix_multiply_elementwise(A, B)`

Return the Hadamard product (elementwise product) of A and B

```
>>> from sympy import Matrix, matrix_multiply_elementwise
>>> A = Matrix([[0, 1, 2], [3, 4, 5]])
>>> B = Matrix([[1, 10, 100], [100, 10, 1]])
>>> matrix_multiply_elementwise(A, B)
Matrix([
 [ 0, 10, 200],
 [300, 40,  5]])
```

See also:

[`sympy.matrices.common.MatrixCommon.\_\_mul\_\_`](#) (page 1327)

`sympy.matrices.dense.zeros(*args, **kwargs)`

Returns a matrix of zeros with rows rows and cols columns; if cols is omitted a square matrix will be returned.

See also:

[`ones`](#) (page 1319), [`eye`](#) (page 1319), [`diag`](#) (page 1319)

`sympy.matrices.dense.ones(*args, **kwargs)`

Returns a matrix of ones with rows rows and cols columns; if cols is omitted a square matrix will be returned.

See also:

[`zeros`](#) (page 1319), [`eye`](#) (page 1319), [`diag`](#) (page 1319)

`sympy.matrices.dense.eye(*args, **kwargs)`

Create square identity matrix n x n

See also:

[`diag`](#) (page 1319), [`zeros`](#) (page 1319), [`ones`](#) (page 1319)

`sympy.matrices.dense.diag(*values, strict=True, unpack=False, **kwargs)`

Returns a matrix with the provided values placed on the diagonal. If non-square matrices are included, they will produce a block-diagonal matrix.

Examples

This version of `diag` is a thin wrapper to `Matrix.diag` that differs in that it treats all lists like matrices – even when a single list is given. If this is not desired, either put a `*` before the list or set `unpack = True`.

```
>>> from sympy import diag
```

```
>>> diag([1, 2, 3], unpack=True) # = diag(1,2,3) or diag(*[1,2,3])
Matrix([
 [1, 0, 0],
 [0, 2, 0],
 [0, 0, 3]])
```

```
>>> from sympy import Function, hessian, pprint
>>> from sympy.abc import x, y
>>> f = Function('f')(x, y)
>>> g1 = Function('g')(x, y)
>>> g2 = x**2 + 3*y
>>> pprint(hessian(f, (x, y), [g1, g2]))
```

$$\begin{bmatrix} 0 & 0 & \frac{\partial}{\partial x} (g(x, y)) & \frac{\partial}{\partial y} (g(x, y)) \\ 0 & 0 & 2x & 3 \\ \frac{\partial}{\partial x} (g(x, y)) & 2x & \frac{\partial^2}{\partial x^2} (f(x, y)) & \frac{\partial^2}{\partial y \partial x} (f(x, y)) \\ \frac{\partial}{\partial y} (g(x, y)) & 3 & \frac{\partial^2}{\partial x \partial y} (f(x, y)) & \frac{\partial^2}{\partial y^2} (f(x, y)) \end{bmatrix}$$

(continued from previous page)

$$\begin{bmatrix} \frac{\partial}{\partial x} & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial x} & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial x} & \frac{\partial}{\partial y} \end{bmatrix} \begin{bmatrix} f(x, y) \\ f(x, y) \\ f(x, y) \end{bmatrix}$$

See also:

[`sympy.matrices.matrices.MatrixCalculus.jacobian`](#) (page 1279), [`wronskian`](#) (page 1321)

References

[R596]

`sympy.matrices.dense.GramSchmidt(vlist, orthonormal=False)`

Apply the Gram-Schmidt process to a set of vectors.

Parameters

vlist : List of Matrix

Vectors to be orthogonalized for.

orthonormal : Bool, optional

If true, return an orthonormal basis.

Returns

vlist : List of Matrix

Orthogonalized vectors

Notes

This routine is mostly duplicate from `Matrix.orthogonalize`, except for some difference that this always raises error when linearly dependent vectors are found, and the keyword `normalize` has been named as `orthonormal` in this function.

See also:

[`matrices.MatrixSubspaces.orthogonalize`](#) (page 1236)

References

[R597]

`sympy.matrices.dense.wronskian(functions, var, method='bareiss')`

Compute Wronskian for [] of functions

$$W(f_1, \dots, f_n) = \begin{vmatrix} f_1 & f_2 & \dots & f_n \\ f_1' & f_2' & \dots & f_n' \\ \vdots & \vdots & \ddots & \vdots \\ f_1^{(n-1)} & f_2^{(n-1)} & \dots & f_n^{(n-1)} \end{vmatrix}$$

(continues on next page)

(continued from previous page)

$$\begin{vmatrix} & (n) & (n) & & (n) \\ D & (f1) & D & (f2) & \dots & D & (fn) \end{vmatrix}$$

see: <https://en.wikipedia.org/wiki/Wronskian>

See also:

`sympy.matrices.matrices.MatrixCalculus.jacobian` (page 1279), `hessian` (page 1320)

`sympy.matrices.dense.casoratian(seqs, n, zero=True)`

Given linear difference operator L of order ' k ' and homogeneous equation $Ly = 0$ we want to compute kernel of L , which is a set of ' k ' sequences: $a(n)$, $b(n)$, ... $z(n)$.

Solutions of L are linearly independent iff their Casoratian, denoted as $C(a, b, \dots, z)$, do not vanish for $n = 0$.

Casoratian is defined by $k \times k$ determinant:

$$\begin{vmatrix} + & a(n) & b(n) & \dots & z(n) & + \\ | & a(n+1) & b(n+1) & \dots & z(n+1) & | \\ | & \cdot & \cdot & \cdot & \cdot & | \\ | & \cdot & \cdot & \cdot & \cdot & | \\ | & \cdot & \cdot & \cdot & \cdot & | \\ + & a(n+k-1) & b(n+k-1) & \dots & z(n+k-1) & + \end{vmatrix}$$

It proves very useful in `rsolve_hyper()` where it is applied to a generating set of a recurrence to factor out linearly dependent solutions and return a basis:

```
>>> from sympy import Symbol, casoratian, factorial
>>> n = Symbol('n', integer=True)
```

Exponential and factorial are linearly independent:

```
>>> casoratian([2**n, factorial(n)], n) != 0
True
```

`sympy.matrices.dense.randMatrix(r, c=None, min=0, max=99, seed=None, symmetric=False, percent=100, prng=None)`

Create random matrix with dimensions $r \times c$. If c is omitted the matrix will be square. If `symmetric` is `True` the matrix must be square. If `percent` is less than 100 then only approximately the given percentage of elements will be non-zero.

The pseudo-random number generator used to generate matrix is chosen in the following way.

- If `prng` is supplied, it will be used as random number generator. It should be an instance of `random.Random`, or at least have `randint` and `shuffle` methods with same signatures.
- if `prng` is not supplied but `seed` is supplied, then new `random.Random` with given seed will be created;
- otherwise, a new `random.Random` with default seed will be used.

Examples

```
>>> from sympy import randMatrix
>>> randMatrix(3)
[25, 45, 27]
[44, 54, 9]
[23, 96, 46]
>>> randMatrix(3, 2)
[87, 29]
[23, 37]
[90, 26]
>>> randMatrix(3, 3, 0, 2)
[0, 2, 0]
[2, 0, 1]
[0, 0, 1]
>>> randMatrix(3, symmetric=True)
[85, 26, 29]
[26, 71, 43]
[29, 43, 57]
>>> A = randMatrix(3, seed=1)
>>> B = randMatrix(3, seed=2)
>>> A == B
False
>>> A == randMatrix(3, seed=1)
True
>>> randMatrix(3, symmetric=True, percent=50)
[77, 70, 0],
[70, 0, 0],
[0, 0, 88]
```

Numpy Utility Functions Reference

`sympy.matrices.dense.list2numpy(l, dtype=<class 'object'>)`
 Converts Python list of SymPy expressions to a NumPy array.

See also:

[`matrix2numpy`](#) (page 1323)

`sympy.matrices.dense.matrix2numpy(m, dtype=<class 'object'>)`
 Converts SymPy's matrix to a NumPy array.

See also:

[`list2numpy`](#) (page 1323)

`sympy.matrices.dense.symarray(prefix, shape, **kwargs)`
 Create a numpy ndarray of symbols (as an object array).

The created symbols are named `prefix_i1_i2_...`. You should thus provide a non-empty prefix if you want your symbols to be unique for different output arrays, as SymPy symbols with identical names are the same object.

Parameters

prefix : string

A prefix prepended to the name of every symbol.

shape : int or tuple

Shape of the created array. If an int, the array is one-dimensional; for more than one dimension the shape must be a tuple.

****kwargs** : dict

keyword arguments passed on to Symbol

Examples

These doctests require numpy.

```
>>> from sympy import symarray
>>> symarray('', 3)
[_0 _1 _2]
```

If you want multiple symarrays to contain distinct symbols, you *must* provide unique prefixes:

```
>>> a = symarray('', 3)
>>> b = symarray('', 3)
>>> a[0] == b[0]
True
>>> a = symarray('a', 3)
>>> b = symarray('b', 3)
>>> a[0] == b[0]
False
```

Creating symarrays with a prefix:

```
>>> symarray('a', 3)
[a_0 a_1 a_2]
```

For more than one dimension, the shape must be given as a tuple:

```
>>> symarray('a', (2, 3))
[[a_0_0 a_0_1 a_0_2]
 [a_1_0 a_1_1 a_1_2]]
>>> symarray('a', (2, 3, 2))
[[[a_0_0_0 a_0_0_1]
   [a_0_1_0 a_0_1_1]
   [a_0_2_0 a_0_2_1]]

 [a_1_0_0 a_1_0_1]
 [a_1_1_0 a_1_1_1]
 [a_1_2_0 a_1_2_1]]]
```

For setting assumptions of the underlying Symbols:

```
>>> [s.is_real for s in symarray('a', 2, real=True)]
[True, True]
```

`sympy.matrices.dense.rot_axis1(theta)`

Returns a rotation matrix for a rotation of θ (in radians) about the 1-axis.

Examples

```
>>> from sympy import pi, rot_axis1
```

A rotation of $\pi/3$ (60 degrees):

```
>>> theta = pi/3
>>> rot_axis1(theta)
Matrix([
[1,      0,      0],
[0,      1/2, sqrt(3)/2],
[0, -sqrt(3)/2,  1/2]])
```

If we rotate by $\pi/2$ (90 degrees):

```
>>> rot_axis1(pi/2)
Matrix([
[1, 0, 0],
[0, 0, 1],
[0, -1, 0]])
```

See also:

[**`rot\_axis2` \(page 1325\)**](#)

Returns a rotation matrix for a rotation of θ (in radians) about the 2-axis

[**`rot\_axis3` \(page 1326\)**](#)

Returns a rotation matrix for a rotation of θ (in radians) about the 3-axis

`sympy.matrices.dense.rot_axis2(theta)`

Returns a rotation matrix for a rotation of θ (in radians) about the 2-axis.

Examples

```
>>> from sympy import pi, rot_axis2
```

A rotation of $\pi/3$ (60 degrees):

```
>>> theta = pi/3
>>> rot_axis2(theta)
Matrix([
[ 1/2, 0, -sqrt(3)/2],
[ 0, 1,      0],
[sqrt(3)/2, 0,  1/2]])
```

If we rotate by $\pi/2$ (90 degrees):

```
>>> rot_axis2(pi/2)
Matrix([
[0, 0, -1],
[0, 1,  0],
[1, 0,  0]])
```

See also:

[rot_axis1](#) (page 1324)

Returns a rotation matrix for a rotation of theta (in radians) about the 1-axis

[rot_axis3](#) (page 1326)

Returns a rotation matrix for a rotation of theta (in radians) about the 3-axis

`sympy.matrices.dense.rot_axis3(theta)`

Returns a rotation matrix for a rotation of theta (in radians) about the 3-axis.

Examples

```
>>> from sympy import pi, rot_axis3
```

A rotation of $\pi/3$ (60 degrees):

```
>>> theta = pi/3
>>> rot_axis3(theta)
Matrix([
[ 1/2, sqrt(3)/2, 0],
[-sqrt(3)/2, 1/2, 0],
[ 0, 0, 1]])
```

If we rotate by $\pi/2$ (90 degrees):

```
>>> rot_axis3(pi/2)
Matrix([
[ 0, 1, 0],
[-1, 0, 0],
[ 0, 0, 1]])
```

See also:

[rot_axis1](#) (page 1324)

Returns a rotation matrix for a rotation of theta (in radians) about the 1-axis

[rot_axis2](#) (page 1325)

Returns a rotation matrix for a rotation of theta (in radians) about the 2-axis

`sympy.matrices.matrices.a2idx(j, n=None)`

Common Matrices

MatrixCommon Class Reference

class sympy.matrices.common.MatrixCommon

All common matrix operations including basic arithmetic, shaping, and special matrices like *zeros*, and *eye*.

property C

By-element conjugation

property H

Return Hermite conjugate.

Examples

```
>>> from sympy import Matrix, I
>>> m = Matrix((0, 1 + I, 2, 3))
>>> m
Matrix([
[ 0],
[1 + I],
[ 2],
[ 3]])
>>> m.H
Matrix([[0, 1 - I, 2, 3]])
```

See also:

[*conjugate*](#) (page 1330)

By-element conjugation

[*sympy.matrices.matrices.MatrixBase.D*](#) (page 1280)

Dirac conjugation

property T

Matrix transposition

__abs__()

Returns a new matrix with entry-wise absolute values.

__add__(*other*)

Return self + other, raising ShapeError if shapes do not match.

__getitem__(*key*)

Implementations of `__getitem__` should accept ints, in which case the matrix is indexed as a flat list, tuples (i,j) in which case the (i,j) entry is returned, slices, or mixed tuples (a,b) where a and b are any combination of slices and integers.

__len__()

The total number of entries in the matrix.

__mul__(*other*)

Return self*other where other is either a scalar or a matrix of compatible dimensions.

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([[1, 2, 3], [4, 5, 6]])
>>> 2*A == A*2 == Matrix([[2, 4, 6], [8, 10, 12]])
True
>>> B = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> A*B
Matrix([
[30, 36, 42],
[66, 81, 96]])
>>> B*A
Traceback (most recent call last):
...
ShapeError: Matrices size mismatch.
>>>
```

See also:

[*matrix_multiply_elementwise*](#) (page 1319)

__pow__(*exp*)

Return $\text{self}^{**}\text{exp}$ a scalar or symbol.

__weakref__

list of weak references to the object (if defined)

adjoint()

Conjugate transpose or Hermitian conjugation.

applyfunc(*f*)

Apply a function to each element of the matrix.

Examples

```
>>> from sympy import Matrix
>>> m = Matrix(2, 2, lambda i, j: i*2+j)
>>> m
Matrix([
[0, 1],
[2, 3]])
>>> m.applyfunc(lambda i: 2*i)
Matrix([
[0, 2],
[4, 6]])
```

as_real_imag(*deep=True, **hints*)

Returns a tuple containing the (real, imaginary) part of matrix.

atoms(**types*)

Returns the atoms that form the current object.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy import Matrix
>>> Matrix([[x]])
Matrix([[x]])
>>> _.atoms()
{x}
>>> Matrix([[x, y], [y, x]])
Matrix([
[x, y],
[y, x]])
>>> _.atoms()
{x, y}
```

col(*j*)

Elementary column selector.

Examples

```
>>> from sympy import eye
>>> eye(2).col(0)
Matrix([
[1],
[0]])
```

See also:

[row](#) (page 1352), [col_del](#) (page 1329), [col_join](#) (page 1329), [col_insert](#) (page 1329)

col_del(*col*)

Delete the specified column.

col_insert(*pos*, *other*)

Insert one or more columns at the given column position.

Examples

```
>>> from sympy import zeros, ones
>>> M = zeros(3)
>>> V = ones(3, 1)
>>> M.col_insert(1, V)
Matrix([
[0, 1, 0, 0],
[0, 1, 0, 0],
[0, 1, 0, 0]])
```

See also:

[col](#) (page 1329), [row_insert](#) (page 1352)

col_join(*other*)

Concatenates two matrices along self's last and other's first row.

Examples

```
>>> from sympy import zeros, ones
>>> M = zeros(3)
>>> V = ones(1, 3)
>>> M.col_join(V)
Matrix([
[0, 0, 0],
[0, 0, 0],
[0, 0, 0],
[1, 1, 1]])
```

See also:[col](#) (page 1329), [row_join](#) (page 1353)**classmethod companion**(*poly*)

Returns a companion matrix of a polynomial.

Examples

```
>>> from sympy import Matrix, Poly, Symbol, symbols
>>> x = Symbol('x')
>>> c0, c1, c2, c3, c4 = symbols('c0:5')
>>> p = Poly(c0 + c1*x + c2*x**2 + c3*x**3 + c4*x**4 + x**5, x)
>>> Matrix.companion(p)
Matrix([
[0, 0, 0, 0, -c0],
[1, 0, 0, 0, -c1],
[0, 1, 0, 0, -c2],
[0, 0, 1, 0, -c3],
[0, 0, 0, 1, -c4]])
```

conjugate()

Return the by-element conjugation.

Examples

```
>>> from sympy import SparseMatrix, I
>>> a = SparseMatrix(((1, 2 + I), (3, 4)), (I, -I))
>>> a
Matrix([
[1, 2 + I],
[3,      4],
[I,     -I]])
>>> a.C
```

(continues on next page)

(continued from previous page)

```
Matrix([
[ 1, 2 - I],
[ 3,      4],
[-I,      I]])
```

See also:**[transpose](#) (page 1355)**

Matrix transposition

[H](#) (page 1327)

Hermite conjugation

[sympy.matrices.matrices.MatrixBase.D](#) (page 1280)

Dirac conjugation

classmethod `diag`(**args*, *strict=False*, *unpack=True*, *rows=None*, *cols=None*, ***kwargs*)

Returns a matrix with the specified diagonal. If matrices are passed, a block-diagonal matrix is created (i.e. the “direct sum” of the matrices).

Kwargs**rows**

[rows of the resulting matrix; computed if] not given.

cols

[columns of the resulting matrix; computed if] not given.

cls : class for the resulting matrix

unpack : bool which, when True (default), unpacks a single sequence rather than interpreting it as a Matrix.

strict : bool which, when False (default), allows Matrices to have variable-length rows.

Examples

```
>>> from sympy import Matrix
>>> Matrix.diag(1, 2, 3)
Matrix([
[1, 0, 0],
[0, 2, 0],
[0, 0, 3]])
```

The current default is to unpack a single sequence. If this is not desired, set *unpack = False* and it will be interpreted as a matrix.

```
>>> Matrix.diag([1, 2, 3]) == Matrix.diag(1, 2, 3)
True
```

When more than one element is passed, each is interpreted as something to put on the diagonal. Lists are converted to matrices. Filling of the diagonal always continues from the bottom right hand corner of the previous item: this will create a block-diagonal matrix whether the matrices are square or not.

```
>>> col = [1, 2, 3]
>>> row = [[4, 5]]
>>> Matrix.diag(col, row)
Matrix([
[1, 0, 0],
[2, 0, 0],
[3, 0, 0],
[0, 4, 5]])
```

When *unpack* is False, elements within a list need not all be of the same length. Setting *strict* to True would raise a ValueError for the following:

```
>>> Matrix.diag([[1, 2, 3], [4, 5], [6]], unpack=False)
Matrix([
[1, 2, 3],
[4, 5, 0],
[6, 0, 0]])
```

The type of the returned matrix can be set with the *cls* keyword.

```
>>> from sympy import ImmutableMatrix
>>> from sympy.utilities.misc import func_name
>>> func_name(Matrix.diag(1, cls=ImmutableMatrix))
'ImmutableDenseMatrix'
```

A zero dimension matrix can be used to position the start of the filling at the start of an arbitrary row or column:

```
>>> from sympy import ones
>>> r2 = ones(0, 2)
>>> Matrix.diag(r2, 1, 2)
Matrix([
[0, 0, 1, 0],
[0, 0, 0, 2]])
```

See also:

[eye](#) (page 1334), [diagonal](#) (page 1332), [dense.diag](#) (page 1319), [expressions.blockmatrix.BlockMatrix](#) (page 1380), [sparsertools.banded](#) (page 1366)

diagonal(*k*=0)

Returns the *k*th diagonal of self. The main diagonal corresponds to $k = 0$; diagonals above and below correspond to $k > 0$ and $k < 0$, respectively. The values of *self*[*i*,*j*] for which $j - i = k$, are returned in order of increasing $i + j$, starting with $i + j = |k|$.

Examples

```
>>> from sympy import Matrix
>>> m = Matrix(3, 3, lambda i, j: j - i); m
Matrix([
[ 0,  1,  2],
[-1,  0,  1],
[-2, -1,  0]])
>>> m.diagonal()
Matrix([[0, 0, 0]])
>>> m.diagonal(1)
Matrix([[1, 1]])
>>> m.diagonal(-2)
Matrix([[-2]])
```

Even though the diagonal is returned as a Matrix, the element retrieval can be done with a single index:

```
>>> Matrix.diag(1, 2, 3).diagonal()[1] # instead of [0, 1]
2
```

See also:

[*diag*](#) (page 1331)

evalf(*n=15, subs=None, maxn=100, chop=False, strict=False, quad=None, verbose=False*)

Apply evalf() to each element of self.

expand(*deep=True, modulus=None, power_base=True, power_exp=True, mul=True, log=True, multinomial=True, basic=True, **hints*)

Apply core.function.expand to each entry of the matrix.

Examples

```
>>> from sympy.abc import x
>>> from sympy import Matrix
>>> Matrix(1, 1, [x*(x+1)])
Matrix([[x*(x + 1)])]
>>> m.expand()
Matrix([[x**2 + x]])
```

extract(*rowsList, colsList*)

Return a submatrix by specifying a list of rows and columns. Negative indices can be given. All indices must be in the range $-n \leq i < n$ where n is the number of rows or columns.

Examples

```
>>> from sympy import Matrix
>>> m = Matrix(4, 3, range(12))
>>> m
Matrix([
[0, 1, 2],
[3, 4, 5],
[6, 7, 8],
[9, 10, 11]])
>>> m.extract([0, 1, 3], [0, 1])
Matrix([
[0, 1],
[3, 4],
[9, 10]])
```

Rows or columns can be repeated:

```
>>> m.extract([0, 0, 1], [-1])
Matrix([
[2],
[2],
[5]])
```

Every other row can be taken by using range to provide the indices:

```
>>> m.extract(range(0, m.rows, 2), [-1])
Matrix([
[2],
[8]])
```

RowsList or colsList can also be a list of booleans, in which case the rows or columns corresponding to the True values will be selected:

```
>>> m.extract([0, 1, 2, 3], [True, False, True])
Matrix([
[0, 2],
[3, 5],
[6, 8],
[9, 11]])
```

classmethod `eye(rows, cols=None, **kwargs)`

Returns an identity matrix.

Args

rows : rows of the matrix cols : cols of the matrix (if None, cols=rows)

Kwargs

cls : class of the returned matrix

property free_symbols

Returns the free symbols within the matrix.

Examples

```
>>> from sympy.abc import x
>>> from sympy import Matrix
>>> Matrix([[x], [1]]).free_symbols
{x}
```

get_diag_blocks()

Obtains the square sub-matrices on the main diagonal of a square matrix.

Useful for inverting symbolic matrices or solving systems of linear equations which may be decoupled by having a block diagonal structure.

Examples

```
>>> from sympy import Matrix
>>> from sympy.abc import x, y, z
>>> A = Matrix([[1, 3, 0, 0], [y, z*z, 0, 0], [0, 0, x, 0], [0, 0, 0, 0]])
>>> a1, a2, a3 = A.get_diag_blocks()
>>> a1
Matrix([
[1, 3],
[y, z**2]])
>>> a2
Matrix([[x]])
>>> a3
Matrix([[0]])
```

has(*patterns)

Test whether any subexpression matches any of the patterns.

Examples

```
>>> from sympy import Matrix, SparseMatrix, Float
>>> from sympy.abc import x, y
>>> A = Matrix(((1, x), (0.2, 3)))
>>> B = SparseMatrix(((1, x), (0.2, 3)))
>>> A.has(x)
True
>>> A.has(y)
False
>>> A.has(Float)
True
>>> B.has(x)
True
>>> B.has(y)
False
>>> B.has(Float)
True
```

classmethod `hstack(*args)`

Return a matrix formed by joining args horizontally (i.e. by repeated application of `row_join`).

Examples

```
>>> from sympy import Matrix, eye
>>> Matrix.hstack(eye(2), 2*eye(2))
Matrix([
[1, 0, 2, 0],
[0, 1, 0, 2]])
```

is_anti_symmetric(*simplify=True*)

Check if matrix M is an antisymmetric matrix, that is, M is a square matrix with all $M[i, j] == -M[j, i]$.

When `simplify=True` (default), the sum $M[i, j] + M[j, i]$ is simplified before testing to see if it is zero. By default, the SymPy `simplify` function is used. To use a custom function set `simplify` to a function that accepts a single argument which returns a simplified expression. To skip simplification, set `simplify` to `False` but note that although this will be faster, it may induce false negatives.

Examples

```
>>> from sympy import Matrix, symbols
>>> m = Matrix(2, 2, [0, 1, -1, 0])
>>> m
Matrix([
[ 0, 1],
[-1, 0]])
>>> m.is_anti_symmetric()
```

(continues on next page)

(continued from previous page)

```

True
>>> x, y = symbols('x y')
>>> m = Matrix(2, 3, [0, 0, x, -y, 0, 0])
>>> m
Matrix([
[ 0, 0, x],
[-y, 0, 0]])
>>> m.is_anti_symmetric()
False

```

```

>>> from sympy.abc import x, y
>>> m = Matrix(3, 3, [0, x**2 + 2*x + 1, y,
...                  -(x + 1)**2, 0, x*y,
...                  -y, -x*y, 0])

```

Simplification of matrix elements is done by default so even though two elements which should be equal and opposite would not pass an equality test, the matrix is still reported as anti-symmetric:

```

>>> m[0, 1] == -m[1, 0]
False
>>> m.is_anti_symmetric()
True

```

If `simplify=False` is used for the case when a Matrix is already simplified, this will speed things up. Here, we see that without simplification the matrix does not appear anti-symmetric:

```

>>> m.is_anti_symmetric(simplify=False)
False

```

But if the matrix were already expanded, then it would appear anti-symmetric and simplification in the `is_anti_symmetric` routine is not needed:

```

>>> m = m.expand()
>>> m.is_anti_symmetric(simplify=False)
True

```

`is_diagonal()`

Check if matrix is diagonal, that is matrix in which the entries outside the main diagonal are all zero.

Examples

```

>>> from sympy import Matrix, diag
>>> m = Matrix(2, 2, [1, 0, 0, 2])
>>> m
Matrix([
[1, 0],
[0, 2]])

```

(continues on next page)

(continued from previous page)

```
>>> m.is_diagonal()
True
```

```
>>> m = Matrix(2, 2, [1, 1, 0, 2])
>>> m
Matrix([
[1, 1],
[0, 2]])
>>> m.is_diagonal()
False
```

```
>>> m = diag(1, 2, 3)
>>> m
Matrix([
[1, 0, 0],
[0, 2, 0],
[0, 0, 3]])
>>> m.is_diagonal()
True
```

See also:

[*is_lower*](#) (page 1338), [*is_upper*](#) (page 1342), [*sympy.matrices.matrices.MatrixEigen.is_diagonalizable*](#) (page 1241), [*diagonalize*](#) (page 1238)

property `is_hermitian`

Checks if the matrix is Hermitian.

In a Hermitian matrix element i,j is the complex conjugate of element j,i .

Examples

```
>>> from sympy import Matrix
>>> from sympy import I
>>> from sympy.abc import x
>>> a = Matrix([[1, I], [-I, 1]])
>>> a
Matrix([
[ 1, I],
[-I, 1]])
>>> a.is_hermitian
True
>>> a[0, 0] = 2*I
>>> a.is_hermitian
False
>>> a[0, 0] = x
>>> a.is_hermitian
>>> a[0, 1] = a[1, 0]*I
>>> a.is_hermitian
False
```

property is_lower

Check if matrix is a lower triangular matrix. True can be returned even if the matrix is not square.

Examples

```
>>> from sympy import Matrix
>>> m = Matrix(2, 2, [1, 0, 0, 1])
>>> m
Matrix([
[1, 0],
[0, 1]])
>>> m.is_lower
True
```

```
>>> m = Matrix(4, 3, [0, 0, 0, 2, 0, 0, 1, 4, 0, 6, 6, 5])
>>> m
Matrix([
[0, 0, 0],
[2, 0, 0],
[1, 4, 0],
[6, 6, 5]])
>>> m.is_lower
True
```

```
>>> from sympy.abc import x, y
>>> m = Matrix(2, 2, [x**2 + y, y**2 + x, 0, x + y])
>>> m
Matrix([
[x**2 + y, x + y**2],
[0, x + y]])
>>> m.is_lower
False
```

See also:

[*is_upper*](#) (page 1342), [*is_diagonal*](#) (page 1337), [*is_lower_hessenberg*](#) (page 1339)

property is_lower_hessenberg

Checks if the matrix is in the lower-Hessenberg form.

The lower hessenberg matrix has zero entries above the first superdiagonal.

Examples

```
>>> from sympy import Matrix
>>> a = Matrix([[1, 2, 0, 0], [5, 2, 3, 0], [3, 4, 3, 7], [5, 6, 1, 1]])
>>> a
Matrix([
[1, 2, 0, 0],
[5, 2, 3, 0],
[3, 4, 3, 7],
[5, 6, 1, 1]])
>>> a.is_lower_hessenberg
True
```

See also:

[*is_upper_hessenberg*](#) (page 1343), [*is_lower*](#) (page 1338)

property `is_square`

Checks if a matrix is square.

A matrix is square if the number of rows equals the number of columns. The empty matrix is square by definition, since the number of rows and the number of columns are both zero.

Examples

```
>>> from sympy import Matrix
>>> a = Matrix([[1, 2, 3], [4, 5, 6]])
>>> b = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> c = Matrix([])
>>> a.is_square
False
>>> b.is_square
True
>>> c.is_square
True
```

property `is_strongly_diagonally_dominant`

Tests if the matrix is row strongly diagonally dominant.

Explanation

A n, n matrix A is row strongly diagonally dominant if

$$|A_{i,i}| > \sum_{j=0, j \neq i}^{n-1} |A_{i,j}| \quad \text{for all } i \in \{0, \dots, n-1\}$$

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([[3, -2, 1], [1, -3, 2], [-1, 2, 4]])
>>> A.is_strongly_diagonally_dominant
False
```

```
>>> A = Matrix([[-2, 2, 1], [1, 3, 2], [1, -2, 0]])
>>> A.is_strongly_diagonally_dominant
False
```

```
>>> A = Matrix([[-4, 2, 1], [1, 6, 2], [1, -2, 5]])
>>> A.is_strongly_diagonally_dominant
True
```

Notes

If you want to test whether a matrix is column diagonally dominant, you can apply the test after transposing the matrix.

`is_symbolic()`

Checks if any elements contain Symbols.

Examples

```
>>> from sympy import Matrix
>>> from sympy.abc import x, y
>>> M = Matrix([x, y], [1, 0])
>>> M.is_symbolic()
True
```

`is_symmetric(simplify=True)`

Check if matrix is symmetric matrix, that is square matrix and is equal to its transpose.

By default, simplifications occur before testing symmetry. They can be skipped using `'simplify=False'`; while speeding things a bit, this may however induce false negatives.

Examples

```
>>> from sympy import Matrix
>>> m = Matrix(2, 2, [0, 1, 1, 2])
>>> m
Matrix([
[0, 1],
[1, 2]])
>>> m.is_symmetric()
True
```

```
>>> m = Matrix(2, 2, [0, 1, 2, 0])
>>> m
Matrix([
[0, 1],
[2, 0]])
>>> m.is_symmetric()
False
```

```
>>> m = Matrix(2, 3, [0, 0, 0, 0, 0, 0])
>>> m
Matrix([
[0, 0, 0],
[0, 0, 0]])
>>> m.is_symmetric()
False
```

```
>>> from sympy.abc import x, y
>>> m = Matrix(3, 3, [1, x**2 + 2*x + 1, y, (x + 1)**2, 2, 0, y, 0,
↪ 3])
>>> m
Matrix([
[      1, x**2 + 2*x + 1, y],
[(x + 1)**2,          2, 0],
[      y,          0, 3]])
>>> m.is_symmetric()
True
```

If the matrix is already simplified, you may speed-up `is_symmetric()` test by using `'simplify=False'`.

```
>>> bool(m.is_symmetric(simplify=False))
False
>>> m1 = m.expand()
>>> m1.is_symmetric(simplify=False)
True
```

property `is_upper`

Check if matrix is an upper triangular matrix. `True` can be returned even if the matrix is not square.

Examples

```
>>> from sympy import Matrix
>>> m = Matrix(2, 2, [1, 0, 0, 1])
>>> m
Matrix([
[1, 0],
[0, 1]])
>>> m.is_upper
True
```

```
>>> m = Matrix(4, 3, [5, 1, 9, 0, 4, 6, 0, 0, 5, 0, 0, 0])
>>> m
Matrix([
[5, 1, 9],
[0, 4, 6],
[0, 0, 5],
[0, 0, 0]])
>>> m.is_upper
True
```

```
>>> m = Matrix(2, 3, [4, 2, 5, 6, 1, 1])
>>> m
Matrix([
[4, 2, 5],
[6, 1, 1]])
>>> m.is_upper
False
```

See also:

[*is_lower*](#) (page 1338), [*is_diagonal*](#) (page 1337), [*is_upper_hessenberg*](#) (page 1343)

property `is_upper_hessenberg`

Checks if the matrix is the upper-Hessenberg form.

The upper hessenberg matrix has zero entries below the first subdiagonal.

Examples

```
>>> from sympy import Matrix
>>> a = Matrix([[1, 4, 2, 3], [3, 4, 1, 7], [0, 2, 3, 4], [0, 0, 1, 3]])
>>> a
Matrix([
[1, 4, 2, 3],
[3, 4, 1, 7],
[0, 2, 3, 4],
[0, 0, 1, 3]])
>>> a.is_upper_hessenberg
True
```

See also:

[*is_lower_hessenberg*](#) (page 1339), [*is_upper*](#) (page 1342)

property `is_weakly_diagonally_dominant`

Tests if the matrix is row weakly diagonally dominant.

Explanation

A n, n matrix A is row weakly diagonally dominant if

$$|A_{i,i}| \geq \sum_{j=0, j \neq i}^{n-1} |A_{i,j}| \quad \text{for all } i \in \{0, \dots, n-1\}$$

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([[3, -2, 1], [1, -3, 2], [-1, 2, 4]])
>>> A.is_weakly_diagonally_dominant
True
```

```
>>> A = Matrix([[-2, 2, 1], [1, 3, 2], [1, -2, 0]])
>>> A.is_weakly_diagonally_dominant
False
```

```
>>> A = Matrix([[-4, 2, 1], [1, 6, 2], [1, -2, 5]])
>>> A.is_weakly_diagonally_dominant
True
```

Notes

If you want to test whether a matrix is column diagonally dominant, you can apply the test after transposing the matrix.

property `is_zero_matrix`

Checks if a matrix is a zero matrix.

A matrix is zero if every element is zero. A matrix need not be square to be considered zero. The empty matrix is zero by the principle of vacuous truth. For a matrix that may or may not be zero (e.g. contains a symbol), this will be `None`

Examples

```
>>> from sympy import Matrix, zeros
>>> from sympy.abc import x
>>> a = Matrix([[0, 0], [0, 0]])
>>> b = zeros(3, 4)
>>> c = Matrix([[0, 1], [0, 0]])
>>> d = Matrix([])
>>> e = Matrix([[x, 0], [0, 0]])
>>> a.is_zero_matrix
True
>>> b.is_zero_matrix
True
>>> c.is_zero_matrix
```

(continues on next page)

(continued from previous page)

```
False
>>> d.is_zero_matrix
True
>>> e.is_zero_matrix
```

classmethod `jordan_block`(*size=None, eigenvalue=None, *, band='upper', **kwargs*)

Returns a Jordan block

Parameters

size : Integer, optional

Specifies the shape of the Jordan block matrix.

eigenvalue : Number or Symbol

Specifies the value for the main diagonal of the matrix.

Note: The keyword `eigenval` is also specified as an alias of this keyword, but it is not recommended to use.

We may deprecate the alias in later release.

band : 'upper' or 'lower', optional

Specifies the position of the off-diagonal to put 1 s on.

cls : Matrix, optional

Specifies the matrix class of the output form.

If it is not specified, the class type where the method is being executed on will be returned.

rows, cols : Integer, optional

Specifies the shape of the Jordan block matrix. See Notes section for the details of how these key works.

Deprecated since version 1.4: The rows and cols parameters are deprecated and will be removed in a future version.

Returns

Matrix

A Jordan block matrix.

Raises

ValueError

If insufficient arguments are given for matrix size specification, or no eigenvalue is given.

Examples

Creating a default Jordan block:

```
>>> from sympy import Matrix
>>> from sympy.abc import x
>>> Matrix.jordan_block(4, x)
Matrix([
[x, 1, 0, 0],
[0, x, 1, 0],
[0, 0, x, 1],
[0, 0, 0, x]])
```

Creating an alternative Jordan block matrix where 1 is on lower off-diagonal:

```
>>> Matrix.jordan_block(4, x, band='lower')
Matrix([
[x, 0, 0, 0],
[1, x, 0, 0],
[0, 1, x, 0],
[0, 0, 1, x]])
```

Creating a Jordan block with keyword arguments

```
>>> Matrix.jordan_block(size=4, eigenvalue=x)
Matrix([
[x, 1, 0, 0],
[0, x, 1, 0],
[0, 0, x, 1],
[0, 0, 0, x]])
```

Notes

Deprecated since version 1.4: This feature is deprecated and will be removed in a future version.

The keyword arguments `size`, `rows`, `cols` relates to the Jordan block size specifications.

If you want to create a square Jordan block, specify either one of the three arguments.

If you want to create a rectangular Jordan block, specify `rows` and `cols` individually.

Arguments Given			Matrix Shape	
size	rows	cols	rows	cols
size	Any		size	size
None	None		ValueError	
	rows	None	rows	rows
	None	cols	cols	cols
	rows	cols	rows	cols

References

[R562]

lower_triangular(*k=0*)

returns the elements on and below the *k*th diagonal of a matrix. If *k* is not specified then simply returns lower-triangular portion of a matrix

Examples

```
>>> from sympy import ones
>>> A = ones(4)
>>> A.lower_triangular()
Matrix([
[1, 0, 0, 0],
[1, 1, 0, 0],
[1, 1, 1, 0],
[1, 1, 1, 1]])
```

```
>>> A.lower_triangular(-2)
Matrix([
[0, 0, 0, 0],
[0, 0, 0, 0],
[1, 0, 0, 0],
[1, 1, 0, 0]])
```

```
>>> A.lower_triangular(1)
Matrix([
[1, 1, 0, 0],
[1, 1, 1, 0],
[1, 1, 1, 1],
[1, 1, 1, 1]])
```

multiply(*other*, *dotprodsimp=None*)

Same as `__mul__()` but with optional simplification.

Parameters

dotprodsimp : bool, optional

Specifies whether intermediate term algebraic simplification is used during matrix multiplications to control expression blowup and thus speed up calculation. Default is off.

multiply_elementwise(*other*)

Return the Hadamard product (elementwise product) of A and B

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([[0, 1, 2], [3, 4, 5]])
>>> B = Matrix([[1, 10, 100], [100, 10, 1]])
>>> A.multiply_elementwise(B)
Matrix([
[ 0, 10, 200],
[300, 40,  5]])
```

See also:

[*sympy.matrices.matrices.MatrixBase.cross*](#) (page 1295), [*sympy.matrices.matrices.MatrixBase.dot*](#) (page 1296), [*multiply*](#) (page 1347)

n(*args, **kwargs)

Apply evalf() to each element of self.

classmethod ones(rows, cols=None, **kwargs)

Returns a matrix of ones.

Args

rows : rows of the matrix cols : cols of the matrix (if None, cols=rows)

Kwargs

cls : class of the returned matrix

permute(perm, orientation='rows', direction='forward')

Permute the rows or columns of a matrix by the given list of swaps.

Parameters

perm : Permutation, list, or list of lists

A representation for the permutation.

If it is `Permutation`, it is used directly with some resizing with respect to the matrix size.

If it is specified as list of lists, (e.g., `[[0, 1], [0, 2]]`), then the permutation is formed from applying the product of cycles. The direction how the cyclic product is applied is described in below.

If it is specified as a list, the list should represent an array form of a permutation. (e.g., `[1, 2, 0]`) which would form the swapping function $0 \mapsto 1, 1 \mapsto 2, 2 \mapsto 0$.

orientation : 'rows', 'cols'

A flag to control whether to permute the rows or the columns

direction : 'forward', 'backward'

A flag to control whether to apply the permutations from the start of the list first, or from the back of the list first.

For example, if the permutation specification is `[[0, 1], [0, 2]]`,

If the flag is set to 'forward', the cycle would be formed as $0 \mapsto 2, 2 \mapsto 1, 1 \mapsto 0$.

If the flag is set to 'backward', the cycle would be formed as $0 \mapsto 1, 1 \mapsto 2, 2 \mapsto 0$.

If the argument perm is not in a form of list of lists, this flag takes no effect.

Examples

```
>>> from sympy import eye
>>> M = eye(3)
>>> M.permute([[0, 1], [0, 2]], orientation='rows', direction='forward
↪')
Matrix([
[0, 0, 1],
[1, 0, 0],
[0, 1, 0]])
```

```
>>> from sympy import eye
>>> M = eye(3)
>>> M.permute([[0, 1], [0, 2]], orientation='rows', direction=
↪'backward')
Matrix([
[0, 1, 0],
[0, 0, 1],
[1, 0, 0]])
```

Notes

If a bijective function $\sigma : \mathbb{N}_0 \rightarrow \mathbb{N}_0$ denotes the permutation.

If the matrix A is the matrix to permute, represented as a horizontal or a vertical stack of vectors:

$$A = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = [\alpha_0 \quad \alpha_1 \quad \cdots \quad \alpha_{n-1}]$$

If the matrix B is the result, the permutation of matrix rows is defined as:

$$B := \begin{bmatrix} a_{\sigma(0)} \\ a_{\sigma(1)} \\ \vdots \\ a_{\sigma(n-1)} \end{bmatrix}$$

And the permutation of matrix columns is defined as:

$$B := [\alpha_{\sigma(0)} \quad \alpha_{\sigma(1)} \quad \cdots \quad \alpha_{\sigma(n-1)}]$$

permute_cols(*swaps*, *direction*='forward')

Alias for `self.permute(swaps, orientation='cols', direction=direction)`

See also:

[*permute*](#) (page 1348)

permute_rows(*swaps*, *direction*='forward')

Alias for `self.permute(swaps, orientation='rows', direction=direction)`

See also:

[*permute*](#) (page 1348)

pow(*exp*, *method*=None)

Return `self**exp` a scalar or symbol.

Parameters

method : multiply, mulsimp, jordan, cayley

If multiply then it returns exponentiation using recursion. If jordan then Jordan form exponentiation will be used. If cayley then the exponentiation is done using Cayley-Hamilton theorem. If mulsimp then the exponentiation is done using recursion with dotprodsimp. This specifies whether intermediate term algebraic simplification is used during naive matrix power to control expression blowup and thus speed up calculation. If None, then it heuristically decides which method to use.

refine(*assumptions*=True)

Apply refine to each element of the matrix.

Examples

```
>>> from sympy import Symbol, Matrix, Abs, sqrt, Q
>>> x = Symbol('x')
>>> Matrix([[Abs(x)**2, sqrt(x**2)], [sqrt(x**2), Abs(x)**2]])
Matrix([
 [ Abs(x)**2, sqrt(x**2)],
 [ sqrt(x**2), Abs(x)**2]])
>>> _.refine(Q.real(x))
Matrix([
 [ x**2, Abs(x)],
 [ Abs(x), x**2]])
```

replace(*F*, *G*, *map*=False, *simultaneous*=True, *exact*=None)

Replaces Function *F* in Matrix entries with Function *G*.

Examples

```
>>> from sympy import symbols, Function, Matrix
>>> F, G = symbols('F, G', cls=Function)
>>> M = Matrix(2, 2, lambda i, j: F(i+j)) ; M
Matrix([
[F(0), F(1)],
[F(1), F(2)]])
>>> N = M.replace(F,G)
>>> N
Matrix([
[G(0), G(1)],
[G(1), G(2)]])
```

reshape(rows, cols)

Reshape the matrix. Total number of elements must remain the same.

Examples

```
>>> from sympy import Matrix
>>> m = Matrix(2, 3, lambda i, j: 1)
>>> m
Matrix([
[1, 1, 1],
[1, 1, 1]])
>>> m.reshape(1, 6)
Matrix([[1, 1, 1, 1, 1, 1]])
>>> m.reshape(3, 2)
Matrix([
[1, 1],
[1, 1],
[1, 1]])
```

rmultiply(other, dotprodsimp=None)

Same as `__rmul__()` but with optional simplification.

Parameters

dotprodsimp : bool, optional

Specifies whether intermediate term algebraic simplification is used during matrix multiplications to control expression blowup and thus speed up calculation. Default is off.

rot90(k=1)

Rotates Matrix by 90 degrees

Parameters

k : int

Specifies how many times the matrix is rotated by 90 degrees (clockwise when positive, counter-clockwise when negative).

Examples

```
>>> from sympy import Matrix, symbols
>>> A = Matrix(2, 2, symbols('a:d'))
>>> A
Matrix([
[a, b],
[c, d]])
```

Rotating the matrix clockwise one time:

```
>>> A.rot90(1)
Matrix([
[c, a],
[d, b]])
```

Rotating the matrix anticlockwise two times:

```
>>> A.rot90(-2)
Matrix([
[d, c],
[b, a]])
```

row(*i*)

Elementary row selector.

Examples

```
>>> from sympy import eye
>>> eye(2).row(0)
Matrix([[1, 0]])
```

See also:

[*col*](#) (page 1329), [*row_del*](#) (page 1352), [*row_join*](#) (page 1353), [*row_insert*](#) (page 1352)

row_del(*row*)

Delete the specified row.

row_insert(*pos*, *other*)

Insert one or more rows at the given row position.

Examples

```
>>> from sympy import zeros, ones
>>> M = zeros(3)
>>> V = ones(1, 3)
>>> M.row_insert(1, V)
Matrix([
[0, 0, 0],
[1, 1, 1],
[0, 0, 0],
[0, 0, 0]])
```

See also:

[row](#) (page 1352), [col_insert](#) (page 1329)

`row_join(other)`

Concatenates two matrices along self's last and rhs's first column

Examples

```
>>> from sympy import zeros, ones
>>> M = zeros(3)
>>> V = ones(3, 1)
>>> M.row_join(V)
Matrix([
[0, 0, 0, 1],
[0, 0, 0, 1],
[0, 0, 0, 1]])
```

See also:

[row](#) (page 1352), [col_join](#) (page 1329)

`property shape`

The shape (dimensions) of the matrix as the 2-tuple (rows, cols).

Examples

```
>>> from sympy import zeros
>>> M = zeros(2, 3)
>>> M.shape
(2, 3)
>>> M.rows
2
>>> M.cols
3
```

`simplify(**kwargs)`

Apply simplify to each element of the matrix.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy import SparseMatrix, sin, cos
>>> SparseMatrix(1, 1, [x*sin(y)**2 + x*cos(y)**2])
Matrix([[x*sin(y)**2 + x*cos(y)**2]])
>>> _.simplify()
Matrix([[x]])
```

subs(*args, **kwargs)

Return a new matrix with subs applied to each entry.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy import SparseMatrix, Matrix
>>> SparseMatrix(1, 1, [x])
Matrix([[x]])
>>> _.subs(x, y)
Matrix([[y]])
>>> Matrix(_).subs(y, x)
Matrix([[x]])
```

todod()

Returns matrix as dict of dicts containing non-zero elements of the Matrix

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([[0, 1],[0, 3]])
>>> A
Matrix([
[0, 1],
[0, 3]])
>>> A.todod()
{0: {1: 1}, 1: {1: 3}}
```

todok()

Return the matrix as dictionary of keys.

Examples

```
>>> from sympy import Matrix
>>> M = Matrix.eye(3)
>>> M.todok()
{(0, 0): 1, (1, 1): 1, (2, 2): 1}
```

tolist()

Return the Matrix as a nested Python list.

Examples

```
>>> from sympy import Matrix, ones
>>> m = Matrix(3, 3, range(9))
>>> m
Matrix([
[0, 1, 2],
[3, 4, 5],
[6, 7, 8]])
>>> m.tolist()
[[0, 1, 2], [3, 4, 5], [6, 7, 8]]
>>> ones(3, 0).tolist()
[[], [], []]
```

When there are no rows then it will not be possible to tell how many columns were in the original matrix:

```
>>> ones(0, 3).tolist()
[]
```

trace()

Returns the trace of a square matrix i.e. the sum of the diagonal elements.

Examples

```
>>> from sympy import Matrix
>>> A = Matrix(2, 2, [1, 2, 3, 4])
>>> A.trace()
5
```

transpose()

Returns the transpose of the matrix.

Examples

```
>>> from sympy import Matrix
>>> A = Matrix(2, 2, [1, 2, 3, 4])
>>> A.transpose()
Matrix([
[1, 3],
[2, 4]])
```

```
>>> from sympy import Matrix, I
>>> m=Matrix(((1, 2+I), (3, 4)))
>>> m
Matrix([
[1, 2 + I],
[3,      4]])
>>> m.transpose()
Matrix([
[      1, 3],
[2 + I, 4]])
>>> m.T == m.transpose()
True
```

See also:

[*conjugate*](#) (page 1330)
By-element conjugation

`upper_triangular(k=0)`

returns the elements on and above the kth diagonal of a matrix. If k is not specified then simply returns upper-triangular portion of a matrix

Examples

```
>>> from sympy import ones
>>> A = ones(4)
>>> A.upper_triangular()
Matrix([
[1, 1, 1, 1],
[0, 1, 1, 1],
[0, 0, 1, 1],
[0, 0, 0, 1]])
```

```
>>> A.upper_triangular(2)
Matrix([
[0, 0, 1, 1],
[0, 0, 0, 1],
[0, 0, 0, 0],
[0, 0, 0, 0]])
```

```
>>> A.upper_triangular(-1)
Matrix([
[1, 1, 1, 1],
[1, 1, 1, 1],
[0, 1, 1, 1],
[0, 0, 1, 1]])
```

values()

Return non-zero values of self.

vec()

Return the Matrix converted into a one column matrix by stacking columns

Examples

```
>>> from sympy import Matrix
>>> m=Matrix([[1, 3], [2, 4]])
>>> m
Matrix([
[1, 3],
[2, 4]])
>>> m.vec()
Matrix([
[1],
[2],
[3],
[4]])
```

See also:

[vech](#) (page 1357)

vech(*diagonal=True, check_symmetry=True*)

Reshapes the matrix into a column vector by stacking the elements in the lower triangle.

Parameters

diagonal : bool, optional

If True, it includes the diagonal elements.

check_symmetry : bool, optional

If True, it checks whether the matrix is symmetric.

Examples

```
>>> from sympy import Matrix
>>> m=Matrix([[1, 2], [2, 3]])
>>> m
Matrix([
[1, 2],
[2, 3]])
>>> m.vech()
Matrix([
[1],
[2],
[3]])
>>> m.vech(diagonal=False)
Matrix([[2]])
```

Notes

This should work for symmetric matrices and `vech` can represent symmetric matrices in vector form with less size than `vec`.

See also:

[vec](#) (page 1357)

classmethod `vstack(*args)`

Return a matrix formed by joining args vertically (i.e. by repeated application of `col_join`).

Examples

```
>>> from sympy import Matrix, eye
>>> Matrix.vstack(eye(2), 2*eye(2))
Matrix([
[1, 0],
[0, 1],
[2, 0],
[0, 2]])
```

classmethod `wilkinson(n, **kwargs)`

Returns two square Wilkinson Matrix of size $2*n + 1$ W_{2n+1}^- , $W_{2n+1}^+ = \text{Wilkinson}(n)$

Examples

```
>>> from sympy import Matrix
>>> wminus, wplus = Matrix.wilkinson(3)
>>> wminus
Matrix([
[-3,  1,  0,  0,  0,  0,  0],
[ 1, -2,  1,  0,  0,  0,  0],
[ 0,  1, -1,  1,  0,  0,  0],
[ 0,  0,  1,  0,  1,  0,  0],
[ 0,  0,  0,  1,  1,  1,  0],
[ 0,  0,  0,  0,  1,  2,  1],
[ 0,  0,  0,  0,  0,  1,  3]])
>>> wplus
Matrix([
[3, 1, 0, 0, 0, 0, 0],
[1, 2, 1, 0, 0, 0, 0],
[0, 1, 1, 1, 0, 0, 0],
[0, 0, 1, 0, 1, 0, 0],
[0, 0, 0, 1, 1, 1, 0],
[0, 0, 0, 0, 1, 2, 1],
[0, 0, 0, 0, 0, 1, 3]])
```

References

[R563], [R564]

xreplace(*rule*)

Return a new matrix with xreplace applied to each entry.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy import SparseMatrix, Matrix
>>> SparseMatrix(1, 1, [x])
Matrix([[x]])
>>> _.xreplace({x: y})
Matrix([[y]])
>>> Matrix(_).xreplace({y: x})
Matrix([[x]])
```

classmethod zeros(*rows*, *cols*=None, ***kwargs*)

Returns a matrix of zeros.

Args

rows : rows of the matrix cols : cols of the matrix (if None, cols=rows)

Kwargs

cls : class of the returned matrix

MatrixKind

class sympy.matrices.common.**MatrixKind**(*element_kind=NumberKind*)

Kind for all matrices in SymPy.

Basic class for this kind is `MatrixBase` and `MatrixExpr`, but any expression representing the matrix can have this.

Parameters

element_kind : Kind

Kind of the element. Default is `sympy.core.kind.NumberKind` (page 1074), which means that the matrix contains only numbers.

Examples

Any instance of matrix class has `MatrixKind`:

```
>>> from sympy import MatrixSymbol
>>> A = MatrixSymbol('A', 2,2)
>>> A.kind
MatrixKind(NumberKind)
```

Although expression representing a matrix may be not instance of matrix class, it will have `MatrixKind` as well:

```
>>> from sympy import MatrixExpr, Integral
>>> from sympy.abc import x
>>> intM = Integral(A, x)
>>> isinstance(intM, MatrixExpr)
False
>>> intM.kind
MatrixKind(NumberKind)
```

Use `isinstance()` to check for `MatrixKind` without specifying the element kind. Use `is` with specifying the element kind:

```
>>> from sympy import Matrix
>>> from sympy.core import NumberKind
>>> from sympy.matrices import MatrixKind
>>> M = Matrix([1, 2])
>>> isinstance(M.kind, MatrixKind)
True
```

(continues on next page)

(continued from previous page)

```
>>> M.kind is MatrixKind(NumberKind)
True
```

See also:

[sympy.core.kind.NumberKind](#) (page 1074), [sympy.core.kind.UndefinedKind](#) (page 1074), [sympy.core.containers.TupleKind](#) (page 1069), [sympy.sets.sets.SetKind](#) (page 1217)

Dense Matrices**Matrix Class Reference**

`sympy.matrices.dense.Matrix`

alias of [MutableDenseMatrix](#) (page 1363)

class `sympy.matrices.dense.DenseMatrix`

Matrix implementation based on DomainMatrix as the internal representation

LDLdecomposition(*hermitian=True*)

Returns the LDL Decomposition (L, D) of matrix A, such that $L * D * L.H == A$ if hermitian flag is True, or $L * D * L.T == A$ if hermitian is False. This method eliminates the use of square root. Further this ensures that all the diagonal entries of L are 1. A must be a Hermitian positive-definite matrix if hermitian is True, or a symmetric matrix otherwise.

Examples

```
>>> from sympy import Matrix, eye
>>> A = Matrix(((25, 15, -5), (15, 18, 0), (-5, 0, 11)))
>>> L, D = A.LDLdecomposition()
>>> L
Matrix([
[ 1, 0, 0],
[ 3/5, 1, 0],
[-1/5, 1/3, 1]])
>>> D
Matrix([
[25, 0, 0],
[ 0, 9, 0],
[ 0, 0, 9]])
>>> L * D * L.T * A.inv() == eye(A.rows)
True
```

The matrix can have complex entries:

```
>>> from sympy import I
>>> A = Matrix(((9, 3*I), (-3*I, 5)))
>>> L, D = A.LDLdecomposition()
>>> L
```

(continues on next page)

(continued from previous page)

```
Matrix([
[ 1, 0],
[-I/3, 1]])
>>> D
Matrix([
[9, 0],
[0, 4]])
>>> L*D*L.H == A
True
```

See also:

[*sympy.matrices.dense.DenseMatrix.cholesky*](#) (page 1362), [*sympy.matrices.matrices.MatrixBase.LUdecomposition*](#) (page 1282), [*QRdecomposition*](#) (page 1287)

as_immutable()

Returns an Immutable version of this Matrix

as_mutable()

Returns a mutable version of this matrix

Examples

```
>>> from sympy import ImmutableMatrix
>>> X = ImmutableMatrix([[1, 2], [3, 4]])
>>> Y = X.as_mutable()
>>> Y[1, 1] = 5 # Can set values in Y
>>> Y
Matrix([
[1, 2],
[3, 5]])
```

cholesky(hermitian=True)

Returns the Cholesky-type decomposition L of a matrix A such that $L * L.H == A$ if hermitian flag is True, or $L * L.T == A$ if hermitian is False.

A must be a Hermitian positive-definite matrix if hermitian is True, or a symmetric matrix if it is False.

Examples

```
>>> from sympy import Matrix
>>> A = Matrix(((25, 15, -5), (15, 18, 0), (-5, 0, 11)))
>>> A.cholesky()
Matrix([
[ 5, 0, 0],
[ 3, 3, 0],
[-1, 1, 3]])
>>> A.cholesky() * A.cholesky().T
```

(continues on next page)

(continued from previous page)

```
Matrix([
[25, 15, -5],
[15, 18,  0],
[-5,  0, 11]])
```

The matrix can have complex entries:

```
>>> from sympy import I
>>> A = Matrix(((9, 3*I), (-3*I, 5)))
>>> A.cholesky()
Matrix([
[ 3, 0],
[-I, 2]])
>>> A.cholesky() * A.cholesky().H
Matrix([
[ 9, 3*I],
[-3*I,  5]])
```

Non-hermitian Cholesky-type decomposition may be useful when the matrix is not positive-definite.

```
>>> A = Matrix([[1, 2], [2, 1]])
>>> L = A.cholesky(hermitian=False)
>>> L
Matrix([
[1,          0],
[2, sqrt(3)*I]])
>>> L*L.T == A
True
```

See also:

[sympy.matrices.dense.DenseMatrix.LDLdecomposition](#) (page 1361),
[sympy.matrices.matrices.MatrixBase.LUdecomposition](#) (page 1282),
[QRdecomposition](#) (page 1287)

lower_triangular_solve(*rhs*)

Solves $Ax = B$, where A is a lower triangular matrix.

See also:

[upper_triangular_solve](#) (page 1363), [gauss_jordan_solve](#) (page 1297),
[cholesky_solve](#) (page 1292), [diagonal_solve](#) (page 1295), [LDLsolve](#) (page 1282),
[LUsolve](#) (page 1287), [QRsolve](#) (page 1291), [pinv_solve](#) (page 1306)

upper_triangular_solve(*rhs*)

Solves $Ax = B$, where A is an upper triangular matrix.

See also:

[lower_triangular_solve](#) (page 1363), [gauss_jordan_solve](#) (page 1297),
[cholesky_solve](#) (page 1292), [diagonal_solve](#) (page 1295), [LDLsolve](#) (page 1282),
[LUsolve](#) (page 1287), [QRsolve](#) (page 1291), [pinv_solve](#) (page 1306)

class `sympy.matrices.dense.MutableDenseMatrix(*args, **kwargs)`

simplify(**kwargs)

Applies simplify to the elements of a matrix in place.

This is a shortcut for `M.applyfunc(lambda x: simplify(x, ratio, measure))`

See also:

[*sympy.simplify.simplify.simplify*](#) (page 661)

ImmutableMatrix Class Reference

class `sympy.matrices.immutable.ImmutableDenseMatrix(*args, **kwargs)`

Create an immutable version of a matrix.

Examples

```
>>> from sympy import eye, ImmutableMatrix
>>> ImmutableMatrix(eye(3))
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> _[0, 0] = 42
Traceback (most recent call last):
...
TypeError: Cannot set values of ImmutableDenseMatrix
```

Sparse Matrices

SparseMatrix Class Reference

`sympy.matrices.sparse.SparseMatrix`

alias of [*MutableSparseMatrix*](#) (page 1364)

class `sympy.matrices.sparse.MutableSparseMatrix(*args, **kwargs)`

ImmutableSparseMatrix Class Reference

class `sympy.matrices.immutable.ImmutableSparseMatrix(*args, **kwargs)`

Create an immutable version of a sparse matrix.

Examples

```
>>> from sympy import eye, ImmutableSparseMatrix
>>> ImmutableSparseMatrix(1, 1, {})
Matrix([[0]])
>>> ImmutableSparseMatrix(eye(3))
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> _[0, 0] = 42
Traceback (most recent call last):
...
TypeError: Cannot set values of ImmutableSparseMatrix
>>> _.shape
(3, 3)
```

Sparse Tools

`sympy.matrices.sparsetools._doktocsr()`

Converts a sparse matrix to Compressed Sparse Row (CSR) format.

Parameters

A : contains non-zero elements sorted by key (row, column)

JA : JA[i] is the column corresponding to A[i]

IA : IA[i] contains the index in A for the first non-zero element

of row[i]. Thus IA[i+1] - IA[i] gives number of non-zero elements row[i]. The length of IA is always 1 more than the number of rows in the matrix.

Examples

```
>>> from sympy.matrices.sparsetools import _doktocsr
>>> from sympy import SparseMatrix, diag
>>> m = SparseMatrix(diag(1, 2, 3))
>>> m[2, 0] = -1
>>> _doktocsr(m)
[[1, 2, -1, 3], [0, 1, 0, 2], [0, 1, 2, 4], [3, 3]]
```

`sympy.matrices.sparsetools._csrto dok()`

Converts a CSR representation to DOK representation.

Examples

```
>>> from sympy.matrices.sparsetools import _csrtodok
>>> _csrtodok([[5, 8, 3, 6], [0, 1, 2, 1], [0, 0, 2, 3, 4], [4, 3]])
Matrix([
[0, 0, 0],
[5, 8, 0],
[0, 0, 3],
[0, 6, 0]])
```

`sympy.matrices.sparsetools.banded(**kwargs)`

Returns a SparseMatrix from the given dictionary describing the diagonals of the matrix. The keys are positive for upper diagonals and negative for those below the main diagonal. The values may be:

- expressions or single-argument functions,
- lists or tuples of values,
- matrices

Unless dimensions are given, the size of the returned matrix will be large enough to contain the largest non-zero value provided.

Kwargs

rows

[rows of the resulting matrix; computed if] not given.

cols

[columns of the resulting matrix; computed if] not given.

Examples

```
>>> from sympy import banded, ones, Matrix
>>> from sympy.abc import x
```

If explicit values are given in tuples, the matrix will autosize to contain all values, otherwise a single value is filled onto the entire diagonal:

```
>>> banded({1: (1, 2, 3), -1: (4, 5, 6), 0: x})
Matrix([
[x, 1, 0, 0],
[4, x, 2, 0],
[0, 5, x, 3],
[0, 0, 6, x]])
```

A function accepting a single argument can be used to fill the diagonal as a function of diagonal index (which starts at 0). The size (or shape) of the matrix must be given to obtain more than a 1x1 matrix:

```
>>> s = lambda d: (1 + d)**2
>>> banded(5, {0: s, 2: s, -2: 2})
Matrix([
[1, 0, 1, 0, 0],
[0, 4, 0, 4, 0],
[2, 0, 9, 0, 9],
[0, 2, 0, 16, 0],
[0, 0, 2, 0, 25]])
```

The diagonal of matrices placed on a diagonal will coincide with the indicated diagonal:

```
>>> vert = Matrix([1, 2, 3])
>>> banded({0: vert}, cols=3)
Matrix([
[1, 0, 0],
[2, 1, 0],
[3, 2, 1],
[0, 3, 2],
[0, 0, 3]])
```

```
>>> banded(4, {0: ones(2)})
Matrix([
[1, 1, 0, 0],
[1, 1, 0, 0],
[0, 0, 1, 1],
[0, 0, 1, 1]])
```

Errors are raised if the designated size will not hold all values an integral number of times. Here, the rows are designated as odd (but an even number is required to hold the off-diagonal 2x2 ones):

```
>>> banded({0: 2, 1: ones(2)}, rows=5)
Traceback (most recent call last):
...
ValueError:
sequence does not fit an integral number of times in the matrix
```

And here, an even number of rows is given...but the square matrix has an even number of columns, too. As we saw in the previous example, an odd number is required:

```
>>> banded(4, {0: 2, 1: ones(2)}) # trying to make 4x4 and cols must be
↳odd
Traceback (most recent call last):
...
ValueError:
sequence does not fit an integral number of times in the matrix
```

A way around having to count rows is to enclosing matrix elements in a tuple and indicate the desired number of them to the right:

```
>>> banded({0: 2, 2: (ones(2),)*3})
Matrix([
[2, 0, 1, 1, 0, 0, 0, 0],
```

(continues on next page)

(continued from previous page)

```
[0, 2, 1, 1, 0, 0, 0, 0],
[0, 0, 2, 0, 1, 1, 0, 0],
[0, 0, 0, 2, 1, 1, 0, 0],
[0, 0, 0, 0, 2, 0, 1, 1],
[0, 0, 0, 0, 0, 2, 1, 1]])
```

An error will be raised if more than one value is written to a given entry. Here, the ones overlap with the main diagonal if they are placed on the first diagonal:

```
>>> banded({0: (2,)*5, 1: (ones(2),)*3})
Traceback (most recent call last):
...
ValueError: collision at (1, 1)
```

By placing a 0 at the bottom left of the 2x2 matrix of ones, the collision is avoided:

```
>>> u2 = Matrix([
... [1, 1],
... [0, 1]])
>>> banded({0: [2]*5, 1: [u2]*3})
Matrix([
[2, 1, 1, 0, 0, 0, 0, 0],
[0, 2, 1, 0, 0, 0, 0, 0],
[0, 0, 2, 1, 1, 0, 0, 0],
[0, 0, 0, 2, 1, 0, 0, 0],
[0, 0, 0, 0, 2, 1, 1, 1],
[0, 0, 0, 0, 0, 0, 1, 1]])
```

Immutable Matrices

The standard *Matrix* (page 1361) class in SymPy is mutable. This is important for performance reasons but means that standard matrices cannot interact well with the rest of SymPy. This is because the *Basic* (page 927) object, from which most SymPy classes inherit, is immutable.

The mission of the *ImmutableDenseMatrix* (page 1369) class, which is aliased as *ImmutableMatrix* (page 1369) for short, is to bridge the tension between performance/mutability and safety/immutability. Immutable matrices can do almost everything that normal matrices can do but they inherit from *Basic* (page 927) and can thus interact more naturally with the rest of SymPy. *ImmutableMatrix* (page 1369) also inherits from *MatrixExpr* (page 1370), allowing it to interact freely with SymPy's Matrix Expression module.

You can turn any Matrix-like object into an *ImmutableMatrix* (page 1369) by calling the constructor

```
>>> from sympy import Matrix, ImmutableMatrix
>>> M = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> M[1, 1] = 0
>>> IM = ImmutableMatrix(M)
>>> IM
Matrix([
[1, 2, 3],
```

(continues on next page)

(continued from previous page)

```
[4, 0, 6],
[7, 8, 9]])
>>> IM[1, 1] = 5
Traceback (most recent call last):
...
TypeError: Can not set values in Immutable Matrix. Use Matrix instead.
```

ImmutableMatrix Class Reference

`sympy.matrices.immutable.ImmutableMatrix`
 alias of *ImmutableDenseMatrix* (page 1369)

class `sympy.matrices.immutable.ImmutableDenseMatrix(*args, **kwargs)`
 Create an immutable version of a matrix.

Examples

```
>>> from sympy import eye, ImmutableMatrix
>>> ImmutableMatrix(eye(3))
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> _[0, 0] = 42
Traceback (most recent call last):
...
TypeError: Cannot set values of ImmutableDenseMatrix
```

Matrix Expressions

The Matrix expression module allows users to write down statements like

```
>>> from sympy import MatrixSymbol, Matrix
>>> X = MatrixSymbol('X', 3, 3)
>>> Y = MatrixSymbol('Y', 3, 3)
>>> (X.T*X).I*Y
X**(-1)*X.T**(-1)*Y
```

```
>>> Matrix(X)
Matrix([
[X[0, 0], X[0, 1], X[0, 2]],
[X[1, 0], X[1, 1], X[1, 2]],
[X[2, 0], X[2, 1], X[2, 2]]])
```

```
>>> (X*Y)[1, 2]
X[1, 0]*Y[0, 2] + X[1, 1]*Y[1, 2] + X[1, 2]*Y[2, 2]
```

where X and Y are [MatrixSymbol](#) (page 1372)'s rather than scalar symbols.

Matrix expression derivatives are supported. The derivative of a matrix by another matrix is generally a 4-dimensional array, but if some dimensions are trivial or diagonal, the derivation algorithm will try to express the result as a matrix expression:

```
>>> a = MatrixSymbol("a", 3, 1)
>>> b = MatrixSymbol("b", 3, 1)
>>> (a.T*X**2*b).diff(X)
a*b.T*X.T + X.T*a*b.T
```

```
>>> X.diff(X)
PermuteDims(ArrayTensorProduct(I, I), (3)(1 2))
```

The last output is an array expression, as the returned symbol is 4-dimensional.

Matrix Expressions Core Reference

class `sympy.matrices.expressions.MatrixExpr(*args, **kwargs)`

Superclass for Matrix Expressions

MatrixExprs represent abstract matrices, linear transformations represented within a particular basis.

Examples

```
>>> from sympy import MatrixSymbol
>>> A = MatrixSymbol('A', 3, 3)
>>> y = MatrixSymbol('y', 3, 1)
>>> x = (A.T*A).I * A * y
```

See also:

[MatrixSymbol](#) (page 1372), [MatAdd](#) (page 1372), [MatMul](#) (page 1373), [Transpose](#) (page 1375), [Inverse](#) (page 1374)

property `T`

Matrix transposition

as_coeff_Mul(*rational=False*)

Efficiently extract the coefficient of a product.

as_explicit()

Returns a dense Matrix with elements represented explicitly

Returns an object of type `ImmutableDenseMatrix`.

Examples

```
>>> from sympy import Identity
>>> I = Identity(3)
>>> I
I
>>> I.as_explicit()
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
```

See also:

[as_mutable](#) (page 1371)
returns mutable Matrix type

as_mutable()

Returns a dense, mutable matrix with elements represented explicitly

Examples

```
>>> from sympy import Identity
>>> I = Identity(3)
>>> I
I
>>> I.shape
(3, 3)
>>> I.as_mutable()
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
```

See also:

[as_explicit](#) (page 1370)
returns ImmutableDenseMatrix

equals(*other*)

Test elementwise equality between matrices, potentially of different types

```
>>> from sympy import Identity, eye
>>> Identity(3).equals(eye(3))
True
```

static from_index_summation(*expr*, *first_index*=None, *last_index*=None, *dimensions*=None)

Parse expression of matrices with explicitly summed indices into a matrix expression without indices, if possible.

This transformation expressed in mathematical notation:

$$\sum_{j=0}^{N-1} A_{i,j} B_{j,k} \implies \mathbf{A} \cdot \mathbf{B}$$

Optional parameter `first_index`: specify which free index to use as the index starting the expression.

Examples

```
>>> from sympy import MatrixSymbol, MatrixExpr, Sum
>>> from sympy.abc import i, j, k, l, N
>>> A = MatrixSymbol("A", N, N)
>>> B = MatrixSymbol("B", N, N)
>>> expr = Sum(A[i, j]*B[j, k], (j, 0, N-1))
>>> MatrixExpr.from_index_summation(expr)
A*B
```

Transposition is detected:

```
>>> expr = Sum(A[j, i]*B[j, k], (j, 0, N-1))
>>> MatrixExpr.from_index_summation(expr)
A.T*B
```

Detect the trace:

```
>>> expr = Sum(A[i, i], (i, 0, N-1))
>>> MatrixExpr.from_index_summation(expr)
Trace(A)
```

More complicated expressions:

```
>>> expr = Sum(A[i, j]*B[k, j]*A[l, k], (j, 0, N-1), (k, 0, N-1))
>>> MatrixExpr.from_index_summation(expr)
A*B.T*A.T
```

class `sympy.matrices.expressions.MatrixSymbol(name, n, m)`

Symbolic representation of a Matrix object

Creates a SymPy Symbol to represent a Matrix. This matrix has a shape and can be included in Matrix Expressions

Examples

```
>>> from sympy import MatrixSymbol, Identity
>>> A = MatrixSymbol('A', 3, 4) # A 3 by 4 Matrix
>>> B = MatrixSymbol('B', 4, 3) # A 4 by 3 Matrix
>>> A.shape
(3, 4)
>>> 2*A*B + Identity(3)
I + 2*A*B
```

class `sympy.matrices.expressions.MatAdd(*args, evaluate=False, check=None, _sympify=True)`

A Sum of Matrix Expressions

MatAdd inherits from and operates like SymPy Add

Examples

```
>>> from sympy import MatAdd, MatrixSymbol
>>> A = MatrixSymbol('A', 5, 5)
>>> B = MatrixSymbol('B', 5, 5)
>>> C = MatrixSymbol('C', 5, 5)
>>> MatAdd(A, B, C)
A + B + C
```

```
class sympy.matrices.expressions.MatMul(*args, evaluate=False, check=None,
                                         _sympify=True)
```

A product of matrix expressions

Examples

```
>>> from sympy import MatMul, MatrixSymbol
>>> A = MatrixSymbol('A', 5, 4)
>>> B = MatrixSymbol('B', 4, 3)
>>> C = MatrixSymbol('C', 3, 6)
>>> MatMul(A, B, C)
A*B*C
```

```
class sympy.matrices.expressions.MatPow(base, exp, evaluate=False, **options)
```

```
class sympy.matrices.expressions.HadamardProduct(*args, evaluate=False,
                                                  check=None)
```

Elementwise product of matrix expressions

Examples

Hadamard product for matrix symbols:

```
>>> from sympy import hadamard_product, HadamardProduct, MatrixSymbol
>>> A = MatrixSymbol('A', 5, 5)
>>> B = MatrixSymbol('B', 5, 5)
>>> isinstance(hadamard_product(A, B), HadamardProduct)
True
```

Notes

This is a symbolic object that simply stores its argument without evaluating it. To actually compute the product, use the function `hadamard_product()` or `HadamardProduct.doit`

class `sympy.matrices.expressions.HadamardPower(base, exp)`

Elementwise power of matrix expressions

Parameters

base : scalar or matrix

exp : scalar or matrix

Notes

There are four definitions for the hadamard power which can be used. Let's consider A, B as (m, n) matrices, and a, b as scalars.

Matrix raised to a scalar exponent:

$$A^{\circ b} = \begin{bmatrix} A_{0,0}^b & A_{0,1}^b & \cdots & A_{0,n-1}^b \\ A_{1,0}^b & A_{1,1}^b & \cdots & A_{1,n-1}^b \\ \vdots & \vdots & \ddots & \vdots \\ A_{m-1,0}^b & A_{m-1,1}^b & \cdots & A_{m-1,n-1}^b \end{bmatrix}$$

Scalar raised to a matrix exponent:

$$a^{\circ B} = \begin{bmatrix} a^{B_{0,0}} & a^{B_{0,1}} & \cdots & a^{B_{0,n-1}} \\ a^{B_{1,0}} & a^{B_{1,1}} & \cdots & a^{B_{1,n-1}} \\ \vdots & \vdots & \ddots & \vdots \\ a^{B_{m-1,0}} & a^{B_{m-1,1}} & \cdots & a^{B_{m-1,n-1}} \end{bmatrix}$$

Matrix raised to a matrix exponent:

$$A^{\circ B} = \begin{bmatrix} A_{0,0}^{B_{0,0}} & A_{0,1}^{B_{0,1}} & \cdots & A_{0,n-1}^{B_{0,n-1}} \\ A_{1,0}^{B_{1,0}} & A_{1,1}^{B_{1,1}} & \cdots & A_{1,n-1}^{B_{1,n-1}} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m-1,0}^{B_{m-1,0}} & A_{m-1,1}^{B_{m-1,1}} & \cdots & A_{m-1,n-1}^{B_{m-1,n-1}} \end{bmatrix}$$

Scalar raised to a scalar exponent:

$$a^{\circ b} = a^b$$

class `sympy.matrices.expressions.Inverse(mat, exp=-1)`

The multiplicative inverse of a matrix expression

This is a symbolic object that simply stores its argument without evaluating it. To actually compute the inverse, use the `.inverse()` method of matrices.

Examples

```
>>> from sympy import MatrixSymbol, Inverse
>>> A = MatrixSymbol('A', 3, 3)
>>> B = MatrixSymbol('B', 3, 3)
>>> Inverse(A)
A**(-1)
>>> A.inverse() == Inverse(A)
True
>>> (A*B).inverse()
B**(-1)*A**(-1)
>>> Inverse(A*B)
(A*B)**(-1)
```

class sympy.matrices.expressions.**Transpose**(*args, **kwargs)

The transpose of a matrix expression.

This is a symbolic object that simply stores its argument without evaluating it. To actually compute the transpose, use the `transpose()` function, or the `.T` attribute of matrices.

Examples

```
>>> from sympy import MatrixSymbol, Transpose, transpose
>>> A = MatrixSymbol('A', 3, 5)
>>> B = MatrixSymbol('B', 5, 3)
>>> Transpose(A)
A.T
>>> A.T == transpose(A) == Transpose(A)
True
>>> Transpose(A*B)
(A*B).T
>>> transpose(A*B)
B.T*A.T
```

class sympy.matrices.expressions.**Trace**(mat)

Matrix Trace

Represents the trace of a matrix expression.

Examples

```
>>> from sympy import MatrixSymbol, Trace, eye
>>> A = MatrixSymbol('A', 3, 3)
>>> Trace(A)
Trace(A)
>>> Trace(eye(3))
Trace(Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]]))
```

(continues on next page)

(continued from previous page)

```
>>> Trace(eye(3)).simplify()
3
```

class sympy.matrices.expressions.**FunctionMatrix**(rows, cols, lamda)

Represents a matrix using a function (Lambda) which gives outputs according to the coordinates of each matrix entries.

Parameters

rows : nonnegative integer. Can be symbolic.

cols : nonnegative integer. Can be symbolic.

lamda : Function, Lambda or str

If it is a SymPy Function or Lambda instance, it should be able to accept two arguments which represents the matrix coordinates.

If it is a pure string containing Python lambda semantics, it is interpreted by the SymPy parser and casted into a SymPy Lambda instance.

Examples

Creating a FunctionMatrix from Lambda:

```
>>> from sympy import FunctionMatrix, symbols, Lambda, MatPow
>>> i, j, n, m = symbols('i,j,n,m')
>>> FunctionMatrix(n, m, Lambda((i, j), i + j))
FunctionMatrix(n, m, Lambda((i, j), i + j))
```

Creating a FunctionMatrix from a SymPy function:

```
>>> from sympy import KroneckerDelta
>>> X = FunctionMatrix(3, 3, KroneckerDelta)
>>> X.as_explicit()
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
```

Creating a FunctionMatrix from a SymPy undefined function:

```
>>> from sympy import Function
>>> f = Function('f')
>>> X = FunctionMatrix(3, 3, f)
>>> X.as_explicit()
Matrix([
[f(0, 0), f(0, 1), f(0, 2)],
[f(1, 0), f(1, 1), f(1, 2)],
[f(2, 0), f(2, 1), f(2, 2)]])
```

Creating a FunctionMatrix from Python lambda:

```
>>> FunctionMatrix(n, m, 'lambda i, j: i + j')
FunctionMatrix(n, m, Lambda((i, j), i + j))
```


Example of lazy evaluation of matrix product:

```
>>> Y = FunctionMatrix(1000, 1000, Lambda((i, j), i + j))
>>> isinstance(Y*Y, MatPow) # this is an expression object
True
>>> (Y**2)[10,10] # So this is evaluated lazily
342923500
```

Notes

This class provides an alternative way to represent an extremely dense matrix with entries in some form of a sequence, in a most sparse way.

class sympy.matrices.expressions.**PermutationMatrix**(perm)

A Permutation Matrix

Parameters

perm : Permutation

The permutation the matrix uses.

The size of the permutation determines the matrix size.

See the documentation of [sympy.combinatorics.permutations.Permutation](#) (page 257) for the further information of how to create a permutation object.

Examples

```
>>> from sympy import Matrix, PermutationMatrix
>>> from sympy.combinatorics import Permutation
```

Creating a permutation matrix:

```
>>> p = Permutation(1, 2, 0)
>>> P = PermutationMatrix(p)
>>> P = P.as_explicit()
>>> P
Matrix([
[0, 1, 0],
[0, 0, 1],
[1, 0, 0]])
```

Permuting a matrix row and column:

```
>>> M = Matrix([0, 1, 2])
>>> Matrix(P*M)
Matrix([
[1],
[2],
[0]])
```

```
>>> Matrix(M.T*P)
Matrix([[2, 0, 1]])
```

See also:

[*sympy.combinatorics.permutations.Permutation*](#) (page 257)

class `sympy.matrices.expressions.MatrixPermute(mat, perm, axis=0)`

Symbolic representation for permuting matrix rows or columns.

Parameters

perm : `Permutation`, `PermutationMatrix`

The permutation to use for permuting the matrix. The permutation can be resized to the suitable one,

axis : 0 or 1

The axis to permute alongside. If 0, it will permute the matrix rows. If 1, it will permute the matrix columns.

Notes

This follows the same notation used in [*sympy.matrices.common.MatrixCommon.permute\(\)*](#) (page 1348).

Examples

```
>>> from sympy import Matrix, MatrixPermute
>>> from sympy.combinatorics import Permutation
```

Permuting the matrix rows:

```
>>> p = Permutation(1, 2, 0)
>>> A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> B = MatrixPermute(A, p, axis=0)
>>> B.as_explicit()
Matrix([
[4, 5, 6],
[7, 8, 9],
[1, 2, 3]])
```

Permuting the matrix columns:

```
>>> B = MatrixPermute(A, p, axis=1)
>>> B.as_explicit()
Matrix([
[2, 3, 1],
[5, 6, 4],
[8, 9, 7]])
```

See also:

[*sympy.matrices.common.MatrixCommon.permute*](#) (page 1348)

class sympy.matrices.expressions.**Identity**(*n*)

The Matrix Identity I - multiplicative identity

Examples

```
>>> from sympy import Identity, MatrixSymbol
>>> A = MatrixSymbol('A', 3, 5)
>>> I = Identity(3)
>>> I*A
A
```

class sympy.matrices.expressions.**ZeroMatrix**(*m, n*)

The Matrix Zero 0 - additive identity

Examples

```
>>> from sympy import MatrixSymbol, ZeroMatrix
>>> A = MatrixSymbol('A', 3, 5)
>>> Z = ZeroMatrix(3, 5)
>>> A + Z
A
>>> Z*A.T
0
```

class sympy.matrices.expressions.**CompanionMatrix**(*poly*)

A symbolic companion matrix of a polynomial.

Examples

```
>>> from sympy import Poly, Symbol, symbols
>>> from sympy.matrices.expressions import CompanionMatrix
>>> x = Symbol('x')
>>> c0, c1, c2, c3, c4 = symbols('c0:5')
>>> p = Poly(c0 + c1*x + c2*x**2 + c3*x**3 + c4*x**4 + x**5, x)
>>> CompanionMatrix(p)
CompanionMatrix(Poly(x**5 + c4*x**4 + c3*x**3 + c2*x**2 + c1*x + c0,
x, domain='ZZ[c0,c1,c2,c3,c4]'))
```

class sympy.matrices.expressions.**MatrixSet**(*n, m, set*)

MatrixSet represents the set of matrices with shape = (*n*, *m*) over the given set.

Examples

```
>>> from sympy.matrices import MatrixSet
>>> from sympy import S, I, Matrix
>>> M = MatrixSet(2, 2, set=S.Reals)
>>> X = Matrix([[1, 2], [3, 4]])
>>> X in M
True
>>> X = Matrix([[1, 2], [I, 4]])
>>> X in M
False
```

Block Matrices

Block matrices allow you to construct larger matrices out of smaller sub-blocks. They can work with *MatrixExpr* (page 1370) or *ImmutableMatrix* (page 1369) objects.

class sympy.matrices.expressions.blockmatrix.**BlockMatrix**(*args, **kwargs)

A BlockMatrix is a Matrix comprised of other matrices.

The submatrices are stored in a SymPy Matrix object but accessed as part of a Matrix Expression

```
>>> from sympy import (MatrixSymbol, BlockMatrix, symbols,
...     Identity, ZeroMatrix, block_collapse)
>>> n,m,l = symbols('n m l')
>>> X = MatrixSymbol('X', n, n)
>>> Y = MatrixSymbol('Y', m, m)
>>> Z = MatrixSymbol('Z', n, m)
>>> B = BlockMatrix([[X, Z], [ZeroMatrix(m,n), Y]])
>>> print(B)
Matrix([
[X, Z],
[0, Y]])
```

```
>>> C = BlockMatrix([[Identity(n), Z]])
>>> print(C)
Matrix([[I, Z]])
```

```
>>> print(block_collapse(C*B))
Matrix([[X, Z + Z*Y]])
```

Some matrices might be comprised of rows of blocks with the matrices in each row having the same height and the rows all having the same total number of columns but not having the same number of columns for each matrix in each row. In this case, the matrix is not a block matrix and should be instantiated by Matrix.

```
>>> from sympy import ones, Matrix
>>> dat = [
...     [ones(3,2), ones(3,3)*2],
...     [ones(2,3)*3, ones(2,2)*4]]
... 
```

(continues on next page)

(continued from previous page)

```
>>> BlockMatrix(dat)
Traceback (most recent call last):
...
ValueError:
Although this matrix is comprised of blocks, the blocks do not fill
the matrix in a size-symmetric fashion. To create a full matrix from
these arguments, pass them directly to Matrix.
>>> Matrix(dat)
Matrix([
[1, 1, 2, 2, 2],
[1, 1, 2, 2, 2],
[1, 1, 2, 2, 2],
[3, 3, 3, 4, 4],
[3, 3, 3, 4, 4]])
```

See also:

[*sympy.matrices.matrices.MatrixBase.irregular*](#) (page 1302)

LDUdecomposition()

Returns the Block LDU decomposition of a 2x2 Block Matrix

Returns

(**L**, **D**, **U**) : Matrices

L : Lower Diagonal Matrix D : Diagonal Matrix U : Upper Diagonal Matrix

Raises**ShapeError**

If the block matrix is not a 2x2 matrix

NonInvertibleMatrixError

If the matrix “A” is non-invertible

Examples

```
>>> from sympy import symbols, MatrixSymbol, BlockMatrix, block_
    collapse
>>> m, n = symbols('m n')
>>> A = MatrixSymbol('A', n, n)
>>> B = MatrixSymbol('B', n, m)
>>> C = MatrixSymbol('C', m, n)
>>> D = MatrixSymbol('D', m, m)
>>> X = BlockMatrix([[A, B], [C, D]])
>>> L, D, U = X.LDUdecomposition()
>>> block_collapse(L*D*U)
Matrix([
[A, B],
[C, D]])
```

See also:

[*sympy.matrices.expressions.blockmatrix.BlockMatrix.UDLdecomposition*](#) (page 1382), [*sympy.matrices.expressions.blockmatrix.BlockMatrix.LUdecomposition*](#) (page 1382)

LUdecomposition()

Returns the Block LU decomposition of a 2x2 Block Matrix

Returns

(L, U) : Matrices

L : Lower Diagonal Matrix U : Upper Diagonal Matrix

Raises**ShapeError**

If the block matrix is not a 2x2 matrix

NonInvertibleMatrixError

If the matrix "A" is non-invertible

Examples

```
>>> from sympy import symbols, MatrixSymbol, BlockMatrix, block_
    collapse
>>> m, n = symbols('m n')
>>> A = MatrixSymbol('A', n, n)
>>> B = MatrixSymbol('B', n, m)
>>> C = MatrixSymbol('C', m, n)
>>> D = MatrixSymbol('D', m, m)
>>> X = BlockMatrix([[A, B], [C, D]])
>>> L, U = X.LUdecomposition()
>>> block_collapse(L*U)
Matrix([
[A, B],
[C, D]])
```

See also:

[*sympy.matrices.expressions.blockmatrix.BlockMatrix.UDLdecomposition*](#) (page 1382), [*sympy.matrices.expressions.blockmatrix.BlockMatrix.LDUdecomposition*](#) (page 1381)

UDLdecomposition()

Returns the Block UDL decomposition of a 2x2 Block Matrix

Returns

(U, D, L) : Matrices

U : Upper Diagonal Matrix D : Diagonal Matrix L : Lower Diagonal Matrix

Raises**ShapeError**

If the block matrix is not a 2x2 matrix

NonInvertibleMatrixError

If the matrix “D” is non-invertible

Examples

```
>>> from sympy import symbols, MatrixSymbol, BlockMatrix, block_
    collapse
>>> m, n = symbols('m n')
>>> A = MatrixSymbol('A', n, n)
>>> B = MatrixSymbol('B', n, m)
>>> C = MatrixSymbol('C', m, n)
>>> D = MatrixSymbol('D', m, m)
>>> X = BlockMatrix([[A, B], [C, D]])
>>> U, D, L = X.UDLdecomposition()
>>> block_collapse(U*D*L)
Matrix([
[A, B],
[C, D]])
```

See also:

[*sympy.matrices.expressions.blockmatrix.BlockMatrix.LDUdecomposition*](#)
(page 1381), [*sympy.matrices.expressions.blockmatrix.BlockMatrix.LUdecomposition*](#) (page 1382)

schur(*mat*='A', *generalized*=False)

Return the Schur Complement of the 2x2 BlockMatrix

Parameters

mat : String, optional

The matrix with respect to which the Schur Complement is calculated. 'A' is used by default

generalized : bool, optional

If True, returns the generalized Schur Component which uses Moore-Penrose Inverse

Returns

M : Matrix

The Schur Complement Matrix

Raises

ShapeError

If the block matrix is not a 2x2 matrix

NonInvertibleMatrixError

If given matrix is non-invertible

Examples

```
>>> from sympy import symbols, MatrixSymbol, BlockMatrix
>>> m, n = symbols('m n')
>>> A = MatrixSymbol('A', n, n)
>>> B = MatrixSymbol('B', n, m)
>>> C = MatrixSymbol('C', m, n)
>>> D = MatrixSymbol('D', m, m)
>>> X = BlockMatrix([[A, B], [C, D]])
```

The default Schur Complement is evaluated with “A”

```
>>> X.schur()
-C*A**(-1)*B + D
>>> X.schur('D')
A - B*D**(-1)*C
```

Schur complement with non-invertible matrices is not defined. Instead, the generalized Schur complement can be calculated which uses the Moore-Penrose Inverse. To achieve this, *generalized* must be set to *True*

```
>>> X.schur('B', generalized=True)
C - D*(B.T*B)**(-1)*B.T*A
>>> X.schur('C', generalized=True)
-A*(C.T*C)**(-1)*C.T*D + B
```

See also:

[*sympy.matrices.matrices.MatrixBase.pinv*](#) (page 1305)

References

[R565]

transpose()

Return transpose of matrix.

Examples

```
>>> from sympy import MatrixSymbol, BlockMatrix, ZeroMatrix
>>> from sympy.abc import m, n
>>> X = MatrixSymbol('X', n, n)
>>> Y = MatrixSymbol('Y', m, m)
>>> Z = MatrixSymbol('Z', n, m)
>>> B = BlockMatrix([[X, Z], [ZeroMatrix(m,n), Y]])
>>> B.transpose()
Matrix([
[X.T, 0],
[Z.T, Y.T]])
>>> _.transpose()
Matrix([
```

(continues on next page)

(continued from previous page)

```
[X, Z],
[0, Y]])
```

class `sympy.matrices.expressions.blockmatrix.BlockDiagMatrix(*mats)`

A sparse matrix with block matrices along its diagonals

Examples

```
>>> from sympy import MatrixSymbol, BlockDiagMatrix, symbols
>>> n, m, l = symbols('n m l')
>>> X = MatrixSymbol('X', n, n)
>>> Y = MatrixSymbol('Y', m, m)
>>> BlockDiagMatrix(X, Y)
Matrix([
[X, 0],
[0, Y]])
```

Notes

If you want to get the individual diagonal blocks, use [`get\_diag\_blocks\(\)`](#) (page 1385).

See also:

[`sympy.matrices.dense.diag`](#) (page 1319)

`get_diag_blocks()`

Return the list of diagonal blocks of the matrix.

Examples

```
>>> from sympy import BlockDiagMatrix, Matrix
```

```
>>> A = Matrix([[1, 2], [3, 4]])
>>> B = Matrix([[5, 6], [7, 8]])
>>> M = BlockDiagMatrix(A, B)
```

How to get diagonal blocks from the block diagonal matrix:

```
>>> diag_blocks = M.get_diag_blocks()
>>> diag_blocks[0]
Matrix([
[1, 2],
[3, 4]])
>>> diag_blocks[1]
Matrix([
[5, 6],
[7, 8]])
```

`sympy.matrices.expressions.blockmatrix.block_collapse(expr)`

Evaluates a block matrix expression

```
>>> from sympy import MatrixSymbol, BlockMatrix, symbols, Identity, \
    ZeroMatrix, block_collapse
>>> n,m,l = symbols('n m l')
>>> X = MatrixSymbol('X', n, n)
>>> Y = MatrixSymbol('Y', m, m)
>>> Z = MatrixSymbol('Z', n, m)
>>> B = BlockMatrix([[X, Z], [ZeroMatrix(m, n), Y]])
>>> print(B)
Matrix([
[X, Z],
[0, Y]])
```

```
>>> C = BlockMatrix([[Identity(n), Z]])
>>> print(C)
Matrix([[I, Z]])
```

```
>>> print(block_collapse(C*B))
Matrix([[X, Z + Z*Y]])
```

Matrix Normal Forms

`sympy.matrices.normalforms.smith_normal_form(m, domain=None)`

Return the Smith Normal Form of a matrix m over the ring $domain$. This will only work if the ring is a principal ideal domain.

Examples

```
>>> from sympy import Matrix, ZZ
>>> from sympy.matrices.normalforms import smith_normal_form
>>> m = Matrix([[12, 6, 4], [3, 9, 6], [2, 16, 14]])
>>> print(smith_normal_form(m, domain=ZZ))
Matrix([[1, 0, 0], [0, 10, 0], [0, 0, -30]])
```

`sympy.matrices.normalforms.hermite_normal_form(A, *, D=None, check_rank=False)`

Compute the Hermite Normal Form of a Matrix A of integers.

Parameters

A : $m \times n$ Matrix of integers.

D : int, optional

Let W be the HNF of A . If known in advance, a positive integer D being any multiple of $\det(W)$ may be provided. In this case, if A also has rank m , then we may use an alternative algorithm that works mod D in order to prevent coefficient explosion.

check_rank : boolean, optional (default=False)

The basic assumption is that, if you pass a value for D , then you already believe that A has rank m , so we do not waste time checking it for you. If you do want this to be checked (and the ordinary, non-modulo D algorithm to be used if the check fails), then set `check_rank` to `True`.

Returns

Matrix

The HNF of matrix A .

Raises

DMDomainError

If the domain of the matrix is not `ZZ` (page 2525).

DMShapeError

If the mod D algorithm is used but the matrix has more rows than columns.

Examples

```
>>> from sympy import Matrix
>>> from sympy.matrices.normalforms import hermite_normal_form
>>> m = Matrix([[12, 6, 4], [3, 9, 6], [2, 16, 14]])
>>> print(hermite_normal_form(m))
Matrix([[10, 0, 2], [0, 15, 3], [0, 0, 2]])
```

References

[R598]

Tensor

A module to manipulate symbolic objects with indices including tensors

Contents

N-dim array

N-dim array module for SymPy.

Four classes are provided to handle N-dim arrays, given by the combinations dense/sparse (i.e. whether to store all elements or only the non-zero ones in memory) and mutable/immutable (immutable classes are SymPy objects, but cannot change after they have been created).

Examples

The following examples show the usage of `Array`. This is an abbreviation for `ImmutableDenseNDimArray`, that is an immutable and dense N-dim array, the other classes are analogous. For mutable classes it is also possible to change element values after the object has been constructed.

Array construction can detect the shape of nested lists and tuples:

```
>>> from sympy import Array
>>> a1 = Array([[1, 2], [3, 4], [5, 6]])
>>> a1
[[1, 2], [3, 4], [5, 6]]
>>> a1.shape
(3, 2)
>>> a1.rank()
2
>>> from sympy.abc import x, y, z
>>> a2 = Array([[[x, y], [z, x*z]], [[1, x*y], [1/x, x/y]]])
>>> a2
[[[x, y], [z, x*z]], [[1, x*y], [1/x, x/y]]]
>>> a2.shape
(2, 2, 2)
>>> a2.rank()
3
```

Otherwise one could pass a 1-dim array followed by a shape tuple:

```
>>> m1 = Array(range(12), (3, 4))
>>> m1
[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
>>> m2 = Array(range(12), (3, 2, 2))
>>> m2
[[[0, 1], [2, 3]], [[4, 5], [6, 7]], [[8, 9], [10, 11]]]
>>> m2[1,1,1]
7
>>> m2.reshape(4, 3)
[[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10, 11]]
```

Slice support:

```
>>> m2[:, 1, 1]
[3, 7, 11]
```

Elementwise derivative:

```
>>> from sympy.abc import x, y, z
>>> m3 = Array([x**3, x*y, z])
>>> m3.diff(x)
[3*x**2, y, 0]
>>> m3.diff(z)
[0, 0, 1]
```

Multiplication with other SymPy expressions is applied elementwisely:

```
>>> (1+x)*m3
[x**3*(x + 1), x*y*(x + 1), z*(x + 1)]
```

To apply a function to each element of the N-dim array, use `applyfunc`:

```
>>> m3.applyfunc(lambda x: x/2)
[x**3/2, x*y/2, z/2]
```

N-dim arrays can be converted to nested lists by the `tolist()` method:

```
>>> m2.tolist()
[[[0, 1], [2, 3]], [[4, 5], [6, 7]], [[8, 9], [10, 11]]]
>>> isinstance(m2.tolist(), list)
True
```

If the rank is 2, it is possible to convert them to matrices with `tomatrix()`:

```
>>> m1.tomatrix()
Matrix([
[0, 1, 2, 3],
[4, 5, 6, 7],
[8, 9, 10, 11]])
```

Products and contractions

Tensor product between arrays $A_{i_1, \dots, i_n}$ and $B_{j_1, \dots, j_m}$ creates the combined array $P = A \otimes B$ defined as

$$P_{i_1, \dots, i_n, j_1, \dots, j_m} := A_{i_1, \dots, i_n} \cdot B_{j_1, \dots, j_m}.$$

It is available through `tensorproduct(...)`:

```
>>> from sympy import Array, tensorproduct
>>> from sympy.abc import x,y,z,t
>>> A = Array([x, y, z, t])
>>> B = Array([1, 2, 3, 4])
>>> tensorproduct(A, B)
[[x, 2*x, 3*x, 4*x], [y, 2*y, 3*y, 4*y], [z, 2*z, 3*z, 4*z], [t, 2*t, 3*t,
↪ 4*t]]
```

Tensor product between a rank-1 array and a matrix creates a rank-3 array:

```
>>> from sympy import eye
>>> p1 = tensorproduct(A, eye(4))
>>> p1
[[[x, 0, 0, 0], [0, x, 0, 0], [0, 0, x, 0], [0, 0, 0, x]], [[y, 0, 0, 0], [0, y,
↪ 0, 0], [0, 0, y, 0], [0, 0, 0, y]], [[z, 0, 0, 0], [0, z, 0, 0], [0, 0,
↪ z, 0], [0, 0, 0, z]], [[t, 0, 0, 0], [0, t, 0, 0], [0, 0, t, 0], [0, 0, 0,
↪ t]]]
```

Now, to get back $A_0 \otimes \mathbf{1}$ one can access $p_{0,m,n}$ by slicing:

```
>>> p1[0, :, :]
[[x, 0, 0, 0], [0, x, 0, 0], [0, 0, x, 0], [0, 0, 0, x]]
```

Tensor contraction sums over the specified axes, for example contracting positions a and b means

$$A_{i_1, \dots, i_a, \dots, i_b, \dots, i_n} \Rightarrow \sum_k A_{i_1, \dots, k, \dots, k, \dots, i_n}$$

Remember that Python indexing is zero starting, to contract the a -th and b -th axes it is therefore necessary to specify $a - 1$ and $b - 1$

```
>>> from sympy import tensorcontraction
>>> C = Array([[x, y], [z, t]])
```

The matrix trace is equivalent to the contraction of a rank-2 array:

$$A_{m,n} \Rightarrow \sum_k A_{k,k}$$

```
>>> tensorcontraction(C, (0, 1))
t + x
```

Matrix product is equivalent to a tensor product of two rank-2 arrays, followed by a contraction of the 2nd and 3rd axes (in Python indexing axes number 1, 2).

$$A_{m,n} \cdot B_{i,j} \Rightarrow \sum_k A_{m,k} \cdot B_{k,j}$$

```
>>> D = Array([[2, 1], [0, -1]])
>>> tensorcontraction(tensorproduct(C, D), (1, 2))
[[2*x, x - y], [2*z, -t + z]]
```

One may verify that the matrix product is equivalent:

```
>>> from sympy import Matrix
>>> Matrix([[x, y], [z, t]])*Matrix([[2, 1], [0, -1]])
Matrix([
 [2*x, x - y],
 [2*z, -t + z]])
```

or equivalently

```
>>> C.tomatrix()*D.tomatrix()
Matrix([
 [2*x, x - y],
 [2*z, -t + z]])
```

Diagonal operator

The `tensordiagonal` function acts in a similar manner as `tensorcontraction`, but the joined indices are not summed over, for example diagonalizing positions a and b means

$$A_{i_1, \dots, i_a, \dots, i_b, \dots, i_n} \Rightarrow A_{i_1, \dots, k, \dots, k, \dots, i_n} \Rightarrow \tilde{A}_{i_1, \dots, i_{a-1}, i_{a+1}, \dots, i_{b-1}, i_{b+1}, \dots, i_n, k}$$

where $\tilde{A}$ is the array equivalent to the diagonal of A at positions a and b moved to the last index slot.

Compare the difference between contraction and diagonal operators:

```
>>> from sympy import tensordiagonal
>>> from sympy.abc import a, b, c, d
>>> m = Matrix([[a, b], [c, d]])
>>> tensorcontraction(m, [0, 1])
a + d
>>> tensordiagonal(m, [0, 1])
[a, d]
```

In short, no summation occurs with `tensordiagonal`.

Derivatives by array

The usual derivative operation may be extended to support derivation with respect to arrays, provided that all elements in the that array are symbols or expressions suitable for derivations.

The definition of a derivative by an array is as follows: given the array $A_{i_1, \dots, i_N}$ and the array $X_{j_1, \dots, j_M}$ the derivative of arrays will return a new array B defined by

$$B_{j_1, \dots, j_M, i_1, \dots, i_N} := \frac{\partial A_{i_1, \dots, i_N}}{\partial X_{j_1, \dots, j_M}}$$

The function `derive_by_array` performs such an operation:

```
>>> from sympy import derive_by_array
>>> from sympy.abc import x, y, z, t
>>> from sympy import sin, exp
```

With scalars, it behaves exactly as the ordinary derivative:

```
>>> derive_by_array(sin(x*y), x)
y*cos(x*y)
```

Scalar derived by an array basis:

```
>>> derive_by_array(sin(x*y), [x, y, z])
[y*cos(x*y), x*cos(x*y), 0]
```

Deriving array by an array basis: $B^{nm} := \frac{\partial A^m}{\partial x^n}$

```
>>> basis = [x, y, z]
>>> ax = derive_by_array([exp(x), sin(y*z), t], basis)
>>> ax
[[exp(x), 0, 0], [0, z*cos(y*z), 0], [0, y*cos(y*z), 0]]
```

Contraction of the resulting array: $\sum_m \frac{\partial A^m}{\partial x^m}$

```
>>> tensorcontraction(ax, (0, 1))
z*cos(y*z) + exp(x)
```

Classes

class sympy.tensor.array.ImmutableDenseNDimArray(*iterable, shape=None, **kwargs*)

class sympy.tensor.array.ImmutableSparseNDimArray(*iterable=None, shape=None, **kwargs*)

class sympy.tensor.array.MutableDenseNDimArray(*iterable=None, shape=None, **kwargs*)

class sympy.tensor.array.MutableSparseNDimArray(*iterable=None, shape=None, **kwargs*)

Functions

sympy.tensor.array.derive_by_array(*expr, dx*)

Derivative by arrays. Supports both arrays and scalars.

Explanation

Given the array $A_{i_1, \dots, i_N}$ and the array $X_{j_1, \dots, j_M}$ this function will return a new array B defined by

$$B_{j_1, \dots, j_M, i_1, \dots, i_N} := \frac{\partial A_{i_1, \dots, i_N}}{\partial X_{j_1, \dots, j_M}}$$

Examples

```
>>> from sympy import derive_by_array
>>> from sympy.abc import x, y, z, t
>>> from sympy import cos
>>> derive_by_array(cos(x*t), x)
-t*sin(t*x)
>>> derive_by_array(cos(x*t), [x, y, z, t])
[-t*sin(t*x), 0, 0, -x*sin(t*x)]
>>> derive_by_array([x, y**2*z], [[x, y], [z, t]])
[[[1, 0], [0, 2*y*z]], [[0, y**2], [0, 0]]]
```

sympy.tensor.array.permutedims(*expr, perm=None, index_order_old=None, index_order_new=None*)

Permutes the indices of an array.

Parameter specifies the permutation of the indices.

Examples

```
>>> from sympy.abc import x, y, z, t
>>> from sympy import sin
>>> from sympy import Array, permutedims
>>> a = Array([[x, y, z], [t, sin(x), 0]])
>>> a
[[x, y, z], [t, sin(x), 0]]
>>> permutedims(a, (1, 0))
[[x, t], [y, sin(x)], [z, 0]]
```

If the array is of second order, transpose can be used:

```
>>> from sympy import transpose
>>> transpose(a)
[[x, t], [y, sin(x)], [z, 0]]
```

Examples on higher dimensions:

```
>>> b = Array([[[[1, 2], [3, 4]], [[5, 6], [7, 8]]]])
>>> permutedims(b, (2, 1, 0))
[[[1, 5], [3, 7]], [[2, 6], [4, 8]]]
>>> permutedims(b, (1, 2, 0))
[[[1, 5], [2, 6]], [[3, 7], [4, 8]]]
```

An alternative way to specify the same permutations as in the previous lines involves passing the *old* and *new* indices, either as a list or as a string:

```
>>> permutedims(b, index_order_old="cba", index_order_new="abc")
[[[1, 5], [3, 7]], [[2, 6], [4, 8]]]
>>> permutedims(b, index_order_old="cab", index_order_new="abc")
[[[1, 5], [2, 6]], [[3, 7], [4, 8]]]
```

Permutation objects are also allowed:

```
>>> from sympy.combinatorics import Permutation
>>> permutedims(b, Permutation([1, 2, 0]))
[[[1, 5], [2, 6]], [[3, 7], [4, 8]]]
```

`sympy.tensor.array.tensorcontraction(array, *contraction_axes)`

Contraction of an array-like object on the specified axes.

Examples

```
>>> from sympy import Array, tensorcontraction
>>> from sympy import Matrix, eye
>>> tensorcontraction(eye(3), (0, 1))
3
>>> A = Array(range(18), (3, 2, 3))
>>> A
[[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]], [[12, 13, 14], [15,
→ 16, 17]]]
```

(continues on next page)

(continued from previous page)

```
>>> tensorcontraction(A, (0, 2))
[21, 30]
```

Matrix multiplication may be emulated with a proper combination of `tensorcontraction` and `tensorproduct`

```
>>> from sympy import tensorproduct
>>> from sympy.abc import a,b,c,d,e,f,g,h
>>> m1 = Matrix([[a, b], [c, d]])
>>> m2 = Matrix([[e, f], [g, h]])
>>> p = tensorproduct(m1, m2)
>>> p
[[[a*e, a*f], [a*g, a*h]], [[b*e, b*f], [b*g, b*h]]], [[c*e, c*f],
→ [c*g, c*h]], [[d*e, d*f], [d*g, d*h]]]]
>>> tensorcontraction(p, (1, 2))
[[a*e + b*g, a*f + b*h], [c*e + d*g, c*f + d*h]]
>>> m1*m2
Matrix([
[a*e + b*g, a*f + b*h],
[c*e + d*g, c*f + d*h]])
```

`sympy.tensor.array.tensorproduct(*args)`

Tensor product among scalars or array-like objects.

Examples

```
>>> from sympy.tensor.array import tensorproduct, Array
>>> from sympy.abc import x, y, z, t
>>> A = Array([[1, 2], [3, 4]])
>>> B = Array([x, y])
>>> tensorproduct(A, B)
[[[x, y], [2*x, 2*y]], [[3*x, 3*y], [4*x, 4*y]]]
>>> tensorproduct(A, x)
[[x, 2*x], [3*x, 4*x]]
>>> tensorproduct(A, B, B)
[[[x**2, x*y], [x*y, y**2]], [[2*x**2, 2*x*y], [2*x*y, 2*y**2]]],
→ [[3*x**2, 3*x*y], [3*x*y, 3*y**2]], [[4*x**2, 4*x*y], [4*x*y,
→ 4*y**2]]]]
```

Applying this function on two matrices will result in a rank 4 array.

```
>>> from sympy import Matrix, eye
>>> m = Matrix([[x, y], [z, t]])
>>> p = tensorproduct(eye(3), m)
>>> p
[[[x, y], [z, t]], [[0, 0], [0, 0]], [[0, 0], [0, 0]]], [[0, 0], [0,
→ 0]], [[x, y], [z, t]], [[0, 0], [0, 0]], [[0, 0], [0, 0]]], [[0, 0], [0,
→ 0]], [[x, y], [z, t]], [[0, 0], [0, 0]], [[0, 0], [0, 0]]], [[0, 0], [0,
→ 0]]]]
```

`sympy.tensor.array.tensordiagonal(array, *diagonal_axes)`

Diagonalization of an array-like object on the specified axes.

This is equivalent to multiplying the expression by Kronecker deltas uniting the axes. The diagonal indices are put at the end of the axes.

Examples

`tensorialagonal` acting on a 2-dimensional array by axes 0 and 1 is equivalent to the diagonal of the matrix:

```
>>> from sympy import Array, tensorialagonal
>>> from sympy import Matrix, eye
>>> tensorialagonal(eye(3), (0, 1))
[1, 1, 1]
```

```
>>> from sympy.abc import a,b,c,d
>>> m1 = Matrix([[a, b], [c, d]])
>>> tensorialagonal(m1, [0, 1])
[a, d]
```

In case of higher dimensional arrays, the diagonalized out dimensions are appended removed and appended as a single dimension at the end:

```
>>> A = Array(range(18), (3, 2, 3))
>>> A
[[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]], [[12, 13, 14], [15,
→16, 17]]]
>>> tensorialagonal(A, (0, 2))
[[0, 7, 14], [3, 10, 17]]
>>> from sympy import permutedims
>>> tensorialagonal(A, (0, 2)) == permutedims(Array([A[0, :, 0], A[1, :,
→1], A[2, :, 2]]), [1, 0])
True
```

N-dim array expressions

Array expressions are expressions representing N-dimensional arrays, without evaluating them. These expressions represent in a certain way abstract syntax trees of operations on N-dimensional arrays.

Every N-dimensional array operator has a corresponding array expression object.

Table of correspondences:

Array operator	Array expression operator
<code>tensorproduct</code>	<code>ArrayTensorProduct</code>
<code>tensorcontraction</code>	<code>ArrayContraction</code>
<code>tensorialagonal</code>	<code>ArrayDiagonal</code>
<code>permutedims</code>	<code>PermuteDims</code>

Examples

ArraySymbol objects are the N-dimensional equivalent of MatrixSymbol objects in the matrix module:

```
>>> from sympy.tensor.array.expressions import ArraySymbol
>>> from sympy.abc import i, j, k
>>> A = ArraySymbol("A", (3, 2, 4))
>>> A.shape
(3, 2, 4)
>>> A[i, j, k]
A[i, j, k]
>>> A.as_explicit()
[[[A[0, 0, 0], A[0, 0, 1], A[0, 0, 2], A[0, 0, 3]],
  [A[0, 1, 0], A[0, 1, 1], A[0, 1, 2], A[0, 1, 3]]],
 [[A[1, 0, 0], A[1, 0, 1], A[1, 0, 2], A[1, 0, 3]],
  [A[1, 1, 0], A[1, 1, 1], A[1, 1, 2], A[1, 1, 3]]],
 [[A[2, 0, 0], A[2, 0, 1], A[2, 0, 2], A[2, 0, 3]],
  [A[2, 1, 0], A[2, 1, 1], A[2, 1, 2], A[2, 1, 3]]]]
```

Component-explicit arrays can be added inside array expressions:

```
>>> from sympy import Array
>>> from sympy import tensorproduct
>>> from sympy.tensor.array.expressions import ArrayTensorProduct
>>> a = Array([1, 2, 3])
>>> b = Array([i, j, k])
>>> expr = ArrayTensorProduct(a, b, b)
>>> expr
ArrayTensorProduct([1, 2, 3], [i, j, k], [i, j, k])
>>> expr.as_explicit() == tensorproduct(a, b, b)
True
```

Constructing array expressions from index-explicit forms

Array expressions are index-implicit. This means they do not use any indices to represent array operations. The function `convert_indexed_to_array(...)` may be used to convert index-explicit expressions to array expressions. It takes as input two parameters: the index-explicit expression and the order of the indices:

```
>>> from sympy.tensor.array.expressions import convert_indexed_to_array
>>> from sympy import Sum
>>> A = ArraySymbol("A", (3, 3))
>>> B = ArraySymbol("B", (3, 3))
>>> convert_indexed_to_array(A[i, j], [i, j])
A
>>> convert_indexed_to_array(A[i, j], [j, i])
PermuteDims(A, (0 1))
>>> convert_indexed_to_array(A[i, j] + B[j, i], [i, j])
ArrayAdd(A, PermuteDims(B, (0 1)))
>>> convert_indexed_to_array(Sum(A[i, j]*B[j, k], (j, 0, 2)), [i, k])
ArrayContraction(ArrayTensorProduct(A, B), (1, 2))
```

The diagonal of a matrix in the array expression form:

```
>>> convert_indexed_to_array(A[i, i], [i])
ArrayDiagonal(A, (0, 1))
```

The trace of a matrix in the array expression form:

```
>>> convert_indexed_to_array(Sum(A[i, i], (i, 0, 2)), [i])
ArrayContraction(A, (0, 1))
```

Compatibility with matrices

Array expressions can be mixed with objects from the matrix module:

```
>>> from sympy import MatrixSymbol
>>> from sympy.tensor.array.expressions import ArrayContraction
>>> M = MatrixSymbol("M", 3, 3)
>>> N = MatrixSymbol("N", 3, 3)
```

Express the matrix product in the array expression form:

```
>>> from sympy.tensor.array.expressions import convert_matrix_to_array
>>> expr = convert_matrix_to_array(M*N)
>>> expr
ArrayContraction(ArrayTensorProduct(M, N), (1, 2))
```

The expression can be converted back to matrix form:

```
>>> from sympy.tensor.array.expressions import convert_array_to_matrix
>>> convert_array_to_matrix(expr)
M*N
```

Add a second contraction on the remaining axes in order to get the trace of $M \cdot N$:

```
>>> expr_tr = ArrayContraction(expr, (0, 1))
>>> expr_tr
ArrayContraction(ArrayContraction(ArrayTensorProduct(M, N), (1, 2)), (0, 1))
```

Flatten the expression by calling `.doit()` and remove the nested array contraction operations:

```
>>> expr_tr.doit()
ArrayContraction(ArrayTensorProduct(M, N), (0, 3), (1, 2))
```

Get the explicit form of the array expression:

```
>>> expr.as_explicit()
[[M[0, 0]*N[0, 0] + M[0, 1]*N[1, 0] + M[0, 2]*N[2, 0], M[0, 0]*N[0, 1] + M[0, 1]*N[1, 1] + M[0, 2]*N[2, 1], M[0, 0]*N[0, 2] + M[0, 1]*N[1, 2] + M[0, 2]*N[2, 2]],
 [M[1, 0]*N[0, 0] + M[1, 1]*N[1, 0] + M[1, 2]*N[2, 0], M[1, 0]*N[0, 1] + M[1, 1]*N[1, 1] + M[1, 2]*N[2, 1], M[1, 0]*N[0, 2] + M[1, 1]*N[1, 2] + M[1, 2]*N[2, 2]],
 [M[2, 0]*N[0, 0] + M[2, 1]*N[1, 0] + M[2, 2]*N[2, 0], M[2, 0]*N[0, 1] + M[2, 1]*N[1, 1] + M[2, 2]*N[2, 1], M[2, 0]*N[0, 2] + M[2, 1]*N[1, 2] + M[2, 2]*N[2, 2]]]
```

(continues on next page)

(continued from previous page)

```
[M[2, 0]*N[0, 0] + M[2, 1]*N[1, 0] + M[2, 2]*N[2, 0], M[2, 0]*N[0, 1] + M[2, 1]*N[1, 1] + M[2, 2]*N[2, 1], M[2, 0]*N[0, 2] + M[2, 1]*N[1, 2] + M[2, 2]*N[2, 2]]]
```

Express the trace of a matrix:

```
>>> from sympy import Trace
>>> convert_matrix_to_array(Trace(M))
ArrayContraction(M, (0, 1))
>>> convert_matrix_to_array(Trace(M*N))
ArrayContraction(ArrayTensorProduct(M, N), (0, 3), (1, 2))
```

Express the transposition of a matrix (will be expressed as a permutation of the axes:

```
>>> convert_matrix_to_array(M.T)
PermuteDims(M, (0 1))
```

Compute the derivative array expressions:

```
>>> from sympy.tensor.array.expressions import array_derive
>>> d = array_derive(M, M)
>>> d
PermuteDims(ArrayTensorProduct(I, I), (3)(1 2))
```

Verify that the derivative corresponds to the form computed with explicit matrices:

```
>>> d.as_explicit()
[[[1, 0, 0], [0, 0, 0], [0, 0, 0]], [[0, 1, 0], [0, 0, 0], [0, 0, 0]], [[0, 0, 1], [0, 0, 0], [0, 0, 0]], [[0, 0, 0], [1, 0, 0], [0, 0, 0]], [[0, 0, 0], [0, 1, 0], [0, 0, 0]], [[0, 0, 0], [0, 0, 1], [0, 0, 0]], [[0, 0, 0], [0, 0, 0], [1, 0, 0]], [[0, 0, 0], [0, 0, 0], [0, 1, 0]], [[0, 0, 0], [0, 0, 0], [0, 0, 1]]]
>>> Me = M.as_explicit()
>>> Me.diff(Me)
[[[1, 0, 0], [0, 0, 0], [0, 0, 0]], [[0, 1, 0], [0, 0, 0], [0, 0, 0]], [[0, 0, 1], [0, 0, 0], [0, 0, 0]], [[0, 0, 0], [1, 0, 0], [0, 0, 0]], [[0, 0, 0], [0, 1, 0], [0, 0, 0]], [[0, 0, 0], [0, 0, 1], [0, 0, 0]], [[0, 0, 0], [0, 0, 0], [1, 0, 0]], [[0, 0, 0], [0, 0, 0], [0, 1, 0]], [[0, 0, 0], [0, 0, 0], [0, 0, 1]]]
```

Indexed Objects

Module that defines indexed objects

The classes `IndexedBase`, `Indexed`, and `Idx` represent a matrix element $M[i, j]$ as in the following diagram:

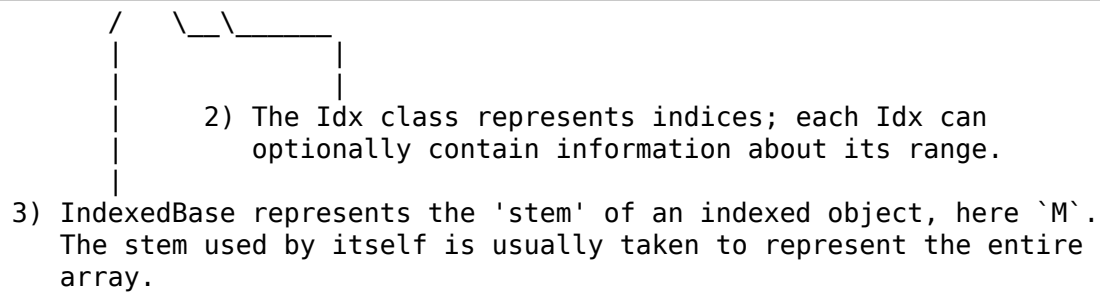
1) The `Indexed` class represents the entire indexed object.

```

      |
  _____
      |_____
      |
      M[i, j]
```

(continues on next page)

(continued from previous page)

- 
- 2) The Idx class represents indices; each Idx can optionally contain information about its range.
 - 3) IndexedBase represents the 'stem' of an indexed object, here `M`. The stem used by itself is usually taken to represent the entire array.

There can be any number of indices on an Indexed object. No transformation properties are implemented in these Base objects, but implicit contraction of repeated indices is supported.

Note that the support for complicated (i.e. non-atomic) integer expressions as indices is limited. (This should be improved in future releases.)

Examples

To express the above matrix element example you would write:

```
>>> from sympy import symbols, IndexedBase, Idx
>>> M = IndexedBase('M')
>>> i, j = symbols('i j', cls=Idx)
>>> M[i, j]
M[i, j]
```

Repeated indices in a product implies a summation, so to express a matrix-vector product in terms of Indexed objects:

```
>>> x = IndexedBase('x')
>>> M[i, j]*x[j]
M[i, j]*x[j]
```

If the indexed objects will be converted to component based arrays, e.g. with the code printers or the autowrap framework, you also need to provide (symbolic or numerical) dimensions. This can be done by passing an optional shape parameter to IndexedBase upon construction:

```
>>> dim1, dim2 = symbols('dim1 dim2', integer=True)
>>> A = IndexedBase('A', shape=(dim1, 2*dim1, dim2))
>>> A.shape
(dim1, 2*dim1, dim2)
>>> A[i, j, 3].shape
(dim1, 2*dim1, dim2)
```

If an IndexedBase object has no shape information, it is assumed that the array is as large as the ranges of its indices:

```
>>> n, m = symbols('n m', integer=True)
>>> i = Idx('i', m)
>>> j = Idx('j', n)
>>> M[i, j].shape
(m, n)
```

(continues on next page)

(continued from previous page)

```
>>> M[i, j].ranges
[(0, m - 1), (0, n - 1)]
```

The above can be compared with the following:

```
>>> A[i, 2, j].shape
(dim1, 2*dim1, dim2)
>>> A[i, 2, j].ranges
[(0, m - 1), None, (0, n - 1)]
```

To analyze the structure of indexed expressions, you can use the methods `get_indices()` and `get_contraction_structure()`:

```
>>> from sympy.tensor import get_indices, get_contraction_structure
>>> get_indices(A[i, j, j])
({i}, {})
>>> get_contraction_structure(A[i, j, j])
{(j,): {A[i, j, j]}}
```

See the appropriate docstrings for a detailed explanation of the output.

class `sympy.tensor.indexed.Idx(label, range=None, **kw_args)`

Represents an integer index as an Integer or integer expression.

There are a number of ways to create an `Idx` object. The constructor takes two arguments:

label

An integer or a symbol that labels the index.

range

Optionally you can specify a range as either

- Symbol or integer: This is interpreted as a dimension. Lower and upper bounds are set to 0 and `range - 1`, respectively.
- tuple: The two elements are interpreted as the lower and upper bounds of the range, respectively.

Note: bounds of the range are assumed to be either integer or infinite (`oo` and `-oo` are allowed to specify an unbounded range). If `n` is given as a bound, then `n.is_integer` must not return false.

For convenience, if the label is given as a string it is automatically converted to an integer symbol. (Note: this conversion is not done for range or dimension arguments.)

Examples

```
>>> from sympy import Idx, symbols, oo
>>> n, i, L, U = symbols('n i L U', integer=True)
```

If a string is given for the label an integer Symbol is created and the bounds are both None:


```
>>> idx = Idx('qwerty'); idx
qwerty
>>> idx.lower, idx.upper
(None, None)
```

Both upper and lower bounds can be specified:

```
>>> idx = Idx(i, (L, U)); idx
i
>>> idx.lower, idx.upper
(L, U)
```

When only a single bound is given it is interpreted as the dimension and the lower bound defaults to 0:

```
>>> idx = Idx(i, n); idx.lower, idx.upper
(0, n - 1)
>>> idx = Idx(i, 4); idx.lower, idx.upper
(0, 3)
>>> idx = Idx(i, oo); idx.lower, idx.upper
(0, oo)
```

property label

Returns the label (Integer or integer expression) of the Idx object.

Examples

```
>>> from sympy import Idx, Symbol
>>> x = Symbol('x', integer=True)
>>> Idx(x).label
x
>>> j = Symbol('j', integer=True)
>>> Idx(j).label
j
>>> Idx(j + 1).label
j + 1
```

property lower

Returns the lower bound of the Idx.

Examples

```
>>> from sympy import Idx
>>> Idx('j', 2).lower
0
>>> Idx('j', 5).lower
0
>>> Idx('j').lower is None
True
```

property upper

Returns the upper bound of the Idx.

Examples

```
>>> from sympy import Idx
>>> Idx('j', 2).upper
1
>>> Idx('j', 5).upper
4
>>> Idx('j').upper is None
True
```

class sympy.tensor.indexed.Indexed(*base, *args, **kw_args*)

Represents a mathematical object with indices.

```
>>> from sympy import Indexed, IndexedBase, Idx, symbols
>>> i, j = symbols('i j', cls=Idx)
>>> Indexed('A', i, j)
A[i, j]
```

It is recommended that Indexed objects be created by indexing IndexedBase: IndexedBase('A')[i, j] instead of Indexed(IndexedBase('A'), i, j).

```
>>> A = IndexedBase('A')
>>> a_ij = A[i, j]           # Prefer this,
>>> b_ij = Indexed(A, i, j)  # over this.
>>> a_ij == b_ij
True
```

property base

Returns the IndexedBase of the Indexed object.

Examples

```
>>> from sympy import Indexed, IndexedBase, Idx, symbols
>>> i, j = symbols('i j', cls=Idx)
>>> Indexed('A', i, j).base
A
>>> B = IndexedBase('B')
>>> B == B[i, j].base
True
```

property indices

Returns the indices of the Indexed object.

Examples

```
>>> from sympy import Indexed, Idx, symbols
>>> i, j = symbols('i j', cls=Idx)
>>> Indexed('A', i, j).indices
(i, j)
```

property ranges

Returns a list of tuples with lower and upper range of each index.

If an index does not define the data members upper and lower, the corresponding slot in the list contains None instead of a tuple.

Examples

```
>>> from sympy import Indexed, Idx, symbols
>>> Indexed('A', Idx('i', 2), Idx('j', 4), Idx('k', 8)).ranges
[(0, 1), (0, 3), (0, 7)]
>>> Indexed('A', Idx('i', 3), Idx('j', 3), Idx('k', 3)).ranges
[(0, 2), (0, 2), (0, 2)]
>>> x, y, z = symbols('x y z', integer=True)
>>> Indexed('A', x, y, z).ranges
[None, None, None]
```

property rank

Returns the rank of the Indexed object.

Examples

```
>>> from sympy import Indexed, Idx, symbols
>>> i, j, k, l, m = symbols('i:m', cls=Idx)
>>> Indexed('A', i, j).rank
2
>>> q = Indexed('A', i, j, k, l, m)
>>> q.rank
5
>>> q.rank == len(q.indices)
True
```

property shape

Returns a list with dimensions of each index.

Dimensions is a property of the array, not of the indices. Still, if the IndexedBase does not define a shape attribute, it is assumed that the ranges of the indices correspond to the shape of the array.

```
>>> from sympy import IndexedBase, Idx, symbols
>>> n, m = symbols('n m', integer=True)
>>> i = Idx('i', m)
>>> j = Idx('j', m)
```

(continues on next page)

(continued from previous page)

```
>>> A = IndexedBase('A', shape=(n, n))
>>> B = IndexedBase('B')
>>> A[i, j].shape
(n, n)
>>> B[i, j].shape
(m, m)
```

```
class sympy.tensor.indexed.IndexedBase(label, shape=None, *, offset=0,
                                       strides=None, **kw_args)
```

Represent the base or stem of an indexed object

The IndexedBase class represent an array that contains elements. The main purpose of this class is to allow the convenient creation of objects of the Indexed class. The `__getitem__` method of IndexedBase returns an instance of Indexed. Alone, without indices, the IndexedBase class can be used as a notation for e.g. matrix equations, resembling what you could do with the Symbol class. But, the IndexedBase class adds functionality that is not available for Symbol instances:

- An IndexedBase object can optionally store shape information. This can be used in to check array conformance and conditions for numpy broadcasting. (TODO)
- An IndexedBase object implements syntactic sugar that allows easy symbolic representation of array operations, using implicit summation of repeated indices.
- The IndexedBase object symbolizes a mathematical structure equivalent to arrays, and is recognized as such for code generation and automatic compilation and wrapping.

```
>>> from sympy.tensor import IndexedBase, Idx
>>> from sympy import symbols
>>> A = IndexedBase('A'); A
A
>>> type(A)
<class 'sympy.tensor.indexed.IndexedBase'>
```

When an IndexedBase object receives indices, it returns an array with named axes, represented by an Indexed object:

```
>>> i, j = symbols('i j', integer=True)
>>> A[i, j, 2]
A[i, j, 2]
>>> type(A[i, j, 2])
<class 'sympy.tensor.indexed.Indexed'>
```

The IndexedBase constructor takes an optional shape argument. If given, it overrides any shape information in the indices. (But not the index ranges!)

```
>>> m, n, o, p = symbols('m n o p', integer=True)
>>> i = Idx('i', m)
>>> j = Idx('j', n)
>>> A[i, j].shape
(m, n)
>>> B = IndexedBase('B', shape=(o, p))
>>> B[i, j].shape
(o, p)
```

Assumptions can be specified with keyword arguments the same way as for Symbol:

```
>>> A_real = IndexedBase('A', real=True)
>>> A_real.is_real
True
>>> A != A_real
True
```

Assumptions can also be inherited if a Symbol is used to initialize the IndexedBase:

```
>>> I = symbols('I', integer=True)
>>> C_inherit = IndexedBase(I)
>>> C_explicit = IndexedBase('I', integer=True)
>>> C_inherit == C_explicit
True
```

property label

Returns the label of the IndexedBase object.

Examples

```
>>> from sympy import IndexedBase
>>> from sympy.abc import x, y
>>> IndexedBase('A', shape=(x, y)).label
A
```

property offset

Returns the offset for the IndexedBase object.

This is the value added to the resulting index when the 2D Indexed object is unrolled to a 1D form. Used in code generation.

Examples

```
>>> from sympy.printing import ccode
>>> from sympy.tensor import IndexedBase, Idx
>>> from sympy import symbols
>>> l, m, n, o = symbols('l m n o', integer=True)
>>> A = IndexedBase('A', strides=(l, m, n), offset=o)
>>> i, j, k = map(Idx, 'ijk')
>>> ccode(A[i, j, k])
'A[l*i + m*j + n*k + o]'
```

property shape

Returns the shape of the IndexedBase object.

Examples

```
>>> from sympy import IndexedBase, Idx
>>> from sympy.abc import x, y
>>> IndexedBase('A', shape=(x, y)).shape
(x, y)
```

Note: If the shape of the IndexedBase is specified, it will override any shape information given by the indices.

```
>>> A = IndexedBase('A', shape=(x, y))
>>> B = IndexedBase('B')
>>> i = Idx('i', 2)
>>> j = Idx('j', 1)
>>> A[i, j].shape
(x, y)
>>> B[i, j].shape
(2, 1)
```

property strides

Returns the strided scheme for the IndexedBase object.

Normally this is a tuple denoting the number of steps to take in the respective dimension when traversing an array. For code generation purposes `strides='C'` and `strides='F'` can also be used.

`strides='C'` would mean that code printer would unroll in row-major order and `'F'` means unroll in column major order.

Methods

Module with functions operating on IndexedBase, Indexed and Idx objects

- Check shape conformance
- Determine indices in resulting expression

etc.

Methods in this module could be implemented by calling methods on Expr objects instead. When things stabilize this could be a useful refactoring.

`sympy.tensor.index_methods.get_contraction_structure(expr)`

Determine dummy indices of `expr` and describe its structure

By *dummy* we mean indices that are summation indices.

The structure of the expression is determined and described as follows:

- 1) A conforming summation of Indexed objects is described with a dict where the keys are summation indices and the corresponding values are sets containing all terms for which the summation applies. All Add objects in the SymPy expression tree are described like this.
- 2) For all nodes in the SymPy expression tree that are *not* of type Add, the following applies:

If a node discovers contractions in one of its arguments, the node itself will be stored as a key in the dict. For that key, the corresponding value is a list of dicts, each of which is the result of a recursive call to `get_contraction_structure()`. The list contains only dicts for the non-trivial deeper contractions, omitting dicts with `None` as the one and only key.

Note: The presence of expressions among the dictionary keys indicates multiple levels of index contractions. A nested dict displays nested contractions and may itself contain dicts from a deeper level. In practical calculations the summation in the deepest nested level must be calculated first so that the outer expression can access the resulting indexed object.

Examples

```
>>> from sympy.tensor.index_methods import get_contraction_structure
>>> from sympy import default_sort_key
>>> from sympy.tensor import IndexedBase, Idx
>>> x, y, A = map(IndexedBase, ['x', 'y', 'A'])
>>> i, j, k, l = map(Idx, ['i', 'j', 'k', 'l'])
>>> get_contraction_structure(x[i]*y[i] + A[j, j])
{(i,): {x[i]*y[i]}, (j,): {A[j, j]}}
>>> get_contraction_structure(x[i]*y[j])
{None: {x[i]*y[j]}}
```

A multiplication of contracted factors results in nested dicts representing the internal contractions.

```
>>> d = get_contraction_structure(x[i, i]*y[j, j])
>>> sorted(d.keys(), key=default_sort_key)
[None, x[i, i]*y[j, j]]
```

In this case, the product has no contractions:

```
>>> d[None]
{x[i, i]*y[j, j]}
```

Factors are contracted “first”:

```
>>> sorted(d[x[i, i]*y[j, j]], key=default_sort_key)
[{(i,): {x[i, i]}}, {(j,): {y[j, j]}}]
```

A parenthesized Add object is also returned as a nested dictionary. The term containing the parenthesis is a Mul with a contraction among the arguments, so it will be found as a key in the result. It stores the dictionary resulting from a recursive call on the Add expression.

```
>>> d = get_contraction_structure(x[i]*(y[i] + A[i, j]*x[j]))
>>> sorted(d.keys(), key=default_sort_key)
[(A[i, j]*x[j] + y[i])*x[i], (i,)]
>>> d[(i,)]
{(A[i, j]*x[j] + y[i])*x[i]}
```

(continues on next page)

(continued from previous page)

```
>>> d[x[i]*(A[i, j]*x[j] + y[i])]
[{None: {y[i]}, (j,): {A[i, j]*x[j]}}]
```

Powers with contractions in either base or exponent will also be found as keys in the dictionary, mapping to a list of results from recursive calls:

```
>>> d = get_contraction_structure(A[j, j]**A[i, i])
>>> d[None]
{A[j, j]**A[i, i]}
>>> nested_contractions = d[A[j, j]**A[i, i]]
>>> nested_contractions[0]
{(j,): {A[j, j]}}
>>> nested_contractions[1]
{(i,): {A[i, i]}}
```

The description of the contraction structure may appear complicated when represented with a string in the above examples, but it is easy to iterate over:

```
>>> from sympy import Expr
>>> for key in d:
...     if isinstance(key, Expr):
...         continue
...     for term in d[key]:
...         if term in d:
...             # treat deepest contraction first
...             pass
...         # treat outermost contractions here
```

`sympy.tensor.index_methods.get_indices(expr)`

Determine the outer indices of expression `expr`

By *outer* we mean indices that are not summation indices. Returns a set and a dict. The set contains outer indices and the dict contains information about index symmetries.

Examples

```
>>> from sympy.tensor.index_methods import get_indices
>>> from sympy import symbols
>>> from sympy.tensor import IndexedBase
>>> x, y, A = map(IndexedBase, ['x', 'y', 'A'])
>>> i, j, a, z = symbols('i j a z', integer=True)
```

The indices of the total expression is determined, Repeated indices imply a summation, for instance the trace of a matrix A:

```
>>> get_indices(A[i, i])
(set(), {})
```

In the case of many terms, the terms are required to have identical outer indices. Else an `IndexConformanceException` is raised.


```
>>> get_indices(x[i] + A[i, j]*y[j])
({i}, {})
```

Exceptions

An `IndexConformanceException` means that the terms are not compatible, e.g.

```
>>> get_indices(x[i] + y[j])
(...)
IndexConformanceException: Indices are not consistent: x(i) + y(j)
```

Warning: The concept of *outer* indices applies recursively, starting on the deepest level. This implies that dummies inside parenthesis are assumed to be summed first, so that the following expression is handled gracefully:

```
>>> get_indices((x[i] + A[i, j]*y[j])*x[j])
({i, j}, {})
```

This is correct and may appear convenient, but you need to be careful with this as SymPy will happily `.expand()` the product, if requested. The resulting expression would mix the outer `j` with the dummies inside the parenthesis, which makes it a different expression. To be on the safe side, it is best to avoid such ambiguities by using unique indices for all contractions that should be held separate.

Tensor

```
class sympy.tensor.tensor.TensorIndexType(name, dummy_name=None, dim=None,
                                           eps_dim=None, metric_symmetry=1,
                                           metric_name='metric', **kwargs)
```

A `TensorIndexType` is characterized by its name and its metric.

Parameters

- name** : name of the tensor type
- dummy_name** : name of the head of dummy indices
- dim** : dimension, it can be a symbol or an integer or `None`
- eps_dim** : dimension of the epsilon tensor
- metric_symmetry** : integer that denotes metric symmetry or `None` for no metric
- metric_name** : string with the name of the metric tensor

Notes

The possible values of the `metric_symmetry` parameter are:

1 : metric tensor is fully symmetric 0 : metric tensor possesses no index symmetry -1 : metric tensor is fully antisymmetric None: there is no metric tensor (metric equals to None)

The metric is assumed to be symmetric by default. It can also be set to a custom tensor by the `.set_metric()` method.

If there is a metric the metric is used to raise and lower indices.

In the case of non-symmetric metric, the following raising and lowering conventions will be adopted:

$$\text{psi}(a) = g(a, b) * \text{psi}(-b); \text{chi}(-a) = \text{chi}(b) * g(-b, -a)$$

From these it is easy to find:

$$g(-a, b) = \text{delta}(-a, b)$$

where $\text{delta}(-a, b) = \text{delta}(b, -a)$ is the Kronecker delta (see `TensorIndex` for the conventions on indices). For antisymmetric metrics there is also the following equality:

$$g(a, -b) = -\text{delta}(a, -b)$$

If there is no metric it is not possible to raise or lower indices; e.g. the index of the defining representation of $SU(N)$ is 'covariant' and the conjugate representation is 'contravariant'; for $N > 2$ they are linearly independent.

`eps_dim` is by default equal to `dim`, if the latter is an integer; else it can be assigned (for use in naive dimensional regularization); if `eps_dim` is not an integer `epsilon` is `None`.

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> Lorentz.metric
metric(Lorentz, Lorentz)
```

Attributes

<code>metric</code>	(the metric tensor)
<code>delta</code>	(Kronecker delta)
<code>epsilon</code>	(the Levi-Civita epsilon tensor)
<code>data</code>	((deprecated) a property to add ndarray values, to work in a specified basis.)

class `sympy.tensor.tensor.TensorIndex`(*name*, *tensor_index_type*, *is_up=True*)

Represents a tensor index

Parameters

name : name of the index, or `True` if you want it to be automatically assigned

tensor_index_type : `TensorIndexType` of the index

is_up : flag for contravariant index (is_up=True by default)

Notes

Tensor indices are contracted with the Einstein summation convention.

An index can be in contravariant or in covariant form; in the latter case it is represented prepending a - to the index name. Adding - to a covariant (is_up=False) index makes it contravariant.

Dummy indices have a name with head given by `tensor_index_type.dummy_name` with underscore and a number.

Similar to symbols multiple contravariant indices can be created at once using `tensor_indices(s, typ)`, where `s` is a string of names.

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, TensorIndex, TensorHead, tensor_indices
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> mu = TensorIndex('mu', Lorentz, is_up=False)
>>> nu, rho = tensor_indices('nu, rho', Lorentz)
>>> A = TensorHead('A', [Lorentz, Lorentz])
>>> A(mu, nu)
A(-mu, nu)
>>> A(-mu, -rho)
A(mu, -rho)
>>> A(mu, -mu)
A(-L_0, L_0)
```

Attributes

name	
tensor_index_type	
is_up	

class `sympy.tensor.tensor.TensorHead(name, index_types, symmetry=None, comm=0)`
Tensor head of the tensor.

Parameters

- name** : name of the tensor
- index_types** : list of `TensorIndexType`
- symmetry** : `TensorSymmetry` of the tensor
- comm** : commutation group number

Notes

Similar to symbols multiple TensorHeads can be created using `tensorhead(s, typ, sym=None, comm=0)` function, where `s` is the string of names and `sym` is the monotermin tensor symmetry (see `tensorsymmetry`).

A TensorHead belongs to a commutation group, defined by a symbol on number `comm` (see `_TensorManager.set_comm`); tensors in a commutation group have the same commutation properties; by default `comm` is 0, the group of the commuting tensors.

Examples

Define a fully antisymmetric tensor of rank 2:

```
>>> from sympy.tensor.tensor import TensorIndexType, TensorHead, _
_TensorSymmetry
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> asym2 = TensorSymmetry.fully_symmetric(-2)
>>> A = TensorHead('A', [Lorentz, Lorentz], asym2)
```

Examples with ndarray values, the components data assigned to the TensorHead object are assumed to be in a fully-contravariant representation. In case it is necessary to assign components data which represents the values of a non-fully covariant tensor, see the other examples.

```
>>> from sympy.tensor.tensor import tensor_indices
>>> from sympy import diag
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> i0, i1 = tensor_indices('i0:2', Lorentz)
```

Specify a replacement dictionary to keep track of the arrays to use for replacements in the tensorial expression. The TensorIndexType is associated to the metric used for contractions (in fully covariant form):

```
>>> repl = {Lorentz: diag(1, -1, -1, -1)}
```

Let's see some examples of working with components with the electromagnetic tensor:

```
>>> from sympy import symbols
>>> Ex, Ey, Ez, Bx, By, Bz = symbols('E_x E_y E_z B_x B_y B_z')
>>> c = symbols('c', positive=True)
```

Let's define F , an antisymmetric tensor:

```
>>> F = TensorHead('F', [Lorentz, Lorentz], asym2)
```

Let's update the dictionary to contain the matrix to use in the replacements:

```
>>> repl.update({F(-i0, -i1): [
... [0, Ex/c, Ey/c, Ez/c],
... [-Ex/c, 0, -Bz, By],
... [-Ey/c, Bz, 0, -Bx],
... [-Ez/c, -By, Bx, 0]]})
```

Now it is possible to retrieve the contravariant form of the Electromagnetic tensor:

```
>>> F(i0, i1).replace_with_arrays(repl, [i0, i1])
[[0, -E_x/c, -E_y/c, -E_z/c], [E_x/c, 0, -B_z, B_y], [E_y/c, B_z, 0, -B_
→x], [E_z/c, -B_y, B_x, 0]]
```

and the mixed contravariant-covariant form:

```
>>> F(i0, -i1).replace_with_arrays(repl, [i0, -i1])
[[0, E_x/c, E_y/c, E_z/c], [E_x/c, 0, B_z, -B_y], [E_y/c, -B_z, 0, B_x],
→[E_z/c, B_y, -B_x, 0]]
```

Energy-momentum of a particle may be represented as:

```
>>> from sympy import symbols
>>> P = TensorHead('P', [Lorentz], TensorSymmetry.no_symmetry(1))
>>> E, px, py, pz = symbols('E p_x p_y p_z', positive=True)
>>> repl.update({P(i0): [E, px, py, pz]})
```

The contravariant and covariant components are, respectively:

```
>>> P(i0).replace_with_arrays(repl, [i0])
[E, p_x, p_y, p_z]
>>> P(-i0).replace_with_arrays(repl, [-i0])
[E, -p_x, -p_y, -p_z]
```

The contraction of a 1-index tensor by itself:

```
>>> expr = P(i0)*P(-i0)
>>> expr.replace_with_arrays(repl, [])
E**2 - p_x**2 - p_y**2 - p_z**2
```

Attributes

name	
index_types	
rank	(total number of indices)
symmetry	
comm	(commutation group)

`commutes_with(other)`

Returns 0 if self and other commute, 1 if they anticommute.

Returns None if self and other neither commute nor anticommute.

`sympy.tensor.tensor.tensor_heads(s, index_types, symmetry=None, comm=0)`

Returns a sequence of TensorHeads from a string *s*

class `sympy.tensor.tensor.TensExpr(*args)`

Abstract base class for tensor expressions

Notes

A tensor expression is an expression formed by tensors; currently the sums of tensors are distributed.

A TensExpr can be a TensAdd or a TensMul.

TensMul objects are formed by products of component tensors, and include a coefficient, which is a SymPy expression.

In the internal representation contracted indices are represented by (ipos1, ipos2, icomp1, icomp2), where icomp1 is the position of the component tensor with contravariant index, ipos1 is the slot which the index occupies in that component tensor.

Contracted indices are therefore nameless in the internal representation.

`get_matrix()`

DEPRECATED: do not use.

Returns ndarray components data as a matrix, if components data are available and ndarray dimension does not exceed 2.

`replace_with_arrays(replacement_dict, indices=None)`

Replace the tensorial expressions with arrays. The final array will correspond to the N-dimensional array with indices arranged according to `indices`.

Parameters

`replacement_dict`

dictionary containing the replacement rules for tensors.

`indices`

the index order with respect to which the array is read. The original index order will be used if no value is passed.

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices
>>> from sympy.tensor.tensor import TensorHead
>>> from sympy import symbols, diag
```

```
>>> L = TensorIndexType("L")
>>> i, j = tensor_indices("i j", L)
>>> A = TensorHead("A", [L])
>>> A(i).replace_with_arrays({A(i): [1, 2]}, [i])
[1, 2]
```

Since 'indices' is optional, we can also call `replace_with_arrays` by this way if no specific index order is needed:

```
>>> A(i).replace_with_arrays({A(i): [1, 2]})
[1, 2]
```

```
>>> expr = A(i)*A(j)
>>> expr.replace_with_arrays({A(i): [1, 2]})
[[1, 2], [2, 4]]
```

For contractions, specify the metric of the `TensorIndexType`, which in this case is `L`, in its covariant form:

```
>>> expr = A(i)*A(-i)
>>> expr.replace_with_arrays({A(i): [1, 2], L: diag(1, -1)})
-3
```

Symmetrization of an array:

```
>>> H = TensorHead("H", [L, L])
>>> a, b, c, d = symbols("a b c d")
>>> expr = H(i, j)/2 + H(j, i)/2
>>> expr.replace_with_arrays({H(i, j): [[a, b], [c, d]]})
[[a, b/2 + c/2], [b/2 + c/2, d]]
```

Anti-symmetrization of an array:

```
>>> expr = H(i, j)/2 - H(j, i)/2
>>> repl = {H(i, j): [[a, b], [c, d]]}
>>> expr.replace_with_arrays(repl)
[[0, b/2 - c/2], [-b/2 + c/2, 0]]
```

The same expression can be read as the transpose by inverting `i` and `j`:

```
>>> expr.replace_with_arrays(repl, [j, i])
[[0, -b/2 + c/2], [b/2 - c/2, 0]]
```

class `sympy.tensor.tensor.TensAdd(*args, **kw_args)`

Sum of tensors.

Parameters

free_args : list of the free indices

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_heads, \
↳ tensor_indices
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> a, b = tensor_indices('a,b', Lorentz)
>>> p, q = tensor_heads('p,q', [Lorentz])
>>> t = p(a) + q(a); t
p(a) + q(a)
```

Examples with components data added to the tensor expression:

```
>>> from sympy import symbols, diag
>>> x, y, z, t = symbols("x y z t")
>>> repl = {}
>>> repl[Lorentz] = diag(1, -1, -1, -1)
>>> repl[p(a)] = [1, 2, 3, 4]
>>> repl[q(a)] = [x, y, z, t]
```

The following are: $2^2 - 3^2 - 2^2 - 7^2 ==> -58$

```
>>> expr = p(a) + q(a)
>>> expr.replace_with_arrays(repl, [a])
[x + 1, y + 2, z + 3, t + 4]
```

Attributes

args	(tuple of addends)
rank	(rank of the tensor)
free_args	(list of the free indices in sorted order)

canon_bp()

Canonicalize using the Butler-Portugal algorithm for canonicalization under monoterm symmetries.

contract_metric(*g*)

Raise or lower indices with the metric *g*.

Parameters

g : metric

contract_all : if True, eliminate all *g* which are contracted

Notes

see the `TensorIndexType` docstring for the contraction conventions

class `sympy.tensor.tensor.TensMul(*args, **kw_args)`

Product of tensors.

Parameters

coeff : SymPy coefficient of the tensor

args

Notes

`args[0]` list of `TensorHead` of the component tensors.

`args[1]` list of (*ind*, *ipos*, *icomp*) where *ind* is a free index, *ipos* is the slot position of *ind* in the *icomp*-th component tensor.

`args[2]` list of tuples representing dummy indices. (*ipos1*, *ipos2*, *icomp1*, *icomp2*) indicates that the contravariant dummy index is the *ipos1*-th slot position in the *icomp1*-th component tensor; the corresponding covariant index is in the *ipos2* slot position in the *icomp2*-th component tensor.

Attributes

<code>components</code>	(list of <code>TensorHead</code> of the component tensors)
<code>types</code>	(list of nonrepeated <code>TensorIndexType</code>)
<code>free</code>	(list of (<code>ind</code> , <code>ipos</code> , <code>icomp</code>), see Notes)
<code>dum</code>	(list of (<code>ipos1</code> , <code>ipos2</code> , <code>icomp1</code> , <code>icomp2</code>), see Notes)
<code>ext_rank</code>	(rank of the tensor counting the dummy indices)
<code>rank</code>	(rank of the tensor)
<code>coeff</code>	(SymPy coefficient of the tensor)
<code>free_args</code>	(list of the free indices in sorted order)
<code>is_canon_bp</code>	(True if the tensor in in canonical form)

`canon_bp()`

Canonicalize using the Butler-Portugal algorithm for canonicalization under monotermin symmetries.

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, ↪
↪ TensorHead, TensorSymmetry
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> m0, m1, m2 = tensor_indices('m0,m1,m2', Lorentz)
>>> A = TensorHead('A', [Lorentz]*2, TensorSymmetry.fully_symmetric(↪
↪ 2))
>>> t = A(m0, -m1)*A(m1, -m0)
>>> t.canon_bp()
-A(L_0, L_1)*A(-L_0, -L_1)
>>> t = A(m0, -m1)*A(m1, -m2)*A(m2, -m0)
>>> t.canon_bp()
0
```

`contract_metric(g)`

Raise or lower indices with the metric `g`.

Parameters

`g` : metric

Notes

See the `TensorIndexType` docstring for the contraction conventions.

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, \
↳ tensor_heads
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> m0, m1, m2 = tensor_indices('m0,m1,m2', Lorentz)
>>> g = Lorentz.metric
>>> p, q = tensor_heads('p,q', [Lorentz])
>>> t = p(m0)*q(m1)*g(-m0, -m1)
>>> t.canon_bp()
metric(L_0, L_1)*p(-L_0)*q(-L_1)
>>> t.contract_metric(g).canon_bp()
p(L_0)*q(-L_0)
```

get_free_indices() → List[[TensorIndex](#) (page 1410)]

Returns the list of free indices of the tensor.

Explanation

The indices are listed in the order in which they appear in the component tensors.

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, \
↳ tensor_heads
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> m0, m1, m2 = tensor_indices('m0,m1,m2', Lorentz)
>>> g = Lorentz.metric
>>> p, q = tensor_heads('p,q', [Lorentz])
>>> t = p(m1)*g(m0,m2)
>>> t.get_free_indices()
[m1, m0, m2]
>>> t2 = p(m1)*g(-m1, m2)
>>> t2.get_free_indices()
[m2]
```

get_indices()

Returns the list of indices of the tensor.

Explanation

The indices are listed in the order in which they appear in the component tensors. The dummy indices are given a name which does not collide with the names of the free indices.

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, \
↳ tensor_heads
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> m0, m1, m2 = tensor_indices('m0,m1,m2', Lorentz)
>>> g = Lorentz.metric
>>> p, q = tensor_heads('p,q', [Lorentz])
>>> t = p(m1)*g(m0,m2)
>>> t.get_indices()
[m1, m0, m2]
>>> t2 = p(m1)*g(-m1, m2)
>>> t2.get_indices()
[L_0, -L_0, m2]
```

perm2tensor(*g*, *is_canon_bp=False*)

Returns the tensor corresponding to the permutation *g*

For further details, see the method in TIDS with the same name.

sorted_components()

Returns a tensor product with sorted components.

split()

Returns a list of tensors, whose product is self.

Explanation

Dummy indices contracted among different tensor components become free indices with the same name as the one used to represent the dummy indices.

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, \
↳ tensor_heads, TensorSymmetry
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> a, b, c, d = tensor_indices('a,b,c,d', Lorentz)
>>> A, B = tensor_heads('A,B', [Lorentz]*2, TensorSymmetry.fully_
↳ symmetric(2))
>>> t = A(a,b)*B(-b,c)
>>> t
A(a, L_0)*B(-L_0, c)
>>> t.split()
[A(a, L_0), B(-L_0, c)]
```

sympy.tensor.tensor.canon_bp(*p*)

Butler-Portugal canonicalization. See `tensor_can.py` from the `combinatorics` module for the details.

sympy.tensor.tensor.riemann_cyclic_replace(*t_r*)

replace Riemann tensor with an equivalent expression

$R(m,n,p,q) \rightarrow \frac{2}{3}R(m,n,p,q) - \frac{1}{3}R(m,q,n,p) + \frac{1}{3}R(m,p,n,q)$

`sympy.tensor.tensor.riemann_cyclic(t2)`

Replace each Riemann tensor with an equivalent expression satisfying the cyclic identity.

This trick is discussed in the reference guide to Cadabra.

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, _
↳ TensorHead, riemann_cyclic, TensorSymmetry
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> i, j, k, l = tensor_indices('i,j,k,l', Lorentz)
>>> R = TensorHead('R', [Lorentz]*4, TensorSymmetry.riemann())
>>> t = R(i,j,k,l)*(R(-i,-j,-k,-l) - 2*R(-i,-k,-j,-l))
>>> riemann_cyclic(t)
0
```

class `sympy.tensor.tensor.TensorSymmetry(*args, **kw_args)`

Monoterm symmetry of a tensor (i.e. any symmetric or anti-symmetric index permutation). For the relevant terminology see `tensor_can.py` section of the combinatorics module.

Parameters

bsgs : tuple (base, sgs) BSGS of the symmetry of the tensor

Notes

A tensor can have an arbitrary monoterm symmetry provided by its BSGS. Multiterm symmetries, like the cyclic symmetry of the Riemann tensor (i.e., Bianchi identity), are not covered. See combinatorics module for information on how to generate BSGS for a general index permutation group. Simple symmetries can be generated using built-in methods.

Examples

Define a symmetric tensor of rank 2

```
>>> from sympy.tensor.tensor import TensorIndexType, TensorSymmetry, get_
↳ symmetric_group_sgs, TensorHead
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> sym = TensorSymmetry(get_symmetric_group_sgs(2))
>>> T = TensorHead('T', [Lorentz]*2, sym)
```

Note, that the same can also be done using built-in `TensorSymmetry` methods

```
>>> sym2 = TensorSymmetry.fully_symmetric(2)
>>> sym == sym2
True
```

See also:

[`sympy.combinatorics.tensor\_can.get\_symmetric\_group\_sgs`](#) (page 370)

Attributes

base	(base of the BSGS)
generators	(generators of the BSGS)
rank	(rank of the tensor)

classmethod `direct_product(*args)`

Returns a `TensorSymmetry` object that is being a direct product of fully (anti)symmetric index permutation groups.

Notes

Some examples for different values of `(*args)`: (1) vector, equivalent to `TensorSymmetry.fully_symmetric(1)` (2) tensor with 2 symmetric indices, equivalent to `.fully_symmetric(2)` (-2) tensor with 2 antisymmetric indices, equivalent to `.fully_symmetric(-2)` (2, -2) tensor with the first 2 indices commuting and the last 2 anticommuting (1, 1, 1) tensor with 3 indices without any symmetry

classmethod `fully_symmetric(rank)`

Returns a fully symmetric (antisymmetric if `rank < 0`) `TensorSymmetry` object for `abs(rank)` indices.

classmethod `no_symmetry(rank)`

`TensorSymmetry` object for rank indices with no symmetry

classmethod `riemann()`

Returns a monoterm symmetry of the Riemann tensor

`sympy.tensor.tensor.tensorsymmetry(*args)`

Returns a `TensorSymmetry` object. This method is deprecated, use `TensorSymmetry.direct_product()` or `.riemann()` instead.

Explanation

One can represent a tensor with any monoterm slot symmetry group using a BSGS.

`args` can be a BSGS `args[0]` base `args[1]` sgs

Usually tensors are in (direct products of) representations of the symmetric group; `args` can be a list of lists representing the shapes of Young tableaux

Notes

For instance: `[[1]]` vector `[[1]*n]` symmetric tensor of rank `n` `[[n]]` antisymmetric tensor of rank `n` `[[2, 2]]` monoterm slot symmetry of the Riemann tensor `[[1],[1]]` vector*vector `[[2],[1],[1]]` (antisymmetric tensor)*vector*vector

Notice that with the shape `[2, 2]` we associate only the monoterm symmetries of the Riemann tensor; this is an abuse of notation, since the shape `[2, 2]` corresponds usually to the irreducible representation characterized by the monoterm symmetries and by the cyclic symmetry.

class sympy.tensor.tensor.**TensorType**(*args, **kwargs)

Class of tensor types. Deprecated, use `tensor_heads()` instead.

Parameters

index_types : list of `TensorIndexType` of the tensor indices

symmetry : `TensorSymmetry` of the tensor

Attributes

index_types	
symmetry	
types	(list of <code>TensorIndexType</code> without repetitions)

class sympy.tensor.tensor.**_TensorManager**

Class to manage tensor properties.

Notes

Tensors belong to tensor commutation groups; each group has a label `comm`; there are predefined labels:

0 tensors commuting with any other tensor

1 tensors anticommuting among themselves

2 tensors not commuting, apart with those with `comm=0`

Other groups can be defined using `set_comm`; tensors in those groups commute with those with `comm=0`; by default they do not commute with any other group.

clear()

Clear the `TensorManager`.

comm_i2symbol(i)

Returns the symbol corresponding to the commutation group number.

comm_symbols2i(i)

Get the commutation group number corresponding to `i`.

`i` can be a symbol or a number or a string.

If `i` is not already defined its commutation group number is set.

get_comm(i, j)

Return the commutation parameter for commutation group numbers `i`, `j`

see `_TensorManager.set_comm`

set_comm(i, j, c)

Set the commutation parameter `c` for commutation groups `i`, `j`.

Parameters

i, j : symbols representing commutation groups

c : group commutation number

Notes

i , j can be symbols, strings or numbers, apart from 0, 1 and 2 which are reserved respectively for commuting, anticommuting tensors and tensors not commuting with any other group apart with the commuting tensors. For the remaining cases, use this method to set the commutation rules; by default $c=None$.

The group commutation number c is assigned in correspondence to the group commutation symbols; it can be

0 commuting

1 anticommuting

None no commutation property

Examples

G and GH do not commute with themselves and commute with each other; A is commuting.

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, \
↳ TensorHead, TensorManager, TensorSymmetry
>>> Lorentz = TensorIndexType('Lorentz')
>>> i0,i1,i2,i3,i4 = tensor_indices('i0:5', Lorentz)
>>> A = TensorHead('A', [Lorentz])
>>> G = TensorHead('G', [Lorentz], TensorSymmetry.no_symmetry(1),
↳ 'Gcomm')
>>> GH = TensorHead('GH', [Lorentz], TensorSymmetry.no_symmetry(1),
↳ 'GHcomm')
>>> TensorManager.set_comm('Gcomm', 'GHcomm', 0)
>>> (GH(i1)*G(i0)).canon_bp()
G(i0)*GH(i1)
>>> (G(i1)*G(i0)).canon_bp()
G(i1)*G(i0)
>>> (G(i1)*A(i0)).canon_bp()
A(i0)*G(i1)
```

set_comms(*args)

Set the commutation group numbers c for symbols i , j .

Parameters

args : sequence of (i , j , c)

Tensor Operators

class sympy.tensor.toperators.**PartialDerivative**(*expr*, **variables*)

Partial derivative for tensor expressions.

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, TensorHead
>>> from sympy.tensor.toperators import PartialDerivative
>>> from sympy import symbols
>>> L = TensorIndexType("L")
>>> A = TensorHead("A", [L])
>>> B = TensorHead("B", [L])
>>> i, j, k = symbols("i j k")
```

```
>>> expr = PartialDerivative(A(i), A(j))
>>> expr
PartialDerivative(A(i), A(j))
```

The `PartialDerivative` object behaves like a tensorial expression:

```
>>> expr.get_indices()
[i, -j]
```

Notice that the deriving variables have opposite valence than the printed one: `A(j)` is printed as covariant, but the index of the derivative is actually contravariant, i.e. `-j`.

Indices can be contracted:

```
>>> expr = PartialDerivative(A(i), A(i))
>>> expr
PartialDerivative(A(L_0), A(L_0))
>>> expr.get_indices()
[L_0, -L_0]
```

The method `.get_indices()` always returns all indices (even the contracted ones). If only uncontracted indices are needed, call `.get_free_indices()`:

```
>>> expr.get_free_indices()
[]
```

Nested partial derivatives are flattened:

```
>>> expr = PartialDerivative(PartialDerivative(A(i), A(j)), A(k))
>>> expr
PartialDerivative(A(i), A(j), A(k))
>>> expr.get_indices()
[i, -j, -k]
```

Replace a derivative with array values:

```
>>> from sympy.abc import x, y
>>> from sympy import sin, log
>>> compA = [sin(x), log(x)*y**3]
>>> compB = [x, y]
>>> expr = PartialDerivative(A(i), B(j))
>>> expr.replace_with_arrays({A(i): compA, B(i): compB})
[[cos(x), 0], [y**3/x, 3*y**2*log(x)]]
```

The returned array is indexed by $(i, -j)$.

Be careful that other SymPy modules put the indices of the deriving variables before the indices of the derivand in the derivative result. For example:

```
>>> expr.get_free_indices()
[i, -j]
```

```
>>> from sympy import Matrix, Array
>>> Matrix(compA).diff(Matrix(compB)).reshape(2, 2)
[[cos(x), y**3/x], [0, 3*y**2*log(x)]]
>>> Array(compA).diff(Array(compB))
[[cos(x), y**3/x], [0, 3*y**2*log(x)]]
```

These are the transpose of the result of `PartialDerivative`, as the matrix and the array modules put the index $-j$ before i in the derivative result. An array read with index order $(-j, i)$ is indeed the transpose of the same array read with index order $(i, -j)$. By specifying the index order to `.replace_with_arrays` one can get a compatible expression:

```
>>> expr.replace_with_arrays({A(i): compA, B(i): compB}, [-j, i])
[[cos(x), y**3/x], [0, 3*y**2*log(x)]]
```

Vector

The vector module provides tools for basic vector math and differential calculus with respect to 3D Cartesian coordinate systems. This documentation provides an overview of all the features offered, and relevant API.

Guide to Vector

Introduction

This page gives a brief conceptual overview of the functionality present in `sympy.vector` (page 1425).

Vectors and Scalars

In vector math, we deal with two kinds of quantities – scalars and vectors.

A **scalar** is an entity which only has a magnitude – no direction. Examples of scalar quantities include mass, electric charge, temperature, distance, etc.

A **vector**, on the other hand, is an entity that is characterized by a magnitude and a direction. Examples of vector quantities are displacement, velocity, magnetic field, etc.

A scalar can be depicted just by a number, for e.g. a temperature of 300 K. On the other hand, vectorial quantities like acceleration are usually denoted by a vector. Given a vector $\mathbf{V}$, the magnitude of the corresponding quantity can be calculated as the magnitude of the vector itself $\|\mathbf{V}\|$, while the direction would be specified by a unit vector in the direction of the original vector, $\hat{\mathbf{V}} = \frac{\mathbf{V}}{\|\mathbf{V}\|}$.

For example, consider a displacement of $(3\hat{\mathbf{i}} + 4\hat{\mathbf{j}} + 5\hat{\mathbf{k}})$ m, where, as per standard convention, $\hat{\mathbf{i}}$, $\hat{\mathbf{j}}$ and $\hat{\mathbf{k}}$ represent unit vectors along the $\mathbf{X}$, $\mathbf{Y}$ and $\mathbf{Z}$ axes respectively. Therefore, it can be

concluded that the distance traveled is $\|3\hat{\mathbf{i}} + 4\hat{\mathbf{j}} + 5\hat{\mathbf{k}}\| \text{ m} = 5\sqrt{2} \text{ m}$. The direction of travel is given by the unit vector $\frac{3}{5\sqrt{2}}\hat{\mathbf{i}} + \frac{4}{5\sqrt{2}}\hat{\mathbf{j}} + \frac{5}{5\sqrt{2}}\hat{\mathbf{k}}$.

Coordinate Systems

A **coordinate system** is an abstract mathematical entity used to define the notion of directions and locations in n -dimensional spaces. This module deals with 3-dimensional spaces, with the conventional X , Y and Z axes defined with respect to each coordinate system.

Each coordinate system also has a special reference point called the ‘origin’ defined for it. This point is used either while referring to locations in 3D space, or while calculating the coordinates of pre-defined points with respect to the system.

It is a pretty well-known concept that there is no absolute notion of location or orientation in space. Any given coordinate system defines a unique ‘perspective’ of quantifying positions and directions. Therefore, even if we assume that all systems deal with the same units of measurement, the expression of vectorial and scalar quantities differs according to the coordinate system a certain observer deals with.

Consider two points P and Q in space. Assuming units to be common throughout, the distance between these points remains the same regardless of the coordinate system in which the measurements are being made. However, the 3-D coordinates of each of the two points, as well as the position vector of any of the points with respect to the other, do not. In fact, these two quantities don’t make sense at all, unless they are being measured keeping in mind a certain location and orientation of the measurer (essentially the coordinate system).

Therefore, it is quite clear that the orientation and location (of the origin) of a coordinate system define the way different quantities will be expressed with respect to it. Neither of the two properties can be measured on an absolute scale, but rather with respect to another coordinate system. The orientation of one system with respect to another is measured using the rotation matrix, while the relative position can be quantified via the position vector of one system’s origin with respect to the other.

Fields

A **field** is a vector or scalar quantity that can be specified everywhere in space as a function of position (Note that in general a field may also be dependent on time and other custom variables). Since we only deal with 3D spaces in this module, a field is defined as a function of the x , y and z coordinates corresponding to a location in the coordinate system. Here, x , y and z act as scalar variables defining the position of a general point.

For example, temperature in 3 dimensional space (a temperature field) can be written as $T(x, y, z)$ – a scalar function of the position. An example of a scalar field in electromagnetism is the electric potential.

In a similar manner, a vector field can be defined as a vectorial function of the location (x, y, z) of any point in space.

For instance, every point on the earth may be considered to be in the gravitational force field of the earth. We may specify the field by the magnitude and the direction of acceleration due to gravity (i.e. force per unit mass) $\vec{g}(x, y, z)$ at every point in space.

To give an example from electromagnetism, consider an electric potential of form $2x^2y$, a scalar field in 3D space. The corresponding conservative electric field can be computed as

the gradient of the electric potential function, and expressed as $4xy\hat{\mathbf{i}} + 2x^2\hat{\mathbf{j}}$. The magnitude of this electric field can in turn be expressed as a scalar field of the form $\sqrt{4x^4 + 16x^2y^2}$.

Basic Implementation details

Coordinate Systems and Vectors

Currently, `sympy.vector` (page 1425) is able to deal with the Cartesian (also called rectangular), spherical and other curvilinear coordinate systems.

A 3D Cartesian coordinate system can be initialized in `sympy.vector` (page 1425) as

```
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
```

The string parameter to the constructor denotes the name assigned to the system, and will primarily be used for printing purposes.

Once a coordinate system (in essence, a `CoordSys3D` instance) has been defined, we can access the orthonormal unit vectors (i.e. the $\hat{\mathbf{i}}$, $\hat{\mathbf{j}}$ and $\hat{\mathbf{k}}$ vectors) and coordinate variables/base scalars (i.e. the x , y and z variables) corresponding to it. We will talk about coordinate variables in the later sections.

The basis vectors for the X , Y and Z axes can be accessed using the `i`, `j` and `k` properties respectively.

```
>>> N.i
N.i
>>> type(N.i)
<class 'sympy.vector.vector.BaseVector'>
```

As seen above, the basis vectors are all instances of a class called `BaseVector`.

When a `BaseVector` is multiplied by a scalar (essentially any SymPy `Expr`), we get a `VectorMul` - the product of a base vector and a scalar.

```
>>> 3*N.i
3*N.i
>>> type(3*N.i)
<class 'sympy.vector.vector.VectorMul'>
```

Addition of `VectorMul` and `BaseVectors` gives rise to formation of `VectorAdd` - except for special cases, ofcourse.

```
>>> v = 2*N.i + N.j
>>> type(v)
<class 'sympy.vector.vector.VectorAdd'>
>>> v - N.j
2*N.i
>>> type(v - N.j)
<class 'sympy.vector.vector.VectorMul'>
```

What about a zero vector? It can be accessed using the `zero` attribute assigned to class `Vector`. Since the notion of a zero vector remains the same regardless of the coordinate system in consideration, we use `Vector.zero` wherever such a quantity is required.

```
>>> from sympy.vector import Vector
>>> Vector.zero
0
>>> type(Vector.zero)
<class 'sympy.vector.vector.VectorZero'>
>>> N.i + Vector.zero
N.i
>>> Vector.zero == 2*Vector.zero
True
```

All the classes shown above - BaseVector, VectorMul, VectorAdd and VectorZero are subclasses of Vector.

You should never have to instantiate objects of any of the subclasses of Vector. Using the BaseVector instances assigned to a CoordSys3D instance and (if needed) Vector.zero as building blocks, any sort of vectorial expression can be constructed with the basic mathematical operators +, -, *, and /.

```
>>> v = N.i - 2*N.j
>>> v/3
1/3*N.i + (-2/3)*N.j
>>> v + N.k
N.i + (-2)*N.j + N.k
>>> Vector.zero/2
0
>>> (v/3)*4
4/3*N.i + (-8/3)*N.j
```

In addition to the elementary mathematical operations, the vector operations of dot and cross can also be performed on Vector.

```
>>> v1 = 2*N.i + 3*N.j - N.k
>>> v2 = N.i - 4*N.j + N.k
>>> v1.dot(v2)
-11
>>> v1.cross(v2)
(-1)*N.i + (-3)*N.j + (-11)*N.k
>>> v2.cross(v1)
N.i + 3*N.j + 11*N.k
```

The & and ^ operators have been overloaded for the dot and cross methods respectively.

```
>>> v1 & v2
-11
>>> v1 ^ v2
(-1)*N.i + (-3)*N.j + (-11)*N.k
```

However, this is not the recommended way of performing these operations. Using the original methods makes the code clearer and easier to follow.

In addition to these operations, it is also possible to compute the outer products of Vector instances in [sympy.vector](#) (page 1425). More on that in a little bit.

SymPy operations on Vectors

The SymPy operations of `simplify`, `trigsimp`, `diff`, and `factor` work on Vector objects, with the standard SymPy API.

In essence, the methods work on the measure numbers (The coefficients of the basis vectors) present in the provided vectorial expression.

```
>>> from sympy.abc import a, b, c
>>> from sympy import sin, cos, trigsimp, diff
>>> v = (a*b + a*c + b**2 + b*c)*N.i + N.j
>>> v.factor()
((a + b)*(b + c))*N.i + N.j
>>> v = (sin(a)**2 + cos(a)**2)*N.i - (2*cos(b)**2 - 1)*N.k
>>> trigsimp(v)
N.i + (-cos(2*b))*N.k
>>> v.simplify()
N.i + (-cos(2*b))*N.k
>>> diff(v, b)
(4*sin(b)*cos(b))*N.k
>>> from sympy import Derivative
>>> Derivative(v, b).doit()
(4*sin(b)*cos(b))*N.k
```

Integral also works with Vector instances, similar to Derivative.

```
>>> from sympy import Integral
>>> v1 = a*N.i + sin(a)*N.j - N.k
>>> Integral(v1, a)
(Integral(a, a))*N.i + (Integral(sin(a), a))*N.j + (Integral(-1, a))*N.k
>>> Integral(v1, a).doit()
a**2/2*N.i + (-cos(a))*N.j + (-a)*N.k
```

Points

As mentioned before, every coordinate system corresponds to a unique origin point. Points, in general, have been implemented in `sympy.vector` (page 1425) in the form of the `Point` class.

To access the origin of system, use the `origin` property of the `CoordSys3D` class.

```
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
>>> N.origin
N.origin
>>> type(N.origin)
<class 'sympy.vector.point.Point'>
```

You can instantiate new points in space using the `locate_new` method of `Point`. The arguments include the name(string) of the new `Point`, and its position vector with respect to the 'parent' `Point`.

```
>>> from sympy.abc import a, b, c
>>> P = N.origin.locate_new('P', a*N.i + b*N.j + c*N.k)
>>> Q = P.locate_new('Q', -b*N.j)
```

Like `Vector`, a user never has to expressly instantiate an object of `Point`. This is because any location in space (albeit relative) can be pointed at by using the origin of a `CoordSys3D` as the reference, and then using `locate_new` on it and subsequent `Point` instances.

The position vector of a `Point` with respect to another `Point` can be computed using the `position_wrt` method.

```
>>> P.position_wrt(Q)
b*N.j
>>> Q.position_wrt(N.origin)
a*N.i + c*N.k
```

Additionally, it is possible to obtain the X , Y and Z coordinates of a `Point` with respect to a `CoordSys3D` in the form of a tuple. This is done using the `express_coordinates` method.

```
>>> Q.express_coordinates(N)
(a, 0, c)
```

Dyadics

A dyadic, or dyadic tensor, is a second-order tensor formed by the juxtaposition of pairs of vectors. Therefore, the outer products of vectors give rise to the formation of dyadics. Dyadic tensors have been implemented in [sympy.vector](#) (page 1425) in the `Dyadic` class.

Once again, you never have to instantiate objects of `Dyadic`. The outer products of vectors can be computed using the `outer` method of `Vector`. The `|` operator has been overloaded for `outer`.

```
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
>>> N.i.outer(N.j)
(N.i|N.j)
>>> N.i|N.j
(N.i|N.j)
```

Similar to `Vector`, `Dyadic` also has subsequent subclasses like `BaseDyadic`, `DyadicMul`, `DyadicAdd`. As with `Vector`, a zero dyadic can be accessed from `Dyadic.zero`.

All basic mathematical operations work with `Dyadic` too.

```
>>> dyad = N.i.outer(N.k)
>>> dyad*3
3*(N.i|N.k)
>>> dyad - dyad
0
>>> dyad + 2*(N.j|N.i)
(N.i|N.k) + 2*(N.j|N.i)
```

`dot` and `cross` also work among `Dyadic` instances as well as between a `Dyadic` and `Vector` (and also vice versa) - as per the respective mathematical definitions. As with `Vector`, & and

$\wedge$ have been overloaded for dot and cross.

```
>>> d = N.i.outer(N.j)
>>> d.dot(N.j|N.j)
(N.i|N.j)
>>> d.dot(N.i)
0
>>> d.dot(N.j)
N.i
>>> N.i.dot(d)
N.j
>>> N.k ^ d
(N.j|N.j)
```

More about Coordinate Systems

We will now look at how we can initialize new coordinate systems in [sympy.vector](#) (page 1425), transformed in user-defined ways with respect to already-existing systems.

Locating new systems

We already know that the `origin` property of a `CoordSys3D` corresponds to the `Point` instance denoting its origin reference point.

Consider a coordinate system N . Suppose we want to define a new system M , whose origin is located at $3\hat{i} + 4\hat{j} + 5\hat{k}$ from N 's origin. In other words, the coordinates of M 's origin from N 's perspective happen to be $(3, 4, 5)$. Moreover, this would also mean that the coordinates of N 's origin with respect to M would be $(-3, -4, -5)$.

This can be achieved programmatically as follows -

```
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
>>> M = N.locate_new('M', 3*N.i + 4*N.j + 5*N.k)
>>> M.position_wrt(N)
3*N.i + 4*N.j + 5*N.k
>>> N.origin.express_coordinates(M)
(-3, -4, -5)
```

It is worth noting that M 's orientation is the same as that of N . This means that the rotation matrix of N with respect to M , and also vice versa, is equal to the identity matrix of dimensions 3×3 . The `locate_new` method initializes a `CoordSys3D` that is only translated in space, not re-oriented, relative to the 'parent' system.

Orienting new systems

Similar to ‘locating’ new systems, [sympy.vector](#) (page 1425) also allows for initialization of new `CoordSys3D` instances that are oriented in user-defined ways with respect to existing systems.

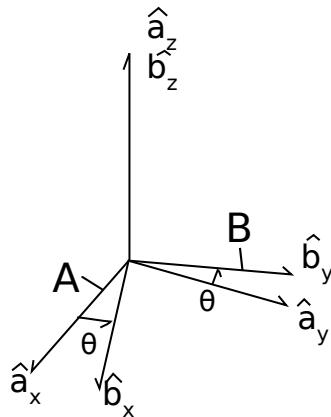
Suppose you have a coordinate system A .

```
>>> from sympy.vector import CoordSys3D
>>> A = CoordSys3D('A')
```

You want to initialize a new coordinate system B , that is rotated with respect to A ’s Z-axis by an angle θ .

```
>>> from sympy import Symbol
>>> theta = Symbol('theta')
```

The orientation is shown in the diagram below:



There are two ways to achieve this.

Using a method of `CoordSys3D` directly

This is the easiest, cleanest, and hence the recommended way of doing it.

```
>>> B = A.orient_new_axis('B', theta, A.k)
```

This initializes B with the required orientation information with respect to A .

`CoordSys3D` provides the following direct orientation methods in its API-

1. `orient_new_axis`
2. `orient_new_body`
3. `orient_new_space`
4. `orient_new_quaternion`

Please look at the `CoordSys3D` class API given in the docs of this module, to know their functionality and required arguments in detail.

Using Orienter(s) and the orient_new method

You would first have to initialize an `AxisOrienter` instance for storing the rotation information.

```
>>> from sympy.vector import AxisOrienter
>>> axis_orienter = AxisOrienter(theta, A.k)
```

And then apply it using the `orient_new` method, to obtain *B*.

```
>>> B = A.orient_new('B', axis_orienter)
```

`orient_new` also lets you orient new systems using multiple `Orienter` instances, provided in an iterable. The rotations/orientations are applied to the new system in the order the `Orienter` instances appear in the iterable.

```
>>> from sympy.vector import BodyOrienter
>>> from sympy.abc import a, b, c
>>> body_orienter = BodyOrienter(a, b, c, 'XYZ')
>>> C = A.orient_new('C', (axis_orienter, body_orienter))
```

The `sympy.vector` (page 1425) API provides the following four `Orienter` classes for orientation purposes-

1. `AxisOrienter`
2. `BodyOrienter`
3. `SpaceOrienter`
4. `QuaternionOrienter`

Please refer to the API of the respective classes in the docs of this module to know more.

In each of the above examples, the origin of the new coordinate system coincides with the origin of the 'parent' system.

```
>>> B.position_wrt(A)
0
```

To compute the rotation matrix of any coordinate system with respect to another one, use the `rotation_matrix` method.

```
>>> B = A.orient_new_axis('B', a, A.k)
>>> B.rotation_matrix(A)
Matrix([
[ cos(a), sin(a), 0],
[-sin(a), cos(a), 0],
[      0,      0, 1]])
>>> B.rotation_matrix(B)
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
```

Orienting AND Locating new systems

What if you want to initialize a new system that is not only oriented in a pre-defined way, but also translated with respect to the parent?

Each of the `orient_new_<method of orientation>` methods, as well as the `orient_new` method, support a `location` keyword argument.

If a `Vector` is supplied as the value for this `kwarg`, the new system's origin is automatically defined to be located at that position vector with respect to the parent coordinate system.

Thus, the orientation methods also act as methods to support orientation+ location of the new systems.

```
>>> C = A.orient_new_axis('C', a, A.k, location=2*A.j)
>>> C.position_wrt(A)
2*A.j
>>> from sympy.vector import express
>>> express(A.position_wrt(C), C)
(-2*sin(a))*C.i + (-2*cos(a))*C.j
```

More on the `express` function in a bit.

Transforming new system

The most general way of creating user-defined system is to use `transformation` parameter in `CoordSys3D`. Here we can define any transformation equations. If we are interested in some typical curvilinear coordinate system different that Cartesian, we can also use some predefined ones. It could be also possible to translate or rotate system by setting appropriate transformation equations.

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import sin, cos
>>> A = CoordSys3D('A', transformation='spherical')
>>> B = CoordSys3D('A', transformation=lambda x,y,z: (x*sin(y), x*cos(y), z))
```

In `CoordSys3D` is also dedicated method, `create_new` which works similarly to methods like `locate_new`, `orient_new_axis` etc.

```
>>> from sympy.vector import CoordSys3D
>>> A = CoordSys3D('A')
>>> B = A.create_new('B', transformation='spherical')
```

Expression of quantities in different coordinate systems

Vectors and Dyadics

As mentioned earlier, the same vector attains different expressions in different coordinate systems. In general, the same is true for scalar expressions and dyadic tensors.

[`sympy.vector`](#) (page 1425) supports the expression of vector/scalar quantities in different coordinate systems using the `express` function.

For purposes of this section, assume the following initializations-

```
>>> from sympy.vector import CoordSys3D, express
>>> from sympy.abc import a, b, c
>>> N = CoordSys3D('N')
>>> M = N.orient_new_axis('M', a, N.k)
```

Vector instances can be expressed in user defined systems using `express`.

```
>>> v1 = N.i + N.j + N.k
>>> express(v1, M)
(sin(a) + cos(a))*M.i + (-sin(a) + cos(a))*M.j + M.k
>>> v2 = N.i + M.j
>>> express(v2, N)
(1 - sin(a))*N.i + (cos(a))*N.j
```

Apart from Vector instances, `express` also supports reexpression of scalars (general SymPy Expr) and Dyadic objects.

`express` also accepts a second coordinate system for re-expressing Dyadic instances.

```
>>> d = 2*(M.i | N.j) + 3*(M.j | N.k)
>>> express(d, M)
(2*sin(a))*(M.i|M.i) + (2*cos(a))*(M.i|M.j) + 3*(M.j|M.k)
>>> express(d, M, N)
2*(M.i|N.j) + 3*(M.j|N.k)
```

Coordinate Variables

The location of a coordinate system's origin does not affect the re-expression of `BaseVector` instances. However, it does affect the way `BaseScalar` instances are expressed in different systems.

`BaseScalar` instances, are coordinate 'symbols' meant to denote the variables used in the definition of vector/scalar fields in `sympy.vector` (page 1425).

For example, consider the scalar field $\mathbf{T}_N(\mathbf{x}, \mathbf{y}, \mathbf{z}) = \mathbf{x} + \mathbf{y} + \mathbf{z}$ defined in system N . Thus, at a point with coordinates (a, b, c) , the value of the field would be $a + b + c$. Now consider system R , whose origin is located at $(1, 2, 3)$ with respect to N (no change of orientation). A point with coordinates (a, b, c) in R has coordinates $(a + 1, b + 2, c + 3)$ in N . Therefore, the expression for $\mathbf{T}_N$ in R becomes $\mathbf{T}_R(x, y, z) = x + y + z + 6$.

Coordinate variables, if present in a vector/scalar/dyadic expression, can also be re-expressed in a given coordinate system, by setting the `variables` keyword argument of `express` to `True`.

The above mentioned example, done programmatically, would look like this -

```
>>> R = N.locate_new('R', N.i + 2*N.j + 3*N.k)
>>> T_N = N.x + N.y + N.z
>>> express(T_N, R, variables=True)
R.x + R.y + R.z + 6
```

Other expression-dependent methods

The `to_matrix` method of `Vector` and `express_coordinates` method of `Point` also return different results depending on the coordinate system being provided.

```
>>> P = R.origin.locate_new('P', a*R.i + b*R.j + c*R.k)
>>> P.express_coordinates(N)
(a + 1, b + 2, c + 3)
>>> P.express_coordinates(R)
(a, b, c)
>>> v = N.i + N.j + N.k
>>> v.to_matrix(M)
Matrix([
[ sin(a) + cos(a)],
[-sin(a) + cos(a)],
[          1]])
>>> v.to_matrix(N)
Matrix([
[1],
[1],
[1]])
```

Scalar and Vector Field Functionality

Implementation in `sympy.vector`

Scalar and vector fields

In [sympy.vector](#) (page 1425), every `CoordSys3D` instance is assigned basis vectors corresponding to the X , Y and Z axes. These can be accessed using the properties named `i`, `j` and `k` respectively. Hence, to define a vector $\mathbf{v}$ of the form $3\hat{\mathbf{i}} + 4\hat{\mathbf{j}} + 5\hat{\mathbf{k}}$ with respect to a given frame $\mathbf{R}$, you would do

```
>>> from sympy.vector import CoordSys3D
>>> R = CoordSys3D('R')
>>> v = 3*R.i + 4*R.j + 5*R.k
```

Vector math and basic calculus operations with respect to vectors have already been elaborated upon in the earlier section of this module's documentation.

On the other hand, base scalars (or coordinate variables) are implemented in a special class called `BaseScalar`, and are assigned to every coordinate system, one for each axis from X , Y and Z . These coordinate variables are used to form the expressions of vector or scalar fields in 3D space. For a system $\mathbf{R}$, the X , Y and Z `BaseScalars` instances can be accessed using the `R.x`, `R.y` and `R.z` expressions respectively.

Therefore, to generate the expression for the aforementioned electric potential field $2x^2y$, you would have to do

```
>>> from sympy.vector import CoordSys3D
>>> R = CoordSys3D('R')
>>> electric_potential = 2*R.x**2*R.y
```

(continues on next page)

(continued from previous page)

```
>>> electric_potential
2*R.x**2*R.y
```

It is to be noted that `BaseScalar` instances can be used just like any other SymPy `Symbol`, except that they store the information about the coordinate system and axis they correspond to.

Scalar fields can be treated just as any other SymPy expression, for any math/calculus functionality. Hence, to differentiate the above electric potential with respect to x (i.e. $R.x$), you would use the `diff` method.

```
>>> from sympy.vector import CoordSys3D
>>> R = CoordSys3D('R')
>>> electric_potential = 2*R.x**2*R.y
>>> from sympy import diff
>>> diff(electric_potential, R.x)
4*R.x*R.y
```

It is worth noting that having a `BaseScalar` in the expression implies that a ‘field’ changes with position, in 3D space. Technically speaking, a simple `Expr` with no `BaseScalar`s is still a field, though constant.

Like scalar fields, vector fields that vary with position can also be constructed using `BaseScalar`s in the measure-number expressions.

```
>>> from sympy.vector import CoordSys3D
>>> R = CoordSys3D('R')
>>> v = R.x**2*R.i + 2*R.x*R.z*R.k
```

The Del operator

The Del, or ‘Nabla’ operator - written as ∇ is commonly known as the vector differential operator. Depending on its usage in a mathematical expression, it may denote the gradient of a scalar field, the divergence of a vector field, or the curl of a vector field.

Essentially, ∇ is not technically an ‘operator’, but a convenient mathematical notation to denote any one of the aforementioned field operations.

In [sympy.vector](#) (page 1425), ∇ has been implemented as the `Del()` class. The instance of this class is independent of coordinate system. Hence, the ∇ operator would be accessible as `Del()`.

Given below is an example of usage of the `Del()` class.

```
>>> from sympy.vector import CoordSys3D, Del
>>> C = CoordSys3D('C')
>>> delop = Del()
>>> gradient_field = delop(C.x*C.y*C.z)
>>> gradient_field
(Derivative(C.x*C.y*C.z, C.x))*C.i + (Derivative(C.x*C.y*C.z, C.y))*C.j
+ (Derivative(C.x*C.y*C.z, C.z))*C.k
```

The above expression can be evaluated using the SymPy `doit()` routine.

```
>>> gradient_field.doit()  
C.y*C.z*C.i + C.x*C.z*C.j + C.x*C.y*C.k
```

Usage of the ∇ notation in [sympy.vector](#) (page 1425) has been described in greater detail in the subsequent subsections.

Field operators and related functions

Here we describe some basic field-related functionality implemented in [sympy.vector](#) (page 1425).

Curl

A curl is a mathematical operator that describes an infinitesimal rotation of a vector in 3D space. The direction is determined by the right-hand rule (along the axis of rotation), and the magnitude is given by the magnitude of rotation.

In the 3D Cartesian system, the curl of a 3D vector $\mathbf{F}$, denoted by $\nabla \times \mathbf{F}$ is given by:

$$\nabla \times \mathbf{F} = \left(\frac{\partial F_z}{\partial y} - \frac{\partial F_y}{\partial z} \right) \hat{\mathbf{i}} + \left(\frac{\partial F_x}{\partial z} - \frac{\partial F_z}{\partial x} \right) \hat{\mathbf{j}} + \left(\frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial y} \right) \hat{\mathbf{k}}$$

where F_x denotes the X component of vector $\mathbf{F}$.

Computing the curl of a vector field in [sympy.vector](#) (page 1425) can be accomplished in two ways.

One, by using the `Del()` class

```
>>> from sympy.vector import CoordSys3D, Del  
>>> C = CoordSys3D('C')  
>>> delop = Del()  
>>> delop.cross(C.x*C.y*C.z*C.i).doit()  
C.x*C.y*C.j + (-C.x*C.z)*C.k  
>>> (delop ^ C.x*C.y*C.z*C.i).doit()  
C.x*C.y*C.j + (-C.x*C.z)*C.k
```

Or by using the dedicated function

```
>>> from sympy.vector import curl  
>>> curl(C.x*C.y*C.z*C.i)  
C.x*C.y*C.j + (-C.x*C.z)*C.k
```

Divergence

Divergence is a vector operator that measures the magnitude of a vector field's source or sink at a given point, in terms of a signed scalar.

The divergence operator always returns a scalar after operating on a vector.

In the 3D Cartesian system, the divergence of a 3D vector $\mathbf{F}$, denoted by $\nabla \cdot \mathbf{F}$ is given by:

$$\nabla \cdot \mathbf{F} = \frac{\partial U}{\partial x} + \frac{\partial V}{\partial y} + \frac{\partial W}{\partial z}$$

where U , V and W denote the X , Y and Z components of $\mathbf{F}$ respectively.

Computing the divergence of a vector field in [sympy.vector](#) (page 1425) can be accomplished in two ways.

One, by using the `Del()` class

```
>>> from sympy.vector import CoordSys3D, Del
>>> C = CoordSys3D('C')
>>> delop = Del()
>>> delop.dot(C.x*C.y*C.z*(C.i + C.j + C.k)).doit()
C.x*C.y + C.x*C.z + C.y*C.z
>>> (delop & C.x*C.y*C.z*(C.i + C.j + C.k)).doit()
C.x*C.y + C.x*C.z + C.y*C.z
```

Or by using the dedicated function

```
>>> from sympy.vector import divergence
>>> divergence(C.x*C.y*C.z*(C.i + C.j + C.k))
C.x*C.y + C.x*C.z + C.y*C.z
```

Gradient

Consider a scalar field $f(x, y, z)$ in 3D space. The gradient of this field is defined as the vector of the 3 partial derivatives of f with respect to x , y and z in the X , Y and Z axes respectively.

In the 3D Cartesian system, the divergence of a scalar field f , denoted by ∇f is given by -

$$\nabla f = \frac{\partial f}{\partial x} \hat{\mathbf{i}} + \frac{\partial f}{\partial y} \hat{\mathbf{j}} + \frac{\partial f}{\partial z} \hat{\mathbf{k}}$$

Computing the divergence of a vector field in [sympy.vector](#) (page 1425) can be accomplished in two ways.

One, by using the `Del()` class

```
>>> from sympy.vector import CoordSys3D, Del
>>> C = CoordSys3D('C')
>>> delop = Del()
>>> delop.gradient(C.x*C.y*C.z).doit()
C.y*C.z*C.i + C.x*C.z*C.j + C.x*C.y*C.k
>>> delop(C.x*C.y*C.z).doit()
C.y*C.z*C.i + C.x*C.z*C.j + C.x*C.y*C.k
```

Or by using the dedicated function

```
>>> from sympy.vector import gradient
>>> gradient(C.x*C.y*C.z)
C.y*C.z*C.i + C.x*C.z*C.j + C.x*C.y*C.k
```

Directional Derivative

Apart from the above three common applications of ∇ , it is also possible to compute the directional derivative of a field wrt a Vector in [sympy.vector](#) (page 1425).

By definition, the directional derivative of a field $\mathbf{F}$ along a vector v at point x represents the instantaneous rate of change of $\mathbf{F}$ moving through x with the velocity v . It is represented mathematically as: $(\vec{v} \cdot \nabla) \mathbf{F}(x)$.

Directional derivatives of vector and scalar fields can be computed in [sympy.vector](#) (page 1425) using the `Del()` class

```
>>> from sympy.vector import CoordSys3D, Del
>>> C = CoordSys3D('C')
>>> delop = Del()
>>> vel = C.i + C.j + C.k
>>> scalar_field = C.x*C.y*C.z
>>> vector_field = C.x*C.y*C.z*C.i
>>> (vel.dot(delop))(scalar_field)
C.x*C.y + C.x*C.z + C.y*C.z
>>> (vel & delop)(vector_field)
(C.x*C.y + C.x*C.z + C.y*C.z)*C.i
```

Or by using the dedicated function

```
>>> from sympy.vector import directional_derivative
>>> directional_derivative(C.x*C.y*C.z, 3*C.i + 4*C.j + C.k)
C.x*C.y + 4*C.x*C.z + 3*C.y*C.z
```

Field operator in orthogonal curvilinear coordinate system

vector package supports calculation in different kind of orthogonal curvilinear coordinate system. To do that, scaling factor (also known as Lamé coefficients) are used to express curl, divergence or gradient in desired type of coordinate system.

For example if we want to calculate gradient in cylindrical coordinate system all we need to do is to create proper coordinate system

```
>>> from sympy.vector import CoordSys3D
>>> c = CoordSys3D('c', transformation='cylindrical', variable_names=("r",
↪ "theta", "z"))
>>> gradient(c.r*c.theta*c.z)
c.theta*c.z*c.i + c.z*c.j + c.r*c.theta*c.k
```


Conservative and Solenoidal fields

In vector calculus, a conservative field is a field that is the gradient of some scalar field. Conservative fields have the property that their line integral over any path depends only on the end-points, and is independent of the path travelled. A conservative vector field is also said to be 'irrotational', since the curl of a conservative field is always zero.

In physics, conservative fields represent forces in physical systems where energy is conserved.

To check if a vector field is conservative in [sympy.vector](#) (page 1425), the `is_conservative` function can be used.

```
>>> from sympy.vector import CoordSys3D, is_conservative
>>> R = CoordSys3D('R')
>>> field = R.y*R.z*R.i + R.x*R.z*R.j + R.x*R.y*R.k
>>> is_conservative(field)
True
>>> curl(field)
0
```

A solenoidal field, on the other hand, is a vector field whose divergence is zero at all points in space.

To check if a vector field is solenoidal in [sympy.vector](#) (page 1425), the `is_solenoidal` function can be used.

```
>>> from sympy.vector import CoordSys3D, is_solenoidal
>>> R = CoordSys3D('R')
>>> field = R.y*R.z*R.i + R.x*R.z*R.j + R.x*R.y*R.k
>>> is_solenoidal(field)
True
>>> divergence(field)
0
```

Scalar potential functions

We have previously mentioned that every conservative field can be defined as the gradient of some scalar field. This scalar field is also called the 'scalar potential field' corresponding to the aforementioned conservative field.

The `scalar_potential` function in [sympy.vector](#) (page 1425) calculates the scalar potential field corresponding to a given conservative vector field in 3D space - minus the extra constant of integration, of course.

Example of usage -

```
>>> from sympy.vector import CoordSys3D, scalar_potential
>>> R = CoordSys3D('R')
>>> conservative_field = 4*R.x*R.y*R.z*R.i + 2*R.x**2*R.z*R.j + 2*R.x**2*R.
↪ y*R.k
>>> scalar_potential(conservative_field, R)
2*R.x**2*R.y*R.z
```

Providing a non-conservative vector field as an argument to `scalar_potential` raises a `ValueError`.

The scalar potential difference, or simply ‘potential difference’, corresponding to a conservative vector field can be defined as the difference between the values of its scalar potential function at two points in space. This is useful in calculating a line integral with respect to a conservative function, since it depends only on the endpoints of the path.

This computation is performed as follows in `sympy.vector` (page 1425).

```
>>> from sympy.vector import CoordSys3D, Point
>>> from sympy.vector import scalar_potential_difference
>>> R = CoordSys3D('R')
>>> P = R.origin.locate_new('P', 1*R.i + 2*R.j + 3*R.k)
>>> vectfield = 4*R.x*R.y*R.i + 2*R.x**2*R.j
>>> scalar_potential_difference(vectfield, R, R.origin, P)
4
```

If provided with a scalar expression instead of a vector field, `scalar_potential_difference` returns the difference between the values of that scalar field at the two given points in space.

General examples of usage

This section details the solution of two basic problems in vector math/calculus using the `sympy.vector` (page 1425) package.

Quadrilateral problem

The Problem

OABC is any quadrilateral in 3D space. P is the midpoint of OA, Q is the midpoint of AB, R is the midpoint of BC and S is the midpoint of OC. Prove that PQ is parallel to SR

Solution

The solution to this problem demonstrates the usage of `Point`, and basic operations on `Vector`.

Define a coordinate system

```
>>> from sympy.vector import CoordSys3D
>>> Sys = CoordSys3D('Sys')
```

Define point O to be Sys’ origin. We can do this without loss of generality

```
>>> O = Sys.origin
```

Define point A with respect to O

```
>>> from sympy import symbols
>>> a1, a2, a3 = symbols('a1 a2 a3')
>>> A = O.locate_new('A', a1*Sys.i + a2*Sys.j + a3*Sys.k)
```

Similarly define points B and C

```
>>> b1, b2, b3 = symbols('b1 b2 b3')
>>> B = O.locate_new('B', b1*Sys.i + b2*Sys.j + b3*Sys.k)
>>> c1, c2, c3 = symbols('c1 c2 c3')
>>> C = O.locate_new('C', c1*Sys.i + c2*Sys.j + c3*Sys.k)
```

P is the midpoint of OA. Lets locate it with respect to O (you could also define it with respect to A).

```
>>> P = O.locate_new('P', A.position_wrt(O) + (O.position_wrt(A) / 2))
```

Similarly define points Q, R and S as per the problem definitions.

```
>>> Q = A.locate_new('Q', B.position_wrt(A) / 2)
>>> R = B.locate_new('R', C.position_wrt(B) / 2)
>>> S = O.locate_new('R', C.position_wrt(O) / 2)
```

Now compute the vectors in the directions specified by PQ and SR.

```
>>> PQ = Q.position_wrt(P)
>>> SR = R.position_wrt(S)
```

Compute cross product

```
>>> PQ.cross(SR)
0
```

Since the cross product is a zero vector, the two vectors have to be parallel, thus proving that $PQ \parallel SR$.

Third product rule for Del operator

See

The Problem

Prove the third rule - $\nabla \cdot (f\vec{v}) = f(\nabla \cdot \vec{v}) + \vec{v} \cdot (\nabla f)$

Solution

Start with a coordinate system

```
>>> from sympy.vector import CoordSys3D, Del
>>> delop = Del()
>>> C = CoordSys3D('C')
```

The scalar field f and the measure numbers of the vector field $\vec{v}$ are all functions of the coordinate variables of the coordinate system in general. Hence, define SymPy functions that way.

```
>>> from sympy import symbols, Function
>>> v1, v2, v3, f = symbols('v1 v2 v3 f', cls=Function)
```

v_1 , v_2 and v_3 are the X , Y and Z components of the vector field respectively.

Define the vector field as `vfield` and the scalar field as `sfield`.

```
>>> vfield = v1(C.x, C.y, C.z)*C.i + v2(C.x, C.y, C.z)*C.j + v3(C.x, C.y, C.
↪z)*C.k
>>> ffield = f(C.x, C.y, C.z)
```

Construct the expression for the LHS of the equation using `Del()`.

```
>>> lhs = (delop.dot(ffield * vfield)).doit()
```

Similarly, the RHS would be defined.

```
>>> rhs = ((vfield.dot(delop(ffield))) + (ffield * (delop.dot(vfield))))
↪.doit()
```

Now, to prove the product rule, we would just need to equate the expanded and simplified versions of the lhs and the rhs, so that the SymPy expressions match.

```
>>> lhs.expand().simplify() == rhs.expand().doit().simplify()
True
```

Thus, the general form of the third product rule mentioned above can be proven using [*sympy.vector*](#) (page 1425).

Applications of Vector Integrals

To integrate a scalar or vector field over a region, we have to first define a region. SymPy provides three methods for defining a region:

1. Using Parametric Equations with [*ParametricRegion*](#) (page 1461).
2. Using Implicit Equation with [*ImplicitRegion*](#) (page 1462).
3. Using objects of geometry module.

The [*vector_integrate\(\)*](#) (page 1475) function is used to integrate scalar or vector field over any type of region. It automatically determines the type of integration (line, surface, or volume) depending on the nature of the object.

We define a coordinate system and make necessary imports for examples.

```
>>> from sympy import sin, cos, exp, pi, symbols
>>> from sympy.vector import CoordSys3D, ParametricRegion, ImplicitRegion,
↪vector_integrate
>>> from sympy.abc import r, x, y, z, theta, phi
>>> C = CoordSys3D('C')
```

Calculation of Perimeter, Surface Area, and Volume

To calculate the perimeter of a circle, we need to define it. Let's define it using its parametric equation.

```
>>> param_circle = ParametricRegion((4*cos(theta), 4*sin(theta)), (theta, 0, 2*pi))
```

We can also define a circle using its implicit equation.

```
>>> implicit_circle = ImplicitRegion((x, y), x**2 + y**2 - 4)
```

The perimeter of a figure is equal to the absolute value of its integral over a unit scalar field.

```
>>> vector_integrate(1, param_circle)
8*pi
>>> vector_integrate(1, implicit_circle)
4*pi
```

Suppose a user wants to calculate the perimeter of a triangle. Determining the parametric representation of a triangle can be difficult. Instead, the user can use an object of *Polygon* (page 2273) class in the geometry module.

```
>>> from sympy.geometry import Point, Polygon
>>> triangle = Polygon(Point(1, 2), (3, 5), (1,6))
>>> vector_integrate(1, triangle)
sqrt(5) + sqrt(13) + 4
```

To define a solid sphere, we need to use three parameters (r, theta and phi). For *ParametricRegion* (page 1461) object, the order of limits determine the sign of the integral.

```
>>> solidsphere = ParametricRegion((r*sin(phi)*cos(theta),
    r*sin(phi)*sin(theta), r*cos(phi)),
    (phi, 0, pi), (theta, 0, 2*pi), (r, 0, 3))
>>> vector_integrate(1, solidsphere)
36*pi
```

Calculation of mass of a body

Consider a triangular lamina R with vertices (0,0), (0, 5), (5,0) and with density $\rho(x, y) = xy \text{ kg/m}^2$. Find the total mass.

```
>>> triangle = ParametricRegion((x, y), (x, 0, 5), (y, 0, 5 - x))
>>> vector_integrate(C.x*C.y, triangle)
625/24
```

Find the mass of a cylinder centered on the z-axis which has height h , radius a , and density $\rho = x^2 + y^2 \text{ kg/m}^2$.

```
>>> a, h = symbols('a h', positive=True)
>>> cylinder = ParametricRegion((r*cos(theta), r*sin(theta), z),
    (theta, 0, 2*pi), (z, 0, h), (r, 0, a))
...
```

(continues on next page)

(continued from previous page)

```
>>> vector_integrate(C.x**2 + C.y**2, cylinder)
pi*a**4*h/2
```

Calculation of Flux

1. Consider a region of space in which there is a constant vectorfield $E(x, y, z) = a\hat{\mathbf{k}}$. A hemisphere of radius r lies on the x - y plane. What is the flux of the field through the sphere?

```
>>> semisphere = ParametricRegion((r*sin(phi)*cos(theta), \
    ↪ r*sin(phi)*sin(theta), r*cos(phi)), \
    ...                               (phi, 0, pi/2), (theta, 0, 2*pi))
>>> flux = vector_integrate(a*C.k, semisphere)
>>> flux
pi*a*r**2
```

2. Consider a region of space in which there is a vector field $E(x, y, z) = x^2\hat{\mathbf{k}}$ above the x - y plane, and a field $E(x, y, z) = y^2\hat{\mathbf{k}}$ below the x - y plane. What is the flux of that vector field through a cube of side length L with its center at the origin?"

The field is parallel to the z -axis so only the top and bottom face of the box will contribute to flux.

```
>>> L = symbols('L', positive=True)
>>> top_face = ParametricRegion((x, y, L/2), (x, -L/2, L/2), (y, -L/2, L/2))
>>> bottom_face = ParametricRegion((x, y, -L/2), (x, -L/2, L/2), (y, -L/2, L/
    ↪ 2))
>>> flux = vector_integrate(C.x**2*C.k, top_face) + vector_integrate(C.y**2*C.
    ↪ k, bottom_face)
>>> flux
L**4/6
```

Verifying Stoke's Theorem

See https://en.wikipedia.org/wiki/Stokes%27_theorem

Example 1

```
>>> from sympy.vector import curl
>>> curve = ParametricRegion((cos(theta), sin(theta)), (theta, 0, pi/2))
>>> surface = ParametricRegion((r*cos(theta), r*sin(theta)), (r, 0, 1), \
    ↪ (theta, 0, pi/2))
>>> F = C.y*C.i + C.z*C.k + C.x*C.k
>>>
>>> vector_integrate(F, curve)
-pi/4
>>> vector_integrate(curl(F), surface)
-pi/4
```

Example 2

```

>>> circle = ParametricRegion((cos(theta), sin(theta), 1), (theta, 0, 2*pi))
>>> cone = ParametricRegion((r*cos(theta), r*sin(theta), r), (r, 0, 1), (theta, 0, 2*pi))
>>> cone = ParametricRegion((r*cos(theta), r*sin(theta), r), (r, 0, 1), (theta, 0, 2*pi))
>>> f = (-C.y**3/3 + sin(C.x))*C.i + (C.x**3/3 + cos(C.y))*C.j + C.x*C.y*C.z*C.k
>>> vector_integrate(f, circle)
pi/2
>>> vector_integrate(curl(f), cone)
pi/2

```

Verifying Divergence Theorem

See https://en.wikipedia.org/wiki/Divergence_theorem

Example 1

```

>>> from sympy.vector import divergence
>>> sphere = ParametricRegion((4*sin(phi)*cos(theta), 4*sin(phi)*sin(theta), 4*cos(phi)), (phi, 0, pi), (theta, 0, 2*pi))
>>> solidsphere = ParametricRegion((r*sin(phi)*cos(theta), r*sin(phi)*sin(theta), r*cos(phi)), (r, 0, 4), (phi, 0, pi), (theta, 0, 2*pi))
>>> field = C.x**3*C.i + C.y**3*C.j + C.z**3*C.k
>>> vector_integrate(field, sphere)
12288*pi/5
>>> vector_integrate(divergence(field), solidsphere)
12288*pi/5

```

Example 2

```

>>> cube = ParametricRegion((x, y, z), (x, 0, 1), (y, 0, 1), (z, 0, 1))
>>> field = 2*C.x*C.y*C.i + 3*C.x*C.y*C.j + C.z*exp(C.x + C.y)*C.k
>>> vector_integrate(divergence(field), cube)
-E + 7/2 + E*(-1 + E)

```

Vector API

Essential Classes in sympy.vector (docstrings)

CoordSys3D

```

class sympy.vector.coordsysrect.CoordSys3D(name, transformation=None,
                                             parent=None, location=None,
                                             rotation_matrix=None,
                                             vector_names=None,
                                             variable_names=None)

```

Represents a coordinate system in 3-D space.

```
__init__(name, location=None, rotation_matrix=None, parent=None,  
          vector_names=None, variable_names=None, latex_vects=None,  
          pretty_vects=None, latex_scalars=None, pretty_scalars=None,  
          transformation=None)
```

The orientation/location parameters are necessary if this system is being defined at a certain orientation or location wrt another.

Parameters

name : str

The name of the new CoordSys3D instance.

transformation : Lambda, Tuple, str

Transformation defined by transformation equations or chosen from predefined ones.

location : Vector

The position vector of the new system's origin wrt the parent instance.

rotation_matrix : SymPy ImmutableMatrix

The rotation matrix of the new coordinate system with respect to the parent. In other words, the output of `new_system.rotation_matrix(parent)`.

parent : CoordSys3D

The coordinate system wrt which the orientation/location (or both) is being defined.

vector_names, variable_names : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

```
create_new(name, transformation, variable_names=None, vector_names=None)
```

Returns a CoordSys3D which is connected to self by transformation.

Parameters

name : str

The name of the new CoordSys3D instance.

transformation : Lambda, Tuple, str

Transformation defined by transformation equations or chosen from predefined ones.

vector_names, variable_names : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> a = CoordSys3D('a')
>>> b = a.create_new('b', transformation='spherical')
>>> b.transformation_to_parent()
(b.r*sin(b.theta)*cos(b.phi), b.r*sin(b.phi)*sin(b.theta), b.r*cos(b.
  ↪ theta))
>>> b.transformation_from_parent()
(sqrt(a.x**2 + a.y**2 + a.z**2), acos(a.z/sqrt(a.x**2 + a.y**2 + a.
  ↪ z**2)), atan2(a.y, a.x))
```

locate_new(*name*, *position*, *vector_names*=None, *variable_names*=None)

Returns a CoordSys3D with its origin located at the given position wrt this coordinate system's origin.

Parameters

name : str

The name of the new CoordSys3D instance.

position : Vector

The position vector of the new system's origin wrt this one.

vector_names, **variable_names** : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> A = CoordSys3D('A')
>>> B = A.locate_new('B', 10 * A.i)
>>> B.origin.position_wrt(A.origin)
10*A.i
```

orient_new(*name*, *orienters*, *location*=None, *vector_names*=None, *variable_names*=None)

Creates a new CoordSys3D oriented in the user-specified way with respect to this system.

Please refer to the documentation of the orienter classes for more information about the orientation procedure.

Parameters

name : str

The name of the new CoordSys3D instance.

orienters : iterable/Orienter

An Orienter or an iterable of Orienters for orienting the new coordinate system. If an Orienter is provided, it is applied to get the new

system. If an iterable is provided, the orienters will be applied in the order in which they appear in the iterable.

location : Vector(optional)

The location of the new coordinate system's origin wrt this system's origin. If not specified, the origins are taken to be coincident.

vector_names, variable_names : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q0, q1, q2, q3 = symbols('q0 q1 q2 q3')
>>> N = CoordSys3D('N')
```

Using an AxisOrienter

```
>>> from sympy.vector import AxisOrienter
>>> axis_orienter = AxisOrienter(q1, N.i + 2 * N.j)
>>> A = N.orient_new('A', (axis_orienter, ))
```

Using a BodyOrienter

```
>>> from sympy.vector import BodyOrienter
>>> body_orienter = BodyOrienter(q1, q2, q3, '123')
>>> B = N.orient_new('B', (body_orienter, ))
```

Using a SpaceOrienter

```
>>> from sympy.vector import SpaceOrienter
>>> space_orienter = SpaceOrienter(q1, q2, q3, '312')
>>> C = N.orient_new('C', (space_orienter, ))
```

Using a QuaternionOrienter

```
>>> from sympy.vector import QuaternionOrienter
>>> q_orienter = QuaternionOrienter(q0, q1, q2, q3)
>>> D = N.orient_new('D', (q_orienter, ))
```

orient_new_axis(*name, angle, axis, location=None, vector_names=None, variable_names=None*)

Axis rotation is a rotation about an arbitrary axis by some angle. The angle is supplied as a SymPy expr scalar, and the axis is supplied as a Vector.

Parameters

name : string

The name of the new coordinate system

angle : Expr

The angle by which the new system is to be rotated

axis : Vector

The axis around which the rotation has to be performed

location : Vector(optional)

The location of the new coordinate system's origin wrt this system's origin. If not specified, the origins are taken to be coincident.

vector_names, variable_names : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q1 = symbols('q1')
>>> N = CoordSys3D('N')
>>> B = N.orient_new_axis('B', q1, N.i + 2 * N.j)
```

orient_new_body(*name, angle1, angle2, angle3, rotation_order, location=None, vector_names=None, variable_names=None*)

Body orientation takes this coordinate system through three successive simple rotations.

Body fixed rotations include both Euler Angles and Tait-Bryan Angles, see https://en.wikipedia.org/wiki/Euler_angles.

Parameters

name : string

The name of the new coordinate system

angle1, angle2, angle3 : Expr

Three successive angles to rotate the coordinate system by

rotation_order : string

String defining the order of axes for rotation

location : Vector(optional)

The location of the new coordinate system's origin wrt this system's origin. If not specified, the origins are taken to be coincident.

vector_names, variable_names : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q1, q2, q3 = symbols('q1 q2 q3')
>>> N = CoordSys3D('N')
```

A ‘Body’ fixed rotation is described by three angles and three body-fixed rotation axes. To orient a coordinate system D with respect to N, each sequential rotation is always about the orthogonal unit vectors fixed to D. For example, a ‘123’ rotation will specify rotations about N.i, then D.j, then D.k. (Initially, D.i is same as N.i) Therefore,

```
>>> D = N.orient_new_body('D', q1, q2, q3, '123')
```

is same as

```
>>> D = N.orient_new_axis('D', q1, N.i)
>>> D = D.orient_new_axis('D', q2, D.j)
>>> D = D.orient_new_axis('D', q3, D.k)
```

Acceptable rotation orders are of length 3, expressed in XYZ or 123, and cannot have a rotation about about an axis twice in a row.

```
>>> B = N.orient_new_body('B', q1, q2, q3, '123')
>>> B = N.orient_new_body('B', q1, q2, 0, 'ZXZ')
>>> B = N.orient_new_body('B', 0, 0, 0, 'XYX')
```

orient_new_quaternion(*name, q0, q1, q2, q3, location=None, vector_names=None, variable_names=None*)

Quaternion orientation orients the new CoordSys3D with Quaternions, defined as a finite rotation about lambda, a unit vector, by some amount theta.

This orientation is described by four parameters:

$q0 = \cos(\theta/2)$

$q1 = \lambda_x \sin(\theta/2)$

$q2 = \lambda_y \sin(\theta/2)$

$q3 = \lambda_z \sin(\theta/2)$

Quaternion does not take in a rotation order.

Parameters

name : string

The name of the new coordinate system

q0, q1, q2, q3 : Expr

The quaternions to rotate the coordinate system by

location : Vector(optional)

The location of the new coordinate system’s origin wrt this system’s origin. If not specified, the origins are taken to be coincident.

vector_names, variable_names : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q0, q1, q2, q3 = symbols('q0 q1 q2 q3')
>>> N = CoordSys3D('N')
>>> B = N.orient_new_quaternion('B', q0, q1, q2, q3)
```

orient_new_space(*name, angle1, angle2, angle3, rotation_order, location=None, vector_names=None, variable_names=None*)

Space rotation is similar to Body rotation, but the rotations are applied in the opposite order.

Parameters

name : string

The name of the new coordinate system

angle1, angle2, angle3 : Expr

Three successive angles to rotate the coordinate system by

rotation_order : string

String defining the order of axes for rotation

location : Vector(optional)

The location of the new coordinate system's origin wrt this system's origin. If not specified, the origins are taken to be coincident.

vector_names, variable_names : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q1, q2, q3 = symbols('q1 q2 q3')
>>> N = CoordSys3D('N')
```

To orient a coordinate system D with respect to N, each sequential rotation is always about N's orthogonal unit vectors. For example, a '123' rotation will specify rotations about N.i, then N.j, then N.k. Therefore,

```
>>> D = N.orient_new_space('D', q1, q2, q3, '312')
```

is same as

```
>>> B = N.orient_new_axis('B', q1, N.i)
>>> C = B.orient_new_axis('C', q2, N.j)
>>> D = C.orient_new_axis('D', q3, N.k)
```

See also:

[*CoordSys3D.orient_new_body*](#) (page 1451)

method to orient via Euler angles

position_wrt(*other*)

Returns the position vector of the origin of this coordinate system with respect to another Point/CoordSys3D.

Parameters

other : Point/CoordSys3D

If other is a Point, the position of this system's origin wrt it is returned. If its an instance of CoordSysRect, the position wrt its origin is returned.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
>>> N1 = N.locate_new('N1', 10 * N.i)
>>> N.position_wrt(N1)
(-10)*N.i
```

rotation_matrix(*other*)

Returns the direction cosine matrix(DCM), also known as the 'rotation matrix' of this coordinate system with respect to another system.

If v_a is a vector defined in system 'A' (in matrix format) and v_b is the same vector defined in system 'B', then $v_a = A.rotation\_matrix(B) * v_b$.

A SymPy Matrix is returned.

Parameters

other : CoordSys3D

The system which the DCM is generated to.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q1 = symbols('q1')
>>> N = CoordSys3D('N')
>>> A = N.orient_new_axis('A', q1, N.i)
>>> N.rotation_matrix(A)
Matrix([
[1,      0,      0],
```

(continues on next page)

(continued from previous page)

```
[0, cos(q1), -sin(q1)],
[0, sin(q1), cos(q1)])]
```

scalar_map(*other*)

Returns a dictionary which expresses the coordinate variables (base scalars) of this frame in terms of the variables of otherframe.

Parameters**otherframe** : CoordSys3D

The other system to map the variables to.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import Symbol
>>> A = CoordSys3D('A')
>>> q = Symbol('q')
>>> B = A.orient_new_axis('B', q, A.k)
>>> A.scalar_map(B)
{A.x: B.x*cos(q) - B.y*sin(q), A.y: B.x*sin(q) + B.y*cos(q), A.z: B.z}
```

Vector**class** sympy.vector.vector.**Vector**(*args)

Super class for all Vector classes. Ideally, neither this class nor any of its subclasses should be instantiated by the user.

property components

Returns the components of this vector in the form of a Python dictionary mapping BaseVector instances to the corresponding measure numbers.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> C = CoordSys3D('C')
>>> v = 3*C.i + 4*C.j + 5*C.k
>>> v.components
{C.i: 3, C.j: 4, C.k: 5}
```

cross(*other*)

Returns the cross product of this Vector with another Vector or Dyadic instance. The cross product is a Vector, if 'other' is a Vector. If 'other' is a Dyadic, this returns a Dyadic instance.

Parameters**other: Vector/Dyadic**

The Vector or Dyadic we are crossing with.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> C = CoordSys3D('C')
>>> C.i.cross(C.j)
C.k
>>> C.i ^ C.i
0
>>> v = 3*C.i + 4*C.j + 5*C.k
>>> v ^ C.i
5*C.j + (-4)*C.k
>>> d = C.i.outer(C.i)
>>> C.j.cross(d)
(-1)*(C.k|C.i)
```

dot(*other*)

Returns the dot product of this Vector, either with another Vector, or a Dyadic, or a Del operator. If 'other' is a Vector, returns the dot product scalar (SymPy expression). If 'other' is a Dyadic, the dot product is returned as a Vector. If 'other' is an instance of Del, returns the directional derivative operator as a Python function. If this function is applied to a scalar expression, it returns the directional derivative of the scalar field wrt this Vector.

Parameters

other: Vector/Dyadic/Del

The Vector or Dyadic we are dotting with, or a Del operator .

Examples

```
>>> from sympy.vector import CoordSys3D, Del
>>> C = CoordSys3D('C')
>>> delop = Del()
>>> C.i.dot(C.j)
0
>>> C.i & C.i
1
>>> v = 3*C.i + 4*C.j + 5*C.k
>>> v.dot(C.k)
5
>>> (C.i & delop)(C.x*C.y*C.z)
C.y*C.z
>>> d = C.i.outer(C.i)
>>> C.i.dot(d)
C.i
```

magnitude()

Returns the magnitude of this vector.

normalize()

Returns the normalized version of this vector.

outer(*other*)

Returns the outer product of this vector with another, in the form of a Dyadic instance.

Parameters

other : Vector

The Vector with respect to which the outer product is to be computed.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
>>> N.i.outer(N.j)
(N.i|N.j)
```

projection(*other*, *scalar=False*)

Returns the vector or scalar projection of the 'other' on 'self'.

Examples

```
>>> from sympy.vector.coordsysrect import CoordSys3D
>>> C = CoordSys3D('C')
>>> i, j, k = C.base_vectors()
>>> v1 = i + j + k
>>> v2 = 3*i + 4*j
>>> v1.projection(v2)
7/3*C.i + 7/3*C.j + 7/3*C.k
>>> v1.projection(v2, scalar=True)
7/3
```

separate()

The constituents of this vector in different coordinate systems, as per its definition.

Returns a dict mapping each CoordSys3D to the corresponding constituent Vector.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> R1 = CoordSys3D('R1')
>>> R2 = CoordSys3D('R2')
>>> v = R1.i + R2.i
>>> v.separate() == {R1: R1.i, R2: R2.i}
True
```

to_matrix(*system*)

Returns the matrix form of this vector with respect to the specified coordinate system.

Parameters

system : CoordSys3D

The system wrt which the matrix form is to be computed

Examples

```
>>> from sympy.vector import CoordSys3D
>>> C = CoordSys3D('C')
>>> from sympy.abc import a, b, c
>>> v = a*C.i + b*C.j + c*C.k
>>> v.to_matrix(C)
Matrix([
[a],
[b],
[c]])
```

Dyadic

class sympy.vector.dyadic.Dyadic(*args)

Super class for all Dyadic-classes.

References

[R978], [R979]

property components

Returns the components of this dyadic in the form of a Python dictionary mapping BaseDyadic instances to the corresponding measure numbers.

cross(other)

Returns the cross product between this Dyadic, and a Vector, as a Vector instance.

Parameters

other : Vector

The Vector that we are crossing this Dyadic with

Examples

```
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
>>> d = N.i.outer(N.i)
>>> d.cross(N.j)
(N.i|N.k)
```

dot(other)

Returns the dot product(also called inner product) of this Dyadic, with another Dyadic or Vector. If 'other' is a Dyadic, this returns a Dyadic. Else, it returns a Vector (unless an error is encountered).

Parameters

other : Dyadic/Vector

The other Dyadic or Vector to take the inner product with

Examples

```
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
>>> D1 = N.i.outer(N.j)
>>> D2 = N.j.outer(N.j)
>>> D1.dot(D2)
(N.i|N.j)
>>> D1.dot(N.j)
N.i
```

to_matrix(system, second_system=None)

Returns the matrix form of the dyadic with respect to one or two coordinate systems.

Parameters

system : CoordSys3D

The coordinate system that the rows and columns of the matrix correspond to. If a second system is provided, this only corresponds to the rows of the matrix.

second_system : CoordSys3D, optional, default=None

The coordinate system that the columns of the matrix correspond to.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
>>> v = N.i + 2*N.j
>>> d = v.outer(N.i)
>>> d.to_matrix(N)
Matrix([
[1, 0, 0],
[2, 0, 0],
[0, 0, 0]])
>>> from sympy import Symbol
>>> q = Symbol('q')
>>> P = N.orient_new_axis('P', q, N.k)
>>> d.to_matrix(N, P)
Matrix([
[ cos(q),  -sin(q),  0],
[2*cos(q), -2*sin(q), 0],
[      0,      0, 0]])
```

Del

class sympy.vector.deloperator.Del

Represents the vector differential operator, usually represented in mathematical expressions as the 'nabla' symbol.

cross(*vect*, *doit=False*)

Represents the cross product between this operator and a given vector - equal to the curl of the vector field.

Parameters

vect : Vector

The vector whose curl is to be calculated.

doit : bool

If True, the result is returned after calling `.doit()` on each component.
Else, the returned expression contains Derivative instances

Examples

```
>>> from sympy.vector import CoordSys3D, Del
>>> C = CoordSys3D('C')
>>> delop = Del()
>>> v = C.x*C.y*C.z * (C.i + C.j + C.k)
>>> delop.cross(v, doit = True)
(-C.x*C.y + C.x*C.z)*C.i + (C.x*C.y - C.y*C.z)*C.j +
(-C.x*C.z + C.y*C.z)*C.k
>>> (delop ^ C.i).doit()
0
```

dot(*vect*, *doit=False*)

Represents the dot product between this operator and a given vector - equal to the divergence of the vector field.

Parameters

vect : Vector

The vector whose divergence is to be calculated.

doit : bool

If True, the result is returned after calling `.doit()` on each component.
Else, the returned expression contains Derivative instances

Examples

```
>>> from sympy.vector import CoordSys3D, Del
>>> delop = Del()
>>> C = CoordSys3D('C')
>>> delop.dot(C.x*C.i)
Derivative(C.x, C.x)
>>> v = C.x*C.y*C.z * (C.i + C.j + C.k)
>>> (delop & v).doit()
C.x*C.y + C.x*C.z + C.y*C.z
```

gradient(*scalar_field*, *doit=False*)

Returns the gradient of the given scalar field, as a Vector instance.

Parameters

scalar_field : SymPy expression

The scalar field to calculate the gradient of.

doit : bool

If True, the result is returned after calling `.doit()` on each component.
Else, the returned expression contains Derivative instances

Examples

```
>>> from sympy.vector import CoordSys3D, Del
>>> C = CoordSys3D('C')
>>> delop = Del()
>>> delop.gradient(9)
0
>>> delop(C.x*C.y*C.z).doit()
C.y*C.z*C.i + C.x*C.z*C.j + C.x*C.y*C.k
```

ParametricRegion

class `sympy.vector.parametricregion.ParametricRegion`(*definition*, **bounds*)

Represents a parametric region in space.

Parameters

definition : tuple to define base scalars in terms of parameters.

bounds : Parameter or a tuple of length 3 to define parameter and corresponding lower and upper bound.

Examples

```
>>> from sympy import cos, sin, pi
>>> from sympy.abc import r, theta, t, a, b, x, y
>>> from sympy.vector import ParametricRegion
```

```
>>> ParametricRegion((t, t**2), (t, -1, 2))
ParametricRegion((t, t**2), (t, -1, 2))
>>> ParametricRegion((x, y), (x, 3, 4), (y, 5, 6))
ParametricRegion((x, y), (x, 3, 4), (y, 5, 6))
>>> ParametricRegion((r*cos(theta), r*sin(theta)), (r, -2, 2), (theta, 0,
↳ pi))
ParametricRegion((r*cos(theta), r*sin(theta)), (r, -2, 2), (theta, 0,
↳ pi))
>>> ParametricRegion((a*cos(t), b*sin(t)), t)
ParametricRegion((a*cos(t), b*sin(t)), t)
```

```
>>> circle = ParametricRegion((r*cos(theta), r*sin(theta)), r, (theta, 0,
↳ pi))
>>> circle.parameters
(r, theta)
>>> circle.definition
(r*cos(theta), r*sin(theta))
>>> circle.limits
{theta: (0, pi)}
```

Dimension of a parametric region determines whether a region is a curve, surface or volume region. It does not represent its dimensions in space.

```
>>> circle.dimensions
1
```

ImplicitRegion

class sympy.vector.implicitregion.**ImplicitRegion**(*variables*, *equation*)

Represents an implicit region in space.

Parameters

variables : tuple to map variables in implicit equation to base scalars.

equation : An expression or Eq denoting the implicit equation of the region.

Examples

```
>>> from sympy import Eq
>>> from sympy.abc import x, y, z, t
>>> from sympy.vector import ImplicitRegion
```

```
>>> ImplicitRegion((x, y), x**2 + y**2 - 4)
ImplicitRegion((x, y), x**2 + y**2 - 4)
>>> ImplicitRegion((x, y), Eq(y*x, 1))
ImplicitRegion((x, y), x*y - 1)
```

```
>>> parabola = ImplicitRegion((x, y), y**2 - 4*x)
>>> parabola.degree
2
>>> parabola.equation
-4*x + y**2
>>> parabola.rational_parametrization(t)
(4/t**2, 4/t)
```

```
>>> r = ImplicitRegion((x, y, z), Eq(z, x**2 + y**2))
>>> r.variables
(x, y, z)
>>> r.singular_points()
EmptySet
>>> r.regular_point()
(-10, -10, 200)
```

multiplicity(point)

Returns the multiplicity of a singular point on the region.

A singular point (x,y) of region is said to be of multiplicity m if all the partial derivatives off to order m - 1 vanish there.

Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.vector import ImplicitRegion
>>> I = ImplicitRegion((x, y, z), x**2 + y**3 - z**4)
>>> I.singular_points()
{(0, 0, 0)}
>>> I.multiplicity((0, 0, 0))
2
```

rational_parametrization(parameters=('t', 's'), reg_point=None)

Returns the rational parametrization of implicit region.

Examples

```
>>> from sympy import Eq
>>> from sympy.abc import x, y, z, s, t
>>> from sympy.vector import ImplicitRegion
```

```
>>> parabola = ImplicitRegion((x, y), y**2 - 4*x)
>>> parabola.rational_parametrization()
(4/t**2, 4/t)
```

```
>>> circle = ImplicitRegion((x, y), Eq(x**2 + y**2, 4))
>>> circle.rational_parametrization()
(4*t/(t**2 + 1), 4*t**2/(t**2 + 1) - 2)
```

```
>>> I = ImplicitRegion((x, y), x**3 + x**2 - y**2)
>>> I.rational_parametrization()
(t**2 - 1, t*(t**2 - 1))
```

```
>>> cubic_curve = ImplicitRegion((x, y), x**3 + x**2 - y**2)
>>> cubic_curve.rational_parametrization(parameters=(t))
(t**2 - 1, t*(t**2 - 1))
```

```
>>> sphere = ImplicitRegion((x, y, z), x**2 + y**2 + z**2 - 4)
>>> sphere.rational_parametrization(parameters=(t, s))
(-2 + 4/(s**2 + t**2 + 1), 4*s/(s**2 + t**2 + 1), 4*t/(s**2 + t**2 + 1))
```

For some conics, `regular_points()` is unable to find a point on curve. To calculate the parametric representation in such cases, user need to determine a point on the region and pass it using `reg_point`.

```
>>> c = ImplicitRegion((x, y), (x - 1/2)**2 + (y)**2 - (1/4)**2)
>>> c.rational_parametrization(reg_point=(3/4, 0))
(0.75 - 0.5/(t**2 + 1), -0.5*t/(t**2 + 1))
```

References

- Christoph M. Hoffmann, “Conversion Methods between Parametric and Implicit Curves and Surfaces”, Purdue e-Pubs, 1990. Available: <https://docs.lib.purdue.edu/cgi/viewcontent.cgi?article=1827&context=cstech>

`regular_point()`

Returns a point on the implicit region.

Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.vector import ImplicitRegion
>>> circle = ImplicitRegion((x, y), (x + 2)**2 + (y - 3)**2 - 16)
>>> circle.regular_point()
(-2, -1)
>>> parabola = ImplicitRegion((x, y), x**2 - 4*y)
>>> parabola.regular_point()
(0, 0)
>>> r = ImplicitRegion((x, y, z), (x + y + z)**4)
>>> r.regular_point()
(-10, -10, 20)
```

References

- Erik Hillgarter, “Rational Points on Conics”, Diploma Thesis, RISC-Linz, J. Kepler Universitat Linz, 1996. Available: https://www3.risc.jku.at/publications/download/risc_1355/Rational%20Points%20on%20Conics.pdf

singular_points()

Returns a set of singular points of the region.

The singular points are those points on the region where all partial derivatives vanish.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy.vector import ImplicitRegion
>>> I = ImplicitRegion((x, y), (y-1)**2 - x**3 + 2*x**2 - x)
>>> I.singular_points()
{(1, 1)}
```

ParametricIntegral

class sympy.vector.integrals.ParametricIntegral(*field*, *parametricregion*)

Represents integral of a scalar or vector field over a Parametric Region

Examples

```
>>> from sympy import cos, sin, pi
>>> from sympy.vector import CoordSys3D, ParametricRegion, _
↳ ParametricIntegral
>>> from sympy.abc import r, t, theta, phi
```

```
>>> C = CoordSys3D('C')
>>> curve = ParametricRegion((3*t - 2, t + 1), (t, 1, 2))
>>> ParametricIntegral(C.x, curve)
5*sqrt(10)/2
>>> length = ParametricIntegral(1, curve)
>>> length
sqrt(10)
>>> semisphere = ParametricRegion((2*sin(phi)*cos(theta), _
↳ 2*sin(phi)*sin(theta), 2*cos(phi)), _ (theta, _
↳ 0, 2*pi), (phi, 0, pi/2))
>>> ParametricIntegral(C.z, semisphere)
8*pi
```

```
>>> ParametricIntegral(C.j + C.k, ParametricRegion((r*cos(theta), _
↳ r*sin(theta)), r, theta))
0
```

Orienter classes (docstrings)

Orienter

class sympy.vector.orienters.**Orienter**(*args)

Super-class for all orienter classes.

rotation_matrix()

The rotation matrix corresponding to this orienter instance.

AxisOrienter

class sympy.vector.orienters.**AxisOrienter**(angle, axis)

Class to denote an axis orienter.

__init__(angle, axis)

Axis rotation is a rotation about an arbitrary axis by some angle. The angle is supplied as a SymPy expr scalar, and the axis is supplied as a Vector.

Parameters

angle : Expr

The angle by which the new system is to be rotated

axis : Vector

The axis around which the rotation has to be performed

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q1 = symbols('q1')
>>> N = CoordSys3D('N')
>>> from sympy.vector import AxisOrienter
>>> orienter = AxisOrienter(q1, N.i + 2 * N.j)
>>> B = N.orient_new('B', (orienter, ))
```

rotation_matrix(*system*)

The rotation matrix corresponding to this orienter instance.

Parameters

system : CoordSys3D

The coordinate system wrt which the rotation matrix is to be computed

BodyOrienter

class sympy.vector.orienters.**BodyOrienter**(*angle1, angle2, angle3, rot_order*)

Class to denote a body-orienter.

__init__(*angle1, angle2, angle3, rot_order*)

Body orientation takes this coordinate system through three successive simple rotations.

Body fixed rotations include both Euler Angles and Tait-Bryan Angles, see https://en.wikipedia.org/wiki/Euler_angles.

Parameters

angle1, angle2, angle3 : Expr

Three successive angles to rotate the coordinate system by

rotation_order : string

String defining the order of axes for rotation

Examples

```
>>> from sympy.vector import CoordSys3D, BodyOrienter
>>> from sympy import symbols
>>> q1, q2, q3 = symbols('q1 q2 q3')
>>> N = CoordSys3D('N')
```

A 'Body' fixed rotation is described by three angles and three body-fixed rotation axes. To orient a coordinate system D with respect to N, each sequential rotation is always about the orthogonal unit vectors fixed to D. For example, a '123' rotation will specify rotations about N.i, then D.j, then D.k. (Initially, D.i is same as N.i) Therefore,

```
>>> body_orienter = BodyOrienter(q1, q2, q3, '123')
>>> D = N.orient_new('D', (body_orienter, ))
```

is same as

```
>>> from sympy.vector import AxisOrienter
>>> axis_orienter1 = AxisOrienter(q1, N.i)
>>> D = N.orient_new('D', (axis_orienter1, ))
>>> axis_orienter2 = AxisOrienter(q2, D.j)
>>> D = D.orient_new('D', (axis_orienter2, ))
>>> axis_orienter3 = AxisOrienter(q3, D.k)
>>> D = D.orient_new('D', (axis_orienter3, ))
```

Acceptable rotation orders are of length 3, expressed in XYZ or 123, and cannot have a rotation about about an axis twice in a row.

```
>>> body_orienter1 = BodyOrienter(q1, q2, q3, '123')
>>> body_orienter2 = BodyOrienter(q1, q2, 0, 'ZXZ')
>>> body_orienter3 = BodyOrienter(0, 0, 0, 'XYX')
```

SpaceOrienter

class sympy.vector.orienters.**SpaceOrienter**(*angle1, angle2, angle3, rot_order*)

Class to denote a space-orienter.

__init__(*angle1, angle2, angle3, rot_order*)

Space rotation is similar to Body rotation, but the rotations are applied in the opposite order.

Parameters

angle1, angle2, angle3 : Expr

Three successive angles to rotate the coordinate system by

rotation_order : string

String defining the order of axes for rotation

Examples

```
>>> from sympy.vector import CoordSys3D, SpaceOrienter
>>> from sympy import symbols
>>> q1, q2, q3 = symbols('q1 q2 q3')
>>> N = CoordSys3D('N')
```

To orient a coordinate system D with respect to N, each sequential rotation is always about N's orthogonal unit vectors. For example, a '123' rotation will specify rotations about N.i, then N.j, then N.k. Therefore,

```
>>> space_orienter = SpaceOrienter(q1, q2, q3, '312')
>>> D = N.orient_new('D', (space_orienter, ))
```

is same as

```
>>> from sympy.vector import AxisOrienter
>>> axis_orienter1 = AxisOrienter(q1, N.i)
>>> B = N.orient_new('B', (axis_orienter1, ))
>>> axis_orienter2 = AxisOrienter(q2, N.j)
>>> C = B.orient_new('C', (axis_orienter2, ))
>>> axis_orienter3 = AxisOrienter(q3, N.k)
>>> D = C.orient_new('C', (axis_orienter3, ))
```

See also:

[BodyOrienter](#) (page 1467)

Orienter to orient systems wrt Euler angles.

QuaternionOrienter

class sympy.vector.orienters.**QuaternionOrienter**(*q0, q1, q2, q3*)

Class to denote a quaternion-orienter.

__init__(*angle1, angle2, angle3, rot_order*)

Quaternion orientation orients the new CoordSys3D with Quaternions, defined as a finite rotation about lambda, a unit vector, by some amount theta.

This orientation is described by four parameters:

$q_0 = \cos(\theta/2)$

$q_1 = \lambda_x \sin(\theta/2)$

$q_2 = \lambda_y \sin(\theta/2)$

$q_3 = \lambda_z \sin(\theta/2)$

Quaternion does not take in a rotation order.

Parameters

q0, q1, q2, q3 : Expr

The quaternions to rotate the coordinate system by

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q0, q1, q2, q3 = symbols('q0 q1 q2 q3')
>>> N = CoordSys3D('N')
>>> from sympy.vector import QuaternionOrienter
>>> q_orienter = QuaternionOrienter(q0, q1, q2, q3)
>>> B = N.orient_new('B', (q_orienter, ))
```

Essential Functions in sympy.vector (docstrings)

matrix_to_vector

`sympy.vector.matrix_to_vector(matrix, system)`

Converts a vector in matrix form to a Vector instance.

It is assumed that the elements of the Matrix represent the measure numbers of the components of the vector along basis vectors of 'system'.

Parameters

matrix : SymPy Matrix, Dimensions: (3, 1)

The matrix to be converted to a vector

system : CoordSys3D

The coordinate system the vector is to be defined in

Examples

```
>>> from sympy import ImmutableMatrix as Matrix
>>> m = Matrix([1, 2, 3])
>>> from sympy.vector import CoordSys3D, matrix_to_vector
>>> C = CoordSys3D('C')
>>> v = matrix_to_vector(m, C)
>>> v
C.i + 2*C.j + 3*C.k
>>> v.to_matrix(C) == m
True
```

express

`sympy.vector.express(expr, system, system2=None, variables=False)`

Global function for 'express' functionality.

Re-expresses a Vector, Dyadic or scalar(sympyifiable) in the given coordinate system.

If 'variables' is True, then the coordinate variables (base scalars) of other coordinate systems present in the vector/scalar field or dyadic are also substituted in terms of the base scalars of the given system.

Parameters

expr : Vector/Dyadic/scalar(sympyifiable)

The expression to re-express in CoordSys3D 'system'

system: CoordSys3D

The coordinate system the expr is to be expressed in

system2: CoordSys3D

The other coordinate system required for re-expression (only for a Dyadic Expr)

variables : boolean

Specifies whether to substitute the coordinate variables present in `expr`, in terms of those of parameter system

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import Symbol, cos, sin
>>> N = CoordSys3D('N')
>>> q = Symbol('q')
>>> B = N.orient_new_axis('B', q, N.k)
>>> from sympy.vector import express
>>> express(B.i, N)
(cos(q))*N.i + (sin(q))*N.j
>>> express(N.x, B, variables=True)
B.x*cos(q) - B.y*sin(q)
>>> d = N.i.outer(N.i)
>>> express(d, B, N) == (cos(q))*(B.i|N.i) + (-sin(q))*(B.j|N.i)
True
```

curl

`sympy.vector.curl(vect, doit=True)`

Returns the curl of a vector field computed wrt the base scalars of the given coordinate system.

Parameters

vect : Vector

The vector operand

doit : bool

If True, the result is returned after calling `.doit()` on each component.
Else, the returned expression contains Derivative instances

Examples

```
>>> from sympy.vector import CoordSys3D, curl
>>> R = CoordSys3D('R')
>>> v1 = R.y*R.z*R.i + R.x*R.z*R.j + R.x*R.y*R.k
>>> curl(v1)
0
>>> v2 = R.x*R.y*R.z*R.i
>>> curl(v2)
R.x*R.y*R.j + (-R.x*R.z)*R.k
```

divergence

`sympy.vector.divergence(vect, doit=True)`

Returns the divergence of a vector field computed wrt the base scalars of the given coordinate system.

Parameters

vector : Vector

The vector operand

doit : bool

If True, the result is returned after calling `.doit()` on each component.
Else, the returned expression contains Derivative instances

Examples

```
>>> from sympy.vector import CoordSys3D, divergence
>>> R = CoordSys3D('R')
>>> v1 = R.x*R.y*R.z * (R.i+R.j+R.k)
```

```
>>> divergence(v1)
R.x*R.y + R.x*R.z + R.y*R.z
>>> v2 = 2*R.y*R.z*R.j
>>> divergence(v2)
2*R.z
```

gradient

`sympy.vector.gradient(scalar_field, doit=True)`

Returns the vector gradient of a scalar field computed wrt the base scalars of the given coordinate system.

Parameters

scalar_field : SymPy Expr

The scalar field to compute the gradient of

doit : bool

If True, the result is returned after calling `.doit()` on each component.
Else, the returned expression contains Derivative instances

Examples

```

>>> from sympy.vector import CoordSys3D, gradient
>>> R = CoordSys3D('R')
>>> s1 = R.x*R.y*R.z
>>> gradient(s1)
R.y*R.z*R.i + R.x*R.z*R.j + R.x*R.y*R.k
>>> s2 = 5*R.x**2*R.z
>>> gradient(s2)
10*R.x*R.z*R.i + 5*R.x**2*R.k

```

is_conservative

`sympy.vector.is_conservative(field)`

Checks if a field is conservative.

Parameters

field : Vector

The field to check for conservative property

Examples

```

>>> from sympy.vector import CoordSys3D
>>> from sympy.vector import is_conservative
>>> R = CoordSys3D('R')
>>> is_conservative(R.y*R.z*R.i + R.x*R.z*R.j + R.x*R.y*R.k)
True
>>> is_conservative(R.z*R.j)
False

```

is_solenoidal

`sympy.vector.is_solenoidal(field)`

Checks if a field is solenoidal.

Parameters

field : Vector

The field to check for solenoidal property

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy.vector import is_solenoidal
>>> R = CoordSys3D('R')
>>> is_solenoidal(R.y*R.z*R.i + R.x*R.z*R.j + R.x*R.y*R.k)
True
>>> is_solenoidal(R.y * R.j)
False
```

scalar_potential

`sympy.vector.scalar_potential(field, coord_sys)`

Returns the scalar potential function of a field in a given coordinate system (without the added integration constant).

Parameters

field : Vector

The vector field whose scalar potential function is to be calculated

coord_sys : CoordSys3D

The coordinate system to do the calculation in

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy.vector import scalar_potential, gradient
>>> R = CoordSys3D('R')
>>> scalar_potential(R.k, R) == R.z
True
>>> scalar_field = 2*R.x**2*R.y*R.z
>>> grad_field = gradient(scalar_field)
>>> scalar_potential(grad_field, R)
2*R.x**2*R.y*R.z
```

scalar_potential_difference

`sympy.vector.scalar_potential_difference(field, coord_sys, point1, point2)`

Returns the scalar potential difference between two points in a certain coordinate system, wrt a given field.

If a scalar field is provided, its values at the two points are considered. If a conservative vector field is provided, the values of its scalar potential function at the two points are used.

Returns (potential at point2) - (potential at point1)

The position vectors of the two Points are calculated wrt the origin of the coordinate system provided.

Parameters**field** : Vector/Expr

The field to calculate wrt

coord_sys : CoordSys3D

The coordinate system to do the calculations in

point1 : Point

The initial Point in given coordinate system

position2 : Point

The second Point in the given coordinate system

Examples

```

>>> from sympy.vector import CoordSys3D
>>> from sympy.vector import scalar_potential_difference
>>> R = CoordSys3D('R')
>>> P = R.origin.locate_new('P', R.x*R.i + R.y*R.j + R.z*R.k)
>>> vectfield = 4*R.x*R.y*R.i + 2*R.x**2*R.j
>>> scalar_potential_difference(vectfield, R, R.origin, P)
2*R.x**2*R.y
>>> Q = R.origin.locate_new('Q', 3*R.i + R.j + 2*R.k)
>>> scalar_potential_difference(vectfield, R, P, Q)
-2*R.x**2*R.y + 18

```

vector_integratesympy.vector.integrals.**vector_integrate**(*field*, **region*)

Compute the integral of a vector/scalar field over a a region or a set of parameters.

Examples

```

>>> from sympy.vector import CoordSys3D, ParametricRegion, vector_
    integrate
>>> from sympy.abc import x, y, t
>>> C = CoordSys3D('C')

```

```

>>> region = ParametricRegion((t, t**2), (t, 1, 5))
>>> vector_integrate(C.x*C.i, region)
12

```

Integrals over some objects of geometry module can also be calculated.

```

>>> from sympy.geometry import Point, Circle, Triangle
>>> c = Circle(Point(0, 2), 5)
>>> vector_integrate(C.x**2 + C.y**2, c)
290*pi

```

(continues on next page)

(continued from previous page)

```
>>> triangle = Triangle(Point(-2, 3), Point(2, 3), Point(0, 5))
>>> vector_integrate(3*C.x**2*C.y*C.i + C.j, triangle)
-8
```

Integrals over some simple implicit regions can be computed. But in most cases, it takes too long to compute over them. This is due to the expressions of parametric representation becoming large.

```
>>> from sympy.vector import ImplicitRegion
>>> c2 = ImplicitRegion((x, y), (x - 2)**2 + (y - 1)**2 - 9)
>>> vector_integrate(1, c2)
6*pi
```

Integral of fields with respect to base scalars:

```
>>> vector_integrate(12*C.y**3, (C.y, 1, 3))
240
>>> vector_integrate(C.x**2*C.z, C.x)
C.x**3*C.z/3
>>> vector_integrate(C.x*C.i - C.y*C.k, C.x)
(Integral(C.x, C.x))*C.i + (Integral(-C.y, C.x))*C.k
>>> _.doit()
C.x**2/2*C.i + (-C.x*C.y)*C.k
```

References for Vector

5.8.5 Number Theory

Contents

Ntheory Class Reference

class sympy.ntheory.generate.Sieve

An infinite list of prime numbers, implemented as a dynamically growing sieve of Eratosthenes. When a lookup is requested involving an odd number that has not been sieved, the sieve is automatically extended up to that number.

Examples

```
>>> from sympy import sieve
>>> sieve._reset() # this line for doctest only
>>> 25 in sieve
False
>>> sieve._list
array('l', [2, 3, 5, 7, 11, 13, 17, 19, 23])
```

extend(*n*)

Grow the sieve to cover all primes $\leq n$ (a real number).

Examples

```
>>> from sympy import sieve
>>> sieve._reset() # this line for doctest only
>>> sieve.extend(30)
>>> sieve[10] == 29
True
```

`extend_to_no(i)`

Extend to include the *i*th prime number.

Parameters

i : integer

Examples

```
>>> from sympy import sieve
>>> sieve._reset() # this line for doctest only
>>> sieve.extend_to_no(9)
>>> sieve._list
array('l', [2, 3, 5, 7, 11, 13, 17, 19, 23])
```

Notes

The list is extended by 50% if it is too short, so it is likely that it will be longer than requested.

`mobiusrange(a, b)`

Generate all mobius numbers for the range $[a, b)$.

Parameters

a : integer

First number in range

b : integer

First number outside of range

Examples

```
>>> from sympy import sieve
>>> print([i for i in sieve.mobiusrange(7, 18)])
[-1, 0, 0, 1, -1, 0, -1, 1, 1, 0, -1]
```

`primerange(a, b=None)`

Generate all prime numbers in the range $[2, a)$ or $[a, b)$.

Examples

```
>>> from sympy import sieve, prime
```

All primes less than 19:

```
>>> print([i for i in sieve.primerange(19)])  
[2, 3, 5, 7, 11, 13, 17]
```

All primes greater than or equal to 7 and less than 19:

```
>>> print([i for i in sieve.primerange(7, 19)])  
[7, 11, 13, 17]
```

All primes through the 10th prime

```
>>> list(sieve.primerange(prime(10) + 1))  
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

search(*n*)

Return the indices *i*, *j* of the primes that bound *n*.

If *n* is prime then *i* == *j*.

Although *n* can be an expression, if ceiling cannot convert it to an integer then an error will be raised.

Examples

```
>>> from sympy import sieve  
>>> sieve.search(25)  
(9, 10)  
>>> sieve.search(23)  
(9, 9)
```

totientrange(*a*, *b*)

Generate all totient numbers for the range [*a*, *b*).

Examples

```
>>> from sympy import sieve  
>>> print([i for i in sieve.totientrange(7, 18)])  
[6, 4, 6, 4, 10, 4, 12, 6, 8, 8, 16]
```

Ntheory Functions Reference

`sympy.ntheory.generate.prime(nth)`

Return the n th prime, with the primes indexed as $\text{prime}(1) = 2$, $\text{prime}(2) = 3$, etc.... The n th prime is approximately $n \log(n)$.

Logarithmic integral of x is a pretty nice approximation for number of primes $\leq x$, i.e. $\text{li}(x) \sim \pi(x)$ In fact, for the numbers we are concerned about($x < 1e11$), $\text{li}(x) - \pi(x) < 50000$

Also, $\text{li}(x) > \pi(x)$ can be safely assumed for the numbers which can be evaluated by this function.

Here, we find the least integer m such that $\text{li}(m) > n$ using binary search. Now $\pi(m-1) < \text{li}(m-1) \leq n$,

We find $\pi(m - 1)$ using `primepi` function.

Starting from m , we have to find $n - \pi(m-1)$ more primes.

For the inputs this implementation can handle, we will have to test primality for at max about 10^{*5} numbers, to get our answer.

Examples

```
>>> from sympy import prime
>>> prime(10)
29
>>> prime(1)
2
>>> prime(100000)
1299709
```

See also:

[`sympy.ntheory.primetest.isprime`](#) (page 1517)

Test if n is prime

[`primerange`](#) (page 1481)

Generate all primes in a given range

[`primepi`](#) (page 1479)

Return the number of primes less than or equal to n

References

[R599], [R600], [R601]

`sympy.ntheory.generate.primepi(n)`

Represents the prime counting function $\pi(n)$ = the number of prime numbers less than or equal to n .

Algorithm Description:

In sieve method, we remove all multiples of prime p except p itself.

Let $\phi(i, j)$ be the number of integers $2 \leq k \leq i$ which remain after sieving from primes less than or equal to j . Clearly, $\pi(n) = \phi(n, \sqrt{n})$

If j is not a prime, $\phi(i, j) = \phi(i, j - 1)$

if j is a prime, We remove all numbers(except j) whose smallest prime factor is j .

Let $x = j \times a$ be such a number, where $2 \leq a \leq i/j$ Now, after sieving from primes $\leq j - 1$, a must remain (because x , and hence a has no prime factor $\leq j - 1$) Clearly, there are $\phi(i/j, j - 1)$ such a which remain on sieving from primes $\leq j - 1$

Now, if a is a prime less than equal to $j - 1$, $x = j \times a$ has smallest prime factor = a , and has already been removed(by sieving from a). So, we do not need to remove it again. (Note: there will be $\pi(j - 1)$ such x)

Thus, number of x , that will be removed are: $\phi(i/j, j - 1) - \phi(j - 1, j - 1)$ (Note that $\pi(j - 1) = \phi(j - 1, j - 1)$)

$\Rightarrow \phi(i, j) = \phi(i, j - 1) - \phi(i/j, j - 1) + \phi(j - 1, j - 1)$

So, following recursion is used and implemented as dp:

$\phi(a, b) = \phi(a, b - 1)$, if b is not a prime
 $\phi(a, b) = \phi(a, b - 1) - \phi(a/b, b - 1) + \phi(b - 1, b - 1)$, if b is prime

Clearly a is always of the form $\text{floor}(n/k)$, which can take at most $2\sqrt{n}$ values. Two arrays `arr1`, `arr2` are maintained `arr1[i] =  $\phi(i, j)$` , `arr2[i] =  $\phi(n // i, j)$`

Finally the answer is `arr2[1]`

Examples

```
>>> from sympy import primepi, prime, prevprime, isprime
>>> primepi(25)
9
```

So there are 9 primes less than or equal to 25. Is 25 prime?

```
>>> isprime(25)
False
```

It is not. So the first prime less than 25 must be the 9th prime:

```
>>> prevprime(25) == prime(9)
True
```

See also:

[`sympy.ntheory.primetest.isprime`](#) (page 1517)

Test if n is prime

[`primerange`](#) (page 1481)

Generate all primes in a given range

[`prime`](#) (page 1479)

Return the n th prime

`sympy.ntheory.generate.nextprime(n, ith=1)`

Return the *ith* prime greater than *n*.

i must be an integer.

Notes

Potential primes are located at $6*j \pm 1$. This property is used during searching.

```
>>> from sympy import nextprime
>>> [(i, nextprime(i)) for i in range(10, 15)]
[(10, 11), (11, 13), (12, 13), (13, 17), (14, 17)]
>>> nextprime(2, ith=2) # the 2nd prime after 2
5
```

See also:

[*prevprime* \(page 1481\)](#)

Return the largest prime smaller than *n*

[*primerange* \(page 1481\)](#)

Generate all primes in a given range

`sympy.ntheory.generate.prevprime(n)`

Return the largest prime smaller than *n*.

Notes

Potential primes are located at $6*j \pm 1$. This property is used during searching.

```
>>> from sympy import prevprime
>>> [(i, prevprime(i)) for i in range(10, 15)]
[(10, 7), (11, 7), (12, 11), (13, 11), (14, 13)]
```

See also:

[*nextprime* \(page 1480\)](#)

Return the *ith* prime greater than *n*

[*primerange* \(page 1481\)](#)

Generates all primes in a given range

`sympy.ntheory.generate.primerange(a, b=None)`

Generate a list of all prime numbers in the range $[2, a)$, or $[a, b)$.

If the range exists in the default sieve, the values will be returned from there; otherwise values will be returned but will not modify the sieve.

Examples

```
>>> from sympy import primerange, prime
```

All primes less than 19:

```
>>> list(primerange(19))
[2, 3, 5, 7, 11, 13, 17]
```

All primes greater than or equal to 7 and less than 19:

```
>>> list(primerange(7, 19))
[7, 11, 13, 17]
```

All primes through the 10th prime

```
>>> list(primerange(prime(10) + 1))
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

The Sieve method, `primerange`, is generally faster but it will occupy more memory as the sieve stores values. The default instance of Sieve, named `sieve`, can be used:

```
>>> from sympy import sieve
>>> list(sieve.primerange(1, 30))
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Notes

Some famous conjectures about the occurrence of primes in a given range are [1]:

- **Twin primes: though often not, the following will give 2 primes**
an infinite number of times:
`primerange(6*n - 1, 6*n + 2)`
- **Legendre's: the following always yields at least one prime**
`primerange(n**2, (n+1)**2+1)`
- **Bertrand's (proven): there is always a prime in the range**
`primerange(n, 2*n)`
- **Brocard's: there are at least four primes in the range**
`primerange(prime(n)**2, prime(n+1)**2)`

The average gap between primes is $\log(n)$ [2]; the gap between primes can be arbitrarily large since sequences of composite numbers are arbitrarily large, e.g. the numbers in the sequence $n! + 2, n! + 3 \dots n! + n$ are all composite.

See also:

[*prime*](#) (page 1479)

Return the n th prime

[*nextprime*](#) (page 1480)

Return the i th prime greater than n

***prevprime* (page 1481)**

Return the largest prime smaller than n

***randprime* (page 1483)**

Returns a random prime in a given range

***primorial* (page 1483)**

Returns the product of primes based on condition

***Sieve.primerange* (page 1477)**

return range from already computed primes or extend the sieve to contain the requested range.

References

[R602], [R603]

`sympy.ntheory.generate.randprime(a, b)`

Return a random prime number in the range $[a, b)$.

Bertrand's postulate assures that `randprime(a, 2*a)` will always succeed for $a > 1$.

Examples

```
>>> from sympy import randprime, isprime
>>> randprime(1, 30)
13
>>> isprime(randprime(1, 30))
True
```

See also:***primerange* (page 1481)**

Generate all primes in a given range

References

[R604]

`sympy.ntheory.generate.primorial(n, nth=True)`

Returns the product of the first n primes (default) or the primes less than or equal to n (when `nth=False`).

Examples

```
>>> from sympy.ntheory.generate import primorial, primerange
>>> from sympy import factorint, Mul, primefactors, sqrt
>>> primorial(4) # the first 4 primes are 2, 3, 5, 7
210
>>> primorial(4, nth=False) # primes <= 4 are 2 and 3
6
>>> primorial(1)
2
>>> primorial(1, nth=False)
1
>>> primorial(sqrt(101), nth=False)
210
```

One can argue that the primes are infinite since if you take a set of primes and multiply them together (e.g. the primorial) and then add or subtract 1, the result cannot be divided by any of the original factors, hence either 1 or more new primes must divide this product of primes.

In this case, the number itself is a new prime:

```
>>> factorint(primorial(4) + 1)
{211: 1}
```

In this case two new primes are the factors:

```
>>> factorint(primorial(4) - 1)
{11: 1, 19: 1}
```

Here, some primes smaller and larger than the primes multiplied together are obtained:

```
>>> p = list(primerange(10, 20))
>>> sorted(set(primefactors(Mul(*p) + 1)).difference(set(p)))
[2, 5, 31, 149]
```

See also:

[primerange](#) (page 1481)

Generate all primes in a given range

`sympy.ntheory.generate.cycle_length(f, x0, nmax=None, values=False)`

For a given iterated sequence, return a generator that gives the length of the iterated cycle (lambda) and the length of terms before the cycle begins (mu); if `values` is `True` then the terms of the sequence will be returned instead. The sequence is started with value `x0`.

Note: more than the first `lambda + mu` terms may be returned and this is the cost of cycle detection with Brent's method; there are, however, generally less terms calculated than would have been calculated if the proper ending point were determined, e.g. by using Floyd's method.

```
>>> from sympy.ntheory.generate import cycle_length
```

This will yield successive values of `i <- func(i)`:

```
>>> def iter(func, i):
...     while 1:
...         ii = func(i)
...         yield ii
...         i = ii
... 
```

A function is defined:

```
>>> func = lambda i: (i**2 + 1) % 51
```

and given a seed of 4 and the mu and lambda terms calculated:

```
>>> next(cycle_length(func, 4))
(6, 2)
```

We can see what is meant by looking at the output:

```
>>> n = cycle_length(func, 4, values=True)
>>> list(ni for ni in n)
[17, 35, 2, 5, 26, 14, 44, 50, 2, 5, 26, 14]
```

There are 6 repeating values after the first 2.

If a sequence is suspected of being longer than you might wish, `nmax` can be used to exit early (and `mu` will be returned as `None`):

```
>>> next(cycle_length(func, 4, nmax = 4))
(4, None)
>>> [ni for ni in cycle_length(func, 4, nmax = 4, values=True)]
[17, 35, 2, 5]
```

Code modified from:

https://en.wikipedia.org/wiki/Cycle_detection.

`sympy.ntheory.generate.composite(nth)`

Return the `nth` composite number, with the composite numbers indexed as `composite(1) = 4`, `composite(2) = 6`, etc....

Examples

```
>>> from sympy import composite
>>> composite(36)
52
>>> composite(1)
4
>>> composite(17737)
20000
```

See also:

[`sympy.ntheory.primetest.isprime`](#) (page 1517)

Test if `n` is prime

***primerange* (page 1481)**

Generate all primes in a given range

***primepi* (page 1479)**

Return the number of primes less than or equal to n

***prime* (page 1479)**

Return the n th prime

***compositepi* (page 1486)**

Return the number of positive composite numbers less than or equal to n

`sympy.ntheory.generate.compositepi( $n$ )`

Return the number of positive composite numbers less than or equal to n . The first positive composite is 4, i.e. `compositepi(4) = 1`.

Examples

```
>>> from sympy import compositepi
>>> compositepi(25)
15
>>> compositepi(1000)
831
```

See also:***sympy.ntheory.primetest.isprime* (page 1517)**

Test if n is prime

***primerange* (page 1481)**

Generate all primes in a given range

***prime* (page 1479)**

Return the n th prime

***primepi* (page 1479)**

Return the number of primes less than or equal to n

***composite* (page 1485)**

Return the n th composite number

`sympy.ntheory.factor_.smoothness( $n$ )`

Return the B-smooth and B-power smooth values of n .

The smoothness of n is the largest prime factor of n ; the power-smoothness is the largest divisor raised to its multiplicity.

Examples

```
>>> from sympy.ntheory.factor_ import smoothness
>>> smoothness(2**7*3**2)
(3, 128)
>>> smoothness(2**4*13)
(13, 16)
>>> smoothness(2)
(2, 2)
```

See also:

[`factorint`](#) (page 1493), [`smoothness\_p`](#) (page 1487)

`sympy.ntheory.factor_.smoothness_p(n, m=-1, power=0, visual=None)`

Return a list of `[m, (p, (M, sm(p + m), psm(p + m)))]` where:

1. p^*M is the base- p divisor of n
2. `sm(p + m)` is the smoothness of $p + m$ ($m = -1$ by default)
3. `psm(p + m)` is the power smoothness of $p + m$

The list is sorted according to smoothness (default) or by power smoothness if `power=1`.

The smoothness of the numbers to the left ($m = -1$) or right ($m = 1$) of a factor govern the results that are obtained from the $p \pm 1$ type factoring methods.

```
>>> from sympy.ntheory.factor_ import smoothness_p, factorint
>>> smoothness_p(10431, m=1)
(1, [(3, (2, 2, 4)), (19, (1, 5, 5)), (61, (1, 31, 31))])
>>> smoothness_p(10431)
(-1, [(3, (2, 2, 2)), (19, (1, 3, 9)), (61, (1, 5, 5))])
>>> smoothness_p(10431, power=1)
(-1, [(3, (2, 2, 2)), (61, (1, 5, 5)), (19, (1, 3, 9))])
```

If `visual=True` then an annotated string will be returned:

```
>>> print(smoothness_p(21477639576571, visual=1))
p**i=4410317**1 has p-1 B=1787, B-pow=1787
p**i=4869863**1 has p-1 B=2434931, B-pow=2434931
```

This string can also be generated directly from a factorization dictionary and vice versa:

```
>>> factorint(17*9)
{3: 2, 17: 1}
>>> smoothness_p(_)
'p**i=3**2 has p-1 B=2, B-pow=2\np**i=17**1 has p-1 B=2, B-pow=16'
>>> smoothness_p(_)
{3: 2, 17: 1}
```

The table of the output logic is:

	Visual		
Input	True	False	other
dict	str	tuple	str
str	str	tuple	dict
tuple	str	tuple	str
n	str	tuple	tuple
mul	str	tuple	tuple

See also:

[*factorint*](#) (page 1493), [*smoothness*](#) (page 1486)

`sympy.ntheory.factor_.trailing(n)`

Count the number of trailing zero digits in the binary representation of *n*, i.e. determine the largest power of 2 that divides *n*.

Examples

```
>>> from sympy import trailing
>>> trailing(128)
7
>>> trailing(63)
0
```

`sympy.ntheory.factor_.multiplicity(p, n)`

Find the greatest integer *m* such that p^m divides *n*.

Examples

```
>>> from sympy import multiplicity, Rational
>>> [multiplicity(5, n) for n in [8, 5, 25, 125, 250]]
[0, 1, 2, 3, 3]
>>> multiplicity(3, Rational(1, 9))
-2
```

Note: when checking for the multiplicity of a number in a large factorial it is most efficient to send it as an unevaluated factorial or to call `multiplicity_in_factorial` directly:

```
>>> from sympy.ntheory import multiplicity_in_factorial
>>> from sympy import factorial
>>> p = factorial(25)
>>> n = 2**100
>>> nfac = factorial(n, evaluate=False)
>>> multiplicity(p, nfac)
52818775009509558395695966887
>>> _ == multiplicity_in_factorial(p, n)
True
```


`sympy.ntheory.factor_.perfect_power(n, candidates=None, big=True, factor=True)`

Return (b, e) such that $n == b**e$ if n is a unique perfect power with $e > 1$, else False (e.g. 1 is not a perfect power). A ValueError is raised if n is not Rational.

By default, the base is recursively decomposed and the exponents collected so the largest possible e is sought. If `big=False` then the smallest possible e (thus prime) will be chosen.

If `factor=True` then simultaneous factorization of n is attempted since finding a factor indicates the only possible root for n. This is True by default since only a few small factors will be tested in the course of searching for the perfect power.

The use of `candidates` is primarily for internal use; if provided, False will be returned if n cannot be written as a power with one of the candidates as an exponent and factoring (beyond testing for a factor of 2) will not be attempted.

Examples

```
>>> from sympy import perfect_power, Rational
>>> perfect_power(16)
(2, 4)
>>> perfect_power(16, big=False)
(4, 2)
```

Negative numbers can only have odd perfect powers:

```
>>> perfect_power(-4)
False
>>> perfect_power(-8)
(-2, 3)
```

Rationals are also recognized:

```
>>> perfect_power(Rational(1, 2)**3)
(1/2, 3)
>>> perfect_power(Rational(-3, 2)**3)
(-3/2, 3)
```

Notes

To know whether an integer is a perfect power of 2 use

```
>>> is2pow = lambda n: bool(n and not n & (n - 1))
>>> [(i, is2pow(i)) for i in range(5)]
[(0, False), (1, True), (2, True), (3, False), (4, True)]
```

It is not necessary to provide `candidates`. When provided it will be assumed that they are ints. The first one that is larger than the computed maximum possible exponent will signal failure for the routine.

```
>>> perfect_power(3**8, [9])
False
```

(continues on next page)

(continued from previous page)

```
>>> perfect_power(3**8, [2, 4, 8])
(3, 8)
>>> perfect_power(3**8, [4, 8], big=False)
(9, 4)
```

See also:

[`sympy.core.power.integer\_nthroot`](#) (page 1008), [`sympy.ntheory.primetest.is\_square`](#) (page 1514)

```
sympy.ntheory.factor_.pollard_rho(n, s=2, a=1, retries=5, seed=1234,
                                  max_steps=None, F=None)
```

Use Pollard's rho method to try to extract a nontrivial factor of n . The returned factor may be a composite number. If no factor is found, `None` is returned.

The algorithm generates pseudo-random values of x with a generator function, replacing x with $F(x)$. If F is not supplied then the function $x^2 + a$ is used. The first value supplied to $F(x)$ is s . Upon failure (if `retries` is > 0) a new a and s will be supplied; the a will be ignored if F was supplied.

The sequence of numbers generated by such functions generally have a lead-up to some number and then loop around back to that number and begin to repeat the sequence, e.g. 1, 2, 3, 4, 5, 3, 4, 5 - this leader and loop look a bit like the Greek letter rho, and thus the name, 'rho'.

For a given function, very different leader-loop values can be obtained so it is a good idea to allow for retries:

```
>>> from sympy.ntheory.generate import cycle_length
>>> n = 16843009
>>> F = lambda x: (2048*pow(x, 2, n) + 32767) % n
>>> for s in range(5):
...     print('loop length = %4i; leader length = %3i' % next(cycle_
...     length(F, s)))
...
loop length = 2489; leader length = 42
loop length = 78; leader length = 120
loop length = 1482; leader length = 99
loop length = 1482; leader length = 285
loop length = 1482; leader length = 100
```

Here is an explicit example where there is a two element leadup to a sequence of 3 numbers (11, 14, 4) that then repeat:

```
>>> x=2
>>> for i in range(9):
...     x=(x**2+12)%17
...     print(x)
...
16
13
11
14
4
```

(continues on next page)

(continued from previous page)

```

11
14
4
11
>>> next(cycle_length(lambda x: (x**2+12)%17, 2))
(3, 2)
>>> list(cycle_length(lambda x: (x**2+12)%17, 2, values=True))
[16, 13, 11, 14, 4]

```

Instead of checking the differences of all generated values for a gcd with n , only the k th and $2*k$ th numbers are checked, e.g. 1st and 2nd, 2nd and 4th, 3rd and 6th until it has been detected that the loop has been traversed. Loops may be many thousands of steps long before rho finds a factor or reports failure. If `max_steps` is specified, the iteration is cancelled with a failure after the specified number of steps.

Examples

```

>>> from sympy import pollard_rho
>>> n=16843009
>>> F=lambda x:(2048*pow(x,2,n) + 32767) % n
>>> pollard_rho(n, F=F)
257

```

Use the default setting with a bad value of a and no retries:

```
>>> pollard_rho(n, a=n-2, retries=0)
```

If `retries` is > 0 then perhaps the problem will correct itself when new values are generated for a :

```

>>> pollard_rho(n, a=n-2, retries=1)
257

```

References

[R605]

`sympy.ntheory.factor_.pollard_pm1( $n$ ,  $B=10$ ,  $a=2$ ,  $retries=0$ ,  $seed=1234$ )`

Use Pollard's $p-1$ method to try to extract a nontrivial factor of n . Either a divisor (perhaps composite) or `None` is returned.

The value of a is the base that is used in the test $\gcd(a^{*M} - 1, n)$. The default is 2. If `retries` > 0 then if no factor is found after the first attempt, a new a will be generated randomly (using the `seed`) and the process repeated.

Note: the value of M is $\text{lcm}(1..B) = \text{reduce}(\text{ilcm}, \text{range}(2, B + 1))$.

A search is made for factors next to even numbers having a power smoothness less than B . Choosing a larger B increases the likelihood of finding a larger factor but takes longer. Whether a factor of n is found or not depends on a and the power smoothness of the even number just less than the factor p (hence the name $p - 1$).

Although some discussion of what constitutes a good a some descriptions are hard to interpret. At the modular.math site referenced below it is stated that if $\gcd(a^{**M} - 1, n) = N$ then $a^{**M} \% q^{**r}$ is 1 for every prime power divisor of N. But consider the following:

```
>>> from sympy.ntheory.factor_ import smoothness_p, pollard_pm1
>>> n=257*1009
>>> smoothness_p(n)
(-1, [(257, (1, 2, 256)), (1009, (1, 7, 16))])
```

So we should (and can) find a root with B=16:

```
>>> pollard_pm1(n, B=16, a=3)
1009
```

If we attempt to increase B to 256 we find that it does not work:

```
>>> pollard_pm1(n, B=256)
>>>
```

But if the value of a is changed we find that only multiples of 257 work, e.g.:

```
>>> pollard_pm1(n, B=256, a=257)
1009
```

Checking different a values shows that all the ones that did not work had a gcd value not equal to n but equal to one of the factors:

```
>>> from sympy import ilcm, igcd, factorint, Pow
>>> M = 1
>>> for i in range(2, 256):
...     M = ilcm(M, i)
...
>>> set([igcd(pow(a, M, n) - 1, n) for a in range(2, 256) if
...       igcd(pow(a, M, n) - 1, n) != n])
{1009}
```

But does $aM \% d$ for every divisor of n give 1?

```
>>> aM = pow(255, M, n)
>>> [(d, aM%Pow(*d.args)) for d in factorint(n, visual=True).args]
[(257**1, 1), (1009**1, 1)]
```

No, only one of them. So perhaps the principle is that a root will be found for a given value of B provided that:

- 1) the power smoothness of the $p - 1$ value next to the root does not exceed B
- 2) $a^{**M} \% p \neq 1$ for any of the divisors of n.

By trying more than one a it is possible that one of them will yield a factor.

Examples

With the default smoothness bound, this number cannot be cracked:

```
>>> from sympy.ntheory import pollard_pm1
>>> pollard_pm1(21477639576571)
```

Increasing the smoothness bound helps:

```
>>> pollard_pm1(21477639576571, B=2000)
4410317
```

Looking at the smoothness of the factors of this number we find:

```
>>> from sympy.ntheory.factor_ import smoothness_p, factorint
>>> print(smoothness_p(21477639576571, visual=1))
p**i=4410317**1 has p-1 B=1787, B-pow=1787
p**i=4869863**1 has p-1 B=2434931, B-pow=2434931
```

The B and B-pow are the same for the $p - 1$ factorizations of the divisors because those factorizations had a very large prime factor:

```
>>> factorint(4410317 - 1)
{2: 2, 617: 1, 1787: 1}
>>> factorint(4869863-1)
{2: 1, 2434931: 1}
```

Note that until B reaches the B-pow value of 1787, the number is not cracked;

```
>>> pollard_pm1(21477639576571, B=1786)
>>> pollard_pm1(21477639576571, B=1787)
4410317
```

The B value has to do with the factors of the number next to the divisor, not the divisors themselves. A worst case scenario is that the number next to the factor p has a large prime divisor or is a perfect power. If these conditions apply then the power-smoothness will be about $p/2$ or p . The more realistic is that there will be a large prime factor next to p requiring a B value on the order of $p/2$. Although primes may have been searched for up to this level, the $p/2$ is a factor of $p - 1$, something that we do not know. The modular.math reference below states that 15% of numbers in the range of 10^{15} to $15^{15} + 10^4$ are 10^6 power smooth so a B of 10^6 will fail 85% of the time in that range. From 10^8 to $10^8 + 10^3$ the percentages are nearly reversed...but in that range the simple trial division is quite fast.

References

[R606], [R607], [R608]

`sympy.ntheory.factor_.factorint(n, limit=None, use_trial=True, use_rho=True, use_pm1=True, use_ecm=True, verbose=False, visual=None, multiple=False)`

Given a positive integer n , `factorint( $n$ )` returns a dict containing the prime factors of n as keys and their respective multiplicities as values. For example:

```
>>> from sympy.ntheory import factorint
>>> factorint(2000)      # 2000 = (2**4) * (5**3)
{2: 4, 5: 3}
>>> factorint(65537)    # This number is prime
{65537: 1}
```

For input less than 2, factorint behaves as follows:

- factorint(1) returns the empty factorization, {}
- factorint(0) returns {0:1}
- factorint(-n) adds -1:1 to the factors and then factors n

Partial Factorization:

If limit (> 3) is specified, the search is stopped after performing trial division up to (and including) the limit (or taking a corresponding number of rho/p-1 steps). This is useful if one has a large number and only is interested in finding small factors (if any). Note that setting a limit does not prevent larger factors from being found early; it simply means that the largest factor may be composite. Since checking for perfect power is relatively cheap, it is done regardless of the limit setting.

This number, for example, has two small factors and a huge semi-prime factor that cannot be reduced easily:

```
>>> from sympy.ntheory import isprime
>>> a = 1407633717262338957430697921446883
>>> f = factorint(a, limit=10000)
>>> f == {991: 1, int(202916782076162456022877024859): 1, 7: 1}
True
>>> isprime(max(f))
False
```

This number has a small factor and a residual perfect power whose base is greater than the limit:

```
>>> factorint(3*101**7, limit=5)
{3: 1, 101: 7}
```

List of Factors:

If multiple is set to True then a list containing the prime factors including multiplicities is returned.

```
>>> factorint(24, multiple=True)
[2, 2, 2, 3]
```

Visual Factorization:

If visual is set to True, then it will return a visual factorization of the integer. For example:

```
>>> from sympy import pprint
>>> pprint(factorint(4200, visual=True))
 3  1  2  1
2 * 3 * 5 * 7
```

Note that this is achieved by using the `evaluate=False` flag in `Mul` and `Pow`. If you do other manipulations with an expression where `evaluate=False`, it may evaluate. Therefore, you should use the visual option only for visualization, and use the normal dictionary returned by `visual=False` if you want to perform operations on the factors.

You can easily switch between the two forms by sending them back to `factorint`:

```
>>> from sympy import Mul
>>> regular = factorint(1764); regular
{2: 2, 3: 2, 7: 2}
>>> pprint(factorint(regular))
  2  2  2
2 *3 *7
```

```
>>> visual = factorint(1764, visual=True); pprint(visual)
  2  2  2
2 *3 *7
>>> print(factorint(visual))
{2: 2, 3: 2, 7: 2}
```

If you want to send a number to be factored in a partially factored form you can do so with a dictionary or unevaluated expression:

```
>>> factorint(factorint({4: 2, 12: 3})) # twice to toggle to dict form
{2: 10, 3: 3}
>>> factorint(Mul(4, 12, evaluate=False))
{2: 4, 3: 1}
```

The table of the output logic is:

Input	True	False	other
dict	mul	dict	mul
n	mul	dict	dict
mul	mul	dict	dict

Notes

Algorithm:

The function switches between multiple algorithms. Trial division quickly finds small factors (of the order 1-5 digits), and finds all large factors if given enough time. The Pollard rho and p-1 algorithms are used to find large factors ahead of time; they will often find factors of the order of 10 digits within a few seconds:

```
>>> factors = factorint(12345678910111213141516)
>>> for base, exp in sorted(factors.items()):
...     print('%s %s' % (base, exp))
...
2 2
2507191691 1
1231026625769 1
```

Any of these methods can optionally be disabled with the following boolean parameters:

- `use_trial`: Toggle use of trial division
- `use_rho`: Toggle use of Pollard's rho method
- `use_pm1`: Toggle use of Pollard's p-1 method

`factorint` also periodically checks if the remaining part is a prime number or a perfect power, and in those cases stops.

For unevaluated factorial, it uses Legendre's formula(theorem).

If `verbose` is set to `True`, detailed progress is printed.

See also:

[`smoothness`](#) (page 1486), [`smoothness\_p`](#) (page 1487), [`divisors`](#) (page 1497)

`sympy.ntheory.factor_.factorrat`(*rat*, *limit=None*, *use_trial=True*, *use_rho=True*, *use_pm1=True*, *verbose=False*, *visual=None*, *multiple=False*)

Given a Rational *r*, `factorrat(r)` returns a dict containing the prime factors of *r* as keys and their respective multiplicities as values. For example:

```
>>> from sympy import factorrat, S
>>> factorrat(S(8)/9)      # 8/9 = (2**3) * (3**-2)
{2: 3, 3: -2}
>>> factorrat(S(-1)/987)   # -1/789 = -1 * (3**-1) * (7**-1) * (47**-1)
{-1: 1, 3: -1, 7: -1, 47: -1}
```

Please see the docstring for `factorint` for detailed explanations and examples of the following keywords:

- `limit`: Integer limit up to which trial division is done
- `use_trial`: Toggle use of trial division
- `use_rho`: Toggle use of Pollard's rho method
- `use_pm1`: Toggle use of Pollard's p-1 method
- `verbose`: Toggle detailed printing of progress
- `multiple`: Toggle returning a list of factors or dict
- `visual`: Toggle product form of output

`sympy.ntheory.factor_.primefactors`(*n*, *limit=None*, *verbose=False*)

Return a sorted list of *n*'s prime factors, ignoring multiplicity and any composite factor that remains if the limit was set too low for complete factorization. Unlike `factorint()`, `primefactors()` does not return -1 or 0.

Examples

```
>>> from sympy.ntheory import primefactors, factorint, isprime
>>> primefactors(6)
[2, 3]
>>> primefactors(-5)
[5]
```

```
>>> sorted(factorint(123456).items())
[(2, 6), (3, 1), (643, 1)]
>>> primefactors(123456)
[2, 3, 643]
```

```
>>> sorted(factorint(10000000001, limit=200).items())
[(101, 1), (99009901, 1)]
>>> isprime(99009901)
False
>>> primefactors(10000000001, limit=300)
[101]
```

See also:

[*divisors*](#) (page 1497)

`sympy.ntheory.factor_.divisors(n, generator=False, proper=False)`

Return all divisors of *n* sorted from 1..*n* by default. If *generator* is *True* an unordered generator is returned.

The number of divisors of *n* can be quite large if there are many prime factors (counting repeated factors). If only the number of factors is desired use `divisor_count(n)`.

Examples

```
>>> from sympy import divisors, divisor_count
>>> divisors(24)
[1, 2, 3, 4, 6, 8, 12, 24]
>>> divisor_count(24)
8
```

```
>>> list(divisors(120, generator=True))
[1, 2, 4, 8, 3, 6, 12, 24, 5, 10, 20, 40, 15, 30, 60, 120]
```

Notes

This is a slightly modified version of Tim Peters referenced at: <https://stackoverflow.com/questions/1010381/python-factorization>

See also:

primefactors (page 1496), *factorint* (page 1493), *divisor_count* (page 1498)

`sympy.ntheory.factor_.proper_divisors(n, generator=False)`

Return all divisors of *n* except *n*, sorted by default. If *generator* is *True* an unordered generator is returned.

Examples

```
>>> from sympy import proper_divisors, proper_divisor_count
>>> proper_divisors(24)
[1, 2, 3, 4, 6, 8, 12]
>>> proper_divisor_count(24)
7
>>> list(proper_divisors(120, generator=True))
[1, 2, 4, 8, 3, 6, 12, 24, 5, 10, 20, 40, 15, 30, 60]
```

See also:

factorint (page 1493), *divisors* (page 1497), *proper_divisor_count* (page 1498)

`sympy.ntheory.factor_.divisor_count(n, modulus=1, proper=False)`

Return the number of divisors of *n*. If *modulus* is not 1 then only those that are divisible by *modulus* are counted. If *proper* is *True* then the divisor of *n* will not be counted.

Examples

```
>>> from sympy import divisor_count
>>> divisor_count(6)
4
>>> divisor_count(6, 2)
2
>>> divisor_count(6, proper=True)
3
```

See also:

factorint (page 1493), *divisors* (page 1497), *totient* (page 1501), *proper_divisor_count* (page 1498)

`sympy.ntheory.factor_.proper_divisor_count(n, modulus=1)`

Return the number of proper divisors of *n*.

Examples

```
>>> from sympy import proper_divisor_count
>>> proper_divisor_count(6)
3
>>> proper_divisor_count(6, modulus=2)
1
```

See also:

[*divisors*](#) (page 1497), [*proper_divisors*](#) (page 1498), [*divisor_count*](#) (page 1498)

`sympy.ntheory.factor_.udivisors(n, generator=False)`

Return all unitary divisors of n sorted from 1..n by default. If generator is True an un-ordered generator is returned.

The number of unitary divisors of n can be quite large if there are many prime factors. If only the number of unitary divisors is desired use `udivisor_count(n)`.

Examples

```
>>> from sympy.ntheory.factor_ import udivisors, udivisor_count
>>> udivisors(15)
[1, 3, 5, 15]
>>> udivisor_count(15)
4
```

```
>>> sorted(udivisors(120, generator=True))
[1, 3, 5, 8, 15, 24, 40, 120]
```

See also:

[*primefactors*](#) (page 1496), [*factorint*](#) (page 1493), [*divisors*](#) (page 1497), [*divisor_count*](#) (page 1498), [*udivisor_count*](#) (page 1499)

References

[R609], [R610]

`sympy.ntheory.factor_.udivisor_count(n)`

Return the number of unitary divisors of n.

Parameters

n : integer

Examples

```
>>> from sympy.ntheory.factor_ import udivisor_count
>>> udivisor_count(120)
8
```

See also:

[*factorint*](#) (page 1493), [*divisors*](#) (page 1497), [*udivisors*](#) (page 1499), [*divisor_count*](#) (page 1498), [*totient*](#) (page 1501)

References

[R611]

`sympy.ntheory.factor_.antidivisors(n, generator=False)`

Return all antidivisors of *n* sorted from 1..*n* by default.

Antidivisors [R612] of *n* are numbers that do not divide *n* by the largest possible margin. If *generator* is `True` an unordered generator is returned.

Examples

```
>>> from sympy.ntheory.factor_ import antidivisors
>>> antidivisors(24)
[7, 16]
```

```
>>> sorted(antidivisors(128, generator=True))
[3, 5, 15, 17, 51, 85]
```

See also:

[*primefactors*](#) (page 1496), [*factorint*](#) (page 1493), [*divisors*](#) (page 1497), [*divisor_count*](#) (page 1498), [*antidivisor_count*](#) (page 1500)

References

[R612]

`sympy.ntheory.factor_.antidivisor_count(n)`

Return the number of antidivisors [R613] of *n*.

Parameters

n : integer

Examples

```
>>> from sympy.ntheory.factor_ import antidivisor_count
>>> antidivisor_count(13)
4
>>> antidivisor_count(27)
5
```

See also:

[factorint](#) (page 1493), [divisors](#) (page 1497), [antidivisors](#) (page 1500), [divisor_count](#) (page 1498), [totient](#) (page 1501)

References

[R613]

class sympy.ntheory.factor_.**totient**(*n*)

Calculate the Euler totient function $\phi(n)$

$\text{totient}(n)$ or $\phi(n)$ is the number of positive integers $\leq n$ that are relatively prime to n .

Parameters

n : integer

Examples

```
>>> from sympy.ntheory import totient
>>> totient(1)
1
>>> totient(25)
20
>>> totient(45) == totient(5)*totient(9)
True
```

See also:

[divisor_count](#) (page 1498)

References

[R614], [R615]

class sympy.ntheory.factor_.**reduced_totient**(*n*)

Calculate the Carmichael reduced totient function $\lambda(n)$

$\text{reduced\_totient}(n)$ or $\lambda(n)$ is the smallest $m > 0$ such that $k^m \equiv 1 \pmod n$ for all k relatively prime to n .

Examples

```
>>> from sympy.ntheory import reduced_totient
>>> reduced_totient(1)
1
>>> reduced_totient(8)
2
>>> reduced_totient(30)
4
```

See also:

[*totient*](#) (page 1501)

References

[R616], [R617]

class sympy.ntheory.factor_.**divisor_sigma**(*n*, *k*=1)

Calculate the divisor function $\sigma_k(n)$ for positive integer *n*

`divisor_sigma(n, k)` is equal to `sum([x**k for x in divisors(n)])`

If *n*'s prime factorization is:

$$n = \prod_{i=1}^{\omega} p_i^{m_i},$$

then

$$\sigma_k(n) = \prod_{i=1}^{\omega} (1 + p_i^k + p_i^{2k} + \cdots + p_i^{m_i k}).$$

Parameters

n : integer

k : integer, optional

power of divisors in the sum

for *k* = 0, 1: `divisor_sigma(n, 0)` is equal to `divisor_count(n)`

`divisor_sigma(n, 1)` is equal to `sum(divisors(n))`

Default for *k* is 1.

Examples

```
>>> from sympy.ntheory import divisor_sigma
>>> divisor_sigma(18, 0)
6
>>> divisor_sigma(39, 1)
56
>>> divisor_sigma(12, 2)
210
>>> divisor_sigma(37)
38
```

See also:

[divisor_count](#) (page 1498), [totient](#) (page 1501), [divisors](#) (page 1497), [factorint](#) (page 1493)

References

[R618]

class `sympy.ntheory.factor_.udivisor_sigma(n, k=1)`

Calculate the unitary divisor function $\sigma_k^*(n)$ for positive integer n

`udivisor_sigma(n, k)` is equal to `sum([x**k for x in divisors(n)])`

If n 's prime factorization is:

$$n = \prod_{i=1}^{\omega} p_i^{m_i},$$

then

$$\sigma_k^*(n) = \prod_{i=1}^{\omega} (1 + p_i^{m_i k}).$$

Parameters

k: power of divisors in the sum

for $k = 0, 1$: `udivisor_sigma(n, 0)` is equal to `udivisor_count(n)`

`udivisor_sigma(n, 1)` is equal to `sum(divisors(n))`

Default for k is 1.

Examples

```
>>> from sympy.ntheory.factor_ import udivisor_sigma
>>> udivisor_sigma(18, 0)
4
>>> udivisor_sigma(74, 1)
114
>>> udivisor_sigma(36, 3)
47450
>>> udivisor_sigma(111)
152
```

See also:

[divisor_count](#) (page 1498), [totient](#) (page 1501), [divisors](#) (page 1497), [udivisors](#) (page 1499), [udivisor_count](#) (page 1499), [divisor_sigma](#) (page 1502), [factorint](#) (page 1493)

References

[R619]

`sympy.ntheory.factor_.core(n, t=2)`

Calculate $\text{core}(n, t) = \text{core}_t(n)$ of a positive integer n

$\text{core}_2(n)$ is equal to the squarefree part of n

If n 's prime factorization is:

$$n = \prod_{i=1}^{\omega} p_i^{m_i},$$

then

$$\text{core}_t(n) = \prod_{i=1}^{\omega} p_i^{m_i \bmod t}.$$

Parameters

n : integer

t : integer

$\text{core}(n, t)$ calculates the t -th power free part of n

$\text{core}(n, 2)$ is the squarefree part of n $\text{core}(n, 3)$ is the cubefree part of n

Default for t is 2.

Examples

```
>>> from sympy.ntheory.factor_ import core
>>> core(24, 2)
6
>>> core(9424, 3)
1178
>>> core(379238)
379238
>>> core(15**11, 10)
15
```

See also:

[*factorint*](#) (page 1493), [*sympy.solvers.diophantine.diophantine.square_factor*](#) (page 730)

References

[R620]

`sympy.ntheory.factor_.digits(n, b=10, digits=None)`

Return a list of the digits of n in base b . The first element in the list is b (or $-b$ if n is negative).

Parameters

n: integer

The number whose digits are returned.

b: integer

The base in which digits are computed.

digits: integer (or None for all digits)

The number of digits to be returned (padded with zeros, if necessary).

Examples

```
>>> from sympy.ntheory.digits import digits
>>> digits(35)
[10, 3, 5]
```

If the number is negative, the negative sign will be placed on the base (which is the first element in the returned list):

```
>>> digits(-35)
[-10, 3, 5]
```

Bases other than 10 (and greater than 1) can be selected with `b`:

```
>>> digits(27, b=2)
[2, 1, 1, 0, 1, 1]
```

Use the `digits` keyword if a certain number of digits is desired:

```
>>> digits(35, digits=4)
[10, 0, 0, 3, 5]
```

class `sympy.ntheory.factor_.primenu(n)`

Calculate the number of distinct prime factors for a positive integer n .

If n 's prime factorization is:

$$n = \prod_{i=1}^k p_i^{m_i},$$

then `primenu(n)` or $\nu(n)$ is:

$$\nu(n) = k.$$

Examples

```
>>> from sympy.ntheory.factor_ import primenu
>>> primenu(1)
0
>>> primenu(30)
3
```

See also:

[*factorint*](#) (page 1493)

References

[R621]

class sympy.ntheory.factor_.**primeomega**(*n*)

Calculate the number of prime factors counting multiplicities for a positive integer *n*.

If *n*'s prime factorization is:

$$n = \prod_{i=1}^k p_i^{m_i},$$

then **primeomega**(*n*) or $\Omega(n)$ is:

$$\Omega(n) = \sum_{i=1}^k m_i.$$

Examples

```
>>> from sympy.ntheory.factor_ import primeomega
>>> primeomega(1)
0
>>> primeomega(20)
3
```

See also:

[*factorint*](#) (page 1493)

References

[R622]

sympy.ntheory.factor_.mersenne_prime_exponent(*nth*)

Returns the exponent *i* for the *nth* Mersenne prime (which has the form $2^i - 1$).

Examples

```
>>> from sympy.ntheory.factor_ import mersenne_prime_exponent
>>> mersenne_prime_exponent(1)
2
>>> mersenne_prime_exponent(20)
4423
```

`sympy.ntheory.factor_.is_perfect(n)`

Returns True if *n* is a perfect number, else False.

A perfect number is equal to the sum of its positive, proper divisors.

Examples

```
>>> from sympy.ntheory.factor_ import is_perfect, divisors, divisor_sigma
>>> is_perfect(20)
False
>>> is_perfect(6)
True
>>> 6 == divisor_sigma(6) - 6 == sum(divisors(6)[: -1])
True
```

References

[R623], [R624]

`sympy.ntheory.factor_.is_mersenne_prime(n)`

Returns True if *n* is a Mersenne prime, else False.

A Mersenne prime is a prime number having the form $2^i - 1$.

Examples

```
>>> from sympy.ntheory.factor_ import is_mersenne_prime
>>> is_mersenne_prime(6)
False
>>> is_mersenne_prime(127)
True
```

References

[R625]

`sympy.ntheory.factor_.abundance(n)`

Returns the difference between the sum of the positive proper divisors of a number and the number.

Examples

```
>>> from sympy.ntheory import abundance, is_perfect, is_abundant
>>> abundance(6)
0
>>> is_perfect(6)
True
>>> abundance(10)
-2
>>> is_abundant(10)
False
```

`sympy.ntheory.factor_.is_abundant(n)`

Returns True if *n* is an abundant number, else False.

A abundant number is smaller than the sum of its positive proper divisors.

Examples

```
>>> from sympy.ntheory.factor_ import is_abundant
>>> is_abundant(20)
True
>>> is_abundant(15)
False
```

References

[R626]

`sympy.ntheory.factor_.is_deficient(n)`

Returns True if *n* is a deficient number, else False.

A deficient number is greater than the sum of its positive proper divisors.

Examples

```
>>> from sympy.ntheory.factor_ import is_deficient
>>> is_deficient(20)
False
>>> is_deficient(15)
True
```

References

[R627]

`sympy.ntheory.factor_.is_amicable(m, n)`

Returns True if the numbers *m* and *n* are “amicable”, else False.

Amicable numbers are two different numbers so related that the sum of the proper divisors of each is equal to that of the other.

Examples

```
>>> from sympy.ntheory.factor_ import is_amicable, divisor_sigma
>>> is_amicable(220, 284)
True
>>> divisor_sigma(220) == divisor_sigma(284)
True
```

References

[R628]

`sympy.ntheory.modular.symmetric_residue(a, m)`

Return the residual mod *m* such that it is within half of the modulus.

```
>>> from sympy.ntheory.modular import symmetric_residue
>>> symmetric_residue(1, 6)
1
>>> symmetric_residue(4, 6)
-2
```

`sympy.ntheory.modular.crt(m, v, symmetric=False, check=True)`

Chinese Remainder Theorem.

The moduli in *m* are assumed to be pairwise coprime. The output is then an integer *f*, such that $f = v_i \bmod m_i$ for each pair out of *v* and *m*. If *symmetric* is False a positive integer will be returned, else $|f|$ will be less than or equal to the LCM of the moduli, and thus *f* may be negative.

If the moduli are not co-prime the correct result will be returned if/when the test of the result is found to be incorrect. This result will be None if there is no solution.

The keyword *check* can be set to False if it is known that the moduli are coprime.

Examples

As an example consider a set of residues $U = [49, 76, 65]$ and a set of moduli $M = [99, 97, 95]$. Then we have:

```
>>> from sympy.ntheory.modular import crt
>>> crt([99, 97, 95], [49, 76, 65])
(639985, 912285)
```

This is the correct result because:

```
>>> [639985 % m for m in [99, 97, 95]]
[49, 76, 65]
```

If the moduli are not co-prime, you may receive an incorrect result if you use `check=False`:

```
>>> crt([12, 6, 17], [3, 4, 2], check=False)
(954, 1224)
>>> [954 % m for m in [12, 6, 17]]
[6, 0, 2]
>>> crt([12, 6, 17], [3, 4, 2]) is None
True
>>> crt([3, 6], [2, 5])
(5, 6)
```

Note: the order of `gf_crt`'s arguments is reversed relative to `crt`, and that `solve_congruence` takes residue, modulus pairs.

Programmer's note: rather than checking that all pairs of moduli share no GCD (an $O(n^2)$ test) and rather than factoring all moduli and seeing that there is no factor in common, a check that the result gives the indicated residuals is performed - an $O(n)$ operation.

See also:

[`solve\_congruence`](#) (page 1511)

[`sympy.polys.galoistools.gf\_crt`](#) (page 2602)

low level crt routine used by this routine

`sympy.ntheory.modular.crt1(m)`

First part of Chinese Remainder Theorem, for multiple application.

Examples

```
>>> from sympy.ntheory.modular import crt1
>>> crt1([18, 42, 6])
(4536, [252, 108, 756], [0, 2, 0])
```

`sympy.ntheory.modular.crt2(m, v, mm, e, s, symmetric=False)`

Second part of Chinese Remainder Theorem, for multiple application.

Examples

```
>>> from sympy.ntheory.modular import crt1, crt2
>>> mm, e, s = crt1([18, 42, 6])
>>> crt2([18, 42, 6], [0, 0, 0], mm, e, s)
(0, 4536)
```

`sympy.ntheory.modular.solve_congruence(*remainder_modulus_pairs, **hint)`

Compute the integer n that has the residual a_i when it is divided by m_i where the a_i and m_i are given as pairs to this function: $((a_1, m_1), (a_2, m_2), \dots)$. If there is no solution, return `None`. Otherwise return n and its modulus.

The m_i values need not be co-prime. If it is known that the moduli are not co-prime then the hint check can be set to `False` (default=`True`) and the check for a quicker solution via `crt()` (valid when the moduli are co-prime) will be skipped.

If the hint `symmetric` is `True` (default is `False`), the value of n will be within $1/2$ of the modulus, possibly negative.

Examples

```
>>> from sympy.ntheory.modular import solve_congruence
```

What number is 2 mod 3, 3 mod 5 and 2 mod 7?

```
>>> solve_congruence((2, 3), (3, 5), (2, 7))
(23, 105)
>>> [23 % m for m in [3, 5, 7]]
[2, 3, 2]
```

If you prefer to work with all remainder in one list and all moduli in another, send the arguments like this:

```
>>> solve_congruence(*zip((2, 3, 2), (3, 5, 7)))
(23, 105)
```

The moduli need not be co-prime; in this case there may or may not be a solution:

```
>>> solve_congruence((2, 3), (4, 6)) is None
True
```

```
>>> solve_congruence((2, 3), (5, 6))
(5, 6)
```

The `symmetric` flag will make the result be within $1/2$ of the modulus:

```
>>> solve_congruence((2, 3), (5, 6), symmetric=True)
(-1, 6)
```

See also:

`crt` (page 1509)

high level routine implementing the Chinese Remainder Theorem

`sympy.ntheory.multinomial.binomial_coefficients(n)`

Return a dictionary containing pairs $(k_1, k_2) : C_k n$ where $C_k n$ are binomial coefficients and $n = k_1 + k_2$.

Examples

```
>>> from sympy.ntheory import binomial_coefficients
>>> binomial_coefficients(9)
{(0, 9): 1, (1, 8): 9, (2, 7): 36, (3, 6): 84,
 (4, 5): 126, (5, 4): 126, (6, 3): 84, (7, 2): 36, (8, 1): 9, (9, 0): 1}
```

See also:

[*binomial_coefficients_list*](#) (page 1512), [*multinomial_coefficients*](#) (page 1512)

`sympy.ntheory.multinomial.binomial_coefficients_list(n)`

Return a list of binomial coefficients as rows of the Pascal's triangle.

Examples

```
>>> from sympy.ntheory import binomial_coefficients_list
>>> binomial_coefficients_list(9)
[1, 9, 36, 84, 126, 126, 84, 36, 9, 1]
```

See also:

[*binomial_coefficients*](#) (page 1511), [*multinomial_coefficients*](#) (page 1512)

`sympy.ntheory.multinomial.multinomial_coefficients(m, n)`

Return a dictionary containing pairs $\{(k_1, k_2, \dots, k_m) : C_{\mathbf{k}} n\}$ where $C_{\mathbf{k}} n$ are multinomial coefficients such that $n = k_1 + k_2 + \dots + k_m$.

Examples

```
>>> from sympy.ntheory import multinomial_coefficients
>>> multinomial_coefficients(2, 5) # indirect doctest
{(0, 5): 1, (1, 4): 5, (2, 3): 10, (3, 2): 10, (4, 1): 5, (5, 0): 1}
```

Notes

The algorithm is based on the following result:

$$\binom{n}{k_1, \dots, k_m} = \frac{k_1 + 1}{n - k_1} \sum_{i=2}^m \binom{n}{k_1 + 1, \dots, k_i - 1, \dots}$$

Code contributed to Sage by Yann Laigle-Chapuy, copied with permission of the author.

See also:

[*binomial_coefficients_list*](#) (page 1512), [*binomial_coefficients*](#) (page 1511)

`sympy.ntheory.multinomial.multinomial_coefficients_iterator(m, n, tuple=<class 'tuple'>)`

multinomial coefficient iterator

This routine has been optimized for m large with respect to n by taking advantage of the fact that when the monomial tuples t are stripped of zeros, their coefficient is the same as that of the monomial tuples from `multinomial_coefficients(n, n)`. Therefore, the latter coefficients are precomputed to save memory and time.

```
>>> from sympy.ntheory.multinomial import multinomial_coefficients
>>> m53, m33 = multinomial_coefficients(5,3), multinomial_coefficients(3,
↪ 3)
>>> m53[(0,0,0,1,2)] == m53[(0,0,1,0,2)] == m53[(1,0,2,0,0)] == m33[(0,1,
↪ 2)]
True
```

Examples

```
>>> from sympy.ntheory.multinomial import multinomial_coefficients_
↪ iterator
>>> it = multinomial_coefficients_iterator(20,3)
>>> next(it)
((3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), 1)
```

`sympy.ntheory.partitions_.npartitions(n, verbose=False)`

Calculate the partition function $P(n)$, i.e. the number of ways that n can be written as a sum of positive integers.

$P(n)$ is computed using the Hardy-Ramanujan-Rademacher formula [R629].

The correctness of this implementation has been tested through 10^{10} .

Examples

```
>>> from sympy.ntheory import npartitions
>>> npartitions(25)
1958
```

References

[R629]

`sympy.ntheory.primetest.is_euler_pseudoprime(n, b)`

Returns True if n is prime or an Euler pseudoprime to base b , else False.

Euler Pseudoprime : In arithmetic, an odd composite integer n is called an euler pseudoprime to base a , if a and n are coprime and satisfy the modular arithmetic congruence relation :

$$a^{(n-1)/2} \equiv \pm 1 \pmod{n} \text{ or } a^{(n-1)/2} \equiv -1 \pmod{n}$$

(where mod refers to the modulo operation).

Examples

```
>>> from sympy.ntheory.primetest import is_euler_pseudoprime
>>> is_euler_pseudoprime(2, 5)
True
```

References

[R630]

`sympy.ntheory.primetest.is_square(n, prep=True)`

Return True if $n == a * a$ for some integer a , else False. If n is suspected of *not* being a square then this is a quick method of confirming that it is not.

Examples

```
>>> from sympy.ntheory.primetest import is_square
>>> is_square(25)
True
>>> is_square(2)
False
```

See also:

[`sympy.core.power.integer\_nthroot`](#) (page 1008)

References

[R631]

`sympy.ntheory.primetest.mr(n, bases)`

Perform a Miller-Rabin strong pseudoprime test on n using a given list of bases/witnesses.

Examples

```
>>> from sympy.ntheory.primetest import mr
>>> mr(1373651, [2, 3])
False
>>> mr(479001599, [31, 73])
True
```

References

A list of thresholds and the bases they require are here: https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test#Deterministic_variants

[R632]

`sympy.ntheory.primetest.is_lucas_prp(n)`

Standard Lucas compositeness test with Selfridge parameters. Returns False if n is definitely composite, and True if n is a Lucas probable prime.

This is typically used in combination with the Miller-Rabin test.

Examples

```
>>> from sympy.ntheory.primetest import isprime, is_lucas_prp
>>> for i in range(10000):
...     if is_lucas_prp(i) and not isprime(i):
...         print(i)
323
377
1159
1829
3827
5459
5777
9071
9179
```

References

- “Lucas Pseudoprimes”, Baillie and Wagstaff, 1980. <http://mpqs.free.fr/LucasPseudoprimes.pdf>
- OEIS A217120: Lucas Pseudoprimes <https://oeis.org/A217120>
- https://en.wikipedia.org/wiki/Lucas_pseudoprime

`sympy.ntheory.primetest.is_strong_lucas_prp(n)`

Strong Lucas compositeness test with Selfridge parameters. Returns False if n is definitely composite, and True if n is a strong Lucas probable prime.

This is often used in combination with the Miller-Rabin test, and in particular, when combined with M-R base 2 creates the strong BPSW test.

Examples

```
>>> from sympy.ntheory.primetest import isprime, is_strong_lucas_prp
>>> for i in range(20000):
...     if is_strong_lucas_prp(i) and not isprime(i):
...         print(i)
5459
5777
10877
16109
18971
```

References

- “Lucas Pseudoprimes”, Baillie and Wagstaff, 1980. <http://mpqs.free.fr/LucasPseudoprimes.pdf>
- OEIS A217255: Strong Lucas Pseudoprimes <https://oeis.org/A217255>
- https://en.wikipedia.org/wiki/Lucas_pseudoprime
- https://en.wikipedia.org/wiki/Baillie-PSW_primality_test

`sympy.ntheory.primetest.is_extra_strong_lucas_prp(n)`

Extra Strong Lucas compositeness test. Returns False if n is definitely composite, and True if n is a “extra strong” Lucas probable prime.

The parameters are selected using $P = 3$, $Q = 1$, then incrementing P until $(D|n) == -1$. The test itself is as defined in Grantham 2000, from the Mo and Jones preprint. The parameter selection and test are the same as used in OEIS A217719, Perl’s `Math::Prime::Util`, and the Lucas pseudoprime page on Wikipedia.

With these parameters, there are no counterexamples below 2^{64} nor any known above that range. It is 20-50% faster than the strong test.

Because of the different parameters selected, there is no relationship between the strong Lucas pseudoprimes and extra strong Lucas pseudoprimes. In particular, one is not a subset of the other.

Examples

```
>>> from sympy.ntheory.primetest import isprime, is_extra_strong_lucas_
→prp
>>> for i in range(20000):
...     if is_extra_strong_lucas_prp(i) and not isprime(i):
...         print(i)
989
3239
5777
10877
```

References

- “Frobenius Pseudoprimes”, Jon Grantham, 2000. <http://www.ams.org/journals/mcom/2001-70-234/S0025-5718-00-01197-2/>
- OEIS A217719: Extra Strong Lucas Pseudoprimes <https://oeis.org/A217719>
- https://en.wikipedia.org/wiki/Lucas_pseudoprime

`sympy.ntheory.primetest.isprime(n)`

Test if n is a prime number (True) or not (False). For $n < 2^{64}$ the answer is definitive; larger n values have a small probability of actually being pseudoprimes.

Negative numbers (e.g. -2) are not considered prime.

The first step is looking for trivial factors, which if found enables a quick return. Next, if the sieve is large enough, use bisection search on the sieve. For small numbers, a set of deterministic Miller-Rabin tests are performed with bases that are known to have no counterexamples in their range. Finally if the number is larger than 2^{64} , a strong BPSW test is performed. While this is a probable prime test and we believe counterexamples exist, there are no known counterexamples.

Examples

```
>>> from sympy.ntheory import isprime
>>> isprime(13)
True
>>> isprime(13.0) # limited precision
False
>>> isprime(15)
False
```

Notes

This routine is intended only for integer input, not numerical expressions which may represent numbers. Floats are also rejected as input because they represent numbers of limited precision. While it is tempting to permit 7.0 to represent an integer there are errors that may “pass silently” if this is allowed:

```
>>> from sympy import Float, S
>>> int(1e3) == 1e3 == 10**3
True
>>> int(1e23) == 1e23
True
>>> int(1e23) == 10**23
False
```

```
>>> near_int = 1 + S(1)/10**19
>>> near_int == int(near_int)
False
>>> n = Float(near_int, 10) # truncated by precision
>>> n == int(n)
```

(continues on next page)

(continued from previous page)

```
True
>>> n = Float(near_int, 20)
>>> n == int(n)
False
```

See also:***sympy.ntheory.generate.primerange* (page 1481)**

Generates all primes in a given range

***sympy.ntheory.generate.primepi* (page 1479)**

Return the number of primes less than or equal to n

***sympy.ntheory.generate.prime* (page 1479)**

Return the nth prime

References

- https://en.wikipedia.org/wiki/Strong_pseudoprime
- “Lucas Pseudoprimes”, Baillie and Wagstaff, 1980. <http://mpqs.free.fr/LucasPseudoprimes.pdf>
- https://en.wikipedia.org/wiki/Baillie-PSW_primality_test

sympy.ntheory.primetest.is_gaussian_prime(num)

Test if num is a Gaussian prime number.

References[\[R633\]](#)***sympy.ntheory.residue_ntheory.n_order(a, n)***

Returns the order of a modulo n.

The order of a modulo n is the smallest integer k such that a^k leaves a remainder of 1 with n.

Examples

```
>>> from sympy.ntheory import n_order
>>> n_order(3, 7)
6
>>> n_order(4, 7)
3
```

sympy.ntheory.residue_ntheory.is_primitive_root(a, p)

Returns True if a is a primitive root of p

a is said to be the primitive root of p if $\gcd(a, p) == 1$ and totient(p) is the smallest positive number s.t.

$$a^{\text{totient}(p)} \text{ cong } 1 \text{ mod}(p)$$

Examples

```
>>> from sympy.ntheory import is_primitive_root, n_order, totient
>>> is_primitive_root(3, 10)
True
>>> is_primitive_root(9, 10)
False
>>> n_order(3, 10) == totient(10)
True
>>> n_order(9, 10) == totient(10)
False
```

`sympy.ntheory.residue_ntheory.primitive_root(p)`

Returns the smallest primitive root or None

Parameters

p : positive integer

Examples

```
>>> from sympy.ntheory.residue_ntheory import primitive_root
>>> primitive_root(19)
2
```

References

[R634], [R635]

`sympy.ntheory.residue_ntheory.sqrt_mod(a, p, all_roots=False)`

Find a root of $x^2 = a \pmod{p}$

Parameters

a : integer

p : positive integer

all_roots : if True the list of roots is returned or None

Notes

If there is no root it is returned None; else the returned root is less or equal to $p // 2$; in general is not the smallest one. It is returned $p // 2$ only if it is the only root.

Use `all_roots` only when it is expected that all the roots fit in memory; otherwise use `sqrt_mod_iter`.

Examples

```
>>> from sympy.ntheory import sqrt_mod
>>> sqrt_mod(11, 43)
21
>>> sqrt_mod(17, 32, True)
[7, 9, 23, 25]
```

`sympy.ntheory.residue_ntheory.sqrt_mod_iter(a, p, domain=<class 'int'>)`

Iterate over solutions to $x^2 = a \bmod p$

Parameters

a : integer

p : positive integer

domain : integer domain, int, ZZ or Integer

Examples

```
>>> from sympy.ntheory.residue_ntheory import sqrt_mod_iter
>>> list(sqrt_mod_iter(11, 43))
[21, 22]
```

`sympy.ntheory.residue_ntheory.quadratic_residues(p) → list[int]`

Returns the list of quadratic residues.

Examples

```
>>> from sympy.ntheory.residue_ntheory import quadratic_residues
>>> quadratic_residues(7)
[0, 1, 2, 4]
```

`sympy.ntheory.residue_ntheory.nthroot_mod(a, n, p, all_roots=False)`

Find the solutions to $x^n = a \bmod p$

Parameters

a : integer

n : positive integer

p : positive integer

all_roots : if False returns the smallest root, else the list of roots

Examples

```
>>> from sympy.ntheory.residue_ntheory import nthroot_mod
>>> nthroot_mod(11, 4, 19)
8
>>> nthroot_mod(11, 4, 19, True)
[8, 11]
>>> nthroot_mod(68, 3, 109)
23
```

`sympy.ntheory.residue_ntheory.is_nthpow_residue(a, n, m)`

Returns True if $x^n \equiv a \pmod{m}$ has solutions.

References

[R636]

`sympy.ntheory.residue_ntheory.is_quad_residue(a, p)`

Returns True if $a \pmod{p}$ is in the set of squares mod p , i.e $a \pmod{p}$ in $\{i^2 \pmod{p} \text{ for } i \text{ in } \text{range}(p)\}$. If p is an odd prime, an iterative method is used to make the determination:

```
>>> from sympy.ntheory import is_quad_residue
>>> sorted(set([i**2 % 7 for i in range(7)]))
[0, 1, 2, 4]
>>> [j for j in range(7) if is_quad_residue(j, 7)]
[0, 1, 2, 4]
```

See also:

[`legendre\_symbol`](#) (page 1521), [`jacobi\_symbol`](#) (page 1522)

`sympy.ntheory.residue_ntheory.legendre_symbol(a, p)`

Returns the Legendre symbol (a/p) .

For an integer a and an odd prime p , the Legendre symbol is defined as

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & \text{if } p \text{ divides } a \\ 1 & \text{if } a \text{ is a quadratic residue modulo } p \\ -1 & \text{if } a \text{ is a quadratic nonresidue modulo } p \end{cases}$$

Parameters

a : integer

p : odd prime

Examples

```
>>> from sympy.ntheory import legendre_symbol
>>> [legendre_symbol(i, 7) for i in range(7)]
[0, 1, 1, -1, 1, -1, -1]
>>> sorted(set([i**2 % 7 for i in range(7)]))
[0, 1, 2, 4]
```

See also:

[*is_quad_residue*](#) (page 1521), [*jacobi_symbol*](#) (page 1522)

`sympy.ntheory.residue_ntheory.jacobi_symbol(m, n)`

Returns the Jacobi symbol (m/n) .

For any integer m and any positive odd integer n the Jacobi symbol is defined as the product of the Legendre symbols corresponding to the prime factors of n :

$$\left(\frac{m}{n}\right) = \left(\frac{m}{p^1}\right)^{\alpha_1} \left(\frac{m}{p^2}\right)^{\alpha_2} \cdots \left(\frac{m}{p^k}\right)^{\alpha_k} \text{ where } n = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k}$$

Like the Legendre symbol, if the Jacobi symbol $(\frac{m}{n}) = -1$ then m is a quadratic nonresidue modulo n .

But, unlike the Legendre symbol, if the Jacobi symbol $(\frac{m}{n}) = 1$ then m may or may not be a quadratic residue modulo n .

Parameters

m : integer

n : odd positive integer

Examples

```
>>> from sympy.ntheory import jacobi_symbol, legendre_symbol
>>> from sympy import S
>>> jacobi_symbol(45, 77)
-1
>>> jacobi_symbol(60, 121)
1
```

The relationship between the `jacobi_symbol` and `legendre_symbol` can be demonstrated as follows:

```
>>> L = legendre_symbol
>>> S(45).factors()
{3: 2, 5: 1}
>>> jacobi_symbol(7, 45) == L(7, 3)**2 * L(7, 5)**1
True
```

See also:

[*is_quad_residue*](#) (page 1521), [*legendre_symbol*](#) (page 1521)

`sympy.ntheory.residue_ntheory.discrete_log(n, a, b, order=None, prime_order=None)`

Compute the discrete logarithm of a to the base b modulo n.

This is a recursive function to reduce the discrete logarithm problem in cyclic groups of composite order to the problem in cyclic groups of prime order.

It employs different algorithms depending on the problem (subgroup order size, prime order or not):

- Trial multiplication
- Baby-step giant-step
- Pollard's Rho
- Pohlig-Hellman

Examples

```
>>> from sympy.ntheory import discrete_log
>>> discrete_log(41, 15, 7)
3
```

References

[R637], [R638]

`sympy.ntheory.continued_fraction.continued_fraction(a) → list`

Return the continued fraction representation of a Rational or quadratic irrational.

Examples

```
>>> from sympy.ntheory.continued_fraction import continued_fraction
>>> from sympy import sqrt
>>> continued_fraction((1 + 2*sqrt(3))/5)
[0, 1, [8, 3, 34, 3]]
```

See also:

[`continued\_fraction\_periodic`](#) (page 1525), [`continued\_fraction\_reduce`](#) (page 1526), [`continued\_fraction\_convergents`](#) (page 1523)

`sympy.ntheory.continued_fraction.continued_fraction_convergents(cf)`

Return an iterator over the convergents of a continued fraction (cf).

The parameter should be an iterable returning successive partial quotients of the continued fraction, such as might be returned by `continued_fraction_iterator`. In computing the convergents, the continued fraction need not be strictly in canonical form (all integers, all but the first positive). Rational and negative elements may be present in the expansion.

Examples

```
>>> from sympy.core import pi
>>> from sympy import S
>>> from sympy.ntheory.continued_fraction import continued_
    ↪ fraction_convergents, continued_fraction_iterator
```

```
>>> list(continued_fraction_convergents([0, 2, 1, 2]))
[0, 1/2, 1/3, 3/8]
```

```
>>> list(continued_fraction_convergents([1, S('1/2'), -7, S('1/4')]))
[1, 3, 19/5, 7]
```

```
>>> it = continued_fraction_convergents(continued_fraction_iterator(pi))
>>> for n in range(7):
...     print(next(it))
3
22/7
333/106
355/113
103993/33102
104348/33215
208341/66317
```

See also:

[*continued_fraction_iterator*](#) (page 1524)

`sympy.ntheory.continued_fraction.continued_fraction_iterator(x)`

Return continued fraction expansion of x as iterator.

Examples

```
>>> from sympy import Rational, pi
>>> from sympy.ntheory.continued_fraction import continued_fraction_
    ↪ iterator
```

```
>>> list(continued_fraction_iterator(Rational(3, 8)))
[0, 2, 1, 2]
>>> list(continued_fraction_iterator(Rational(-3, 8)))
[-1, 1, 1, 1, 2]
```

```
>>> for i, v in enumerate(continued_fraction_iterator(pi)):
...     if i > 7:
...         break
...     print(v)
3
7
15
1
292
```

(continues on next page)

(continued from previous page)

```
1
1
1
```

References

[R639]

`sympy.ntheory.continued_fraction.continued_fraction_periodic(p, q, d=0, s=1) → list`

Find the periodic continued fraction expansion of a quadratic irrational.

Compute the continued fraction expansion of a rational or a quadratic irrational number, i.e. $\frac{p+s\sqrt{d}}{q}$, where $p, q \neq 0$ and $d \geq 0$ are integers.

Returns the continued fraction representation (canonical form) as a list of integers, optionally ending (for quadratic irrationals) with list of integers representing the repeating digits.

Parameters

p : int

the rational part of the number's numerator

q : int

the denominator of the number

d : int, optional

the irrational part (discriminator) of the number's numerator

s : int, optional

the coefficient of the irrational part

Examples

```
>>> from sympy.ntheory.continued_fraction import continued_fraction_
    ↪ periodic
>>> continued_fraction_periodic(3, 2, 7)
[2, [1, 4, 1, 1]]
```

Golden ratio has the simplest continued fraction expansion:

```
>>> continued_fraction_periodic(1, 2, 5)
[[1]]
```

If the discriminator is zero or a perfect square then the number will be a rational number:

```
>>> continued_fraction_periodic(4, 3, 0)
[1, 3]
>>> continued_fraction_periodic(4, 3, 49)
[3, 1, 2]
```

See also:

[*continued_fraction_iterator*](#) (page 1524), [*continued_fraction_reduce*](#) (page 1526)

References

[R640], [R641]

`sympy.ntheory.continued_fraction.continued_fraction_reduce(cf)`

Reduce a continued fraction to a rational or quadratic irrational.

Compute the rational or quadratic irrational number from its terminating or periodic continued fraction expansion. The continued fraction expansion (cf) should be supplied as a terminating iterator supplying the terms of the expansion. For terminating continued fractions, this is equivalent to `list(continued_fraction_convergents(cf))[-1]`, only a little more efficient. If the expansion has a repeating part, a list of the repeating terms should be returned as the last element from the iterator. This is the format returned by `continued_fraction_periodic`.

For quadratic irrationals, returns the largest solution found, which is generally the one sought, if the fraction is in canonical form (all terms positive except possibly the first).

Examples

```
>>> from sympy.ntheory.continued_fraction import continued_fraction_
    ↪ reduce
>>> continued_fraction_reduce([1, 2, 3, 4, 5])
225/157
>>> continued_fraction_reduce([-2, 1, 9, 7, 1, 2])
-256/233
>>> continued_fraction_reduce([2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8]).n(10)
2.718281835
>>> continued_fraction_reduce([1, 4, 2, [3, 1]])
(sqrt(21) + 287)/238
>>> continued_fraction_reduce([[1]])
(1 + sqrt(5))/2
>>> from sympy.ntheory.continued_fraction import continued_fraction_
    ↪ periodic
>>> continued_fraction_reduce(continued_fraction_periodic(8, 5, 13))
(sqrt(13) + 8)/5
```

See also:

[*continued_fraction_periodic*](#) (page 1525)

`sympy.ntheory.digits.count_digits(n, b=10)`

Return a dictionary whose keys are the digits of `n` in the given base, `b`, with keys indicating the digits appearing in the number and values indicating how many times that digit appeared.

Examples

```
>>> from sympy.ntheory import count_digits
```

```
>>> count_digits(1111339)
{1: 4, 3: 2, 9: 1}
```

The digits returned are always represented in base-10 but the number itself can be entered in any format that is understood by Python; the base of the number can also be given if it is different than 10:

```
>>> n = 0xFA; n
250
>>> count_digits(_)
{0: 1, 2: 1, 5: 1}
>>> count_digits(n, 16)
{10: 1, 15: 1}
```

The default dictionary will return a 0 for any digit that did not appear in the number. For example, which digits appear 7 times in 77!:

```
>>> from sympy import factorial
>>> c77 = count_digits(factorial(77))
>>> [i for i in range(10) if c77[i] == 7]
[1, 3, 7, 9]
```

`sympy.ntheory.digits.digits(n, b=10, digits=None)`

Return a list of the digits of *n* in base *b*. The first element in the list is *b* (or *-b* if *n* is negative).

Parameters

n: integer

The number whose digits are returned.

b: integer

The base in which digits are computed.

digits: integer (or None for all digits)

The number of digits to be returned (padded with zeros, if necessary).

Examples

```
>>> from sympy.ntheory.digits import digits
>>> digits(35)
[10, 3, 5]
```

If the number is negative, the negative sign will be placed on the base (which is the first element in the returned list):

```
>>> digits(-35)
[-10, 3, 5]
```

Bases other than 10 (and greater than 1) can be selected with `b`:

```
>>> digits(27, b=2)
[2, 1, 1, 0, 1, 1]
```

Use the `digits` keyword if a certain number of digits is desired:

```
>>> digits(35, digits=4)
[10, 0, 0, 3, 5]
```

`sympy.ntheory.digits.is_palindromic(n, b=10)`

return True if *n* is the same when read from left to right or right to left in the given base, *b*.

Examples

```
>>> from sympy.ntheory import is_palindromic
```

```
>>> all(is_palindromic(i) for i in (-11, 1, 22, 121))
True
```

The second argument allows you to test numbers in other bases. For example, 88 is palindromic in base-10 but not in base-8:

```
>>> is_palindromic(88, 8)
False
```

On the other hand, a number can be palindromic in base-8 but not in base-10:

```
>>> 0o121, is_palindromic(0o121)
(81, False)
```

Or it might be palindromic in both bases:

```
>>> oct(121), is_palindromic(121, 8) and is_palindromic(121)
('0o171', True)
```

class `sympy.ntheory.mobius(n)`

Mobius function maps natural number to $\{-1, 0, 1\}$

It is defined as follows:

- 1) 1 if $n = 1$.
- 2) 0 if n has a squared prime factor.
- 3) $(-1)^k$ if n is a square-free positive integer with k number of prime factors.

It is an important multiplicative function in number theory and combinatorics. It has applications in mathematical series, algebraic number theory and also physics (Fermion operator has very concrete realization with Mobius Function model).

Parameters

n : positive integer

Examples

```
>>> from sympy.ntheory import mobius
>>> mobius(13*7)
1
>>> mobius(1)
1
>>> mobius(13*7*5)
-1
>>> mobius(13**2)
0
```

References

[R642], [R643]

`sympy.ntheory.egyptian_fraction.egyptian_fraction(r, algorithm='Greedy')`

Return the list of denominators of an Egyptian fraction expansion [R644] of the said rational *r*.

Parameters

r : Rational or (p, q)

a positive rational number, p/q.

algorithm : { "Greedy", "Graham Jewett", "Takenouchi", "Golomb" }, optional

Denotes the algorithm to be used (the default is "Greedy").

Examples

```
>>> from sympy import Rational
>>> from sympy.ntheory.egyptian_fraction import egyptian_fraction
>>> egyptian_fraction(Rational(3, 7))
[3, 11, 231]
>>> egyptian_fraction((3, 7), "Graham Jewett")
[7, 8, 9, 56, 57, 72, 3192]
>>> egyptian_fraction((3, 7), "Takenouchi")
[4, 7, 28]
>>> egyptian_fraction((3, 7), "Golomb")
[3, 15, 35]
>>> egyptian_fraction((11, 5), "Golomb")
[1, 2, 3, 4, 9, 234, 1118, 2580]
```

Notes

Currently the following algorithms are supported:

1) Greedy Algorithm

Also called the Fibonacci-Sylvester algorithm [R645]. At each step, extract the largest unit fraction less than the target and replace the target with the remainder.

It has some distinct properties:

- a) Given p/q in lowest terms, generates an expansion of maximum length p . Even as the numerators get large, the number of terms is seldom more than a handful.
- b) Uses minimal memory.
- c) The terms can blow up (standard examples of this are $5/121$ and $31/311$). The denominator is at most squared at each step (doubly-exponential growth) and typically exhibits singly-exponential growth.

2) Graham Jewett Algorithm

The algorithm suggested by the result of Graham and Jewett. Note that this has a tendency to blow up: the length of the resulting expansion is always $2^{**}(x/\text{gcd}(x, y)) - 1$. See [R646].

3) Takenouchi Algorithm

The algorithm suggested by Takenouchi (1921). Differs from the Graham-Jewett algorithm only in the handling of duplicates. See [R646].

4) Golomb's Algorithm

A method given by Golomb (1962), using modular arithmetic and inverses. It yields the same results as a method using continued fractions proposed by Bleicher (1972). See [R647].

If the given rational is greater than or equal to 1, a greedy algorithm of summing the harmonic sequence $1/1 + 1/2 + 1/3 + \dots$ is used, taking all the unit fractions of this sequence until adding one more would be greater than the given number. This list of denominators is prefixed to the result from the requested algorithm used on the remainder. For example, if r is $8/3$, using the Greedy algorithm, we get $[1, 2, 3, 4, 5, 6, 7, 14, 420]$, where the beginning of the sequence, $[1, 2, 3, 4, 5, 6, 7]$ is part of the harmonic sequence summing to $363/140$, leaving a remainder of $31/420$, which yields $[14, 420]$ by the Greedy algorithm. The result of `egyptian_fraction(Rational(8, 3), "Golomb")` is $[1, 2, 3, 4, 5, 6, 7, 14, 574, 2788, 6460, 11590, 33062, 113820]$, and so on.

See also:

[`sympy.core.numbers.Rational`](#) (page 985)

References

[R644], [R645], [R646], [R647]

`sympy.ntheory.bbp_pi.pi_hex_digits(n, prec=14)`

Returns a string containing `prec` (default 14) digits starting at the `n`th digit of pi in hex. Counting of digits starts at 0 and the decimal is not counted, so for `n = 0` the returned value starts with 3; `n = 1` corresponds to the first digit past the decimal point (which in hex is 2).

Examples

```
>>> from sympy.ntheory.bbp_pi import pi_hex_digits
>>> pi_hex_digits(0)
'3243f6a8885a30'
>>> pi_hex_digits(0, 3)
'324'
```

References

[R648]

ECM function

The *ecm* function is a subexponential factoring algorithm capable of factoring numbers of around ~35 digits comfortably within few seconds. The time complexity of *ecm* is dependent on the smallest proper factor of the number. So even if the number is really large but its factors are comparatively smaller then *ecm* can easily factor them. For example we take *N* with 15 digit factors 15154262241479, 15423094826093, 79933355511111, 809709509409109, 888888877777777, 914148152112161. Now *N* is a 87 digit number. *ECM* takes under around 47s to factorise this.

`sympy.ntheory.ecm.ecm(n, B1=10000, B2=100000, max_curve=200, seed=1234)`

Performs factorization using Lenstra's Elliptic curve method.

This function repeatedly calls *ecm_one_factor* to compute the factors of *n*. First all the small factors are taken out using trial division. Then *ecm_one_factor* is used to compute one factor at a time.

Parameters

n : Number to be Factored

B1 : Stage 1 Bound

B2 : Stage 2 Bound

max_curve : Maximum number of curves generated

seed : Initialize pseudorandom generator

Examples

```
>>> from sympy.ntheory import ecm
>>> ecm(25645121643901801)
{5394769, 4753701529}
>>> ecm(9804659461513846513)
{4641991, 2112166839943}
```

Examples

```
>>> from sympy.ntheory import ecm
>>> ecm(7060005655815754299976961394452809, B1=100000, B2=1000000)
{6988699669998001, 1010203040506070809}
>>> ↵
↵ ecm(122921448543883967430908091422761898618349713604256384403202282756086473494959648313,
↵ B1=100000, B2=1000000)
{15154262241479,
15423094826093,
79933355511111,
809709509409109,
888888877777777,
914148152112161}
```

QS function

The *qs* function is a subexponential factoring algorithm, the fastest factoring algorithm for numbers within 100 digits. The time complexity of *qs* is dependent on the size of the number so it is used if the number contains large factors. Due to this while factoring numbers first *ecm* is used to get smaller factors of around ~15 digits then *qs* is used to get larger factors.

For factoring 2709077133180915240135586837960864768806330782747 which is a semi-prime number with two 25 digit factors. *qs* is able to factorize this in around 248s.

`sympy.ntheory.qs.qs(N, prime_bound, M, ERROR_TERM=25, seed=1234)`

Performs factorization using Self-Initializing Quadratic Sieve. In SIQS, let N be a number to be factored, and this N should not be a perfect power. If we find two integers such that $X^2 = Y^2 \pmod{N}$ and $X \not\equiv \pm Y \pmod{N}$, then $\gcd(X + Y, N)$ will reveal a proper factor of N . In order to find these integers X and Y we try to find relations of form $t^2 = u \pmod{N}$ where u is a product of small primes. If we have enough of these relations then we can form $(t_1 t_2 \dots t_i)^2 = u_1 u_2 \dots u_i \pmod{N}$ such that the right hand side is a square, thus we found a relation of $X^2 = Y^2 \pmod{N}$.

Here, several optimizations are done like using multiple polynomials for sieving, fast changing between polynomials and using partial relations. The use of partial relations can speed up the factoring by 2 times.

Parameters

N : Number to be Factored

prime_bound : upper bound for primes in the factor base

M : Sieve Interval

ERROR_TERM : Error term for checking smoothness

threshold : Extra smooth relations for factorization

seed : generate pseudo prime numbers

Examples

```
>>> from sympy.ntheory import qs
>>> qs(25645121643901801, 2000, 10000)
{5394769, 4753701529}
>>> qs(9804659461513846513, 2000, 10000)
{4641991, 2112166839943}
```

References

[R649], [R650]

Examples

```
>>> from sympy.ntheory import qs
>>> qs(5915587277*3267000013, 1000, 10000)
{3267000013, 5915587277}
```

5.8.6 Physics

A module that helps solving problems in physics.

Contents

Hydrogen Wavefunctions

`sympy.physics.hydrogen.E_nl(n, Z=1)`

Returns the energy of the state (n, l) in Hartree atomic units.

The energy does not depend on “l”.

Parameters

n : integer

Principal Quantum Number which is an integer with possible values as 1, 2, 3, 4,...

Z :

Atomic number (1 for Hydrogen, 2 for Helium, ...)

Examples

```
>>> from sympy.physics.hydrogen import E_nl
>>> from sympy.abc import n, Z
>>> E_nl(n, Z)
-Z**2/(2*n**2)
>>> E_nl(1)
-1/2
>>> E_nl(2)
-1/8
>>> E_nl(3)
-1/18
>>> E_nl(3, 47)
-2209/18
```

`sympy.physics.hydrogen.E_nl_dirac(n, l, spin_up=True, Z=1, c=137.035999037000)`

Returns the relativistic energy of the state (n, l, spin) in Hartree atomic units.

The energy is calculated from the Dirac equation. The rest mass energy is *not* included.

Parameters

n : integer

Principal Quantum Number which is an integer with possible values as 1, 2, 3, 4,...

l : integer

l is the Angular Momentum Quantum Number with values ranging from 0 to n-1.

spin_up :

True if the electron spin is up (default), otherwise down

Z :

Atomic number (1 for Hydrogen, 2 for Helium, ...)

c :

Speed of light in atomic units. Default value is 137.035999037, taken from <http://arxiv.org/abs/1012.3627>

Examples

```
>>> from sympy.physics.hydrogen import E_nl_dirac
>>> E_nl_dirac(1, 0)
-0.500006656595360
```

```
>>> E_nl_dirac(2, 0)
-0.125002080189006
>>> E_nl_dirac(2, 1)
-0.125000416028342
>>> E_nl_dirac(2, 1, False)
-0.125002080189006
```

```

>>> E_nl_dirac(3, 0)
-0.05555562951740285
>>> E_nl_dirac(3, 1)
-0.05555558020932949
>>> E_nl_dirac(3, 1, False)
-0.05555562951740285
>>> E_nl_dirac(3, 2)
-0.05555556377366884
>>> E_nl_dirac(3, 2, False)
-0.05555558020932949

```

`sympy.physics.hydrogen.Psi_nlm(n, l, m, r, phi, theta, Z=1)`

Returns the Hydrogen wave function ψ_{nlm} . It's the product of the radial wavefunction R_{nl} and the spherical harmonic Y_l^m .

Parameters

n : integer

Principal Quantum Number which is an integer with possible values as 1, 2, 3, 4,...

l : integer

l is the Angular Momentum Quantum Number with values ranging from 0 to $n-1$.

m : integer

m is the Magnetic Quantum Number with values ranging from $-l$ to l .

r :

radial coordinate

phi :

azimuthal angle

theta :

polar angle

Z :

atomic number (1 for Hydrogen, 2 for Helium, ...)

Everything is in Hartree atomic units.

Examples

```

>>> from sympy.physics.hydrogen import Psi_nlm
>>> from sympy import Symbol
>>> r=Symbol("r", positive=True)
>>> phi=Symbol("phi", real=True)
>>> theta=Symbol("theta", real=True)
>>> Z=Symbol("Z", positive=True, integer=True, nonzero=True)
>>> Psi_nlm(1,0,0,r,phi,theta,Z)

```

(continues on next page)

(continued from previous page)

```

Z**(3/2)*exp(-Z*r)/sqrt(pi)
>>> Psi_nlm(2,1,1,r,phi,theta,Z)
-Z**(5/2)*r*exp(I*phi)*exp(-Z*r/2)*sin(theta)/(8*sqrt(pi))

```

Integrating the absolute square of a hydrogen wavefunction `psi_{nlm}` over the whole space leads 1.

The normalization of the hydrogen wavefunctions `Psi_nlm` is:

```

>>> from sympy import integrate, conjugate, pi, oo, sin
>>> wf=Psi_nlm(2,1,1,r,phi,theta,Z)
>>> abs_sqrd=wf*conjugate(wf)
>>> jacobi=r**2*sin(theta)
>>> integrate(abs_sqrd*jacobi, (r,0,oo), (phi,0,2*pi), (theta,0,pi))
1

```

`sympy.physics.hydrogen.R_nl(n, l, r, Z=1)`

Returns the Hydrogen radial wavefunction `R_{nl}`.

Parameters

n : integer

Principal Quantum Number which is an integer with possible values as 1, 2, 3, 4,...

l : integer

l is the Angular Momentum Quantum Number with values ranging from 0 to n-1.

r :

Radial coordinate.

Z :

Atomic number (1 for Hydrogen, 2 for Helium, ...)

Everything is in Hartree atomic units.

Examples

```

>>> from sympy.physics.hydrogen import R_nl
>>> from sympy.abc import r, Z
>>> R_nl(1, 0, r, Z)
2*sqrt(Z**3)*exp(-Z*r)
>>> R_nl(2, 0, r, Z)
sqrt(2)*(-Z*r + 2)*sqrt(Z**3)*exp(-Z*r/2)/4
>>> R_nl(2, 1, r, Z)
sqrt(6)*Z*r*sqrt(Z**3)*exp(-Z*r/2)/12

```

For Hydrogen atom, you can just use the default value of `Z=1`:

```

>>> R_nl(1, 0, r)
2*exp(-r)
>>> R_nl(2, 0, r)

```

(continues on next page)

(continued from previous page)

```
sqrt(2)*(2 - r)*exp(-r/2)/4
>>> R_nl(3, 0, r)
2*sqrt(3)*(2*r**2/9 - 2*r + 3)*exp(-r/3)/27
```

For Silver atom, you would use $Z=47$:

```
>>> R_nl(1, 0, r, Z=47)
94*sqrt(47)*exp(-47*r)
>>> R_nl(2, 0, r, Z=47)
47*sqrt(94)*(2 - 47*r)*exp(-47*r/2)/4
>>> R_nl(3, 0, r, Z=47)
94*sqrt(141)*(4418*r**2/9 - 94*r + 3)*exp(-47*r/3)/27
```

The normalization of the radial wavefunction is:

```
>>> from sympy import integrate, oo
>>> integrate(R_nl(1, 0, r)**2 * r**2, (r, 0, oo))
1
>>> integrate(R_nl(2, 0, r)**2 * r**2, (r, 0, oo))
1
>>> integrate(R_nl(2, 1, r)**2 * r**2, (r, 0, oo))
1
```

It holds for any atomic number:

```
>>> integrate(R_nl(1, 0, r, Z=2)**2 * r**2, (r, 0, oo))
1
>>> integrate(R_nl(2, 0, r, Z=3)**2 * r**2, (r, 0, oo))
1
>>> integrate(R_nl(2, 1, r, Z=4)**2 * r**2, (r, 0, oo))
1
```

Matrices

Known matrices related to physics

`sympy.physics.matrices.mdfn(n)`

Deprecated since version 1.9: Use DFT from `sympy.matrices.expressions.fourier` instead.

To get identical behavior to `mdfn(n)`, use `DFT(n).as_explicit()`.

`sympy.physics.matrices.mgamma(mu, lower=False)`

Returns a Dirac gamma matrix γ^μ in the standard (Dirac) representation.

Explanation

If you want γ_μ , use `gamma(mu, True)`.

We use a convention:

$$\gamma^5 = i \cdot \gamma^0 \cdot \gamma^1 \cdot \gamma^2 \cdot \gamma^3$$

$$\gamma_5 = i \cdot \gamma_0 \cdot \gamma_1 \cdot \gamma_2 \cdot \gamma_3 = -\gamma^5$$

Examples

```
>>> from sympy.physics.matrices import mgamma
>>> mgamma(1)
Matrix([
[ 0,  0, 0, 1],
[ 0,  0, 1, 0],
[ 0, -1, 0, 0],
[-1,  0, 0, 0]])
```

References

[R658]

`sympy.physics.matrices.msigma(i)`

Returns a Pauli matrix σ_i with $i = 1, 2, 3$.

Examples

```
>>> from sympy.physics.matrices import msigma
>>> msigma(1)
Matrix([
[0, 1],
[1, 0]])
```

References

[R659]

`sympy.physics.matrices.pat_matrix(m, dx, dy, dz)`

Returns the Parallel Axis Theorem matrix to translate the inertia matrix a distance of (dx, dy, dz) for a body of mass m .

Examples

To translate a body having a mass of 2 units a distance of 1 unit along the x -axis we get:

```
>>> from sympy.physics.matrices import pat_matrix
>>> pat_matrix(2, 1, 0, 0)
Matrix([
[0, 0, 0],
[0, 2, 0],
[0, 0, 2]])
```

Pauli Algebra

This module implements Pauli algebra by subclassing Symbol. Only algebraic properties of Pauli matrices are used (we do not use the Matrix class).

See the documentation to the class Pauli for examples.

References

`sympy.physics.paulialgebra.evaluate_pauli_product(arg)`

Help function to evaluate Pauli matrices product with symbolic objects.

Parameters

arg: symbolic expression that contains Paulimatrices

Examples

```
>>> from sympy.physics.paulialgebra import Pauli, evaluate_pauli_product
>>> from sympy import I
>>> evaluate_pauli_product(I*Pauli(1)*Pauli(2))
-sigma3
```

```
>>> from sympy.abc import x
>>> evaluate_pauli_product(x**2*Pauli(2)*Pauli(1))
-I*x**2*sigma3
```

Quantum Harmonic Oscillator in 1-D

`sympy.physics.qho_1d.E_n(n, omega)`

Returns the Energy of the One-dimensional harmonic oscillator.

Parameters

```n```:

The “nodal” quantum number.

```omega```:

The harmonic oscillator angular frequency.

Notes

The unit of the returned value matches the unit of $\hbar\omega$, since the energy is calculated as:

$$E_n = \hbar\omega(n + 1/2)$$

Examples

```
>>> from sympy.physics.qho_1d import E_n
>>> from sympy.abc import x, omega
>>> E_n(x, omega)
hbar*omega*(x + 1/2)
```

`sympy.physics.qho_1d.coherent_state(n, alpha)`

Returns $\langle n|\alpha\rangle$ for the coherent states of 1D harmonic oscillator. See https://en.wikipedia.org/wiki/Coherent_states

Parameters

```n``` :

The “nodal” quantum number.

```alpha``` :

The eigen value of annihilation operator.

`sympy.physics.qho_1d.psi_n(n, x, m, omega)`

Returns the wavefunction $\psi_{\{n\}}$ for the One-dimensional harmonic oscillator.

Parameters

```n``` :

the “nodal” quantum number. Corresponds to the number of nodes in the wavefunction.  $n \geq 0$

```x``` :

x coordinate.

```m``` :

Mass of the particle.

```omega``` :

Angular frequency of the oscillator.

Examples

```
>>> from sympy.physics.qho_1d import psi_n
>>> from sympy.abc import m, x, omega
>>> psi_n(0, x, m, omega)
(m*omega)**(1/4)*exp(-m*omega*x**2/(2*hbar))/(hbar**(1/4)*pi**(1/4))
```

Quantum Harmonic Oscillator in 3-D

`sympy.physics.sho.E_nl(n, l, hw)`

Returns the Energy of an isotropic harmonic oscillator.

Parameters

```n``` :

The “nodal” quantum number.

```l``` :

The orbital angular momentum.

```hw``` :

The harmonic oscillator parameter.

### Notes

The unit of the returned value matches the unit of  $hw$ , since the energy is calculated as:

$$E_{nl} = (2*n + l + 3/2)*hw$$

### Examples

```
>>> from sympy.physics.sho import E_nl
>>> from sympy import symbols
>>> x, y, z = symbols('x, y, z')
>>> E_nl(x, y, z)
z*(2*x + y + 3/2)
```

`sympy.physics.sho.R_nl(n, l, nu, r)`

Returns the radial wavefunction  $R_{\{nl\}}$  for a 3d isotropic harmonic oscillator.

### Parameters

```n``` :

The “nodal” quantum number. Corresponds to the number of nodes in the wavefunction. $n \geq 0$

```l``` :

The quantum number for orbital angular momentum.

```nu``` :

mass-scaled frequency: $\nu = m*\omega/(2*\hbar)$ where m is the mass and ω the frequency of the oscillator. (in atomic units $\nu == \omega/2$)

```r``` :

Radial coordinate.

## Examples

```
>>> from sympy.physics.sho import R_nl
>>> from sympy.abc import r, nu, l
>>> R_nl(0, 0, 1, r)
2*2**(3/4)*exp(-r**2)/pi**(1/4)
>>> R_nl(1, 0, 1, r)
4*2**(1/4)*sqrt(3)*(3/2 - 2*r**2)*exp(-r**2)/(3*pi**(1/4))
```

$l$ ,  $nu$  and  $r$  may be symbolic:

```
>>> R_nl(0, 0, nu, r)
2*2**(3/4)*sqrt(nu**(3/2))*exp(-nu*r**2)/pi**(1/4)
>>> R_nl(0, l, 1, r)
r**l*sqrt(2**(l + 3/2)*2**(l + 2)/factorial2(2*l + 1))*exp(-r**2)/pi**(1/
↪ 4)
```

The normalization of the radial wavefunction is:

```
>>> from sympy import Integral, oo
>>> Integral(R_nl(0, 0, 1, r)**2*r**2, (r, 0, oo)).n()
1.00000000000000
>>> Integral(R_nl(1, 0, 1, r)**2*r**2, (r, 0, oo)).n()
1.00000000000000
>>> Integral(R_nl(1, 1, 1, r)**2*r**2, (r, 0, oo)).n()
1.00000000000000
```

## Second Quantization

Second quantization operators and states for bosons.

This follows the formulation of Fetter and Welecka, “Quantum Theory of Many-Particle Systems.”

**class** sympy.physics.secondquant.AnnihilateBoson( $k$ )  
Bosonic annihilation operator.

## Examples

```
>>> from sympy.physics.secondquant import B
>>> from sympy.abc import x
>>> B(x)
AnnihilateBoson(x)
```

**apply\_operator**( $state$ )

Apply state to self if self is not symbolic and state is a FockStateKet, else multiply self by state.

## Examples

```
>>> from sympy.physics.secondquant import B, BKet
>>> from sympy.abc import x, y, n
>>> B(x).apply_operator(y)
y*AnnihilateBoson(x)
>>> B(0).apply_operator(BKet((n,)))
sqrt(n)*FockStateBosonKet((n - 1,))
```

**class** sympy.physics.secondquant.AnnihilateFermion(*k*)

Fermionic annihilation operator.

**apply\_operator**(*state*)

Apply state to self if self is not symbolic and state is a FockStateKet, else multiply self by state.

## Examples

```
>>> from sympy.physics.secondquant import B, Dagger, BKet
>>> from sympy.abc import x, y, n
>>> Dagger(B(x)).apply_operator(y)
y*CreateBoson(x)
>>> B(0).apply_operator(BKet((n,)))
sqrt(n)*FockStateBosonKet((n - 1,))
```

**property** is\_only\_q\_annihilator

Always destroy a quasi-particle? (annihilate hole or annihilate particle)

## Examples

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import F
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
```

```
>>> F(a).is_only_q_annihilator
True
>>> F(i).is_only_q_annihilator
False
>>> F(p).is_only_q_annihilator
False
```

**property** is\_only\_q\_creator

Always create a quasi-particle? (create hole or create particle)

### Examples

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import F
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
```

```
>>> F(a).is_only_q_creator
False
>>> F(i).is_only_q_creator
True
>>> F(p).is_only_q_creator
False
```

#### property `is_q_annihilator`

Can we destroy a quasi-particle? (annihilate hole or annihilate particle) If so, would that be above or below the fermi surface?

### Examples

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import F
>>> a = Symbol('a', above_fermi=1)
>>> i = Symbol('i', below_fermi=1)
>>> p = Symbol('p')
```

```
>>> F(a).is_q_annihilator
1
>>> F(i).is_q_annihilator
0
>>> F(p).is_q_annihilator
1
```

#### property `is_q_creator`

Can we create a quasi-particle? (create hole or create particle) If so, would that be above or below the fermi surface?

### Examples

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import F
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
```

```
>>> F(a).is_q_creator
0
>>> F(i).is_q_creator
```

(continues on next page)



(continued from previous page)

```
-1
>>> F(p).is_q_creator
-1
```

**class** `sympy.physics.secondquant.AntiSymmetricTensor(symbol, upper, lower)`

Stores upper and lower indices in separate Tuple's.

Each group of indices is assumed to be antisymmetric.

### Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import AntiSymmetricTensor
>>> i, j = symbols('i j', below_fermi=True)
>>> a, b = symbols('a b', above_fermi=True)
>>> AntiSymmetricTensor('v', (a, i), (b, j))
AntiSymmetricTensor(v, (a, i), (b, j))
>>> AntiSymmetricTensor('v', (i, a), (b, j))
-AntiSymmetricTensor(v, (a, i), (b, j))
```

As you can see, the indices are automatically sorted to a canonical form.

#### property `lower`

Returns the lower indices.

### Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import AntiSymmetricTensor
>>> i, j = symbols('i,j', below_fermi=True)
>>> a, b = symbols('a,b', above_fermi=True)
>>> AntiSymmetricTensor('v', (a, i), (b, j))
AntiSymmetricTensor(v, (a, i), (b, j))
>>> AntiSymmetricTensor('v', (a, i), (b, j)).lower
(b, j)
```

#### property `symbol`

Returns the symbol of the tensor.

### Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import AntiSymmetricTensor
>>> i, j = symbols('i,j', below_fermi=True)
>>> a, b = symbols('a,b', above_fermi=True)
>>> AntiSymmetricTensor('v', (a, i), (b, j))
AntiSymmetricTensor(v, (a, i), (b, j))
>>> AntiSymmetricTensor('v', (a, i), (b, j)).symbol
v
```

**property upper**

Returns the upper indices.

**Examples**

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import AntiSymmetricTensor
>>> i, j = symbols('i,j', below_fermi=True)
>>> a, b = symbols('a,b', above_fermi=True)
>>> AntiSymmetricTensor('v', (a, i), (b, j))
AntiSymmetricTensor(v, (a, i), (b, j))
>>> AntiSymmetricTensor('v', (a, i), (b, j)).upper
(a, i)
```

`sympy.physics.secondquant.B`

alias of [\*AnnihilateBoson\*](#) (page 1542)

`sympy.physics.secondquant.BBra`

alias of [\*FockStateBosonBra\*](#) (page 1551)

`sympy.physics.secondquant.BKet`

alias of [\*FockStateBosonKet\*](#) (page 1552)

`sympy.physics.secondquant.Bd`

alias of [\*CreateBoson\*](#) (page 1547)

**class** `sympy.physics.secondquant.BosonicBasis`

Base class for a basis set of bosonic Fock states.

**class** `sympy.physics.secondquant.Commutator(a, b)`

The Commutator:  $[A, B] = A*B - B*A$

The arguments are ordered according to `._cmp_()`

**Examples**

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import Commutator
>>> A, B = symbols('A,B', commutative=False)
>>> Commutator(B, A)
-Commutator(A, B)
```

Evaluate the commutator with `.doit()`

```
>>> comm = Commutator(A,B); comm
Commutator(A, B)
>>> comm.doit()
A*B - B*A
```

For two second quantization operators the commutator is evaluated immediately:

```
>>> from sympy.physics.secondquant import Fd, F
>>> a = symbols('a', above_fermi=True)
>>> i = symbols('i', below_fermi=True)
>>> p,q = symbols('p,q')
```

```
>>> Commutator(Fd(a),Fd(i))
2*N0(CreateFermion(a)*CreateFermion(i))
```

But for more complicated expressions, the evaluation is triggered by a call to `.doit()`

```
>>> comm = Commutator(Fd(p)*Fd(q),F(i)); comm
Commutator(CreateFermion(p)*CreateFermion(q), AnnihilateFermion(i))
>>> comm.doit(wicks=True)
-KroneckerDelta(i, p)*CreateFermion(q) +
KroneckerDelta(i, q)*CreateFermion(p)
```

**doit(\*\*hints)**

Enables the computation of complex expressions.

### Examples

```
>>> from sympy.physics.secondquant import Commutator, F, Fd
>>> from sympy import symbols
>>> i, j = symbols('i,j', below_fermi=True)
>>> a, b = symbols('a,b', above_fermi=True)
>>> c = Commutator(Fd(a)*F(i),Fd(b)*F(j))
>>> c.doit(wicks=True)
0
```

**classmethod eval(a, b)**

The Commutator [A,B] is on canonical form if  $A < B$ .

### Examples

```
>>> from sympy.physics.secondquant import Commutator, F, Fd
>>> from sympy.abc import x
>>> c1 = Commutator(F(x), Fd(x))
>>> c2 = Commutator(Fd(x), F(x))
>>> Commutator.eval(c1, c2)
0
```

**class sympy.physics.secondquant.CreateBoson(k)**

Bosonic creation operator.

**apply\_operator(state)**

Apply state to self if self is not symbolic and state is a FockStateKet, else multiply self by state.

### Examples

```
>>> from sympy.physics.secondquant import B, Dagger, BKet
>>> from sympy.abc import x, y, n
>>> Dagger(B(x)).apply_operator(y)
y*CreateBoson(x)
>>> B(0).apply_operator(BKet((n,)))
sqrt(n)*FockStateBosonKet((n - 1,))
```

**class** sympy.physics.secondquant.CreateFermion(*k*)

Fermionic creation operator.

**apply\_operator**(*state*)

Apply state to self if self is not symbolic and state is a FockStateKet, else multiply self by state.

### Examples

```
>>> from sympy.physics.secondquant import B, Dagger, BKet
>>> from sympy.abc import x, y, n
>>> Dagger(B(x)).apply_operator(y)
y*CreateBoson(x)
>>> B(0).apply_operator(BKet((n,)))
sqrt(n)*FockStateBosonKet((n - 1,))
```

**property** is\_only\_q\_annihilator

Always destroy a quasi-particle? (annihilate hole or annihilate particle)

### Examples

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import Fd
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
```

```
>>> Fd(a).is_only_q_annihilator
False
>>> Fd(i).is_only_q_annihilator
True
>>> Fd(p).is_only_q_annihilator
False
```

**property** is\_only\_q\_creator

Always create a quasi-particle? (create hole or create particle)

## Examples

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import Fd
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
```

```
>>> Fd(a).is_only_q_creator
True
>>> Fd(i).is_only_q_creator
False
>>> Fd(p).is_only_q_creator
False
```

### property is\_q\_annihilator

Can we destroy a quasi-particle? (annihilate hole or annihilate particle) If so, would that be above or below the fermi surface?

## Examples

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import Fd
>>> a = Symbol('a', above_fermi=1)
>>> i = Symbol('i', below_fermi=1)
>>> p = Symbol('p')
```

```
>>> Fd(a).is_q_annihilator
0
>>> Fd(i).is_q_annihilator
-1
>>> Fd(p).is_q_annihilator
-1
```

### property is\_q\_creator

Can we create a quasi-particle? (create hole or create particle) If so, would that be above or below the fermi surface?

## Examples

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import Fd
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
```

```
>>> Fd(a).is_q_creator
1
>>> Fd(i).is_q_creator
```

(continues on next page)

(continued from previous page)

```
0
>>> Fd(p).is_q_creator
1
```

**class** `sympy.physics.secondquant.Dagger(arg)`  
Hermitian conjugate of creation/annihilation operators.

### Examples

```
>>> from sympy import I
>>> from sympy.physics.secondquant import Dagger, B, Bd
>>> Dagger(2*I)
-2*I
>>> Dagger(B(0))
CreateBoson(0)
>>> Dagger(Bd(0))
AnnihilateBoson(0)
```

**classmethod** `eval(arg)`  
Evaluates the Dagger instance.

### Examples

```
>>> from sympy import I
>>> from sympy.physics.secondquant import Dagger, B, Bd
>>> Dagger(2*I)
-2*I
>>> Dagger(B(0))
CreateBoson(0)
>>> Dagger(Bd(0))
AnnihilateBoson(0)
```

The `eval()` method is called automatically.

`sympy.physics.secondquant.F`  
alias of [\*AnnihilateFermion\*](#) (page 1543)

`sympy.physics.secondquant.FBra`  
alias of [\*FockStateFermionBra\*](#) (page 1552)

`sympy.physics.secondquant.FKet`  
alias of [\*FockStateFermionKet\*](#) (page 1552)

`sympy.physics.secondquant.Fd`  
alias of [\*CreateFermion\*](#) (page 1548)

**class** `sympy.physics.secondquant.FixedBosonicBasis(n_particles, n_levels)`  
Fixed particle number basis set.

## Examples

```
>>> from sympy.physics.secondquant import FixedBosonicBasis
>>> b = FixedBosonicBasis(2, 2)
>>> state = b.state(1)
>>> b
[FockState((2, 0)), FockState((1, 1)), FockState((0, 2))]
>>> state
FockStateBosonKet((1, 1))
>>> b.index(state)
1
```

**index**(*state*)

Returns the index of state in basis.

## Examples

```
>>> from sympy.physics.secondquant import FixedBosonicBasis
>>> b = FixedBosonicBasis(2, 3)
>>> b.index(b.state(3))
3
```

**state**(*i*)Returns the state that lies at index *i* of the basis

## Examples

```
>>> from sympy.physics.secondquant import FixedBosonicBasis
>>> b = FixedBosonicBasis(2, 3)
>>> b.state(3)
FockStateBosonKet((1, 0, 1))
```

**class** sympy.physics.secondquant.**FockState**(*occupations*)

Many particle Fock state with a sequence of occupation numbers.

Anywhere you can have a FockState, you can also have S.Zero. All code must check for this!

Base class to represent FockStates.

**class** sympy.physics.secondquant.**FockStateBosonBra**(*occupations*)

Describes a collection of BosonBra particles.

### Examples

```
>>> from sympy.physics.secondquant import BBra
>>> BBra([1, 2])
FockStateBosonBra((1, 2))
```

**class** sympy.physics.secondquant.**FockStateBosonKet**(*occupations*)

Many particle Fock state with a sequence of occupation numbers.

Occupation numbers can be any integer  $\geq 0$ .

### Examples

```
>>> from sympy.physics.secondquant import BKet
>>> BKet([1, 2])
FockStateBosonKet((1, 2))
```

**class** sympy.physics.secondquant.**FockStateBra**(*occupations*)

Representation of a bra.

**class** sympy.physics.secondquant.**FockStateFermionBra**(*occupations*, *fermi\_level*=0)

### Examples

```
>>> from sympy.physics.secondquant import FBra
>>> FBra([1, 2])
FockStateFermionBra((1, 2))
```

#### See also:

[\*FockStateFermionKet\*](#) (page 1552)

**class** sympy.physics.secondquant.**FockStateFermionKet**(*occupations*, *fermi\_level*=0)

Many-particle Fock state with a sequence of occupied orbits.

### Explanation

Each state can only have one particle, so we choose to store a list of occupied orbits rather than a tuple with occupation numbers (zeros and ones).

states below fermi level are holes, and are represented by negative labels in the occupation list.

For symbolic state labels, the *fermi\_level* caps the number of allowed hole- states.



## Examples

```
>>> from sympy.physics.secondquant import FKet
>>> FKet([1, 2])
FockStateFermionKet((1, 2))
```

**class** sympy.physics.secondquant.**FockStateKet**(*occupations*)

Representation of a ket.

**class** sympy.physics.secondquant.**InnerProduct**(*bra, ket*)

An unevaluated inner product between a bra and ket.

## Explanation

Currently this class just reduces things to a product of Kronecker Deltas. In the future, we could introduce abstract states like  $|a\rangle$  and  $|b\rangle$ , and leave the inner product unevaluated as  $\langle a|b\rangle$ .

### property bra

Returns the bra part of the state

### property ket

Returns the ket part of the state

**class** sympy.physics.secondquant.**KroneckerDelta**(*i, j, delta\_range=None*)

The discrete, or Kronecker, delta function.

### Parameters

**i** : Number, Symbol

The first index of the delta function.

**j** : Number, Symbol

The second index of the delta function.

## Explanation

A function that takes in two integers  $i$  and  $j$ . It returns 0 if  $i$  and  $j$  are not equal, or it returns 1 if  $i$  and  $j$  are equal.

## Examples

An example with integer indices:

```
>>> from sympy import KroneckerDelta
>>> KroneckerDelta(1, 2)
0
>>> KroneckerDelta(3, 3)
1
```

Symbolic indices:

```
>>> from sympy.abc import i, j, k
>>> KroneckerDelta(i, j)
KroneckerDelta(i, j)
>>> KroneckerDelta(i, i)
1
>>> KroneckerDelta(i, i + 1)
0
>>> KroneckerDelta(i, i + 1 + k)
KroneckerDelta(i, i + k + 1)
```

**See also:**

*eval* (page 1554), *DiracDelta* (page 450)

**References**

[R691]

**classmethod** `eval(i, j, delta_range=None)`

Evaluates the discrete delta function.

**Examples**

```
>>> from sympy import KroneckerDelta
>>> from sympy.abc import i, j, k
```

```
>>> KroneckerDelta(i, j)
KroneckerDelta(i, j)
>>> KroneckerDelta(i, i)
1
>>> KroneckerDelta(i, i + 1)
0
>>> KroneckerDelta(i, i + 1 + k)
KroneckerDelta(i, i + k + 1)
```

# indirect doctest

**property** `indices_contain_equal_information`

Returns True if indices are either both above or below fermi.

**Examples**

```
>>> from sympy import KroneckerDelta, Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, q).indices_contain_equal_information
True
```

(continues on next page)

(continued from previous page)

```
>>> KroneckerDelta(p, q+1).indices_contain_equal_information
True
>>> KroneckerDelta(i, p).indices_contain_equal_information
False
```

**property `is_above_fermi`**

True if Delta can be non-zero above fermi.

**Examples**

```
>>> from sympy import KroneckerDelta, Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_above_fermi
True
>>> KroneckerDelta(p, i).is_above_fermi
False
>>> KroneckerDelta(p, q).is_above_fermi
True
```

**See also:**

[`is\_below\_fermi`](#) (page 1555), [`is\_only\_below\_fermi`](#) (page 1556),  
[`is\_only\_above\_fermi`](#) (page 1555)

**property `is_below_fermi`**

True if Delta can be non-zero below fermi.

**Examples**

```
>>> from sympy import KroneckerDelta, Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_below_fermi
False
>>> KroneckerDelta(p, i).is_below_fermi
True
>>> KroneckerDelta(p, q).is_below_fermi
True
```

**See also:**

[`is\_above\_fermi`](#) (page 1555), [`is\_only\_above\_fermi`](#) (page 1555),  
[`is\_only\_below\_fermi`](#) (page 1556)

**property `is_only_above_fermi`**

True if Delta is restricted to above fermi.

### Examples

```
>>> from sympy import KroneckerDelta, Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_only_above_fermi
True
>>> KroneckerDelta(p, q).is_only_above_fermi
False
>>> KroneckerDelta(p, i).is_only_above_fermi
False
```

#### See also:

[\*is\\_above\\_fermi\*](#) (page 1555), [\*is\\_below\\_fermi\*](#) (page 1555), [\*is\\_only\\_below\\_fermi\*](#) (page 1556)

#### property `is_only_below_fermi`

True if Delta is restricted to below fermi.

### Examples

```
>>> from sympy import KroneckerDelta, Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, i).is_only_below_fermi
True
>>> KroneckerDelta(p, q).is_only_below_fermi
False
>>> KroneckerDelta(p, a).is_only_below_fermi
False
```

#### See also:

[\*is\\_above\\_fermi\*](#) (page 1555), [\*is\\_below\\_fermi\*](#) (page 1555), [\*is\\_only\\_above\\_fermi\*](#) (page 1555)

#### property `killable_index`

Returns the index which is preferred to substitute in the final expression.

## Explanation

The index to substitute is the index with less information regarding fermi level. If indices contain the same information, 'a' is preferred before 'b'.

## Examples

```
>>> from sympy import KroneckerDelta, Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> j = Symbol('j', below_fermi=True)
>>> p = Symbol('p')
>>> KroneckerDelta(p, i).killable_index
p
>>> KroneckerDelta(p, a).killable_index
p
>>> KroneckerDelta(i, j).killable_index
j
```

### See also:

[\*preferred\\_index\*](#) (page 1557)

### property `preferred_index`

Returns the index which is preferred to keep in the final expression.

## Explanation

The preferred index is the index with more information regarding fermi level. If indices contain the same information, 'a' is preferred before 'b'.

## Examples

```
>>> from sympy import KroneckerDelta, Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> j = Symbol('j', below_fermi=True)
>>> p = Symbol('p')
>>> KroneckerDelta(p, i).preferred_index
i
>>> KroneckerDelta(p, a).preferred_index
a
>>> KroneckerDelta(i, j).preferred_index
i
```

### See also:

[\*killable\\_index\*](#) (page 1556)

**class** `sympy.physics.secondquant.NO(arg)`

This Object is used to represent normal ordering brackets.

i.e. {abcd} sometimes written :abcd:

## Explanation

Applying the function `NO(arg)` to an argument means that all operators in the argument will be assumed to anticommute, and have vanishing contractions. This allows an immediate reordering to canonical form upon object creation.

## Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import NO, F, Fd
>>> p,q = symbols('p,q')
>>> NO(Fd(p)*F(q))
NO(CreateFermion(p)*AnnihilateFermion(q))
>>> NO(F(q)*Fd(p))
-NO(CreateFermion(p)*AnnihilateFermion(q))
```

## Note

If you want to generate a normal ordered equivalent of an expression, you should use the function `wicks()`. This class only indicates that all operators inside the brackets anticommute, and have vanishing contractions. Nothing more, nothing less.

**`doit(**hints)`**

Either removes the brackets or enables complex computations in its arguments.

## Examples

```
>>> from sympy.physics.secondquant import NO, Fd, F
>>> from textwrap import fill
>>> from sympy import symbols, Dummy
>>> p,q = symbols('p,q', cls=Dummy)
>>> print(fill(str(NO(Fd(p)*F(q)).doit())))
KroneckerDelta(_a, _p)*KroneckerDelta(_a,
_q)*CreateFermion(_a)*AnnihilateFermion(_a) + KroneckerDelta(_a,
_p)*KroneckerDelta(_i, _q)*CreateFermion(_a)*AnnihilateFermion(_i) -
KroneckerDelta(_a, _q)*KroneckerDelta(_i,
_p)*AnnihilateFermion(_a)*CreateFermion(_i) - KroneckerDelta(_i,
_p)*KroneckerDelta(_i, _q)*AnnihilateFermion(_i)*CreateFermion(_i)
```

**`get_subNO(i)`**

Returns a `NO()` without `FermionicOperator` at index `i`.

## Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import F, NO
>>> p, q, r = symbols('p,q,r')
```

```
>>> NO(F(p)*F(q)*F(r)).get_subNO(1)
NO(AnnihilateFermion(p)*AnnihilateFermion(r))
```

### property `has_q_annihilators`

Return 0 if the rightmost argument of the first argument is a not a `q_annihilator`, else 1 if it is above fermi or -1 if it is below fermi.

## Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import NO, F, Fd
```

```
>>> a = symbols('a', above_fermi=True)
>>> i = symbols('i', below_fermi=True)
>>> NO(Fd(a)*Fd(i)).has_q_annihilators
-1
>>> NO(F(i)*F(a)).has_q_annihilators
1
>>> NO(Fd(a)*F(i)).has_q_annihilators
0
```

### property `has_q_creators`

Return 0 if the leftmost argument of the first argument is a not a `q_creator`, else 1 if it is above fermi or -1 if it is below fermi.

## Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import NO, F, Fd
```

```
>>> a = symbols('a', above_fermi=True)
>>> i = symbols('i', below_fermi=True)
>>> NO(Fd(a)*Fd(i)).has_q_creators
1
>>> NO(F(i)*F(a)).has_q_creators
-1
>>> NO(Fd(i)*F(a)).has_q_creators
0
```

### `iter_q_annihilators()`

Iterates over the annihilation operators.

### Examples

```
>>> from sympy import symbols
>>> i, j = symbols('i j', below_fermi=True)
>>> a, b = symbols('a b', above_fermi=True)
>>> from sympy.physics.secondquant import NO, F, Fd
>>> no = NO(Fd(a)*F(i)*F(b)*Fd(j))
```

```
>>> no.iter_q_creators()
<generator object... at 0x...>
>>> list(no.iter_q_creators())
[0, 1]
>>> list(no.iter_q_annihilators())
[3, 2]
```

#### `iter_q_creators()`

Iterates over the creation operators.

### Examples

```
>>> from sympy import symbols
>>> i, j = symbols('i j', below_fermi=True)
>>> a, b = symbols('a b', above_fermi=True)
>>> from sympy.physics.secondquant import NO, F, Fd
>>> no = NO(Fd(a)*F(i)*F(b)*Fd(j))
```

```
>>> no.iter_q_creators()
<generator object... at 0x...>
>>> list(no.iter_q_creators())
[0, 1]
>>> list(no.iter_q_annihilators())
[3, 2]
```

#### `class sympy.physics.secondquant.PermutationOperator(i, j)`

Represents the index permutation operator  $P(ij)$ .

$P(ij)*f(i)*g(j) = f(i)*g(j) - f(j)*g(i)$

#### `get_permuted(expr)`

Returns  $-expr$  with permuted indices.

### Explanation

```
>>> from sympy import symbols, Function
>>> from sympy.physics.secondquant import PermutationOperator
>>> p, q = symbols('p, q')
>>> f = Function('f')
>>> PermutationOperator(p, q).get_permuted(f(p, q))
-f(q, p)
```



**class** `sympy.physics.secondquant.VarBosonicBasis(n_max)`

A single state, variable particle number basis set.

### Examples

```
>>> from sympy.physics.secondquant import VarBosonicBasis
>>> b = VarBosonicBasis(5)
>>> b
[FockState((0,)), FockState((1,)), FockState((2,)),
 FockState((3,)), FockState((4,))]
```

**index(*state*)**

Returns the index of state in basis.

### Examples

```
>>> from sympy.physics.secondquant import VarBosonicBasis
>>> b = VarBosonicBasis(3)
>>> state = b.state(1)
>>> b
[FockState((0,)), FockState((1,)), FockState((2,))]
>>> state
FockStateBosonKet((1,))
>>> b.index(state)
1
```

**state(*i*)**

The state of a single basis.

### Examples

```
>>> from sympy.physics.secondquant import VarBosonicBasis
>>> b = VarBosonicBasis(5)
>>> b.state(3)
FockStateBosonKet((3,))
```

**`sympy.physics.secondquant.apply_operators(e)`**

Take a SymPy expression with operators and states and apply the operators.

### Examples

```
>>> from sympy.physics.secondquant import apply_operators
>>> from sympy import sympify
>>> apply_operators(sympify(3)+4)
7
```

`sympy.physics.secondquant.contraction(a, b)`

Calculates contraction of Fermionic operators a and b.

### Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import F, Fd, contraction
>>> p, q = symbols('p,q')
>>> a, b = symbols('a,b', above_fermi=True)
>>> i, j = symbols('i,j', below_fermi=True)
```

A contraction is non-zero only if a quasi-creator is to the right of a quasi-annihilator:

```
>>> contraction(F(a), Fd(b))
KroneckerDelta(a, b)
>>> contraction(Fd(i), F(j))
KroneckerDelta(i, j)
```

For general indices a non-zero result restricts the indices to below/above the fermi surface:

```
>>> contraction(Fd(p), F(q))
KroneckerDelta(_i, q)*KroneckerDelta(p, q)
>>> contraction(F(p), Fd(q))
KroneckerDelta(_a, q)*KroneckerDelta(p, q)
```

Two creators or two annihilators always vanishes:

```
>>> contraction(F(p), F(q))
0
>>> contraction(Fd(p), Fd(q))
0
```

`sympy.physics.secondquant.evaluate_deltas(e)`

We evaluate KroneckerDelta symbols in the expression assuming Einstein summation.

## Explanation

If one index is repeated it is summed over and in effect substituted with the other one. If both indices are repeated we substitute according to what is the preferred index. This is determined by `KroneckerDelta.preferred_index` and `KroneckerDelta.killable_index`.

In case there are no possible substitutions or if a substitution would imply a loss of information, nothing is done.

In case an index appears in more than one `KroneckerDelta`, the resulting substitution depends on the order of the factors. Since the ordering is platform dependent, the literal expression resulting from this function may be hard to predict.

## Examples

We assume the following:

```
>>> from sympy import symbols, Function, Dummy, KroneckerDelta
>>> from sympy.physics.secondquant import evaluate_deltas
>>> i,j = symbols('i j', below_fermi=True, cls=Dummy)
>>> a,b = symbols('a b', above_fermi=True, cls=Dummy)
>>> p,q = symbols('p q', cls=Dummy)
>>> f = Function('f')
>>> t = Function('t')
```

The order of preference for these indices according to `KroneckerDelta` is (a, b, i, j, p, q).

Trivial cases:

```
>>> evaluate_deltas(KroneckerDelta(i,j)*f(i)) # d_ij f(i) -> f(j)
f(_j)
>>> evaluate_deltas(KroneckerDelta(i,j)*f(j)) # d_ij f(j) -> f(i)
f(_i)
>>> evaluate_deltas(KroneckerDelta(i,p)*f(p)) # d_ip f(p) -> f(i)
f(_i)
>>> evaluate_deltas(KroneckerDelta(q,p)*f(p)) # d_qp f(p) -> f(q)
f(_q)
>>> evaluate_deltas(KroneckerDelta(q,p)*f(q)) # d_qp f(q) -> f(p)
f(_p)
```

More interesting cases:

```
>>> evaluate_deltas(KroneckerDelta(i,p)*t(a,i)*f(p,q))
f(_i, _q)*t(_a, _i)
>>> evaluate_deltas(KroneckerDelta(a,p)*t(a,i)*f(p,q))
f(_a, _q)*t(_a, _i)
>>> evaluate_deltas(KroneckerDelta(p,q)*f(p,q))
f(_p, _p)
```

Finally, here are some cases where nothing is done, because that would imply a loss of information:

```
>>> evaluate_deltas(KroneckerDelta(i,p)*f(q))
f(_q)*KroneckerDelta(_i, _p)
```

(continues on next page)

(continued from previous page)

```
>>> evaluate_deltas(KroneckerDelta(i,p)*f(i))
f(_i)*KroneckerDelta(_i, _p)
```

`sympy.physics.secondquant.matrix_rep(op, basis)`  
Find the representation of an operator in a basis.

### Examples

```
>>> from sympy.physics.secondquant import VarBosonicBasis, B, matrix_rep
>>> b = VarBosonicBasis(5)
>>> o = B(0)
>>> matrix_rep(o, b)
Matrix([
[0, 1, 0, 0, 0],
[0, 0, sqrt(2), 0, 0],
[0, 0, 0, sqrt(3), 0],
[0, 0, 0, 0, 2],
[0, 0, 0, 0, 0]])
```

`sympy.physics.secondquant.simplify_index_permutations(expr, permutation_operators)`  
Performs simplification by introducing PermutationOperators where appropriate.

### Explanation

#### Schematically:

$$[abij] - [abji] - [baij] + [baji] \rightarrow P(ab)*P(ij)*[abij]$$

`permutation_operators` is a list of PermutationOperators to consider.

If `permutation_operators=[P(ab),P(ij)]` we will try to introduce the permutation operators `P(ij)` and `P(ab)` in the expression. If there are other possible simplifications, we ignore them.

```
>>> from sympy import symbols, Function
>>> from sympy.physics.secondquant import simplify_index_permutations
>>> from sympy.physics.secondquant import PermutationOperator
>>> p,q,r,s = symbols('p,q,r,s')
>>> f = Function('f')
>>> g = Function('g')
```

```
>>> expr = f(p)*g(q) - f(q)*g(p); expr
f(p)*g(q) - f(q)*g(p)
>>> simplify_index_permutations(expr,[PermutationOperator(p,q)])
f(p)*g(q)*PermutationOperator(p, q)
```

```
>>> PermutList = [PermutationOperator(p,q),PermutationOperator(r,s)]
>>> expr = f(p,r)*g(q,s) - f(q,r)*g(p,s) + f(q,s)*g(p,r) - f(p,s)*g(q,r)
>>> simplify_index_permutations(expr,PermutList)
f(p, r)*g(q, s)*PermutationOperator(p, q)*PermutationOperator(r, s)
```

```
sympy.physics.secondquant.substitute_dummies(expr, new_indices=False,
 pretty_indices={})
```

Collect terms by substitution of dummy variables.

### Explanation

This routine allows simplification of Add expressions containing terms which differ only due to dummy variables.

The idea is to substitute all dummy variables consistently depending on the structure of the term. For each term, we obtain a sequence of all dummy variables, where the order is determined by the index range, what factors the index belongs to and its position in each factor. See `_get_ordered_dummies()` for more information about the sorting of dummies. The index sequence is then substituted consistently in each term.

### Examples

```
>>> from sympy import symbols, Function, Dummy
>>> from sympy.physics.secondquant import substitute_dummies
>>> a,b,c,d = symbols('a b c d', above_fermi=True, cls=Dummy)
>>> i,j = symbols('i j', below_fermi=True, cls=Dummy)
>>> f = Function('f')
```

```
>>> expr = f(a,b) + f(c,d); expr
f(_a, _b) + f(_c, _d)
```

Since a, b, c and d are equivalent summation indices, the expression can be simplified to a single term (for which the dummy indices are still summed over)

```
>>> substitute_dummies(expr)
2*f(_a, _b)
```

Controlling output:

By default the dummy symbols that are already present in the expression will be reused in a different permutation. However, if `new_indices=True`, new dummies will be generated and inserted. The keyword 'pretty\_indices' can be used to control this generation of new symbols.

By default the new dummies will be generated on the form `i_1`, `i_2`, `a_1`, etc. If you supply a dictionary with key:value pairs in the form:

```
{ index_group: string_of_letters }
```

The letters will be used as labels for the new dummy symbols. The index\_groups must be one of 'above', 'below' or 'general'.

```
>>> expr = f(a,b,i,j)
>>> my_dummies = { 'above':'st', 'below':'uv' }
>>> substitute_dummies(expr, new_indices=True, pretty_indices=my_dummies)
f(_s, _t, _u, _v)
```

If we run out of letters, or if there is no keyword for some index\_group the default dummy generator will be used as a fallback:

```
>>> p,q = symbols('p q', cls=Dummy) # general indices
>>> expr = f(p,q)
>>> substitute_dummies(expr, new_indices=True, pretty_indices=my_dummies)
f(_p_0, _p_1)
```

`sympy.physics.secondquant.wicks(e, **kw_args)`

Returns the normal ordered equivalent of an expression using Wicks Theorem.

### Examples

```
>>> from sympy import symbols, Dummy
>>> from sympy.physics.secondquant import wicks, F, Fd
>>> p, q, r = symbols('p,q,r')
>>> wicks(Fd(p)*F(q))
KroneckerDelta(_i, q)*KroneckerDelta(p, q) +
↳ NO(CreateFermion(p)*AnnihilateFermion(q))
```

By default, the expression is expanded:

```
>>> wicks(F(p)*(F(q)+F(r)))
NO(AnnihilateFermion(p)*AnnihilateFermion(q)) +
↳ NO(AnnihilateFermion(p)*AnnihilateFermion(r))
```

With the keyword 'keep\_only\_fully\_contracted=True', only fully contracted terms are returned.

**By request, the result can be simplified in the following order:**

- KroneckerDelta functions are evaluated
- Dummy variables are substituted consistently across terms

```
>>> p, q, r = symbols('p q r', cls=Dummy)
>>> wicks(Fd(p)*(F(q)+F(r)), keep_only_fully_contracted=True)
KroneckerDelta(_i, _q)*KroneckerDelta(_p, _q) + KroneckerDelta(_i, _
↳ r)*KroneckerDelta(_p, _r)
```

### Wigner Symbols

Wigner, Clebsch-Gordan, Racah, and Gaunt coefficients

Collection of functions for calculating Wigner 3j, 6j, 9j, Clebsch-Gordan, Racah as well as Gaunt coefficients exactly, all evaluating to a rational number times the square root of a rational number [Rasch03].

Please see the description of the individual functions for further details and examples.

## References

## Credits and Copyright

This code was taken from Sage with the permission of all authors:

<https://groups.google.com/forum/#!topic/sage-devel/M4NZdu-7O38>

## Authors

- Jens Rasch (2009-03-24): initial version for Sage
- Jens Rasch (2009-05-31): updated to sage-4.0
- Oscar Gerardo Lazo Arjona (2017-06-18): added Wigner D matrices

Copyright (C) 2008 Jens Rasch <jyr2000@gmail.com>

`sympy.physics.wigner.clebsch_gordan(j_1, j_2, j_3, m_1, m_2, m_3)`

Calculates the Clebsch-Gordan coefficient.  $\langle j_1 m_1 j_2 m_2 | j_3 m_3 \rangle$ .

The reference for this function is [Edmonds74].

### Parameters

**j\_1, j\_2, j\_3, m\_1, m\_2, m\_3 :**

Integer or half integer.

### Returns

Rational number times the square root of a rational number.

## Examples

```
>>> from sympy import S
>>> from sympy.physics.wigner import clebsch_gordan
>>> clebsch_gordan(S(3)/2, S(1)/2, 2, S(3)/2, S(1)/2, 2)
1
>>> clebsch_gordan(S(3)/2, S(1)/2, 1, S(3)/2, -S(1)/2, 1)
sqrt(3)/2
>>> clebsch_gordan(S(3)/2, S(1)/2, 1, -S(1)/2, S(1)/2, 0)
-sqrt(2)/2
```

## Notes

The Clebsch-Gordan coefficient will be evaluated via its relation to Wigner 3j symbols:

$$\langle j_1 m_1 j_2 m_2 | j_3 m_3 \rangle = (-1)^{j_1 - j_2 + m_3} \sqrt{2j_3 + 1} \text{Wigner3j}(j_1, j_2, j_3, m_1, m_2, -m_3)$$

See also the documentation on Wigner 3j symbols which exhibit much higher symmetry relations than the Clebsch-Gordan coefficient.

## Authors

- Jens Rasch (2009-03-24): initial version

`sympy.physics.wigner.dot_rot_grad_Ynm(j, p, l, m, theta, phi)`

Returns dot product of rotational gradients of spherical harmonics.

## Explanation

This function returns the right hand side of the following expression:

$$\vec{R}Y_j^p \cdot \vec{R}Y_l^m = (-1)^{m+p} \sum_{k=|l-j|}^{l+j} Y_k^{m+p} * \alpha_{l,m,j,p,k} * \frac{1}{2}(k^2 - j^2 - l^2 + k - j - l)$$

## Arguments

j, p, l, m .... indices in spherical harmonics (expressions or integers) theta, phi .... angle arguments in spherical harmonics

## Example

```
>>> from sympy import symbols
>>> from sympy.physics.wigner import dot_rot_grad_Ynm
>>> theta, phi = symbols("theta phi")
>>> dot_rot_grad_Ynm(3, 2, 2, 0, theta, phi).doit()
3*sqrt(55)*Ynm(5, 2, theta, phi)/(11*sqrt(pi))
```

`sympy.physics.wigner.gaunt(l_1, l_2, l_3, m_1, m_2, m_3, prec=None)`

Calculate the Gaunt coefficient.

### Parameters

**`l_1, l_2, l_3, m_1, m_2, m_3`** :

Integer.

**`prec - precision, default: ``None```**.

Providing a precision can drastically speed up the calculation.

### Returns

Rational number times the square root of a rational number

(if `prec=None`), or real number if a precision is given.



## Explanation

The Gaunt coefficient is defined as the integral over three spherical harmonics:

$$\begin{aligned}\text{Gaunt}(l_1, l_2, l_3, m_1, m_2, m_3) &= \int Y_{l_1, m_1}(\Omega) Y_{l_2, m_2}(\Omega) Y_{l_3, m_3}(\Omega) d\Omega \\ &= \sqrt{\frac{(2l_1 + 1)(2l_2 + 1)(2l_3 + 1)}{4\pi}} \text{Wigner3j}(l_1, l_2, l_3, 0, 0, 0) \text{Wigner3j}(l_1, l_2, l_3, m_1, m_2, m_3)\end{aligned}$$

## Examples

```
>>> from sympy.physics.wigner import gaunt
>>> gaunt(1,0,1,1,0,-1)
-1/(2*sqrt(pi))
>>> gaunt(1000,1000,1200,9,3,-12).n(64)
0.00689500421922113448...
```

It is an error to use non-integer values for  $l$  and  $m$ :

```
sage: gaunt(1.2,0,1.2,0,0,0)
Traceback (most recent call last):
...
ValueError: l values must be integer
sage: gaunt(1,0,1,1.1,0,-1.1)
Traceback (most recent call last):
...
ValueError: m values must be integer
```

## Notes

The Gaunt coefficient obeys the following symmetry rules:

- invariant under any permutation of the columns

$$\begin{aligned}Y(l_1, l_2, l_3, m_1, m_2, m_3) &= Y(l_3, l_1, l_2, m_3, m_1, m_2) \\ &= Y(l_2, l_3, l_1, m_2, m_3, m_1) \\ &= Y(l_3, l_2, l_1, m_3, m_2, m_1) \\ &= Y(l_1, l_3, l_2, m_1, m_3, m_2) \\ &= Y(l_2, l_1, l_3, m_2, m_1, m_3)\end{aligned}$$

- invariant under space inflection, i.e.

$$Y(l_1, l_2, l_3, m_1, m_2, m_3) = Y(l_1, l_2, l_3, -m_1, -m_2, -m_3)$$

- symmetric with respect to the 72 Regge symmetries as inherited for the  $3j$  symbols [Regge58]
- zero for  $l_1, l_2, l_3$  not fulfilling triangle relation
- zero for violating any one of the conditions:  $l_1 \geq |m_1|, l_2 \geq |m_2|, l_3 \geq |m_3|$
- non-zero only for an even sum of the  $l_i$ , i.e.  $L = l_1 + l_2 + l_3 = 2n$  for  $n$  in  $\mathbb{N}$

## Algorithms

This function uses the algorithm of [Liberatodebrito82] to calculate the value of the Gaunt coefficient exactly. Note that the formula contains alternating sums over large factorials and is therefore unsuitable for finite precision arithmetic and only useful for a computer algebra system [Rasch03].

## Authors

Jens Rasch (2009-03-24): initial version for Sage.

```
sympy.physics.wigner.racah(aa, bb, cc, dd, ee, ff, prec=None)
```

Calculate the Racah symbol  $W(a, b, c, d; e, f)$ .

### Parameters

**a, ..., f :**

Integer or half integer.

**prec :**

Precision, default: None. Providing a precision can drastically speed up the calculation.

### Returns

Rational number times the square root of a rational number

(if `prec=None`), or real number if a precision is given.

## Examples

```
>>> from sympy.physics.wigner import racah
>>> racah(3,3,3,3,3,3)
-1/14
```

## Notes

The Racah symbol is related to the Wigner 6j symbol:

$$\text{Wigner6j}(j_1, j_2, j_3, j_4, j_5, j_6) = (-1)^{j_1+j_2+j_4+j_5} W(j_1, j_2, j_5, j_4, j_3, j_6)$$

Please see the 6j symbol for its much richer symmetries and for additional properties.

## Algorithm

This function uses the algorithm of [Edmonds74] to calculate the value of the 6j symbol exactly. Note that the formula contains alternating sums over large factorials and is therefore unsuitable for finite precision arithmetic and only useful for a computer algebra system [Rasch03].

## Authors

- Jens Rasch (2009-03-24): initial version

`sympy.physics.wigner.wigner_3j(j_1, j_2, j_3, m_1, m_2, m_3)`

Calculate the Wigner 3j symbol  $\text{Wigner3j}(j_1, j_2, j_3, m_1, m_2, m_3)$ .

### Parameters

**j\_1, j\_2, j\_3, m\_1, m\_2, m\_3 :**

Integer or half integer.

### Returns

Rational number times the square root of a rational number.

## Examples

```
>>> from sympy.physics.wigner import wigner_3j
>>> wigner_3j(2, 6, 4, 0, 0, 0)
sqrt(715)/143
>>> wigner_3j(2, 6, 4, 0, 0, 1)
0
```

It is an error to have arguments that are not integer or half integer values:

```
sage: wigner_3j(2.1, 6, 4, 0, 0, 0)
Traceback (most recent call last):
...
ValueError: j values must be integer or half integer
sage: wigner_3j(2, 6, 4, 1, 0, -1.1)
Traceback (most recent call last):
...
ValueError: m values must be integer or half integer
```

## Notes

The Wigner 3j symbol obeys the following symmetry rules:

- invariant under any permutation of the columns (with the exception of a sign change where  $J := j_1 + j_2 + j_3$ ):

$$\begin{aligned} \text{Wigner3j}(j_1, j_2, j_3, m_1, m_2, m_3) &= \text{Wigner3j}(j_3, j_1, j_2, m_3, m_1, m_2) \\ &= \text{Wigner3j}(j_2, j_3, j_1, m_2, m_3, m_1) \\ &= (-1)^J \text{Wigner3j}(j_3, j_2, j_1, m_3, m_2, m_1) \\ &= (-1)^J \text{Wigner3j}(j_1, j_3, j_2, m_1, m_3, m_2) \\ &= (-1)^J \text{Wigner3j}(j_2, j_1, j_3, m_2, m_1, m_3) \end{aligned}$$

- invariant under space inflection, i.e.

$$\text{Wigner3j}(j_1, j_2, j_3, m_1, m_2, m_3) = (-1)^J \text{Wigner3j}(j_1, j_2, j_3, -m_1, -m_2, -m_3)$$

- symmetric with respect to the 72 additional symmetries based on the work by [Regge58]
- zero for  $j_1, j_2, j_3$  not fulfilling triangle relation
- zero for  $m_1 + m_2 + m_3 \neq 0$
- zero for violating any one of the conditions  $j_1 \geq |m_1|, j_2 \geq |m_2|, j_3 \geq |m_3|$

### Algorithm

This function uses the algorithm of [Edmonds74] to calculate the value of the 3j symbol exactly. Note that the formula contains alternating sums over large factorials and is therefore unsuitable for finite precision arithmetic and only useful for a computer algebra system [Rasch03].

### Authors

- Jens Rasch (2009-03-24): initial version

`sympy.physics.wigner.wigner_6j(j_1, j_2, j_3, j_4, j_5, j_6, prec=None)`

Calculate the Wigner 6j symbol  $\text{Wigner6j}(j_1, j_2, j_3, j_4, j_5, j_6)$ .

#### Parameters

**j\_1, ..., j\_6 :**

Integer or half integer.

**prec :**

Precision, default: None. Providing a precision can drastically speed up the calculation.

#### Returns

Rational number times the square root of a rational number  
(if `prec=None`), or real number if a precision is given.

### Examples

```
>>> from sympy.physics.wigner import wigner_6j
>>> wigner_6j(3,3,3,3,3,3)
-1/14
>>> wigner_6j(5,5,5,5,5,5)
1/52
```

It is an error to have arguments that are not integer or half integer values or do not fulfill the triangle relation:

```
sage: wigner_6j(2.5,2.5,2.5,2.5,2.5,2.5)
Traceback (most recent call last):
...
ValueError: j values must be integer or half integer and fulfill the
→triangle relation
```

(continues on next page)

(continued from previous page)

```
sage: wigner_6j(0.5,0.5,1.1,0.5,0.5,1.1)
Traceback (most recent call last):
...
ValueError: j values must be integer or half integer and fulfill the
↳ triangle relation
```

## Notes

The Wigner 6j symbol is related to the Racah symbol but exhibits more symmetries as detailed below.

$$\text{Wigner6j}(j_1, j_2, j_3, j_4, j_5, j_6) = (-1)^{j_1+j_2+j_4+j_5} W(j_1, j_2, j_5, j_4, j_3, j_6)$$

The Wigner 6j symbol obeys the following symmetry rules:

- Wigner 6j symbols are left invariant under any permutation of the columns:

$$\begin{aligned} \text{Wigner6j}(j_1, j_2, j_3, j_4, j_5, j_6) &= \text{Wigner6j}(j_3, j_1, j_2, j_6, j_4, j_5) \\ &= \text{Wigner6j}(j_2, j_3, j_1, j_5, j_6, j_4) \\ &= \text{Wigner6j}(j_3, j_2, j_1, j_6, j_5, j_4) \\ &= \text{Wigner6j}(j_1, j_3, j_2, j_4, j_6, j_5) \\ &= \text{Wigner6j}(j_2, j_1, j_3, j_5, j_4, j_6) \end{aligned}$$

- They are invariant under the exchange of the upper and lower arguments in each of any two columns, i.e.

$$\text{Wigner6j}(j_1, j_2, j_3, j_4, j_5, j_6) = \text{Wigner6j}(j_1, j_5, j_6, j_4, j_2, j_3) = \text{Wigner6j}(j_4, j_2, j_6, j_1, j_5, j_3) = \text{Wigner6j}(j_4, j_5, j_6, j_1, j_2, j_3)$$

- additional 6 symmetries [Regge59] giving rise to 144 symmetries in total
- only non-zero if any triple of  $j$ 's fulfill a triangle relation

## Algorithm

This function uses the algorithm of [Edmonds74] to calculate the value of the 6j symbol exactly. Note that the formula contains alternating sums over large factorials and is therefore unsuitable for finite precision arithmetic and only useful for a computer algebra system [Rasch03].

`sympy.physics.wigner.wigner_9j(j_1, j_2, j_3, j_4, j_5, j_6, j_7, j_8, j_9, prec=None)`

Calculate the Wigner 9j symbol  $\text{Wigner9j}(j_1, j_2, j_3, j_4, j_5, j_6, j_7, j_8, j_9)$ .

### Parameters

**j\_1, ..., j\_9 :**

Integer or half integer.

**prec :** precision, default

None. Providing a precision can drastically speed up the calculation.

### Returns

Rational number times the square root of a rational number

(if `prec=None`), or real number if a precision is given.

## Examples

```
>>> from sympy.physics.wigner import wigner_9j
>>> wigner_9j(1,1,1, 1,1,1, 1,1,0, prec=64) # ==1/18
0.05555555...
```

```
>>> wigner_9j(1/2,1/2,0, 1/2,3/2,1, 0,1,1, prec=64) # ==1/6
0.1666666...
```

It is an error to have arguments that are not integer or half integer values or do not fulfill the triangle relation:

```
sage: wigner_9j(0.5,0.5,0.5, 0.5,0.5,0.5, 0.5,0.5,0.5,prec=64)
Traceback (most recent call last):
...
ValueError: j values must be integer or half integer and fulfill the
→triangle relation
sage: wigner_9j(1,1,1, 0.5,1,1.5, 0.5,1,2.5,prec=64)
Traceback (most recent call last):
...
ValueError: j values must be integer or half integer and fulfill the
→triangle relation
```

## Algorithm

This function uses the algorithm of [Edmonds74] to calculate the value of the 3j symbol exactly. Note that the formula contains alternating sums over large factorials and is therefore unsuitable for finite precision arithmetic and only useful for a computer algebra system [Rasch03].

`sympy.physics.wigner.wigner_d(J, alpha, beta, gamma)`

Return the Wigner D matrix for angular momentum J.

### Returns

A matrix representing the corresponding Euler angle rotation( in the basis of eigenvectors of  $J_z$ ).

$$\mathcal{D}_{\alpha\beta\gamma} = \exp\left(\frac{i\alpha}{\hbar}J_z\right) \exp\left(\frac{i\beta}{\hbar}J_y\right) \exp\left(\frac{i\gamma}{\hbar}J_z\right)$$

The components are calculated using the general form [Edmonds74], equation 4.1.12.

## Explanation

**J :**

An integer, half-integer, or SymPy symbol for the total angular momentum of the angular momentum space being rotated.

**alpha, beta, gamma - Real numbers representing the Euler.**

Angles of rotation about the so-called vertical, line of nodes, and figure axes. See [Edmonds74].

## Examples

The simplest possible example:

```
>>> from sympy.physics.wigner import wigner_d
>>> from sympy import Integer, symbols, pprint
>>> half = 1/Integer(2)
>>> alpha, beta, gamma = symbols("alpha, beta, gamma", real=True)
>>> pprint(wigner_d(half, alpha, beta, gamma), use_unicode=True)
```

$$\begin{bmatrix} \frac{i \cdot \alpha}{2} & \frac{i \cdot \gamma}{2} & & \frac{i \cdot \alpha}{2} & -\frac{i \cdot \gamma}{2} \\ e^{\frac{i \cdot \alpha}{2}} \cdot e^{\frac{i \cdot \gamma}{2}} \cdot \cos\left(\frac{\beta}{2}\right) & e^{\frac{i \cdot \alpha}{2}} \cdot e^{\frac{i \cdot \gamma}{2}} \cdot \sin\left(\frac{\beta}{2}\right) & & & \\ & & & & \\ -\frac{i \cdot \alpha}{2} & \frac{i \cdot \gamma}{2} & & -\frac{i \cdot \alpha}{2} & -\frac{i \cdot \gamma}{2} \\ -e^{\frac{i \cdot \alpha}{2}} \cdot e^{\frac{i \cdot \gamma}{2}} \cdot \sin\left(\frac{\beta}{2}\right) & e^{\frac{i \cdot \alpha}{2}} \cdot e^{\frac{i \cdot \gamma}{2}} \cdot \cos\left(\frac{\beta}{2}\right) & & & \end{bmatrix}$$

`sympy.physics.wigner.wigner_d_small(J, beta)`

Return the small Wigner d matrix for angular momentum J.

### Returns

A matrix representing the corresponding Euler angle rotation( in the basis of eigenvectors of  $J_z$ ).

$$[\beta] = \exp\left(\frac{i\beta}{\hbar} J_y\right)$$

The components are calculated using the general form [Edmonds74], equation 4.1.15.

## Explanation

**J**

[An integer, half-integer, or SymPy symbol for the total angular] momentum of the angular momentum space being rotated.

**beta**

[A real number representing the Euler angle of rotation about] the so-called line of nodes. See [Edmonds74].

## Examples

```
>>> from sympy import Integer, symbols, pi, pprint
>>> from sympy.physics.wigner import wigner_d_small
>>> half = 1/Integer(2)
>>> beta = symbols("beta", real=True)
>>> pprint(wigner_d_small(half, beta), use_unicode=True)
[cos(β/2) sin(β/2)]
[-sin(β/2) cos(β/2)]
```

```
>>> pprint(wigner_d_small(2*half, beta), use_unicode=True)
[cos(β) √2·sin(β/2)·cos(β/2) sin(β)]
[-√2·sin(β/2)·cos(β/2) -sin(β) + cos(β) √2·sin(β/2)·cos(β/2)]
[sin(β) -√2·sin(β/2)·cos(β/2) cos(β)]
```

From table 4 in [Edmonds74]

```
>>> pprint(wigner_d_small(half, beta).subs({beta:pi/2}), use_
→ unicode=True)
[√2 √2]
[— —]
[2 2]
[-√2 √2]
[— —]
[2 2]
```



```
>>> pprint(wigner_d_small(2*half, beta).subs({beta:pi/2}),
... use_unicode=True)

$$\begin{bmatrix} \frac{1}{2} & \frac{\sqrt{2}}{2} & \frac{1}{2} \\ -\frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} \\ \frac{1}{2} & -\frac{\sqrt{2}}{2} & \frac{1}{2} \end{bmatrix}$$

```

```
>>> pprint(wigner_d_small(3*half, beta).subs({beta:pi/2}),
... use_unicode=True)

$$\begin{bmatrix} \frac{\sqrt{2}}{4} & \frac{\sqrt{6}}{4} & \frac{\sqrt{6}}{4} & \frac{\sqrt{2}}{4} \\ -\frac{\sqrt{6}}{4} & -\frac{\sqrt{2}}{4} & \frac{\sqrt{2}}{4} & \frac{\sqrt{6}}{4} \\ \frac{\sqrt{6}}{4} & -\frac{\sqrt{2}}{4} & -\frac{\sqrt{2}}{4} & \frac{\sqrt{6}}{4} \\ -\frac{\sqrt{2}}{4} & \frac{\sqrt{6}}{4} & -\frac{\sqrt{6}}{4} & \frac{\sqrt{2}}{4} \end{bmatrix}$$

```

```
>>> pprint(wigner_d_small(4*half, beta).subs({beta:pi/2}),
... use_unicode=True)

$$\begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{\sqrt{6}}{4} & \frac{1}{2} & \frac{1}{4} \\ -\frac{1}{2} & -\frac{1}{2} & 0 & \frac{1}{2} & \frac{1}{2} \\ \frac{\sqrt{6}}{4} & 0 & -\frac{1}{2} & 0 & \frac{\sqrt{6}}{4} \\ -\frac{1}{2} & \frac{1}{2} & 0 & -\frac{1}{2} & \frac{1}{2} \\ \frac{1}{4} & -\frac{1}{2} & \frac{\sqrt{6}}{4} & -\frac{1}{2} & \frac{1}{4} \end{bmatrix}$$

```

## Unit systems

This module integrates unit systems into SymPy, allowing a user choose which system to use when doing their computations and providing utilities to display and convert units.

Unit systems are composed of units and constants, which are themselves described from dimensions and numbers, and possibly a prefix. Quantities are defined by their unit and their numerical value, with respect to the current system.

The main advantage of this implementation over the old unit module is that it divides the units in unit systems, so that the user can decide which units to use, instead of having all in the name space (for example astrophysicists can only use units with ua, Earth or Sun masses, the theoreticians will use natural system, etc.). Moreover it allows a better control over the dimensions and conversions.

Ideas about future developments can be found on the [Github wiki](#).

## Philosophy behind unit systems

### Dimensions

#### Introduction

At the root of unit systems are dimension systems, whose structure mainly determines the one of unit systems. Our definition could seem rough but they are largely sufficient for our purposes.

A dimension will be defined as a property which is measurable and assigned to a specific phenomenon. In this sense dimensions are different from pure numbers because they carry some extra-sense, and for this reason two different dimensions cannot be added. For example time or length are dimensions, but also any other things which has some sense for us, like angle, number of particles (moles...) or information (bits...).

From this point of view the only truly dimensionless quantity are pure numbers. The idea of being dimensionless is very system-dependent, as can be seen from the  $(c, \hbar, G)$ , in which all units appears to be dimensionless in the usual common sense. This is unavoidable for computability of generic unit systems (but at the end we can tell the program what is dimensionless).

Dimensions can be composed together by taking their product or their ratio (to be defined below). For example the velocity is defined as length divided by time, or we can see the length as velocity multiplied by time, depending of what we see as the more fundamental: in general we can select a set of base dimensions from which we can describe all the others.

### Group structure

After this short introduction whose aim was to introduce the dimensions from an intuitive perspective, we describe the mathematical structure. A dimension system with  $n$  independent dimensions  $\{d_i\}_{i=1,\dots,n}$  is described by a multiplicative group  $G$ :

- there an identity element 1 corresponding to pure numbers;
- the product  $D_3 = D_1 D_2$  of two elements  $D_1, D_2 \in G$  is also in  $G$ ;
- any element  $D \in G$  has an inverse  $D^{-1} \in G$ .

We denote

$$D^n = \underbrace{D \times \cdots \times D}_{n \text{ times}},$$

and by definition  $D^0 = 1$ . The  $\{d_i\}_{i=1,\dots,n}$  are called generators of the group since any element  $D \in G$  can be expressed as the product of powers of the generators:

$$D = \prod_{i=1}^n d_i^{a_i}, \quad a_i \in \mathbf{Z}.$$

The identity is given for  $a_i = 0, \forall i$ , while we recover the generator  $d_i$  for  $a_i = 1, a_j = 0, \forall j \neq i$ . This group has the following properties:

1. abelian, since the generator commutes,  $[d_i, d_j] = 0$ ;
2. countable (infinite but discrete) since the elements are indexed by the powers of the generators<sup>1</sup>.

One can change the dimension basis  $\{d'_i\}_{i=1,\dots,n}$  by taking some combination of the old generators:

$$d'_i = \prod_{j=1}^n d_j^{P_{ij}}.$$

## Linear space representation

It is possible to use the linear space  $\mathbf{Z}^n$  as a representation of the group since the power coefficients  $a_i$  carry all the information one needs (we do not distinguish between the element of the group and its representation):

$$(d_i)_j = \delta_{ij}, \quad D = \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}.$$

The change of basis to  $d'_i$  follows the usual rule of change of basis for linear space, the matrix being given by the coefficients  $P_{ij}$ , which are simply the coefficients of the new vectors in term of the old basis:

$$d'_i = P_{ij} d_j.$$

We will use this last solution in our algorithm.

## An example

In order to illustrate all this formalism, we end this section with a specific example, the MKS system (m, kg, s) with dimensions (L: length, M: mass, T: time). They are represented as (we will always sort the vectors in alphabetic order)

$$L = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad M = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \quad T = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

<sup>1</sup> In general we will consider only dimensions with a maximum coefficient, so we can only a truncation of the group; but this is not useful for the algorithm.

Other dimensions can be derived, for example velocity  $V$  or action  $A$

$$V = LT^{-1}, \quad A = ML^2T^{-2},$$
$$V = \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}, \quad A = \begin{pmatrix} 2 \\ 1 \\ -2 \end{pmatrix}.$$

We can change the basis to go to the natural system  $(m, c, \hbar)$  with dimension (L: length, V: velocity, A: action)<sup>2</sup>. In this basis the generators are

$$A = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad L = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \quad V = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix},$$

whereas the mass and time are given by

$$T = LV^{-1}, \quad M = AV^{-2},$$
$$T = \begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix}, \quad M = \begin{pmatrix} 1 \\ 0 \\ -2 \end{pmatrix}.$$

Finally the inverse change of basis matrix  $P^{-1}$  is obtained by gluing the vectors expressed in the old basis:

$$P^{-1} = \begin{pmatrix} 2 & 1 & 1 \\ 1 & 0 & 0 \\ -2 & 0 & -1 \end{pmatrix}.$$

To find the change of basis matrix we just have to take the inverse

$$P = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & -2 & -1 \end{pmatrix}.$$

## Quantities

A quantity is defined by its name, dimension and factor to a canonical quantity of the same dimension. The canonical quantities are an internal reference of the units module and should not be relevant for end-users. Both units and physical constants are quantities.

## Units

Units, such as meters, seconds and kilograms, are usually reference quantities chosen by men to refer to other quantities.

After defining several units of different dimensions we can form a unit system, which is basically a dimension system with a notion of scale.

---

<sup>2</sup> We anticipate a little by considering  $c$  and  $\hbar$  as units and not as physical constants.

## Constants

Physical constants are just quantities. They indicate that we used not to understand that two dimensions are in fact the same. For example, we see a velocity for the light different from 1 because we do not think that time is the same as space (which is normal because of our sense; but it is different at the fundamental level). For example, once there was the “heat constant” which allowed to convert between joules and calories since people did not know that heat was energy. As soon as they understood it they fixed this constant to 1 (this is a very schematic story).

We can interpret the fact that now we fix the value of fundamental constants in the SI as showing that they are units (and we use them to define the other usual units).

## The need for a reference

It is not possible to define from scratch units and unit systems: one needs to define some references, and then build the rest over them. Said in another way, we need an origin for the scales of our units (i.e. a unit with factor 1), and to be sure that all units of a given dimension are defined consistently we need to use the same origin for all of them. This can happen if we want to use a derived unit as a base units in another system: we should not define it as having a scale 1, because, even if it is inconsistent inside the system, we could not convert to the first system since we have two different units (from our point of view) of same scale (which means they are equal for the computer).

We will say that the dimensions and scales defined outside systems are canonical, because we use them for all computations. On the other side the dimensions and scales obtained with reference to a system are called physical, because they ultimately carry a sense.

Let’s use a concrete (and important) example: the case of the mass units. We would like to define the gram as the origin. We would like to define the gram as the canonical origin for the mass, so we assign it a scale 1. Then we can define a system (e.g. in chemistry) that take it as a base unit. The MKS system prefers to use the kilogram; a naive choice would be to attribute it a scale if 1 since it is a base, but we see that we could not convert to the chemistry system because g and kg have both been given the same factor. So we need to define kg as 1000 g, and only then use it as a base in MKS. But as soon as we ask the question “what is the factor of kg in MKS?”, we get the answer 1, since it is a base unit.

Thus we will define all computations without referring to a system, and it is only at the end that we can plug the result into a system to give the context we are interested in.

## Literature

## Examples

In the following sections we give few examples of what can be done with this module.

## Dimensional analysis

We will start from Newton's second law

$$ma = F$$

where  $m$ ,  $a$  and  $F$  are the mass, the acceleration and the force respectively. Knowing the dimensions of  $m$  ( $M$ ) and  $a$  ( $LT^{-2}$ ), we will determine the dimension of  $F$ ; obviously we will find that it is a force:  $MLT^{-2}$ .

From there we will use the expression of the gravitational force between the particle of mass  $m$  and the body of mass  $M$ , at a distance  $r$

$$F = \frac{GmM}{r^2}$$

to determine the dimension of the Newton's constant  $G$ . The result should be  $L^3M^{-1}T^{-2}$ .

```
>>> from sympy import symbols
>>> from sympy.physics.units.systems import SI
>>> from sympy.physics.units import length, mass, acceleration, force
>>> from sympy.physics.units import gravitational_constant as G
>>> from sympy.physics.units.systems.si import dimsys_SI
>>> F = mass*acceleration
>>> F
Dimension(acceleration*mass)
>>> dimsys_SI.get_dimensional_dependencies(F)
{Dimension(length): 1, Dimension(mass, M): 1, Dimension(time): -2}
>>> dimsys_SI.get_dimensional_dependencies(force)
{Dimension(length): 1, Dimension(mass): 1, Dimension(time): -2}
```

Dimensions cannot be compared directly, even if in the SI convention they are the same:

```
>>> F == force
False
```

Dimension system objects provide a way to test the equivalence of dimensions:

```
>>> dimsys_SI.equivalent_dims(F, force)
True
```

```
>>> m1, m2, r = symbols("m1 m2 r")
>>> grav_eq = G * m1 * m2 / r**2
>>> F2 = grav_eq.subs({m1: mass, m2: mass, r: length, G: G.dimension})
↪
>>> F2
Dimension(mass*length*time**-2)
>>> F2.get_dimensional_dependencies()
{'length': 1, 'mass': 1, 'time': -2}
```

Note that one should first solve the equation, and then substitute with the dimensions.

## Equation with quantities

Using Kepler's third law

$$\frac{T^2}{a^3} = \frac{4\pi^2}{GM}$$

we can find the Venus orbital period using the known values for the other variables (taken from Wikipedia). The result should be 224.701 days.

```
>>> from sympy import solve, symbols, pi, Eq
>>> from sympy.physics.units import Quantity, length, mass
>>> from sympy.physics.units import day, gravitational_constant as G
>>> from sympy.physics.units import meter, kilogram
>>> T = symbols("T")
>>> a = Quantity("venus_a")
```

Specify the dimension and scale in SI units:

```
>>> SI.set_quantity_dimension(a, length)
>>> SI.set_quantity_scale_factor(a, 108208000e3*meter)
```

Add the solar mass as quantity:

```
>>> M = Quantity("solar_mass")
>>> SI.set_quantity_dimension(M, mass)
>>> SI.set_quantity_scale_factor(M, 1.9891e30*kilogram)
```

Now Kepler's law:

```
>>> eq = Eq(T**2 / a**3, 4*pi**2 / G / M)
>>> eq
Eq(T**2/venus_a**3, 4*pi**2/(gravitational_constant*solar_mass))
>>> q = solve(eq, T)[1]
>>> q
2*pi*venus_a**(3/2)/(sqrt(gravitational_constant)*sqrt(solar_mass))
```

To convert to days, use the `convert_to` function (and possibly approximate the outcoming result):

```
>>> from sympy.physics.units import convert_to
>>> convert_to(q, day)
71.5112118495813*pi*day
>>> convert_to(q, day).n()
224.659097795948*day
```

We could also have the solar mass and the day as units coming from the astrophysical system, but we wanted to show how to create a unit that one needs.

We can see in this example that intermediate dimensions can be ill-defined, such as `sqrt(G)`, but one should check that the final result - when all dimensions are combined - is well defined.

## Dimensions and dimension systems

Definition of physical dimensions.

Unit systems will be constructed on top of these dimensions.

Most of the examples in the doc use MKS system and are presented from the computer point of view: from a human point, adding length to time is not legal in MKS but it is in natural system; for a computer in natural system there is no time dimension (but a velocity dimension instead) - in the basis - so the question of adding time to length has no meaning.

**class** `sympy.physics.units.dimensions.Dimension(name, symbol=None)`

This class represent the dimension of a physical quantities.

The Dimension constructor takes as parameters a name and an optional symbol.

For example, in classical mechanics we know that time is different from temperature and dimensions make this difference (but they do not provide any measure of these quantites.

```
>>> from sympy.physics.units import Dimension
>>> length = Dimension('length')
>>> length
Dimension(length)
>>> time = Dimension('time')
>>> time
Dimension(time)
```

Dimensions can be composed using multiplication, division and exponentiation (by a number) to give new dimensions. Addition and subtraction is defined only when the two objects are the same dimension.

```
>>> velocity = length / time
>>> velocity
Dimension(length/time)
```

It is possible to use a dimension system object to get the dimensionals dependencies of a dimension, for example the dimension system used by the SI units convention can be used:

```
>>> from sympy.physics.units.systems.si import dimsys_SI
>>> dimsys_SI.get_dimensional_dependencies(velocity)
{Dimension(length, L): 1, Dimension(time, T): -1}
>>> length + length
Dimension(length)
>>> l2 = length**2
>>> l2
Dimension(length**2)
>>> dimsys_SI.get_dimensional_dependencies(l2)
{Dimension(length, L): 2}
```

**has\_integer\_powers**(*dim\_sys*)

Check if the dimension object has only integer powers.

All the dimension powers should be integers, but rational powers may appear in intermediate steps. This method may be used to check that the final result is well-defined.



```
class sympy.physics.units.dimensions.DimensionSystem(base_dims, derived_dims=(),
 dimensional_dependencies={})
```

DimensionSystem represents a coherent set of dimensions.

The constructor takes three parameters:

- base dimensions;
- derived dimensions: these are defined in terms of the base dimensions (for example velocity is defined from the division of length by time);
- dependency of dimensions: how the derived dimensions depend on the base dimensions.

Optionally either the `derived_dims` or the `dimensional_dependencies` may be omitted.

**property can\_transf\_matrix**

Useless method, kept for compatibility with previous versions.

DO NOT USE.

Return the canonical transformation matrix from the canonical to the base dimension basis.

It is the inverse of the matrix computed with `inv_can_transf_matrix()`.

**property dim**

Useless method, kept for compatibility with previous versions.

DO NOT USE.

Give the dimension of the system.

That is return the number of dimensions forming the basis.

**dim\_can\_vector(dim)**

Useless method, kept for compatibility with previous versions.

DO NOT USE.

Dimensional representation in terms of the canonical base dimensions.

**dim\_vector(dim)**

Useless method, kept for compatibility with previous versions.

DO NOT USE.

Vector representation in terms of the base dimensions.

**property inv\_can\_transf\_matrix**

Useless method, kept for compatibility with previous versions.

DO NOT USE.

Compute the inverse transformation matrix from the base to the canonical dimension basis.

It corresponds to the matrix where columns are the vector of base dimensions in canonical basis.

This matrix will almost never be used because dimensions are always defined with respect to the canonical basis, so no work has to be done to get them in this basis.

Nonetheless if this matrix is not square (or not invertible) it means that we have chosen a bad basis.

**property `is_consistent`**

Useless method, kept for compatibility with previous versions.

DO NOT USE.

Check if the system is well defined.

**`is_dimensionless`(*dimension*)**

Check if the dimension object really has a dimension.

A dimension should have at least one component with non-zero power.

**property `list_can_dims`**

Useless method, kept for compatibility with previous versions.

DO NOT USE.

List all canonical dimension names.

**`print_dim_base`(*dim*)**

Give the string expression of a dimension in term of the basis symbols.

## Unit prefixes

Module defining unit prefixe class and some constants.

Constant dict for SI and binary prefixes are defined as `PREFIXES` and `BIN_PREFIXES`.

**`class` `sympy.physics.units.prefixes.Prefix`(*name*, *abbrev*, *exponent*, *base*=10, *latex\_repr*=None)**

This class represent prefixes, with their name, symbol and factor.

Prefixes are used to create derived units from a given unit. They should always be encapsulated into units.

The factor is constructed from a base (default is 10) to some power, and it gives the total multiple or fraction. For example the kilometer km is constructed from the meter (factor 1) and the kilo (10 to the power 3, i.e. 1000). The base can be changed to allow e.g. binary prefixes.

A prefix multiplied by something will always return the product of this other object times the factor, except if the other object:

- is a prefix and they can be combined into a new prefix;
- defines multiplication with prefixes (which is the case for the `Unit` class).

## Units and unit systems

Unit system for physical quantities; include definition of constants.

```
class sympy.physics.units.unitsystem.UnitSystem(base_units, units=(), name="",
 descr="", dimension_system=None,
 derived_units: Dict[Dimension
 (page 1584), Quantity (page 1587)]
 = {})
```

UnitSystem represents a coherent set of units.

A unit system is basically a dimension system with notions of scales. Many of the methods are defined in the same way.

It is much better if all base units have a symbol.

### property **dim**

Give the dimension of the system.

That is return the number of units forming the basis.

```
extend(base, units=(), name="", description="", dimension_system=None,
 derived_units: Dict[Dimension (page 1584), Quantity (page 1587)] = {})
```

Extend the current system into a new one.

Take the base and normal units of the current system to merge them to the base and normal units given in argument. If not provided, name and description are overridden by empty strings.

```
get_units_non_prefixed() → Set[Quantity (page 1587)]
```

Return the units of the system that do not have a prefix.

### property **is\_consistent**

Check if the underlying dimension system is consistent.

## Physical quantities

Physical quantities.

```
class sympy.physics.units.quantities.Quantity(name, abbrev=None,
 dimension=None,
 scale_factor=None, latex_repr=None,
 pretty_unicode_repr=None,
 pretty_ascii_repr=None,
 mathml_presentation_repr=None,
 is_prefixed=False, **assumptions)
```

Physical quantity: can be a unit of measure, a constant or a generic quantity.

### property **abbrev**

Symbol representing the unit name.

Prepend the abbreviation with the prefix symbol if it is defines.

```
convert_to(other, unit_system='SI')
```

Convert the quantity to another quantity of same dimensions.

## Examples

```
>>> from sympy.physics.units import speed_of_light, meter, second
>>> speed_of_light
speed_of_light
>>> speed_of_light.convert_to(meter/second)
299792458*meter/second
```

```
>>> from sympy.physics.units import liter
>>> liter.convert_to(meter**3)
meter**3/1000
```

### property free\_symbols

Return free symbols from quantity.

### property is\_prefixed

Whether or not the quantity is prefixed. Eg. *kilogram* is prefixed, but *gram* is not.

### property scale\_factor

Overall magnitude of the quantity as compared to the canonical units.

### set\_global\_relative\_scale\_factor(scale\_factor, reference\_quantity)

Setting a scale factor that is valid across all unit system.

## Conversion between quantities

Several methods to simplify expressions involving unit objects.

`sympy.physics.units.util.convert_to(expr, target_units, unit_system='SI')`

Convert `expr` to the same expression with all of its units and quantities represented as factors of `target_units`, whenever the dimension is compatible.

`target_units` may be a single unit/quantity, or a collection of units/quantities.

## Examples

```
>>> from sympy.physics.units import speed_of_light, meter, gram, second, _
→ day
>>> from sympy.physics.units import mile, newton, kilogram, atomic_mass_
→ constant
>>> from sympy.physics.units import kilometer, centimeter
>>> from sympy.physics.units import gravitational_constant, hbar
>>> from sympy.physics.units import convert_to
>>> convert_to(mile, kilometer)
25146*kilometer/15625
>>> convert_to(mile, kilometer).n()
1.609344*kilometer
>>> convert_to(speed_of_light, meter/second)
299792458*meter/second
>>> convert_to(day, second)
86400*second
```

(continues on next page)

(continued from previous page)

```
>>> 3*newton
3*newton
>>> convert_to(3*newton, kilogram*meter/second**2)
3*kilogram*meter/second**2
>>> convert_to(atomic_mass_constant, gram)
1.660539060e-24*gram
```

Conversion to multiple units:

```
>>> convert_to(speed_of_light, [meter, second])
299792458*meter/second
>>> convert_to(3*newton, [centimeter, gram, second])
300000*centimeter*gram/second**2
```

Conversion to Planck units:

```
>>> convert_to(atomic_mass_constant, [gravitational_constant, speed_of_
→light, hbar]).n()
7.62963087839509e-20*hbar**0.5*speed_of_light**0.5/gravitational_
→constant**0.5
```

## High energy physics

### Abstract

Contains docstrings for methods in high energy physics.

## Gamma matrices

Module to handle gamma matrices expressed as tensor objects.

## Examples

```
>>> from sympy.physics.hep.gamma_matrices import GammaMatrix as G,
→ LorentzIndex
>>> from sympy.tensor.tensor import tensor_indices
>>> i = tensor_indices('i', LorentzIndex)
>>> G(i)
GammaMatrix(i)
```

Note that there is already an instance of GammaMatrixHead in four dimensions: GammaMatrix, which is simply declare as

```
>>> from sympy.physics.hep.gamma_matrices import GammaMatrix
>>> from sympy.tensor.tensor import tensor_indices
>>> i = tensor_indices('i', LorentzIndex)
```

(continues on next page)

(continued from previous page)

```
>>> GammaMatrix(i)
GammaMatrix(i)
```

To access the metric tensor

```
>>> LorentzIndex.metric
metric(LorentzIndex, LorentzIndex)
```

`sympy.physics.hep.gamma_matrices.extract_type_tens(expression, component)`

Extract from a TensExpr all tensors with *component*.

Returns two tensor expressions:

- the first contains all Tensor of having *component*.
- the second contains all remaining.

`sympy.physics.hep.gamma_matrices.gamma_trace(t)`

trace of a single line of gamma matrices

## Examples

```
>>> from sympy.physics.hep.gamma_matrices import GammaMatrix as G,
↳ gamma_trace, LorentzIndex
>>> from sympy.tensor.tensor import tensor_indices, tensor_heads
>>> p, q = tensor_heads('p, q', [LorentzIndex])
>>> i0,i1,i2,i3,i4,i5 = tensor_indices('i0:6', LorentzIndex)
>>> ps = p(i0)*G(-i0)
>>> qs = q(i0)*G(-i0)
>>> gamma_trace(G(i0)*G(i1))
4*metric(i0, i1)
>>> gamma_trace(ps*ps) - 4*p(i0)*p(-i0)
0
>>> gamma_trace(ps*qs + ps*ps) - 4*p(i0)*p(-i0) - 4*p(i0)*q(-i0)
0
```

`sympy.physics.hep.gamma_matrices.kahane_simplify(expression)`

This function cancels contracted elements in a product of four dimensional gamma matrices, resulting in an expression equal to the given one, without the contracted gamma matrices.

### Parameters

``expression`` the tensor expression containing the gamma matrices to simplify.

## Notes

If spinor indices are given, the matrices must be given in the order given in the product.

## Algorithm

The idea behind the algorithm is to use some well-known identities, i.e., for contractions enclosing an even number of  $\gamma$  matrices

$$\gamma^\mu \gamma_{a_1} \cdots \gamma_{a_{2N}} \gamma_\mu = 2(\gamma_{a_{2N}} \gamma_{a_1} \cdots \gamma_{a_{2N-1}} + \gamma_{a_{2N-1}} \cdots \gamma_{a_1} \gamma_{a_{2N}})$$

for an odd number of  $\gamma$  matrices

$$\gamma^\mu \gamma_{a_1} \cdots \gamma_{a_{2N+1}} \gamma_\mu = -2\gamma_{a_{2N+1}} \gamma_{a_{2N}} \cdots \gamma_{a_1}$$

Instead of repeatedly applying these identities to cancel out all contracted indices, it is possible to recognize the links that would result from such an operation, the problem is thus reduced to a simple rearrangement of free gamma matrices.

## Examples

When using, always remember that the original expression coefficient has to be handled separately

```
>>> from sympy.physics.hep.gamma_matrices import GammaMatrix as G, LorentzIndex
>>> from sympy.physics.hep.gamma_matrices import kahane_simplify
>>> from sympy.tensor.tensor import tensor_indices
>>> i0, i1, i2 = tensor_indices('i0:3', LorentzIndex)
>>> ta = G(i0)*G(-i0)
>>> kahane_simplify(ta)
Matrix([
[4, 0, 0, 0],
[0, 4, 0, 0],
[0, 0, 4, 0],
[0, 0, 0, 4]])
>>> tb = G(i0)*G(i1)*G(-i0)
>>> kahane_simplify(tb)
-2*GammaMatrix(i1)
>>> t = G(i0)*G(-i0)
>>> kahane_simplify(t)
Matrix([
[4, 0, 0, 0],
[0, 4, 0, 0],
[0, 0, 4, 0],
[0, 0, 0, 4]])
>>> t = G(i0)*G(-i0)
>>> kahane_simplify(t)
Matrix([
[4, 0, 0, 0],
[0, 4, 0, 0],
[0, 0, 4, 0],
[0, 0, 0, 4]])
```

If there are no contractions, the same expression is returned

```
>>> tc = G(i0)*G(i1)
>>> kahane_simplify(tc)
GammaMatrix(i0)*GammaMatrix(i1)
```

## References

[1] Algorithm for Reducing Contracted Products of gamma Matrices, Joseph Kahane, Journal of Mathematical Physics, Vol. 9, No. 10, October 1968.

`sympy.physics.hep.gamma_matrices.simplify_gpgp(ex, sort=True)`  
simplify products  $G(i)*p(-i)*G(j)*p(-j) \rightarrow p(i)*p(-i)$

## Examples

```
>>> from sympy.physics.hep.gamma_matrices import GammaMatrix as G,
↳ LorentzIndex, simplify_gpgp
>>> from sympy.tensor.tensor import tensor_indices, tensor_heads
>>> p, q = tensor_heads('p, q', [LorentzIndex])
>>> i0,i1,i2,i3,i4,i5 = tensor_indices('i0:6', LorentzIndex)
>>> ps = p(i0)*G(-i0)
>>> qs = q(i0)*G(-i0)
>>> simplify_gpgp(ps*qs*qs)
GammaMatrix(-L_0)*p(L_0)*q(L_1)*q(-L_1)
```

## The Physics Vector Module

### Abstract

In this documentation the components of the `sympy.physics.vector` module have been discussed. [sympy.physics.vector](#) (page 1592) has been written to facilitate the operations pertaining to 3-dimensional vectors, as functions of time or otherwise, in [sympy.physics](#) (page 1533).

## References for Physics/Vector

### Guide to Vector

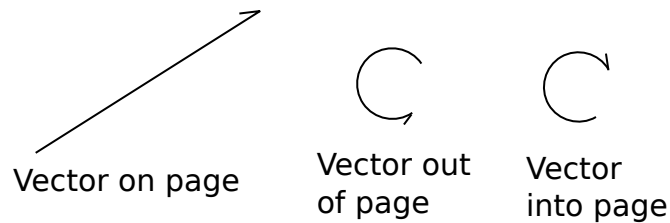
### Vector & ReferenceFrame

In [sympy.physics.vector](#) (page 1592), vectors and reference frames are the “building blocks” of dynamic systems. This document will describe these mathematically and describe how to use them with this module’s code.



## Vector

A vector is a geometric object that has a magnitude (or length) and a direction. Vectors in 3-space are often represented on paper as:



## Vector Algebra

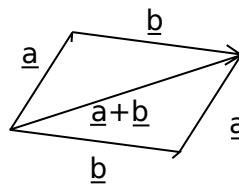
Vector algebra is the first topic to be discussed.

Two vectors are said to be equal if and only if (iff) they have the same magnitude and orientation.

## Vector Operations

Multiple algebraic operations can be done with vectors: addition between vectors, scalar multiplication, and vector multiplication.

Vector addition as based on the parallelogram law.

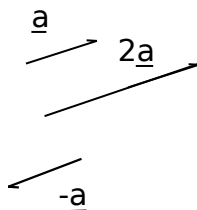


Vector addition is also commutative:

$$\mathbf{a} + \mathbf{b} = \mathbf{b} + \mathbf{a}$$

$$(\mathbf{a} + \mathbf{b}) + \mathbf{c} = \mathbf{a} + (\mathbf{b} + \mathbf{c})$$

Scalar multiplication is the product of a vector and a scalar; the result is a vector with the same orientation but whose magnitude is scaled by the scalar. Note that multiplication by -1 is equivalent to rotating the vector by 180 degrees about an arbitrary axis in the plane perpendicular to the vector.



A unit vector is simply a vector whose magnitude is equal to 1. Given any vector  $\mathbf{v}$  we can define a unit vector as:

$$\hat{\mathbf{n}}_{\mathbf{v}} = \frac{\mathbf{v}}{\|\mathbf{v}\|}$$

Note that every vector can be written as the product of a scalar and unit vector.

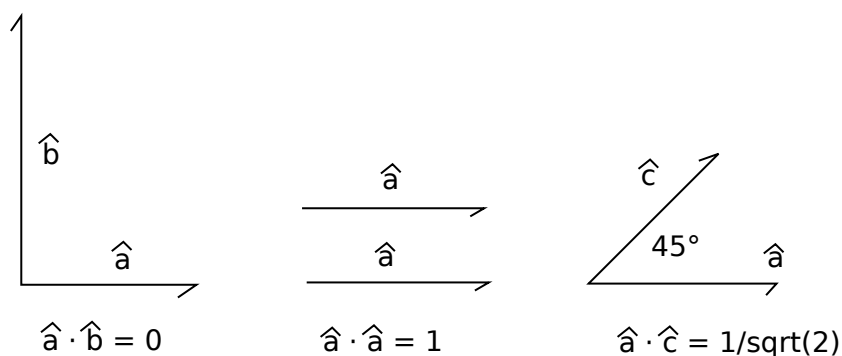
Three vector products are implemented in `sympy.physics.vector` (page 1592): the dot product, the cross product, and the outer product.

The dot product operation maps two vectors to a scalar. It is defined as:

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \|\mathbf{b}\| \cos(\theta)$$

where  $\theta$  is the angle between  $\mathbf{a}$  and  $\mathbf{b}$ .

The dot product of two unit vectors represent the magnitude of the common direction; for other vectors, it is the product of the magnitude of the common direction and the two vectors' magnitudes. The dot product of two perpendicular is zero. The figure below shows some examples:



The dot product is commutative:

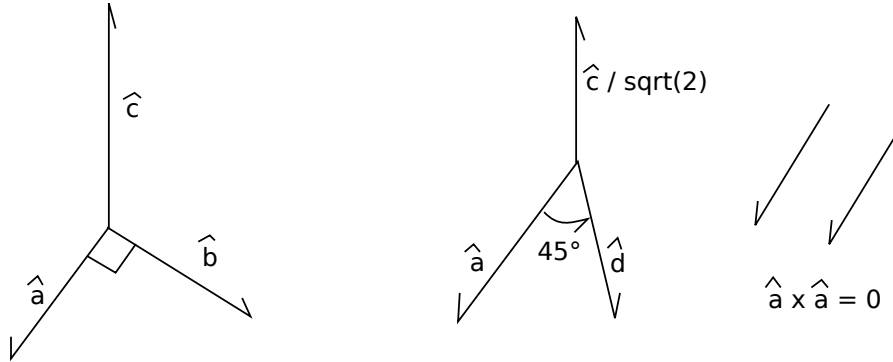
$$\mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a}$$

The cross product vector multiplication operation of two vectors returns a vector:

$$\mathbf{a} \times \mathbf{b} = \mathbf{c}$$

The vector  $\mathbf{c}$  has the following properties: it's orientation is perpendicular to both  $\mathbf{a}$  and  $\mathbf{b}$ , it's magnitude is defined as  $\|\mathbf{c}\| = \|\mathbf{a}\| \|\mathbf{b}\| \sin(\theta)$  (where  $\theta$  is the angle between  $\mathbf{a}$  and  $\mathbf{b}$ ), and

has a sense defined by using the right hand rule between  $\|\mathbf{a}\| \|\mathbf{b}\|$ . The figure below shows this:



The cross product has the following properties:

It is not commutative:

$$\begin{aligned}\mathbf{a} \times \mathbf{b} &\neq \mathbf{b} \times \mathbf{a} \\ \mathbf{a} \times \mathbf{b} &= -\mathbf{b} \times \mathbf{a}\end{aligned}$$

and not associative:

$$(\mathbf{a} \times \mathbf{b}) \times \mathbf{c} \neq \mathbf{a} \times (\mathbf{b} \times \mathbf{c})$$

Two parallel vectors will have a zero cross product.

The outer product between two vectors will not be discussed here, but instead in the inertia section (that is where it is used). Other useful vector properties and relationships are:

$$\begin{aligned}\alpha(\mathbf{a} + \mathbf{b}) &= \alpha\mathbf{a} + \alpha\mathbf{b} \\ \mathbf{a} \cdot (\mathbf{b} + \mathbf{c}) &= \mathbf{a} \cdot \mathbf{b} + \mathbf{a} \cdot \mathbf{c} \\ \mathbf{a} \times (\mathbf{b} + \mathbf{c}) &= \mathbf{a} \times \mathbf{b} + \mathbf{a} \times \mathbf{c} \\ (\mathbf{a} \times \mathbf{b}) \cdot \mathbf{c} &\text{ gives the scalar triple product.} \\ \mathbf{a} \times (\mathbf{b} \cdot \mathbf{c}) &\text{ does not work, as you cannot cross a vector and a scalar.} \\ (\mathbf{a} \times \mathbf{b}) \cdot \mathbf{c} &= \mathbf{a} \cdot (\mathbf{b} \times \mathbf{c}) \\ (\mathbf{a} \times \mathbf{b}) \cdot \mathbf{c} &= (\mathbf{b} \times \mathbf{c}) \cdot \mathbf{a} = (\mathbf{c} \times \mathbf{a}) \cdot \mathbf{b} \\ (\mathbf{a} \times \mathbf{b}) \times \mathbf{c} &= \mathbf{b}(\mathbf{a} \cdot \mathbf{c}) - \mathbf{a}(\mathbf{b} \cdot \mathbf{c}) \\ \mathbf{a} \times (\mathbf{b} \times \mathbf{c}) &= \mathbf{b}(\mathbf{a} \cdot \mathbf{c}) - \mathbf{c}(\mathbf{a} \cdot \mathbf{b})\end{aligned}$$

## Alternative Representation

If we have three non-coplanar unit vectors  $\hat{\mathbf{n}}_x, \hat{\mathbf{n}}_y, \hat{\mathbf{n}}_z$ , we can represent any vector  $\mathbf{a}$  as  $\mathbf{a} = a_x \hat{\mathbf{n}}_x + a_y \hat{\mathbf{n}}_y + a_z \hat{\mathbf{n}}_z$ . In this situation  $\hat{\mathbf{n}}_x, \hat{\mathbf{n}}_y, \hat{\mathbf{n}}_z$  are referred to as a basis.  $a_x, a_y, a_z$  are called the measure numbers. Usually the unit vectors are mutually perpendicular, in which case we can refer to them as an orthonormal basis, and they are usually right-handed.

To test equality between two vectors, now we can do the following. With vectors:

$$\begin{aligned}\mathbf{a} &= a_x \hat{\mathbf{n}}_x + a_y \hat{\mathbf{n}}_y + a_z \hat{\mathbf{n}}_z \\ \mathbf{b} &= b_x \hat{\mathbf{n}}_x + b_y \hat{\mathbf{n}}_y + b_z \hat{\mathbf{n}}_z\end{aligned}$$

We can claim equality if:  $a_x = b_x, a_y = b_y, a_z = b_z$ .

Vector addition is then represented, for the same two vectors, as:

$$\mathbf{a} + \mathbf{b} = (a_x + b_x) \hat{\mathbf{n}}_x + (a_y + b_y) \hat{\mathbf{n}}_y + (a_z + b_z) \hat{\mathbf{n}}_z$$

Multiplication operations are now defined as:

$$\begin{aligned}\alpha \mathbf{b} &= \alpha b_x \hat{\mathbf{n}}_x + \alpha b_y \hat{\mathbf{n}}_y + \alpha b_z \hat{\mathbf{n}}_z \\ \mathbf{a} \cdot \mathbf{b} &= a_x b_x + a_y b_y + a_z b_z \\ \mathbf{a} \times \mathbf{b} &= \det \begin{bmatrix} \hat{\mathbf{n}}_x & \hat{\mathbf{n}}_y & \hat{\mathbf{n}}_z \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{bmatrix} \\ (\mathbf{a} \times \mathbf{b}) \cdot \mathbf{c} &= \det \begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \end{bmatrix}\end{aligned}$$

To write a vector in a given basis, we can do the follow:

$$\mathbf{a} = (\mathbf{a} \cdot \hat{\mathbf{n}}_x) \hat{\mathbf{n}}_x + (\mathbf{a} \cdot \hat{\mathbf{n}}_y) \hat{\mathbf{n}}_y + (\mathbf{a} \cdot \hat{\mathbf{n}}_z) \hat{\mathbf{n}}_z$$

## Examples

Some numeric examples of these operations follow:

$$\begin{aligned}\mathbf{a} &= \hat{\mathbf{n}}_x + 5\hat{\mathbf{n}}_y \\ \mathbf{b} &= \hat{\mathbf{n}}_y + \alpha\hat{\mathbf{n}}_z \\ \mathbf{a} + \mathbf{b} &= \hat{\mathbf{n}}_x + 6\hat{\mathbf{n}}_y + \alpha\hat{\mathbf{n}}_z \\ \mathbf{a} \cdot \mathbf{b} &= 5 \\ \mathbf{a} \cdot \hat{\mathbf{n}}_y &= 5 \\ \mathbf{a} \cdot \hat{\mathbf{n}}_z &= 0 \\ \mathbf{a} \times \mathbf{b} &= 5\alpha\hat{\mathbf{n}}_x - \alpha\hat{\mathbf{n}}_y + \hat{\mathbf{n}}_z \\ \mathbf{b} \times \mathbf{a} &= -5\alpha\hat{\mathbf{n}}_x + \alpha\hat{\mathbf{n}}_y - \hat{\mathbf{n}}_z\end{aligned}$$

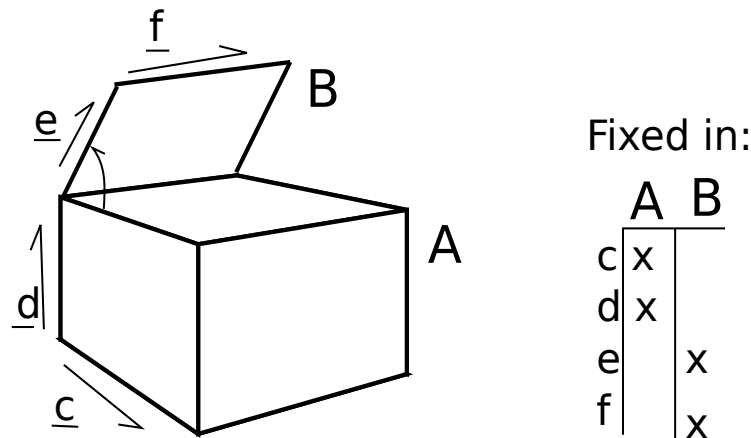
## Vector Calculus

To deal with the calculus of vectors with moving object, we have to introduce the concept of a reference frame. A classic example is a train moving along its tracks, with you and a friend inside. If both you and your friend are sitting, the relative velocity between the two of you is zero. From an observer outside the train, you will both have velocity though.

We will now apply more rigor to this definition. A reference frame is a virtual “platform” which we choose to observe vector quantities from. If we have a reference frame  $\mathbf{N}$ , vector  $\mathbf{a}$  is said to be fixed in the frame  $\mathbf{N}$  if none of its properties ever change when observed from  $\mathbf{N}$ . We will typically assign a fixed orthonormal basis vector set with each reference frame;  $\mathbf{N}$  will have  $\hat{\mathbf{n}}_x, \hat{\mathbf{n}}_y, \hat{\mathbf{n}}_z$  as its basis vectors.

## Derivatives of Vectors

A vector which is not fixed in a reference frame therefore has changing properties when observed from that frame. Calculus is the study of change, and in order to deal with the peculiarities of vectors fixed and not fixed in different reference frames, we need to be more explicit in our definitions.



In the above figure, we have vectors **c**, **d**, **e**, **f**. If one were to take the derivative of **e** with respect to  $\theta$ :

$$\frac{d\mathbf{e}}{d\theta}$$

it is not clear what the derivative is. If you are observing from frame **A**, it is clearly non-zero. If you are observing from frame **B**, the derivative is zero. We will therefore introduce the

frame as part of the derivative notation:

$\frac{{}^{\mathbf{A}}d\mathbf{e}}{d\theta} \neq 0$ , the derivative of  $\mathbf{e}$  with respect to  $\theta$  in the reference frame  $\mathbf{A}$

$\frac{{}^{\mathbf{B}}d\mathbf{e}}{d\theta} = 0$ , the derivative of  $\mathbf{e}$  with respect to  $\theta$  in the reference frame  $\mathbf{B}$

$\frac{{}^{\mathbf{A}}d\mathbf{c}}{d\theta} = 0$ , the derivative of  $\mathbf{c}$  with respect to  $\theta$  in the reference frame  $\mathbf{A}$

$\frac{{}^{\mathbf{B}}d\mathbf{c}}{d\theta} \neq 0$ , the derivative of  $\mathbf{c}$  with respect to  $\theta$  in the reference frame  $\mathbf{B}$

Here are some additional properties of derivatives of vectors in specific frames:

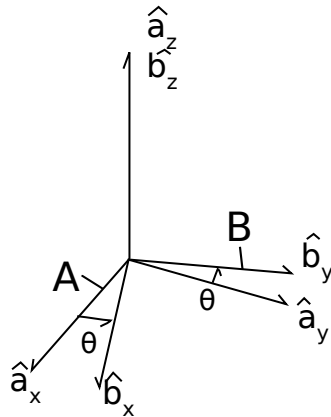
$$\begin{aligned}\frac{{}^{\mathbf{A}}d}{dt}(\mathbf{a} + \mathbf{b}) &= \frac{{}^{\mathbf{A}}d\mathbf{a}}{dt} + \frac{{}^{\mathbf{A}}d\mathbf{b}}{dt} \\ \frac{{}^{\mathbf{A}}d}{dt}\gamma\mathbf{a} &= \frac{d\gamma}{dt}\mathbf{a} + \gamma\frac{{}^{\mathbf{A}}d\mathbf{a}}{dt} \\ \frac{{}^{\mathbf{A}}d}{dt}(\mathbf{a} \times \mathbf{b}) &= \frac{{}^{\mathbf{A}}d\mathbf{a}}{dt} \times \mathbf{b} + \mathbf{a} \times \frac{{}^{\mathbf{A}}d\mathbf{b}}{dt}\end{aligned}$$

## Relating Sets of Basis Vectors

We need to now define the relationship between two different reference frames; or how to relate the basis vectors of one frame to another. We can do this using a direction cosine matrix (DCM). The direction cosine matrix relates the basis vectors of one frame to another, in the following fashion:

$$\begin{bmatrix} \hat{\mathbf{a}}_x \\ \hat{\mathbf{a}}_y \\ \hat{\mathbf{a}}_z \end{bmatrix} = [{}^{\mathbf{A}}\mathbf{C}^{\mathbf{B}}] \begin{bmatrix} \hat{\mathbf{b}}_x \\ \hat{\mathbf{b}}_y \\ \hat{\mathbf{b}}_z \end{bmatrix}$$

When two frames (say,  $\mathbf{A}$  &  $\mathbf{B}$ ) are initially aligned, then one frame has all of its basis vectors rotated around an axis which is aligned with a basis vector, we say the frames are related by a simple rotation. The figure below shows this:



The above rotation is a simple rotation about the Z axis by an angle  $\theta$ . Note that after the rotation, the basis vectors  $\hat{\mathbf{a}}_z$  and  $\hat{\mathbf{b}}_z$  are still aligned.

This rotation can be characterized by the following direction cosine matrix:

$${}^{\mathbf{A}}\mathbf{C}^{\mathbf{B}} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Simple rotations about the X and Y axes are defined by:

$$\begin{aligned} \text{DCM for x-axis rotation: } & \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix} \\ \text{DCM for y-axis rotation: } & \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \end{aligned}$$

Rotation in the positive direction here will be defined by using the right-hand rule.

The direction cosine matrix is also involved with the definition of the dot product between sets of basis vectors. If we have two reference frames with associated basis vectors, their direction cosine matrix can be defined as:

$$\begin{bmatrix} C_{xx} & C_{xy} & C_{xz} \\ C_{yx} & C_{yy} & C_{yz} \\ C_{zx} & C_{zy} & C_{zz} \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{a}}_{\mathbf{x}} \cdot \hat{\mathbf{b}}_{\mathbf{x}} & \hat{\mathbf{a}}_{\mathbf{x}} \cdot \hat{\mathbf{b}}_{\mathbf{y}} & \hat{\mathbf{a}}_{\mathbf{x}} \cdot \hat{\mathbf{b}}_{\mathbf{z}} \\ \hat{\mathbf{a}}_{\mathbf{y}} \cdot \hat{\mathbf{b}}_{\mathbf{x}} & \hat{\mathbf{a}}_{\mathbf{y}} \cdot \hat{\mathbf{b}}_{\mathbf{y}} & \hat{\mathbf{a}}_{\mathbf{y}} \cdot \hat{\mathbf{b}}_{\mathbf{z}} \\ \hat{\mathbf{a}}_{\mathbf{z}} \cdot \hat{\mathbf{b}}_{\mathbf{x}} & \hat{\mathbf{a}}_{\mathbf{z}} \cdot \hat{\mathbf{b}}_{\mathbf{y}} & \hat{\mathbf{a}}_{\mathbf{z}} \cdot \hat{\mathbf{b}}_{\mathbf{z}} \end{bmatrix}$$

Additionally, the direction cosine matrix is orthogonal, in that:

$$\begin{aligned} {}^{\mathbf{A}}\mathbf{C}^{\mathbf{B}} &= ({}^{\mathbf{B}}\mathbf{C}^{\mathbf{A}})^{-1} \\ &= ({}^{\mathbf{B}}\mathbf{C}^{\mathbf{A}})^T \end{aligned}$$

If we have reference frames  $\mathbf{A}$  and  $\mathbf{B}$ , which in this example have undergone a simple z-axis rotation by an amount  $\theta$ , we will have two sets of basis vectors. We can then define two vectors:  $\mathbf{a} = \hat{\mathbf{a}}_{\mathbf{x}} + \hat{\mathbf{a}}_{\mathbf{y}} + \hat{\mathbf{a}}_{\mathbf{z}}$  and  $\mathbf{b} = \hat{\mathbf{b}}_{\mathbf{x}} + \hat{\mathbf{b}}_{\mathbf{y}} + \hat{\mathbf{b}}_{\mathbf{z}}$ . If we wish to express  $\mathbf{b}$  in the  $\mathbf{A}$  frame, we do the following:

$$\begin{aligned} \mathbf{b} &= \hat{\mathbf{b}}_{\mathbf{x}} + \hat{\mathbf{b}}_{\mathbf{y}} + \hat{\mathbf{b}}_{\mathbf{z}} \\ \mathbf{b} &= [\hat{\mathbf{a}}_{\mathbf{x}} \cdot (\hat{\mathbf{b}}_{\mathbf{x}} + \hat{\mathbf{b}}_{\mathbf{y}} + \hat{\mathbf{b}}_{\mathbf{z}})] \hat{\mathbf{a}}_{\mathbf{x}} + [\hat{\mathbf{a}}_{\mathbf{y}} \cdot (\hat{\mathbf{b}}_{\mathbf{x}} + \hat{\mathbf{b}}_{\mathbf{y}} + \hat{\mathbf{b}}_{\mathbf{z}})] \hat{\mathbf{a}}_{\mathbf{y}} + [\hat{\mathbf{a}}_{\mathbf{z}} \cdot (\hat{\mathbf{b}}_{\mathbf{x}} + \hat{\mathbf{b}}_{\mathbf{y}} + \hat{\mathbf{b}}_{\mathbf{z}})] \hat{\mathbf{a}}_{\mathbf{z}} \\ \mathbf{b} &= (\cos(\theta) - \sin(\theta))\hat{\mathbf{a}}_{\mathbf{x}} + (\sin(\theta) + \cos(\theta))\hat{\mathbf{a}}_{\mathbf{y}} + \hat{\mathbf{a}}_{\mathbf{z}} \end{aligned}$$

And if we wish to express  $\mathbf{a}$  in the  $\mathbf{B}$ , we do:

$$\begin{aligned} \mathbf{a} &= \hat{\mathbf{a}}_{\mathbf{x}} + \hat{\mathbf{a}}_{\mathbf{y}} + \hat{\mathbf{a}}_{\mathbf{z}} \\ \mathbf{a} &= [\hat{\mathbf{b}}_{\mathbf{x}} \cdot (\hat{\mathbf{a}}_{\mathbf{x}} + \hat{\mathbf{a}}_{\mathbf{y}} + \hat{\mathbf{a}}_{\mathbf{z}})] \hat{\mathbf{b}}_{\mathbf{x}} + [\hat{\mathbf{b}}_{\mathbf{y}} \cdot (\hat{\mathbf{a}}_{\mathbf{x}} + \hat{\mathbf{a}}_{\mathbf{y}} + \hat{\mathbf{a}}_{\mathbf{z}})] \hat{\mathbf{b}}_{\mathbf{y}} + [\hat{\mathbf{b}}_{\mathbf{z}} \cdot (\hat{\mathbf{a}}_{\mathbf{x}} + \hat{\mathbf{a}}_{\mathbf{y}} + \hat{\mathbf{a}}_{\mathbf{z}})] \hat{\mathbf{b}}_{\mathbf{z}} \\ \mathbf{a} &= (\cos(\theta) + \sin(\theta))\hat{\mathbf{b}}_{\mathbf{x}} + (-\sin(\theta) + \cos(\theta))\hat{\mathbf{b}}_{\mathbf{y}} + \hat{\mathbf{b}}_{\mathbf{z}} \end{aligned}$$

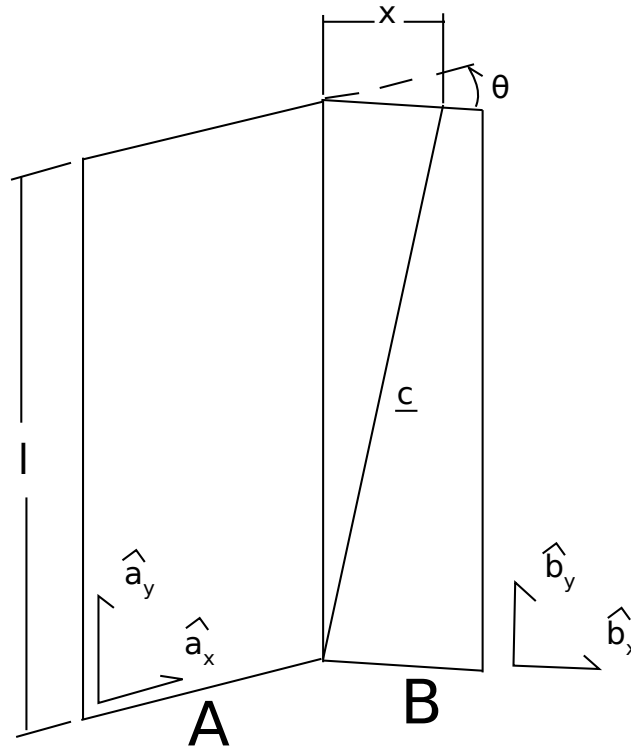
## Derivatives with Multiple Frames

If we have reference frames  $\mathbf{A}$  and  $\mathbf{B}$  we will have two sets of basis vectors. We can then define two vectors:  $\mathbf{a} = a_x \hat{\mathbf{a}}_{\mathbf{x}} + a_y \hat{\mathbf{a}}_{\mathbf{y}} + a_z \hat{\mathbf{a}}_{\mathbf{z}}$  and  $\mathbf{b} = b_x \hat{\mathbf{b}}_{\mathbf{x}} + b_y \hat{\mathbf{b}}_{\mathbf{y}} + b_z \hat{\mathbf{b}}_{\mathbf{z}}$ . If we want to take the derivative of  $\mathbf{b}$  in the reference frame  $\mathbf{A}$ , we must first express it in  $\mathbf{A}$ , and then take the derivatives of the measure numbers:

$$\frac{{}^{\mathbf{A}}d\mathbf{b}}{dx} = \frac{d(\mathbf{b} \cdot \hat{\mathbf{a}}_{\mathbf{x}})}{dx} \hat{\mathbf{a}}_{\mathbf{x}} + \frac{d(\mathbf{b} \cdot \hat{\mathbf{a}}_{\mathbf{y}})}{dx} \hat{\mathbf{a}}_{\mathbf{y}} + \frac{d(\mathbf{b} \cdot \hat{\mathbf{a}}_{\mathbf{z}})}{dx} \hat{\mathbf{a}}_{\mathbf{z}} +$$

## Examples

An example of vector calculus:



In this example we have two bodies, each with an attached reference frame. We will say that  $\theta$  and  $x$  are functions of time. We wish to know the time derivative of vector  $\underline{c}$  in both the **A** and **B** frames.

First, we need to define  $\underline{c}$ ;  $\underline{c} = x\hat{\mathbf{b}}_x + l\hat{\mathbf{b}}_y$ . This provides a definition in the **B** frame. We can now do the following:

$$\begin{aligned}\frac{{}^{\mathbf{B}}d\underline{\mathbf{c}}}{dt} &= \frac{dx}{dt}\hat{\mathbf{b}}_x + \frac{dl}{dt}\hat{\mathbf{b}}_y \\ &= \dot{x}\hat{\mathbf{b}}_x\end{aligned}$$

To take the derivative in the **A** frame, we have to first relate the two frames:

$${}^{\mathbf{A}}\mathbf{C}^{\mathbf{B}} = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$



Now we can do the following:

$$\begin{aligned}\frac{{}^A d\mathbf{c}}{dt} &= \frac{d(\mathbf{c} \cdot \hat{\mathbf{a}}_x)}{dt} \hat{\mathbf{a}}_x + \frac{d(\mathbf{c} \cdot \hat{\mathbf{a}}_y)}{dt} \hat{\mathbf{a}}_y + \frac{d(\mathbf{c} \cdot \hat{\mathbf{a}}_z)}{dt} \hat{\mathbf{a}}_z \\ &= \frac{d(\cos(\theta)x)}{dt} \hat{\mathbf{a}}_x + \frac{d(l)}{dt} \hat{\mathbf{a}}_y + \frac{d(-\sin(\theta)x)}{dt} \hat{\mathbf{a}}_z \\ &= (-\dot{\theta} \sin(\theta)x + \cos(\theta)\dot{x}) \hat{\mathbf{a}}_x + (\dot{\theta} \cos(\theta)x + \sin(\theta)\dot{x}) \hat{\mathbf{a}}_z\end{aligned}$$

Note that this is the time derivative of  $\mathbf{c}$  in  $\mathbf{A}$ , and is expressed in the  $\mathbf{A}$  frame. We can express it in the  $\mathbf{B}$  frame however, and the expression will still be valid:

$$\begin{aligned}\frac{{}^A d\mathbf{c}}{dt} &= (-\dot{\theta} \sin(\theta)x + \cos(\theta)\dot{x}) \hat{\mathbf{a}}_x + (\dot{\theta} \cos(\theta)x + \sin(\theta)\dot{x}) \hat{\mathbf{a}}_z \\ &= \dot{x} \hat{\mathbf{b}}_x - \theta x \hat{\mathbf{b}}_z\end{aligned}$$

Note the difference in expression complexity between the two forms. They are equivalent, but one is much simpler. This is an extremely important concept, as defining vectors in the more complex forms can vastly slow down formulation of the equations of motion and increase their length, sometimes to a point where they cannot be shown on screen.

## Using Vectors and Reference Frames

We have waited until after all of the relevant mathematical relationships have been defined for vectors and reference frames to introduce code. This is due to how vectors are formed. When starting any problem in [sympy.physics.vector](#) (page 1592), one of the first steps is defining a reference frame (remember to import `sympy.physics.vector` first):

```
>>> from sympy.physics.vector import *
>>> N = ReferenceFrame('N')
```

Now we have created a reference frame,  $\mathbf{N}$ . To have access to any basis vectors, first a reference frame needs to be created. Now that we have made an object representing  $\mathbf{N}$ , we can access its basis vectors:

```
>>> N.x
N.x
>>> N.y
N.y
>>> N.z
N.z
```

## Vector Algebra, in `physics.vector`

We can now do basic algebraic operations on these vectors.:

```
>>> N.x == N.x
True
>>> N.x == N.y
False
>>> N.x + N.y
N.x + N.y
>>> 2 * N.x + N.y
2*N.x + N.y
```

Remember, don't add a scalar quantity to a vector ( $N.x + 5$ ); this will raise an error. At this point, we'll use SymPy's `Symbol` in our vectors. Remember to refer to SymPy's Gotchas and Pitfalls when dealing with symbols.:

```
>>> from sympy import Symbol, symbols
>>> x = Symbol('x')
>>> x * N.x
x*N.x
>>> x*(N.x + N.y)
x*N.x + x*N.y
```

In [\*sympy.physics.vector\*](#) (page 1592) multiple interfaces to vector multiplication have been implemented, at the operator level, method level, and function level. The vector dot product can work as follows:

```
>>> N.x.dot(N.x)
1
>>> N.x.dot(N.y)
0
>>> dot(N.x, N.x)
1
>>> dot(N.x, N.y)
0
```

The “official” interface is the function interface; this is what will be used in all examples. This is to avoid confusion with the attribute and methods being next to each other, and in the case of the operator operation priority. The operators used in [\*sympy.physics.vector\*](#) (page 1592) for vector multiplication do not possess the correct order of operations; this can lead to errors. Care with parentheses is needed when using operators to represent vector multiplication.

The cross product is the other vector multiplication which will be discussed here. It offers similar interfaces to the dot product, and comes with the same warnings.

```
>>> N.x.cross(N.x)
0
>>> N.x.cross(N.z)
- N.y
>>> cross(N.x, N.y)
N.z
>>> cross(N.x, (N.y + N.z))
- N.y + N.z
```

Two additional operations can be done with vectors: normalizing the vector to length 1, and getting its magnitude. These are done as follows:

```
>>> (N.x + N.y).normalize()
sqrt(2)/2*N.x + sqrt(2)/2*N.y
>>> (N.x + N.y).magnitude()
sqrt(2)
```

Vectors are often expressed in a matrix form, especially for numerical purposes. Since the matrix form does not contain any information about the reference frame the vector is defined in, you must provide a reference frame to extract the measure numbers from the vector. There is a convenience function to do this:

```
>>> (x * N.x + 2 * x * N.y + 3 * x * N.z).to_matrix(N)
Matrix([
 [x],
 [2*x],
 [3*x]])
```

## Vector Calculus, in `physics.vector`

We have already introduced our first reference frame. We can take the derivative in that frame right now, if we desire:

```
>>> (x * N.x + N.y).diff(x, N)
N.x
```

SymPy has a `diff` function, but it does not currently work with `sympy.physics.vector` (page 1592) Vectors, so please use Vector's `diff` method. The reason for this is that when differentiating a Vector, the frame of reference must be specified in addition to what you are taking the derivative with respect to; SymPy's `diff` function doesn't fit this mold.

The more interesting case arise with multiple reference frames. If we introduce a second reference frame, **A**, we now have two frames. Note that at this point we can add components of **N** and **A** together, but cannot perform vector multiplication, as no relationship between the two frames has been defined.

```
>>> A = ReferenceFrame('A')
>>> A.x + N.x
N.x + A.x
```

If we want to do vector multiplication, first we have to define an orientation. The `orient` method of `ReferenceFrame` provides that functionality.

```
>>> A.orient(N, 'Axis', [x, N.y])
```

If we desire, we can view the DCM between these two frames at any time. This can be calculated with the `dcm` method. This code: `N.dcm(A)` gives the dcm  ${}^A\mathbf{C}^N$ .

This orients the **A** frame relative to the **N** frame by a simple rotation around the Y axis, by an amount  $x$ . Other, more complicated rotation types include Body rotations, Space rotations, quaternions, and arbitrary axis rotations. Body and space rotations are equivalent to doing 3 simple rotations in a row, each about a basis vector in the new frame. An example follows:

```
>>> N = ReferenceFrame('N')
>>> Bp = ReferenceFrame('Bp')
>>> Bpp = ReferenceFrame('Bpp')
>>> B = ReferenceFrame('B')
>>> q1,q2,q3 = symbols('q1 q2 q3')
>>> Bpp.orient(N, 'Axis', [q1, N.x])
>>> Bp.orient(Bpp, 'Axis', [q2, Bpp.y])
>>> B.orient(Bp, 'Axis', [q3, Bp.z])
>>> N.dcm(B)
Matrix([
 [
 sin(q3)*cos(q2), cos(q2)*cos(q3), -
 -sin(q3)*cos(q2), sin(q2)],
```

(continues on next page)

(continued from previous page)

```

[sin(q1)*sin(q2)*cos(q3) + sin(q3)*cos(q1), -sin(q1)*sin(q2)*sin(q3) +
 ↪ cos(q1)*cos(q3), -sin(q1)*cos(q2)],
[sin(q1)*sin(q3) - sin(q2)*cos(q1)*cos(q3), sin(q1)*cos(q3) +
 ↪ sin(q2)*sin(q3)*cos(q1), cos(q1)*cos(q2)]]
>>> B.orient(N, 'Body', [q1, q2, q3], 'XYZ')
>>> N.dcm(B)
Matrix([
[
 cos(q2)*cos(q3),
 ↪ sin(q3)*cos(q2), sin(q2)],
[sin(q1)*sin(q2)*cos(q3) + sin(q3)*cos(q1), -sin(q1)*sin(q2)*sin(q3) +
 ↪ cos(q1)*cos(q3), -sin(q1)*cos(q2)],
[sin(q1)*sin(q3) - sin(q2)*cos(q1)*cos(q3), sin(q1)*cos(q3) +
 ↪ sin(q2)*sin(q3)*cos(q1), cos(q1)*cos(q2)]]])

```

Space orientations are similar to body orientation, but applied from the frame to body. Body and space rotations can involve either two or three axes: 'XYZ' works, as does 'YZX', 'ZXZ', 'YXY', etc. What is key is that each simple rotation is about a different axis than the previous one; 'ZZX' does not completely orient a set of basis vectors in 3 space.

Sometimes it will be more convenient to create a new reference frame and orient relative to an existing one in one step. The `orientnew` method allows for this functionality, and essentially wraps the `orient` method. All of the things you can do in `orient`, you can do in `orientnew`.

```
>>> C = N.orientnew('C', 'Axis', [q1, N.x])
```

Quaternions (or Euler Parameters) use 4 value to characterize the orientation of the frame. This and arbitrary axis rotations are described in the `orient` and `orientnew` method help, or in the references [Kane1983].

Finally, before starting multiframe calculus operations, we will introduce another [sympy.physics.vector](#) (page 1592) tool: `dynamicsymbols`. `dynamicsymbols` is a shortcut function to create undefined functions of time within SymPy. The derivative of such a 'dynamicsymbol' is shown below.

```

>>> from sympy import diff
>>> q1, q2, q3 = dynamicsymbols('q1 q2 q3')
>>> diff(q1, Symbol('t'))
Derivative(q1(t), t)

```

The 'dynamicsymbol' printing is not very clear above; we will also introduce a few other tools here. We can use `vprint` instead of `print` for non-interactive sessions.

```

>>> q1
q1(t)
>>> q1d = diff(q1, Symbol('t'))
>>> vprint(q1)
q1
>>> vprint(q1d)
q1'

```

For interactive sessions use `init_vprinting`. There also exist analogs for SymPy's `vprint`, `vpprint`, and `latex`, `vlatex`.

```
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> q1
q1
>>> q1d
q1'
```

A ‘dynamicsymbol’ should be used to represent any time varying quantity in [sympy.physics.vector](#) (page 1592), whether it is a coordinate, varying position, or force. The primary use of a ‘dynamicsymbol’ is for speeds and coordinates (of which there will be more discussion in the Kinematics Section of the documentation).

Now we will define the orientation of our new frames with a ‘dynamicsymbol’, and can take derivatives and time derivatives with ease. Some examples follow.

```
>>> N = ReferenceFrame('N')
>>> B = N.orientnew('B', 'Axis', [q1, N.x])
>>> (B.y*q2 + B.z).diff(q2, N)
B.y
>>> (B.y*q2 + B.z).dt(N)
(-q1' + q2')*B.y + q2*q1'*B.z
```

Note that the output vectors are kept in the same frames that they were provided in. This remains true for vectors with components made of basis vectors from multiple frames:

```
>>> (B.y*q2 + B.z + q2*N.x).diff(q2, N)
N.x + B.y
```

## How Vectors are Coded

What follows is a short description of how vectors are defined by the code in [sympy.physics.vector](#) (page 1592). It is provided for those who want to learn more about how this part of [sympy.physics.vector](#) (page 1592) works, and does not need to be read to use this module; don’t read it unless you want to learn how this module was implemented.

Every Vector’s main information is stored in the `args` attribute, which stores the three measure numbers for each basis vector in a frame, for every relevant frame. A vector does not exist in code until a ReferenceFrame is created. At this point, the `x`, `y`, and `z` attributes of the reference frame are immutable Vector’s which have measure numbers of `[1,0,0]`, `[0,1,0]`, and `[0,0,1]` associated with that ReferenceFrame. Once these vectors are accessible, new vectors can be created by doing algebraic operations with the basis vectors. A vector can have components from multiple frames though. That is why `args` is a list; it has as many elements in the list as there are unique ReferenceFrames in its components, i.e. if there are A and B frame basis vectors in our new vector, `args` is of length 2; if it has A, B, and C frame basis vector, `args` is of length three.

Each element in the `args` list is a 2-tuple; the first element is a SymPy Matrix (this is where the measure numbers for each set of basis vectors are stored) and the second element is a ReferenceFrame to associate those measure numbers with.

ReferenceFrame stores a few things. First, it stores the name you supply it on creation (name attribute). It also stores the direction cosine matrices, defined upon creation with the `orientnew` method, or calling the `orient` method after creation. The direction cosine matrices are represented by SymPy’s Matrix, and are part of a dictionary where the keys are the

ReferenceFrame and the value the Matrix; these are set bi-directionally; in that when you orient A to N you are setting A's orientation dictionary to include N and its Matrix, but you are also setting N's orientation dictionary to include A and its Matrix (that DCM being the transpose of the other).

## Vector: Kinematics

This document will give some mathematical background to describing a system's kinematics as well as how to represent the kinematics in [\*sympy.physics.vector\*](#) (page 1592).

### Introduction to Kinematics

The first topic is rigid motion kinematics. A rigid body is an idealized representation of a physical object which has mass and rotational inertia. Rigid bodies are obviously not flexible. We can break down rigid body motion into translational motion, and rotational motion (when dealing with particles, we only have translational motion). Rotational motion can further be broken down into simple rotations and general rotations.

Translation of a rigid body is defined as a motion where the orientation of the body does not change during the motion; or during the motion any line segment would be parallel to itself at the start of the motion.

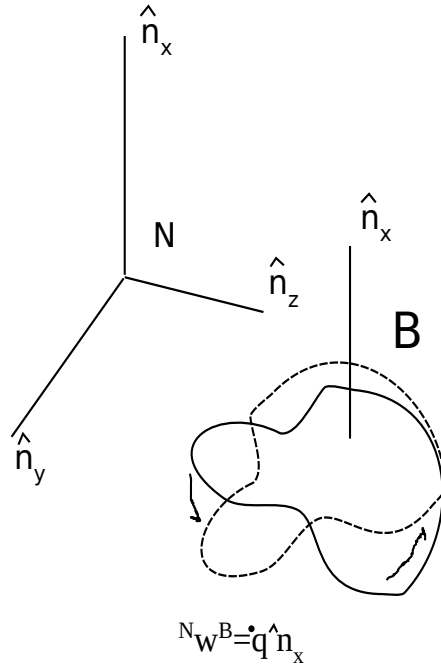
Simple rotations are rotations in which the orientation of the body may change, but there is always one line which remains parallel to itself at the start of the motion.

General rotations are rotations which there is not always one line parallel to itself at the start of the motion.

### Angular Velocity

The angular velocity of a rigid body refers to the rate of change of its orientation. The angular velocity of a body is written down as:  ${}^{\mathbf{N}}\omega^{\mathbf{B}}$ , or the angular velocity of **B** in **N**, which is a vector. Note that here, the term rigid body was used, but reference frames can also have angular velocities. Further discussion of the distinction between a rigid body and a reference frame will occur later when describing the code representation.

Angular velocity is defined as being positive in the direction which causes the orientation angles to increase (for simple rotations, or series of simple rotations).



The angular velocity vector represents the time derivative of the orientation. As a time derivative vector quantity, like those covered in the [Vector & ReferenceFrame](#) documentation, this quantity (angular velocity) needs to be defined in a reference frame. That is what the  $N$  is in the above definition of angular velocity; the frame in which the angular velocity is defined in.

The angular velocity of  $B$  in  $N$  can also be defined by:

$${}^N\omega^B = \left( \frac{{}^N d\hat{\mathbf{b}}_y}{dt} \cdot \hat{\mathbf{b}}_z \right) \hat{\mathbf{b}}_x + \left( \frac{{}^N d\hat{\mathbf{b}}_z}{dt} \cdot \hat{\mathbf{b}}_x \right) \hat{\mathbf{b}}_y + \left( \frac{{}^N d\hat{\mathbf{b}}_x}{dt} \cdot \hat{\mathbf{b}}_y \right) \hat{\mathbf{b}}_z$$

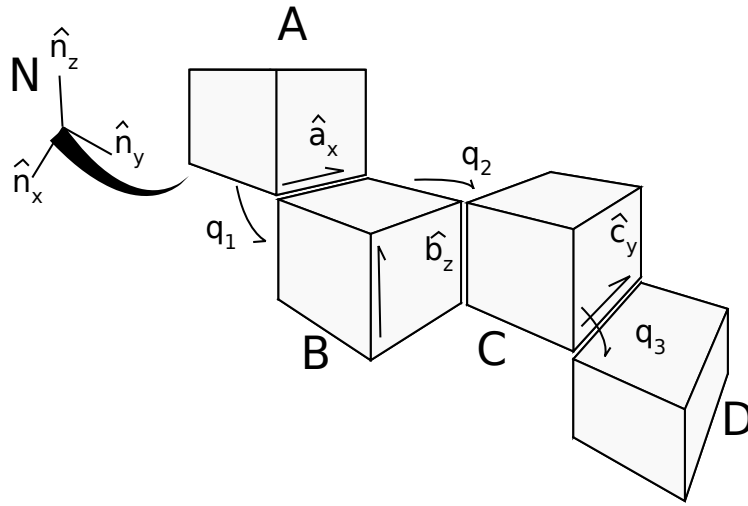
It is also common for a body's angular velocity to be written as:

$${}^N\omega^B = w_x \hat{\mathbf{b}}_x + w_y \hat{\mathbf{b}}_y + w_z \hat{\mathbf{b}}_z$$

There are a few additional important points relating to angular velocity. The first is the addition theorem for angular velocities, a way of relating the angular velocities of multiple bodies and frames. The theorem follows:

$${}^N\omega^D = {}^N\omega^A + {}^A\omega^B + {}^B\omega^C + {}^C\omega^D$$

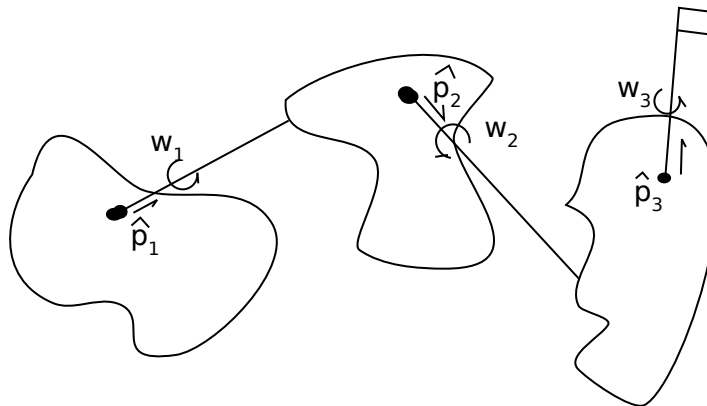
This is also shown in the following example:



$$\begin{aligned}
 {}^N\omega^A &= 0 \\
 {}^A\omega^B &= \dot{q}_1 \hat{\mathbf{a}}_x \\
 {}^B\omega^C &= -\dot{q}_2 \hat{\mathbf{b}}_z \\
 {}^C\omega^D &= \dot{q}_3 \hat{\mathbf{c}}_y \\
 {}^N\omega^D &= \dot{q}_1 \hat{\mathbf{a}}_x - \dot{q}_2 \hat{\mathbf{b}}_z + \dot{q}_3 \hat{\mathbf{c}}_y
 \end{aligned}$$

Note the signs used in the angular velocity definitions, which are related to how the displacement angle is defined in this case.

This theorem makes defining angular velocities of multibody systems much easier, as the angular velocity of a body in a chain needs to only be defined to the previous body in order to be fully defined (and the first body needs to be defined in the desired reference frame). The following figure shows an example of when using this theorem can make things easier.



Here we can easily write the angular velocity of the body **D** in the reference frame of the first body **A**:

$${}^A\omega^D = w_1 \hat{\mathbf{p}}_1 + w_2 \hat{\mathbf{p}}_2 + w_3 \hat{\mathbf{p}}_3$$



It is very important to remember to only use this with angular velocities; you cannot use this theorem with the velocities of points.

There is another theorem commonly used: the derivative theorem. It provides an alternative method (which can be easier) to calculate the time derivative of a vector in a reference frame:

$$\frac{{}^{\mathbf{N}}d\mathbf{v}}{dt} = \frac{{}^{\mathbf{B}}d\mathbf{v}}{dt} + {}^{\mathbf{N}}\omega^{\mathbf{B}} \times \mathbf{v}$$

The vector  $\mathbf{v}$  can be any vector quantity: a position vector, a velocity vector, angular velocity vector, etc. Instead of taking the time derivative of the vector in  $\mathbf{N}$ , we take it in  $\mathbf{B}$ , where  $\mathbf{B}$  can be any reference frame or body, usually one in which it is easy to take the derivative on  $\mathbf{v}$  in ( $\mathbf{v}$  is usually composed only of the basis vector set belonging to  $\mathbf{B}$ ). Then we add the cross product of the angular velocity of our newer frame,  ${}^{\mathbf{N}}\omega^{\mathbf{B}}$  and our vector quantity  $\mathbf{v}$ . Again, you can choose any alternative frame for this. Examples follow:

## Angular Acceleration

Angular acceleration refers to the time rate of change of the angular velocity vector. Just as the angular velocity vector is for a body and is specified in a frame, the angular acceleration vector is for a body and is specified in a frame:  ${}^{\mathbf{N}}\alpha^{\mathbf{B}}$ , or the angular acceleration of  $\mathbf{B}$  in  $\mathbf{N}$ , which is a vector.

Calculating the angular acceleration is relatively straight forward:

$${}^{\mathbf{N}}\alpha^{\mathbf{B}} = \frac{{}^{\mathbf{N}}d{}^{\mathbf{N}}\omega^{\mathbf{B}}}{dt}$$

Note that this can be calculated with the derivative theorem, and when the angular velocity is defined in a body fixed frame, becomes quite simple:

$${}^{\mathbf{N}}\alpha^{\mathbf{B}} = \frac{{}^{\mathbf{B}}d{}^{\mathbf{N}}\omega^{\mathbf{B}}}{dt}$$

$${}^{\mathbf{N}}\alpha^{\mathbf{B}} = \frac{{}^{\mathbf{B}}d{}^{\mathbf{N}}\omega^{\mathbf{B}}}{dt} + {}^{\mathbf{N}}\omega^{\mathbf{B}} \times {}^{\mathbf{N}}\omega^{\mathbf{B}}$$

$$\text{if } {}^{\mathbf{N}}\omega^{\mathbf{B}} = w_x \hat{\mathbf{b}}_x + w_y \hat{\mathbf{b}}_y + w_z \hat{\mathbf{b}}_z$$

$$\text{then } {}^{\mathbf{N}}\alpha^{\mathbf{B}} = \frac{{}^{\mathbf{B}}d{}^{\mathbf{N}}\omega^{\mathbf{B}}}{dt} + \underbrace{{}^{\mathbf{N}}\omega^{\mathbf{B}} \times {}^{\mathbf{N}}\omega^{\mathbf{B}}}_{\text{this is 0 by definition}}$$

$${}^{\mathbf{N}}\alpha^{\mathbf{B}} = \frac{dw_x}{dt} \hat{\mathbf{b}}_x + \frac{dw_y}{dt} \hat{\mathbf{b}}_y + \frac{dw_z}{dt} \hat{\mathbf{b}}_z$$

$${}^{\mathbf{N}}\alpha^{\mathbf{B}} = \dot{w}_x \hat{\mathbf{b}}_x + \dot{w}_y \hat{\mathbf{b}}_y + \dot{w}_z \hat{\mathbf{b}}_z$$

Again, this is only for the case in which the angular velocity of the body is defined in body fixed components.

## Point Velocity & Acceleration

Consider a point,  $P$ : we can define some characteristics of the point. First, we can define a position vector from some other point to  $P$ . Second, we can define the velocity vector of  $P$  in a reference frame of our choice. Third, we can define the acceleration vector of  $P$  in a reference frame of our choice.

These three quantities are read as:

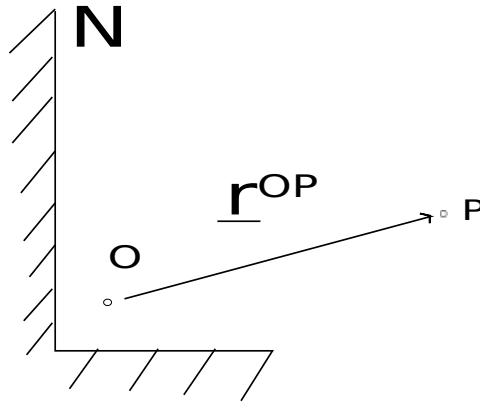
$\mathbf{r}^{OP}$ , the position vector from  $O$  to  $P$

${}^N\mathbf{v}^P$ , the velocity of  $P$  in the reference frame  $\mathbf{N}$

${}^N\mathbf{a}^P$ , the acceleration of  $P$  in the reference frame  $\mathbf{N}$

Note that the position vector does not have a frame associated with it; this is because there is no time derivative involved, unlike the velocity and acceleration vectors.

We can find these quantities for a simple example easily:



Let's define:  $\mathbf{r}^{OP} = q_x \hat{\mathbf{n}}_x + q_y \hat{\mathbf{n}}_y$

$${}^N\mathbf{v}^P = \frac{{}^N d\mathbf{r}^{OP}}{dt}$$

then we can calculate:  ${}^N\mathbf{v}^P = \dot{q}_x \hat{\mathbf{n}}_x + \dot{q}_y \hat{\mathbf{n}}_y$

$$\text{and } {}^N\mathbf{a}^P = \frac{{}^N d({}^N\mathbf{v}^P)}{dt}$$

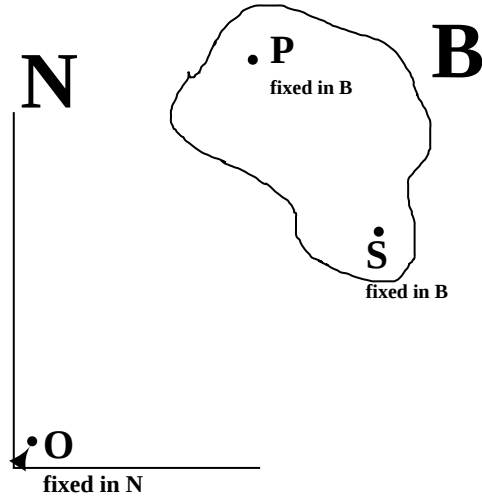
$${}^N\mathbf{a}^P = \ddot{q}_x \hat{\mathbf{n}}_x + \ddot{q}_y \hat{\mathbf{n}}_y$$

It is critical to understand in the above example that the point  $O$  is fixed in the reference frame  $\mathbf{N}$ . There is no addition theorem for translational velocities; alternatives will be discussed later though. Also note that the position of every point might not always need to be defined to form the dynamic equations of motion. When you don't want to define the position vector of a point, you can start by just defining the velocity vector. For the above example:

Let us instead define the velocity vector as:  ${}^N\mathbf{v}^P = u_x \hat{\mathbf{n}}_x + u_y \hat{\mathbf{n}}_y$

then acceleration can be written as:  ${}^N\mathbf{a}^P = \dot{u}_x \hat{\mathbf{n}}_x + \dot{u}_y \hat{\mathbf{n}}_y$

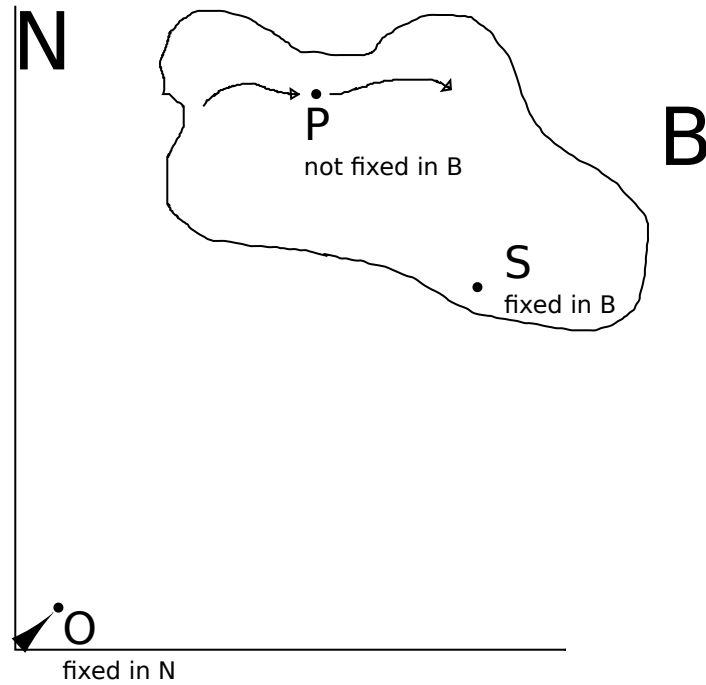
There will often be cases when the velocity of a point is desired and a related point's velocity is known. For the cases in which we have two points fixed on a rigid body, we use the 2-Point Theorem:



Let's say we know the velocity of the point  $S$  and the angular velocity of the body  $\mathbf{B}$ , both defined in the reference frame  $\mathbf{N}$ . We can calculate the velocity and acceleration of the point  $P$  in  $\mathbf{N}$  as follows:

$$\begin{aligned}\mathbf{N}\mathbf{v}^P &= \mathbf{N}\mathbf{v}^S + \mathbf{N}\boldsymbol{\omega}^{\mathbf{B}} \times \mathbf{r}^{SP} \\ \mathbf{N}\mathbf{a}^P &= \mathbf{N}\mathbf{a}^S + \mathbf{N}\boldsymbol{\alpha}^{\mathbf{B}} \times \mathbf{r}^{SP} + \mathbf{N}\boldsymbol{\omega}^{\mathbf{B}} \times (\mathbf{N}\boldsymbol{\omega}^{\mathbf{B}} \times \mathbf{r}^{SP})\end{aligned}$$

When only one of the two points is fixed on a body, the 1 point theorem is used instead.

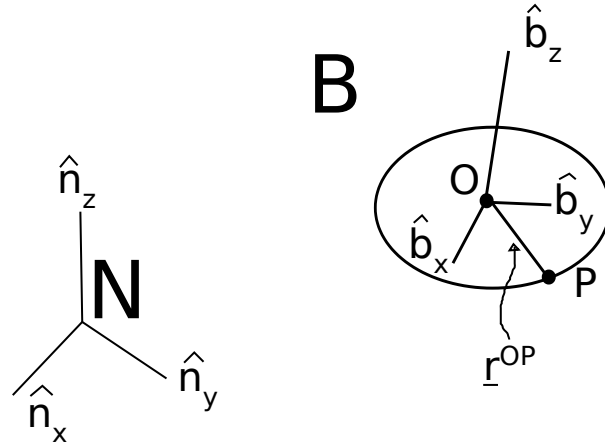


Here, the velocity of point  $S$  is known in the frame  $\mathbf{N}$ , the angular velocity of  $\mathbf{B}$  is known in  $\mathbf{N}$ , and the velocity of the point  $P$  is known in the frame associated with body  $\mathbf{B}$ . We can then write the velocity and acceleration of  $P$  in  $\mathbf{N}$  as:

$$\mathbf{N}\mathbf{v}^P = \mathbf{B}\mathbf{v}^P + \mathbf{N}\mathbf{v}^S + \mathbf{N}\boldsymbol{\omega}^{\mathbf{B}} \times \mathbf{r}^{SP}$$

$$\mathbf{N}\mathbf{a}^P = \mathbf{B}\mathbf{a}^S + \mathbf{N}\mathbf{a}^O + \mathbf{N}\boldsymbol{\alpha}^{\mathbf{B}} \times \mathbf{r}^{SP} + \mathbf{N}\boldsymbol{\omega}^{\mathbf{B}} \times (\mathbf{N}\boldsymbol{\omega}^{\mathbf{B}} \times \mathbf{r}^{SP}) + 2\mathbf{N}\boldsymbol{\omega}^{\mathbf{B}} \times \mathbf{B}\mathbf{v}^P$$

Examples of applications of the 1 point and 2 point theorem follow.



This example has a disc translating and rotating in a plane. We can easily define the angular velocity of the body **B** and velocity of the point *O*:

$${}^N\omega^B = u_3 \hat{\mathbf{n}}_z = u_3 \hat{\mathbf{b}}_z$$

$${}^N\mathbf{v}^O = u_1 \hat{\mathbf{n}}_x + u_2 \hat{\mathbf{n}}_y$$

and accelerations can be written as:

$${}^N\alpha^B = \dot{u}_3 \hat{\mathbf{n}}_z = \dot{u}_3 \hat{\mathbf{b}}_z$$

$${}^N\mathbf{a}^O = \dot{u}_1 \hat{\mathbf{n}}_x + \dot{u}_2 \hat{\mathbf{n}}_y$$

We can use the 2 point theorem to calculate the velocity and acceleration of point *P* now.

$$\mathbf{r}^{OP} = R \hat{\mathbf{b}}_x$$

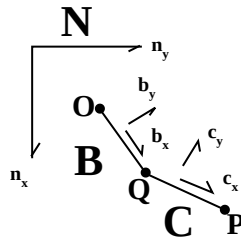
$${}^N\mathbf{v}^P = {}^N\mathbf{v}^O + {}^N\omega^B \times \mathbf{r}^{OP}$$

$${}^N\mathbf{v}^P = u_1 \hat{\mathbf{n}}_x + u_2 \hat{\mathbf{n}}_y + u_3 \hat{\mathbf{b}}_z \times R \hat{\mathbf{b}}_x = u_1 \hat{\mathbf{n}}_x + u_2 \hat{\mathbf{n}}_y + u_3 R \hat{\mathbf{b}}_y$$

$${}^N\mathbf{a}^P = {}^N\mathbf{a}^O + {}^N\alpha^B \times \mathbf{r}^{OP} + {}^N\omega^B \times ({}^N\omega^B \times \mathbf{r}^{OP})$$

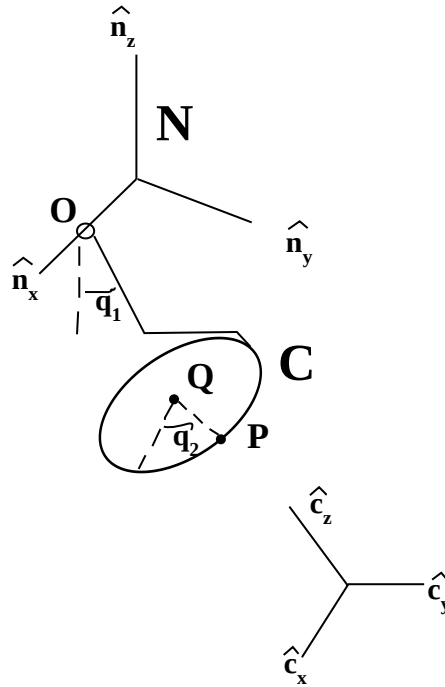
$${}^N\mathbf{a}^P = \dot{u}_1 \hat{\mathbf{n}}_x + \dot{u}_2 \hat{\mathbf{n}}_y + \dot{u}_3 \hat{\mathbf{b}}_z \times R \hat{\mathbf{b}}_x + u_3 \hat{\mathbf{b}}_z \times (u_3 \hat{\mathbf{b}}_z \times R \hat{\mathbf{b}}_x)$$

$${}^N\mathbf{a}^P = \dot{u}_1 \hat{\mathbf{n}}_x + \dot{u}_2 \hat{\mathbf{n}}_y + R \dot{u}_3 \hat{\mathbf{b}}_y - R u_3^2 \hat{\mathbf{b}}_x$$



In this example we have a double pendulum. We can use the two point theorem twice here in order to find the velocity of points  $Q$  and  $P$ ; point  $O$ 's velocity is zero in  $\mathbf{N}$ .

$$\begin{aligned}\mathbf{r}^{OQ} &= l\hat{\mathbf{b}}_x \\ \mathbf{r}^{QP} &= l\hat{\mathbf{c}}_x \\ {}^N\boldsymbol{\omega}^{\mathbf{B}} &= u_1\hat{\mathbf{b}}_z \\ {}^N\boldsymbol{\omega}^{\mathbf{C}} &= u_2\hat{\mathbf{c}}_z \\ {}^N\mathbf{v}^Q &= {}^N\mathbf{v}^O + {}^N\boldsymbol{\omega}^{\mathbf{B}} \times \mathbf{r}^{OQ} \\ {}^N\mathbf{v}^Q &= u_1l\hat{\mathbf{b}}_y \\ {}^N\mathbf{v}^P &= {}^N\mathbf{v}^Q + {}^N\boldsymbol{\omega}^{\mathbf{C}} \times \mathbf{r}^{QP} \\ {}^N\mathbf{v}^P &= u_1l\hat{\mathbf{b}}_y + u_2\hat{\mathbf{c}}_z \times l\hat{\mathbf{c}}_x \\ {}^N\mathbf{v}^P &= u_1l\hat{\mathbf{b}}_y + u_2l\hat{\mathbf{c}}_y\end{aligned}$$

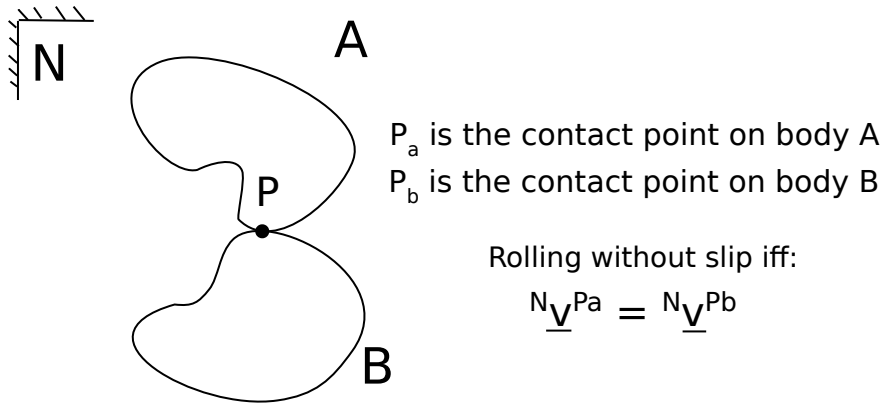


In this example we have a particle moving on a ring; the ring is supported by a rod which can rotate about the  $\hat{\mathbf{n}}_x$  axis. First we use the two point theorem to find the velocity of the center point of the ring,  $Q$ , then use the 1 point theorem to find the velocity of the particle on the

ring.

$$\begin{aligned}
 {}^N\omega^C &= u_1 \hat{\mathbf{n}}_x \\
 \mathbf{r}^{OQ} &= -l \hat{\mathbf{c}}_z \\
 {}^N\mathbf{v}^Q &= u_1 l \hat{\mathbf{c}}_y \\
 \mathbf{r}^{QP} &= R(\cos(q_2) \hat{\mathbf{c}}_x + \sin(q_2) \hat{\mathbf{c}}_y) \\
 {}^C\mathbf{v}^P &= Ru_2(-\sin(q_2) \hat{\mathbf{c}}_x + \cos(q_2) \hat{\mathbf{c}}_y) \\
 {}^N\mathbf{v}^P &= {}^C\mathbf{v}^P + {}^N\mathbf{v}^Q + {}^N\omega^C \times \mathbf{r}^{QP} \\
 {}^N\mathbf{v}^P &= Ru_2(-\sin(q_2) \hat{\mathbf{c}}_x + \cos(q_2) \hat{\mathbf{c}}_y) + u_1 l \hat{\mathbf{c}}_y + u_1 \hat{\mathbf{c}}_x \times R(\cos(q_2) \hat{\mathbf{c}}_x + \sin(q_2) \hat{\mathbf{c}}_y) \\
 {}^N\mathbf{v}^P &= -Ru_2 \sin(q_2) \hat{\mathbf{c}}_x + (Ru_2 \cos(q_2) + u_1 l) \hat{\mathbf{c}}_y + Ru_1 \sin(q_2) \hat{\mathbf{c}}_z
 \end{aligned}$$

A final topic in the description of velocities of points is that of rolling, or rather, rolling without slip. Two bodies are said to be rolling without slip if and only if the point of contact on each body has the same velocity in another frame. See the following figure:



This is commonly used to form the velocity of a point on one object rolling on another fixed object, such as in the following example:

### Kinematics in `physics.vector`

It should be clear by now that the topic of kinematics here has been mostly describing the correct way to manipulate vectors into representing the velocities of points. Within [sympy.physics.vector](#) (page 1592) there are convenient methods for storing these velocities associated with frames and points. We'll now revisit the above examples and show how to represent them in sympy.

The topic of reference frame creation has already been covered. When a `ReferenceFrame` is created though, it automatically calculates the angular velocity of the frame using the time derivative of the DCM and the angular velocity definition.

```

>>> from sympy import Symbol, sin, cos
>>> from sympy.physics.vector import *
>>> init_vprinting(pretty_print=False)
>>> N = ReferenceFrame('N')

```

(continues on next page)

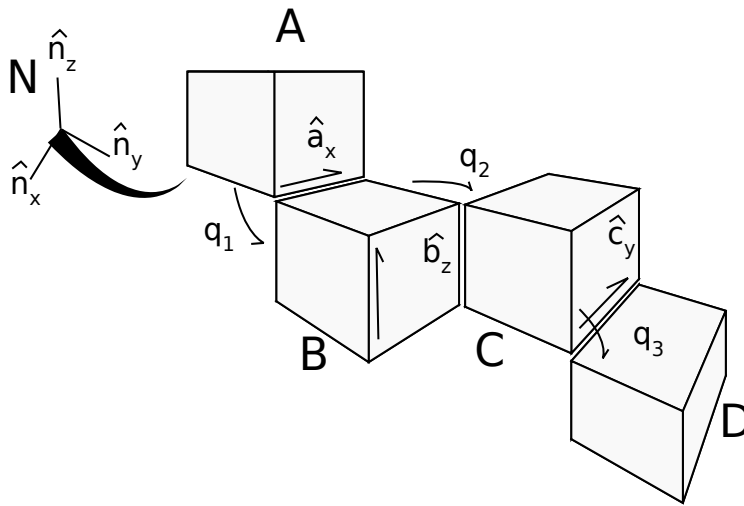
(continued from previous page)

```
>>> q1 = dynamicsymbols('q1')
>>> A = N.orientnew('A', 'Axis', [q1, N.x])
>>> A.ang_vel_in(N)
q1'*N.x
```

Note that the angular velocity can be defined in an alternate way:

```
>>> B = ReferenceFrame('B')
>>> u1 = dynamicsymbols('u1')
>>> B.set_ang_vel(N, u1 * B.y)
>>> B.ang_vel_in(N)
u1*B.y
>>> N.ang_vel_in(B)
- u1*B.y
```

Both upon frame creation during `orientnew` and when calling `set_ang_vel`, the angular velocity is set in both frames involved, as seen above.



Here we have multiple bodies with angular velocities defined relative to each other. This is coded as:

```
>>> N = ReferenceFrame('N')
>>> A = ReferenceFrame('A')
>>> B = ReferenceFrame('B')
>>> C = ReferenceFrame('C')
>>> D = ReferenceFrame('D')
>>> u1, u2, u3 = dynamicsymbols('u1 u2 u3')
>>> A.set_ang_vel(N, 0)
>>> B.set_ang_vel(A, u1 * A.x)
>>> C.set_ang_vel(B, -u2 * B.z)
>>> D.set_ang_vel(C, u3 * C.y)
>>> D.ang_vel_in(N)
u1*A.x - u2*B.z + u3*C.y
```



In `sympy.physics.vector` (page 1592) the shortest path between two frames is used when finding the angular velocity. That would mean if we went back and set:

```
>>> D.set_ang_vel(N, 0)
>>> D.ang_vel_in(N)
0
```

The path that was just defined is what is used. This can cause problems though, as now the angular velocity definitions are inconsistent. It is recommended that you avoid doing this.

Points are a translational analog to the rotational `ReferenceFrame`. Creating a `Point` can be done in two ways, like `ReferenceFrame`:

```
>>> O = Point('O')
>>> P = O.locatenew('P', 3 * N.x + N.y)
>>> P.pos_from(O)
3*N.x + N.y
>>> Q = Point('Q')
>>> Q.set_pos(P, N.z)
>>> Q.pos_from(P)
N.z
>>> Q.pos_from(O)
3*N.x + N.y + N.z
```

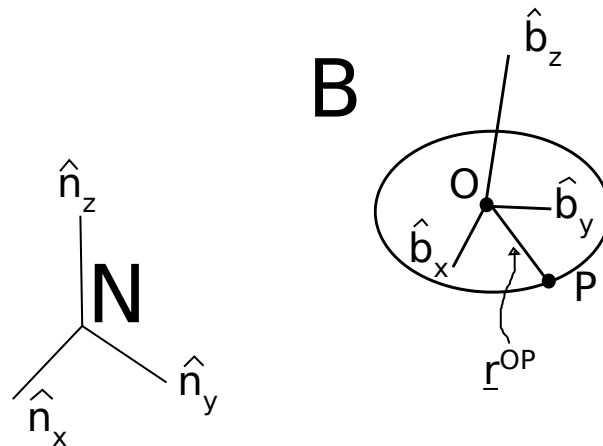
Similar to `ReferenceFrame`, the position vector between two points is found by the shortest path (number of intermediate points) between them. Unlike rotational motion, there is no addition theorem for the velocity of points. In order to have the velocity of a `Point` in a `ReferenceFrame`, you have to set the value.

```
>>> O = Point('O')
>>> O.set_vel(N, u1*N.x)
>>> O.vel(N)
u1*N.x
```

For both translational and rotational accelerations, the value is computed by taking the time derivative of the appropriate velocity, unless the user sets it otherwise.

```
>>> O.acc(N)
u1'*N.x
>>> O.set_acc(N, u2*u1*N.y)
>>> O.acc(N)
u1*u2*N.y
```

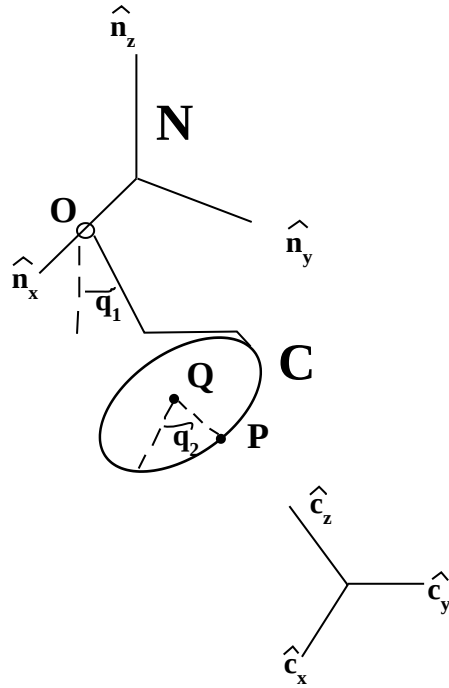
Next is a description of the 2 point and 1 point theorems, as used in `sympy`.



First is the translating, rotating disc.

```
>>> N = ReferenceFrame('N')
>>> u1, u2, u3 = dynamicsymbols('u1 u2 u3')
>>> R = Symbol('R')
>>> B = ReferenceFrame('B')
>>> O = Point('O')
>>> O.set_vel(N, u1 * N.x + u2 * N.y)
>>> P = O.locatenew('P', R * B.x)
>>> B.set_ang_vel(N, u3 * B.z)
>>> P.v2pt_theory(O, N, B)
u1*N.x + u2*N.y + R*u3*B.y
>>> P.a2pt_theory(O, N, B)
u1'*N.x + u2'*N.y - R*u3**2*B.x + R*u3'*B.y
```

We will also cover implementation of the 1 point theorem.



This is the particle moving on a ring, again.

```
>>> N = ReferenceFrame('N')
>>> u1, u2 = dynamicsymbols('u1 u2')
>>> q1, q2 = dynamicsymbols('q1 q2')
>>> l = Symbol('l')
>>> R = Symbol('R')
>>> C = N.orientnew('C', 'Axis', [q1, N.x])
>>> C.set_ang_vel(N, u1 * N.x)
>>> O = Point('O')
>>> O.set_vel(N, 0)
>>> Q = O.locatenew('Q', -l * C.z)
>>> P = Q.locatenew('P', R * (cos(q2) * C.x + sin(q2) * C.y))
>>> P.set_vel(C, R * u2 * (-sin(q2) * C.x + cos(q2) * C.y))
>>> Q.v2pt_theory(O, N, C)
l*u1*C.y
>>> P.vlpt_theory(Q, N, C)
- R*u2*sin(q2)*C.x + (R*u2*cos(q2) + l*u1)*C.y + R*u1*sin(q2)*C.z
```

## Potential Issues/Advanced Topics/Future Features in Physics/Vector Module

This document will describe some of the more advanced functionality that this module offers but which is not part of the “official” interface. Here, some of the features that will be implemented in the future will also be covered, along with unanswered questions about proper functionality. Also, common problems will be discussed, along with some solutions.

### Inertia (Dyadics)

A dyadic tensor is a second order tensor formed by the juxtaposition of a pair of vectors. There are various operations defined with respect to dyadics, which have been implemented in [vector](#) (page 1592) in the form of class [sympy.physics.vector.dyadic.Dyadic](#) (page 1648). To know more, refer to the [sympy.physics.vector.dyadic.Dyadic](#) (page 1648) and [sympy.physics.vector.vector.Vector](#) (page 1641) class APIs. Dyadics are used to define the inertia of bodies within [sympy.physics.mechanics](#) (page 1675). Inertia dyadics can be defined explicitly but the `inertia` function is typically much more convenient for the user:

```
>>> from sympy.physics.mechanics import ReferenceFrame, inertia
>>> N = ReferenceFrame('N')
```

Supply a reference frame and the moments of inertia if the object is symmetrical:

```
>>> inertia(N, 1, 2, 3)
(N.x|N.x) + 2*(N.y|N.y) + 3*(N.z|N.z)
```

Supply a reference frame along with the products and moments of inertia for a general object:

```
>>> inertia(N, 1, 2, 3, 4, 5, 6)
(N.x|N.x) + 4*(N.x|N.y) + 6*(N.x|N.z) + 4*(N.y|N.x) + 2*(N.y|N.y) + 5*(N.y|N.
↪z) + 6*(N.z|N.x) + 5*(N.z|N.y) + 3*(N.z|N.z)
```

Notice that the `inertia` function returns a dyadic with each component represented as two unit vectors separated by a `|`. Refer to the [sympy.physics.vector.dyadic.Dyadic](#) (page 1648) section for more information about dyadics.

Inertia is often expressed in a matrix, or tensor, form, especially for numerical purposes. Since the matrix form does not contain any information about the reference frame(s) the inertia dyadic is defined in, you must provide one or two reference frames to extract the measure numbers from the dyadic. There is a convenience function to do this:

```
>>> inertia(N, 1, 2, 3, 4, 5, 6).to_matrix(N)
Matrix([
[1, 4, 6],
[4, 2, 5],
[6, 5, 3]])
```

## Common Issues

Here issues with numerically integrating code, choice of *dynamicsymbols* for coordinate and speed representation, printing, differentiating, and substitution will occur.

## Printing

The default printing options are to use sorting for Vector and Dyadic measure numbers, and have unsorted output from the `vprint`, `vpprint`, and `vlatex` functions. If you are printing something large, please use one of those functions, as the sorting can increase printing time from seconds to minutes.

## Substitution

Substitution into large expressions can be slow, and take a few minutes.

## Acceleration of Points

At a minimum, points need to have their velocities defined, as the acceleration can be calculated by taking the time derivative of the velocity in the same frame. If the 1 point or 2 point theorems were used to compute the velocity, the time derivative of the velocity expression will most likely be more complex than if you were to use the acceleration level 1 point and 2 point theorems. Using the acceleration level methods can result in shorted expressions at this point, which will result in shorter expressions later (such as when forming Kane's equations).

## Advanced Interfaces

Here we will cover advanced options in: `ReferenceFrame`, `dynamicsymbols`, and some associated functionality.

## ReferenceFrame

`ReferenceFrame` is shown as having a `.name` attribute and `.x`, `.y`, and `.z` attributes for accessing the basis vectors, as well as a fairly rigidly defined print output. If you wish to have a different set of indices defined, there is an option for this. This will also require a different interface for accessing the basis vectors.

```
>>> from sympy.physics.vector import ReferenceFrame, vprint, vpprint, vlatex
>>> N = ReferenceFrame('N', indices=['i', 'j', 'k'])
>>> N['i']
N['i']
>>> N.x
N['i']
>>> vlatex(N.x)
'\\mathbf{\\hat{n}}_{i}'
```

Also, the latex output can have custom strings; rather than just indices though, the entirety of each basis vector can be specified. The custom latex strings can occur without custom indices, and also overwrites the latex string that would be used if there were custom indices.

```
>>> from sympy.physics.vector import ReferenceFrame, vlatex
>>> N = ReferenceFrame('N', latexs=['n1', '\\mathbf{n}_2', 'cat'])
>>> vlatex(N.x)
'n1'
>>> vlatex(N.y)
'\\mathbf{n}_2'
>>> vlatex(N.z)
'cat'
```

## dynamicsymbols

The dynamicsymbols function also has ‘hidden’ functionality; the variable which is associated with time can be changed, as well as the notation for printing derivatives.

```
>>> from sympy import symbols
>>> from sympy.physics.vector import dynamicsymbols, vprint
>>> q1 = dynamicsymbols('q1')
>>> q1
q1(t)
>>> dynamicsymbols._t = symbols('T')
>>> q2 = dynamicsymbols('q2')
>>> q2
q2(T)
>>> q1
q1(t)
>>> q1d = dynamicsymbols('q1', 1)
>>> vprint(q1d)
q1'
>>> dynamicsymbols._str = 'd'
>>> vprint(q1d)
q1d
>>> dynamicsymbols._str = '\\'
>>> dynamicsymbols._t = symbols('t')
```

Note that only dynamic symbols created after the change are different. The same is not true for the `.str` attribute; this affects the printing output only, so dynamic symbols created before or after will print the same way.

Also note that Vector’s `.dt` method uses the `._t` attribute of `dynamicsymbols`, along with a number of other important functions and methods. Don’t mix and match symbols representing time.

## Scalar and Vector Field Functionality

### Introduction

#### Vectors and Scalars

In physics, we deal with two kinds of quantities – scalars and vectors.

A scalar is an entity which only has a magnitude – no direction. Examples of scalar quantities include mass, electric charge, temperature, distance, etc.

A vector, on the other hand, is an entity that is characterized by a magnitude and a direction. Examples of vector quantities are displacement, velocity, magnetic field, etc.

A scalar can be depicted just by a number, for e.g. a temperature of 300 K. On the other hand, vectorial quantities like acceleration are usually denoted by a vector. Given a vector  $\mathbf{V}$ , the magnitude of the corresponding quantity can be calculated as the magnitude of the vector itself  $\|\mathbf{V}\|$ , while the direction would be specified by a unit vector in the direction of the original vector,  $\hat{\mathbf{V}} = \frac{\mathbf{V}}{\|\mathbf{V}\|}$ .

For example, consider a displacement of  $(3\hat{\mathbf{i}} + 4\hat{\mathbf{j}} + 5\hat{\mathbf{k}})$  m, where , as per standard convention,  $\hat{\mathbf{i}}$ ,  $\hat{\mathbf{j}}$  and  $\hat{\mathbf{k}}$  represent unit vectors in the  $\mathbf{X}$ ,  $\mathbf{Y}$  and  $\mathbf{Z}$  directions respectively. Therefore, it can be concluded that the distance traveled is  $\|3\hat{\mathbf{i}} + 4\hat{\mathbf{j}} + 5\hat{\mathbf{k}}\| \text{ m} = 5\sqrt{2} \text{ m}$ . The direction of travel is given by the unit vector  $\frac{3}{5\sqrt{2}}\hat{\mathbf{i}} + \frac{4}{5\sqrt{2}}\hat{\mathbf{j}} + \frac{5}{5\sqrt{2}}\hat{\mathbf{k}}$ .

#### Fields

In general, a *field* is a vector or scalar quantity that can be specified everywhere in space as a function of position (Note that in general a field may also be dependent on time and other custom variables). In this module, we deal with 3-dimensional spaces only. Hence, a field is defined as a function of the  $x$ ,  $y$  and  $z$  coordinates corresponding to a location in 3D space.

For example, temperate in 3 dimensional space (a temperature field) can be written as  $T(x, y, z)$  – a scalar function of the position. An example of a scalar field in electromagnetism is the electric potential.

In a similar manner, a vector field can be defined as a vectorial function of the location  $(x, y, z)$  of any point in space.

For instance, every point on the earth may be considered to be in the gravitational force field of the earth. We may specify the field by the magnitude and the direction of acceleration due to gravity (i.e. force per unit mass )  $g(x, y, z)$  at every point in space.

To give an example from electromagnetism, consider an electric potential of form  $2x^2y$ , a scalar field in 3D space. The corresponding conservative electric field can be computed as the gradient of the electric potential function, and expressed as  $4xy\hat{\mathbf{i}} + 2x^2\hat{\mathbf{j}}$ . The magnitude of this electric field can in turn be expressed as a scalar field of the form  $\sqrt{4x^4 + 16x^2y^2}$ .

## Implementation of fields in sympy.physics.vector

In `sympy.physics.vector` (page 1592), every `ReferenceFrame` (page 1629) instance is assigned basis vectors corresponding to the  $X$ ,  $Y$  and  $Z$  directions. These can be accessed using the attributes named `x`, `y` and `z` respectively. Hence, to define a vector  $\mathbf{v}$  of the form  $3\hat{\mathbf{i}} + 4\hat{\mathbf{j}} + 5\hat{\mathbf{k}}$  with respect to a given frame  $\mathbf{R}$ , you would do

```
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> v = 3*R.x + 4*R.y + 5*R.z
```

Vector math and basic calculus operations with respect to vectors have already been elaborated upon in other sections of this module's documentation.

On the other hand, base scalars (or coordinate variables) are implemented as special SymPy `Symbol` (page 976)s assigned to every frame, one for each direction from  $X$ ,  $Y$  and  $Z$ . For a frame  $\mathbf{R}$ , the  $X$ ,  $Y$  and  $Z$  base scalar `Symbol` (page 976)s can be accessed using the `R[0]`, `R[1]` and `R[2]` expressions respectively.

Therefore, to generate the expression for the aforementioned electric potential field  $2x^2y$ , you would have to do

```
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> electric_potential = 2*R[0]**2*R[1]
>>> electric_potential
2*R_x**2*R_y
```

In string representation, `R_x` denotes the  $X$  base scalar assigned to `ReferenceFrame` (page 1629) `R`. Essentially, `R_x` is the string representation of `R[0]`.

Scalar fields can be treated just as any other SymPy expression, for any math/calculus functionality. Hence, to differentiate the above electric potential with respect to  $x$  (i.e. `R[0]`), you would have to use the `diff` (page 1048) function.

```
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> electric_potential = 2*R[0]**2*R[1]
>>> from sympy import diff
>>> diff(electric_potential, R[0])
4*R_x*R_y
```

Like vectors (and vector fields), scalar fields can also be re-expressed in other frames of reference, apart from the one they were defined in - assuming that an orientation relationship exists between the concerned frames. This can be done using the `sympy.physics.vector.Vector.Vector.express` (page 1644) method, in a way similar to vectors - but with the `variables` parameter set to `True`.

```
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> electric_potential = 2*R[0]**2*R[1]
>>> from sympy.physics.vector import dynamicsymbols, express
>>> q = dynamicsymbols('q')
>>> R1 = R.orientnew('R1', rot_type = 'Axis', amounts = [q, R.z])
>>> express(electric_potential, R1, variables=True)
2*(R1_x*sin(q(t)) + R1_y*cos(q(t)))*(R1_x*cos(q(t)) - R1_y*sin(q(t)))*2
```



Moreover, considering scalars can also be functions of time just as vectors, differentiation with respect to time is also possible. Depending on the [Symbol](#) (page 976)s present in the expression and the frame with respect to which the time differentiation is being done, the output will change/remain the same.

```
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> electric_potential = 2*R[0]**2*R[1]
>>> q = dynamicsymbols('q')
>>> R1 = R.orientnew('R1', rot_type = 'Axis', amounts = [q, R.z])
>>> from sympy.physics.vector import time_derivative
>>> time_derivative(electric_potential, R)
0
>>> time_derivative(electric_potential, R1).simplify()
2*(R1_x*cos(q(t)) - R1_y*sin(q(t)))*(3*R1_x**2*cos(2*q(t))/2 -
R1_x**2/2 - 3*R1_x*R1_y*sin(2*q(t)) - 3*R1_y**2*cos(2*q(t))/2 -
R1_y**2/2)*Derivative(q(t), t)
```

## Field operators and other related functions

Here we describe some basic field-related functionality implemented in `sympy.physics.vector`

### Curl

A curl is a mathematical operator that describes an infinitesimal rotation of a vector in 3D space. The direction is determined by the right-hand rule (along the axis of rotation), and the magnitude is given by the magnitude of rotation.

In the 3D Cartesian system, the curl of a 3D vector  $\mathbf{F}$ , denoted by  $\nabla \times \mathbf{F}$  is given by -

$$\nabla \times \mathbf{F} = \left( \frac{\partial F_z}{\partial y} - \frac{\partial F_y}{\partial z} \right) \hat{\mathbf{i}} + \left( \frac{\partial F_x}{\partial z} - \frac{\partial F_z}{\partial x} \right) \hat{\mathbf{j}} + \left( \frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial y} \right) \hat{\mathbf{k}}$$

where  $F_x$  denotes the  $X$  component of vector  $\mathbf{F}$ .

To compute the curl of a vector field in `sympy.physics.vector` (page 1592), you would do

```
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> from sympy.physics.vector import curl
>>> field = R[0]*R[1]*R[2]*R.x
>>> curl(field, R)
R_x*R_y*R.y - R_x*R_z*R.z
```

## Divergence

Divergence is a vector operator that measures the magnitude of a vector field's source or sink at a given point, in terms of a signed scalar.

The divergence operator always returns a scalar after operating on a vector.

In the 3D Cartesian system, the divergence of a 3D vector  $\mathbf{F}$ , denoted by  $\nabla \cdot \mathbf{F}$  is given by -

$$\nabla \cdot \mathbf{F} = \frac{\partial U}{\partial x} + \frac{\partial V}{\partial y} + \frac{\partial W}{\partial z}$$

where  $U$ ,  $V$  and  $W$  denote the  $X$ ,  $Y$  and  $Z$  components of  $\mathbf{F}$  respectively.

To compute the divergence of a vector field in [sympy.physics.vector](#) (page 1592), you would do

```
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> from sympy.physics.vector import divergence
>>> field = R[0]*R[1]*R[2] * (R.x+R.y+R.z)
>>> divergence(field, R)
R_x*R_y + R_x*R_z + R_y*R_z
```

## Gradient

Consider a scalar field  $f(x, y, z)$  in 3D space. The gradient of this field is defined as the vector of the 3 partial derivatives of  $f$  with respect to  $x$ ,  $y$  and  $z$  in the  $X$ ,  $Y$  and  $Z$  directions respectively.

In the 3D Cartesian system, the gradient of a scalar field  $f$ , denoted by  $\nabla f$  is given by -

$$\nabla f = \frac{\partial f}{\partial x} \hat{\mathbf{i}} + \frac{\partial f}{\partial y} \hat{\mathbf{j}} + \frac{\partial f}{\partial z} \hat{\mathbf{k}}$$

To compute the gradient of a scalar field in [sympy.physics.vector](#) (page 1592), you would do

```
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> from sympy.physics.vector import gradient
>>> scalar_field = R[0]*R[1]*R[2]
>>> gradient(scalar_field, R)
R_y*R_z*R.x + R_x*R_z*R.y + R_x*R_y*R.z
```

## Conservative and Solenoidal fields

In vector calculus, a conservative field is a field that is the gradient of some scalar field. Conservative fields have the property that their line integral over any path depends only on the end-points, and is independent of the path between them. A conservative vector field is also said to be 'irrotational', since the curl of a conservative field is always zero.

In physics, conservative fields represent forces in physical systems where energy is conserved.

To check if a vector field is conservative in [sympy.physics.vector](#) (page 1592), use the [sympy.physics.vector.fieldfunctions.is\\_conservative](#) (page 1674) function.

```
>>> from sympy.physics.vector import ReferenceFrame, is_conservative
>>> R = ReferenceFrame('R')
>>> field = R[1]*R[2]*R.x + R[0]*R[2]*R.y + R[0]*R[1]*R.z
>>> is_conservative(field)
True
>>> curl(field, R)
0
```

A solenoidal field, on the other hand, is a vector field whose divergence is zero at all points in space.

To check if a vector field is solenoidal in *sympy.physics.vector* (page 1592), use the *sympy.physics.vector.fieldsfunctions.is\_solenoidal* (page 1674) function.

```
>>> from sympy.physics.vector import ReferenceFrame, is_solenoidal
>>> R = ReferenceFrame('R')
>>> field = R[1]*R[2]*R.x + R[0]*R[2]*R.y + R[0]*R[1]*R.z
>>> is_solenoidal(field)
True
>>> divergence(field, R)
0
```

## Scalar potential functions

We have previously mentioned that every conservative field can be defined as the gradient of some scalar field. This scalar field is also called the ‘scalar potential field’ corresponding to the aforementioned conservative field.

The *sympy.physics.vector.fieldsfunctions.scalar\_potential* (page 1672) function in *sympy.physics.vector* (page 1592) calculates the scalar potential field corresponding to a given conservative vector field in 3D space - minus the extra constant of integration, of course.

Example of usage -

```
>>> from sympy.physics.vector import ReferenceFrame, scalar_potential
>>> R = ReferenceFrame('R')
>>> conservative_field = 4*R[0]*R[1]*R[2]*R.x + 2*R[0]**2*R[2]*R.y +
↳ 2*R[0]**2*R[1]*R.z
>>> scalar_potential(conservative_field, R)
2*R_x**2*R_y*R_z
```

Providing a non-conservative vector field as an argument to *sympy.physics.vector.fieldsfunctions.scalar\_potential* (page 1672) raises a `ValueError`.

The scalar potential difference, or simply ‘potential difference’, corresponding to a conservative vector field can be defined as the difference between the values of its scalar potential function at two points in space. This is useful in calculating a line integral with respect to a conservative function, since it depends only on the endpoints of the path.

This computation is performed as follows in *sympy.physics.vector* (page 1592).

```
>>> from sympy.physics.vector import ReferenceFrame, Point
>>> from sympy.physics.vector import scalar_potential_difference
```

(continues on next page)

(continued from previous page)

```
>>> R = ReferenceFrame('R')
>>> O = Point('O')
>>> P = O.locatenew('P', 1*R.x + 2*R.y + 3*R.z)
>>> vectfield = 4*R[0]*R[1]*R.x + 2*R[0]**2*R.y
>>> scalar_potential_difference(vectfield, R, O, P, O)
4
```

If provided with a scalar expression instead of a vector field, [\*sympy.physics.vector.fieldfunctions.scalar\\_potential\\_difference\*](#) (page 1673) returns the difference between the values of that scalar field at the two given points in space.

## Vector API

### Essential Classes

#### CoordinateSym

**class** sympy.physics.vector.frame.CoordinateSym(name, frame, index)

A coordinate symbol/base scalar associated wrt a Reference Frame.

Ideally, users should not instantiate this class. Instances of this class must only be accessed through the corresponding frame as 'frame[index]'.

CoordinateSyms having the same frame and index parameters are equal (even though they may be instantiated separately).

##### Parameters

**name** : string

The display name of the CoordinateSym

**frame** : ReferenceFrame

The reference frame this base scalar belongs to

**index** : 0, 1 or 2

The index of the dimension denoted by this coordinate variable

#### Examples

```
>>> from sympy.physics.vector import ReferenceFrame, CoordinateSym
>>> A = ReferenceFrame('A')
>>> A[1]
A_y
>>> type(A[0])
<class 'sympy.physics.vector.frame.CoordinateSym'>
>>> a_y = CoordinateSym('a_y', A, 1)
>>> a_y == A[1]
True
```

## ReferenceFrame

**class** `sympy.physics.vector.frame.ReferenceFrame`(*name*, *indices=None*, *latexs=None*, *variables=None*)

A reference frame in classical mechanics.

ReferenceFrame is a class used to represent a reference frame in classical mechanics. It has a standard basis of three unit vectors in the frame's x, y, and z directions.

It also can have a rotation relative to a parent frame; this rotation is defined by a direction cosine matrix relating this frame's basis vectors to the parent frame's basis vectors. It can also have an angular velocity vector, defined in another frame.

**ang\_acc\_in**(*otherframe*)

Returns the angular acceleration Vector of the ReferenceFrame.

Effectively returns the Vector:

$N_{\alpha}B$

which represent the angular acceleration of B in N, where B is self, and N is other-frame.

### Parameters

**otherframe** : ReferenceFrame

The ReferenceFrame which the angular acceleration is returned in.

## Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> N = ReferenceFrame('N')
>>> A = ReferenceFrame('A')
>>> V = 10 * N.x
>>> A.set_ang_acc(N, V)
>>> A.ang_acc_in(N)
10*N.x
```

**ang\_vel\_in**(*otherframe*)

Returns the angular velocity Vector of the ReferenceFrame.

Effectively returns the Vector:

${}^N\omega^B$

which represent the angular velocity of B in N, where B is self, and N is otherframe.

### Parameters

**otherframe** : ReferenceFrame

The ReferenceFrame which the angular velocity is returned in.

## Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> N = ReferenceFrame('N')
>>> A = ReferenceFrame('A')
>>> V = 10 * N.x
>>> A.set_ang_vel(N, V)
>>> A.ang_vel_in(N)
10*N.x
```

**dcm**(*otherframe*)

Returns the direction cosine matrix of this reference frame relative to the provided reference frame.

The returned matrix can be used to express the orthogonal unit vectors of this frame in terms of the orthogonal unit vectors of *otherframe*.

### Parameters

**otherframe** : ReferenceFrame

The reference frame which the direction cosine matrix of this frame is formed relative to.

## Examples

The following example rotates the reference frame A relative to N by a simple rotation and then calculates the direction cosine matrix of N relative to A.

```
>>> from sympy import symbols, sin, cos
>>> from sympy.physics.vector import ReferenceFrame
>>> q1 = symbols('q1')
>>> N = ReferenceFrame('N')
>>> A = ReferenceFrame('A')
>>> A.orient_axis(N, q1, N.x)
>>> N.dcm(A)
Matrix([
[1, 0, 0],
[0, cos(q1), -sin(q1)],
[0, sin(q1), cos(q1)])
```

The second row of the above direction cosine matrix represents the N.y unit vector in N expressed in A. Like so:

```
>>> Ny = 0*A.x + cos(q1)*A.y - sin(q1)*A.z
```

Thus, expressing N.y in A should return the same result:

```
>>> N.y.express(A)
cos(q1)*A.y - sin(q1)*A.z
```

## Notes

It is important to know what form of the direction cosine matrix is returned. If `B.dcm(A)` is called, it means the “direction cosine matrix of B rotated relative to A”. This is the matrix  ${}^B\mathbf{C}^A$  shown in the following relationship:

$$\begin{bmatrix} \hat{\mathbf{b}}_1 \\ \hat{\mathbf{b}}_2 \\ \hat{\mathbf{b}}_3 \end{bmatrix} = {}^B\mathbf{C}^A \begin{bmatrix} \hat{\mathbf{a}}_1 \\ \hat{\mathbf{a}}_2 \\ \hat{\mathbf{a}}_3 \end{bmatrix}.$$

${}^B\mathbf{C}^A$  is the matrix that expresses the B unit vectors in terms of the A unit vectors.

**orient**(parent, rot\_type, amounts, rot\_order=“”)

Sets the orientation of this reference frame relative to another (parent) reference frame.

---

**Note:** It is now recommended to use the `.orient_axis`, `.orient_body_fixed`, `.orient_space_fixed`, `.orient_quaternion` methods for the different rotation types.

---

## Parameters

**parent** : ReferenceFrame

Reference frame that this reference frame will be rotated relative to.

**rot\_type** : str

The method used to generate the direction cosine matrix. Supported methods are:

- 'Axis': simple rotations about a single common axis
- 'DCM': for setting the direction cosine matrix directly
- 'Body': three successive rotations about new intermediate axes, also called “Euler and Tait-Bryan angles”
- 'Space': three successive rotations about the parent frames’ unit vectors
- 'Quaternion': rotations defined by four parameters which result in a singularity free direction cosine matrix

**amounts** :

Expressions defining the rotation angles or direction cosine matrix. These must match the `rot_type`. See examples below for details. The input types are:

- 'Axis': 2-tuple (expr/sym/func, Vector)
- 'DCM': Matrix, shape(3,3)
- 'Body': 3-tuple of expressions, symbols, or functions
- 'Space': 3-tuple of expressions, symbols, or functions
- 'Quaternion': 4-tuple of expressions, symbols, or functions

**rot\_order** : str or int, optional

If applicable, the order of the successive of rotations. The string '123' and integer 123 are equivalent, for example. Required for 'Body' and 'Space'.

**Warns****UserWarning**

If the orientation creates a kinematic loop.

**orient\_axis**(*parent, axis, angle*)

Sets the orientation of this reference frame with respect to a parent reference frame by rotating through an angle about an axis fixed in the parent reference frame.

**Parameters**

**parent** : ReferenceFrame

Reference frame that this reference frame will be rotated relative to.

**axis** : Vector

Vector fixed in the parent frame about which this frame is rotated. It need not be a unit vector and the rotation follows the right hand rule.

**angle** : sympifiable

Angle in radians by which it the frame is to be rotated.

**Warns****UserWarning**

If the orientation creates a kinematic loop.

**Examples**

Setup variables for the examples:

```
>>> from sympy import symbols
>>> from sympy.physics.vector import ReferenceFrame
>>> q1 = symbols('q1')
>>> N = ReferenceFrame('N')
>>> B = ReferenceFrame('B')
>>> B.orient_axis(N, N.x, q1)
```

The `orient_axis()` method generates a direction cosine matrix and its transpose which defines the orientation of B relative to N and vice versa. Once orient is called, `dcm()` outputs the appropriate direction cosine matrix:

```
>>> B.dcm(N)
Matrix([
[1, 0, 0],
[0, cos(q1), sin(q1)],
[0, -sin(q1), cos(q1)]]
>>> N.dcm(B)
Matrix([
[1, 0, 0],
[0, cos(q1), -sin(q1)],
[0, sin(q1), cos(q1)]]
```



The following two lines show that the sense of the rotation can be defined by negating the vector direction or the angle. Both lines produce the same result.

```
>>> B.orient_axis(N, -N.x, q1)
>>> B.orient_axis(N, N.x, -q1)
```

### **orient\_body\_fixed**(parent, angles, rotation\_order)

Rotates this reference frame relative to the parent reference frame by right hand rotating through three successive body fixed simple axis rotations. Each subsequent axis of rotation is about the “body fixed” unit vectors of a new intermediate reference frame. This type of rotation is also referred to rotating through the [Euler and Tait-Bryan Angles](#).

#### **Parameters**

**parent** : ReferenceFrame

Reference frame that this reference frame will be rotated relative to.

**angles** : 3-tuple of sympifiable

Three angles in radians used for the successive rotations.

**rotation\_order** : 3 character string or 3 digit integer

Order of the rotations about each intermediate reference frames’ unit vectors. The Euler rotation about the X, Z’, X’’ axes can be specified by the strings 'XZX', '131', or the integer 131. There are 12 unique valid rotation orders (6 Euler and 6 Tait-Bryan): zxz, yxy, yzy, zyz, xzx, yxy, xyz, yzx, zxy, xzy, zyx, and yxz.

#### **Warns**

##### **UserWarning**

If the orientation creates a kinematic loop.

### **Examples**

Setup variables for the examples:

```
>>> from sympy import symbols
>>> from sympy.physics.vector import ReferenceFrame
>>> q1, q2, q3 = symbols('q1, q2, q3')
>>> N = ReferenceFrame('N')
>>> B = ReferenceFrame('B')
>>> B1 = ReferenceFrame('B1')
>>> B2 = ReferenceFrame('B2')
>>> B3 = ReferenceFrame('B3')
```

For example, a classic Euler Angle rotation can be done by:

```
>>> B.orient_body_fixed(N, (q1, q2, q3), 'XYZ')
>>> B.dcm(N)
Matrix([
[cos(q2), sin(q1)*sin(q2),
 ↪ -sin(q2)*cos(q1)],
[sin(q2)*sin(q3), -sin(q1)*sin(q3)*cos(q2) + cos(q1)*cos(q3),
```

(continues on next page)

(continued from previous page)

```

↪sin(q1)*cos(q3) + sin(q3)*cos(q1)*cos(q2)],
[sin(q2)*cos(q3), -sin(q1)*cos(q2)*cos(q3) - sin(q3)*cos(q1), -
↪sin(q1)*sin(q3) + cos(q1)*cos(q2)*cos(q3)]]))

```

This rotates reference frame B relative to reference frame N through  $q_1$  about N.x, then rotates B again through  $q_2$  about B.y, and finally through  $q_3$  about B.x. It is equivalent to three successive `orient_axis()` calls:

```

>>> B1.orient_axis(N, N.x, q1)
>>> B2.orient_axis(B1, B1.y, q2)
>>> B3.orient_axis(B2, B2.x, q3)
>>> B3.dcm(N)
Matrix([
[cos(q2), sin(q1)*sin(q2),
↪ -sin(q2)*cos(q1)],
[sin(q2)*sin(q3), -sin(q1)*sin(q3)*cos(q2) + cos(q1)*cos(q3),
↪sin(q1)*cos(q3) + sin(q3)*cos(q1)*cos(q2)],
[sin(q2)*cos(q3), -sin(q1)*cos(q2)*cos(q3) - sin(q3)*cos(q1), -
↪sin(q1)*sin(q3) + cos(q1)*cos(q2)*cos(q3)]]))

```

Acceptable rotation orders are of length 3, expressed in as a string 'XYZ' or '123' or integer 123. Rotations about an axis twice in a row are prohibited.

```

>>> B.orient_body_fixed(N, (q1, q2, 0), 'ZXZ')
>>> B.orient_body_fixed(N, (q1, q2, 0), '121')
>>> B.orient_body_fixed(N, (q1, q2, q3), 123)

```

### **orient\_explicit**(parent, dcm)

Sets the orientation of this reference frame relative to a parent reference frame by explicitly setting the direction cosine matrix.

#### **Parameters**

**parent** : ReferenceFrame

Reference frame that this reference frame will be rotated relative to.

**dcm** : Matrix, shape(3, 3)

Direction cosine matrix that specifies the relative rotation between the two reference frames.

#### **Warns**

##### **UserWarning**

If the orientation creates a kinematic loop.

## Examples

Setup variables for the examples:

```
>>> from sympy import symbols, Matrix, sin, cos
>>> from sympy.physics.vector import ReferenceFrame
>>> q1 = symbols('q1')
>>> A = ReferenceFrame('A')
>>> B = ReferenceFrame('B')
>>> N = ReferenceFrame('N')
```

A simple rotation of A relative to N about N.x is defined by the following direction cosine matrix:

```
>>> dcm = Matrix([[1, 0, 0],
... [0, cos(q1), -sin(q1)],
... [0, sin(q1), cos(q1)]])
>>> A.orient_explicit(N, dcm)
>>> A.dcm(N)
Matrix([
[1, 0, 0],
[0, cos(q1), sin(q1)],
[0, -sin(q1), cos(q1)]])
```

This is equivalent to using `orient_axis()`:

```
>>> B.orient_axis(N, N.x, q1)
>>> B.dcm(N)
Matrix([
[1, 0, 0],
[0, cos(q1), sin(q1)],
[0, -sin(q1), cos(q1)]])
```

**Note carefully that `N.dcm(B)` (the transpose) would be passed into `orient_explicit()` for `A.dcm(N)` to match `B.dcm(N)`:**

```
>>> A.orient_explicit(N, N.dcm(B))
>>> A.dcm(N)
Matrix([
[1, 0, 0],
[0, cos(q1), sin(q1)],
[0, -sin(q1), cos(q1)]])
```

### `orient_quaternion(parent, numbers)`

Sets the orientation of this reference frame relative to a parent reference frame via an orientation quaternion. An orientation quaternion is defined as a finite rotation a unit vector, (`lambda_x`, `lambda_y`, `lambda_z`), by an angle `theta`. The orientation quaternion is described by four parameters:

- `q0 = cos(theta/2)`
- `q1 = lambda_x*sin(theta/2)`
- `q2 = lambda_y*sin(theta/2)`
- `q3 = lambda_z*sin(theta/2)`

See [Quaternions and Spatial Rotation](#) on Wikipedia for more information.

**Parameters**

**parent** : ReferenceFrame

Reference frame that this reference frame will be rotated relative to.

**numbers** : 4-tuple of sympifiable

The four quaternion scalar numbers as defined above:  $q_0$ ,  $q_1$ ,  $q_2$ ,  $q_3$ .

**Warns**

**UserWarning**

If the orientation creates a kinematic loop.

**Examples**

Setup variables for the examples:

```
>>> from sympy import symbols
>>> from sympy.physics.vector import ReferenceFrame
>>> q0, q1, q2, q3 = symbols('q0 q1 q2 q3')
>>> N = ReferenceFrame('N')
>>> B = ReferenceFrame('B')
```

Set the orientation:

```
>>> B.orient_quaternion(N, (q0, q1, q2, q3))
>>> B.dcm(N)
Matrix([
[q0**2 + q1**2 - q2**2 - q3**2, 2*q0*q3 + 2*q1*q2,
↪ -2*q0*q2 + 2*q1*q3],
[
↪ -2*q0*q3 + 2*q1*q2, q0**2 - q1**2 + q2**2 - q3**2,
2*q0*q1 + 2*q2*q3],
[
↪ 2*q0*q2 + 2*q1*q3, -2*q0*q1 + 2*q2*q3, q0**2 -
q1**2 - q2**2 + q3**2]])
```

**orient\_space\_fixed**(*parent*, *angles*, *rotation\_order*)

Rotates this reference frame relative to the parent reference frame by right hand rotating through three successive space fixed simple axis rotations. Each subsequent axis of rotation is about the “space fixed” unit vectors of the parent reference frame.

**Parameters**

**parent** : ReferenceFrame

Reference frame that this reference frame will be rotated relative to.

**angles** : 3-tuple of sympifiable

Three angles in radians used for the successive rotations.

**rotation\_order** : 3 character string or 3 digit integer

Order of the rotations about the parent reference frame’s unit vectors. The order can be specified by the strings 'XZX', '131', or the integer 131. There are 12 unique valid rotation orders.

**Warns**

**UserWarning**

If the orientation creates a kinematic loop.

## Examples

Setup variables for the examples:

```
>>> from sympy import symbols
>>> from sympy.physics.vector import ReferenceFrame
>>> q1, q2, q3 = symbols('q1, q2, q3')
>>> N = ReferenceFrame('N')
>>> B = ReferenceFrame('B')
>>> B1 = ReferenceFrame('B1')
>>> B2 = ReferenceFrame('B2')
>>> B3 = ReferenceFrame('B3')
```

```
>>> B.orient_space_fixed(N, (q1, q2, q3), '312')
>>> B.dcm(N)
Matrix([
[sin(q1)*sin(q2)*sin(q3) + cos(q1)*cos(q3), sin(q1)*cos(q2),
 ↪ sin(q1)*sin(q2)*cos(q3) - sin(q3)*cos(q1)],
[-sin(q1)*cos(q3) + sin(q2)*sin(q3)*cos(q1), cos(q1)*cos(q2),
 ↪ sin(q1)*sin(q3) + sin(q2)*cos(q1)*cos(q3)],
[
 ↪ sin(q3)*cos(q2), -sin(q2),
 ↪ cos(q2)*cos(q3)]])
```

is equivalent to:

```
>>> B1.orient_axis(N, N.z, q1)
>>> B2.orient_axis(B1, N.x, q2)
>>> B3.orient_axis(B2, N.y, q3)
>>> B3.dcm(N).simplify()
Matrix([
[sin(q1)*sin(q2)*sin(q3) + cos(q1)*cos(q3), sin(q1)*cos(q2),
 ↪ sin(q1)*sin(q2)*cos(q3) - sin(q3)*cos(q1)],
[-sin(q1)*cos(q3) + sin(q2)*sin(q3)*cos(q1), cos(q1)*cos(q2),
 ↪ sin(q1)*sin(q3) + sin(q2)*cos(q1)*cos(q3)],
[
 ↪ sin(q3)*cos(q2), -sin(q2),
 ↪ cos(q2)*cos(q3)]])
```

It is worth noting that space-fixed and body-fixed rotations are related by the order of the rotations, i.e. the reverse order of body fixed will give space fixed and vice versa.

```
>>> B.orient_space_fixed(N, (q1, q2, q3), '231')
>>> B.dcm(N)
Matrix([
[cos(q1)*cos(q2), sin(q1)*sin(q3) + sin(q2)*cos(q1)*cos(q3), -
 ↪ sin(q1)*cos(q3) + sin(q2)*sin(q3)*cos(q1)],
[-sin(q2), cos(q2)*cos(q3),
 ↪ sin(q3)*cos(q2)],
[sin(q1)*cos(q2), sin(q1)*sin(q2)*cos(q3) - sin(q3)*cos(q1),
 ↪ sin(q1)*sin(q2)*sin(q3) + cos(q1)*cos(q3)]])
```

```
>>> B.orient_body_fixed(N, (q3, q2, q1), '132')
>>> B.dcm(N)
Matrix([
[cos(q1)*cos(q2), sin(q1)*sin(q3) + sin(q2)*cos(q1)*cos(q3), -
↪ sin(q1)*cos(q3) + sin(q2)*sin(q3)*cos(q1)],
[-sin(q2), cos(q2)*cos(q3),
↪ sin(q3)*cos(q2)],
[sin(q1)*cos(q2), sin(q1)*sin(q2)*cos(q3) - sin(q3)*cos(q1),
↪ sin(q1)*sin(q2)*sin(q3) + cos(q1)*cos(q3)]])
```

**orientnew**(*newname*, *rot\_type*, *amounts*, *rot\_order*="", *variables*=None, *indices*=None, *latexs*=None)

Returns a new reference frame oriented with respect to this reference frame.

See `ReferenceFrame.orient()` for detailed examples of how to orient reference frames.

#### Parameters

**newname** : str

Name for the new reference frame.

**rot\_type** : str

The method used to generate the direction cosine matrix. Supported methods are:

- 'Axis': simple rotations about a single common axis
- 'DCM': for setting the direction cosine matrix directly
- 'Body': three successive rotations about new intermediate axes, also called "Euler and Tait-Bryan angles"
- 'Space': three successive rotations about the parent frames' unit vectors
- 'Quaternion': rotations defined by four parameters which result in a singularity free direction cosine matrix

**amounts** :

Expressions defining the rotation angles or direction cosine matrix. These must match the *rot\_type*. See examples below for details. The input types are:

- 'Axis': 2-tuple (expr/sym/func, Vector)
- 'DCM': Matrix, shape(3,3)
- 'Body': 3-tuple of expressions, symbols, or functions
- 'Space': 3-tuple of expressions, symbols, or functions
- 'Quaternion': 4-tuple of expressions, symbols, or functions

**rot\_order** : str or int, optional

If applicable, the order of the successive of rotations. The string '123' and integer 123 are equivalent, for example. Required for 'Body' and 'Space'.

**indices** : tuple of str

Enables the reference frame's basis unit vectors to be accessed by Python's square bracket indexing notation using the provided three indice strings and alters the printing of the unit vectors to reflect this choice.

**latexs** : tuple of str

Alters the LaTeX printing of the reference frame's basis unit vectors to the provided three valid LaTeX strings.

## Examples

```
>>> from sympy import symbols
>>> from sympy.physics.vector import ReferenceFrame, vlatex
>>> q0, q1, q2, q3 = symbols('q0 q1 q2 q3')
>>> N = ReferenceFrame('N')
```

Create a new reference frame A rotated relative to N through a simple rotation.

```
>>> A = N.orientnew('A', 'Axis', (q0, N.x))
```

Create a new reference frame B rotated relative to N through body-fixed rotations.

```
>>> B = N.orientnew('B', 'Body', (q1, q2, q3), '123')
```

Create a new reference frame C rotated relative to N through a simple rotation with unique indices and LaTeX printing.

```
>>> C = N.orientnew('C', 'Axis', (q0, N.x), indices=('1', '2', '3'),
... latexs=(r'\hat{\mathbf{c}}_1', r'\hat{\mathbf{c}}_2',
... r'\hat{\mathbf{c}}_3'))
>>> C['1']
C['1']
>>> print(vlatex(C['1']))
\hat{\mathbf{c}}_1
```

**partial\_velocity**(*frame*, \**gen\_speeds*)

Returns the partial angular velocities of this frame in the given frame with respect to one or more provided generalized speeds.

### Parameters

**frame** : ReferenceFrame

The frame with which the angular velocity is defined in.

**gen\_speeds** : functions of time

The generalized speeds.

### Returns

**partial\_velocities** : tuple of Vector

The partial angular velocity vectors corresponding to the provided generalized speeds.

## Examples

```
>>> from sympy.physics.vector import ReferenceFrame, dynamicsymbols
>>> N = ReferenceFrame('N')
>>> A = ReferenceFrame('A')
>>> u1, u2 = dynamicsymbols('u1, u2')
>>> A.set_ang_vel(N, u1 * A.x + u2 * N.y)
>>> A.partial_velocity(N, u1)
A.x
>>> A.partial_velocity(N, u1, u2)
(A.x, N.y)
```

**set\_ang\_acc**(*otherframe*, *value*)

Define the angular acceleration Vector in a ReferenceFrame.

Defines the angular acceleration of this ReferenceFrame, in another. Angular acceleration can be defined with respect to multiple different ReferenceFrames. Care must be taken to not create loops which are inconsistent.

### Parameters

**otherframe** : ReferenceFrame

A ReferenceFrame to define the angular acceleration in

**value** : Vector

The Vector representing angular acceleration

## Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> N = ReferenceFrame('N')
>>> A = ReferenceFrame('A')
>>> V = 10 * N.x
>>> A.set_ang_acc(N, V)
>>> A.ang_acc_in(N)
10*N.x
```

**set\_ang\_vel**(*otherframe*, *value*)

Define the angular velocity vector in a ReferenceFrame.

Defines the angular velocity of this ReferenceFrame, in another. Angular velocity can be defined with respect to multiple different ReferenceFrames. Care must be taken to not create loops which are inconsistent.

### Parameters

**otherframe** : ReferenceFrame

A ReferenceFrame to define the angular velocity in

**value** : Vector

The Vector representing angular velocity



## Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> N = ReferenceFrame('N')
>>> A = ReferenceFrame('A')
>>> V = 10 * N.x
>>> A.set_ang_vel(N, V)
>>> A.ang_vel_in(N)
10*N.x
```

### `variable_map(otherframe)`

Returns a dictionary which expresses the coordinate variables of this frame in terms of the variables of otherframe.

If Vector.simp is True, returns a simplified version of the mapped values. Else, returns them without simplification.

Simplification of the expressions may take time.

#### Parameters

**otherframe** : ReferenceFrame

The other frame to map the variables to

## Examples

```
>>> from sympy.physics.vector import ReferenceFrame, dynamicsymbols
>>> A = ReferenceFrame('A')
>>> q = dynamicsymbols('q')
>>> B = A.orientnew('B', 'Axis', [q, A.z])
>>> A.variable_map(B)
{A_x: B_x*cos(q(t)) - B_y*sin(q(t)), A_y: B_x*sin(q(t)) + B_
→ y*cos(q(t)), A_z: B_z}
```

### property `x`

The basis Vector for the ReferenceFrame, in the x direction.

### property `y`

The basis Vector for the ReferenceFrame, in the y direction.

### property `z`

The basis Vector for the ReferenceFrame, in the z direction.

## Vector

### `class sympy.physics.vector.vector.Vector(inlist)`

The class used to define vectors.

It along with ReferenceFrame are the building blocks of describing a classical mechanics system in PyDy and sympy.physics.vector.

## Attributes

|                   |                                                             |
|-------------------|-------------------------------------------------------------|
| <code>simp</code> | (Boolean) Let certain methods use trigsimp on their outputs |
|-------------------|-------------------------------------------------------------|

### `angle_between(vec)`

Returns the smallest angle between Vector '*vec*' and self.

**Warning:** Python ignores the leading negative sign so that might give wrong results. `-A.x.angle_between()` would be treated as `-(A.x.angle_between())`, instead of `(-A.x).angle_between()`.

## Parameter

### `vec`

[Vector] The Vector between which angle is needed.

## Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> A = ReferenceFrame("A")
>>> v1 = A.x
>>> v2 = A.y
>>> v1.angle_between(v2)
pi/2
```

```
>>> v3 = A.x + A.y + A.z
>>> v1.angle_between(v3)
acos(sqrt(3)/3)
```

### `applyfunc(f)`

Apply a function to each component of a vector.

### `cross(other)`

The cross product operator for two Vectors.

Returns a Vector, expressed in the same ReferenceFrames as self.

#### Parameters

**other** : Vector

The Vector which we are crossing with

## Examples

```
>>> from sympy import symbols
>>> from sympy.physics.vector import ReferenceFrame, cross
>>> q1 = symbols('q1')
>>> N = ReferenceFrame('N')
>>> cross(N.x, N.y)
N.z
>>> A = ReferenceFrame('A')
>>> A.orient_axis(N, q1, N.x)
>>> cross(A.x, N.y)
N.z
>>> cross(N.y, A.x)
- sin(q1)*A.y - cos(q1)*A.z
```

**diff**(var, frame, var\_in\_dcm=True)

Returns the partial derivative of the vector with respect to a variable in the provided reference frame.

### Parameters

**var** : Symbol

What the partial derivative is taken with respect to.

**frame** : ReferenceFrame

The reference frame that the partial derivative is taken in.

**var\_in\_dcm** : boolean

If true, the differentiation algorithm assumes that the variable may be present in any of the direction cosine matrices that relate the frame to the frames of any component of the vector. But if it is known that the variable is not present in the direction cosine matrices, false can be set to skip full reexpression in the desired frame.

## Examples

```
>>> from sympy import Symbol
>>> from sympy.physics.vector import dynamicsymbols, ReferenceFrame
>>> from sympy.physics.vector import Vector
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> Vector.simp = True
>>> t = Symbol('t')
>>> q1 = dynamicsymbols('q1')
>>> N = ReferenceFrame('N')
>>> A = N.orientnew('A', 'Axis', [q1, N.y])
>>> A.x.diff(t, N)
- sin(q1)*q1'*N.x - cos(q1)*q1'*N.z
>>> A.x.diff(t, N).express(A)
- q1'*A.z
>>> B = ReferenceFrame('B')
>>> u1, u2 = dynamicsymbols('u1, u2')
```

(continues on next page)

(continued from previous page)

```
>>> v = u1 * A.x + u2 * B.y
>>> v.diff(u2, N, var_in_dcm=False)
B.y
```

**doit**(*\*\*hints*)

Calls `.doit()` on each term in the Vector

**dot**(*other*)

Dot product of two vectors.

Returns a scalar, the dot product of the two Vectors

**Parameters****other** : Vector

The Vector which we are dotting with

**Examples**

```
>>> from sympy.physics.vector import ReferenceFrame, dot
>>> from sympy import symbols
>>> q1 = symbols('q1')
>>> N = ReferenceFrame('N')
>>> dot(N.x, N.x)
1
>>> dot(N.x, N.y)
0
>>> A = N.orientnew('A', 'Axis', [q1, N.x])
>>> dot(N.y, A.y)
cos(q1)
```

**dt**(*otherframe*)

Returns a Vector which is the time derivative of the self Vector, taken in frame *otherframe*.

Calls the global `time_derivative` method

**Parameters****otherframe** : ReferenceFrame

The frame to calculate the time derivative in

**express**(*otherframe*, *variables=False*)

Returns a Vector equivalent to this one, expressed in *otherframe*. Uses the global `express` method.

**Parameters****otherframe** : ReferenceFrame

The frame for this Vector to be described in

**variables** : boolean

If True, the coordinate symbols(if present) in this Vector are re-expressed in terms *otherframe*

## Examples

```
>>> from sympy.physics.vector import ReferenceFrame, dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> q1 = dynamicsymbols('q1')
>>> N = ReferenceFrame('N')
>>> A = N.orientnew('A', 'Axis', [q1, N.y])
>>> A.x.express(N)
cos(q1)*N.x - sin(q1)*N.z
```

### **free\_dynamicsymbols**(*reference\_frame*)

Returns the free dynamic symbols (functions of time  $t$ ) in the measure numbers of the vector expressed in the given reference frame.

#### **Parameters**

**reference\_frame** : ReferenceFrame

The frame with respect to which the free dynamic symbols of the given vector is to be determined.

#### **Returns**

set

Set of functions of time  $t$ , e.g. `Function('f')(me.dynamicsymbols._t)`.

### **free\_symbols**(*reference\_frame*)

Returns the free symbols in the measure numbers of the vector expressed in the given reference frame.

#### **Parameters**

**reference\_frame** : ReferenceFrame

The frame with respect to which the free symbols of the given vector is to be determined.

#### **Returns**

set of Symbol

set of symbols present in the measure numbers of *reference\_frame*.

### **property func**

Returns the class Vector.

### **magnitude()**

Returns the magnitude (Euclidean norm) of self.

**Warning:** Python ignores the leading negative sign so that might give wrong results. `-A.x.magnitude()` would be treated as `-(A.x.magnitude())`, instead of `(-A.x).magnitude()`.

### **normalize()**

Returns a Vector of magnitude 1, codirectional with self.

**outer**(*other*)

Outer product between two Vectors.

A rank increasing operation, which returns a Dyadic from two Vectors

**Parameters****other** : Vector

The Vector to take the outer product with

**Examples**

```
>>> from sympy.physics.vector import ReferenceFrame, outer
>>> N = ReferenceFrame('N')
>>> outer(N.x, N.x)
(N.x|N.x)
```

**separate**()

The constituents of this vector in different reference frames, as per its definition.

Returns a dict mapping each ReferenceFrame to the corresponding constituent Vector.

**Examples**

```
>>> from sympy.physics.vector import ReferenceFrame
>>> R1 = ReferenceFrame('R1')
>>> R2 = ReferenceFrame('R2')
>>> v = R1.x + R2.x
>>> v.separate() == {R1: R1.x, R2: R2.x}
True
```

**simplify**()

Returns a simplified Vector.

**subs**(\*args, \*\*kwargs)

Substitution on the Vector.

**Examples**

```
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy import Symbol
>>> N = ReferenceFrame('N')
>>> s = Symbol('s')
>>> a = N.x * s
>>> a.subs({s: 2})
2*N.x
```

**to\_matrix**(*reference\_frame*)

Returns the matrix form of the vector with respect to the given frame.

**Parameters****reference\_frame** : ReferenceFrame

The reference frame that the rows of the matrix correspond to.

**Returns****matrix** : ImmutableMatrix, shape(3,1)

The matrix that gives the 1D vector.

**Examples**

```

>>> from sympy import symbols
>>> from sympy.physics.vector import ReferenceFrame
>>> a, b, c = symbols('a, b, c')
>>> N = ReferenceFrame('N')
>>> vector = a * N.x + b * N.y + c * N.z
>>> vector.to_matrix(N)
Matrix([
[a],
[b],
[c]])
>>> beta = symbols('beta')
>>> A = N.orientnew('A', 'Axis', (beta, N.x))
>>> vector.to_matrix(A)
Matrix([
[a],
[b*cos(beta) + c*sin(beta)],
[-b*sin(beta) + c*cos(beta)]]

```

**xreplace(rule)**

Replace occurrences of objects within the measure numbers of the vector.

**Parameters****rule** : dict-like

Expresses a replacement rule.

**Returns**

Vector

Result of the replacement.

**Examples**

```

>>> from sympy import symbols, pi
>>> from sympy.physics.vector import ReferenceFrame
>>> A = ReferenceFrame('A')
>>> x, y, z = symbols('x y z')
>>> ((1 + x*y) * A.x).xreplace({x: pi})
(pi*y + 1)*A.x
>>> ((1 + x*y) * A.x).xreplace({x: pi, y: 2})
(1 + 2*pi)*A.x

```

Replacements occur only if an entire node in the expression tree is matched:

```
>>> ((x*y + z) * A.x).xreplace({x*y: pi})
(z + pi)*A.x
>>> ((x*y*z) * A.x).xreplace({x*y: pi})
x*y*z*A.x
```

## Dyadic

**class** sympy.physics.vector.dyadic.**Dyadic**(*inlist*)

A Dyadic object.

See: [https://en.wikipedia.org/wiki/Dyadic\\_tensor](https://en.wikipedia.org/wiki/Dyadic_tensor) Kane, T., Levinson, D. Dynamics Theory and Applications. 1985 McGraw-Hill

A more powerful way to represent a rigid body's inertia. While it is more complex, by choosing Dyadic components to be in body fixed basis vectors, the resulting matrix is equivalent to the inertia tensor.

**applyfunc**(*f*)

Apply a function to each component of a Dyadic.

**cross**(*other*)

For a cross product in the form: Dyadic x Vector.

**Parameters**

**other** : Vector

The Vector that we are crossing this Dyadic with

## Examples

```
>>> from sympy.physics.vector import ReferenceFrame, outer, cross
>>> N = ReferenceFrame('N')
>>> d = outer(N.x, N.x)
>>> cross(d, N.y)
(N.x|N.z)
```

**doit**(*\*\*hints*)

Calls .doit() on each term in the Dyadic

**dot**(*other*)

The inner product operator for a Dyadic and a Dyadic or Vector.

**Parameters**

**other** : Dyadic or Vector

The other Dyadic or Vector to take the inner product with



## Examples

```
>>> from sympy.physics.vector import ReferenceFrame, outer
>>> N = ReferenceFrame('N')
>>> D1 = outer(N.x, N.y)
>>> D2 = outer(N.y, N.y)
>>> D1.dot(D2)
(N.x|N.y)
>>> D1.dot(N.y)
N.x
```

### **dt**(*frame*)

Take the time derivative of this Dyadic in a frame.

This function calls the global `time_derivative` method

#### Parameters

**frame** : ReferenceFrame

The frame to take the time derivative in

## Examples

```
>>> from sympy.physics.vector import ReferenceFrame, outer, ↵
↵ dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> N = ReferenceFrame('N')
>>> q = dynamicsymbols('q')
>>> B = N.orientnew('B', 'Axis', [q, N.z])
>>> d = outer(N.x, N.x)
>>> d.dt(B)
- q'*(N.y|N.x) - q'*(N.x|N.y)
```

### **express**(*frame1*, *frame2*=None)

Expresses this Dyadic in alternate frame(s)

The first frame is the list side expression, the second frame is the right side; if Dyadic is in form  $A.x|B.y$ , you can express it in two different frames. If no second frame is given, the Dyadic is expressed in only one frame.

Calls the global `express` function

#### Parameters

**frame1** : ReferenceFrame

The frame to express the left side of the Dyadic in

**frame2** : ReferenceFrame

If provided, the frame to express the right side of the Dyadic in

### Examples

```
>>> from sympy.physics.vector import ReferenceFrame, outer, d
↳ dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> N = ReferenceFrame('N')
>>> q = dynamicsymbols('q')
>>> B = N.orientnew('B', 'Axis', [q, N.z])
>>> d = outer(N.x, N.x)
>>> d.express(B, N)
cos(q)*(B.x|N.x) - sin(q)*(B.y|N.x)
```

#### property func

Returns the class Dyadic.

#### simplify()

Returns a simplified Dyadic.

#### subs(\*args, \*\*kwargs)

Substitution on the Dyadic.

### Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy import Symbol
>>> N = ReferenceFrame('N')
>>> s = Symbol('s')
>>> a = s*(N.x|N.x)
>>> a.subs({s: 2})
2*(N.x|N.x)
```

#### to\_matrix(reference\_frame, second\_reference\_frame=None)

Returns the matrix form of the dyadic with respect to one or two reference frames.

##### Parameters

**reference\_frame** : ReferenceFrame

The reference frame that the rows and columns of the matrix correspond to. If a second reference frame is provided, this only corresponds to the rows of the matrix.

**second\_reference\_frame** : ReferenceFrame, optional, default=None

The reference frame that the columns of the matrix correspond to.

##### Returns

**matrix** : ImmutableMatrix, shape(3,3)

The matrix that gives the 2D tensor form.

## Examples

```
>>> from sympy import symbols
>>> from sympy.physics.vector import ReferenceFrame, Vector
>>> Vector.simp = True
>>> from sympy.physics.mechanics import inertia
>>> Ixx, Iyy, Izz, Ixy, Iyz, Ixz = symbols('Ixx, Iyy, Izz, Ixy, Iyz, Ixz')
>>> N = ReferenceFrame('N')
>>> inertia_dyadic = inertia(N, Ixx, Iyy, Izz, Ixy, Iyz, Ixz)
>>> inertia_dyadic.to_matrix(N)
Matrix([
 [Ixx, Ixy, Ixz],
 [Ixy, Iyy, Iyz],
 [Ixz, Iyz, Izz]])
>>> beta = symbols('beta')
>>> A = N.orientnew('A', 'Axis', (beta, N.x))
>>> inertia_dyadic.to_matrix(A)
Matrix([
 [
 Ixx,
 Ixy*cos(beta) + Ixz*sin(beta),
 -Ixy*sin(beta) + Ixz*cos(beta)],
 [Ixy*cos(beta) + Ixz*sin(beta), Iyy*cos(2*beta)/2 + Iyy/2 +
 Iyz*sin(2*beta) - Izz*cos(2*beta)/2 + Izz/2,
 -Iyy*sin(2*beta)/2 + Iyz*cos(2*beta) + Izz*sin(2*beta)/2],
 [-Ixy*sin(beta) + Ixz*cos(beta),
 -Iyy*sin(2*beta)/2 +
 Iyz*cos(2*beta) + Izz*sin(2*beta)/2, -Iyy*cos(2*beta)/2 + Iyy/2 -
 Iyz*sin(2*beta) + Izz*cos(2*beta)/2 + Izz/2]])
```

### `xreplace(rule)`

Replace occurrences of objects within the measure numbers of the Dyadic.

#### Parameters

**rule** : dict-like

Expresses a replacement rule.

#### Returns

Dyadic

Result of the replacement.

## Examples

```
>>> from sympy import symbols, pi
>>> from sympy.physics.vector import ReferenceFrame, outer
>>> N = ReferenceFrame('N')
>>> D = outer(N.x, N.x)
>>> x, y, z = symbols('x y z')
>>> ((1 + x*y) * D).xreplace({x: pi})
(pi*y + 1)*(N.x|N.x)
>>> ((1 + x*y) * D).xreplace({x: pi, y: 2})
(1 + 2*pi)*(N.x|N.x)
```

Replacements occur only if an entire node in the expression tree is matched:

```
>>> ((x*y + z) * D).xreplace({x*y: pi})
(z + pi)*(N.x|N.x)
>>> ((x*y*z) * D).xreplace({x*y: pi})
x*y*z*(N.x|N.x)
```

## Kinematics (Docstrings)

### Point

**class** `sympy.physics.vector.point.Point(name)`

This object represents a point in a dynamic system.

It stores the: position, velocity, and acceleration of a point. The position is a vector defined as the vector distance from a parent point to this point.

#### Parameters

**name** : string

The display name of the Point

### Examples

```
>>> from sympy.physics.vector import Point, ReferenceFrame, u
↳ dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> P = Point('P')
>>> u1, u2, u3 = dynamicsymbols('u1 u2 u3')
>>> O.set_vel(N, u1 * N.x + u2 * N.y + u3 * N.z)
>>> O.acc(N)
u1'*N.x + u2'*N.y + u3'*N.z
```

`symbols()` can be used to create multiple Points in a single step, for example:

```
>>> from sympy.physics.vector import Point, ReferenceFrame, u
↳ dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> from sympy import symbols
>>> N = ReferenceFrame('N')
>>> u1, u2 = dynamicsymbols('u1 u2')
>>> A, B = symbols('A B', cls=Point)
>>> type(A)
<class 'sympy.physics.vector.point.Point'>
>>> A.set_vel(N, u1 * N.x + u2 * N.y)
>>> B.set_vel(N, u2 * N.x + u1 * N.y)
>>> A.acc(N) - B.acc(N)
(u1' - u2')*N.x + (-u1' + u2')*N.y
```

**a1pt\_theory**(*otherpoint*, *outframe*, *interframe*)

Sets the acceleration of this point with the 1-point theory.

The 1-point theory for point acceleration looks like this:

$${}^N a^P = {}^B a^P + {}^N a^O + {}^N \alpha^B \times r^{OP} + {}^N \omega^B \times ({}^N \omega^B \times r^{OP}) + 2 {}^N \omega^B \times {}^B v^P$$

where O is a point fixed in B, P is a point moving in B, and B is rotating in frame N.

#### Parameters

**otherpoint** : Point

The first point of the 1-point theory (O)

**outframe** : ReferenceFrame

The frame we want this point's acceleration defined in (N)

**fixedframe** : ReferenceFrame

The intermediate frame in this calculation (B)

#### Examples

```
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> from sympy.physics.vector import dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> q = dynamicsymbols('q')
>>> q2 = dynamicsymbols('q2')
>>> qd = dynamicsymbols('q', 1)
>>> q2d = dynamicsymbols('q2', 1)
>>> N = ReferenceFrame('N')
>>> B = ReferenceFrame('B')
>>> B.set_ang_vel(N, 5 * B.y)
>>> O = Point('O')
>>> P = O.locatenew('P', q * B.x)
>>> P.set_vel(B, qd * B.x + q2d * B.y)
>>> O.set_vel(N, 0)
>>> P.a1pt_theory(O, N, B)
(-25*q + q'')*B.x + q2''*B.y - 10*q'*B.z
```

**a2pt\_theory**(*otherpoint*, *outframe*, *fixedframe*)

Sets the acceleration of this point with the 2-point theory.

The 2-point theory for point acceleration looks like this:

$${}^N a^P = {}^N a^O + {}^N \alpha^B \times r^{OP} + {}^N \omega^B \times ({}^N \omega^B \times r^{OP})$$

where O and P are both points fixed in frame B, which is rotating in frame N.

#### Parameters

**otherpoint** : Point

The first point of the 2-point theory (O)

**outframe** : ReferenceFrame

The frame we want this point's acceleration defined in (N)

**fixedframe** : ReferenceFrame

The frame in which both points are fixed (B)

### Examples

```
>>> from sympy.physics.vector import Point, ReferenceFrame, _
_ dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> q = dynamicsymbols('q')
>>> qd = dynamicsymbols('q', 1)
>>> N = ReferenceFrame('N')
>>> B = N.orientnew('B', 'Axis', [q, N.z])
>>> O = Point('O')
>>> P = O.locatenew('P', 10 * B.x)
>>> O.set_vel(N, 5 * N.x)
>>> P.a2pt_theory(O, N, B)
- 10*q''*2*B.x + 10*q'*B.y
```

**acc**(*frame*)

The acceleration Vector of this Point in a ReferenceFrame.

#### Parameters

**frame** : ReferenceFrame

The frame in which the returned acceleration vector will be defined in.

### Examples

```
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> N = ReferenceFrame('N')
>>> p1 = Point('p1')
>>> p1.set_acc(N, 10 * N.x)
>>> p1.acc(N)
10*N.x
```

**locatenew**(*name*, *value*)

Creates a new point with a position defined from this point.

#### Parameters

**name** : str

The name for the new point

**value** : Vector

The position of the new point relative to this point

## Examples

```
>>> from sympy.physics.vector import ReferenceFrame, Point
>>> N = ReferenceFrame('N')
>>> P1 = Point('P1')
>>> P2 = P1.locatenew('P2', 10 * N.x)
```

### **partial\_velocity**(*frame*, \**gen\_speeds*)

Returns the partial velocities of the linear velocity vector of this point in the given frame with respect to one or more provided generalized speeds.

#### Parameters

**frame** : ReferenceFrame

The frame with which the velocity is defined in.

**gen\_speeds** : functions of time

The generalized speeds.

#### Returns

**partial\_velocities** : tuple of Vector

The partial velocity vectors corresponding to the provided generalized speeds.

## Examples

```
>>> from sympy.physics.vector import ReferenceFrame, Point
>>> from sympy.physics.vector import dynamicsymbols
>>> N = ReferenceFrame('N')
>>> A = ReferenceFrame('A')
>>> p = Point('p')
>>> u1, u2 = dynamicsymbols('u1, u2')
>>> p.set_vel(N, u1 * N.x + u2 * A.y)
>>> p.partial_velocity(N, u1)
N.x
>>> p.partial_velocity(N, u1, u2)
(N.x, A.y)
```

### **pos\_from**(*otherpoint*)

Returns a Vector distance between this Point and the other Point.

#### Parameters

**otherpoint** : Point

The otherpoint we are locating this one relative to

### Examples

```
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> N = ReferenceFrame('N')
>>> p1 = Point('p1')
>>> p2 = Point('p2')
>>> p1.set_pos(p2, 10 * N.x)
>>> p1.pos_from(p2)
10*N.x
```

**set\_acc**(*frame, value*)

Used to set the acceleration of this Point in a ReferenceFrame.

#### Parameters

**frame** : ReferenceFrame

The frame in which this point's acceleration is defined

**value** : Vector

The vector value of this point's acceleration in the frame

### Examples

```
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> N = ReferenceFrame('N')
>>> p1 = Point('p1')
>>> p1.set_acc(N, 10 * N.x)
>>> p1.acc(N)
10*N.x
```

**set\_pos**(*otherpoint, value*)

Used to set the position of this point w.r.t. another point.

#### Parameters

**otherpoint** : Point

The other point which this point's location is defined relative to

**value** : Vector

The vector which defines the location of this point

### Examples

```
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> N = ReferenceFrame('N')
>>> p1 = Point('p1')
>>> p2 = Point('p2')
>>> p1.set_pos(p2, 10 * N.x)
>>> p1.pos_from(p2)
10*N.x
```



**set\_vel**(*frame*, *value*)

Sets the velocity Vector of this Point in a ReferenceFrame.

**Parameters**

**frame** : ReferenceFrame

The frame in which this point's velocity is defined

**value** : Vector

The vector value of this point's velocity in the frame

**Examples**

```
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> N = ReferenceFrame('N')
>>> p1 = Point('p1')
>>> p1.set_vel(N, 10 * N.x)
>>> p1.vel(N)
10*N.x
```

**v1pt\_theory**(*otherpoint*, *outframe*, *interframe*)

Sets the velocity of this point with the 1-point theory.

The 1-point theory for point velocity looks like this:

$${}^N \mathbf{v}^P = {}^B \mathbf{v}^P + {}^N \mathbf{v}^O + {}^N \boldsymbol{\omega}^B \times \mathbf{r}^{OP}$$

where O is a point fixed in B, P is a point moving in B, and B is rotating in frame N.

**Parameters**

**otherpoint** : Point

The first point of the 1-point theory (O)

**outframe** : ReferenceFrame

The frame we want this point's velocity defined in (N)

**interframe** : ReferenceFrame

The intermediate frame in this calculation (B)

**Examples**

```
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> from sympy.physics.vector import dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> q = dynamicsymbols('q')
>>> q2 = dynamicsymbols('q2')
>>> qd = dynamicsymbols('q', 1)
>>> q2d = dynamicsymbols('q2', 1)
>>> N = ReferenceFrame('N')
>>> B = ReferenceFrame('B')
>>> B.set_ang_vel(N, 5 * B.y)
```

(continues on next page)

(continued from previous page)

```

>>> O = Point('O')
>>> P = O.locatenew('P', q * B.x)
>>> P.set_vel(B, qd * B.x + q2d * B.y)
>>> O.set_vel(N, 0)
>>> P.v1pt_theory(O, N, B)
q'*B.x + q2'*B.y - 5*q*B.z

```

**v2pt\_theory**(*otherpoint*, *outframe*, *fixedframe*)

Sets the velocity of this point with the 2-point theory.

The 2-point theory for point velocity looks like this:

$${}^N v^P = {}^N v^O + {}^N \omega^B \times r^{OP}$$

where O and P are both points fixed in frame B, which is rotating in frame N.

#### Parameters

**otherpoint** : Point

The first point of the 2-point theory (O)

**outframe** : ReferenceFrame

The frame we want this point's velocity defined in (N)

**fixedframe** : ReferenceFrame

The frame in which both points are fixed (B)

#### Examples

```

>>> from sympy.physics.vector import Point, ReferenceFrame, _
 ↪ dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> q = dynamicsymbols('q')
>>> qd = dynamicsymbols('q', 1)
>>> N = ReferenceFrame('N')
>>> B = N.orientnew('B', 'Axis', [q, N.z])
>>> O = Point('O')
>>> P = O.locatenew('P', 10 * B.x)
>>> O.set_vel(N, 5 * N.x)
>>> P.v2pt_theory(O, N, B)
5*N.x + 10*q'*B.y

```

**vel**(*frame*)

The velocity Vector of this Point in the ReferenceFrame.

#### Parameters

**frame** : ReferenceFrame

The frame in which the returned velocity vector will be defined in

## Examples

```
>>> from sympy.physics.vector import Point, ReferenceFrame,
↳ dynamicsymbols
>>> N = ReferenceFrame('N')
>>> p1 = Point('p1')
>>> p1.set_vel(N, 10 * N.x)
>>> p1.vel(N)
10*N.x
```

Velocities will be automatically calculated if possible, otherwise a `ValueError` will be returned. If it is possible to calculate multiple different velocities from the relative points, the points defined most directly relative to this point will be used. In the case of inconsistent relative positions of points, incorrect velocities may be returned. It is up to the user to define prior relative positions and velocities of points in a self-consistent way.

```
>>> p = Point('p')
>>> q = dynamicsymbols('q')
>>> p.set_vel(N, 10 * N.x)
>>> p2 = Point('p2')
>>> p2.set_pos(p, q*N.x)
>>> p2.vel(N)
(Derivative(q(t), t) + 10)*N.x
```

## kinematic\_equations

`sympy.physics.vector.functions.get_motion_params(frame, **kwargs)`

Returns the three motion parameters - (acceleration, velocity, and position) as vectorial functions of time in the given frame.

If a higher order differential function is provided, the lower order functions are used as boundary conditions. For example, given the acceleration, the velocity and position parameters are taken as boundary conditions.

The values of time at which the boundary conditions are specified are taken from `timevalue1` (for position boundary condition) and `timevalue2` (for velocity boundary condition).

If any of the boundary conditions are not provided, they are taken to be zero by default (zero vectors, in case of vectorial inputs). If the boundary conditions are also functions of time, they are converted to constants by substituting the time values in the dynamic-symbols.\_t time Symbol.

This function can also be used for calculating rotational motion parameters. Have a look at the Parameters and Examples for more clarity.

### Parameters

**frame** : ReferenceFrame

The frame to express the motion parameters in

**acceleration** : Vector

Acceleration of the object/frame as a function of time

**velocity** : Vector

Velocity as function of time or as boundary condition of velocity at  
time = timevalue1

**position** : Vector

Velocity as function of time or as boundary condition of velocity at  
time = timevalue1

**timevalue1** : sympyfiabe

Value of time for position boundary condition

**timevalue2** : sympyfiabe

Value of time for velocity boundary condition

## Examples

```
>>> from sympy.physics.vector import ReferenceFrame, get_motion_params, _
_ dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> from sympy import symbols
>>> R = ReferenceFrame('R')
>>> v1, v2, v3 = dynamicsymbols('v1 v2 v3')
>>> v = v1*R.x + v2*R.y + v3*R.z
>>> get_motion_params(R, position = v)
(v1'*R.x + v2'*R.y + v3'*R.z, v1'*R.x + v2'*R.y + v3'*R.z, v1*R.x + _
_ v2*R.y + v3*R.z)
>>> a, b, c = symbols('a b c')
>>> v = a*R.x + b*R.y + c*R.z
>>> get_motion_params(R, velocity = v)
(0, a*R.x + b*R.y + c*R.z, a*t*R.x + b*t*R.y + c*t*R.z)
>>> parameters = get_motion_params(R, acceleration = v)
>>> parameters[1]
a*t*R.x + b*t*R.y + c*t*R.z
>>> parameters[2]
a*t**2/2*R.x + b*t**2/2*R.y + c*t**2/2*R.z
```

`sympy.physics.vector.functions.kinematic_equations(speeds, coords, rot_type, rot_order="")`

Gives equations relating the qdot's to u's for a rotation type.

Supply rotation type and order as in orient. Speeds are assumed to be body-fixed; if we are defining the orientation of B in A using by rot\_type, the angular velocity of B in A is assumed to be in the form:  $\text{speed}[0]*B.x + \text{speed}[1]*B.y + \text{speed}[2]*B.z$

### Parameters

**speeds** : list of length 3

The body fixed angular velocity measure numbers.

**coords** : list of length 3 or 4

The coordinates used to define the orientation of the two frames.

**rot\_type** : str

The type of rotation used to create the equations. Body, Space, or Quaternion only

**rot\_order** : str or int

If applicable, the order of a series of rotations.

## Examples

```
>>> from sympy.physics.vector import dynamicsymbols
>>> from sympy.physics.vector import kinematic_equations, vprint
>>> u1, u2, u3 = dynamicsymbols('u1 u2 u3')
>>> q1, q2, q3 = dynamicsymbols('q1 q2 q3')
>>> vprint(kinematic_equations([u1,u2,u3], [q1,q2,q3], 'body', '313'),
... order=None)
[-(u1*sin(q3) + u2*cos(q3))/sin(q2) + q1', -u1*cos(q3) + u2*sin(q3) + q2
→', (u1*sin(q3) + u2*cos(q3))*cos(q2)/sin(q2) - u3 + q3']
```

`sympy.physics.vector.functions.partial_velocity(vel_vecs, gen_speeds, frame)`

Returns a list of partial velocities with respect to the provided generalized speeds in the given reference frame for each of the supplied velocity vectors.

The output is a list of lists. The outer list has a number of elements equal to the number of supplied velocity vectors. The inner lists are, for each velocity vector, the partial derivatives of that velocity vector with respect to the generalized speeds supplied.

### Parameters

**vel\_vecs** : iterable

An iterable of velocity vectors (angular or linear).

**gen\_speeds** : iterable

An iterable of generalized speeds.

**frame** : ReferenceFrame

The reference frame that the partial derivatives are going to be taken in.

## Examples

```
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> from sympy.physics.vector import dynamicsymbols
>>> from sympy.physics.vector import partial_velocity
>>> u = dynamicsymbols('u')
>>> N = ReferenceFrame('N')
>>> P = Point('P')
>>> P.set_vel(N, u * N.x)
>>> vel_vecs = [P.vel(N)]
>>> gen_speeds = [u]
>>> partial_velocity(vel_vecs, gen_speeds, N)
[[N.x]]
```

## Printing (Docstrings)

### init\_vprinting

`sympy.physics.vector.printing.init_vprinting(**kwargs)`

Initializes time derivative printing for all SymPy objects, i.e. any functions of time will be displayed in a more compact notation. The main benefit of this is for printing of time derivatives; instead of displaying as `Derivative(f(t), t)`, it will display `f'`. This is only actually needed for when derivatives are present and are not in a `physics.vector.Vector` or `physics.vector.Dyadic` object. This function is a light wrapper to `init_printing()` (page 2119). Any keyword arguments for it are valid here.

Initializes pretty-printer depending on the environment.

#### Parameters

**pretty\_print** : bool, default=True

If True, use `pretty_print()` (page 2139) to stringify or the provided pretty printer; if False, use `sstrrepr()` (page 2178) to stringify or the provided string printer.

**order** : string or None, default='lex'

There are a few different settings for this parameter: 'lex' (default), which is lexicographic order; 'grlex', which is graded lexicographic order; 'grevlex', which is reversed graded lexicographic order; 'old', which is used for compatibility reasons and for long expressions; None, which sets it to lex.

**use\_unicode** : bool or None, default=None

If True, use unicode characters; if False, do not use unicode characters; if None, make a guess based on the environment.

**use\_latex** : string, bool, or None, default=None

If True, use default LaTeX rendering in GUI interfaces (png and mathjax); if False, do not use LaTeX rendering; if None, make a guess based on the environment; if 'png', enable LaTeX rendering with an external LaTeX compiler, falling back to matplotlib if external compilation fails; if 'matplotlib', enable LaTeX rendering with matplotlib; if 'mathjax', enable LaTeX text generation, for example MathJax rendering in IPython notebook or text rendering in LaTeX documents; if 'svg', enable LaTeX rendering with an external latex compiler, no fallback

**wrap\_line** : bool

If True, lines will wrap at the end; if False, they will not wrap but continue as one line. This is only relevant if `pretty_print` is True.

**num\_columns** : int or None, default=None

If int, number of columns before wrapping is set to `num_columns`; if None, number of columns before wrapping is set to terminal width. This is only relevant if `pretty_print` is True.

**no\_global** : bool, default=False

If True, the settings become system wide; if False, use just for this console/session.

**ip** : An interactive console

This can either be an instance of IPython, or a class that derives from `code.InteractiveConsole`.

**euler** : bool, optional, default=False

Loads the euler package in the LaTeX preamble for handwritten style fonts (<http://www.ctan.org/pkg/euler>).

**forecolor** : string or None, optional, default=None

DVI setting for foreground color. None means that either 'Black', 'White', or 'Gray' will be selected based on a guess of the IPython terminal color setting. See notes.

**backcolor** : string, optional, default='Transparent'

DVI setting for background color. See notes.

**fontsize** : string or int, optional, default='10pt'

A font size to pass to the LaTeX documentclass function in the preamble. Note that the options are limited by the documentclass. Consider using scale instead.

**latex\_mode** : string, optional, default='plain'

The mode used in the LaTeX printer. Can be one of: {'inline'|'plain'|'equation'|'equation\*'}.

**print\_builtin** : boolean, optional, default=True

If True then floats and integers will be printed. If False the printer will only print SymPy types.

**str\_printer** : function, optional, default=None

A custom string printer function. This should mimic `sstrrepr()` (page 2178).

**pretty\_printer** : function, optional, default=None

A custom pretty printer. This should mimic `pretty()` (page 2139).

**latex\_printer** : function, optional, default=None

A custom LaTeX printer. This should mimic `latex()` (page 2170).

**scale** : float, optional, default=1.0

Scale the LaTeX output when using the 'png' or 'svg' backends. Useful for high dpi screens.

**settings** :

Any additional settings for the latex and pretty commands can be used to fine-tune the output.

## Examples

```
>>> from sympy import Function, symbols
>>> t, x = symbols('t, x')
>>> omega = Function('omega')
>>> omega(x).diff()
Derivative(omega(x), x)
>>> omega(t).diff()
Derivative(omega(t), t)
```

Now use the string printer:

```
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> omega(x).diff()
Derivative(omega(x), x)
>>> omega(t).diff()
omega'
```

## vprint

`sympy.physics.vector.printing.vprint(expr, **settings)`

Function for printing of expressions generated in the `sympy.physics` vector package.

Extends SymPy's `StrPrinter`, takes the same setting accepted by SymPy's `sstr()` (page 2178), and is equivalent to `print(sstr(foo))`.

### Parameters

**expr** : valid SymPy object

SymPy expression to print.

**settings** : args

Same as the settings accepted by SymPy's `sstr()`.

## Examples

```
>>> from sympy.physics.vector import vprint, dynamicsymbols
>>> u1 = dynamicsymbols('u1')
>>> print(u1)
u1(t)
>>> vprint(u1)
u1
```



## vpprint

`sympy.physics.vector.printing.vpprint(expr, **settings)`

Function for pretty printing of expressions generated in the `sympy.physics` vector package.

Mainly used for expressions not inside a vector; the output of running scripts and generating equations of motion. Takes the same options as SymPy's `pretty_print()` (page 2139); see that function for more information.

### Parameters

**expr** : valid SymPy object

SymPy expression to pretty print

**settings** : args

Same as those accepted by SymPy's `pretty_print`.

## vlatex

`sympy.physics.vector.printing.vlatex(expr, **settings)`

Function for printing latex representation of `sympy.physics.vector` objects.

For latex representation of Vectors, Dyadics, and dynamicsymbols. Takes the same options as SymPy's `latex()` (page 2170); see that function for more information;

### Parameters

**expr** : valid SymPy object

SymPy expression to represent in LaTeX form

**settings** : args

Same as `latex()`

## Examples

```

>>> from sympy.physics.vector import vlatex, ReferenceFrame, _
 dynamicsymbols
>>> N = ReferenceFrame('N')
>>> q1, q2 = dynamicsymbols('q1 q2')
>>> q1d, q2d = dynamicsymbols('q1 q2', 1)
>>> q1dd, q2dd = dynamicsymbols('q1 q2', 2)
>>> vlatex(N.x + N.y)
'\mathbf{\hat{n}_x} + \mathbf{\hat{n}_y}'
>>> vlatex(q1 + q2)
'q_{1} + q_{2}'
>>> vlatex(q1d)
'\dot{q}_{1}'
>>> vlatex(q1 * q2d)
'q_{1} \dot{q}_{2}'
>>> vlatex(q1dd * q1 / q1d)
'\frac{q_{1} \ddot{q}_{1}}{\dot{q}_{1}}'

```

## Essential Functions (Docstrings)

### dynamicsymbols

`sympy.physics.vector.dynamicsymbols(names, level=0, **assumptions)`

Uses symbols and Function for functions of time.

Creates a SymPy UndefinedFunction, which is then initialized as a function of a variable, the default being `Symbol('t')`.

#### Parameters

**names** : str

Names of the dynamic symbols you want to create; works the same way as inputs to symbols

**level** : int

Level of differentiation of the returned function; d/dt once of t, twice of t, etc.

**assumptions** :

- **real(bool)**  
[This is used to set the dynamicsymbol as real,] by default is False.
- **positive(bool)**  
[This is used to set the dynamicsymbol as positive,] by default is False.
- **commutative(bool)**  
[This is used to set the commutative property of] a dynamicsymbol, by default is True.
- **integer(bool)**  
[This is used to set the dynamicsymbol as integer,] by default is False.

### Examples

```
>>> from sympy.physics.vector import dynamicsymbols
>>> from sympy import diff, Symbol
>>> q1 = dynamicsymbols('q1')
>>> q1
q1(t)
>>> q2 = dynamicsymbols('q2', real=True)
>>> q2.is_real
True
>>> q3 = dynamicsymbols('q3', positive=True)
>>> q3.is_positive
True
>>> q4, q5 = dynamicsymbols('q4,q5', commutative=False)
>>> bool(q4*q5 != q5*q4)
True
>>> q6 = dynamicsymbols('q6', integer=True)
>>> q6.is_integer
True
```

(continues on next page)

(continued from previous page)

```
>>> diff(q1, Symbol('t'))
Derivative(q1(t), t)
```

**dot**

`sympy.physics.vector.functions.dot(vec1, vec2)`

Dot product convenience wrapper for `Vector.dot()`: Dot product of two vectors.

Returns a scalar, the dot product of the two Vectors

**Parameters**

**other** : Vector

The Vector which we are dotting with

**Examples**

```
>>> from sympy.physics.vector import ReferenceFrame, dot
>>> from sympy import symbols
>>> q1 = symbols('q1')
>>> N = ReferenceFrame('N')
>>> dot(N.x, N.x)
1
>>> dot(N.x, N.y)
0
>>> A = N.orientnew('A', 'Axis', [q1, N.x])
>>> dot(N.y, A.y)
cos(q1)
```

**cross**

`sympy.physics.vector.functions.cross(vec1, vec2)`

Cross product convenience wrapper for `Vector.cross()`: The cross product operator for two Vectors.

Returns a Vector, expressed in the same ReferenceFrames as self.

**Parameters**

**other** : Vector

The Vector which we are crossing with

## Examples

```
>>> from sympy import symbols
>>> from sympy.physics.vector import ReferenceFrame, cross
>>> q1 = symbols('q1')
>>> N = ReferenceFrame('N')
>>> cross(N.x, N.y)
N.z
>>> A = ReferenceFrame('A')
>>> A.orient_axis(N, q1, N.x)
>>> cross(A.x, N.y)
N.z
>>> cross(N.y, A.x)
- sin(q1)*A.y - cos(q1)*A.z
```

## outer

`sympy.physics.vector.functions.outer(vec1, vec2)`

Outer product convenience wrapper for `Vector.outer()`: Outer product between two Vectors.

A rank increasing operation, which returns a Dyadic from two Vectors

### Parameters

**other** : Vector

The Vector to take the outer product with

## Examples

```
>>> from sympy.physics.vector import ReferenceFrame, outer
>>> N = ReferenceFrame('N')
>>> outer(N.x, N.x)
(N.x|N.x)
```

## express

`sympy.physics.vector.functions.express(expr, frame, frame2=None, variables=False)`

Global function for ‘express’ functionality.

Re-expresses a Vector, scalar(sympyifiable) or Dyadic in given frame.

Refer to the local methods of Vector and Dyadic for details. If ‘variables’ is True, then the coordinate variables (CoordinateSym instances) of other frames present in the vector/scalar field or dyadic expression are also substituted in terms of the base scalars of this frame.

### Parameters

**expr** : Vector/Dyadic/scalar(sympyifiable)

The expression to re-express in ReferenceFrame ‘frame’

**frame: ReferenceFrame**

The reference frame to express `expr` in

**frame2** : ReferenceFrame

The other frame required for re-expression(only for Dyadic `expr`)

**variables** : boolean

Specifies whether to substitute the coordinate variables present in `expr`, in terms of those of `frame`

**Examples**

```
>>> from sympy.physics.vector import ReferenceFrame, outer, _
_ dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> N = ReferenceFrame('N')
>>> q = dynamicsymbols('q')
>>> B = N.orientnew('B', 'Axis', [q, N.z])
>>> d = outer(N.x, N.x)
>>> from sympy.physics.vector import express
>>> express(d, B, N)
cos(q)*(B.x|N.x) - sin(q)*(B.y|N.x)
>>> express(B.x, N)
cos(q)*N.x + sin(q)*N.y
>>> express(N[0], B, variables=True)
B_x*cos(q) - B_y*sin(q)
```

**time\_derivative**

`sympy.physics.vector.functions.time_derivative(expr, frame, order=1)`

Calculate the time derivative of a vector/scalar field function or dyadic expression in given frame.

**Parameters**

**expr** : Vector/Dyadic/sympifyable

The expression whose time derivative is to be calculated

**frame** : ReferenceFrame

The reference frame to calculate the time derivative in

**order** : integer

The order of the derivative to be calculated

## Examples

```
>>> from sympy.physics.vector import ReferenceFrame, dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> from sympy import Symbol
>>> q1 = Symbol('q1')
>>> u1 = dynamicsymbols('u1')
>>> N = ReferenceFrame('N')
>>> A = N.orientnew('A', 'Axis', [q1, N.x])
>>> v = u1 * N.x
>>> A.set_ang_vel(N, 10*A.x)
>>> from sympy.physics.vector import time_derivative
>>> time_derivative(v, N)
u1'*N.x
>>> time_derivative(u1*A[0], N)
N_x*u1'
>>> B = N.orientnew('B', 'Axis', [u1, N.z])
>>> from sympy.physics.vector import outer
>>> d = outer(N.x, N.x)
>>> time_derivative(d, B)
- u1'*(N.y|N.x) - u1'*(N.x|N.y)
```

## References

[https://en.wikipedia.org/wiki/Rotating\\_reference\\_frame#Time\\_derivatives\\_in\\_the\\_two\\_frames](https://en.wikipedia.org/wiki/Rotating_reference_frame#Time_derivatives_in_the_two_frames)

## Docstrings for basic field functions

### Field operation functions

These functions implement some basic operations pertaining to fields in general.

#### curl

`sympy.physics.vector.fieldfunctions.curl(vect, frame)`

Returns the curl of a vector field computed wrt the coordinate symbols of the given frame.

#### Parameters

**vect** : Vector

The vector operand

**frame** : ReferenceFrame

The reference frame to calculate the curl in

## Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy.physics.vector import curl
>>> R = ReferenceFrame('R')
>>> v1 = R[1]*R[2]*R.x + R[0]*R[2]*R.y + R[0]*R[1]*R.z
>>> curl(v1, R)
0
>>> v2 = R[0]*R[1]*R[2]*R.x
>>> curl(v2, R)
R_x*R_y*R.y - R_x*R_z*R.z
```

## divergence

`sympy.physics.vector.fieldfunctions.divergence(vect, frame)`

Returns the divergence of a vector field computed wrt the coordinate symbols of the given frame.

### Parameters

**vect** : Vector

The vector operand

**frame** : ReferenceFrame

The reference frame to calculate the divergence in

## Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy.physics.vector import divergence
>>> R = ReferenceFrame('R')
>>> v1 = R[0]*R[1]*R[2] * (R.x+R.y+R.z)
>>> divergence(v1, R)
R_x*R_y + R_x*R_z + R_y*R_z
>>> v2 = 2*R[1]*R[2]*R.y
>>> divergence(v2, R)
2*R_z
```

## gradient

`sympy.physics.vector.fieldfunctions.gradient(scalar, frame)`

Returns the vector gradient of a scalar field computed wrt the coordinate symbols of the given frame.

### Parameters

**scalar** : sympifiable

The scalar field to take the gradient of

**frame** : ReferenceFrame

The frame to calculate the gradient in

### Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy.physics.vector import gradient
>>> R = ReferenceFrame('R')
>>> s1 = R[0]*R[1]*R[2]
>>> gradient(s1, R)
R_y*R_z*R.x + R_x*R_z*R.y + R_x*R_y*R.z
>>> s2 = 5*R[0]**2*R[2]
>>> gradient(s2, R)
10*R_x*R_z*R.x + 5*R_x**2*R.z
```

### scalar\_potential

`sympy.physics.vector.fieldfunctions.scalar_potential(field, frame)`

Returns the scalar potential function of a field in a given frame (without the added integration constant).

#### Parameters

**field** : Vector

The vector field whose scalar potential function is to be calculated

**frame** : ReferenceFrame

The frame to do the calculation in

### Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy.physics.vector import scalar_potential, gradient
>>> R = ReferenceFrame('R')
>>> scalar_potential(R.z, R) == R[2]
True
>>> scalar_field = 2*R[0]**2*R[1]*R[2]
>>> grad_field = gradient(scalar_field, R)
>>> scalar_potential(grad_field, R)
2*R_x**2*R_y*R_z
```



## scalar\_potential\_difference

`sympy.physics.vector.fieldfunctions.scalar_potential_difference(field, frame, point1, point2, origin)`

Returns the scalar potential difference between two points in a certain frame, wrt a given field.

If a scalar field is provided, its values at the two points are considered. If a conservative vector field is provided, the values of its scalar potential function at the two points are used.

Returns (potential at position 2) - (potential at position 1)

### Parameters

**field** : Vector/sympyifiable

The field to calculate wrt

**frame** : ReferenceFrame

The frame to do the calculations in

**point1** : Point

The initial Point in given frame

**position2** : Point

The second Point in the given frame

**origin** : Point

The Point to use as reference point for position vector calculation

### Examples

```
>>> from sympy.physics.vector import ReferenceFrame, Point
>>> from sympy.physics.vector import scalar_potential_difference
>>> R = ReferenceFrame('R')
>>> O = Point('O')
>>> P = O.locatenew('P', R[0]*R.x + R[1]*R.y + R[2]*R.z)
>>> vectfield = 4*R[0]*R[1]*R.x + 2*R[0]**2*R.y
>>> scalar_potential_difference(vectfield, R, O, P, O)
2*R_x**2*R_y
>>> Q = O.locatenew('Q', 3*R.x + R.y + 2*R.z)
>>> scalar_potential_difference(vectfield, R, P, Q, O)
-2*R_x**2*R_y + 18
```

## Checking the type of vector field

### is\_conservative

`sympy.physics.vector.fieldfunctions.is_conservative(field)`

Checks if a field is conservative.

#### Parameters

**field** : Vector

The field to check for conservative property

### Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy.physics.vector import is_conservative
>>> R = ReferenceFrame('R')
>>> is_conservative(R[1]*R[2]*R.x + R[0]*R[2]*R.y + R[0]*R[1]*R.z)
True
>>> is_conservative(R[2] * R.y)
False
```

### is\_solenoidal

`sympy.physics.vector.fieldfunctions.is_solenoidal(field)`

Checks if a field is solenoidal.

#### Parameters

**field** : Vector

The field to check for solenoidal property

### Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy.physics.vector import is_solenoidal
>>> R = ReferenceFrame('R')
>>> is_solenoidal(R[1]*R[2]*R.x + R[0]*R[2]*R.y + R[0]*R[1]*R.z)
True
>>> is_solenoidal(R[1] * R.y)
False
```

## Classical Mechanics

### Abstract

In this documentation many components of the physics/mechanics module will be discussed. `sympy.physics.mechanics` (page 1675) has been written to allow for creation of symbolic equations of motion for complicated multibody systems.

### Vector

This module derives the vector-related abilities and related functionalities from `sympy.physics.vector` (page 1592). Please have a look at the documentation of `sympy.physics.vector` (page 1592) and its necessary API to understand the vector capabilities of `sympy.physics.mechanics` (page 1675).

### Mechanics

In physics, mechanics describes conditions of rest (statics) or motion (dynamics). There are a few common steps to all mechanics problems. First, an idealized representation of a system is described. Next, we use physical laws to generate equations that define the system's behavior. Then, we solve these equations, sometimes analytically but usually numerically. Finally, we extract information from these equations and solutions. The current scope of the module is multi-body dynamics: the motion of systems of multiple particles and/or rigid bodies. For example, this module could be used to understand the motion of a double pendulum, planets, robotic manipulators, bicycles, and any other system of rigid bodies that may fascinate us.

Often, the objective in multi-body dynamics is to obtain the trajectory of a system of rigid bodies through time. The challenge for this task is to first formulate the equations of motion of the system. Once they are formulated, they must be solved, that is, integrated forward in time. When digital computers came around, solving became the easy part of the problem. Now, we can tackle more complicated problems, which leaves the challenge of formulating the equations.

The term “equations of motion” is used to describe the application of Newton's second law to multi-body systems. The form of the equations of motion depends on the method used to generate them. This package implements two of these methods: Kane's method and Lagrange's method. This module facilitates the formulation of equations of motion, which can then be solved (integrated) using generic ordinary differential equation (ODE) solvers.

The approach to a particular class of dynamics problems, that of forward dynamics, has the following steps:

1. describing the system's geometry and configuration,
2. specifying the way the system can move, including constraints on its motion
3. describing the external forces and moments on the system,
4. combining the above information according to Newton's second law ( $\mathbf{F} = m\mathbf{a}$ ), and
5. organizing the resulting equations so that they can be integrated to obtain the system's trajectory through time.

Together with the rest of SymPy, this module performs steps 4 and 5, provided that the user can perform 1 through 3 for the module. That is to say, the user must provide a complete representation of the free body diagrams that themselves represent the system, with which this code can provide equations of motion in a form amenable to numerical integration. Step 5 above amounts to arduous algebra for even fairly simple multi-body systems. Thus, it is desirable to use a symbolic math package, such as SymPy, to perform this step. It is for this reason that this module is a part of SymPy. Step 4 amounts to this specific module, `sympy.physics.mechanics`.

## Guide to Mechanics

### Masses, Inertias, Particles and Rigid Bodies in Physics/Mechanics

This document will describe how to represent masses and inertias in `sympy.physics.mechanics` (page 1675) and use of the `RigidBody` and `Particle` classes.

It is assumed that the reader is familiar with the basics of these topics, such as finding the center of mass for a system of particles, how to manipulate an inertia tensor, and the definition of a particle and rigid body. Any advanced dynamics text can provide a reference for these details.

## Mass

The only requirement for a mass is that it needs to be a sympify-able expression. Keep in mind that masses can be time varying.

## Particle

Particles are created with the class `Particle` in `sympy.physics.mechanics` (page 1675). A `Particle` object has an associated point and an associated mass which are the only two attributes of the object.:

```
>>> from sympy.physics.mechanics import Particle, Point
>>> from sympy import Symbol
>>> m = Symbol('m')
>>> po = Point('po')
>>> # create a particle container
>>> pa = Particle('pa', po, m)
```

The associated point contains the position, velocity and acceleration of the particle. `sympy.physics.mechanics` (page 1675) allows one to perform kinematic analysis of points separate from their association with masses.

## Inertia

See the Inertia (Dyadics) section in ‘Advanced Topics’ part of [sympy.physics.vector](#) (page 1592) docs.

## Rigid Body

Rigid bodies are created in a similar fashion as particles. The `RigidBody` class generates objects with four attributes: mass, center of mass, a reference frame, and an inertia tuple:

```
>>> from sympy import Symbol
>>> from sympy.physics.mechanics import ReferenceFrame, Point, RigidBody
>>> from sympy.physics.mechanics import outer
>>> m = Symbol('m')
>>> A = ReferenceFrame('A')
>>> P = Point('P')
>>> I = outer(A.x, A.x)
>>> # create a rigid body
>>> B = RigidBody('B', P, A, m, (I, P))
```

The mass is specified exactly as is in a particle. Similar to the `Particle`’s `.point`, the `RigidBody`’s center of mass, `.masscenter` must be specified. The reference frame is stored in an analogous fashion and holds information about the body’s orientation and angular velocity. Finally, the inertia for a rigid body needs to be specified about a point. In [sympy.physics.mechanics](#) (page 1675), you are allowed to specify any point for this. The most common is the center of mass, as shown in the above code. If a point is selected which is not the center of mass, ensure that the position between the point and the center of mass has been defined. The inertia is specified as a tuple of length two with the first entry being a Dyadic and the second entry being a `Point` of which the inertia dyadic is defined about.

## Dyadic

In [sympy.physics.mechanics](#) (page 1675), dyadics are used to represent inertia ([[Kane1985](#)], [[WikiDyadics](#)], [[WikiDyadicProducts](#)]). A dyadic is a linear polynomial of component unit dyadics, similar to a vector being a linear polynomial of component unit vectors. A dyadic is the outer product between two vectors which returns a new quantity representing the juxtaposition of these two vectors. For example:

$$\begin{aligned}\hat{\mathbf{a}}_{\mathbf{x}} \otimes \hat{\mathbf{a}}_{\mathbf{x}} &= \hat{\mathbf{a}}_{\mathbf{x}} \hat{\mathbf{a}}_{\mathbf{x}} \\ \hat{\mathbf{a}}_{\mathbf{x}} \otimes \hat{\mathbf{a}}_{\mathbf{y}} &= \hat{\mathbf{a}}_{\mathbf{x}} \hat{\mathbf{a}}_{\mathbf{y}}\end{aligned}$$

Where  $\hat{\mathbf{a}}_{\mathbf{x}} \hat{\mathbf{a}}_{\mathbf{x}}$  and  $\hat{\mathbf{a}}_{\mathbf{x}} \hat{\mathbf{a}}_{\mathbf{y}}$  are the outer products obtained by multiplying the left side as a column vector by the right side as a row vector. Note that the order is significant.

Some additional properties of a dyadic are:

$$\begin{aligned}(x\mathbf{v}) \otimes \mathbf{w} &= \mathbf{v} \otimes (x\mathbf{w}) = x(\mathbf{v} \otimes \mathbf{w}) \\ \mathbf{v} \otimes (\mathbf{w} + \mathbf{u}) &= \mathbf{v} \otimes \mathbf{w} + \mathbf{v} \otimes \mathbf{u} \\ (\mathbf{v} + \mathbf{w}) \otimes \mathbf{u} &= \mathbf{v} \otimes \mathbf{u} + \mathbf{w} \otimes \mathbf{u}\end{aligned}$$

A vector in a reference frame can be represented as  $\begin{bmatrix} a \\ b \\ c \end{bmatrix}$  or  $a\hat{\mathbf{i}} + b\hat{\mathbf{j}} + c\hat{\mathbf{k}}$ . Similarly, a dyadic can be represented in tensor form:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

or in dyadic form:

$$a_{11}\hat{\mathbf{a}}_x\hat{\mathbf{a}}_x + a_{12}\hat{\mathbf{a}}_x\hat{\mathbf{a}}_y + a_{13}\hat{\mathbf{a}}_x\hat{\mathbf{a}}_z + a_{21}\hat{\mathbf{a}}_y\hat{\mathbf{a}}_x + a_{22}\hat{\mathbf{a}}_y\hat{\mathbf{a}}_y + a_{23}\hat{\mathbf{a}}_y\hat{\mathbf{a}}_z + a_{31}\hat{\mathbf{a}}_z\hat{\mathbf{a}}_x + a_{32}\hat{\mathbf{a}}_z\hat{\mathbf{a}}_y + a_{33}\hat{\mathbf{a}}_z\hat{\mathbf{a}}_z$$

Just as with vectors, the later representation makes it possible to keep track of which frames the dyadic is defined with respect to. Also, the two components of each term in the dyadic need not be in the same frame. The following is valid:

$$\hat{\mathbf{a}}_x \otimes \hat{\mathbf{b}}_y = \hat{\mathbf{a}}_x \hat{\mathbf{b}}_y$$

Dyadics can also be crossed and dotted with vectors; again, order matters:

$$\begin{aligned} \hat{\mathbf{a}}_x \hat{\mathbf{a}}_x \cdot \hat{\mathbf{a}}_x &= \hat{\mathbf{a}}_x \\ \hat{\mathbf{a}}_y \hat{\mathbf{a}}_x \cdot \hat{\mathbf{a}}_x &= \hat{\mathbf{a}}_y \\ \hat{\mathbf{a}}_x \hat{\mathbf{a}}_y \cdot \hat{\mathbf{a}}_x &= 0 \\ \hat{\mathbf{a}}_x \cdot \hat{\mathbf{a}}_x \hat{\mathbf{a}}_x &= \hat{\mathbf{a}}_x \\ \hat{\mathbf{a}}_x \cdot \hat{\mathbf{a}}_x \hat{\mathbf{a}}_y &= \hat{\mathbf{a}}_y \\ \hat{\mathbf{a}}_x \cdot \hat{\mathbf{a}}_y \hat{\mathbf{a}}_x &= 0 \\ \hat{\mathbf{a}}_x \times \hat{\mathbf{a}}_y \hat{\mathbf{a}}_x &= \hat{\mathbf{a}}_z \hat{\mathbf{a}}_x \\ \hat{\mathbf{a}}_x \times \hat{\mathbf{a}}_x \hat{\mathbf{a}}_x &= 0 \\ \hat{\mathbf{a}}_y \hat{\mathbf{a}}_x \times \hat{\mathbf{a}}_z &= -\hat{\mathbf{a}}_y \hat{\mathbf{a}}_y \end{aligned}$$

One can also take the time derivative of dyadics or express them in different frames, just like with vectors.

## Linear Momentum

The linear momentum of a particle P is defined as:

$$L_P = m\mathbf{v}$$

where  $m$  is the mass of the particle P and  $\mathbf{v}$  is the velocity of the particle in the inertial frame.[Likins1973].

Similarly the linear momentum of a rigid body is defined as:

$$L_B = m\mathbf{v}^*$$

where  $m$  is the mass of the rigid body, B, and  $\mathbf{v}^*$  is the velocity of the mass center of B in the inertial frame.

## Angular Momentum

The angular momentum of a particle P about an arbitrary point O in an inertial frame N is defined as:

$${}^N\mathbf{H}^{P/O} = \mathbf{r} \times m\mathbf{v}$$

where  $\mathbf{r}$  is a position vector from point O to the particle of mass  $m$  and  $\mathbf{v}$  is the velocity of the particle in the inertial frame.

Similarly the angular momentum of a rigid body B about a point O in an inertial frame N is defined as:

$${}^N\mathbf{H}^{B/O} = {}^N\mathbf{H}^{B/B^*} + {}^N\mathbf{H}^{B^*/O}$$

where the angular momentum of the body about it's mass center is:

$${}^N\mathbf{H}^{B/B^*} = \mathbf{I}^* \cdot \boldsymbol{\omega}$$

and the angular momentum of the mass center about O is:

$${}^N\mathbf{H}^{B^*/O} = \mathbf{r}^* \times m\mathbf{v}^*$$

where  $\mathbf{I}^*$  is the central inertia dyadic of rigid body B,  $\boldsymbol{\omega}$  is the inertial angular velocity of B,  $\mathbf{r}^*$  is a position vector from point O to the mass center of B,  $m$  is the mass of B and  $\mathbf{v}^*$  is the velocity of the mass center in the inertial frame.

## Using momenta functions in Mechanics

The following example shows how to use the momenta functions in [\*sympy.physics.mechanics\*](#) (page 1675).

One begins by creating the requisite symbols to describe the system. Then the reference frame is created and the kinematics are done.

```
>>> from sympy import symbols
>>> from sympy.physics.mechanics import dynamicsymbols, ReferenceFrame
>>> from sympy.physics.mechanics import RigidBody, Particle, Point, outer
>>> from sympy.physics.mechanics import linear_momentum, angular_momentum
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> m, M, l1 = symbols('m M l1')
>>> q1d = dynamicsymbols('q1d')
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> O.set_vel(N, 0 * N.x)
>>> Ac = O.locatenew('Ac', l1 * N.x)
>>> P = Ac.locatenew('P', l1 * N.x)
>>> a = ReferenceFrame('a')
>>> a.set_ang_vel(N, q1d * N.z)
>>> Ac.v2pt_theory(O, N, a)
l1*q1d*N.y
>>> P.v2pt_theory(O, N, a)
2*l1*q1d*N.y
```

Finally, the bodies that make up the system are created. In this case the system consists of a particle Pa and a RigidBody A.

```
>>> Pa = Particle('Pa', P, m)
>>> I = outer(N.z, N.z)
>>> A = RigidBody('A', Ac, a, M, (I, Ac))
```

Then one can either choose to evaluate the momenta of individual components of the system or of the entire system itself.

```
>>> linear_momentum(N,A)
M*l1*qld*N.y
>>> angular_momentum(0, N, Pa)
4*l1**2*m*qld*N.z
>>> linear_momentum(N, A, Pa)
(M*l1*qld + 2*l1*m*qld)*N.y
>>> angular_momentum(0, N, A, Pa)
(M*l1**2*qld + 4*l1**2*m*qld + qld)*N.z
```

It should be noted that the user can determine either momenta in any frame in [sympy.physics.mechanics](#) (page 1675) as the user is allowed to specify the reference frame when calling the function. In other words the user is not limited to determining just inertial linear and angular momenta. Please refer to the docstrings on each function to learn more about how each function works precisely.

## Kinetic Energy

The kinetic energy of a particle P is defined as

$$T_P = \frac{1}{2}m\mathbf{v}^2$$

where  $m$  is the mass of the particle P and  $\mathbf{v}$  is the velocity of the particle in the inertial frame. Similarly the kinetic energy of a rigid body B is defined as

$$T_B = T_t + T_r$$

where the translational kinetic energy is given by:

$$T_t = \frac{1}{2}m\mathbf{v}^* \cdot \mathbf{v}^*$$

and the rotational kinetic energy is given by:

$$T_r = \frac{1}{2}\omega \cdot \mathbf{I}^* \cdot \omega$$

where  $m$  is the mass of the rigid body,  $\mathbf{v}^*$  is the velocity of the mass center in the inertial frame,  $\omega$  is the inertial angular velocity of the body and  $\mathbf{I}^*$  is the central inertia dyadic.



## Potential Energy

Potential energy is defined as the energy possessed by a body or system by virtue of its position or arrangement.

Since there are a variety of definitions for potential energy, this is not discussed further here. One can learn more about this in any elementary text book on dynamics.

## Lagrangian

The Lagrangian of a body or a system of bodies is defined as:

$$\mathcal{L} = T - V$$

where  $T$  and  $V$  are the kinetic and potential energies respectively.

## Using energy functions in Mechanics

The following example shows how to use the energy functions in *sympy.physics.mechanics* (page 1675).

As was discussed above in the momenta functions, one first creates the system by going through an identical procedure.

```
>>> from sympy import symbols
>>> from sympy.physics.mechanics import dynamicsymbols, ReferenceFrame, outer
>>> from sympy.physics.mechanics import RigidBody, Particle
>>> from sympy.physics.mechanics import kinetic_energy, potential_energy, Point
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> m, M, l1, g, h, H = symbols('m M l1 g h H')
>>> omega = dynamicsymbols('omega')
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> O.set_vel(N, 0 * N.x)
>>> Ac = O.locatenew('Ac', l1 * N.x)
>>> P = Ac.locatenew('P', l1 * N.x)
>>> a = ReferenceFrame('a')
>>> a.set_ang_vel(N, omega * N.z)
>>> Ac.v2pt_theory(O, N, a)
l1*omega*N.y
>>> P.v2pt_theory(O, N, a)
2*l1*omega*N.y
>>> Pa = Particle('Pa', P, m)
>>> I = outer(N.z, N.z)
>>> A = RigidBody('A', Ac, a, M, (I, Ac))
```

The user can then determine the kinetic energy of any number of entities of the system:

```
>>> kinetic_energy(N, Pa)
2*l1**2*m*omega**2
```

(continues on next page)

(continued from previous page)

```
>>> kinetic_energy(N, Pa, A)
M*l1**2*omega**2/2 + 2*l1**2*m*omega**2 + omega**2/2
```

It should be noted that the user can determine either kinetic energy relative to any frame in *sympy.physics.mechanics* (page 1675) as the user is allowed to specify the reference frame when calling the function. In other words the user is not limited to determining just inertial kinetic energy.

For potential energies, the user must first specify the potential energy of every entity of the system using the *sympy.physics.mechanics.rigidbody.RigidBody.potential\_energy* (page 1749) property. The potential energy of any number of entities comprising the system can then be determined:

```
>>> Pa.potential_energy = m * g * h
>>> A.potential_energy = M * g * H
>>> potential_energy(A, Pa)
H*M*g + g*h*m
```

One can also determine the Lagrangian for this system:

```
>>> from sympy.physics.mechanics import Lagrangian
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> Lagrangian(N, Pa, A)
-H*M*g + M*l1**2*omega**2/2 - g*h*m + 2*l1**2*m*omega**2 + omega**2/2
```

Please refer to the docstrings to learn more about each function.

## Kane's Method in Physics/Mechanics

*sympy.physics.mechanics* (page 1675) provides functionality for deriving equations of motion using Kane's method [Kane1985]. This document will describe Kane's method as used in this module, but not how the equations are actually derived.

## Structure of Equations

In *sympy.physics.mechanics* (page 1675) we are assuming there are 5 basic sets of equations needed to describe a system. They are: holonomic constraints, non-holonomic constraints, kinematic differential equations, dynamic equations, and differentiated non-holonomic equations.

$$\begin{aligned} \mathbf{f}_h(q, t) &= 0 \\ \mathbf{k}_{nh}(q, t)u + \mathbf{f}_{nh}(q, t) &= 0 \\ \mathbf{k}_{kq}(q, t)\dot{q} + \mathbf{k}_{ku}(q, t)u + \mathbf{f}_k(q, t) &= 0 \\ \mathbf{k}_d(q, t)\dot{u} + \mathbf{f}_d(q, \dot{q}, u, t) &= 0 \\ \mathbf{k}_{dnh}(q, t)\dot{u} + \mathbf{f}_{dnh}(q, \dot{q}, u, t) &= 0 \end{aligned}$$

In *sympy.physics.mechanics* (page 1675) holonomic constraints are only used for the linearization process; it is assumed that they will be too complicated to solve for the dependent coordinate(s). If you are able to easily solve a holonomic constraint, you should consider

redefining your problem in terms of a smaller set of coordinates. Alternatively, the time-differentiated holonomic constraints can be supplied.

Kane's method forms two expressions,  $F_r$  and  $F_r^*$ , whose sum is zero. In this module, these expressions are rearranged into the following form:

$$\mathbf{M}(q, t)\dot{u} = \mathbf{f}(q, \dot{q}, u, t)$$

For a non-holonomic system with  $o$  total speeds and  $m$  motion constraints, we will get  $o - m$  equations. The mass-matrix/forcing equations are then augmented in the following fashion:

$$\mathbf{M}(q, t) = \begin{bmatrix} \mathbf{k}_d(q, t) \\ \mathbf{k}_{dnh}(q, t) \end{bmatrix}$$

$$(\text{forcing})(q, \dot{q}, u, t) = \begin{bmatrix} -\mathbf{f}_d(q, \dot{q}, u, t) \\ -\mathbf{f}_{dnh}(q, \dot{q}, u, t) \end{bmatrix}$$

## Kane's Method in Physics/Mechanics

The formulation of the equations of motion in `sympy.physics.mechanics` (page 1675) starts with creation of a `KanesMethod` object. Upon initialization of the `KanesMethod` object, an inertial reference frame needs to be supplied. along with some basic system information, such as coordinates and speeds

```
>>> from sympy.physics.mechanics import *
>>> N = ReferenceFrame('N')
>>> q1, q2, u1, u2 = dynamicsymbols('q1 q2 u1 u2')
>>> q1d, q2d, u1d, u2d = dynamicsymbols('q1 q2 u1 u2', 1)
>>> KM = KanesMethod(N, [q1, q2], [u1, u2])
```

It is also important to supply the order of coordinates and speeds properly if there are dependent coordinates and speeds. They must be supplied after independent coordinates and speeds or as a keyword argument; this is shown later.

```
>>> q1, q2, q3, q4 = dynamicsymbols('q1 q2 q3 q4')
>>> u1, u2, u3, u4 = dynamicsymbols('u1 u2 u3 u4')
>>> # Here we will assume q2 is dependent, and u2 and u3 are dependent
>>> # We need the constraint equations to enter them though
>>> KM = KanesMethod(N, [q1, q3, q4], [u1, u4])
```

Additionally, if there are auxiliary speeds, they need to be identified here. See the examples for more information on this. In this example `u4` is the auxiliary speed.

```
>>> KM = KanesMethod(N, [q1, q3, q4], [u1, u2, u3], u_auxiliary=[u4])
```

Kinematic differential equations must also be supplied; there are to be provided as a list of expressions which are each equal to zero. A trivial example follows:

```
>>> kd = [q1d - u1, q2d - u2]
```

Turning on `mechanics_printing()` makes the expressions significantly shorter and is recommended. Alternatively, the `mprint` and `mpprint` commands can be used.

If there are non-holonomic constraints, dependent speeds need to be specified (and so do dependent coordinates, but they only come into play when linearizing the system). The constraints need to be supplied in a list of expressions which are equal to zero, trivial motion and configuration constraints are shown below:

```

>>> N = ReferenceFrame('N')
>>> q1, q2, q3, q4 = dynamicsymbols('q1 q2 q3 q4')
>>> q1d, q2d, q3d, q4d = dynamicsymbols('q1 q2 q3 q4', 1)
>>> u1, u2, u3, u4 = dynamicsymbols('u1 u2 u3 u4')
>>> #Here we will assume q2 is dependent, and u2 and u3 are dependent
>>> speed_cons = [u2 - u1, u3 - u1 - u4]
>>> coord_cons = [q2 - q1]
>>> q_ind = [q1, q3, q4]
>>> q_dep = [q2]
>>> u_ind = [u1, u4]
>>> u_dep = [u2, u3]
>>> kd = [q1d - u1, q2d - u2, q3d - u3, q4d - u4]
>>> KM = KanesMethod(N, q_ind, u_ind, kd,
... q_dependent=q_dep,
... configuration_constraints=coord_cons,
... u_dependent=u_dep,
... velocity_constraints=speed_cons)

```

A dictionary returning the solved  $\dot{q}$ 's can also be solved for:

```

>>> mechanics_printing(pretty_print=False)
>>> KM.kindiffdict()
{q1': u1, q2': u2, q3': u3, q4': u4}

```

The final step in forming the equations of motion is supplying a list of bodies and particles, and a list of 2-tuples of the form (Point, Vector) or (ReferenceFrame, Vector) to represent applied forces and torques.

```

>>> N = ReferenceFrame('N')
>>> q, u = dynamicsymbols('q u')
>>> qd, ud = dynamicsymbols('q u', 1)
>>> P = Point('P')
>>> P.set_vel(N, u * N.x)
>>> Pa = Particle('Pa', P, 5)
>>> BL = [Pa]
>>> FL = [(P, 7 * N.x)]
>>> KM = KanesMethod(N, [q], [u], [qd - u])
>>> (fr, frstar) = KM.kanes_equations(BL, FL)
>>> KM.mass_matrix
Matrix([[5]])
>>> KM.forcing
Matrix([[7]])

```

When there are motion constraints, the mass matrix is augmented by the  $k_{dnh}(q, t)$  matrix, and the forcing vector by the  $f_{dnh}(q, \dot{q}, u, t)$  vector.

There are also the “full” mass matrix and “full” forcing vector terms, these include the kinematic differential equations; the mass matrix is of size  $(n + o) \times (n + o)$ , or square and the size of all coordinates and speeds.

```

>>> KM.mass_matrix_full
Matrix([
[1, 0],
[0, 5]])

```

(continues on next page)

(continued from previous page)

```
>>> KM.forcing_full
Matrix([
[u],
[7]])
```

Exploration of the provided examples is encouraged in order to gain more understanding of the KanesMethod object.

## Lagrange's Method in Physics/Mechanics

`sympy.physics.mechanics` (page 1675) provides functionality for deriving equations of motion using [Lagrange's method](#). This document will describe Lagrange's method as used in this module, but not how the equations are actually derived.

### Structure of Equations

In `sympy.physics.mechanics` (page 1675) we are assuming there are 3 basic sets of equations needed to describe a system; the constraint equations, the time differentiated constraint equations and the dynamic equations.

$$\begin{aligned}\mathbf{m}_c(q, t)\dot{q} + \mathbf{f}_c(q, t) &= 0 \\ \mathbf{m}_{dc}(\dot{q}, q, t)\ddot{q} + \mathbf{f}_{dc}(\dot{q}, q, t) &= 0 \\ \mathbf{m}_d(\dot{q}, q, t)\ddot{q} + \mathbf{c}(q, t)\lambda + \mathbf{f}_d(\dot{q}, q, t) &= 0\end{aligned}$$

In this module, the expressions formed by using Lagrange's equations of the second kind are rearranged into the following form:

$$\mathbf{M}(q, t)x = \mathbf{f}(q, \dot{q}, t)$$

where in the case of a system without constraints:

$$x = \ddot{q}$$

For a constrained system with  $n$  generalized speeds and  $m$  constraints, we will get  $n - m$  equations. The mass-matrix/forcing equations are then augmented in the following fashion:

$$\begin{aligned}x &= \begin{bmatrix} \ddot{q} \\ \lambda \end{bmatrix} \\ \mathbf{M}(q, t) &= [\mathbf{m}_d(q, t) \quad \mathbf{c}(q, t)] \\ \mathbf{F}(\dot{q}, q, t) &= [\mathbf{f}_d(q, \dot{q}, t)]\end{aligned}$$

## Lagrange's Method in Physics/Mechanics

The formulation of the equations of motion in `sympy.physics.mechanics` (page 1675) using Lagrange's Method starts with the creation of generalized coordinates and a Lagrangian. The Lagrangian can either be created with the Lagrangian function or can be a user supplied function. In this case we will supply the Lagrangian.

```
>>> from sympy.physics.mechanics import *
>>> q1, q2 = dynamicsymbols('q1 q2')
>>> q1d, q2d = dynamicsymbols('q1 q2', 1)
>>> L = q1d**2 + q2d**2
```

To formulate the equations of motion we create a `LagrangesMethod` object. The Lagrangian and generalized coordinates need to be supplied upon initialization.

```
>>> LM = LagrangesMethod(L, [q1, q2])
```

With that the equations of motion can be formed.

```
>>> mechanics_printing(pretty_print=False)
>>> LM.form_lagranges_equations()
Matrix([
[2*q1''],
[2*q2'']])
```

It is possible to obtain the mass matrix and the forcing vector.

```
>>> LM.mass_matrix
Matrix([
[2, 0],
[0, 2]])

>>> LM.forcing
Matrix([
[0],
[0]])
```

If there are any holonomic or non-holonomic constraints, they must be supplied as keyword arguments (`hol_coneqs` and `nonhol_coneqs` respectively) in a list of expressions which are equal to zero. Modifying the example above, the equations of motion can then be generated:

```
>>> LM = LagrangesMethod(L, [q1, q2], hol_coneqs=[q1 - q2])
```

When the equations of motion are generated in this case, the Lagrange multipliers are introduced; they are represented by `lam1` in this case. In general, there will be as many multipliers as there are constraint equations.

```
>>> LM.form_lagranges_equations()
Matrix([
[lam1 + 2*q1''],
[-lam1 + 2*q2'']])
```

Also in the case of systems with constraints, the ‘full’ mass matrix is augmented by the  $k_{dc}(q, t)$  matrix, and the forcing vector by the  $f_{dc}(q, \dot{q}, t)$  vector. The ‘full’ mass matrix is of size  $(2n + o) \times (2n + o)$ , i.e. it’s a square matrix.

```
>>> LM.mass_matrix_full
Matrix([
[1, 0, 0, 0, 0],
[0, 1, 0, 0, 0],
[0, 0, 2, 0, -1],
```

(continues on next page)

(continued from previous page)

```
[0, 0, 0, 2, 1],
[0, 0, 1, -1, 0]])
>>> LM.forcing_full
Matrix([
[q1'],
[q2'],
[0],
[0],
[0]])
```

If there are any non-conservative forces or moments acting on the system, they must also be supplied as keyword arguments in a list of 2-tuples of the form (Point, Vector) or (ReferenceFrame, Vector) where the Vector represents the non-conservative forces and torques. Along with this 2-tuple, the inertial frame must also be specified as a keyword argument. This is shown below by modifying the example above:

```
>>> N = ReferenceFrame('N')
>>> P = Point('P')
>>> P.set_vel(N, q1d * N.x)
>>> FL = [(P, 7 * N.x)]
>>> LM = LagrangesMethod(L, [q1, q2], forcelist=FL, frame=N)
>>> LM.form_lagranges_equations()
Matrix([
[2*q1'' - 7],
[2*q2'']])
```

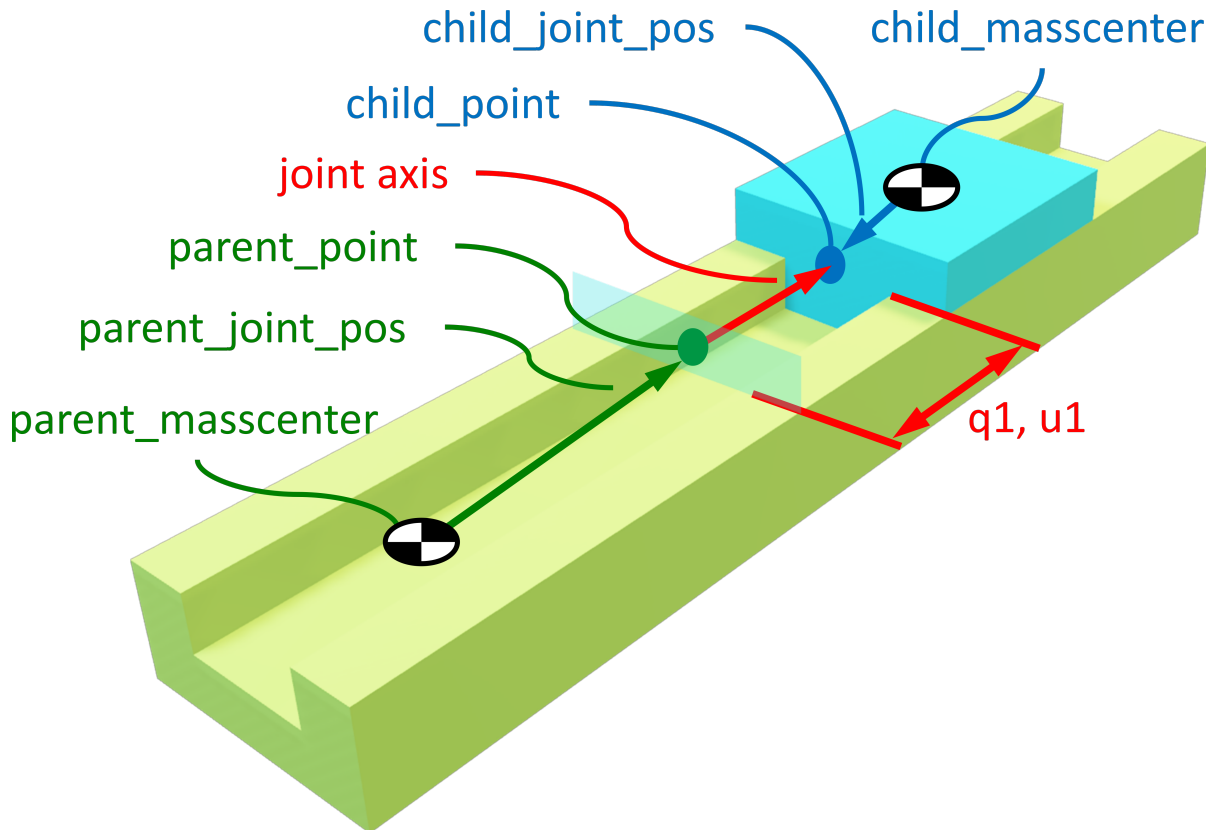
Exploration of the provided examples is encouraged in order to gain more understanding of the LagrangesMethod object.

## Joints Framework in Physics/Mechanics

*sympy.physics.mechanics* (page 1675) provides a joints framework. This system consists of two parts. The first are the *joints* (page 1770) themselves, which are used to create connections between *bodies* (page 1756). The second part is the *JointsMethod* (page 1780), which is used to form the equations of motion. Both of these parts are doing what we can call “book-keeping”: keeping track of the relationships between *bodies* (page 1756).

## Joints in Physics/Mechanics

The general task of the *joints* (page 1770) is creating kinematic relationships between *bodies* (page 1756). Each joint has a setup as shown in the image below (this is the example of the *PrismaticJoint* (page 1777)).



As can be seen in this image, each joint needs several objects in order to define the relationships. First off it needs two bodies: the parent body (shown in green) and the child body (shown in blue). Both of these bodies have a mass center from which the position of the joint is defined. In the parent body the vector from the mass center to the `parent_point` is called the `parent_joint_pos`. For the child body these are called the `child_point` and `child_joint_pos`. The orientation of the joint in each body is defined by the `parent_axis` and `child_axis`. These two vectors are aligned as explained in the *Joint* (page 1770) notes and are in the image parallel to the red vector. As last the joint also needs `dynamicsymbols()` (page 1666) as generalized coordinates and speeds. In the case of the *PrismaticJoint* (page 1777) shown above, the generalized coordinate `q_1` distance along the joint axis. And the generalized speed `u_1` is its velocity.

With the information listed above, the joint defines the following relationships. It first defines the kinematic differential equations, which relate the generalized coordinates to the generalized speeds. Next, it orients the parent and child body with respect to each other. After which it also defines their velocity relationships.

The following code shows the creation of a *PrismaticJoint* (page 1777) as shown above with arbitrary linked position vectors:

```
>>> from sympy.physics.mechanics import *
>>> mechanics_printing(pretty_print=False)
```

(continues on next page)



(continued from previous page)

```

>>> q1, u1 = dynamicsymbols('q1, u1')
>>> parent = Body('parent')
>>> child = Body('child')
>>> joint = PrismaticJoint(
... 'slider', parent, child, q1, u1,
... parent_joint_pos=parent.frame.x / 2 + parent.frame.y / 10,
... child_joint_pos=-(child.frame.x + child.frame.y) / 10,
... parent_axis=parent.frame.x, child_axis=child.frame.x)
>>> joint.kdes
[u1 - q1']
>>> child.masscenter.pos_from(parent.masscenter)
(q1 + 1/2)*parent.frame.x + 1/10*parent.frame.y + 1/10*child.frame.x + 1/
↪ 10*child.frame.y
>>> child.masscenter.vel(parent.frame)
u1*parent.frame.x

```

## JointsMethod in Physics/Mechanics

After defining the entire system you can use the [JointsMethod](#) (page 1780) to parse the system and form the equations of motion. In this process the [JointsMethod](#) (page 1780) only does the “book-keeping” of the joints. It uses another method, like the [KanesMethod](#) (page 1764), as its backend for forming the equations of motion.

In the code below we form the equations of motion of the single [PrismaticJoint](#) (page 1777) above.

```

>>> method = JointsMethod(parent, joint)
>>> method.form_eoms()
Matrix([[-child_mass*u1]])
>>> type(method.method) # The method working in the backend
<class 'sympy.physics.mechanics.kane.KanesMethod'>

```

## Symbolic Systems in Physics/Mechanics

The *SymbolicSystem* class in physics/mechanics is a location for the pertinent information of a multibody dynamic system. In its most basic form it contains the equations of motion for the dynamic system, however, it can also contain information regarding the loads that the system is subject to, the bodies that the system is comprised of and any additional equations the user feels is important for the system. The goal of this class is to provide a unified output format for the equations of motion that numerical analysis code can be designed around.

## SymbolicSystem Example Usage

This code will go over the manual input of the equations of motion for the simple pendulum that uses the Cartesian location of the mass as the generalized coordinates into *SymbolicSystem*.

The equations of motion are formed in the `physics/mechanics/examples`. In that spot the variables `q1` and `q2` are used in place of `x` and `y` and the reference frame is rotated 90 degrees.

```
>>> from sympy import atan, symbols, Matrix
>>> from sympy.physics.mechanics import (dynamicsymbols, ReferenceFrame,
... Particle, Point)
>>> import sympy.physics.mechanics.system as system
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
```

The first step will be to initialize all of the dynamic and constant symbols.

```
>>> x, y, u, v, lam = dynamicsymbols('x y u v lambda')
>>> m, l, g = symbols('m l g')
```

Next step is to define the equations of motion in multiple forms:

**[1] Explicit form where the kinematics and dynamics are combined**

$$\mathbf{x}' = \mathbf{F}_1(\mathbf{x}, t, \mathbf{r}, \mathbf{p})$$

**[2] Implicit form where the kinematics and dynamics are combined**

$$\mathbf{M}_2(\mathbf{x}, \mathbf{p}) \mathbf{x}' = \mathbf{F}_2(\mathbf{x}, t, \mathbf{r}, \mathbf{p})$$

**[3] Implicit form where the kinematics and dynamics are separate**

$$\mathbf{M}_3(\mathbf{q}, \mathbf{p}) \mathbf{u}' = \mathbf{F}_3(\mathbf{q}, \mathbf{u}, t, \mathbf{r}, \mathbf{p}) \quad \mathbf{q}' = \mathbf{G}(\mathbf{q}, \mathbf{u}, t, \mathbf{r}, \mathbf{p})$$

where

$\mathbf{x}$  : states, e.g.  $[\mathbf{q}, \mathbf{u}]$   $t$  : time  $\mathbf{r}$  : specified (exogenous) inputs  $\mathbf{p}$  : constants  $\mathbf{q}$  : generalized coordinates  $\mathbf{u}$  : generalized speeds  $\mathbf{F}_1$  : right hand side of the combined equations in explicit form  $\mathbf{F}_2$  : right hand side of the combined equations in implicit form  $\mathbf{F}_3$  : right hand side of the dynamical equations in implicit form  $\mathbf{M}_2$  : mass matrix of the combined equations in implicit form  $\mathbf{M}_3$  : mass matrix of the dynamical equations in implicit form  $\mathbf{G}$  : right hand side of the kinematical differential equations

```
>>> dyn_implicit_mat = Matrix([[1, 0, -x/m],
... [0, 1, -y/m],
... [0, 0, l**2/m]])
>>> dyn_implicit_rhs = Matrix([0, 0, u**2 + v**2 - g*y])
>>> comb_implicit_mat = Matrix([[1, 0, 0, 0, 0],
... [0, 1, 0, 0, 0],
... [0, 0, 1, 0, -x/m],
... [0, 0, 0, 1, -y/m],
... [0, 0, 0, 0, l**2/m]])
>>> comb_implicit_rhs = Matrix([u, v, 0, 0, u**2 + v**2 - g*y])
>>> kin_explicit_rhs = Matrix([u, v])
>>> comb_explicit_rhs = comb_implicit_mat.LUsolve(comb_implicit_rhs)
```

Now the reference frames, points and particles will be set up so this information can be passed into *system.SymbolicSystem* in the form of a bodies and loads iterable.

```

>>> theta = atan(x/y)
>>> omega = dynamicsymbols('omega')
>>> N = ReferenceFrame('N')
>>> A = N.orientnew('A', 'Axis', [theta, N.z])
>>> A.set_ang_vel(N, omega * N.z)
>>> O = Point('O')
>>> O.set_vel(N, 0)
>>> P = O.locatenew('P', l * A.x)
>>> P.v2pt_theory(O, N, A)
l*omega*A.y
>>> Pa = Particle('Pa', P, m)

```

Now the bodies and loads iterables need to be initialized.

```

>>> bodies = [Pa]
>>> loads = [(P, g * m * N.x)]

```

The equations of motion are in the form of a differential algebraic equation (DAE) and DAE solvers need to know which of the equations are the algebraic expressions. This information is passed into *SymbolicSystem* as a list specifying which rows are the algebraic equations. In this example it is a different row based on the chosen equations of motion format. The row index should always correspond to the mass matrix that is being input to the *SymbolicSystem* class but will always correspond to the row index of the combined dynamics and kinematics when being accessed from the *SymbolicSystem* class.

```

>>> alg_con = [2]
>>> alg_con_full = [4]

```

An iterable containing the states now needs to be created for the system. The *SymbolicSystem* class can determine which of the states are considered coordinates or speeds by passing in the indexes of the coordinates and speeds. If these indexes are not passed in the object will not be able to differentiate between coordinates and speeds.

```

>>> states = (x, y, u, v, lam)
>>> coord_idx = (0, 1)
>>> speed_idx = (2, 3)

```

Now the equations of motion instances can be created using the above mentioned equations of motion formats.

```

>>> symsystem1 = system.SymbolicSystem(states, comb_explicit_rhs,
... alg_con=alg_con_full, bodies=bodies,
... loads=loads)
>>> symsystem2 = system.SymbolicSystem(states, comb_implicit_rhs,
... mass_matrix=comb_implicit_mat,
... alg_con=alg_con_full,
... coord_idx=coord_idx)
>>> symsystem3 = system.SymbolicSystem(states, dyn_implicit_rhs,
... mass_matrix=dyn_implicit_mat,
... coordinate_derivatives=kin_explicit_
... rhs,
... alg_con=alg_con,
... coord_idx=coord_idx,
... speed_idx=speed_idx)

```

Like coordinates and speeds, the bodies and loads attributes can only be accessed if they are specified during initialization of the *SymbolicSystem* class. Lastly here are some attributes accessible from the *SymbolicSystem* class.

```
>>> symsystem1.states
Matrix([
[x],
[y],
[u],
[v],
[lambda]])
>>> symsystem2.coordinates
Matrix([
[x],
[y]])
>>> symsystem3.speeds
Matrix([
[u],
[v]])
>>> symsystem1.comb_explicit_rhs
Matrix([
[u],
[v],
[(-g*y + u**2 + v**2)*x/l**2],
[(-g*y + u**2 + v**2)*y/l**2],
[m*(-g*y + u**2 + v**2)/l**2]])
>>> symsystem2.comb_implicit_rhs
Matrix([
[u],
[v],
[0],
[0],
[-g*y + u**2 + v**2]])
>>> symsystem2.comb_implicit_mat
Matrix([
[1, 0, 0, 0, 0],
[0, 1, 0, 0, 0],
[0, 0, 1, 0, -x/m],
[0, 0, 0, 1, -y/m],
[0, 0, 0, 0, l**2/m]])
>>> symsystem3.dyn_implicit_rhs
Matrix([
[0],
[0],
[-g*y + u**2 + v**2]])
>>> symsystem3.dyn_implicit_mat
Matrix([
[1, 0, -x/m],
[0, 1, -y/m],
[0, 0, l**2/m]])
>>> symsystem3.kin_explicit_rhs
Matrix([
[u],
```

(continues on next page)

(continued from previous page)

```
[v]])
>>> symsystem1.alg_con
[4]
>>> symsystem1.bodies
(Pa,)
>>> symsystem1.loads
((P, g*m*N.x),)
```

## Linearization in Physics/Mechanics

*sympy.physics.mechanics* (page 1675) includes methods for linearizing the generated equations of motion (EOM) about an operating point (also known as the trim condition). Note that this operating point doesn't have to be an equilibrium position, it just needs to satisfy the equations of motion.

Linearization is accomplished by taking the first order Taylor expansion of the EOM about the operating point. When there are no dependent coordinates or speeds this is simply the jacobian of the right hand side about  $q$  and  $u$ . However, in the presence of constraints more care needs to be taken. The linearization methods provided here handle these constraints correctly.

## Background

In *sympy.physics.mechanics* (page 1675) we assume all systems can be represented in the following general form:

$$\begin{aligned} f_c(q, t) &= 0_{l \times 1} \\ f_v(q, u, t) &= 0_{m \times 1} \\ f_a(q, \dot{q}, u, \dot{u}, t) &= 0_{m \times 1} \\ f_0(q, \dot{q}, t) + f_1(q, u, t) &= 0_{n \times 1} \\ f_2(q, u, \dot{u}, t) + f_3(q, \dot{q}, u, r, t) + f_4(q, \lambda, t) &= 0_{(o-m+k) \times 1} \end{aligned}$$

where

$$\begin{aligned} q, \dot{q} &\in \mathbb{R}^n \\ u, \dot{u} &\in \mathbb{R}^o \\ r &\in \mathbb{R}^s \\ \lambda &\in \mathbb{R}^k \end{aligned}$$

In this form,

- $f_c$  represents the configuration constraint equations
- $f_v$  represents the velocity constraint equations
- $f_a$  represents the acceleration constraint equations
- $f_0$  and  $f_1$  form the kinematic differential equations
- $f_2$ ,  $f_3$ , and  $f_4$  form the dynamic differential equations
- $q$  and  $\dot{q}$  are the generalized coordinates and their derivatives

- $u$  and  $\dot{u}$  are the generalized speeds and their derivatives
- $r$  is the system inputs
- $\lambda$  is the Lagrange multipliers

This generalized form is held inside the `Linearizer` class, which performs the actual linearization. Both `KanesMethod` and `LagrangesMethod` objects have methods for forming the linearizer using the `to_linearizer` class method.

### A Note on Dependent Coordinates and Speeds

If the system being linearized contains constraint equations, this results in not all generalized coordinates being independent (i.e.  $q_1$  may depend on  $q_2$ ). With  $l$  configuration constraints, and  $m$  velocity constraints, there are  $l$  dependent coordinates and  $m$  dependent speeds.

In general, you may pick any of the coordinates and speeds to be dependent, but in practice some choices may result in undesirable singularities. Methods for deciding which coordinates/speeds to make dependent is behind the scope of this guide. For more information, please see [Blajer1994].

Once the system is coerced into the generalized form, the linearized EOM can be solved for. The methods provided in `sympy.physics.mechanics` (page 1675) allow for two different forms of the linearized EOM:

#### $M$ , $A$ , and $B$

In this form, the forcing matrix is linearized into two separate matrices  $A$  and  $B$ . This is the default form of the linearized EOM. The resulting equations are:

$$M \begin{bmatrix} \delta \dot{q} \\ \delta \dot{u} \\ \delta \lambda \end{bmatrix} = A \begin{bmatrix} \delta q_i \\ \delta u_i \end{bmatrix} + B [\delta r]$$

where

$$\begin{aligned} M &\in \mathbb{R}^{(n+o+k) \times (n+o+k)} \\ A &\in \mathbb{R}^{(n+o+k) \times (n-l+o-m)} \\ B &\in \mathbb{R}^{(n+o+k) \times s} \end{aligned}$$

Note that  $q_i$  and  $u_i$  are just the independent coordinates and speeds, while  $q$  and  $u$  contains both the independent and dependent coordinates and speeds.

#### $A$ and $B$

In this form, the linearized EOM are brought into explicit first order form, in terms of just the independent coordinates and speeds. This form is often used in stability analysis or control theory. The resulting equations are:

$$\begin{bmatrix} \delta \dot{q}_i \\ \delta \dot{u}_i \end{bmatrix} = A \begin{bmatrix} \delta q_i \\ \delta u_i \end{bmatrix} + B [\delta r]$$

where

$$\begin{aligned} A &\in \mathbb{R}^{(n-l+o-m) \times (n-l+o-m)} \\ B &\in \mathbb{R}^{(n-l+o-m) \times s} \end{aligned}$$

To use this form set `A_and_B=True` in the `linearize` class method.

## Linearizing Kane's Equations

After initializing the KanesMethod object and forming  $F_r$  and  $F_r^*$  using the kanes\_equations class method, linearization can be accomplished in a couple ways. The different methods will be demonstrated with a simple pendulum system:

```
>>> from sympy import symbols, Matrix
>>> from sympy.physics.mechanics import *
>>> q1 = dynamicsymbols('q1') # Angle of pendulum
>>> u1 = dynamicsymbols('u1') # Angular velocity
>>> q1d = dynamicsymbols('q1', 1)
>>> L, m, t, g = symbols('L, m, t, g')

>>> # Compose world frame
>>> N = ReferenceFrame('N')
>>> pN = Point('N*')
>>> pN.set_vel(N, 0)

>>> # A.x is along the pendulum
>>> A = N.orientnew('A', 'axis', [q1, N.z])
>>> A.set_ang_vel(N, u1*N.z)

>>> # Locate point P relative to the origin N*
>>> P = pN.locatenew('P', L*A.x)
>>> vel_P = P.v2pt_theory(pN, N, A)
>>> pP = Particle('pP', P, m)

>>> # Create Kinematic Differential Equations
>>> kde = Matrix([q1d - u1])

>>> # Input the force resultant at P
>>> R = m*g*N.x

>>> # Solve for eom with kanes method
>>> KM = KanesMethod(N, q_ind=[q1], u_ind=[u1], kd_eqs=kde)
>>> fr, frstar = KM.kanes_equations([pP], [(P, R)])
```

### 1. Using the Linearizer class directly:

A linearizer object can be created using the to\_linearizer class method. This coerces the representation found in the KanesMethod object into the generalized form described above. As the independent and dependent coordinates and speeds are specified upon creation of the KanesMethod object, there is no need to specify them here.

```
>>> linearizer = KM.to_linearizer()
```

The linearized EOM can then be formed with the linearize method of the Linearizer object:

```
>>> M, A, B = linearizer.linearize()
>>> M
Matrix([
[1, 0],
```

(continues on next page)

(continued from previous page)

```
[0, -L**2*m]])
>>> A
Matrix([
[0, 1],
[L*g*m*cos(q1(t)), 0]])
>>> B
Matrix(0, 0, [])
```

Alternatively, the  $A$  and  $B$  form can be generated instead by specifying `A_and_B=True`:

```
>>> A, B = linearizer.linearize(A_and_B=True)
>>> A
Matrix([
[0, 1],
[-g*cos(q1(t))/L, 0]])
>>> B
Matrix(0, 0, [])
```

An operating point can also be specified as a dictionary or an iterable of dictionaries. This will evaluate the linearized form at the specified point before returning the matrices:

```
>>> op_point = {q1: 0, u1: 0}
>>> A_op, B_op = linearizer.linearize(A_and_B=True, op_point=op_point)
>>> A_op
Matrix([
[0, 1],
[-g/L, 0]])
```

Note that the same effect can be had by applying `msubs` to the matrices generated without the `op_point` kwarg:

```
>>> assert msubs(A, op_point) == A_op
```

Sometimes the returned matrices may not be in the most simplified form. Simplification can be performed after the fact, or the `Linearizer` object can be made to perform simplification internally by setting the `simplify` kwarg to `True`.

## 2. Using the linearize class method:

The `linearize` method of the `KanesMethod` class is provided as a nice wrapper that calls to `_linearizer` internally, performs the linearization, and returns the result. Note that all the kwargs available in the `linearize` method described above are also available here:

```
>>> A, B, inp_vec = KM.linearize(A_and_B=True, op_point=op_point, new_
 ↳method=True)
>>> A
Matrix([
[0, 1],
[-g/L, 0]])
```

The additional output `inp_vec` is a vector containing all found dynamicsymbols not included in the generalized coordinate or speed vectors. These are assumed to be inputs to the system,



forming the  $r$  vector described in the background above. In this example there are no inputs, so the vector is empty:

```
>>> inp_vec
Matrix(0, 0, [])
```

### What's with the new\_method kwarg?

Previous releases of SymPy contained a linearization method for KanesMethod objects. This method is deprecated, and will be removed from future releases. Until then, you must set new\_method=True in all calls to KanesMethod.linearize. After the old method is removed, this kwarg will no longer be needed.

## Linearizing Lagrange's Equations

Linearization of Lagrange's equations proceeds much the same as that of Kane's equations. As before, the process will be demonstrated with a simple pendulum system:

```
>>> # Redefine A and P in terms of q1d, not u1
>>> A = N.orientnew('A', 'axis', [q1, N.z])
>>> A.set_ang_vel(N, q1d*N.z)
>>> P = pN.locatenew('P', L*A.x)
>>> vel_P = P.v2pt_theory(pN, N, A)
>>> pP = Particle('pP', P, m)

>>> # Solve for eom with Lagrange's method
>>> Lag = Lagrangian(N, pP)
>>> LM = LagrangesMethod(Lag, [q1], forcelist=[(P, R)], frame=N)
>>> lag_eqs = LM.form_lagranges_equations()
```

### 1. Using the Linearizer class directly:

A Linearizer object can be formed from a LagrangesMethod object using the to\_linearizer class method. The only difference between this process and that of the KanesMethod class is that the LagrangesMethod object doesn't already have its independent and dependent coordinates and speeds specified internally. These must be specified in the call to to\_linearizer. In this example there are no dependent coordinates and speeds, but if there were they would be included in the q\_dep and qd\_dep kwargs:

```
>>> linearizer = LM.to_linearizer(q_ind=[q1], qd_ind=[q1d])
```

Once in this form, everything is the same as it was before with the KanesMethod example:

```
>>> A, B = linearizer.linearize(A_and_B=True, op_point=op_point)
>>> A
Matrix([
[0, 1],
[-g/L, 0]])
```

## 2. Using the linearize class method:

Similar to KanesMethod, the LagrangesMethod class also provides a linearize method as a nice wrapper that calls to\_linearizer internally, performs the linearization, and returns the result. As before, the only difference is that the independent and dependent coordinates and speeds must be specified in the call as well:

```
>>> A, B, inp_vec = LM.linearize(q_ind=[q1], qd_ind=[q1d], A_and_B=True, op_
 ↪point=op_point)
>>> A
Matrix([
[0, 1],
[-g/L, 0]])
```

## Potential Issues

While the Linearizer class *should* be able to linearize all systems, there are some potential issues that could occur. These are discussed below, along with some troubleshooting tips for solving them.

### 1. Symbolic linearization with A\_and\_B=True is slow

This could be due to a number of things, but the most likely one is that solving a large linear system symbolically is an expensive operation. Specifying an operating point will reduce the expression size and speed this up. If a purely symbolic solution is desired though (for application of many operating points at a later period, for example) a way to get around this is to evaluate with A\_and\_B=False, and then solve manually after applying the operating point:

```
>>> M, A, B = linearizer.linearize()
>>> M_op = msubs(M, op_point)
>>> A_op = msubs(A, op_point)
>>> perm_mat = linearizer.perm_mat
>>> A_lin = perm_mat.T * M_op.LUsolve(A_op)
>>> A_lin
Matrix([
[0, 1],
[-g/L, 0]])
```

The fewer symbols in A and M before solving, the faster this solution will be. Thus, for large expressions, it may be to your benefit to delay conversion to the A and B form until most symbols are subbed in for their numeric values.

## 2. The linearized form has nan, zoo, or oo as matrix elements

There are two potential causes for this. The first (and the one you should check first) is that some choices of dependent coordinates will result in singularities at certain operating points. Coordinate partitioning in a systemic manner to avoid this is beyond the scope of this guide; see [Blajer1994] for more information.

The other potential cause for this is that the matrices may not have been in the most reduced form before the operating point was substituted in. A simple example of this behavior is:

```
>>> from sympy import sin, tan
>>> expr = sin(q1)/tan(q1)
>>> op_point = {q1: 0}
>>> expr.subs(op_point)
nan
```

Note that if this expression was simplified before substitution, the correct value results:

```
>>> expr.simplify().subs(op_point)
1
```

A good way of avoiding this hasn't been found yet. For expressions of reasonable size, using `msubs` with `smart=True` will apply an algorithm that tries to avoid these conditions. For large expressions though this is extremely time consuming.

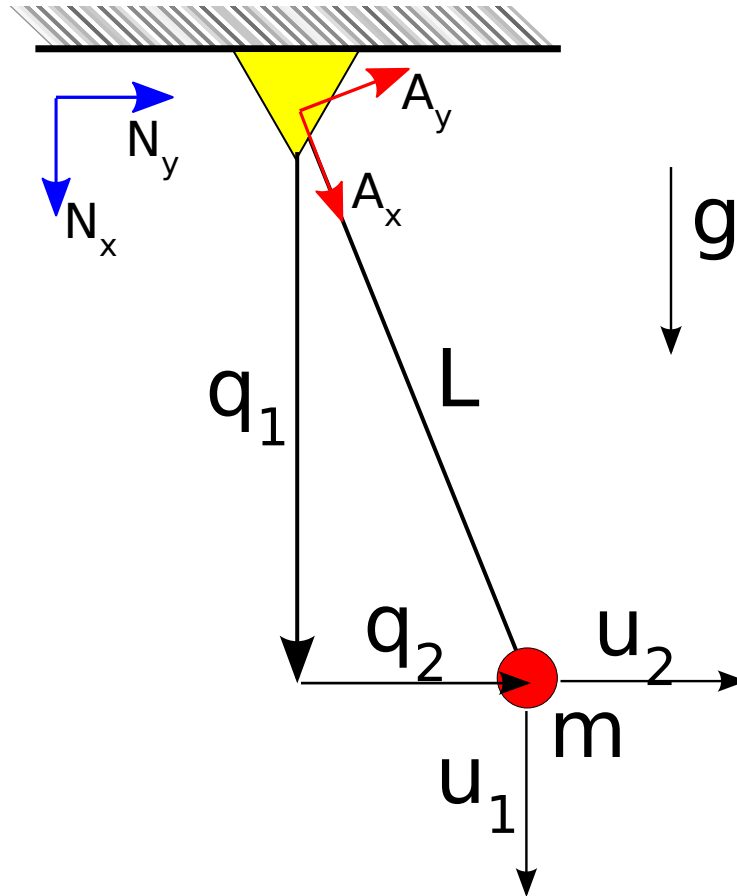
```
>>> msubs(expr, op_point, smart=True)
1
```

## Further Examples

The pendulum example used above was simple, but didn't include any dependent coordinates or speeds. For a more thorough example, the same pendulum was linearized with dependent coordinates using both Kane's and Lagrange's methods:

### Nonminimal Coordinates Pendulum

In this example we demonstrate the use of the functionality provided in `sympy.physics.mechanics` (page 1675) for deriving the equations of motion (EOM) for a pendulum with a nonminimal set of coordinates. As the pendulum is a one degree of freedom system, it can be described using one coordinate and one speed (the pendulum angle, and the angular velocity respectively). Choosing instead to describe the system using the  $x$  and  $y$  coordinates of the mass results in a need for constraints. The system is shown below:



The system will be modeled using both Kane's and Lagrange's methods, and the resulting EOM linearized. While this is a simple problem, it should illustrate the use of the linearization methods in the presence of constraints.

### Kane's Method

First we need to create the dynamicsymbols needed to describe the system as shown in the above diagram. In this case, the generalized coordinates  $q_1$  and  $q_2$  represent the mass  $x$  and  $y$  coordinates in the inertial  $N$  frame. Likewise, the generalized speeds  $u_1$  and  $u_2$  represent the velocities in these directions. We also create some symbols to represent the length and mass of the pendulum, as well as gravity and time.

```
>>> from sympy.physics.mechanics import *
>>> from sympy import symbols, atan, Matrix, solve
>>> # Create generalized coordinates and speeds for this non-minimal_
 realization
>>> # q1, q2 = N.x and N.y coordinates of pendulum
>>> # u1, u2 = N.x and N.y velocities of pendulum
>>> q1, q2 = dynamicsymbols('q1:3')
>>> q1d, q2d = dynamicsymbols('q1:3', level=1)
>>> u1, u2 = dynamicsymbols('u1:3')
>>> u1d, u2d = dynamicsymbols('u1:3', level=1)
>>> L, m, g, t = symbols('L, m, g, t')
```

Next, we create a world coordinate frame  $N$ , and its origin point  $N^*$ . The velocity of the origin is set to 0. A second coordinate frame  $A$  is oriented such that its x-axis is along the pendulum (as shown in the diagram above).

```
>>> # Compose world frame
>>> N = ReferenceFrame('N')
>>> pN = Point('N*')
>>> pN.set_vel(N, 0)

>>> # A.x is along the pendulum
>>> theta1 = atan(q2/q1)
>>> A = N.orientnew('A', 'axis', [theta1, N.z])
```

Locating the pendulum mass is then as easy as specifying its location with in terms of its x and y coordinates in the world frame. A Particle object is then created to represent the mass at this location.

```
>>> # Locate the pendulum mass
>>> P = pN.locatenew('P1', q1*N.x + q2*N.y)
>>> pP = Particle('pP', P, m)
```

The kinematic differential equations (KDEs) relate the derivatives of the generalized coordinates to the generalized speeds. In this case the speeds are the derivatives, so these are simple. A dictionary is also created to map  $\dot{q}$  to  $u$ :

```
>>> # Calculate the kinematic differential equations
>>> kde = Matrix([q1d - u1,
... q2d - u2])
>>> dq_dict = solve(kde, [q1d, q2d])
```

The velocity of the mass is then the time derivative of the position from the origin  $N^*$ :

```
>>> # Set velocity of point P
>>> P.set_vel(N, P.pos_from(pN).dt(N).subs(dq_dict))
```

As this system has more coordinates than degrees of freedom, constraints are needed. The configuration constraints relate the coordinates to each other. In this case the constraint is that the distance from the origin to the mass is always the length  $L$  (the pendulum doesn't get longer). Likewise, the velocity constraint is that the mass velocity in the  $A.x$  direction is always 0 (no radial velocity).

```
>>> f_c = Matrix([P.pos_from(pN).magnitude() - L])
>>> f_v = Matrix([P.vel(N).express(A).dot(A.x)])
>>> f_v.simplify()
```

The force on the system is just gravity, at point P.

```
>>> # Input the force resultant at P
>>> R = m*g*N.x
```

With the problem setup, the equations of motion can be generated using the `KanesMethod` class. As there are constraints, dependent and independent coordinates need to be provided to the class. In this case we'll use  $q_2$  and  $u_2$  as the independent coordinates and speeds:

```
>>> # Derive the equations of motion using the KanesMethod class.
>>> KM = KanesMethod(N, q_ind=[q2], u_ind=[u2], q_dependent=[q1],
... u_dependent=[u1], configuration_constraints=f_c,
... velocity_constraints=f_v, kd_eqs=kde)
>>> (fr, frstar) = KM.kanes_equations([pP],[P, R])
```

For linearization, operating points can be specified on the call, or be substituted in afterwards. In this case we'll provide them in the call, supplied in a list. The `A_and_B=True` kwarg indicates to solve invert the  $M$  matrix and solve for just the explicit linearized  $A$  and  $B$  matrices. The `simplify=True` kwarg indicates to simplify inside the linearize call, and return the presimplified matrices. The cost of doing this is small for simple systems, but for larger systems this can be a costly operation, and should be avoided.

```
>>> # Set the operating point to be straight down, and non-moving
>>> q_op = {q1: L, q2: 0}
>>> u_op = {u1: 0, u2: 0}
>>> ud_op = {u1d: 0, u2d: 0}
>>> # Perform the linearization
>>> A, B, inp_vec = KM.linearize(op_point=[q_op, u_op, ud_op], A_and_B=True,
... new_method=True, simplify=True)
>>> A
Matrix([
[0, 1],
[-g/L, 0]])
>>> B
Matrix(0, 0, [])
```

The resulting  $A$  matrix has dimensions  $2 \times 2$ , while the number of total states is  $\text{len}(q) + \text{len}(u) = 2 + 2 = 4$ . This is because for constrained systems the resulting  $A$  and  $B$  form has a partitioned state vector only containing the independent coordinates and speeds. Written out mathematically, the system linearized about this point would be written as:

$$\begin{bmatrix} \dot{q}_2 \\ \dot{u}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{g}{L} & 0 \end{bmatrix} \begin{bmatrix} q_2 \\ u_2 \end{bmatrix}$$

## Lagrange's Method

The derivation using Lagrange's method is very similar to the approach using Kane's method described above. As before, we first create the dynamicsymbols needed to describe the system. In this case, the generalized coordinates  $q_1$  and  $q_2$  represent the mass  $x$  and  $y$  coordinates in the inertial  $N$  frame. This results in the time derivatives  $\dot{q}_1$  and  $\dot{q}_2$  representing the velocities in these directions. We also create some symbols to represent the length and mass of the pendulum, as well as gravity and time.

```
>>> from sympy.physics.mechanics import *
>>> from sympy import symbols, atan, Matrix
>>> q1, q2 = dynamicsymbols('q1:3')
>>> q1d, q2d = dynamicsymbols('q1:3', level=1)
>>> L, m, g, t = symbols('L, m, g, t')
```

Next, we create a world coordinate frame  $N$ , and its origin point  $N^*$ . The velocity of the origin is set to 0. A second coordinate frame  $A$  is oriented such that its x-axis is along the pendulum (as shown in the diagram above).

```
>>> # Compose World Frame
>>> N = ReferenceFrame('N')
>>> pN = Point('N*')
>>> pN.set_vel(N, 0)
>>> # A.x is along the pendulum
>>> theta1 = atan(q2/q1)
>>> A = N.orientnew('A', 'axis', [theta1, N.z])
```

Locating the pendulum mass is then as easy as specifying its location with in terms of its x and y coordinates in the world frame. A Particle object is then created to represent the mass at this location.

```
>>> # Create point P, the pendulum mass
>>> P = pN.locatenew('P1', q1*N.x + q2*N.y)
>>> P.set_vel(N, P.pos_from(pN).dt(N))
>>> pP = Particle('pP', P, m)
```

As this system has more coordinates than degrees of freedom, constraints are needed. In this case only a single holonomic constraints is needed: the distance from the origin to the mass is always the length  $L$  (the pendulum doesn't get longer).

```
>>> # Holonomic Constraint Equations
>>> f_c = Matrix([q1**2 + q2**2 - L**2])
```

The force on the system is just gravity, at point P.

```
>>> # Input the force resultant at P
>>> R = m*g*N.x
```

With the problem setup, the Lagrangian can be calculated, and the equations of motion formed. Note that the call to `LagrangesMethod` includes the Lagrangian, the generalized coordinates, the constraints (specified by `hol_coneqs` or `nonhol_coneqs`), the list of (body, force) pairs, and the inertial frame. In contrast to the `KanesMethod` initializer, independent and dependent coordinates are not partitioned inside the `LagrangesMethod` object. Such a partition is supplied later.

```
>>> # Calculate the lagrangian, and form the equations of motion
>>> Lag = Lagrangian(N, pP)
>>> LM = LagrangesMethod(Lag, [q1, q2], hol_coneqs=f_c, forcelist=[(P, R)],
→ frame=N)
>>> lag_eqs = LM.form_lagranges_equations()
```

Next, we compose the operating point dictionary, set in the hanging at rest position:

```
>>> # Compose operating point
>>> op_point = {q1: L, q2: 0, q1d: 0, q2d: 0, q1d.diff(t): 0, q2d.diff(t): 0}
```

As there are constraints in the formulation, there will be corresponding Lagrange Multipliers. These may appear inside the linearized form as well, and thus should also be included inside the operating point dictionary. Fortunately, the `LagrangesMethod` class provides an easy way of solving for the multipliers at a given operating point using the `solve_multipliers` method.

```
>>> # Solve for multiplier operating point
>>> lam_op = LM.solve_multipliers(op_point=op_point)
```

With this solution, linearization can be completed. Note that in contrast to the `KanesMethod` approach, the `LagrangesMethod.linearize` method also requires the partitioning of the generalized coordinates and their time derivatives into independent and dependent vectors. This is the same as what was passed into the `KanesMethod` constructor above:

```
>>> op_point.update(lam_op)
>>> # Perform the Linearization
>>> A, B, inp_vec = LM.linearize([q2], [q2d], [q1], [q1d],
... op_point=op_point, A_and_B=True)
>>> A
Matrix([
[0, 1],
[-g/L, 0]])
>>> B
Matrix(0, 0, [])
```

The resulting  $A$  matrix has dimensions  $2 \times 2$ , while the number of total states is  $2 \times \text{len}(q) = 4$ . This is because for constrained systems the resulting  $A\_and\_B$  form has a partitioned state vector only containing the independent coordinates and their derivatives. Written out mathematically, the system linearized about this point would be written as:

$$\begin{bmatrix} \dot{q}_2 \\ \ddot{q}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ \frac{-g}{L} & 0 \end{bmatrix} \begin{bmatrix} q_2 \\ \dot{q}_2 \end{bmatrix}$$

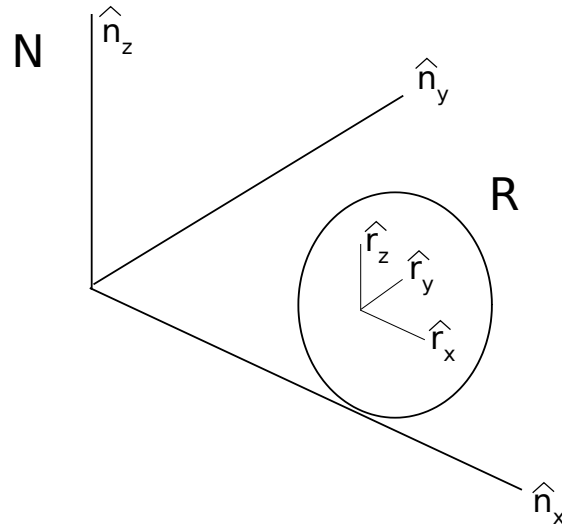
## Examples for Physics/Mechanics

Here are some examples that illustrate how one typically uses this module. We have ordered the examples roughly according to increasing difficulty. If you have used this module to do something others might find useful or interesting, consider adding it here!

### A rolling disc

The disc is assumed to be infinitely thin, in contact with the ground at only 1 point, and it is rolling without slip on the ground. See the image below.





We model the rolling disc in three different ways, to show more of the functionality of this module.

### A rolling disc, with Kane's method

Here the definition of the rolling disc's kinematics is formed from the contact point up, removing the need to introduce generalized speeds. Only 3 configuration and three speed variables are need to describe this system, along with the disc's mass and radius, and the local gravity (note that mass will drop out).

```
>>> from sympy import symbols, sin, cos, tan
>>> from sympy.physics.mechanics import *
>>> q1, q2, q3, u1, u2, u3 = dynamicsymbols('q1 q2 q3 u1 u2 u3')
>>> q1d, q2d, q3d, u1d, u2d, u3d = dynamicsymbols('q1 q2 q3 u1 u2 u3', 1)
>>> r, m, g = symbols('r m g')
>>> mechanics_printing(pretty_print=False)
```

The kinematics are formed by a series of simple rotations. Each simple rotation creates a new frame, and the next rotation is defined by the new frame's basis vectors. This example uses a 3-1-2 series of rotations, or Z, X, Y series of rotations. Angular velocity for this is defined using the second frame's basis (the lean frame); it is for this reason that we defined intermediate frames, rather than using a body-three orientation.

```
>>> N = ReferenceFrame('N')
>>> Y = N.orientnew('Y', 'Axis', [q1, N.z])
>>> L = Y.orientnew('L', 'Axis', [q2, Y.x])
>>> R = L.orientnew('R', 'Axis', [q3, L.y])
>>> w_R_N_qd = R.ang_vel_in(N)
>>> R.set_ang_vel(N, u1 * L.x + u2 * L.y + u3 * L.z)
```

This is the translational kinematics. We create a point with no velocity in N; this is the contact point between the disc and ground. Next we form the position vector from the contact point to the disc's center of mass. Finally we form the velocity and acceleration of the disc.

```
>>> C = Point('C')
>>> C.set_vel(N, 0)
>>> Dmc = C.locatenew('Dmc', r * L.z)
>>> Dmc.v2pt_theory(C, N, R)
r*u2*L.x - r*u1*L.y
```

This is a simple way to form the inertia dyadic. The inertia of the disc does not change within the lean frame as the disc rolls; this will make for simpler equations in the end.

```
>>> I = inertia(L, m / 4 * r**2, m / 2 * r**2, m / 4 * r**2)
>>> mprint(I)
m*r**2/4*(L.x|L.x) + m*r**2/2*(L.y|L.y) + m*r**2/4*(L.z|L.z)
```

Kinematic differential equations; how the generalized coordinate time derivatives relate to generalized speeds.

```
>>> kd = [dot(R.ang_vel_in(N) - w_R_N_qd, uv) for uv in L]
```

Creation of the force list; it is the gravitational force at the center of mass of the disc. Then we create the disc by assigning a Point to the center of mass attribute, a ReferenceFrame to the frame attribute, and mass and inertia. Then we form the body list.

```
>>> ForceList = [(Dmc, - m * g * Y.z)]
>>> BodyD = RigidBody('BodyD', Dmc, R, m, (I, Dmc))
>>> BodyList = [BodyD]
```

Finally we form the equations of motion, using the same steps we did before. Specify inertial frame, supply generalized coordinates and speeds, supply kinematic differential equation dictionary, compute  $F_r$  from the force list and  $F_r^*$  from the body list, compute the mass matrix and forcing terms, then solve for the  $u$  dots (time derivatives of the generalized speeds).

```
>>> KM = KanesMethod(N, q_ind=[q1, q2, q3], u_ind=[u1, u2, u3], kd_eqs=kd)
>>> (fr, frstar) = KM.kanes_equations(BodyList, ForceList)
>>> MM = KM.mass_matrix
>>> forcing = KM.forcing
>>> rhs = MM.inv() * forcing
>>> kdd = KM.kindiffdict()
>>> rhs = rhs.subs(kdd)
>>> rhs.simplify()
>>> mprint(rhs)
Matrix([
[(4*g*sin(q2) + 6*r*u2*u3 - r*u3**2*tan(q2))/(5*r)],
[
-2*u1*u3/3],
[
(-2*u2 + u3*tan(q2))*u1]])
```

## A rolling disc, with Kane's method and constraint forces

We will now revisit the rolling disc example, except this time we are bringing the non-contributing (constraint) forces into evidence. See [Kane1985] for a more thorough explanation of this. Here, we will turn on the automatic simplification done when doing vector operations. It makes the outputs nicer for small problems, but can cause larger vector operations to hang.

```
>>> from sympy import symbols, sin, cos, tan
>>> from sympy.physics.mechanics import *
>>> mechanics_printing(pretty_print=False)
>>> q1, q2, q3, u1, u2, u3 = dynamicsymbols('q1 q2 q3 u1 u2 u3')
>>> q1d, q2d, q3d, u1d, u2d, u3d = dynamicsymbols('q1 q2 q3 u1 u2 u3', 1)
>>> r, m, g = symbols('r m g')
```

These two lines introduce the extra quantities needed to find the constraint forces.

```
>>> u4, u5, u6, f1, f2, f3 = dynamicsymbols('u4 u5 u6 f1 f2 f3')
```

Most of the main code is the same as before.

```
>>> N = ReferenceFrame('N')
>>> Y = N.orientnew('Y', 'Axis', [q1, N.z])
>>> L = Y.orientnew('L', 'Axis', [q2, Y.x])
>>> R = L.orientnew('R', 'Axis', [q3, L.y])
>>> w_R_N_qd = R.ang_vel_in(N)
>>> R.set_ang_vel(N, u1 * L.x + u2 * L.y + u3 * L.z)
```

The definition of rolling without slip necessitates that the velocity of the contact point is zero; as part of bringing the constraint forces into evidence, we have to introduce speeds at this point, which will by definition always be zero. They are normal to the ground, along the path which the disc is rolling, and along the ground in a perpendicular direction.

```
>>> C = Point('C')
>>> C.set_vel(N, u4 * L.x + u5 * cross(Y.z, L.x) + u6 * Y.z)
>>> Dmc = C.locatenew('Dmc', r * L.z)
>>> vel = Dmc.v2pt_theory(C, N, R)
>>> I = inertia(L, m / 4 * r**2, m / 2 * r**2, m / 4 * r**2)
>>> kd = [dot(R.ang_vel_in(N) - w_R_N_qd, uv) for uv in L]
```

Just as we previously introduced three speeds as part of this process, we also introduce three forces; they are in the same direction as the speeds, and represent the constraint forces in those directions.

```
>>> ForceList = [(Dmc, - m * g * Y.z), (C, f1 * L.x + f2 * cross(Y.z, L.x) +
 ↪ f3 * Y.z)]
>>> BodyD = RigidBody('BodyD', Dmc, R, m, (I, Dmc))
>>> BodyList = [BodyD]

>>> KM = KanesMethod(N, q_ind=[q1, q2, q3], u_ind=[u1, u2, u3], kd_eqs=kd,
 ... u_auxiliary=[u4, u5, u6])
>>> (fr, frstar) = KM.kanes_equations(BodyList, ForceList)
>>> MM = KM.mass_matrix
>>> forcing = KM.forcing
```

(continues on next page)

(continued from previous page)

```

>>> rhs = MM.inv() * forcing
>>> kdd = KM.kindiffdict()
>>> rhs = rhs.subs(kdd)
>>> rhs.simplify()
>>> mprint(rhs)
Matrix([
[(4*g*sin(q2) + 6*r*u2*u3 - r*u3**2*tan(q2))/(5*r)],
[
(-2*u1*u3/3],
[
(-2*u2 + u3*tan(q2))*u1]])
>>> from sympy import trigsimp, signsimp, collect, factor_terms
>>> def simplify_auxiliary_eqs(w):
... return signsimp(trigsimp(collect(collect(factor_terms(w), f2), m*r)))
>>> mprint(KM.auxiliary_eqs.applyfunc(simplify_auxiliary_eqs))
Matrix([
[
-m*r*(u1*u3 + u2') + f1],
[-m*r*u1**2*sin(q2) - m*r*u2*u3/cos(q2) + m*r*cos(q2)*u1' + f2],
[
-g*m + m*r*(u1**2*cos(q2) + sin(q2)*u1') + f3]])

```

## A rolling disc using Lagrange's Method

Here the rolling disc is formed from the contact point up, removing the need to introduce generalized speeds. Only 3 configuration and 3 speed variables are needed to describe this system, along with the disc's mass and radius, and the local gravity.

```

>>> from sympy import symbols, cos, sin
>>> from sympy.physics.mechanics import *
>>> mechanics_printing(pretty_print=False)
>>> q1, q2, q3 = dynamicsymbols('q1 q2 q3')
>>> q1d, q2d, q3d = dynamicsymbols('q1 q2 q3', 1)
>>> r, m, g = symbols('r m g')

```

The kinematics are formed by a series of simple rotations. Each simple rotation creates a new frame, and the next rotation is defined by the new frame's basis vectors. This example uses a 3-1-2 series of rotations, or Z, X, Y series of rotations. Angular velocity for this is defined using the second frame's basis (the lean frame).

```

>>> N = ReferenceFrame('N')
>>> Y = N.orientnew('Y', 'Axis', [q1, N.z])
>>> L = Y.orientnew('L', 'Axis', [q2, Y.x])
>>> R = L.orientnew('R', 'Axis', [q3, L.y])

```

This is the translational kinematics. We create a point with no velocity in N; this is the contact point between the disc and ground. Next we form the position vector from the contact point to the disc's center of mass. Finally we form the velocity and acceleration of the disc.

```

>>> C = Point('C')
>>> C.set_vel(N, 0)
>>> Dmc = C.locatenew('Dmc', r * L.z)
>>> Dmc.v2pt_theory(C, N, R)
r*(sin(q2)*q1' + q3')*L.x - r*q2'*L.y

```

Forming the inertia dyadic.

```
>>> I = inertia(L, m / 4 * r**2, m / 2 * r**2, m / 4 * r**2)
>>> mprint(I)
m*r**2/4*(L.x|L.x) + m*r**2/2*(L.y|L.y) + m*r**2/4*(L.z|L.z)
>>> BodyD = RigidBody('BodyD', Dmc, R, m, (I, Dmc))
```

We then set the potential energy and determine the Lagrangian of the rolling disc.

```
>>> BodyD.potential_energy = - m * g * r * cos(q2)
>>> Lag = Lagrangian(N, BodyD)
```

Then the equations of motion are generated by initializing the `LagrangesMethod` object. Finally we solve for the generalized accelerations( $q$  double dots) with the `rhs` method.

```
>>> q = [q1, q2, q3]
>>> l = LagrangesMethod(Lag, q)
>>> le = l.form_lagranges_equations()
>>> le.simplify(); le
Matrix([
[m*r**2*(6*sin(q2)*q3'' + 5*sin(2*q2)*q1'*q2' + 6*cos(q2)*q2'*q3' -
 ↪ 5*cos(2*q2)*q1''/2 + 7*q1''/2)/4],
[
 ↪ m*r*(4*g*sin(q2) - 5*r*sin(2*q2)*q1''**2/2 -
 ↪ 6*r*cos(q2)*q1'*q3' + 5*r*q2'')/4],
[
 ↪ 3*m*r**2*(sin(q2)*q1'' +
 ↪ cos(q2)*q1'*q2' + q3'')/2]])
>>> lrhs = l.rhs(); lrhs.simplify(); lrhs
Matrix([
[
 ↪ q1'],
[
 ↪ q2'],
[
 ↪ q3'],
[
 ↪ -2*(2*tan(q2)*q1' + 3*q3'/cos(q2))*q2'],
[-4*g*sin(q2)/(5*r) + sin(2*q2)*q1''**2/2 + 6*cos(q2)*q1'*q3'/5],
[
 ↪ (-5*cos(q2)*q1' + 6*tan(q2)*q3' + 4*q1'/cos(q2))*q2']]])
```

## A bicycle

The bicycle is an interesting system in that it has multiple rigid bodies, non-holonomic constraints, and a holonomic constraint. The linearized equations of motion are presented in [Meijaard2007]. This example will go through construction of the equations of motion in *sympy.physics.mechanics* (page 1675).

```
>>> from sympy import *
>>> from sympy.physics.mechanics import *
>>> print('Calculation of Linearized Bicycle \'A\' Matrix, '
... 'with States: Roll, Steer, Roll Rate, Steer Rate')
Calculation of Linearized Bicycle "A" Matrix, with States: Roll, Steer, Roll
 ↪ Rate, Steer Rate
```

Note that this code has been crudely ported from Autolev, which is the reason for some of the unusual naming conventions. It was purposefully as similar as possible in order to aid initial porting & debugging. We set `Vector.simp` to `False` (in case it has been set `True` elsewhere), since it slows down the computations:

```
>>> Vector.simp = False
>>> mechanics_printing(pretty_print=False)
```

Declaration of Coordinates & Speeds: A simple definition for  $\dot{q}$ ,  $\dot{q}_d = u$ , is used in this code. Speeds are: yaw frame ang. rate, roll frame ang. rate, rear wheel frame ang. rate (spinning motion), frame ang. rate (pitching motion), steering frame ang. rate, and front wheel ang. rate (spinning motion). Wheel positions are ignorable coordinates, so they are not introduced.

```
>>> q1, q2, q3, q4, q5 = dynamicsymbols('q1 q2 q3 q4 q5')
>>> q1d, q2d, q4d, q5d = dynamicsymbols('q1 q2 q4 q5', 1)
>>> u1, u2, u3, u4, u5, u6 = dynamicsymbols('u1 u2 u3 u4 u5 u6')
>>> u1d, u2d, u3d, u4d, u5d, u6d = dynamicsymbols('u1 u2 u3 u4 u5 u6', 1)
```

Declaration of System's Parameters: The below symbols should be fairly self-explanatory.

```
>>> WFrads, WRrads, htangle, forkoffset = symbols('WFrads WRrads htangle_
↳ forkoffset')
>>> forklength, framelength, forkcg1 = symbols('forklength framelength forkcg1
↳ ')
>>> forkcg3, framecg1, framecg3, Iwr11 = symbols('forkcg3 framecg1 framecg3_
↳ Iwr11')
>>> Iwr22, Iwf11, Iwf22, Iframe11 = symbols('Iwr22 Iwf11 Iwf22 Iframe11')
>>> Iframe22, Iframe33, Iframe31, Ifork11 = \
... symbols('Iframe22 Iframe33 Iframe31 Ifork11')
>>> Ifork22, Ifork33, Ifork31, g = symbols('Ifork22 Ifork33 Ifork31 g')
>>> mframe, mfork, mwf, mwr = symbols('mframe mfork mwf mwr')
```

Set up reference frames for the system: N - inertial Y - yaw R - roll WR - rear wheel, rotation angle is ignorable coordinate so not oriented Frame - bicycle frame TempFrame - statically rotated frame for easier reference inertia definition Fork - bicycle fork TempFork - statically rotated frame for easier reference inertia definition WF - front wheel, again poses an ignorable coordinate

```
>>> N = ReferenceFrame('N')
>>> Y = N.orientnew('Y', 'Axis', [q1, N.z])
>>> R = Y.orientnew('R', 'Axis', [q2, Y.x])
>>> Frame = R.orientnew('Frame', 'Axis', [q4 + htangle, R.y])
>>> WR = ReferenceFrame('WR')
>>> TempFrame = Frame.orientnew('TempFrame', 'Axis', [-htangle, Frame.y])
>>> Fork = Frame.orientnew('Fork', 'Axis', [q5, Frame.x])
>>> TempFork = Fork.orientnew('TempFork', 'Axis', [-htangle, Fork.y])
>>> WF = ReferenceFrame('WF')
```

Kinematics of the Bicycle: First block of code is forming the positions of the relevant points rear wheel contact -> rear wheel's center of mass -> frame's center of mass + frame/fork connection -> fork's center of mass + front wheel's center of mass -> front wheel contact point.

```
>>> WR_cont = Point('WR_cont')
>>> WR_mc = WR_cont.locatenew('WR_mc', WRrads * R.z)
>>> Steer = WR_mc.locatenew('Steer', framelength * Frame.z)
>>> Frame_mc = WR_mc.locatenew('Frame_mc', -framecg1 * Frame.x + framecg3 *_
↳)
```

(continues on next page)

(continued from previous page)

```

 ↪Frame.z)
>>> Fork_mc = Steer.locatenew('Fork_mc', -forkcg1 * Fork.x + forkcg3 * Fork.z)
>>> WF_mc = Steer.locatenew('WF_mc', forklength * Fork.x + forkoffset * Fork.
 ↪z)
>>> WF_cont = WF_mc.locatenew('WF_cont', WFrads*(dot(Fork.y, Y.z)*Fork.y - \
 ... Y.z).normalize())

```

Set the angular velocity of each frame: Angular accelerations end up being calculated automatically by differentiating the angular velocities when first needed. ::  $u_1$  is yaw rate  $u_2$  is roll rate  $u_3$  is rear wheel rate  $u_4$  is frame pitch rate  $u_5$  is fork steer rate  $u_6$  is front wheel rate

```

>>> Y.set_ang_vel(N, u1 * Y.z)
>>> R.set_ang_vel(Y, u2 * R.x)
>>> WR.set_ang_vel(Frame, u3 * Frame.y)
>>> Frame.set_ang_vel(R, u4 * Frame.y)
>>> Fork.set_ang_vel(Frame, u5 * Fork.x)
>>> WF.set_ang_vel(Fork, u6 * Fork.y)

```

Form the velocities of the points, using the 2-point theorem. Accelerations again are calculated automatically when first needed.

```

>>> WR_cont.set_vel(N, 0)
>>> WR_mc.v2pt_theory(WR_cont, N, WR)
WFrads*(u1*sin(q2) + u3 + u4)*R.x - WFrads*u2*R.y
>>> Steer.v2pt_theory(WR_mc, N, Frame)
WFrads*(u1*sin(q2) + u3 + u4)*R.x - WFrads*u2*R.y + framelength*(u1*sin(q2) +
 ↪u4)*Frame.x - framelength*(-u1*sin(htangle + q4)*cos(q2) + u2*cos(htangle +
 ↪q4))*Frame.y
>>> Frame_mc.v2pt_theory(WR_mc, N, Frame)
WFrads*(u1*sin(q2) + u3 + u4)*R.x - WFrads*u2*R.y + framecg3*(u1*sin(q2) +
 ↪u4)*Frame.x + (-framecg1*(u1*cos(htangle + q4)*cos(q2) + u2*sin(htangle +
 ↪q4)) - framecg3*(-u1*sin(htangle + q4)*cos(q2) + u2*cos(htangle +
 ↪q4)))*Frame.y + framecg1*(u1*sin(q2) + u4)*Frame.z
>>> Fork_mc.v2pt_theory(Steer, N, Fork)
WFrads*(u1*sin(q2) + u3 + u4)*R.x - WFrads*u2*R.y + framelength*(u1*sin(q2) +
 ↪u4)*Frame.x - framelength*(-u1*sin(htangle + q4)*cos(q2) + u2*cos(htangle +
 ↪q4))*Frame.y + forkcg3*((sin(q2)*cos(q5) + sin(q5)*cos(htangle +
 ↪q4)*cos(q2))*u1 + u2*sin(htangle + q4)*sin(q5) + u4*cos(q5))*Fork.x + (-
 ↪forkcg1*((-sin(q2)*sin(q5) + cos(htangle + q4)*cos(q2)*cos(q5))*u1 +
 ↪u2*sin(htangle + q4)*cos(q5) - u4*sin(q5)) - forkcg3*(-u1*sin(htangle +
 ↪q4)*cos(q2) + u2*cos(htangle + q4) + u5))*Fork.y +
 ↪forkcg1*((sin(q2)*cos(q5) + sin(q5)*cos(htangle + q4)*cos(q2))*u1 +
 ↪u2*sin(htangle + q4)*sin(q5) + u4*cos(q5))*Fork.z
>>> WF_mc.v2pt_theory(Steer, N, Fork)
WFrads*(u1*sin(q2) + u3 + u4)*R.x - WFrads*u2*R.y + framelength*(u1*sin(q2) +
 ↪u4)*Frame.x - framelength*(-u1*sin(htangle + q4)*cos(q2) + u2*cos(htangle +
 ↪q4))*Frame.y + forkoffset*((sin(q2)*cos(q5) + sin(q5)*cos(htangle +
 ↪q4)*cos(q2))*u1 + u2*sin(htangle + q4)*sin(q5) + u4*cos(q5))*Fork.x +
 ↪(forklength*((-sin(q2)*sin(q5) + cos(htangle + q4)*cos(q2)*cos(q5))*u1 +
 ↪u2*sin(htangle + q4)*cos(q5) - u4*sin(q5)) - forkoffset*(-u1*sin(htangle +
 ↪q4)*cos(q2) + u2*cos(htangle + q4) + u5))*Fork.y -

```

(continues on next page)

(continued from previous page)

```

→ forklength*((sin(q2)*cos(q5) + sin(q5)*cos(htangle + q4)*cos(q2))*u1 +
→ u2*sin(htangle + q4)*sin(q5) + u4*cos(q5))*Fork.z
>>> WF_cont.v2pt_theory(WF_mc, N, WF)
- WFrads*((-sin(q2)*sin(q5)*cos(htangle + q4) + cos(q2)*cos(q5))*u6 +
→ u4*cos(q2) + u5*sin(htangle + q4)*sin(q2))/sqrt((-sin(q2)*cos(q5) -
→ sin(q5)*cos(htangle + q4)*cos(q2))*(sin(q2)*cos(q5) + sin(q5)*cos(htangle +
→ q4)*cos(q2)) + 1)*Y.x + WFrads*(u2 + u5*cos(htangle + q4) + u6*sin(htangle +
→ q4)*sin(q5))/sqrt((-sin(q2)*cos(q5) - sin(q5)*cos(htangle +
→ q4)*cos(q2))*(sin(q2)*cos(q5) + sin(q5)*cos(htangle + q4)*cos(q2)) + 1)*Y.y
→ + WRad*(u1*sin(q2) + u3 + u4)*R.x - WRad*u2*R.y + framelength*(u1*sin(q2)
→ + u4)*Frame.x - framelength*(-u1*sin(htangle + q4)*cos(q2) + u2*cos(htangle
→ + q4))*Frame.y + (-WFrads*(sin(q2)*cos(q5) + sin(q5)*cos(htangle +
→ q4)*cos(q2))*((-sin(q2)*sin(q5) + cos(htangle + q4)*cos(q2)*cos(q5))*u1 +
→ u2*sin(htangle + q4)*cos(q5) - u4*sin(q5))/sqrt((-sin(q2)*cos(q5) -
→ sin(q5)*cos(htangle + q4)*cos(q2))*(sin(q2)*cos(q5) + sin(q5)*cos(htangle +
→ q4)*cos(q2)) + 1) + forkoffset*((sin(q2)*cos(q5) + sin(q5)*cos(htangle +
→ q4)*cos(q2))*u1 + u2*sin(htangle + q4)*sin(q5) + u4*cos(q5))*Fork.x +
→ (forklength*((-sin(q2)*sin(q5) + cos(htangle + q4)*cos(q2)*cos(q5))*u1 +
→ u2*sin(htangle + q4)*cos(q5) - u4*sin(q5)) - forkoffset*(-u1*sin(htangle +
→ q4)*cos(q2) + u2*cos(htangle + q4) + u5))*Fork.y + (WFrads*(sin(q2)*cos(q5)
→ + sin(q5)*cos(htangle + q4)*cos(q2))*(-u1*sin(htangle + q4)*cos(q2) +
→ u2*cos(htangle + q4) + u5)/sqrt((-sin(q2)*cos(q5) - sin(q5)*cos(htangle +
→ q4)*cos(q2))*(sin(q2)*cos(q5) + sin(q5)*cos(htangle + q4)*cos(q2)) + 1) -
→ forklength*((sin(q2)*cos(q5) + sin(q5)*cos(htangle + q4)*cos(q2))*u1 +
→ u2*sin(htangle + q4)*sin(q5) + u4*cos(q5))*Fork.z

```

Sets the inertias of each body. Uses the inertia frame to construct the inertia dyadics. Wheel inertias are only defined by principal moments of inertia, and are in fact constant in the frame and fork reference frames; it is for this reason that the orientations of the wheels does not need to be defined. The frame and fork inertias are defined in the 'Temp' frames which are fixed to the appropriate body frames; this is to allow easier input of the reference values of the benchmark paper. Note that due to slightly different orientations, the products of inertia need to have their signs flipped; this is done later when entering the numerical value.

```

>>> Frame_I = (inertia(TempFrame, Iframe11, Iframe22, Iframe33, 0, 0,
... Iframe31), Frame_mc)
>>> Fork_I = (inertia(TempFork, Ifork11, Ifork22, Ifork33, 0, 0, Ifork31),
→ Fork_mc)
>>> WR_I = (inertia(Frame, Iwr11, Iwr22, Iwr11), WR_mc)
>>> WF_I = (inertia(Fork, Iwf11, Iwf22, Iwf11), WF_mc)

```

Declaration of the RigidBody containers.

```

>>> BodyFrame = RigidBody('BodyFrame', Frame_mc, Frame, mframe, Frame_I)
>>> BodyFork = RigidBody('BodyFork', Fork_mc, Fork, mfork, Fork_I)
>>> BodyWR = RigidBody('BodyWR', WR_mc, WR, mwr, WR_I)
>>> BodyWF = RigidBody('BodyWF', WF_mc, WF, mwf, WF_I)

>>> print('Before Forming the List of Nonholonomic Constraints.')
Before Forming the List of Nonholonomic Constraints.

```

The kinematic differential equations; they are defined quite simply. Each entry in this list is equal to zero.



```
>>> kd = [q1d - u1, q2d - u2, q4d - u4, q5d - u5]
```

The nonholonomic constraints are the velocity of the front wheel contact point dotted into the X, Y, and Z directions; the yaw frame is used as it is “closer” to the front wheel (1 less DCM connecting them). These constraints force the velocity of the front wheel contact point to be 0 in the inertial frame; the X and Y direction constraints enforce a “no-slip” condition, and the Z direction constraint forces the front wheel contact point to not move away from the ground frame, essentially replicating the holonomic constraint which does not allow the frame pitch to change in an invalid fashion.

```
>>> conlist_speed = [dot(WF_cont.vel(N), Y.x),
... dot(WF_cont.vel(N), Y.y),
... dot(WF_cont.vel(N), Y.z)]
```

The holonomic constraint is that the position from the rear wheel contact point to the front wheel contact point when dotted into the normal-to-ground plane direction must be zero; effectively that the front and rear wheel contact points are always touching the ground plane. This is actually not part of the dynamic equations, but instead is necessary for the linearization process.

```
>>> conlist_coord = [dot(WF_cont.pos_from(WR_cont), Y.z)]
```

The force list; each body has the appropriate gravitational force applied at its center of mass.

```
>>> FL = [(Frame_mc, -mframe * g * Y.z), (Fork_mc, -mfork * g * Y.z),
... (WF_mc, -mwf * g * Y.z), (WR_mc, -mwr * g * Y.z)]
>>> BL = [BodyFrame, BodyFork, BodyWR, BodyWF]
```

The N frame is the inertial frame, coordinates are supplied in the order of independent, dependent coordinates. The kinematic differential equations are also entered here. Here the independent speeds are specified, followed by the dependent speeds, along with the non-holonomic constraints. The dependent coordinate is also provided, with the holonomic constraint. Again, this is only comes into play in the linearization process, but is necessary for the linearization to correctly work.

```
>>> KM = KanesMethod(N, q_ind=[q1, q2, q5],
... q_dependent=[q4], configuration_constraints=conlist_coord,
... u_ind=[u2, u3, u5],
... u_dependent=[u1, u4, u6], velocity_constraints=conlist_speed,
... kd_eqs=kd)
>>> print('Before Forming Generalized Active and Inertia Forces, Fr and Fr*')
Before Forming Generalized Active and Inertia Forces, Fr and Fr*
>>> (fr, frstar) = KM.kanes_equations(BL, FL)
>>> print('Base Equations of Motion Computed')
Base Equations of Motion Computed
```

This is the start of entering in the numerical values from the benchmark paper to validate the eigenvalues of the linearized equations from this model to the reference eigenvalues. Look at the aforementioned paper for more information. Some of these are intermediate values, used to transform values from the paper into the coordinate systems used in this model.

```
>>> PaperRadRear = 0.3
>>> PaperRadFront = 0.35
>>> HTA = evalf.N(pi/2-pi/10)
```

(continues on next page)

(continued from previous page)

```

>>> TrailPaper = 0.08
>>> rake = evalf.N(-(TrailPaper*sin(HTA) - (PaperRadFront*cos(HTA))))
>>> PaperWb = 1.02
>>> PaperFrameCgX = 0.3
>>> PaperFrameCgZ = 0.9
>>> PaperForkCgX = 0.9
>>> PaperForkCgZ = 0.7
>>> FrameLength = evalf.N(PaperWb*sin(HTA) - (rake - \
... (PaperRadFront - PaperRadRear)*cos(HTA)))
>>> FrameCGNorm = evalf.N((PaperFrameCgZ - PaperRadRear - \
... (PaperFrameCgX/sin(HTA))*cos(HTA))*sin(HTA))
>>> FrameCGPar = evalf.N((PaperFrameCgX / sin(HTA) + \
... (PaperFrameCgZ - PaperRadRear - \
... PaperFrameCgX / sin(HTA) * cos(HTA)) *
... cos(HTA)))
>>> tempa = evalf.N((PaperForkCgZ - PaperRadFront))
>>> tempb = evalf.N((PaperWb-PaperForkCgX))
>>> tempc = evalf.N(sqrt(tempa**2 + tempb**2))
>>> PaperForkL = evalf.N((PaperWb*cos(HTA) - \
... (PaperRadFront - PaperRadRear)*sin(HTA)))
>>> ForkCGNorm = evalf.N(rake + (tempc * sin(pi/2 - \
... HTA - acos(tempa/tempc))))
>>> ForkCGPar = evalf.N(tempc * cos((pi/2 - HTA) - \
... acos(tempa/tempc)) - PaperForkL)

```

Here is the final assembly of the numerical values. The symbol 'v' is the forward speed of the bicycle (a concept which only makes sense in the upright, static equilibrium case?). These are in a dictionary which will later be substituted in. Again the sign on the *product* of inertia values is flipped here, due to different orientations of coordinate systems.

```

>>> v = Symbol('v')
>>> val_dict = {
... WFr1: PaperRadFront,
... WRr1: PaperRadRear,
... htangle: HTA,
... forkoffset: rake,
... forklenght: PaperForkL,
... framelength: FrameLength,
... forkcg1: ForkCGPar,
... forkcg3: ForkCGNorm,
... framecg1: FrameCGNorm,
... framecg3: FrameCGPar,
... Iwr11: 0.0603,
... Iwr22: 0.12,
... Iwf11: 0.1405,
... Iwf22: 0.28,
... Ifork11: 0.05892,
... Ifork22: 0.06,
... Ifork33: 0.00708,
... Ifork31: 0.00756,
... Iframe11: 9.2,
... Iframe22: 11,

```

(continues on next page)

(continued from previous page)

```

... Iframe33: 2.8,
... Iframe31: -2.4,
... mfork: 4,
... mframe: 85,
... mwf: 3,
... mwr: 2,
... g: 9.81,
... q1: 0,
... q2: 0,
... q4: 0,
... q5: 0,
... u1: 0,
... u2: 0,
... u3: v/PaperRadRear,
... u4: 0,
... u5: 0,
... u6: v/PaperRadFront}
>>> kdd = KM.kindiffdict()
>>> print('Before Linearization of the \"Forcing\" Term')
Before Linearization of the \"Forcing\" Term

```

Linearizes the forcing vector; the equations are set up as  $MM \dot{u} = \text{forcing}$ , where  $MM$  is the mass matrix,  $\dot{u}$  is the vector representing the time derivatives of the generalized speeds, and forcing is a vector which contains both external forcing terms and internal forcing terms, such as centripetal or Coriolis forces. This actually returns a matrix with as many rows as *total* coordinates and speeds, but only as many columns as independent coordinates and speeds. (Note that below this is commented out, as it takes a few minutes to run, which is not good when performing the doctests)

```
>>> # forcing_lin = KM.linearize()[0].subs(sub_dict)
```

As mentioned above, the size of the linearized forcing terms is expanded to include both  $q$ 's and  $u$ 's, so the mass matrix must have this done as well. This will likely be changed to be part of the linearized process, for future reference.

```

>>> MM_full = (KM._k_kqdot).row_join(zeros(4, 6)).col_join(
... (zeros(6, 4)).row_join(KM.mass_matrix))
>>> print('Before Substitution of Numerical Values')
Before Substitution of Numerical Values

```

I think this is pretty self explanatory. It takes a really long time though. I've experimented with using `evalf` with substitution, this failed due to maximum recursion depth being exceeded; I also tried lambdifying this, and it is also not successful. (again commented out due to speed)

```

>>> # MM_full = MM_full.subs(val_dict)
>>> # forcing_lin = forcing_lin.subs(val_dict)
>>> # print('Before .evalf() call')

>>> # MM_full = MM_full.evalf()
>>> # forcing_lin = forcing_lin.evalf()

```

Finally, we construct an "A" matrix for the form  $\dot{x} = A x$  ( $x$  being the state vector, although in this case, the sizes are a little off). The following line extracts only the minimum entries

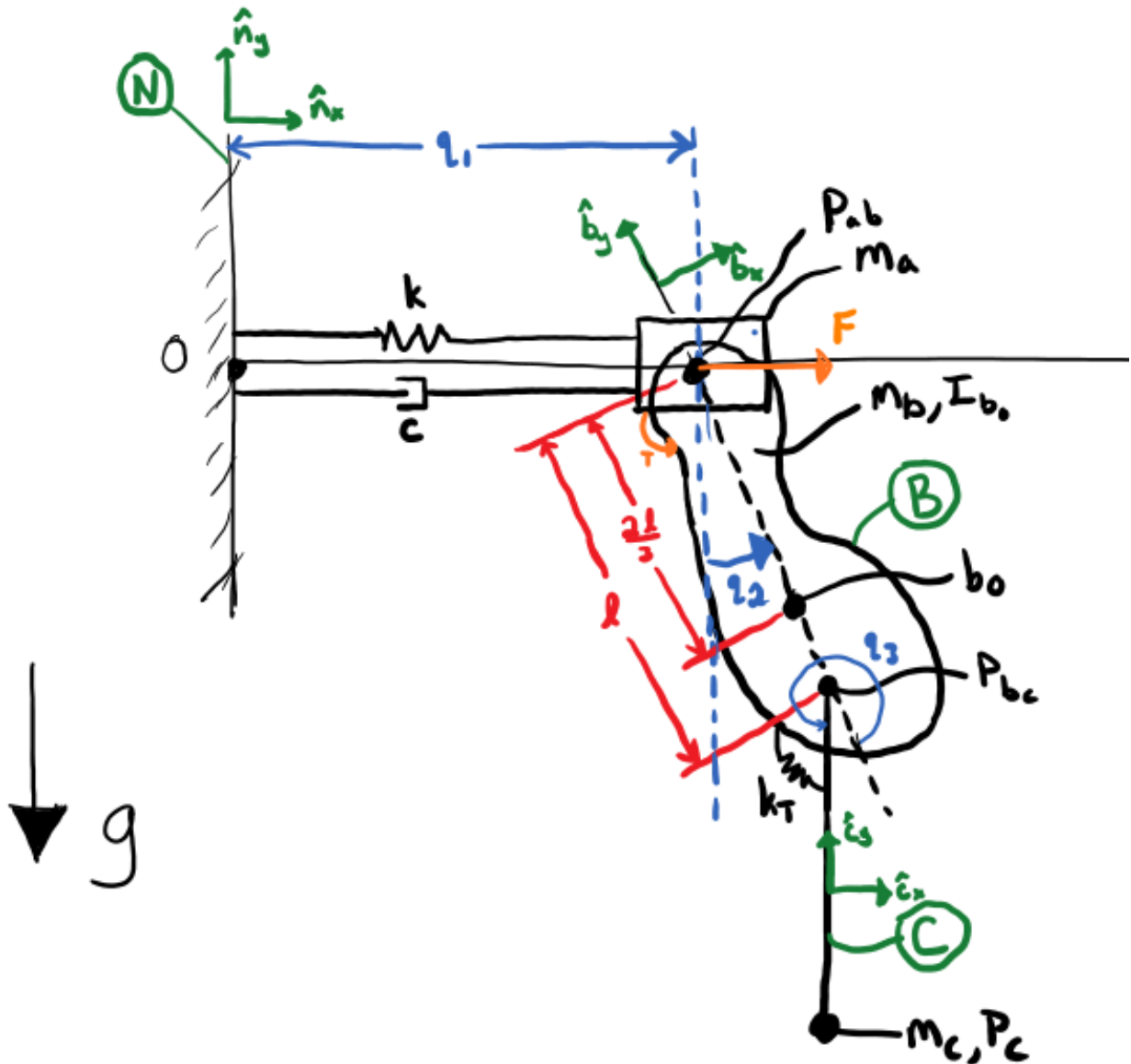
required for eigenvalue analysis, which correspond to rows and columns for lean, steer, lean rate, and steer rate. (this is all commented out due to being dependent on the above code, which is also commented out):

```
>>> # Amat = MM_full.inv() * forcing_lin
>>> # A = Amat.extract([1,2,4,6],[1,2,3,5])
>>> # print(A)
>>> # print('v = 1')
>>> # print(A.subs(v, 1).eigenvals())
>>> # print('v = 2')
>>> # print(A.subs(v, 2).eigenvals())
>>> # print('v = 3')
>>> # print(A.subs(v, 3).eigenvals())
>>> # print('v = 4')
>>> # print(A.subs(v, 4).eigenvals())
>>> # print('v = 5')
>>> # print(A.subs(v, 5).eigenvals())
```

Upon running the above code yourself, enabling the commented out lines, compare the computed eigenvalues to those in the referenced paper. This concludes the bicycle example.

## Multi Degree of Freedom Holonomic System

In this example we demonstrate the use of the functionality provided in [\*sympy.physics.mechanics\*](#) (page 1675) for deriving the equations of motion (EOM) of a holonomic system that includes both particles and rigid bodies with contributing forces and torques, some of which are specified forces and torques. The system is shown below:



The system will be modeled using `JointsMethod`. First we need to create the dynamicsymbols needed to describe the system as shown in the above diagram. In this case, the generalized coordinates  $q_1$  represent lateral distance of block from wall,  $q_2$  represents angle of the compound pendulum from vertical,  $q_3$  represents angle of the simple pendulum from the compound pendulum. The generalized speeds  $u_1$  represents lateral speed of block,  $u_2$  represents lateral speed of compound pendulum and  $u_3$  represents angular speed of C relative to B.

We also create some symbols to represent the length and mass of the pendulum, as well as gravity and others.

```
>>> from sympy import zeros, symbols
>>> from sympy.physics.mechanics import Body, PinJoint, PrismaticJoint,
 ↳ JointsMethod, inertia
>>> from sympy.physics.mechanics import dynamicsymbols
>>> q1, q2, q3, u1, u2, u3 = dynamicsymbols('q1, q2, q3, u1, u2, u3')
```

(continues on next page)

(continued from previous page)

```
>>> l, k, c, g, kT = symbols('l, k, c, g, kT')
>>> ma, mb, mc, IBzz = symbols('ma, mb, mc, IBzz')
```

Next, we create the bodies and connect them using joints to establish the kinematics.

```
>>> wall = Body('N')
>>> block = Body('A', mass=ma)
>>> IB = inertia(block.frame, 0, 0, IBzz)
>>> compound_pend = Body('B', mass=mb, central_inertia=IB)
>>> simple_pend = Body('C', mass=mc)

>>> bodies = (wall, block, compound_pend, simple_pend)

>>> slider = PrismaticJoint('J1', wall, block, coordinates=q1, speeds=u1)
>>> rev1 = PinJoint('J2', block, compound_pend, coordinates=q2, speeds=u2,
... child_axis=compound_pend.z, child_joint_pos=l*2/
↳ 3*compound_pend.y,
... parent_axis=block.z)
>>> rev2 = PinJoint('J3', compound_pend, simple_pend, coordinates=q3,
↳ speeds=u3,
... child_axis=simple_pend.z, parent_joint_pos=-l/3*compound_
↳ pend.y,
... parent_axis=compound_pend.z, child_joint_pos=l*simple_
↳ pend.y)

>>> joints = (slider, rev1, rev2)
```

Now we can apply loads (forces and torques) to the bodies, gravity acts on all bodies, a linear spring and damper act on block and wall, a rotational linear spring acts on C relative to B specified torque T acts on compound\_pend and block, specified force F acts on block.

```
>>> F, T = dynamicsymbols('F, T')
>>> block.apply_force(F*block.x)
>>> block.apply_force(-k*q1*block.x, reaction_body=wall)
>>> block.apply_force(-c*u1*block.x, reaction_body=wall)
>>> compound_pend.apply_torque(T*compound_pend.z, reaction_body=block)
>>> simple_pend.apply_torque(-kT*q3*simple_pend.z, reaction_body=compound_
↳ pend)
>>> block.apply_force(-wall.y*block.mass*g)
>>> compound_pend.apply_force(-wall.y*compound_pend.mass*g)
>>> simple_pend.apply_force(-wall.y*simple_pend.mass*g)
```

With the problem setup, the equations of motion can be generated using the JointsMethod class with KanesMethod in backend.

```
>>> method = JointsMethod(wall, slider, rev1, rev2)
>>> method.form_eoms()
Matrix([
[
... -c*u1(t) - k*q1(t) +
↳ 2*l*mb*u2(t)**2*sin(q2(t))/3 - l*mc*(-sin(q2(t))*sin(q3(t)) +
↳ cos(q2(t))*cos(q3(t)))*Derivative(u3(t), t) - l*mc*(-sin(q2(t))*cos(q3(t)) -
↳ sin(q3(t))*cos(q2(t)))*(u2(t) + u3(t))**2 + l*mc*u2(t)**2*sin(q2(t)) -
↳
```

(continues on next page)

(continued from previous page)

```

→ (2*l*mb*cos(q2(t))/3 + mc*(l*(-sin(q2(t))*sin(q3(t)) +
→ cos(q2(t))*cos(q3(t))) + l*cos(q2(t)))*Derivative(u2(t), t) - (ma + mb +
→ mc)*Derivative(u1(t), t) + F(t)],
[-2*g*l*mb*sin(q2(t))/3 - g*l*mc*(sin(q2(t))*cos(q3(t)) +
→ sin(q3(t))*cos(q2(t))) - g*l*mc*sin(q2(t)) + l**2*mc*(u2(t) +
→ u3(t))**2*sin(q3(t)) - l**2*mc*u2(t)**2*sin(q3(t)) - mc*(l**2*cos(q3(t)) +
→ l**2)*Derivative(u3(t), t) - (2*l*mb*cos(q2(t))/3 + mc*(l*(-
→ sin(q2(t))*sin(q3(t)) + cos(q2(t))*cos(q3(t))) +
→ l*cos(q2(t)))*Derivative(u1(t), t) - (IBzz + 4*l**2*mb/9 +
→ mc*(2*l**2*cos(q3(t)) + 2*l**2))*Derivative(u2(t), t) + T(t)],
[
→
→
→ -g*l*mc*(sin(q2(t))*cos(q3(t)) + sin(q3(t))*cos(q2(t))) -
→ kT*q3(t) - l**2*mc*u2(t)**2*sin(q3(t)) - l**2*mc*Derivative(u3(t), t) -
→ l*mc*(-sin(q2(t))*sin(q3(t)) + cos(q2(t))*cos(q3(t)))*Derivative(u1(t), t) -
→ mc*(l**2*cos(q3(t)) + l**2)*Derivative(u2(t), t)]]

>>> method.mass_matrix_full
Matrix([
[1, 0, 0,
→
→
→ 0,
→
→
→ 0],
[0, 1, 0,
→
→
→ 0,
→
→
→ 0],
[0, 0, 1,
→
→
→ 0,
→
→
→ 0],
[0, 0, 0,
→
→
→ ma + mb + mc, 2*l*mb*cos(q2(t))/3 + mc*(l*(-
→ sin(q2(t))*sin(q3(t)) + cos(q2(t))*cos(q3(t))) + l*cos(q2(t))), l*mc*(-
→ sin(q2(t))*sin(q3(t)) + cos(q2(t))*cos(q3(t)))],
[0, 0, 0, 2*l*mb*cos(q2(t))/3 + mc*(l*(-sin(q2(t))*sin(q3(t)) +
→ cos(q2(t))*cos(q3(t))) + l*cos(q2(t))),
→
→ IBzz + 4*l**2*mb/9 + mc*(2*l**2*cos(q3(t)) + 2*l**2),
→
→ mc*(l**2*cos(q3(t)) + l**2)],
[0, 0, 0,
→
→
→ l*mc*(-sin(q2(t))*sin(q3(t))
→ + cos(q2(t))*cos(q3(t))),
→
→
→ mc*(l**2*cos(q3(t)) + l**2),
→
→
→ l**2*mc]])

>>> method.forcing_full
Matrix([
[
→
→
→ u1(t)],
[
→
→
→ u2(t)],

```

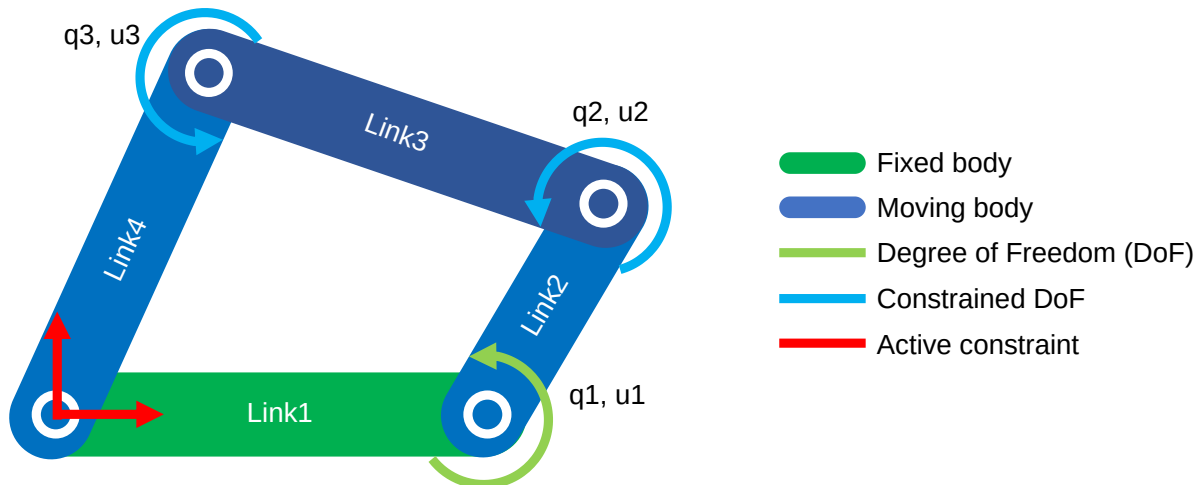
(continues on next page)

(continued from previous page)

```
[
 u3(t)],
 -c*u1(t) - k*q1(t) + 2*l*mb*u2(t)**2*sin(q2(t))/3 - l*mc*(-
 sin(q2(t))*cos(q3(t)) - sin(q3(t))*cos(q2(t)))*(u2(t) + u3(t))**2 +
 l*mc*u2(t)**2*sin(q2(t)) + F(t)],
 [-2*g*l*mb*sin(q2(t))/3 - g*l*mc*(sin(q2(t))*cos(q3(t)) +
 sin(q3(t))*cos(q2(t))) - g*l*mc*sin(q2(t)) + l**2*mc*(u2(t) +
 u3(t))**2*sin(q3(t)) - l**2*mc*u2(t)**2*sin(q3(t)) + T(t)],
 [
 -g*l*mc*(sin(q2(t))*cos(q3(t)) + sin(q3(t))*cos(q2(t))) - kT*q3(t) -
 l**2*mc*u2(t)**2*sin(q3(t))]]]
```

## A four bar linkage

The four bar linkage is a common example used in mechanics, which can be formulated with only two holonomic constraints. This example will make use of joints functionality provided in [sympy.physics.mechanics](#) (page 1675). In summary we will use bodies and joints to define the open loop system. Next, we define the configuration constraints to close the loop. The `JointsMethod` will be used to do the “book-keeping” of the open-loop system. From this we will get the input used in combination with the constraints to manually setup the [KanesMethod](#) (page 1764) as the backend.



First we need to create the `dynamicsymbols()` (page 1666) needed to describe the system as shown in the above diagram. In this case, the generalized coordinates  $q_1$ ,  $q_2$  and  $q_3$  represent the angles between the links. Likewise, the generalized speeds  $u_1$ ,  $u_2$  and  $u_3$  represent the angular velocities between the links. We also create some `symbols()` (page 978) to represent the lengths and density of the links.

```
>>> from sympy import symbols, Matrix, solve, simplify
>>> from sympy.physics.mechanics import *
>>> mechanics_printing(pretty_print=False)
>>> q1, q2, q3, u1, u2, u3 = dynamicsymbols('q1:4, u1:4')
>>> l1, l2, l3, l4, rho = symbols('l1:5, rho')
```

With all symbols defined, we can now define the bodies.



```
>>> N = ReferenceFrame('N')
>>> inertias = [inertia(N, 0, 0, rho * l ** 3 / 12) for l in (l1, l2, l3, l4)]
>>> link1 = Body('Link1', frame=N, mass=rho * l1, central_inertia=inertias[0])
>>> link2 = Body('Link2', mass=rho * l2, central_inertia=inertias[1])
>>> link3 = Body('Link3', mass=rho * l3, central_inertia=inertias[2])
>>> link4 = Body('Link4', mass=rho * l4, central_inertia=inertias[3])
```

Next, we also define the first three joints.

```
>>> joint1 = PinJoint('J1', link1, link2, coordinates=q1, speeds=u1,
... parent_axis=link1.z, parent_joint_pos=l1 / 2 * link1.x,
... child_axis=link2.z, child_joint_pos=-l2 / 2 * link2.x)
>>> joint2 = PinJoint('J2', link2, link3, coordinates=q2, speeds=u2,
... parent_axis=link2.z, parent_joint_pos=l2 / 2 * link2.x,
... child_axis=link3.z, child_joint_pos=-l3 / 2 * link3.x)
>>> joint3 = PinJoint('J3', link3, link4, coordinates=q3, speeds=u3,
... parent_axis=link3.z, parent_joint_pos=l3 / 2 * link3.x,
... child_axis=link4.z, child_joint_pos=-l4 / 2 * link4.x)
```

Now we can formulate the holonomic constraint that will close the kinematic loop.

```
>>> loop = link4.masscenter.pos_from(link1.masscenter) + l1 / 2 * link1.x +
↳ l4 / 2 * link4.x
>>> fh = Matrix([loop.dot(link1.x), loop.dot(link1.y)])
```

In order to generate the equations of motions, we will use the `JointsMethod` as our frontend. Before setting up the [KanesMethod](#) (page 1764) as its backend we need to calculate the velocity constraints.

```
>>> method = JointsMethod(link1, joint1, joint2, joint3)
>>> t = dynamicsymbols._t
>>> qdots = solve(method.kdes, [q1.diff(t), q2.diff(t), q3.diff(t)])
>>> fhd = fh.diff(t).subs(qdots)
```

Now we can setup the [KanesMethod](#) (page 1764) as the backend and compute the equations of motion.

```
>>> method._method = KanesMethod(
... method.frame, q_ind=[q1], u_ind=[u1], q_dependent=[q2, q3],
... u_dependent=[u2, u3], kd_eqs=method.kdes,
... configuration_constraints=fh, velocity_constraints=fhd,
... forcelist=method.loads, bodies=method.bodies)
>>> simplify(method.method._form_eoms())
Matrix([[l2*rho*(-2*l2**2*sin(q3)*u1' + 3*l2*l3*u1**2*sin(q2 + q3)*sin(q2) +
↳ 3*l2*l3*sin(q2)*cos(q2 + q3)*u1' - 3*l2*l3*sin(q3)*u1' +
↳ 3*l2*l4*u1**2*sin(q2 + q3)*sin(q2) + 3*l2*l4*sin(q2)*cos(q2 + q3)*u1' +
↳ 3*l3**2*u1**2*sin(q2)*sin(q3) + 6*l3**2*u1*u2*sin(q2)*sin(q3) +
↳ 3*l3**2*u2**2*sin(q2)*sin(q3) + 2*l3**2*sin(q2)*cos(q3)*u1' +
↳ 2*l3**2*sin(q2)*cos(q3)*u2' - l3**2*sin(q3)*cos(q2)*u1' -
↳ l3**2*sin(q3)*cos(q2)*u2' + 3*l3*l4*u1**2*sin(q2)*sin(q3) +
↳ 6*l3*l4*u1*u2*sin(q2)*sin(q3) + 3*l3*l4*u2**2*sin(q2)*sin(q3) +
↳ 3*l3*l4*sin(q2)*cos(q3)*u1' + 3*l3*l4*sin(q2)*cos(q3)*u2' + l4**2*sin(q2)*u1
↳ ' + l4**2*sin(q2)*u2' + l4**2*sin(q2)*u3')/(6*sin(q3))]])
```

## Potential Issues/Advanced Topics/Future Features in Physics/Mechanics

This document will describe some of the more advanced functionality that this module offers but which is not part of the “official” interface. Here, some of the features that will be implemented in the future will also be covered, along with unanswered questions about proper functionality. Also, common problems will be discussed, along with some solutions.

### Common Issues

Here issues with numerically integrating code, choice of `dynamicsymbols` for coordinate and speed representation, printing, differentiating, and substitution will occur.

### Numerically Integrating Code

See Future Features: Code Output

### Differentiating

Differentiation of very large expressions can take some time in SymPy; it is possible for large expressions to take minutes for the derivative to be evaluated. This will most commonly come up in linearization.

### Choice of Coordinates and Speeds

The Kane object is set up with the assumption that the generalized speeds are not the same symbol as the time derivatives of the generalized coordinates. This isn't to say that they can't be the same, just that they have to have a different symbol. If you did this:

```
>> KM.coords([q1, q2, q3])
>> KM.speeds([q1d, q2d, q3d])
```

Your code would not work. Currently, kinematic differential equations are required to be provided. It is at this point that we hope the user will discover they should not attempt the behavior shown in the code above.

This behavior might not be true for other methods of forming the equations of motion though.

### Printing

The default printing options are to use sorting for `Vector` and `Dyad` measure numbers, and have unsorted output from the `mprint`, `mpprint`, and `mlatex` functions. If you are printing something large, please use one of those functions, as the sorting can increase printing time from seconds to minutes.

## Substitution

There are two common issues with substitution in mechanics:

- When subbing in expressions for `dynamicsymbols`, sympy's normal subs will substitute in for derivatives of the dynamic symbol as well:

```
>>> from sympy.physics.mechanics import dynamicsymbols
>>> x = dynamicsymbols('x')
>>> expr = x.diff() + x
>>> sub_dict = {x: 1}
>>> expr.subs(sub_dict)
Derivative(1, t) + 1
```

In this case, `x` was replaced with `1` inside the `Derivative` as well, which is undesired.

- Substitution into large expressions can be slow.

If your substitution is simple (direct replacement of expressions with other expressions, such as when evaluating at an operating point) it is recommended to use the provided `msubs` function, as it is significantly faster, and handles the derivative issue appropriately:

```
>>> from sympy.physics.mechanics import msubs
>>> msubs(expr, sub_dict)
Derivative(x(t), t) + 1
```

## Linearization

Currently, the linearization methods don't support cases where there are non-coordinate, non-speed dynamic symbols outside of the "dynamic equations". It also does not support cases where time derivatives of these types of dynamic symbols show up. This means if you have kinematic differential equations which have a non-coordinate, non-speed dynamic symbol, it will not work. It also means if you have defined a system parameter (say a length or distance or mass) as a dynamic symbol, its time derivative is likely to show up in the dynamic equations, and this will prevent linearization.

## Acceleration of Points

At a minimum, points need to have their velocities defined, as the acceleration can be calculated by taking the time derivative of the velocity in the same frame. If the 1 point or 2 point theorems were used to compute the velocity, the time derivative of the velocity expression will most likely be more complex than if you were to use the acceleration level 1 point and 2 point theorems. Using the acceleration level methods can result in shorter expressions at this point, which will result in shorter expressions later (such as when forming Kane's equations).

## Advanced Interfaces

### Advanced Functionality

Remember that the Kane object supports bodies which have time-varying masses and inertias, although this functionality isn't completely compatible with the linearization method.

Operators were discussed earlier as a potential way to do mathematical operations on Vector and Dyad objects. The majority of the code in this module is actually coded with them, as it can (subjectively) result in cleaner, shorter, more readable code. If using this interface in your code, remember to take care and use parentheses; the default order of operations in Python results in addition occurring before some of the vector products, so use parentheses liberally.

### Future Features

This will cover the planned features to be added to this submodule.

### Code Output

A function for generating code output for numerical integration is the highest priority feature to implement next. There are a number of considerations here.

Code output for C (using the GSL libraries), Fortran 90 (using LSODA), MATLAB, and SciPy is the goal. Things to be considered include: use of cse on large expressions for MATLAB and SciPy, which are interpretive. It is currently unclear whether compiled languages will benefit from common subexpression elimination, especially considering that it is a common part of compiler optimization, and there can be a significant time penalty when calling cse.

Care needs to be taken when constructing the strings for these expressions, as well as handling of input parameters, and other dynamic symbols. How to deal with output quantities when integrating also needs to be decided, with the potential for multiple options being considered.

## References for Physics/Mechanics

### Autolev Parser

#### Introduction

Autolev (now superseded by MotionGenesis) is a domain specific language used for symbolic multibody dynamics. The SymPy mechanics module now has enough power and functionality to be a fully featured symbolic dynamics module. This parser parses Autolev (version 4.1) code to SymPy code by making use of SymPy's math libraries and the mechanics module.

The parser has been built using the [ANTLR](#) framework and its main purpose is to help former users of Autolev to get familiarized with multibody dynamics in SymPy.

The sections below shall discuss details of the parser like usage, gotchas, issues and future improvements. For a detailed comparison of Autolev and SymPy Mechanics you might want to look at the [SymPy Mechanics for Autolev Users guide](#) (page 1734).

## Usage

We first start with an Autolev code file.

Let us take this example (Comments % have been included to show the Autolev responses):

```
% double_pendulum.al
%-----
MOTIONVARIABLES' Q{2}', U{2}'
CONSTANTS L,M,G
NEWTONIAN N
FRAMES A,B
SIMPROT(N, A, 3, Q1)
% -> N_A = [COS(Q1), -SIN(Q1), 0; SIN(Q1), COS(Q1), 0; 0, 0, 1]
SIMPROT(N, B, 3, Q2)
% -> N_B = [COS(Q2), -SIN(Q2), 0; SIN(Q2), COS(Q2), 0; 0, 0, 1]
W_A_N>=U1*N3>
% -> W_A_N> = U1*N3>
W_B_N>=U2*N3>
% -> W_B_N> = U2*N3>
POINT O
PARTICLES P,R
P_O_P> = L*A1>
% -> P_O_P> = L*A1>
P_P_R> = L*B1>
% -> P_P_R> = L*B1>
V_O_N> = 0>
% -> V_O_N> = 0>
V2PTS(N, A, O, P)
% -> V_P_N> = L*U1*A2>
V2PTS(N, B, P, R)
% -> V_R_N> = L*U1*A2> + L*U2*B2>
MASS P=M, R=M
Q1' = U1
Q2' = U2
GRAVITY(G*N1>)
% -> FORCE_P> = G*M*N1>
% -> FORCE_R> = G*M*N1>
ZERO = FR() + FRSTAR()
% -> ZERO[1] = -L*M*(2*G*SIN(Q1)+L*(U2^2*SIN(Q1-Q2)+2*U1'+COS(Q1-Q2)*U2'))
% -> ZERO[2] = -L*M*(G*SIN(Q2)-L*(U1^2*SIN(Q1-Q2)-U2'-COS(Q1-Q2)*U1'))
KANE()
INPUT M=1,G=9.81,L=1
INPUT Q1=.1,Q2=.2,U1=0,U2=0
INPUT TFINAL=10, INTEGSTEP=.01
CODE DYNAMICS() some_filename.c
```

The parser can be used as follows:

```
>>> from sympy.parsing.autolev import parse_autolev
>>> sympy_code = parse_autolev(open('double_pendulum.al'), include_
 ↳ numeric=True)

The include_pydy flag is False by default. Setting it to True will
```

(continues on next page)

(continued from previous page)

```
enable PyDy simulation code to be outputted if applicable.

>>> print(sympy_code)
import sympy.physics.mechanics as me
import sympy as sm
import math as m
import numpy as np

q1, q2, u1, u2 = me.dynamicsymbols('q1 q2 u1 u2')
q1d, q2d, u1d, u2d = me.dynamicsymbols('q1 q2 u1 u2', 1)
l, m, g=sm.symbols('l m g', real=True)
frame_n=me.ReferenceFrame('n')
frame_a=me.ReferenceFrame('a')
frame_b=me.ReferenceFrame('b')
frame_a.orient(frame_n, 'Axis', [q1, frame_n.z])
print(frame_n.dcm(frame_a))
frame_b.orient(frame_n, 'Axis', [q2, frame_n.z])
print(frame_n.dcm(frame_b))
frame_a.set_ang_vel(frame_n, u1*frame_n.z)
print(frame_a.ang_vel_in(frame_n))
frame_b.set_ang_vel(frame_n, u2*frame_n.z)
print(frame_b.ang_vel_in(frame_n))
point_o=me.Point('o')
particle_p=me.Particle('p', me.Point('p_pt'), sm.Symbol('m'))
particle_r=me.Particle('r', me.Point('r_pt'), sm.Symbol('m'))
particle_p.point.set_pos(point_o, l*frame_a.x)
print(particle_p.point.pos_from(point_o))
particle_r.point.set_pos(particle_p.point, l*frame_b.x)
print(particle_p.point.pos_from(particle_r.point))
point_o.set_vel(frame_n, 0)
print(point_o.vel(frame_n))
particle_p.point.v2pt_theory(point_o, frame_n, frame_a)
print(particle_p.point.vel(frame_n))
particle_r.point.v2pt_theory(particle_p.point, frame_n, frame_b)
print(particle_r.point.vel(frame_n))
particle_p.mass = m
particle_r.mass = m
force_p = particle_p.mass*(g*frame_n.x)
print(force_p)
force_r = particle_r.mass*(g*frame_n.x)
print(force_r)
kd_eqs = [q1d - u1, q2d - u2]
forceList = [(particle_p.point, particle_p.mass*(g*frame_n.x)), (particle_r.
 ↳ point, particle_r.mass*(g*frame_n.x))]
kane = me.KanesMethod(frame_n, q_ind=[q1,q2], u_ind=[u1, u2], kd_eqs = kd_eqs)
fr, frstar = kane.kanes_equations([particle_p, particle_r], forceList)
zero = fr+frstar
print(zero)
#-----PyDy code for integration-----
from pydy.system import System
sys = System(kane, constants = {l:1, m:1, g:9.81},
 specifieds={},
```

(continues on next page)

(continued from previous page)

```
initial_conditions={q1:.1, q2:.2, u1:0, u2:0},
times = np.linspace(0.0, 10, 10/.01))

y=sys.integrate()
```

The commented code is not part of the output code. The print statements demonstrate how to get responses similar to the ones in the Autolev file. Note that we need to use SymPy functions like `.ang_vel_in()`, `.dcm()` etc in many cases unlike directly printing out the variables like zero. If you are completely new to SymPy mechanics, the [SymPy Mechanics for Autolev Users guide](#) (page 1734) guide should help. You might also have to use basic SymPy simplifications and manipulations like `trigsimp()`, `expand()`, `evalf()` etc for getting outputs similar to Autolev. Refer to the [SymPy Tutorial](#) to know more about these.

## Gotchas

- Don't use variable names that conflict with Python's reserved words. This is one example where this is violated:

```
%Autolev Code
%-----
LAMBDA = EIG(M)
```

```
#SymPy Code
#-----
lambda = sm.Matrix([i.evalf() for i in (m).eigenvals().keys()])
```

- Make sure that the names of vectors and scalars are different. Autolev treats these differently but these will get overwritten in Python. The parser currently allows the names of bodies and scalars/vectors to coincide but doesn't do this between scalars and vectors. This should probably be changed in the future.

```
%Autolev Code
%-----
VARIABLES X,Y
FRAMES A
A> = X*A1> + Y*A2>
A = X+Y
```

```
#SymPy Code
#-----
x, y = me.dynamicsymbols('x y')
frame_a = me.ReferenceFrame('a')
a = x*frame_a.x + y*frame_a.y
a = x + y
Note how frame_a is named differently so it doesn't cause a problem.
On the other hand, 'a' gets rewritten from a scalar to a vector.
This should be changed in the future.
```

- When dealing with Matrices returned by functions, one must check the order of the values as they may not be the same as in Autolev. This is especially the case for eigenvalues and eigenvectors.

```
%Autolev Code
%-----
EIG(M, E1, E2)
% -> [5; 14; 13]
E2ROW = ROWS(E2, 1)
EIGVEC> = VECTOR(A, E2ROW)
```

```
#SymPy Code
#-----
e1 = sm.Matrix([i.evalf() for i in m.eigenvals().keys()])
sm.Matrix([5;13;14]) different order
e2 = sm.Matrix([i[2][0].evalf() for i in m.eigenvecs()]).reshape(m.
 ↳shape[0], m.shape[1])
e2row = e2.row(0)
This result depends on the order of the vectors in the eigenvecs.
eigenvector = e2row[0]*a.x + e2row[1]*a.y + e2row[2]*a.y
```

- 
- When using EVALUATE, use something like `90*UNITS(deg,rad)` for angle substitutions as radians are the default in SymPy. You could also add `np.deg2rad()` directly in the SymPy code.

This need not be done for the output code (generated on parsing the CODE commands) as the parser takes care of this when deg units are given in the INPUT declarations.

The DEGREES setting, on the other hand, works only in some cases like in SIMPROT where an angle is expected.

```
%Autolev Code
%-----
A> = Q1*A1> + Q2*A2>
B> = EVALUATE(A>, Q1:30*UNITS(DEG,RAD))
```

```
#SymPy Code
#-----
a = q1*a.frame_a.x + q2*frame_a.y
b = a.subs({q1:30*0.0174533})
b = a.subs({q1:np.deg2rad(30)})
```

- 
- Most of the Autolev settings have not been parsed and have no effect on the parser. The only ones that work somewhat are COMPLEX and DEGREES. It is advised to look into alternatives to these in SymPy and Python.

- 
- The REPRESENT command is not supported. Use the MATRIX, VECTOR or DYADIC commands instead. Autolev 4.1 suggests these over REPRESENT as well while still allowing it but the parser doesn't parse it.
-



- Do not use variables declarations of the type  $W0\{3\}RD\{2,4\}$ . The parser can only handle one variable name followed by one pair of curly braces and any number of ' ' s. You would have to declare all the cases manually if you want to achieve something like  $W0\{3\}RD\{2,4\}$ .

- The parser can handle normal versions of most commands but it may not parse functions with Matrix arguments properly in most cases. Eg:

```
M=COEF([E1;E2],[U1,U2,U3])
```

This would compute the coefficients of U1, U2 and U3 in E1 and E2. It is preferable to manually construct a Matrix using the regular versions of these commands.

```
%Autolev Code
%-----
% COEF([E1;E2],[U1,U2,U3])
M = [COEF(E1,U1),COEF(E1,U2),COEF(E1,U3) &
 ;COEF(E2,U1),COEF(E2,U2),COEF(E2,U3)]
```

- MOTIONVARIABLE declarations must be used for the generalized coordinates and speeds and all other variables must be declared in regular VARIABLE declarations. The parser requires this to distinguish between them to pass the correct parameters to the Kane's method object.

It is also preferred to always declare the speeds corresponding to the coordinates and to pass in the kinematic differential equations. The parser is able to handle some cases where this isn't the case by introducing some dummy variables of its own but SymPy on its own does require them.

Also note that older Autolev declarations like `VARIABLES U{3}'` are not supported either.

```
%Autolev Code
%-----
MOTIONVARIABLES' Q{2}', U{2}'
% ----- OTHER LINES -----
Q1' = U1
Q2' = U2
----- OTHER LINES -----
ZERO = FR() + FRSTAR()
```

```
#SymPy Code
#-----
q1, q2, u1, u2 = me.dynamicsymbols('q1 q2 u1 u2')
q1d, q2d, u1d, u2d = me.dynamicsymbols('q1 q2 u1 u2', 1)

----- other lines -----

kd_eqs = [q1d - u1, q2d - u2]
kane = me.KanesMethod(frame_n, q_ind=[q1,q2], u_ind=[u1, u2], kd_eqs = ukd_eqs)
fr, frstar = kane.kanes_equations([particle_p, particle_r], forceList)
zero = fr+frstar
```

- Need to change `me.dynamicsymbols._t` to `me.dynamicsymbols('t')` for all occurrences of it in the Kane's equations. For example have a look at line 10 of this [spring damper example](#). This equation is used in forming the Kane's equations so we need to change `me.dynamicsymbols._t` to `me.dynamicsymbols('t')` in this case.

The main reason that this needs to be done is because PyDy requires time dependent specifieds to be explicitly laid out while Autolev simply takes care of the stray time variables in the equations by itself.

The problem is that PyDy's System class does not accept `dynamicsymbols._t` as a specified. Refer to issue [#396](#). This change is not actually ideal so a better solution should be figured out in the future.

---

- The parser creates SymPy symbols and dynamicsymbols by parsing variable declarations in the Autolev Code.

For intermediate expressions which are directly initialized the parser does not create SymPy symbols. It just assigns them to the expression.

On the other hand, when a declared variable is assigned to an expression, the parser stores the expression against the variable in a dictionary so as to not reassign it to a completely different entity. This constraint is due to the inherent nature of Python and how it differs from a language like Autolev.

Also, Autolev seems to be able to assume whether to use a variable or the rhs expression that variable has been assigned to in equations even without an explicit `RHS()` call in some cases. For the parser to work correctly however, it is better to use `RHS()` wherever a variable's rhs expression is meant to be used.

```
%Autolev Code
%-----
VARIABLES X, Y
E = X + Y
X = 2*Y

RHS_X = RHS(X)

I1 = X
I2 = Y
I3 = X + Y

INERTIA B,I1,I2,I3
% -> I_B_B0>> = I1*B1>*B1> + I2*B2>*B2> + I3*B3>*B3>
```

```
#SymPy Code
#-----
x,y = me.dynamicsymbols('x y')
e = x + y # No symbol is made out of 'e'

an entry like {x:2*y} is stored in an rhs dictionary

rhs_x = 2*y
```

(continues on next page)

(continued from previous page)

```

i1 = x # again these are not made into SymPy symbols
i2 = y
i3 = x + y

body_b.inertia = (me.inertia(body_b_f, i1, i2, i3), b_cm)
This prints as:
x*b_f.x*b_f.x + y*b_f.y*b_f.y + (x+y)*b_f.z*b_f.z
while Autolev's output has I1,I2 and I3 in it.
Autolev however seems to know when to use the RHS of I1,I2 and I3
based on the context.

```

- This is how the SOLVE command is parsed:

```

%Autolev Code
%-----
SOLVE(ZERO,X,Y)
A = RHS(X)*2 + RHS(Y)

```

```

#SymPy Code
#-----
print(sm.solve(zero,x,y))
Behind the scenes the rhs of x
is set to sm.solve(zero,x,y)[x].
a = sm.solve(zero,x,y)[x]*2 + sm.solve(zero,x,y)[y]

```

The indexing like [x] and [y] doesn't always work so you might want to look at the underlying dictionary that solve returns and index it correctly.

- Inertia declarations and Inertia functions work somewhat differently in the context of the parser. This might be hard to understand at first but this had to be done to bridge the gap due to the differences in SymPy and Autolev. Here are some points about them:
  1. Inertia declarations (INERTIA B,I1,I2,I3) set the inertias of rigid bodies.
  2. Inertia setters of the form I\_C\_D>> = expr however, set the inertias only when C is a body. If C is a particle then I\_C\_D>> = expr simply parses to i\_c\_d = expr and i\_c\_d acts like a regular variable.
  3. When it comes to inertia getters (I\_C\_D>> used in an expression or INERTIA commands), these MUST be used with the EXPRESS command to specify the frame as SymPy needs this information to compute the inertia dyadic.

```

%Autolev Code
%-----
INERTIA B,I1,I2,I3
I_B_B0>> = X*A1>*A1> + Y*A2>*A2> % Parser will set the inertia of B
I_P_Q>> = X*A1>*A1> + Y^2*A2>*A2> % Parser just parses it as i_p_q = expr

E1 = 2*EXPRESS(I_B_0>>,A)
E2 = I_P_Q>>

```

(continues on next page)

(continued from previous page)

```
E3 = EXPRESS(I_P_0>>,A)
E4 = EXPRESS(INERTIA(0),A)

% In E1 we are using the EXPRESS command with I_B_0>> which makes
% the parser and SymPy compute the inertia of Body B about point 0.

% In E2 we are just using the dyadic object I_P_Q>> (as I_P_Q>> = expr
% doesn't act as a setter) defined above and not asking the parser
% or SymPy to compute anything.

% E3 asks the parser to compute the inertia of P about point 0.

% E4 asks the parser to compute the inertias of all bodies wrt about 0.
```

- 
- In an inertia declaration of a body, if the inertia is being set about a point other than the center of mass, one needs to make sure that the position vector setter for that point and the center of mass appears before the inertia declaration as SymPy will throw an error otherwise.

```
%Autolev Code
%------
P_S0_0> = X*A1>
INERTIA S_(0) I1,I2,I3
```

- 
- Note that all Autolev commands have not been implemented. The parser now covers the important ones in their basic forms. If you are doubtful whether a command is included or not, please have a look at [this file](#) in the source code. Search for “<command>” to verify this. Looking at the code for the specific command will also give an idea about what form it is expected to work in.

## Limitations and Issues

- A lot of the issues have already been discussed in the Gotchas section. Some of these are:
  - Vector names coinciding with scalar names are overwritten in Python.
  - Some convenient variable declarations aren’t parsed.
  - Some convenient forms of functions to return matrices aren’t parsed.
  - Settings aren’t parsed.
  - symbols and rhs expressions work very differently in Python which might cause undesirable results.
  - Dictionary indexing for the parsed code of the SOLVE command is not proper in many cases.
  - Need to change `dynamicsymbols._t` to `dynamicsymbols('t')` for the PyDy simulation code to work properly.

Here are some other ones:

- Eigenvectors do not seem to work as expected. The values in Autolev and SymPy are not the same in many cases.
- Block matrices aren't parsed by the parser. It would actually be easier to make a change in SymPy to allow matrices to accept other matrices for arguments.
- The SymPy equivalent of the TAYLOR command `.series()` does not work with `dynamicsymbols()`.
- Only DEPENDENT constraints are currently parsed. Need to parse AUXILIARY constraints as well. This should be done soon as it isn't very difficult.
- None of the energy and momentum functions are parsed right now. It would be nice to get these working as well. Some changes should probably be made to SymPy. For instance, SymPy doesn't have a function equivalent to `NICHECK()`.
- The numerical integration parts work properly only in the case of the KANE command with no arguments. Things like `KANE(F1,F2)` do not currently work.
- Also, the PyDy numerical simulation code works only for cases where the matrix say `ZERO = FR() + FRSTAR()` is solved for. It doesn't work well when the matrix has some other equations plugged in as well. One hurdle faced in achieving this was that PyDy's `System` class automatically takes in the `forcing_full` and `mass_matrix_full` and solves them without giving the user the flexibility to specify the equations. It would be nice to add this functionality to the `System` class.

## Future Improvements

### 1. Completing Dynamics Online

The parser has been built by referring to and parsing codes from the [Autolev Tutorial](#) and the book *Dynamics Online: Theory and Implementation Using Autolev*. Basically, the process involved going through each of these codes, validating the parser results and improving the rules if required to make sure the codes parsed well.

The parsed codes of these are available on GitLab [here](#). The repo is private so access needs to be requested. As of now, most codes till Chapter 4 of *Dynamics Online* have been parsed.

Completing all the remaining codes of the book (namely, 2-10, 2-11, rest of Ch4, Ch5 and Ch6 (less important) ) would make the parser more complete.

### 2. Fixing Issues

The second thing to do would be to go about fixing the problems described above in the [Gotchas](#) (page 1727) and [Limitations and Issues](#) (page 1732) sections in order of priority and ease. Many of these require changes in the parser code while some of these are better fixed by adding some functionality to SymPy.

### 3. Switching to an AST

The parser is currently built using a kind of Concrete Syntax Tree (CST) using the [ANTLR](#) framework. It would be ideal to switch from a CST to an Abstract Syntax Tree (AST). This way, the parser code will be independent of the ANTLR grammar which makes it a lot more flexible. It would also be easier to make changes to the grammar and the rules of the parser.

## SymPy Mechanics for Autolev Users

### Introduction

Autolev (now superseded by MotionGenesis) is a domain specific programming language which is used for symbolic multibody dynamics. The SymPy mechanics module now has enough power and functionality to be a fully featured symbolic dynamics module. The PyDy package extends the SymPy output to the numerical domain for simulation, analyses and visualization. Autolev and SymPy Mechanics have a lot in common but there are also many differences between them. This page shall expand upon their differences. It is meant to be a go-to reference for Autolev users who want to transition to SymPy Mechanics.

It would be nice to have a basic understanding of SymPy and SymPy Mechanics before going over this page. If you are completely new to Python, you can check out the official [Python Tutorial](#). Check out the *SymPy Documentation* (page ??), especially the tutorial to get a feel for SymPy. For an introduction to Multibody dynamics in Python, [this](#) lecture is very helpful.

You might also find the [Autolev Parser](#) (page 1724) which is a part of SymPy to be helpful.

## Some Key Differences

| Autolev                                                                                                                                                                                                                                                                                                      | SymPy Mechanics                                                                                                                                                                                                                                                                                                                                                                                                          |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Autolev is a domain specific programming language designed to perform multibody dynamics. Since it is a language of its own, it has a very rigid language specification. It predefines, assumes and computes many things based on the input code. Its code is a lot cleaner and concise as a result of this. | SymPy is a library written in the general purpose language Python. Although Autolev's code is more compact, SymPy (by virtue of being an add on to Python) is more flexible. The users have more control over what they can do. For example, one can create a class in their code for let's say a type of rigibodies with common properties. The wide array of scientific Python libraries available is also a big plus. |
| Autolev generates Matlab, C, or Fortran code from a small set of symbolic mathematics.                                                                                                                                                                                                                       | SymPy generates numerical Python, C or Octave/Matlab code from a large set of symbolic mathematics created with SymPy. It also builds on the popular scientific Python stack such as NumPy, SciPy, IPython, matplotlib, Cython and Theano.                                                                                                                                                                               |
| Autolev uses 1 (one) based indexing. The initial element of a sequence is found using <code>a[1]</code> .                                                                                                                                                                                                    | Python uses 0 (zero) based indexing. The initial element of a sequence is found using <code>a[0]</code> .                                                                                                                                                                                                                                                                                                                |
| Autolev is case insensitive.                                                                                                                                                                                                                                                                                 | SymPy code being Python code is case sensitive.                                                                                                                                                                                                                                                                                                                                                                          |
| One can define their own commands in Autolev by making <code>.R</code> and <code>.A</code> files which can be used in their programs.                                                                                                                                                                        | SymPy code is Python code, so one can define functions in their code. This is a lot more convenient.                                                                                                                                                                                                                                                                                                                     |
| Autolev is proprietary.                                                                                                                                                                                                                                                                                      | SymPy is open source.                                                                                                                                                                                                                                                                                                                                                                                                    |

## Rough Autolev-SymPy Equivalents

The tables below give rough equivalents for some common Autolev expressions. **These are not exact equivalents**, but rather should be taken as hints to get you going in the right direction. For more detail read the built-in documentation on [SymPy vectors](#) (page 1592), [SymPy mechanics](#) (page 1675) and [PyDy](#).

In the tables below, it is assumed that you have executed the following commands in Python:

```
import sympy.physics.mechanics as me
import sympy as sm
```



## Mathematical Equivalents

| Autolev               | SymPy                                                                                                                                  | Notes                                                                                                                                                                                                      |
|-----------------------|----------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Constants A, B        | <code>a, b = sm.symbols('a b', real=True)</code>                                                                                       | Note that the names of the symbols can be different from the names of the variables they are assigned to. We can define <code>a, b = symbols('b a')</code> but its good practice to follow the convention. |
| Constants C+          | <code>c = sm.symbols('c', real=True, nonnegative=True)</code>                                                                          | Refer to SymPy <a href="#">assumptions</a> (page 186) for more information.                                                                                                                                |
| Constants D-          | <code>d = sm.symbols('d', real=True, nonpositive=True)</code>                                                                          |                                                                                                                                                                                                            |
| Constants K{4}        | <code>k1, k2, k3, k4 = sm.symbols('k1 k2 k3 k4', real=True)</code>                                                                     |                                                                                                                                                                                                            |
| Constants a{2:4}      | <code>a2, a3, a4 = sm.symbols('a2 a3 a4', real=True)</code>                                                                            |                                                                                                                                                                                                            |
| Constants b{1:2, 1:2} | <code>b11, b12, b21, b22 = sm.symbols('b11 b12 b21 b22', real=True)</code>                                                             |                                                                                                                                                                                                            |
| Specified Phi         | <code>phi = me.dynamicsymbols('phi')</code>                                                                                            |                                                                                                                                                                                                            |
| Variables q, s        | <code>q, s = me.dynamicsymbols(q, s)</code>                                                                                            |                                                                                                                                                                                                            |
| Variables x''         | <code>x = me.dynamicsymbols('x' )</code><br><code>xd = me.dynamicsymbols('x' , 1)</code><br><code>xd2 = me.dynamicsymbols('x' ,</code> |                                                                                                                                                                                                            |
| <b>5.8. Topics</b>    |                                                                                                                                        | <b>1737</b>                                                                                                                                                                                                |

## Physical Equivalents

| Autolev                                                                                         | SymPy                                                                                                                                                                                                                                                    | Notes                                                                                                                                                                                                                                                                                                                          |
|-------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Bodies A<br>Declares A, its masscenter Ao, and orthonormal vectors A1>, A2> and A3> fixed in A. | <pre>m = sm.symbols('m') Ao = sm.symbols('Ao') Af = me. ReferenceFrame('Af' ) I = me.outer(Af.x,Af.x) P = me.Point('P') A =me.RigidBody('A', Ao, Af, m, (I, P)) Af.x, Af.y and Af.z are equiv- alent to A1&gt;, A2&gt; and A3&gt;.</pre>                 | The 4th and 5th arguments are for the mass and inertia. These are specified after the declaration in Autolev. One can pass a dummy for the parameters and use setters <code>A.mass = \_</code> and <code>A.inertia = \_</code> to set them later. For more information refer to <a href="#">mechanics/masses</a> . (page 1676) |
| Frames A<br>$V1> = X1*A1> + X2*A2>$                                                             | <pre>A = me. ReferenceFrame('A' ) v1 = x1*A.x + x2*A.y</pre>                                                                                                                                                                                             | For more information refer to <a href="#">physics/vectors</a> . (page 51)                                                                                                                                                                                                                                                      |
| Newtonian N                                                                                     | <pre>N = me. ReferenceFrame('N' )</pre>                                                                                                                                                                                                                  | SymPy doesn't specify that a frame is inertial during declaration. Many functions such as <code>set_ang_vel()</code> take the inertial reference frame as a parameter.                                                                                                                                                         |
| Particles C                                                                                     | <pre>m = sm.symbols('m') Po = me.Point('Po') C = me.Particle('C', Po, m)</pre>                                                                                                                                                                           | The 2nd and 3rd arguments are for the point and mass. In Autolev, these are specified after the declaration. One can pass a dummy and use setters ( <code>A.point = \_</code> and <code>A.mass = \_</code> ) to set them later.                                                                                                |
| Points P, Q                                                                                     | <pre>P = me.Point('P') Q = me.Point('Q')</pre>                                                                                                                                                                                                           |                                                                                                                                                                                                                                                                                                                                |
| Mass B=mB                                                                                       | <pre>mB = symbols('mB') B.mass = mB</pre>                                                                                                                                                                                                                |                                                                                                                                                                                                                                                                                                                                |
| Inertia B, I1, I2, I3, I12, I23, I31                                                            | <pre>I = me.inertia(Bf, i1, i2, i3, i12, i23, i31) B.inertia = (I, P) where B is a rigidbody, Bf is the re- lated frame and P is the cen- ter of mass of B. Inertia dyadics can also be formed using vector outer products. I = me.outer(N.x, N.x)</pre> | For more information refer to the <a href="#">mechanics api</a> . (page 1743)                                                                                                                                                                                                                                                  |

continues on next page

Table 1 – continued from previous page

| Autolev                                                                                                                                                                                                                                                                                                                                        | SymPy                                                                                                                                                                                                                                                                                                                                               | Notes                                                                                                                                                                                                                     |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\text{vec} > = \mathbf{P\_0\_Q} > / L$<br>$\text{vec} > = u_1 * \mathbf{N_1} > + u_2 * \mathbf{N_2} >$<br>$\text{Cross}(a >, b >)$<br>$\text{Dot}(a >, b >)$<br>$\text{Mag}(v >)$<br>$\text{Unitvec}(v >)$<br>$\text{DYAD} > > = 3 * \mathbf{A_1} > * \mathbf{A_1} > + \mathbf{A_2} > * \mathbf{A_2} > + 2 * \mathbf{A_3} > * \mathbf{A_3} >$ | $\text{vec} = (\mathbf{Q_0}.\text{pos\_from}(0)) / L$<br>$\text{vec} = u_1 * \mathbf{N.x} + u_2 * \mathbf{N.y}$<br>$\text{cross}(a, b)$<br>$\text{dot}(a, b)$<br>$v.\text{magnitude}()$<br>$v.\text{normalize}()$<br>$\text{dyad} = 3 * \text{me}.\text{outer}(a.x, a.x) + \text{me}.\text{outer}(a.y, a.y) + 2 * \text{me}.\text{outer}(a.z, a.z)$ | For more information refer to <a href="#">physics/vectors</a> . (page 1592)                                                                                                                                               |
| $\mathbf{P\_0\_Q} > = L * \mathbf{A_1} >$<br>$\mathbf{P\_P\_Q} > = L * \mathbf{A_1} >$                                                                                                                                                                                                                                                         | $\mathbf{Q}.\text{point} = 0.$<br>$\text{locatenew}('Q_0', L * \mathbf{A.x})$<br>where A is a reference frame.<br>$\mathbf{Q}.\text{point} = \mathbf{P}.\text{point}.$<br>$\text{locatenew}('Q_0', L * \mathbf{A.x})$                                                                                                                               | For more information refer to the <a href="#">kinematics api</a> . (page 1652)<br>All these vector and kinematic functions are to be used on Point objects and not Particle objects so .point must be used for particles. |
| $\mathbf{V\_0\_N} > = u_3 * \mathbf{N.1} > + u_4 * \mathbf{N.2} >$<br>$\text{Partials}(\mathbf{V\_0\_N} >, u_3)$                                                                                                                                                                                                                               | $\mathbf{0}.\text{set\_vel}(\mathbf{N}, u_1 * \mathbf{N.x} + u_2 * \mathbf{N.y})$<br>$\mathbf{0}.\text{partial\_velocity}(\mathbf{N}, u_3)$                                                                                                                                                                                                         | The getter would be $\mathbf{0}.\text{vel}(\mathbf{N})$ .                                                                                                                                                                 |
| $\mathbf{A\_0\_N} > = \mathbf{0} >$<br>Acceleration of point O in reference frame N.                                                                                                                                                                                                                                                           | $\mathbf{0}.\text{set\_acc}(\mathbf{N}, \mathbf{0})$                                                                                                                                                                                                                                                                                                | The getter would be $\mathbf{0}.\text{acc}(\mathbf{N})$ .                                                                                                                                                                 |
| $\mathbf{W\_B\_N} > = \mathbf{qB'} * \mathbf{B_3} >$<br>Angular velocity of body B in reference frame F.                                                                                                                                                                                                                                       | $\mathbf{B}.\text{set\_ang\_vel}(\mathbf{N}, \mathbf{qBd} * \mathbf{Bf.z})$<br>where Bf is the frame associated with the body B.                                                                                                                                                                                                                    | The getter would be $\mathbf{B}.\text{ang\_vel\_in}(\mathbf{N})$ .                                                                                                                                                        |
| $\mathbf{ALF\_B\_N} > = \text{Dt}(\mathbf{W\_B\_N} >, \mathbf{N})$<br>Angular acceleration of body B in reference frame N.                                                                                                                                                                                                                     | $\mathbf{B}.\text{set\_ang\_acc}(\mathbf{N}, \text{diff}(\mathbf{B}.\text{ang\_vel\_in}(\mathbf{N})))$                                                                                                                                                                                                                                              | The getter would be $\mathbf{B}.\text{ang\_acc\_in}(\mathbf{N})$ .                                                                                                                                                        |
| $\text{Force\_0} > = \mathbf{F_1} * \mathbf{N_1} > + \mathbf{F_2} * \mathbf{N_2} >$<br>$\text{Torque\_A} > = -c * \mathbf{qA'} * \mathbf{A_3} >$                                                                                                                                                                                               | In SymPy one should have a list which contains all the forces and torques.<br>$\text{fL.append}((0, \mathbf{f_1} * \mathbf{N.x} + \mathbf{f_2} * \mathbf{N.y}))$<br>where fL is the force list.<br>$\text{fL.append}((\mathbf{A}, -c * \mathbf{qAd} * \mathbf{A.z}))$                                                                               |                                                                                                                                                                                                                           |
| $\mathbf{A\_B} = \mathbf{M}$ where M is a matrix and A, B are frames.<br>$\mathbf{D} = \mathbf{A\_B} * 2 + 1$                                                                                                                                                                                                                                  | $\mathbf{B}.\text{orient}(\mathbf{A}, 'DCM', \mathbf{M})$<br>where M is a SymPy Matrix.<br>$\mathbf{D} = \mathbf{A}.\text{dcm}(\mathbf{B}) * 2 + 1$                                                                                                                                                                                                 |                                                                                                                                                                                                                           |
| $\text{CM}(\mathbf{B})$                                                                                                                                                                                                                                                                                                                        | $\mathbf{B}.\text{masscenter}$                                                                                                                                                                                                                                                                                                                      |                                                                                                                                                                                                                           |
| $\text{Mass}(\mathbf{A}, \mathbf{B}, \mathbf{C})$                                                                                                                                                                                                                                                                                              | $\mathbf{A}.\text{mass} + \mathbf{B}.\text{mass} + \mathbf{C}.\text{mass}$                                                                                                                                                                                                                                                                          |                                                                                                                                                                                                                           |

continues on next page

Table 1 – continued from previous page

| Autolev              | SymPy                                                                                                                                                                                         | Notes                                                                                                                                                                         |
|----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| V1pt(A,B,P,Q)        | Q.v1pt_theory(P, A, B)                                                                                                                                                                        | P and Q are assumed to be Point objects here. Remember to use .point for particles.                                                                                           |
| V2pts(A,B,P,Q)       | Q.v2pt_theory(P, A, B)                                                                                                                                                                        |                                                                                                                                                                               |
| A1pt(A,B,P,Q)        | Q.alpt_theory(P, A, B)                                                                                                                                                                        |                                                                                                                                                                               |
| A2pts(A,B,P,Q)       | Q.a2pt_theory(P, A, B)                                                                                                                                                                        |                                                                                                                                                                               |
| Angvel(A,B)          | B.ang_vel_in(A)                                                                                                                                                                               |                                                                                                                                                                               |
| Simprot(A, B, 1, qA) | B.orient(A, 'Axis', qA, A.x)                                                                                                                                                                  |                                                                                                                                                                               |
| Gravity(G*N1>)       | fL.extend(gravity( g*N.x, P1, P2, ...))                                                                                                                                                       | In SymPy we must use a forceList (here fL) which contains tuples of the form (point, force_vector). This is passed to the kanes_equations() method of the KanesMethod object. |
| CM(O,P1,R)           | me.functions.<br>center_of_mass(o, p1, r)                                                                                                                                                     |                                                                                                                                                                               |
| Force(P/Q, v>)       | fL.append((P, -1*v),<br>(Q, v))                                                                                                                                                               |                                                                                                                                                                               |
| Torque(A/B, v>)      | fL.append((A, -1*v),<br>(B, v))                                                                                                                                                               |                                                                                                                                                                               |
| Kindiffs(A, B ...)   | KM.kindiffdict()                                                                                                                                                                              |                                                                                                                                                                               |
| Momentum(option)     | linear_momentum(N, B1,<br>B2 ...)<br>reference frame followed by<br>one or more bodies<br>angular_momentum(O, N,<br>B1, B2 ...)<br>point, reference frame fol-<br>lowed by one or more bodies |                                                                                                                                                                               |
| KE()                 | kinetic_energy(N, B1,<br>B2 ...)<br>reference frame followed by<br>one or more bodies                                                                                                         |                                                                                                                                                                               |
| Constrain(...)       | velocity_constraints =<br>[...]<br>u_dependent = [...]<br>u_auxiliary = [...]<br>These lists are passed to the<br>KanesMethod object.                                                         | For more details refer to <a href="#">mechanics/kane</a> (page 1682) and the <a href="#">kane api</a> . (page 1764)                                                           |

continues on next page

Table 1 – continued from previous page

| Autolev       | SymPy                                                                                                                                                                                                                                                                                                                                                                           | Notes                                                                                                               |
|---------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------|
| Fr() FrStar() | <p>KM = KanesMethod(f, q_ind, u_ind, kd_eqs, q_dependent, configuration_constraints, u_dependent, velocity_constraints, acceleration_constraints, u_auxiliary)</p> <p>The KanesMethod object takes a reference frame followed by multiple lists as arguments.</p> <p>(fr, frstar) = KM.kanes_equations(fL, bL) where fL and bL are lists of forces and bodies respectively.</p> | For more details refer to <a href="#">mechanics/kane</a> (page 1682) and the <a href="#">kane api</a> . (page 1764) |

## Numerical Evaluation and Visualization

Autolev's CODE Option() command allows one to generate Matlab, C, or Fortran code for numerical evaluation and visualization. Option can be Dynamics, ODE, Nonlinear or Algebraic.

Numerical evaluation for dynamics can be achieved using PyDy. One can pass in the KanesMethod object to the System class along with the values for the constants, specifieds, initial conditions and time steps. The equations of motion can then be integrated. The plotting is achieved using matplotlib. Here is an example from the [PyDy Documentation](#) on how it is done:

```
from numpy import array, linspace, sin
from pydy.system import System

sys = System(kane,
 constants = {mass: 1.0, stiffness: 1.0,
 damping: 0.2, gravity: 9.8},
 specifieds = {force: lambda x, t: sin(t)},
 initial_conditions = {position: 0.1, speed: -1.0},
 times = linspace(0.0, 10.0, 1000))

y = sys.integrate()

import matplotlib.pyplot as plt
plt.plot(sys.times, y)
plt.legend((str(position), str(speed)))
plt.show()
```

For information on all the things PyDy can accomplish refer to the [PyDy Documentation](#).

The tools in the PyDy workflow are :

- **SymPy: SymPy is a Python library for** symbolic computation. It provides computer algebra capabilities either as a stan-

alone application, as a library to other applications, or live on the web as SymPy Live or SymPy Gamma.

- **NumPy: NumPy is a library for the**  
Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.
- **SciPy: SciPy is an open source**  
Python library used for scientific computing and technical computing. SciPy contains modules for optimization, linear algebra, integration, interpolation, special functions, FFT, signal and image processing, ODE solvers and other tasks common in science and engineering.
- **IPython: IPython is a command shell**  
for interactive computing in multiple programming languages, originally developed for the Python programming language, that offers introspection, rich media, shell syntax, tab completion, and history.
- **Aesara: Aesara is**  
a numerical computation library for Python. In Aesara, computations are expressed using a NumPy-esque syntax and compiled to run efficiently on either CPU or GPU architectures.
- **Cython: Cython is a superset of the**  
Python programming language, designed to give C-like performance with code that is mostly written in Python. Cython is a compiled language that generates CPython extension modules.
- **matplotlib: matplotlib is a**  
plotting library for the Python programming language and its numerical mathematics extension NumPy.

One will be able to write code equivalent to the Matlab, C or Fortran code generated by Autolev using these scientific computing tools. It is recommended to go over these modules to gain an understanding of scientific computing with Python.

## Links

[SymPy Introductory Tutorial](#) (page 5)

[SymPy Documentation](#) (page ??)

[SymPy Physics Vector Documentation](#) (page 1592)

[SymPy Mechanics Documentation](#) (page 1675)

[PyDy Documentation](#)

[MultiBody Dynamics with Python](#)

## Mechanics API

### Masses, Inertias & Particles, RigidBody (Docstrings)

#### Particle

**class** sympy.physics.mechanics.particle.**Particle**(*name*, *point*, *mass*)

A particle.

#### Parameters

**name** : str

Name of particle

**point** : Point

A physics/mechanics Point which represents the position, velocity, and acceleration of this Particle

**mass** : sympifyable

A SymPy expression representing the Particle's mass

#### Explanation

Particles have a non-zero mass and lack spatial extension; they take up no space.

Values need to be supplied on initialization, but can be changed later.

#### Examples

```
>>> from sympy.physics.mechanics import Particle, Point
>>> from sympy import Symbol
>>> po = Point('po')
>>> m = Symbol('m')
>>> pa = Particle('pa', po, m)
>>> # Or you could change these later
>>> pa.mass = m
>>> pa.point = po
```

**angular\_momentum**(*point*, *frame*)

Angular momentum of the particle about the point.

#### Parameters

**point** : Point

The point about which angular momentum of the particle is desired.

**frame** : ReferenceFrame

The frame in which angular momentum is desired.

### Explanation

The angular momentum  $H$ , about some point  $O$  of a particle,  $P$ , is given by:

$$H = \mathbf{r} \times m * \mathbf{v}$$

where  $\mathbf{r}$  is the position vector from point  $O$  to the particle  $P$ ,  $m$  is the mass of the particle, and  $\mathbf{v}$  is the velocity of the particle in the inertial frame,  $N$ .

### Examples

```
>>> from sympy.physics.mechanics import Particle, Point, _
↳ReferenceFrame
>>> from sympy.physics.mechanics import dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> m, v, r = dynamicsymbols('m v r')
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> A = O.locatenew('A', r * N.x)
>>> P = Particle('P', A, m)
>>> P.point.set_vel(N, v * N.y)
>>> P.angular_momentum(O, N)
m*r*v*N.z
```

#### **kinetic\_energy**(*frame*)

Kinetic energy of the particle.

##### **Parameters**

**frame** : ReferenceFrame

The Particle's velocity is typically defined with respect to an inertial frame but any relevant frame in which the velocity is known can be supplied.

### Explanation

The kinetic energy,  $T$ , of a particle,  $P$ , is given by

$$T = 1/2 m v^2$$

where  $m$  is the mass of particle  $P$ , and  $\mathbf{v}$  is the velocity of the particle in the supplied ReferenceFrame.



## Examples

```
>>> from sympy.physics.mechanics import Particle, Point, ↵
↵ReferenceFrame
>>> from sympy import symbols
>>> m, v, r = symbols('m v r')
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> P = Particle('P', O, m)
>>> P.point.set_vel(N, v * N.y)
>>> P.kinetic_energy(N)
m*v**2/2
```

**linear\_momentum**(*frame*)

Linear momentum of the particle.

### Parameters

**frame** : ReferenceFrame

The frame in which linear momentum is desired.

## Explanation

The linear momentum  $L$ , of a particle  $P$ , with respect to frame  $N$  is given by

$$L = m * v$$

where  $m$  is the mass of the particle, and  $v$  is the velocity of the particle in the frame  $N$ .

## Examples

```
>>> from sympy.physics.mechanics import Particle, Point, ↵
↵ReferenceFrame
>>> from sympy.physics.mechanics import dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> m, v = dynamicsymbols('m v')
>>> N = ReferenceFrame('N')
>>> P = Point('P')
>>> A = Particle('A', P, m)
>>> P.set_vel(N, v * N.x)
>>> A.linear_momentum(N)
m*v*N.x
```

**property mass**

Mass of the particle.

**parallel\_axis**(*point*, *frame*)

Returns an inertia dyadic of the particle with respect to another point and frame.

### Parameters

**point** : sympy.physics.vector.Point

The point to express the inertia dyadic about.

**frame** : sympy.physics.vector.ReferenceFrame

The reference frame used to construct the dyadic.

**Returns**

**inertia** : sympy.physics.vector.Dyadic

The inertia dyadic of the particle expressed about the provided point and frame.

**property point**

Point of the particle.

**property potential\_energy**

The potential energy of the Particle.

### Examples

```
>>> from sympy.physics.mechanics import Particle, Point
>>> from sympy import symbols
>>> m, g, h = symbols('m g h')
>>> O = Point('O')
>>> P = Particle('P', O, m)
>>> P.potential_energy = m * g * h
>>> P.potential_energy
g*h*m
```

## RigidBody

**class** sympy.physics.mechanics.rigidbody.**RigidBody**(*name, masscenter, frame, mass, inertia*)

An idealized rigid body.

### Explanation

This is essentially a container which holds the various components which describe a rigid body: a name, mass, center of mass, reference frame, and inertia.

All of these need to be supplied on creation, but can be changed afterwards.

### Examples

```
>>> from sympy import Symbol
>>> from sympy.physics.mechanics import ReferenceFrame, Point, RigidBody
>>> from sympy.physics.mechanics import outer
>>> m = Symbol('m')
>>> A = ReferenceFrame('A')
>>> P = Point('P')
```

(continues on next page)

(continued from previous page)

```

>>> I = outer (A.x, A.x)
>>> inertia_tuple = (I, P)
>>> B = RigidBody('B', P, A, m, inertia_tuple)
>>> # Or you could change them afterwards
>>> m2 = Symbol('m2')
>>> B.mass = m2

```

## Attributes

|            |                                                                                       |
|------------|---------------------------------------------------------------------------------------|
| name       | (string) The body's name.                                                             |
| masscenter | (Point) The point which represents the center of mass of the rigid body.              |
| frame      | (ReferenceFrame) The ReferenceFrame which the rigid body is fixed in.                 |
| mass       | (Sympifyable) The body's mass.                                                        |
| inertia    | ((Dyadic, Point)) The body's inertia about a point; stored in a tuple as shown above. |

### **angular\_momentum**(point, frame)

Returns the angular momentum of the rigid body about a point in the given frame.

#### **Parameters**

**point** : Point

The point about which angular momentum is desired.

**frame** : ReferenceFrame

The frame in which angular momentum is desired.

## Explanation

The angular momentum  $H$  of a rigid body  $B$  about some point  $O$  in a frame  $N$  is given by:

$$H = I \cdot \omega + r \times Mv$$

where  $I$  is the central inertia dyadic of  $B$ ,  $\omega$  is the angular velocity of body  $B$  in the frame,  $N$ ,  $r$  is the position vector from point  $O$  to the mass center of  $B$ , and  $v$  is the velocity of the mass center in the frame,  $N$ .

## Examples

```

>>> from sympy.physics.mechanics import Point, ReferenceFrame, outer
>>> from sympy.physics.mechanics import RigidBody, dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> M, v, r, omega = dynamicsymbols('M v r omega')
>>> N = ReferenceFrame('N')
>>> b = ReferenceFrame('b')

```

(continues on next page)

(continued from previous page)

```

>>> b.set_ang_vel(N, omega * b.x)
>>> P = Point('P')
>>> P.set_vel(N, 1 * N.x)
>>> I = outer(b.x, b.x)
>>> B = RigidBody('B', P, b, M, (I, P))
>>> B.angular_momentum(P, N)
omega*b.x

```

**property central\_inertia**

The body's central inertia dyadic.

**kinetic\_energy(*frame*)**

Kinetic energy of the rigid body.

**Parameters**

**frame** : ReferenceFrame

The RigidBody's angular velocity and the velocity of it's mass center are typically defined with respect to an inertial frame but any relevant frame in which the velocities are known can be supplied.

**Explanation**

The kinetic energy,  $T$ , of a rigid body,  $B$ , is given by

$$T = \frac{1}{2} (I \omega^2 + m v^2)$$

where  $I$  and  $m$  are the central inertia dyadic and mass of rigid body  $B$ , respectively,  $\omega$  is the body's angular velocity and  $v$  is the velocity of the body's mass center in the supplied ReferenceFrame.

**Examples**

```

>>> from sympy.physics.mechanics import Point, ReferenceFrame, outer
>>> from sympy.physics.mechanics import RigidBody
>>> from sympy import symbols
>>> M, v, r, omega = symbols('M v r omega')
>>> N = ReferenceFrame('N')
>>> b = ReferenceFrame('b')
>>> b.set_ang_vel(N, omega * b.x)
>>> P = Point('P')
>>> P.set_vel(N, v * N.x)
>>> I = outer(b.x, b.x)
>>> inertia_tuple = (I, P)
>>> B = RigidBody('B', P, b, M, inertia_tuple)
>>> B.kinetic_energy(N)
M*v**2/2 + omega**2/2

```

**linear\_momentum(*frame*)**

Linear momentum of the rigid body.

**Parameters**

**frame** : ReferenceFrame

The frame in which linear momentum is desired.

### Explanation

The linear momentum  $L$ , of a rigid body  $B$ , with respect to frame  $N$  is given by

$$L = M * v^*$$

where  $M$  is the mass of the rigid body and  $v^*$  is the velocity of the mass center of  $B$  in the frame,  $N$ .

### Examples

```
>>> from sympy.physics.mechanics import Point, ReferenceFrame, outer
>>> from sympy.physics.mechanics import RigidBody, dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> M, v = dynamicsymbols('M v')
>>> N = ReferenceFrame('N')
>>> P = Point('P')
>>> P.set_vel(N, v * N.x)
>>> I = outer (N.x, N.x)
>>> Inertia_tuple = (I, P)
>>> B = RigidBody('B', P, N, M, Inertia_tuple)
>>> B.linear_momentum(N)
M*v*N.x
```

#### `parallel_axis(point)`

Returns the inertia dyadic of the body with respect to another point.

##### Parameters

**point** : sympy.physics.vector.Point

The point to express the inertia dyadic about.

##### Returns

**inertia** : sympy.physics.vector.Dyadic

The inertia dyadic of the rigid body expressed about the provided point.

#### `property potential_energy`

The potential energy of the RigidBody.

### Examples

```
>>> from sympy.physics.mechanics import RigidBody, Point, outer,
↳ReferenceFrame
>>> from sympy import symbols
>>> M, g, h = symbols('M g h')
>>> b = ReferenceFrame('b')
>>> P = Point('P')
```

(continues on next page)

(continued from previous page)

```
>>> I = outer (b.x, b.x)
>>> Inertia_tuple = (I, P)
>>> B = RigidBody('B', P, b, M, Inertia_tuple)
>>> B.potential_energy = M * g * h
>>> B.potential_energy
M*g*h
```

## inertia

`sympy.physics.mechanics.functions.inertia`(*frame*, *ixx*, *iyy*, *izz*, *ixy=0*, *iyz=0*, *izx=0*)

Simple way to create inertia Dyadic object.

### Parameters

**frame** : ReferenceFrame

The frame the inertia is defined in

**ixx** : Sympifyable

the xx element in the inertia dyadic

**iyy** : Sympifyable

the yy element in the inertia dyadic

**izz** : Sympifyable

the zz element in the inertia dyadic

**ixy** : Sympifyable

the xy element in the inertia dyadic

**iyz** : Sympifyable

the yz element in the inertia dyadic

**izx** : Sympifyable

the zx element in the inertia dyadic

### Explanation

If you do not know what a Dyadic is, just treat this like the inertia tensor. Then, do the easy thing and define it in a body-fixed frame.

## Examples

```
>>> from sympy.physics.mechanics import ReferenceFrame, inertia
>>> N = ReferenceFrame('N')
>>> inertia(N, 1, 2, 3)
(N.x|N.x) + 2*(N.y|N.y) + 3*(N.z|N.z)
```

## inertia\_of\_point\_mass

`sympy.physics.mechanics.functions.inertia_of_point_mass(mass, pos_vec, frame)`

Inertia dyadic of a point mass relative to point O.

### Parameters

**mass** : Sympifyable

Mass of the point mass

**pos\_vec** : Vector

Position from point O to point mass

**frame** : ReferenceFrame

Reference frame to express the dyadic in

## Examples

```
>>> from sympy import symbols
>>> from sympy.physics.mechanics import ReferenceFrame, inertia_of_point_
↳ mass
>>> N = ReferenceFrame('N')
>>> r, m = symbols('r m')
>>> px = r * N.x
>>> inertia_of_point_mass(m, px, N)
m*r**2*(N.y|N.y) + m*r**2*(N.z|N.z)
```

## linear\_momentum

`sympy.physics.mechanics.functions.linear_momentum(frame, *body)`

Linear momentum of the system.

### Parameters

**frame** : ReferenceFrame

The frame in which linear momentum is desired.

**body1, body2, body3...** : Particle and/or RigidBody

The body (or bodies) whose linear momentum is required.

### Explanation

This function returns the linear momentum of a system of Particle's and/or RigidBody's. The linear momentum of a system is equal to the vector sum of the linear momentum of its constituents. Consider a system, S, comprised of a rigid body, A, and a particle, P. The linear momentum of the system, L, is equal to the vector sum of the linear momentum of the particle, L1, and the linear momentum of the rigid body, L2, i.e.

$$L = L1 + L2$$

### Examples

```
>>> from sympy.physics.mechanics import Point, Particle, ReferenceFrame
>>> from sympy.physics.mechanics import RigidBody, outer, linear_momentum
>>> N = ReferenceFrame('N')
>>> P = Point('P')
>>> P.set_vel(N, 10 * N.x)
>>> Pa = Particle('Pa', P, 1)
>>> Ac = Point('Ac')
>>> Ac.set_vel(N, 25 * N.y)
>>> I = outer(N.x, N.x)
>>> A = RigidBody('A', Ac, N, 20, (I, Ac))
>>> linear_momentum(N, A, Pa)
10*N.x + 500*N.y
```

### angular\_momentum

`sympy.physics.mechanics.functions.angular_momentum(point, frame, *body)`

Angular momentum of a system.

#### Parameters

**point** : Point

The point about which angular momentum of the system is desired.

**frame** : ReferenceFrame

The frame in which angular momentum is desired.

**body1, body2, body3...** : Particle and/or RigidBody

The body (or bodies) whose angular momentum is required.

### Explanation

This function returns the angular momentum of a system of Particle's and/or RigidBody's. The angular momentum of such a system is equal to the vector sum of the angular momentum of its constituents. Consider a system, S, comprised of a rigid body, A, and a particle, P. The angular momentum of the system, H, is equal to the vector sum of the angular momentum of the particle, H1, and the angular momentum of the rigid body, H2, i.e.

$$H = H1 + H2$$



## Examples

```
>>> from sympy.physics.mechanics import Point, Particle, ReferenceFrame
>>> from sympy.physics.mechanics import RigidBody, outer, angular_
 ↪ momentum
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> O.set_vel(N, 0 * N.x)
>>> P = O.locatenew('P', 1 * N.x)
>>> P.set_vel(N, 10 * N.x)
>>> Pa = Particle('Pa', P, 1)
>>> Ac = O.locatenew('Ac', 2 * N.y)
>>> Ac.set_vel(N, 5 * N.y)
>>> a = ReferenceFrame('a')
>>> a.set_ang_vel(N, 10 * N.z)
>>> I = outer(N.z, N.z)
>>> A = RigidBody('A', Ac, a, 20, (I, Ac))
>>> angular_momentum(O, N, Pa, A)
10*N.z
```

## kinetic\_energy

`sympy.physics.mechanics.functions.kinetic_energy(frame, *body)`

Kinetic energy of a multibody system.

### Parameters

**frame** : ReferenceFrame

The frame in which the velocity or angular velocity of the body is defined.

**body1, body2, body3...** : Particle and/or RigidBody

The body (or bodies) whose kinetic energy is required.

## Explanation

This function returns the kinetic energy of a system of Particle's and/or RigidBody's. The kinetic energy of such a system is equal to the sum of the kinetic energies of its constituents. Consider a system,  $S$ , comprising a rigid body,  $A$ , and a particle,  $P$ . The kinetic energy of the system,  $T$ , is equal to the vector sum of the kinetic energy of the particle,  $T_1$ , and the kinetic energy of the rigid body,  $T_2$ , i.e.

$$T = T_1 + T_2$$

Kinetic energy is a scalar.

## Examples

```
>>> from sympy.physics.mechanics import Point, Particle, ReferenceFrame
>>> from sympy.physics.mechanics import RigidBody, outer, kinetic_energy
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> O.set_vel(N, 0 * N.x)
>>> P = O.locatenew('P', 1 * N.x)
>>> P.set_vel(N, 10 * N.x)
>>> Pa = Particle('Pa', P, 1)
>>> Ac = O.locatenew('Ac', 2 * N.y)
>>> Ac.set_vel(N, 5 * N.y)
>>> a = ReferenceFrame('a')
>>> a.set_ang_vel(N, 10 * N.z)
>>> I = outer(N.z, N.z)
>>> A = RigidBody('A', Ac, a, 20, (I, Ac))
>>> kinetic_energy(N, Pa, A)
350
```

## potential\_energy

`sympy.physics.mechanics.functions.potential_energy(*body)`

Potential energy of a multibody system.

### Parameters

**body1, body2, body3...** : Particle and/or RigidBody

The body (or bodies) whose potential energy is required.

## Explanation

This function returns the potential energy of a system of Particle's and/or RigidBody's. The potential energy of such a system is equal to the sum of the potential energy of its constituents. Consider a system,  $S$ , comprising a rigid body,  $A$ , and a particle,  $P$ . The potential energy of the system,  $V$ , is equal to the vector sum of the potential energy of the particle,  $V_1$ , and the potential energy of the rigid body,  $V_2$ , i.e.

$$V = V_1 + V_2$$

Potential energy is a scalar.

## Examples

```
>>> from sympy.physics.mechanics import Point, Particle, ReferenceFrame
>>> from sympy.physics.mechanics import RigidBody, outer, potential_
 ↪energy
>>> from sympy import symbols
>>> M, m, g, h = symbols('M m g h')
>>> N = ReferenceFrame('N')
>>> O = Point('O')
```

(continues on next page)

(continued from previous page)

```

>>> O.set_vel(N, 0 * N.x)
>>> P = O.locatenew('P', 1 * N.x)
>>> Pa = Particle('Pa', P, m)
>>> Ac = O.locatenew('Ac', 2 * N.y)
>>> a = ReferenceFrame('a')
>>> I = outer(N.z, N.z)
>>> A = RigidBody('A', Ac, a, M, (I, Ac))
>>> Pa.potential_energy = m * g * h
>>> A.potential_energy = M * g * h
>>> potential_energy(Pa, A)
M*g*h + g*h*m

```

## Lagrangian

`sympy.physics.mechanics.functions.Lagrangian(frame, *body)`

Lagrangian of a multibody system.

### Parameters

**frame** : ReferenceFrame

The frame in which the velocity or angular velocity of the body is defined to determine the kinetic energy.

**body1, body2, body3...** : Particle and/or RigidBody

The body (or bodies) whose Lagrangian is required.

## Explanation

This function returns the Lagrangian of a system of Particle's and/or RigidBody's. The Lagrangian of such a system is equal to the difference between the kinetic energies and potential energies of its constituents. If  $T$  and  $V$  are the kinetic and potential energies of a system then it's Lagrangian,  $L$ , is defined as

$$L = T - V$$

The Lagrangian is a scalar.

## Examples

```

>>> from sympy.physics.mechanics import Point, Particle, ReferenceFrame
>>> from sympy.physics.mechanics import RigidBody, outer, Lagrangian
>>> from sympy import symbols
>>> M, m, g, h = symbols('M m g h')
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> O.set_vel(N, 0 * N.x)
>>> P = O.locatenew('P', 1 * N.x)
>>> P.set_vel(N, 10 * N.x)
>>> Pa = Particle('Pa', P, 1)

```

(continues on next page)

(continued from previous page)

```
>>> Ac = O.locatenew('Ac', 2 * N.y)
>>> Ac.set_vel(N, 5 * N.y)
>>> a = ReferenceFrame('a')
>>> a.set_ang_vel(N, 10 * N.z)
>>> I = outer(N.z, N.z)
>>> A = RigidBody('A', Ac, a, 20, (I, Ac))
>>> Pa.potential_energy = m * g * h
>>> A.potential_energy = M * g * h
>>> Lagrangian(N, Pa, A)
-M*g*h - g*h*m + 350
```

## Body (Docstrings)

### Body

**class** sympy.physics.mechanics.body.**Body**(*name*, *masscenter=None*, *mass=None*, *frame=None*, *central\_inertia=None*)

Body is a common representation of either a RigidBody or a Particle SymPy object depending on what is passed in during initialization. If a mass is passed in and `central_inertia` is left as `None`, the Particle object is created. Otherwise a RigidBody object will be created.

#### Parameters

**name** : String

Defines the name of the body. It is used as the base for defining body specific properties.

**masscenter** : Point, optional

A point that represents the center of mass of the body or particle. If no point is given, a point is generated.

**mass** : Sympifyable, optional

A Sympifyable object which represents the mass of the body. If no mass is passed, one is generated.

**frame** : ReferenceFrame, optional

The ReferenceFrame that represents the reference frame of the body. If no frame is given, a frame is generated.

**central\_inertia** : Dyadic, optional

Central inertia dyadic of the body. If none is passed while creating RigidBody, a default inertia is generated.

## Explanation

The attributes that Body possesses will be the same as a Particle instance or a Rigid Body instance depending on which was created. Additional attributes are listed below.

## Examples

Default behaviour. This results in the creation of a RigidBody object for which the mass, mass center, frame and inertia attributes are given default values.

```
>>> from sympy.physics.mechanics import Body
>>> body = Body('name_of_body')
```

This next example demonstrates the code required to specify all of the values of the Body object. Note this will also create a RigidBody version of the Body object.

```
>>> from sympy import Symbol
>>> from sympy.physics.mechanics import ReferenceFrame, Point, inertia
>>> from sympy.physics.mechanics import Body
>>> mass = Symbol('mass')
>>> masscenter = Point('masscenter')
>>> frame = ReferenceFrame('frame')
>>> ixx = Symbol('ixx')
>>> body_inertia = inertia(frame, ixx, 0, 0)
>>> body = Body('name_of_body', masscenter, mass, frame, body_inertia)
```

The minimal code required to create a Particle version of the Body object involves simply passing in a name and a mass.

```
>>> from sympy import Symbol
>>> from sympy.physics.mechanics import Body
>>> mass = Symbol('mass')
>>> body = Body('name_of_body', mass=mass)
```

The Particle version of the Body object can also receive a masscenter point and a reference frame, just not an inertia.

## Attributes

|             |                                                                                                                                                                                               |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| name        | (string) The body's name                                                                                                                                                                      |
| mass-center | (Point) The point which represents the center of mass of the rigid body                                                                                                                       |
| frame       | (ReferenceFrame) The reference frame which the body is fixed in                                                                                                                               |
| mass        | (Sympifyable) The body's mass                                                                                                                                                                 |
| inertia     | ((Dyadic, Point)) The body's inertia around its center of mass. This attribute is specific to the rigid body form of Body and is left undefined for the Particle form                         |
| loads       | (iterable) This list contains information on the different loads acting on the Body. Forces are listed as a (point, vector) tuple and torques are listed as (reference frame, vector) tuples. |

**ang\_vel\_in(*body*)**

Returns this body's angular velocity with respect to the provided rigid body or reference frame.

**Parameters**

**body: Body or ReferenceFrame**

The rigid body or reference frame to calculate the angular velocity in.

**Example**

```
>>> from sympy.physics.mechanics import Body, ReferenceFrame
>>> A = Body('A')
>>> N = ReferenceFrame('N')
>>> B = Body('B', frame=N)
>>> A.frame.set_ang_vel(N, 5*N.x)
>>> A.ang_vel_in(B)
5*N.x
>>> A.ang_vel_in(N)
5*N.x
```

**apply\_force(*force*, *point*=None, *reaction\_body*=None, *reaction\_point*=None)**

Add force to the body(s).

**Parameters**

**force: Vector**

The force to be applied.

**point: Point, optional**

The point on self on which force is applied. By default self's masscenter.

**reaction\_body: Body, optional**

Second body on which equal and opposite force is to be applied.

**reaction\_point** : Point, optional

The point on other body on which equal and opposite force is applied. By default masscenter of other body.

**Explanation**

Applies the force on self or equal and opposite forces on self and other body if both are given on the desired point on the bodies. The force applied on other body is taken opposite of self, i.e., -force.

### Example

```
>>> from sympy import symbols
>>> from sympy.physics.mechanics import Body, Point, dynamicsymbols
>>> m, g = symbols('m g')
>>> B = Body('B')
>>> force1 = m*g*B.z
>>> B.apply_force(force1) #Applying force on B's masscenter
>>> B.loads
[(B_masscenter, g*m*B_frame.z)]
```

We can also remove some part of force from any point on the body by adding the opposite force to the body on that point.

```
>>> f1, f2 = dynamicsymbols('f1 f2')
>>> P = Point('P') #Considering point P on body B
>>> B.apply_force(f1*B.x + f2*B.y, P)
>>> B.loads
[(B_masscenter, g*m*B_frame.z), (P, f1(t)*B_frame.x + f2(t)*B_frame.
↪y)]
```

Let's remove  $f_1$  from point  $P$  on body  $B$ .

```
>>> B.apply_force(-f1*B.x, P)
>>> B.loads
[(B_masscenter, g*m*B_frame.z), (P, f2(t)*B_frame.y)]
```

To further demonstrate the use of `apply_force` attribute, consider two bodies connected through a spring.

```
>>> from sympy.physics.mechanics import Body, dynamicsymbols
>>> N = Body('N') #Newtonian Frame
>>> x = dynamicsymbols('x')
>>> B1 = Body('B1')
>>> B2 = Body('B2')
>>> spring_force = x*N.x
```

Now let's apply equal and opposite spring force to the bodies.

```
>>> P1 = Point('P1')
>>> P2 = Point('P2')
>>> B1.apply_force(spring_force, point=P1, reaction_body=B2, reaction_
↪point=P2)
```

We can check the loads(forces) applied to bodies now.

```
>>> B1.loads
[(P1, x(t)*N_frame.x)]
>>> B2.loads
[(P2, - x(t)*N_frame.x)]
```

## Notes

If a new force is applied to a body on a point which already has some force applied on it, then the new force is added to the already applied force on that point.

**apply\_torque**(torque, reaction\_body=None)

Add torque to the body(s).

### Parameters

**torque: Vector**

The torque to be applied.

**reaction\_body: Body, optional**

Second body on which equal and opposite torque is to be applied.

## Explanation

Applies the torque on self or equal and opposite torquess on self and other body if both are given. The torque applied on other body is taken opposite of self, i.e, -torque.

## Example

```
>>> from sympy import symbols
>>> from sympy.physics.mechanics import Body, dynamicsymbols
>>> t = symbols('t')
>>> B = Body('B')
>>> torque1 = t*B.z
>>> B.apply_torque(torque1)
>>> B.loads
[(B_frame, t*B_frame.z)]
```

We can also remove some part of torque from the body by adding the opposite torque to the body.

```
>>> t1, t2 = dynamicsymbols('t1 t2')
>>> B.apply_torque(t1*B.x + t2*B.y)
>>> B.loads
[(B_frame, t1(t)*B_frame.x + t2(t)*B_frame.y + t*B_frame.z)]
```

Let's remove t1 from Body B.

```
>>> B.apply_torque(-t1*B.x)
>>> B.loads
[(B_frame, t2(t)*B_frame.y + t*B_frame.z)]
```

To further demonstrate the use, let us consider two bodies such that a torque  $T$  is acting on one body, and  $-T$  on the other.

```
>>> from sympy.physics.mechanics import Body, dynamicsymbols
>>> N = Body('N') #Newtonian frame
```

(continues on next page)



(continued from previous page)

```
>>> B1 = Body('B1')
>>> B2 = Body('B2')
>>> v = dynamicsymbols('v')
>>> T = v*N.y #Torque
```

Now let's apply equal and opposite torque to the bodies.

```
>>> B1.apply_torque(T, B2)
```

We can check the loads (torques) applied to bodies now.

```
>>> B1.loads
[(B1_frame, v(t)*N_frame.y)]
>>> B2.loads
[(B2_frame, - v(t)*N_frame.y)]
```

## Notes

If a new torque is applied on body which already has some torque applied on it, then the new torque is added to the previous torque about the body's frame.

### `clear_loads()`

Clears the Body's loads list.

## Example

```
>>> from sympy.physics.mechanics import Body
>>> B = Body('B')
>>> force = B.x + B.y
>>> B.apply_force(force)
>>> B.loads
[(B_masscenter, B_frame.x + B_frame.y)]
>>> B.clear_loads()
>>> B.loads
[]
```

### `dcm(body)`

Returns the direction cosine matrix of this body relative to the provided rigid body or reference frame.

#### Parameters

**body:** Body or ReferenceFrame

The rigid body or reference frame to calculate the dcm.

### Example

```
>>> from sympy.physics.mechanics import Body
>>> A = Body('A')
>>> B = Body('B')
>>> A.frame.orient_axis(B.frame, B.frame.x, 5)
>>> A.dcm(B)
Matrix([
[1, 0, 0],
[0, cos(5), sin(5)],
[0, -sin(5), cos(5)]])
>>> A.dcm(B.frame)
Matrix([
[1, 0, 0],
[0, cos(5), sin(5)],
[0, -sin(5), cos(5)]])
```

#### **kinetic\_energy**(*frame*)

Kinetic energy of the body.

##### Parameters

**frame** : ReferenceFrame or Body

The Body's angular velocity and the velocity of it's mass center are typically defined with respect to an inertial frame but any relevant frame in which the velocities are known can be supplied.

### Examples

```
>>> from sympy.physics.mechanics import Body, ReferenceFrame, Point
>>> from sympy import symbols
>>> m, v, r, omega = symbols('m v r omega')
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> P = Body('P', masscenter=O, mass=m)
>>> P.masscenter.set_vel(N, v * N.y)
>>> P.kinetic_energy(N)
m*v**2/2
```

```
>>> N = ReferenceFrame('N')
>>> b = ReferenceFrame('b')
>>> b.set_ang_vel(N, omega * b.x)
>>> P = Point('P')
>>> P.set_vel(N, v * N.x)
>>> B = Body('B', masscenter=P, frame=b)
>>> B.kinetic_energy(N)
B_ixx*omega**2/2 + B_mass*v**2/2
```

#### See also:

[\*sympy.physics.mechanics\*](#) (page 1675)

Particle, RigidBody

**masscenter\_vel(*body*)**

Returns the velocity of the mass center with respect to the provided rigid body or reference frame.

**Parameters**

**body:** **Body** or **ReferenceFrame**

The rigid body or reference frame to calculate the velocity in.

**Example**

```
>>> from sympy.physics.mechanics import Body
>>> A = Body('A')
>>> B = Body('B')
>>> A.masscenter.set_vel(B.frame, 5*B.frame.x)
>>> A.masscenter_vel(B)
5*B_frame.x
>>> A.masscenter_vel(B.frame)
5*B_frame.x
```

**remove\_load(*about=None*)**

Remove load about a point or frame.

**Parameters**

**about :** Point or ReferenceFrame, optional

The point about which force is applied, and is to be removed. If about is None, then the torque about self's frame is removed.

**Example**

```
>>> from sympy.physics.mechanics import Body, Point
>>> B = Body('B')
>>> P = Point('P')
>>> f1 = B.x
>>> f2 = B.y
>>> B.apply_force(f1)
>>> B.apply_force(f2, P)
>>> B.loads
[(B_masscenter, B_frame.x), (P, B_frame.y)]
```

```
>>> B.remove_load(P)
>>> B.loads
[(B_masscenter, B_frame.x)]
```

**property x**

The basis Vector for the Body, in the x direction.

**property y**

The basis Vector for the Body, in the y direction.

**property z**

The basis Vector for the Body, in the z direction.

## Kane's Method & Lagrange's Method (Docstrings)

### KaneMethod

```
class sympy.physics.mechanics.kane.KanesMethod(frame, q_ind, u_ind, kd_eqs=None,
 q_dependent=None,
 configuration_constraints=None,
 u_dependent=None,
 velocity_constraints=None,
 acceleration_constraints=None,
 u_auxiliary=None, bodies=None,
 forcelist=None)
```

Kane's method object.

### Explanation

This object is used to do the “book-keeping” as you go through and form equations of motion in the way Kane presents in: Kane, T., Levinson, D. Dynamics Theory and Applications. 1985 McGraw-Hill

The attributes are for equations in the form  $[M] \dot{u} = \text{forcing}$ .

### Examples

This is a simple example for a one degree of freedom translational spring-mass-damper.

In this example, we first need to do the kinematics. This involves creating generalized speeds and coordinates and their derivatives. Then we create a point and set its velocity in a frame.

```
>>> from sympy import symbols
>>> from sympy.physics.mechanics import dynamicsymbols, ReferenceFrame
>>> from sympy.physics.mechanics import Point, Particle, KanesMethod
>>> q, u = dynamicsymbols('q u')
>>> qd, ud = dynamicsymbols('q u', 1)
>>> m, c, k = symbols('m c k')
>>> N = ReferenceFrame('N')
>>> P = Point('P')
>>> P.set_vel(N, u * N.x)
```

Next we need to arrange/store information in the way that KanesMethod requires. The kinematic differential equations need to be stored in a dict. A list of forces/torques must be constructed, where each entry in the list is a (Point, Vector) or (ReferenceFrame, Vector) tuple, where the Vectors represent the Force or Torque. Next a particle needs to be created, and it needs to have a point and mass assigned to it. Finally, a list of all bodies and particles needs to be created.

```
>>> kd = {qd - u}
>>> FL = [(P, (-k * q - c * u) * N.x)]
>>> pa = Particle('pa', P, m)
>>> BL = [pa]
```

Finally we can generate the equations of motion. First we create the KanesMethod object and supply an inertial frame, coordinates, generalized speeds, and the kinematic differential equations. Additional quantities such as configuration and motion constraints, dependent coordinates and speeds, and auxiliary speeds are also supplied here (see the online documentation). Next we form  $FR^*$  and  $FR$  to complete:  $Fr + Fr^* = 0$ . We have the equations of motion at this point. It makes sense to rearrange them though, so we calculate the mass matrix and the forcing terms, for E.o.M. in the form:  $[MM] \dot{u} = \text{forcing}$ , where  $MM$  is the mass matrix,  $\dot{u}$  is a vector of the time derivatives of the generalized speeds, and forcing is a vector representing “forcing” terms.

```
>>> KM = KanesMethod(N, q_ind=[q], u_ind=[u], kd_eqs=kd)
>>> (fr, frstar) = KM.kanes_equations(BL, FL)
>>> MM = KM.mass_matrix
>>> forcing = KM.forcing
>>> rhs = MM.inv() * forcing
>>> rhs
Matrix([[(-c*u(t) - k*q(t))/m]])
>>> KM.linearize(A_and_B=True)[0]
Matrix([
[0, 1],
[-k/m, -c/m]])
```

Please look at the documentation pages for more information on how to perform linearization and how to deal with dependent coordinates & speeds, and how to deal with bringing non-contributing forces into evidence.

## Attributes

|                               |                                                                                                                |
|-------------------------------|----------------------------------------------------------------------------------------------------------------|
| <code>q, u</code>             | (Matrix) Matrices of the generalized coordinates and speeds                                                    |
| <code>bodies</code>           | (iterable) Iterable of Point and RigidBody objects in the system.                                              |
| <code>loads</code>            | (iterable) Iterable of (Point, vector) or (ReferenceFrame, vector) tuples describing the forces on the system. |
| <code>auxiliary</code>        | (Matrix) If applicable, the set of auxiliary Kane’s equations used to solve for non-contributing forces.       |
| <code>mass_matrix</code>      | (Matrix) The system’s mass matrix                                                                              |
| <code>forcing</code>          | (Matrix) The system’s forcing vector                                                                           |
| <code>mass_matrix_full</code> | (Matrix) The “mass matrix” for the $u$ ’s and $q$ ’s                                                           |
| <code>forcing_full</code>     | (Matrix) The “forcing vector” for the $u$ ’s and $q$ ’s                                                        |

### property `auxiliary_eqs`

A matrix containing the auxiliary equations.

### property `forcing`

The forcing vector of the system.

### property `forcing_full`

The forcing vector of the system, augmented by the kinematic differential equations.

### `kanes_equations(bodies=None, loads=None)`

Method to form Kane’s equations,  $Fr + Fr^* = 0$ .

**Parameters****bodies** : iterable

An iterable of all RigidBody's and Particle's in the system. A system must have at least one body.

**loads** : iterable

Takes in an iterable of (Particle, Vector) or (ReferenceFrame, Vector) tuples which represent the force at a point or torque on a frame. Must be either a non-empty iterable of tuples or None which corresponds to a system with no constraints.

**Explanation**

Returns (Fr, Fr\*). In the case where auxiliary generalized speeds are present (say, s auxiliary speeds, o generalized speeds, and m motion constraints) the length of the returned vectors will be o - m + s in length. The first o - m equations will be the constrained Kane's equations, then the s auxiliary Kane's equations. These auxiliary equations can be accessed with the auxiliary\_eqs().

**kindiffdict()**

Returns a dictionary mapping  $q'$  to  $u$ .

**linearize(\*, new\_method=None, \*\*kwargs)**

Linearize the equations of motion about a symbolic operating point.

**Explanation**

If kwarg `A_and_B` is False (default), returns M, A, B, r for the linearized form,  $M[q', u']^T = A[q_{ind}, u_{ind}]^T + B*r$ .

If kwarg `A_and_B` is True, returns A, B, r for the linearized form  $dx = A*x + B*r$ , where  $x = [q_{ind}, u_{ind}]^T$ . Note that this is computationally intensive if there are many symbolic parameters. For this reason, it may be more desirable to use the default `A_and_B=False`, returning M, A, and B. Values may then be substituted in to these matrices, and the state space form found as  $A = P.T*M.inv()*A$ ,  $B = P.T*M.inv()*B$ , where  $P = \text{Linearizer.perm\_mat}$ .

In both cases, r is found as all dynamicsymbols in the equations of motion that are not part of q, u, q', or u'. They are sorted in canonical form.

The operating points may be also entered using the `op_point` kwarg. This takes a dictionary of {symbol: value}, or a an iterable of such dictionaries. The values may be numeric or symbolic. The more values you can specify beforehand, the faster this computation will run.

For more documentation, please see the Linearizer class.

**property mass\_matrix**

The mass matrix of the system.

**property mass\_matrix\_full**

The mass matrix of the system, augmented by the kinematic differential equations.

**rhs**(*inv\_method=None*)

Returns the system's equations of motion in first order form. The output is the right hand side of:

$$\begin{pmatrix} \dot{q} \\ \dot{u} \end{pmatrix} =: f(q, u, r, p, t)$$

The right hand side is what is needed by most numerical ODE integrators.

#### Parameters

**inv\_method** : str

The specific sympy inverse matrix calculation method to use. For a list of valid methods, see [inv\(\)](#) (page 1299)

**to\_linearizer()**

Returns an instance of the Linearizer class, initiated from the data in the Kanes-Method class. This may be more desirable than using the linearize class method, as the Linearizer object will allow more efficient recalculation (i.e. about varying operating points).

## LagrangesMethod

```
class sympy.physics.mechanics.lagrange.LagrangesMethod(Lagrangian, qs,
 forcelist=None,
 bodies=None,
 frame=None,
 hol_coneqs=None,
 nonhol_coneqs=None)
```

Lagrange's method object.

### Explanation

This object generates the equations of motion in a two step procedure. The first step involves the initialization of LagrangesMethod by supplying the Lagrangian and the generalized coordinates, at the bare minimum. If there are any constraint equations, they can be supplied as keyword arguments. The Lagrange multipliers are automatically generated and are equal in number to the constraint equations. Similarly any non-conservative forces can be supplied in an iterable (as described below and also shown in the example) along with a ReferenceFrame. This is also discussed further in the `__init__` method.

### Examples

This is a simple example for a one degree of freedom translational spring-mass-damper. In this example, we first need to do the kinematics. This involves creating generalized coordinates and their derivatives. Then we create a point and set its velocity in a frame.

```
>>> from sympy.physics.mechanics import LagrangesMethod, Lagrangian
>>> from sympy.physics.mechanics import ReferenceFrame, Particle, Point
>>> from sympy.physics.mechanics import dynamicsymbols
```

(continues on next page)

(continued from previous page)

```
>>> from sympy import symbols
>>> q = dynamicsymbols('q')
>>> qd = dynamicsymbols('q', 1)
>>> m, k, b = symbols('m k b')
>>> N = ReferenceFrame('N')
>>> P = Point('P')
>>> P.set_vel(N, qd * N.x)
```

We need to then prepare the information as required by `LagrangesMethod` to generate equations of motion. First we create the `Particle`, which has a point attached to it. Following this the lagrangian is created from the kinetic and potential energies. Then, an iterable of nonconservative forces/torques must be constructed, where each item is a `(Point, Vector)` or `(ReferenceFrame, Vector)` tuple, with the Vectors representing the nonconservative forces or torques.

```
>>> Pa = Particle('Pa', P, m)
>>> Pa.potential_energy = k * q**2 / 2.0
>>> L = Lagrangian(N, Pa)
>>> fl = [(P, -b * qd * N.x)]
```

Finally we can generate the equations of motion. First we create the `LagrangesMethod` object. To do this one must supply the Lagrangian, and the generalized coordinates. The constraint equations, the forcelist, and the inertial frame may also be provided, if relevant. Next we generate Lagrange's equations of motion, such that: Lagrange's equations of motion = 0. We have the equations of motion at this point.

```
>>> l = LagrangesMethod(L, [q], forcelist = fl, frame = N)
>>> print(l.form_lagranges_equations())
Matrix([[b*Derivative(q(t), t) + 1.0*k*q(t) + m*Derivative(q(t), (t,
↪2))]])
```

We can also solve for the states using the 'rhs' method.

```
>>> print(l.rhs())
Matrix([[Derivative(q(t), t)], [(-b*Derivative(q(t), t) - 1.0*k*q(t))/
↪m]])
```

Please refer to the docstrings on each method for more details.

## Attributes

|                               |                                                                                                                                          |
|-------------------------------|------------------------------------------------------------------------------------------------------------------------------------------|
| <code>q, u</code>             | (Matrix) Matrices of the generalized coordinates and speeds                                                                              |
| <code>loads</code>            | (iterable) Iterable of <code>(Point, vector)</code> or <code>(ReferenceFrame, vector)</code> tuples describing the forces on the system. |
| <code>bodies</code>           | (iterable) Iterable containing the rigid bodies and particles of the system.                                                             |
| <code>mass_matrix</code>      | (Matrix) The system's mass matrix                                                                                                        |
| <code>forcing</code>          | (Matrix) The system's forcing vector                                                                                                     |
| <code>mass_matrix_full</code> | (Matrix) The "mass matrix" for the <code>qdot</code> 's, <code>qddot</code> 's, and the lagrange multipliers ( <code>lam</code> )        |
| <code>forcing_full</code>     | (Matrix) The forcing vector for the <code>qdot</code> 's, <code>qddot</code> 's and lagrange multipliers ( <code>lam</code> )            |



**property forcing**

Returns the forcing vector from 'lagranges\_equations' method.

**property forcing\_full**

Augments qdots to the forcing vector above.

**form\_lagranges\_equations()**

Method to form Lagrange's equations of motion.

Returns a vector of equations of motion using Lagrange's equations of the second kind.

**linearize(*q\_ind=None, qd\_ind=None, q\_dep=None, qd\_dep=None, \*\*kwargs*)**

Linearize the equations of motion about a symbolic operating point.

**Explanation**

If kwarg *A\_and\_B* is False (default), returns *M*, *A*, *B*, *r* for the linearized form,  $M[q', u']^T = A[q_{ind}, u_{ind}]^T + B*r$ .

If kwarg *A\_and\_B* is True, returns *A*, *B*, *r* for the linearized form  $dx = A*x + B*r$ , where  $x = [q_{ind}, u_{ind}]^T$ . Note that this is computationally intensive if there are many symbolic parameters. For this reason, it may be more desirable to use the default *A\_and\_B=False*, returning *M*, *A*, and *B*. Values may then be substituted in to these matrices, and the state space form found as  $A = P.T*M.inv()*A$ ,  $B = P.T*M.inv()*B$ , where  $P = \text{Linearizer.perm\_mat}$ .

In both cases, *r* is found as all dynamicsymbols in the equations of motion that are not part of *q*, *u*, *q'*, or *u'*. They are sorted in canonical form.

The operating points may be also entered using the *op\_point* kwarg. This takes a dictionary of {symbol: value}, or a an iterable of such dictionaries. The values may be numeric or symbolic. The more values you can specify beforehand, the faster this computation will run.

For more documentation, please see the Linearizer class.

**property mass\_matrix**

Returns the mass matrix, which is augmented by the Lagrange multipliers, if necessary.

**Explanation**

If the system is described by '*n*' generalized coordinates and there are no constraint equations then an *n* X *n* matrix is returned.

If there are '*n*' generalized coordinates and '*m*' constraint equations have been supplied during initialization then an *n* X (*n*+*m*) matrix is returned. The (*n* + *m* - 1)th and (*n* + *m*)th columns contain the coefficients of the Lagrange multipliers.

**property mass\_matrix\_full**

Augments the coefficients of qdots to the mass\_matrix.

**rhs(*inv\_method=None, \*\*kwargs*)**

Returns equations that can be solved numerically.

**Parameters****inv\_method** : str

The specific sympy inverse matrix calculation method to use. For a list of valid methods, see [\*inv\(\)\*](#) (page 1299)

**solve\_multipliers**(*op\_point=None, sol\_type='dict'*)

Solves for the values of the lagrange multipliers symbolically at the specified operating point.

**Parameters****op\_point** : dict or iterable of dicts, optional

Point at which to solve at. The operating point is specified as a dictionary or iterable of dictionaries of {symbol: value}. The value may be numeric or symbolic itself.

**sol\_type** : str, optional

Solution return type. Valid options are: - 'dict': A dict of {symbol : value} (default) - 'Matrix': An ordered column matrix of the solution

**to\_linearizer**(*q\_ind=None, qd\_ind=None, q\_dep=None, qd\_dep=None*)

Returns an instance of the Linearizer class, initiated from the data in the Lagranges-Method class. This may be more desirable than using the linearize class method, as the Linearizer object will allow more efficient recalculation (i.e. about varying operating points).

**Parameters****q\_ind, qd\_ind** : array\_like, optional

The independent generalized coordinates and speeds.

**q\_dep, qd\_dep** : array\_like, optional

The dependent generalized coordinates and speeds.

## Joints Framework (Docstrings)

### Joint (Docstrings)

#### Joint

```
class sympy.physics.mechanics.joint.Joint(name, parent, child, coordinates=None,
 speeds=None, parent_joint_pos=None,
 child_joint_pos=None, parent_axis=None,
 child_axis=None)
```

Abstract base class for all specific joints.

**Parameters****name** : string

A unique name for the joint.

**parent** : Body

The parent body of joint.

**child** : Body

The child body of joint.

**coordinates:** List of dynamicsymbols, optional

Generalized coordinates of the joint.

**speeds :** List of dynamicsymbols, optional

Generalized speeds of joint.

**parent\_joint\_pos :** Vector, optional

Vector from the parent body's mass center to the point where the parent and child are connected. The default value is the zero vector.

**child\_joint\_pos :** Vector, optional

Vector from the child body's mass center to the point where the parent and child are connected. The default value is the zero vector.

**parent\_axis :** Vector, optional

Axis fixed in the parent body which aligns with an axis fixed in the child body. The default is x axis in parent's reference frame.

**child\_axis :** Vector, optional

Axis fixed in the child body which aligns with an axis fixed in the parent body. The default is x axis in child's reference frame.

## Explanation

A joint subtracts degrees of freedom from a body. This is the base class for all specific joints and holds all common methods acting as an interface for all joints. Custom joint can be created by inheriting Joint class and defining all abstract functions.

The abstract methods are:

- `_generate_coordinates`
- `_generate_speeds`
- `_orient_frames`
- `_set_angular_velocity`
- `_set_linear_velocity`

## Notes

The direction cosine matrix between the child and parent is formed using a simple rotation about an axis that is normal to both `child_axis` and `parent_axis`. In general, the normal axis is formed by crossing the `child_axis` into the `parent_axis` except if the child and parent axes are in exactly opposite directions. In that case the rotation vector is chosen using the rules in the following table where C is the child reference frame and P is the parent reference frame:

| child_axis   | parent_axis | rotation_axis              |
|--------------|-------------|----------------------------|
| -C.x         | P.x         | P.z                        |
| -C.y         | P.y         | P.x                        |
| -C.z         | P.z         | P.y                        |
| -C.x-C.y     | P.x+P.y     | P.z                        |
| -C.y-C.z     | P.y+P.z     | P.x                        |
| -C.x-C.z     | P.x+P.z     | P.y                        |
| -C.x-C.y-C.z | P.x+P.y+P.z | $(P.x+P.y+P.z) \times P.x$ |

## Attributes

|              |                                                                        |
|--------------|------------------------------------------------------------------------|
| name         | (string) The joint's name.                                             |
| parent       | (Body) The joint's parent body.                                        |
| child        | (Body) The joint's child body.                                         |
| coordinates  | (list) List of the joint's generalized coordinates.                    |
| speeds       | (list) List of the joint's generalized speeds.                         |
| parent_point | (Point) The point fixed in the parent body that represents the joint.  |
| child_point  | (Point) The point fixed in the child body that represents the joint.   |
| parent_axis  | (Vector) The axis fixed in the parent frame that represents the joint. |
| child_axis   | (Vector) The axis fixed in the child frame that represents the joint.  |
| kdes         | (list) Kinematical differential equations of the joint.                |

### property child

Child body of Joint.

### property child\_axis

The axis of child frame.

### property child\_point

The joint's point where child body is connected to the joint.

### property coordinates

List generalized coordinates of the joint.

### property kdes

Kinematical differential equations of the joint.

### property parent

Parent body of Joint.

### property parent\_axis

The axis of parent frame.

### property parent\_point

The joint's point where parent body is connected to the joint.

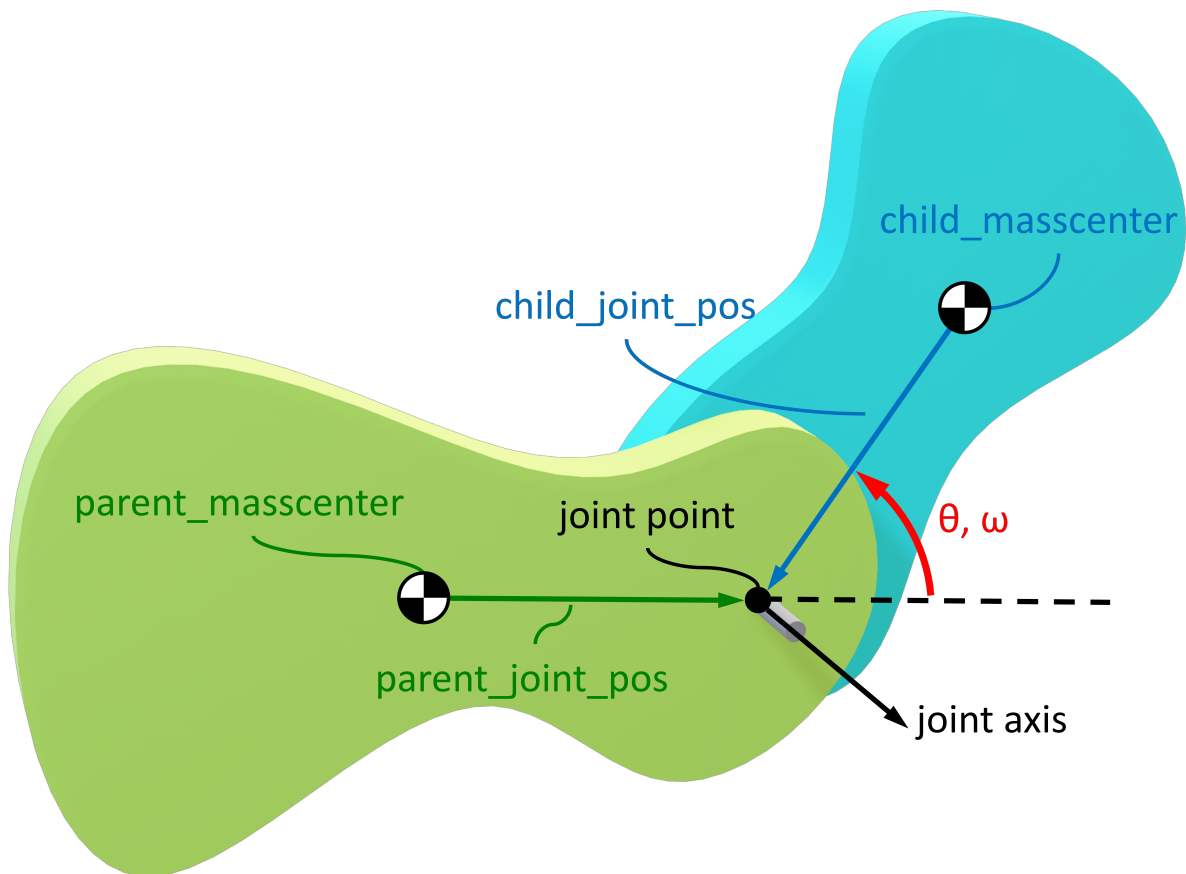
### property speeds

List generalized coordinates of the joint..

## PinJoint

```
class sympy.physics.mechanics.joint.PinJoint(name, parent, child,
 coordinates=None, speeds=None,
 parent_joint_pos=None,
 child_joint_pos=None,
 parent_axis=None, child_axis=None)
```

Pin (Revolute) Joint.



### Parameters

**name** : string

A unique name for the joint.

**parent** : Body

The parent body of joint.

**child** : Body

The child body of joint.

**coordinates**: dynamicsymbol, optional

Generalized coordinates of the joint.

**speeds** : dynamicsymbol, optional

Generalized speeds of joint.

**parent\_joint\_pos** : Vector, optional

Vector from the parent body's mass center to the point where the parent and child are connected. The default value is the zero vector.

**child\_joint\_pos** : Vector, optional

Vector from the child body's mass center to the point where the parent and child are connected. The default value is the zero vector.

**parent\_axis** : Vector, optional

Axis fixed in the parent body which aligns with an axis fixed in the child body. The default is x axis in parent's reference frame.

**child\_axis** : Vector, optional

Axis fixed in the child body which aligns with an axis fixed in the parent body. The default is x axis in child's reference frame.

## Explanation

A pin joint is defined such that the joint rotation axis is fixed in both the child and parent and the location of the joint is relative to the mass center of each body. The child rotates an angle,  $\theta$ , from the parent about the rotation axis and has a simple angular speed,  $\omega$ , relative to the parent. The direction cosine matrix between the child and parent is formed using a simple rotation about an axis that is normal to both `child_axis` and `parent_axis`, see the Notes section for a detailed explanation of this.

## Examples

A single pin joint is created from two bodies and has the following basic attributes:

```
>>> from sympy.physics.mechanics import Body, PinJoint
>>> parent = Body('P')
>>> parent
P
>>> child = Body('C')
>>> child
C
>>> joint = PinJoint('PC', parent, child)
>>> joint
PinJoint: PC parent: P child: C
>>> joint.name
'PC'
>>> joint.parent
P
>>> joint.child
C
>>> joint.parent_point
PC_P_joint
>>> joint.child_point
PC_C_joint
>>> joint.parent_axis
P_frame.x
>>> joint.child_axis
```

(continues on next page)

(continued from previous page)

```

C_frame.x
>>> joint.coordinates
[theta_PC(t)]
>>> joint.speeds
[omega_PC(t)]
>>> joint.child.frame.ang_vel_in(joint.parent.frame)
omega_PC(t)*P_frame.x
>>> joint.child.frame.dcm(joint.parent.frame)
Matrix([
[1, 0, 0],
[0, cos(theta_PC(t)), sin(theta_PC(t))],
[0, -sin(theta_PC(t)), cos(theta_PC(t))]])
>>> joint.child_point.pos_from(joint.parent_point)
0

```

To further demonstrate the use of the pin joint, the kinematics of simple double pendulum that rotates about the Z axis of each connected body can be created as follows.

```

>>> from sympy import symbols, trigsimp
>>> from sympy.physics.mechanics import Body, PinJoint
>>> l1, l2 = symbols('l1 l2')

```

First create bodies to represent the fixed ceiling and one to represent each pendulum bob.

```

>>> ceiling = Body('C')
>>> upper_bob = Body('U')
>>> lower_bob = Body('L')

```

The first joint will connect the upper bob to the ceiling by a distance of  $l_1$  and the joint axis will be about the Z axis for each body.

```

>>> ceiling_joint = PinJoint('P1', ceiling, upper_bob,
... child_joint_pos=-l1*upper_bob.frame.x,
... parent_axis=ceiling.frame.z,
... child_axis=upper_bob.frame.z)

```

The second joint will connect the lower bob to the upper bob by a distance of  $l_2$  and the joint axis will also be about the Z axis for each body.

```

>>> pendulum_joint = PinJoint('P2', upper_bob, lower_bob,
... child_joint_pos=-l2*lower_bob.frame.x,
... parent_axis=upper_bob.frame.z,
... child_axis=lower_bob.frame.z)

```

Once the joints are established the kinematics of the connected bodies can be accessed. First the direction cosine matrices of pendulum link relative to the ceiling are found:

```

>>> upper_bob.frame.dcm(ceiling.frame)
Matrix([
[cos(theta_P1(t)), sin(theta_P1(t)), 0],
[-sin(theta_P1(t)), cos(theta_P1(t)), 0],
[0, 0, 1]])

```

(continues on next page)

(continued from previous page)

```
>>> trigsimp(lower_bob.frame.dcm(ceiling.frame))
Matrix([
[cos(theta_P1(t) + theta_P2(t)), sin(theta_P1(t) + theta_P2(t)), 0],
[-sin(theta_P1(t) + theta_P2(t)), cos(theta_P1(t) + theta_P2(t)), 0],
[0, 0, 1]])
```

The position of the lower bob's masscenter is found with:

```
>>> lower_bob.masscenter.pos_from(ceiling.masscenter)
l1*U_frame.x + l2*L_frame.x
```

The angular velocities of the two pendulum links can be computed with respect to the ceiling.

```
>>> upper_bob.frame.ang_vel_in(ceiling.frame)
omega_P1(t)*C_frame.z
>>> lower_bob.frame.ang_vel_in(ceiling.frame)
omega_P1(t)*C_frame.z + omega_P2(t)*U_frame.z
```

And finally, the linear velocities of the two pendulum bobs can be computed with respect to the ceiling.

```
>>> upper_bob.masscenter.vel(ceiling.frame)
l1*omega_P1(t)*U_frame.y
>>> lower_bob.masscenter.vel(ceiling.frame)
l1*omega_P1(t)*U_frame.y + l2*(omega_P1(t) + omega_P2(t))*L_frame.y
```

## Attributes

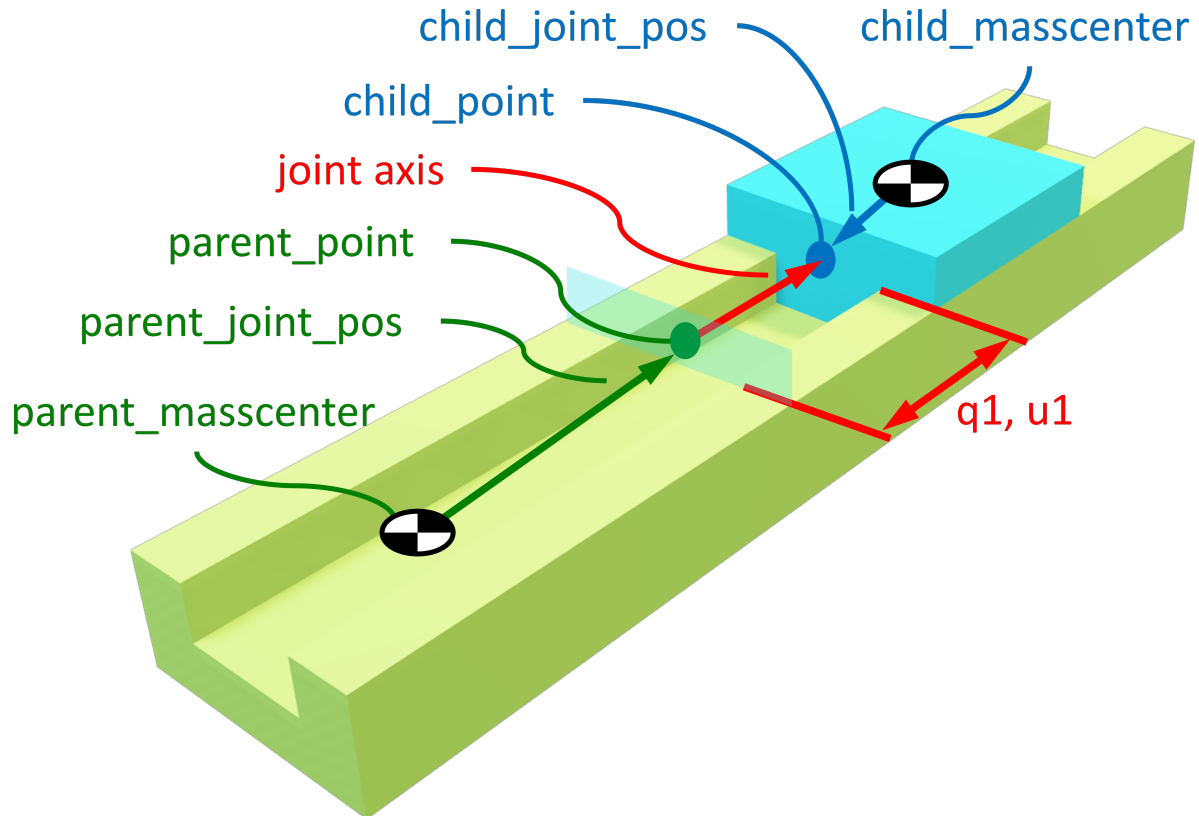
|              |                                                                        |
|--------------|------------------------------------------------------------------------|
| name         | (string) The joint's name.                                             |
| parent       | (Body) The joint's parent body.                                        |
| child        | (Body) The joint's child body.                                         |
| coordinates  | (list) List of the joint's generalized coordinates.                    |
| speeds       | (list) List of the joint's generalized speeds.                         |
| parent_point | (Point) The point fixed in the parent body that represents the joint.  |
| child_point  | (Point) The point fixed in the child body that represents the joint.   |
| parent_axis  | (Vector) The axis fixed in the parent frame that represents the joint. |
| child_axis   | (Vector) The axis fixed in the child frame that represents the joint.  |
| kdes         | (list) Kinematical differential equations of the joint.                |



**PrismaticJoint**

```
class sympy.physics.mechanics.joint.PrismaticJoint(name, parent, child,
 coordinates=None,
 speeds=None,
 parent_joint_pos=None,
 child_joint_pos=None,
 parent_axis=None,
 child_axis=None)
```

Prismatic (Sliding) Joint.

**Parameters**

**name** : string

A unique name for the joint.

**parent** : Body

The parent body of joint.

**child** : Body

The child body of joint.

**coordinates**: dynamicsymbol, optional

Generalized coordinates of the joint.

**speeds** : dynamicsymbol, optional

Generalized speeds of joint.

**parent\_joint\_pos** : Vector, optional

Vector from the parent body's mass center to the point where the parent and child are connected. The default value is the zero vector.

**child\_joint\_pos** : Vector, optional

Vector from the child body's mass center to the point where the parent and child are connected. The default value is the zero vector.

**parent\_axis** : Vector, optional

Axis fixed in the parent body which aligns with an axis fixed in the child body. The default is x axis in parent's reference frame.

**child\_axis** : Vector, optional

Axis fixed in the child body which aligns with an axis fixed in the parent body. The default is x axis in child's reference frame.

## Explanation

It is defined such that the child body translates with respect to the parent body along the body fixed parent axis. The location of the joint is defined by two points in each body which coincides when the generalized coordinate is zero. The direction cosine matrix between the child and parent is formed using a simple rotation about an axis that is normal to both `child_axis` and `parent_axis`, see the Notes section for a detailed explanation of this.

## Examples

A single prismatic joint is created from two bodies and has the following basic attributes:

```
>>> from sympy.physics.mechanics import Body, PrismaticJoint
>>> parent = Body('P')
>>> parent
P
>>> child = Body('C')
>>> child
C
>>> joint = PrismaticJoint('PC', parent, child)
>>> joint
PrismaticJoint: PC parent: P child: C
>>> joint.name
'PC'
>>> joint.parent
P
>>> joint.child
C
>>> joint.parent_point
PC_P_joint
>>> joint.child_point
PC_C_joint
>>> joint.parent_axis
P_frame.x
>>> joint.child_axis
```

(continues on next page)

(continued from previous page)

```

C_frame.x
>>> joint.coordinates
[x_PC(t)]
>>> joint.speeds
[v_PC(t)]
>>> joint.child.frame.ang_vel_in(joint.parent.frame)
0
>>> joint.child.frame.dcm(joint.parent.frame)
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> joint.child_point.pos_from(joint.parent_point)
x_PC(t)*P_frame.x

```

To further demonstrate the use of the prismatic joint, the kinematics of two masses sliding, one moving relative to a fixed body and the other relative to the moving body. about the X axis of each connected body can be created as follows.

```
>>> from sympy.physics.mechanics import PrismaticJoint, Body
```

First create bodies to represent the fixed ceiling and one to represent a particle.

```

>>> wall = Body('W')
>>> Part1 = Body('P1')
>>> Part2 = Body('P2')

```

The first joint will connect the particle to the ceiling and the joint axis will be about the X axis for each body.

```
>>> J1 = PrismaticJoint('J1', wall, Part1)
```

The second joint will connect the second particle to the first particle and the joint axis will also be about the X axis for each body.

```
>>> J2 = PrismaticJoint('J2', Part1, Part2)
```

Once the joint is established the kinematics of the connected bodies can be accessed. First the direction cosine matrices of Part relative to the ceiling are found:

```

>>> Part1.dcm(wall)
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])

```

```

>>> Part2.dcm(wall)
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])

```

The position of the particles' masscenter is found with:

```
>>> Part1.masscenter.pos_from(wall.masscenter)
x_J1(t)*W_frame.x
```

```
>>> Part2.masscenter.pos_from(wall.masscenter)
x_J1(t)*W_frame.x + x_J2(t)*P1_frame.x
```

The angular velocities of the two particle links can be computed with respect to the ceiling.

```
>>> Part1.ang_vel_in(wall)
0
```

```
>>> Part2.ang_vel_in(wall)
0
```

And finally, the linear velocities of the two particles can be computed with respect to the ceiling.

```
>>> Part1.masscenter_vel(wall)
v_J1(t)*W_frame.x
```

```
>>> Part2.masscenter.vel(wall.frame)
v_J1(t)*W_frame.x + Derivative(x_J2(t), t)*P1_frame.x
```

## Attributes

|              |                                                                        |
|--------------|------------------------------------------------------------------------|
| name         | (string) The joint's name.                                             |
| parent       | (Body) The joint's parent body.                                        |
| child        | (Body) The joint's child body.                                         |
| coordinates  | (list) List of the joint's generalized coordinates.                    |
| speeds       | (list) List of the joint's generalized speeds.                         |
| parent_point | (Point) The point fixed in the parent body that represents the joint.  |
| child_point  | (Point) The point fixed in the child body that represents the joint.   |
| parent_axis  | (Vector) The axis fixed in the parent frame that represents the joint. |
| child_axis   | (Vector) The axis fixed in the child frame that represents the joint.  |
| kdes         | (list) Kinematical differential equations of the joint.                |

## JointsMethod (Docstring)

**class** sympy.physics.mechanics.jointsmethod.**JointsMethod**(*newtonion*, *\*joints*)  
Method for formulating the equations of motion using a set of interconnected bodies with joints.

### Parameters

**newtonion** : Body or ReferenceFrame

The newtonion(inertial) frame.

**\*joints** : Joint

The joints in the system

## Examples

This is a simple example for a one degree of freedom translational spring-mass-damper.

```
>>> from sympy import symbols
>>> from sympy.physics.mechanics import Body, JointsMethod, _
↳ PrismaticJoint
>>> from sympy.physics.vector import dynamicsymbols
>>> c, k = symbols('c k')
>>> x, v = dynamicsymbols('x v')
>>> wall = Body('W')
>>> body = Body('B')
>>> J = PrismaticJoint('J', wall, body, coordinates=x, speeds=v)
>>> wall.apply_force(c*v*wall.x, reaction_body=body)
>>> wall.apply_force(k*x*wall.x, reaction_body=body)
>>> method = JointsMethod(wall, J)
>>> method.form_eoms()
Matrix([[-B_mass*Derivative(v(t), t) - c*v(t) - k*x(t)]])
>>> M = method.mass_matrix_full
>>> F = method.forcing_full
>>> rhs = M.LUsolve(F)
>>> rhs
Matrix([
[
v(t)],
[(-c*v(t) - k*x(t))/B_mass]])
```

## Notes

JointsMethod currently only works with systems that do not have any configuration or motion constraints.

## Attributes

|                  |                                                                                                                |
|------------------|----------------------------------------------------------------------------------------------------------------|
| q, u             | (iterable) Iterable of the generalized coordinates and speeds                                                  |
| bodies           | (iterable) Iterable of Body objects in the system.                                                             |
| loads            | (iterable) Iterable of (Point, vector) or (ReferenceFrame, vector) tuples describing the forces on the system. |
| mass_matrix      | (Matrix, shape(n, n)) The system's mass matrix                                                                 |
| forcing          | (Matrix, shape(n, 1)) The system's forcing vector                                                              |
| mass_matrix_full | (Matrix, shape(2*n, 2*n)) The "mass matrix" for the u's and q's                                                |
| forcing_full     | (Matrix, shape(2*n, 1)) The "forcing vector" for the u's and q's                                               |
| method           | (KanesMethod or Lagrange's method) Method's object.                                                            |
| kdes             | (iterable) Iterable of kde in they system.                                                                     |

### property bodies

List of bodies in they system.

### property forcing

The system's forcing vector.

**property forcing\_full**

The “forcing vector” for the u’s and q’s.

**form\_eoms**(*method*=<class 'sympy.physics.mechanics.kane.KanesMethod'>)

Method to form system’s equation of motions.

**Parameters**

**method** : Class

Class name of method.

**Returns**

Matrix

Vector of equations of motions.

**Examples**

This is a simple example for a one degree of freedom translational spring-mass-damper.

```
>>> from sympy import S, symbols
>>> from sympy.physics.mechanics import LagrangesMethod, dynamicsymbols, Body
>>> from sympy.physics.mechanics import PrismaticJoint, JointsMethod
>>> q = dynamicsymbols('q')
>>> qd = dynamicsymbols('q', 1)
>>> m, k, b = symbols('m k b')
>>> wall = Body('W')
>>> part = Body('P', mass=m)
>>> part.potential_energy = k * q**2 / S(2)
>>> J = PrismaticJoint('J', wall, part, coordinates=q, speeds=qd)
>>> wall.apply_force(b * qd * wall.x, reaction_body=part)
>>> method = JointsMethod(wall, J)
>>> method.form_eoms(LagrangesMethod)
Matrix([[b*Derivative(q(t), t) + k*q(t) + m*Derivative(q(t), (t, 2))]])
```

We can also solve for the states using the ‘rhs’ method.

```
>>> method.rhs()
Matrix([
[Derivative(q(t), t)],
[(-b*Derivative(q(t), t) - k*q(t))/m]])
```

**property kdes**

List of the generalized coordinates.

**property loads**

List of loads on the system.

**property mass\_matrix**

The system’s mass matrix.

**property mass\_matrix\_full**

The “mass matrix” for the u’s and q’s.

**property method**

Object of method used to form equations of systems.

**property q**

List of the generalized coordinates.

**rhs(*inv\_method=None*)**

Returns equations that can be solved numerically.

**Parameters**

**inv\_method** : str

The specific sympy inverse matrix calculation method to use. For a list of valid methods, see [\*inv\(\)\*](#) (page 1299)

**Returns**

Matrix

Numerically solveable equations.

**See also:**

[\*sympy.physics.mechanics.kane.KanesMethod.rhs\*](#) (page 1766)

KanesMethod's rhs function.

[\*sympy.physics.mechanics.lagrange.LagrangesMethod.rhs\*](#) (page 1769)

LagrangesMethod's rhs function.

**property u**

List of the generalized speeds.

**SymbolicSystem (Docstrings)****SymbolicSystem**

```
class sympy.physics.mechanics.system.SymbolicSystem(coord_states, right_hand_side,
 speeds=None,
 mass_matrix=None,
 coordinate_derivatives=None,
 alg_con=None,
 output_eqns={},
 coord_idx=None,
 speed_idx=None,
 bodies=None, loads=None)
```

SymbolicSystem is a class that contains all the information about a system in a symbolic format such as the equations of motions and the bodies and loads in the system.

There are three ways that the equations of motion can be described for Symbolic System:

**[1] Explicit form where the kinematics and dynamics are combined**

$x' = F_1(x, t, r, p)$

**[2] Implicit form where the kinematics and dynamics are combined**

$M_2(x, p) x' = F_2(x, t, r, p)$

**[3] Implicit form where the kinematics and dynamics are separate**

$M_3(q, p) u' = F_3(q, u, t, r, p)$   $q' = G(q, u, t, r, p)$

where

x : states, e.g. [q, u] t : time r : specified (exogenous) inputs p : constants q : generalized coordinates u : generalized speeds F\_1 : right hand side of the combined equations in explicit form F\_2 : right hand side of the combined equations in implicit form F\_3 : right hand side of the dynamical equations in implicit form M\_2 : mass matrix of the combined equations in implicit form M\_3 : mass matrix of the dynamical equations in implicit form G : right hand side of the kinematical differential equations

### Parameters

**coord\_states** : ordered iterable of functions of time

This input will either be a collection of the coordinates or states of the system depending on whether or not the speeds are also given. If speeds are specified this input will be assumed to be the coordinates otherwise this input will be assumed to be the states.

### right\_hand\_side

[Matrix] This variable is the right hand side of the equations of motion in any of the forms. The specific form will be assumed depending on whether a mass matrix or coordinate derivatives are given.

### speeds

[ordered iterable of functions of time, optional] This is a collection of the generalized speeds of the system. If given it will be assumed that the first argument (coord\_states) will represent the generalized coordinates of the system.

### mass\_matrix

[Matrix, optional] The matrix of the implicit forms of the equations of motion (forms [2] and [3]). The distinction between the forms is determined by whether or not the coordinate derivatives are passed in. If they are given form [3] will be assumed otherwise form [2] is assumed.

### coordinate\_derivatives

[Matrix, optional] The right hand side of the kinematical equations in explicit form. If given it will be assumed that the equations of motion are being entered in form [3].

### alg\_con

[Iterable, optional] The indexes of the rows in the equations of motion that contain algebraic constraints instead of differential equations. If the equations are input in form [3], it will be assumed the indexes are referencing the mass\_matrix/right\_hand\_side combination and not the coordinate\_derivatives.

### output\_eqns

[Dictionary, optional] Any output equations that are desired to be tracked are stored in a dictionary where the key corresponds to the name given for the specific equation and the value is the equation itself in symbolic form

### coord\_idx

[Iterable, optional] If coord\_states corresponds to the states rather than the coordinates this variable will tell SymbolicSystem which indexes of the states correspond to generalized coordinates.



**speed\_idx**

[Iterable, optional] If coord\_states corresponds to the states rather than the coordinates this variable will tell SymbolicSystem which indexes of the states correspond to generalized speeds.

**bodies**

[iterable of Body/Rigidbody objects, optional] Iterable containing the bodies of the system

**loads**

[iterable of load instances (described below), optional] Iterable containing the loads of the system where forces are given by (point of application, force vector) and torques are given by (reference frame acting upon, torque vector). Ex [(point, force), (ref\_frame, torque)]

**Example**

As a simple example, the dynamics of a simple pendulum will be input into a SymbolicSystem object manually. First some imports will be needed and then symbols will be set up for the length of the pendulum ( $l$ ), mass at the end of the pendulum ( $m$ ), and a constant for gravity ( $g$ ).

```
>>> from sympy import Matrix, sin, symbols
>>> from sympy.physics.mechanics import dynamicsymbols, SymbolicSystem
>>> l, m, g = symbols('l m g')
```

The system will be defined by an angle of  $\theta$  from the vertical and a generalized speed of  $\omega$  will be used where  $\omega = \dot{\theta}$ .

```
>>> theta, omega = dynamicsymbols('theta omega')
```

Now the equations of motion are ready to be formed and passed to the SymbolicSystem object.

```
>>> kin_explicit_rhs = Matrix([omega])
>>> dyn_implicit_mat = Matrix([l**2 * m])
>>> dyn_implicit_rhs = Matrix([-g * l * m * sin(theta)])
>>> symsystem = SymbolicSystem([theta], dyn_implicit_rhs, [omega],
... dyn_implicit_mat)
```

**Notes**

$m$  : number of generalized speeds  $n$  : number of generalized coordinates  $o$  : number of states

## Attributes

|                   |                                                                                                                                                                                                                                                                                                                                                                                                                    |
|-------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| co-or-di-nates    | (Matrix, shape(n, 1)) This is a matrix containing the generalized coordinates of the system                                                                                                                                                                                                                                                                                                                        |
| speeds            | (Matrix, shape(m, 1)) This is a matrix containing the generalized speeds of the system                                                                                                                                                                                                                                                                                                                             |
| states            | (Matrix, shape(o, 1)) This is a matrix containing the state variables of the system                                                                                                                                                                                                                                                                                                                                |
| alg_con           | (List) This list contains the indices of the algebraic constraints in the combined equations of motion. The presence of these constraints requires that a DAE solver be used instead of an ODE solver. If the system is given in form [3] the alg_con variable will be adjusted such that it is a representation of the combined kinematics and dynamics thus make sure it always matches the mass matrix entered. |
| dyn_implicit      | (Matrix, shape(m, m)) This is the M matrix in form [3] of the equations of motion (the mass matrix or generalized inertia matrix of the dynamical equations of motion in implicit form).                                                                                                                                                                                                                           |
| dyn_implicit_rhs  | (Matrix, shape(m, 1)) This is the F vector in form [3] of the equations of motion (the right hand side of the dynamical equations of motion in implicit form).                                                                                                                                                                                                                                                     |
| comb_implicit     | (Matrix, shape(o, o)) This is the M matrix in form [2] of the equations of motion. This matrix contains a block diagonal structure where the top left block (the first rows) represent the matrix in the implicit form of the kinematical equations and the bottom right block (the last rows) represent the matrix in the implicit form of the dynamical equations.                                               |
| comb_implicit_rhs | (Matrix, shape(o, 1)) This is the F vector in form [2] of the equations of motion. The top part of the vector represents the right hand side of the implicit form of the kinematical equations and the bottom of the vector represents the right hand side of the implicit form of the dynamical equations of motion.                                                                                              |
| comb_explicit_rhs | (Matrix, shape(o, 1)) This vector represents the right hand side of the combined equations of motion in explicit form (form [1] from above).                                                                                                                                                                                                                                                                       |
| kin_explicit_rhs  | (Matrix, shape(m, 1)) This is the right hand side of the explicit form of the kinematical equations of motion as can be seen in form [3] (the G matrix).                                                                                                                                                                                                                                                           |
| output_eqs        | (Dictionary) If output equations were given they are stored in a dictionary where the key corresponds to the name given for the specific equation and the value is the equation itself in symbolic form                                                                                                                                                                                                            |
| bodies            | (Tuple) If the bodies in the system were given they are stored in a tuple for future access                                                                                                                                                                                                                                                                                                                        |
| loads             | (Tuple) If the loads in the system were given they are stored in a tuple for future access. This includes forces and torques where forces are given by (point of application, force vector) and torques are given by (reference frame acted upon, torque vector).                                                                                                                                                  |

### property alg\_con

Returns a list with the indices of the rows containing algebraic constraints in the combined form of the equations of motion

### property bodies

Returns the bodies in the system

### property comb\_explicit\_rhs

Returns the right hand side of the equations of motion in explicit form,  $\mathbf{x}' = \mathbf{F}$ , where the kinematical equations are included

**property comb\_implicit\_mat**

Returns the matrix,  $M$ , corresponding to the equations of motion in implicit form (form [2]),  $M \dot{x} = F$ , where the kinematical equations are included

**property comb\_implicit\_rhs**

Returns the column matrix,  $F$ , corresponding to the equations of motion in implicit form (form [2]),  $M \dot{x} = F$ , where the kinematical equations are included

**compute\_explicit\_form()**

If the explicit right hand side of the combined equations of motion is to provided upon initialization, this method will calculate it. This calculation can potentially take awhile to compute.

**constant\_symbols()**

Returns a column matrix containing all of the symbols in the system that do not depend on time

**property coordinates**

Returns the column matrix of the generalized coordinates

**property dyn\_implicit\_mat**

Returns the matrix,  $M$ , corresponding to the dynamic equations in implicit form,  $M \dot{x} = F$ , where the kinematical equations are not included

**property dyn\_implicit\_rhs**

Returns the column matrix,  $F$ , corresponding to the dynamic equations in implicit form,  $M \dot{x} = F$ , where the kinematical equations are not included

**dynamic\_symbols()**

Returns a column matrix containing all of the symbols in the system that depend on time

**property kin\_explicit\_rhs**

Returns the right hand side of the kinematical equations in explicit form,  $q' = G$

**property loads**

Returns the loads in the system

**property speeds**

Returns the column matrix of generalized speeds

**property states**

Returns the column matrix of the state variables

**Linearization (Docstrings)****Linearizer**

```
class sympy.physics.mechanics.linearize.Linearizer(f_0, f_1, f_2, f_3, f_4, f_c, f_v,
 f_a, q, u, q_i=None, q_d=None,
 u_i=None, u_d=None, r=None,
 lams=None)
```

This object holds the general model form for a dynamic system. This model is used for computing the linearized form of the system, while properly dealing with constraints leading to dependent coordinates and speeds.

## Attributes

|                                                                     |                                                                                   |
|---------------------------------------------------------------------|-----------------------------------------------------------------------------------|
| <code>f_0, f_1, f_2, f_3, f_4,</code><br><code>f_c, f_v, f_a</code> | (Matrix) Matrices holding the general system form.                                |
| <code>q, u, r</code>                                                | (Matrix) Matrices holding the generalized coordinates, speeds, and input vectors. |
| <code>q_i, u_i</code>                                               | (Matrix) Matrices of the independent generalized coordinates and speeds.          |
| <code>q_d, u_d</code>                                               | (Matrix) Matrices of the dependent generalized coordinates and speeds.            |
| <code>perm_mat</code>                                               | (Matrix) Permutation matrix such that $[q\_ind, u\_ind]^T = perm\_mat*[q, u]^T$   |

**linearize**(*op\_point=None, A\_and\_B=False, simplify=False*)

Linearize the system about the operating point. Note that `q_op`, `u_op`, `qd_op`, `ud_op` must satisfy the equations of motion. These may be either symbolic or numeric.

### Parameters

**op\_point** : dict or iterable of dicts, optional

Dictionary or iterable of dictionaries containing the operating point conditions. These will be substituted in to the linearized system before the linearization is complete. Leave blank if you want a completely symbolic form. Note that any reduction in symbols (whether substituted for numbers or expressions with a common parameter) will result in faster runtime.

**A\_and\_B** : bool, optional

If `A_and_B=False` (default),  $(M, A, B)$  is returned for forming  $[M]*[q, u]^T = [A]*[q\_ind, u\_ind]^T + [B]r$ . If `A_and_B=True`,  $(A, B)$  is returned for forming  $dx = [A]x + [B]r$ , where  $x = [q\_ind, u\_ind]^T$ .

**simplify** : bool, optional

Determines if returned values are simplified before return. For large expressions this may be time consuming. Default is `False`.

## Potential Issues

Note that the process of solving with `A_and_B=True` is computationally intensive if there are many symbolic parameters. For this reason, it may be more desirable to use the default `A_and_B=False`, returning `M`, `A`, and `B`. More values may then be substituted in to these matrices later on. The state space form can then be found as  $A = P.T*M.LUsolve(A)$ ,  $B = P.T*M.LUsolve(B)$ , where  $P = Linearizer.perm\_mat$ .

## Expression Manipulation (Docstrings)

### msubs

`sympy.physics.mechanics.msubs(expr, *sub_dicts, smart=False, **kwargs)`

A custom subs for use on expressions derived in `physics.mechanics`.

Traverses the expression tree once, performing the subs found in `sub_dicts`. Terms inside Derivative expressions are ignored:

### Examples

```
>>> from sympy.physics.mechanics import dynamicsymbols, msubs
>>> x = dynamicsymbols('x')
>>> msubs(x.diff() + x, {x: 1})
Derivative(x(t), t) + 1
```

Note that `sub_dicts` can be a single dictionary, or several dictionaries:

```
>>> x, y, z = dynamicsymbols('x, y, z')
>>> sub1 = {x: 1, y: 2}
>>> sub2 = {z: 3, x.diff(): 4}
>>> msubs(x.diff() + x + y + z, sub1, sub2)
10
```

If `smart=True` (default `False`), also checks for conditions that may result in `nan`, but if simplified would yield a valid expression. For example:

```
>>> from sympy import sin, tan
>>> (sin(x)/tan(x)).subs(x, 0)
nan
>>> msubs(sin(x)/tan(x), {x: 0}, smart=True)
1
```

It does this by first replacing all `tan` with `sin/cos`. Then each node is traversed. If the node is a fraction, `subs` is first evaluated on the denominator. If this results in 0, simplification of the entire fraction is attempted. Using this selective simplification, only subexpressions that result in `1/0` are targeted, resulting in faster performance.

### find\_dynamicsymbols

`sympy.physics.mechanics.find_dynamicsymbols(expression, exclude=None, reference_frame=None)`

Find all `dynamicsymbols` in expression.

#### Parameters

**expression** : SymPy expression

**exclude** : iterable of `dynamicsymbols`, optional

**reference\_frame** : `ReferenceFrame`, optional

The frame with respect to which the dynamic symbols of the given vector is to be determined.

### Explanation

If the optional `exclude` kwarg is used, only `dynamicsymbols` not in the iterable `exclude` are returned. If we intend to apply this function on a vector, the optional `reference_frame` is also used to inform about the corresponding frame with respect to which the dynamic symbols of the given vector is to be determined.

### Examples

```
>>> from sympy.physics.mechanics import dynamicsymbols, find_
 ↪dynamicsymbols
>>> from sympy.physics.mechanics import ReferenceFrame
>>> x, y = dynamicsymbols('x, y')
>>> expr = x + x.diff()*y
>>> find_dynamicsymbols(expr)
{x(t), y(t), Derivative(x(t), t)}
>>> find_dynamicsymbols(expr, exclude=[x, y])
{Derivative(x(t), t)}
>>> a, b, c = dynamicsymbols('a, b, c')
>>> A = ReferenceFrame('A')
>>> v = a * A.x + b * A.y + c * A.z
>>> find_dynamicsymbols(v, reference_frame=A)
{a(t), b(t), c(t)}
```

## Printing (Docstrings)

### `mechanics_printing`

This function is the same as [`sympy.physics.vector.printing.init\_vprinting`](#) (page 1662).

### `mprint`

This function is the same as [`sympy.physics.vector.printing.vprint`](#) (page 1664).

### `mpprint`

This function is the same as [`sympy.physics.vector.printing.vpprint`](#) (page 1665).

## mlatex

This function is the same as `sympy.physics.vector.printing.vlatex` (page 1665).

## Quantum Mechanics

### Abstract

Contains Docstrings of Physics-Quantum module

## Quantum Functions

### Anticommutator

The anti-commutator:  $\{A, B\} = A*B + B*A$ .

**class** `sympy.physics.quantum.anticommutator.AntiCommutator(A, B)`

The standard anticommutator, in an unevaluated state.

#### Parameters

**A** : Expr

The first argument of the anticommutator  $\{A, B\}$ .

**B** : Expr

The second argument of the anticommutator  $\{A, B\}$ .

### Explanation

Evaluating an anticommutator is defined [R669] as:  $\{A, B\} = A*B + B*A$ . This class returns the anticommutator in an unevaluated form. To evaluate the anticommutator, use the `.doit()` method.

Canonical ordering of an anticommutator is  $\{A, B\}$  for  $A < B$ . The arguments of the anticommutator are put into canonical order using `__cmp__`. If  $B < A$ , then  $\{A, B\}$  is returned as  $\{B, A\}$ .

### Examples

```

>>> from sympy import symbols
>>> from sympy.physics.quantum import AntiCommutator
>>> from sympy.physics.quantum import Operator, Dagger
>>> x, y = symbols('x,y')
>>> A = Operator('A')
>>> B = Operator('B')

```

Create an anticommutator and use `doit()` to multiply them out.

```
>>> ac = AntiCommutator(A,B); ac
{A,B}
>>> ac.doit()
A*B + B*A
```

The commutator orders its arguments in canonical order:

```
>>> ac = AntiCommutator(B,A); ac
{A,B}
```

Commutative constants are factored out:

```
>>> AntiCommutator(3*x*A,x*y*B)
3*x**2*y*{A,B}
```

Adjoint operations applied to the anticommutator are properly applied to the arguments:

```
>>> Dagger(AntiCommutator(A,B))
{Dagger(A),Dagger(B)}
```

## References

[R669]

**doit(\*\*hints)**

Evaluate anticommutator

## Clebsch-Gordan Coefficients

Clebsch-Gordan Coefficients.

**class** sympy.physics.quantum.cg.**CG**(*j1, m1, j2, m2, j3, m3*)

Class for Clebsch-Gordan coefficient.

### Parameters

**j1, m1, j2, m2** : Number, Symbol

Angular momenta of states 1 and 2.

**j3, m3**: Number, Symbol

Total angular momentum of the coupled system.

## Explanation

Clebsch-Gordan coefficients describe the angular momentum coupling between two systems. The coefficients give the expansion of a coupled total angular momentum state and an uncoupled tensor product state. The Clebsch-Gordan coefficients are defined as [R670]:

$$C_{j_1, m_1, j_2, m_2}^{j_3, m_3} = \langle j_1, m_1; j_2, m_2 | j_3, m_3 \rangle$$



## Examples

Define a Clebsch-Gordan coefficient and evaluate its value

```
>>> from sympy.physics.quantum.cg import CG
>>> from sympy import S
>>> cg = CG(S(3)/2, S(3)/2, S(1)/2, -S(1)/2, 1, 1)
>>> cg
CG(3/2, 3/2, 1/2, -1/2, 1, 1)
>>> cg.doit()
sqrt(3)/2
>>> CG(j1=S(1)/2, m1=-S(1)/2, j2=S(1)/2, m2=+S(1)/2, j3=1, m3=0).doit()
sqrt(2)/2
```

Compare [R671].

**See also:**

***Wigner3j*** (page 1793)

Wigner-3j symbols

## References

[R670], [R671]

**class** sympy.physics.quantum.cg.**Wigner3j**(*j1, m1, j2, m2, j3, m3*)

Class for the Wigner-3j symbols.

### Parameters

**j1, m1, j2, m2, j3, m3** : Number, Symbol

Terms determining the angular momentum of coupled angular momentum systems.

## Explanation

Wigner 3j-symbols are coefficients determined by the coupling of two angular momenta. When created, they are expressed as symbolic quantities that, for numerical parameters, can be evaluated using the `.doit()` method [R672].

## Examples

Declare a Wigner-3j coefficient and calculate its value

```
>>> from sympy.physics.quantum.cg import Wigner3j
>>> w3j = Wigner3j(6,0,4,0,2,0)
>>> w3j
Wigner3j(6, 0, 4, 0, 2, 0)
>>> w3j.doit()
sqrt(715)/143
```

**See also:**

**CG (page 1792)**

Clebsch-Gordan coefficients

**References**[\[R672\]](#)**class** sympy.physics.quantum.cg.**Wigner6j**(*j1, j2, j12, j3, j, j23*)

Class for the Wigner-6j symbols

**See also:****[Wigner3j \(page 1793\)](#)**

Wigner-3j symbols

**class** sympy.physics.quantum.cg.**Wigner9j**(*j1, j2, j12, j3, j4, j34, j13, j24, j*)

Class for the Wigner-9j symbols

**See also:****[Wigner3j \(page 1793\)](#)**

Wigner-3j symbols

sympy.physics.quantum.cg.**cg\_simp**(*e*)

Simplify and combine CG coefficients.

**Explanation**

This function uses various symmetry and properties of sums and products of Clebsch-Gordan coefficients to simplify statements involving these terms [\[R673\]](#).

**Examples**

Simplify the sum over CG(*a, alpha, 0, 0, a, alpha*) for all alpha to  $2*a+1$

```
>>> from sympy.physics.quantum.cg import CG, cg_simp
>>> a = CG(1, 1, 0, 0, 1, 1)
>>> b = CG(1, 0, 0, 0, 1, 0)
>>> c = CG(1, -1, 0, 0, 1, -1)
>>> cg_simp(a+b+c)
3
```

**See also:****[CG \(page 1792\)](#)**

Clebsch-Gordan coefficients

## References

[R673]

## Commutator

The commutator:  $[A, B] = A*B - B*A$ .

**class** `sympy.physics.quantum.commutator.Commutator(A, B)`

The standard commutator, in an unevaluated state.

### Parameters

**A** : Expr

The first argument of the commutator  $[A, B]$ .

**B** : Expr

The second argument of the commutator  $[A, B]$ .

## Explanation

Evaluating a commutator is defined [R674] as:  $[A, B] = A*B - B*A$ . This class returns the commutator in an unevaluated form. To evaluate the commutator, use the `.doit()` method.

Canonical ordering of a commutator is  $[A, B]$  for  $A < B$ . The arguments of the commutator are put into canonical order using `__cmp__`. If  $B < A$ , then  $[B, A]$  is returned as  $-[A, B]$ .

## Examples

```
>>> from sympy.physics.quantum import Commutator, Dagger, Operator
>>> from sympy.abc import x, y
>>> A = Operator('A')
>>> B = Operator('B')
>>> C = Operator('C')
```

Create a commutator and use `.doit()` to evaluate it:

```
>>> comm = Commutator(A, B)
>>> comm
[A,B]
>>> comm.doit()
A*B - B*A
```

The commutator orders its arguments in canonical order:

```
>>> comm = Commutator(B, A); comm
-[A,B]
```

Commutative constants are factored out:

```
>>> Commutator(3*x*A, x*y*B)
3*x**2*y*[A,B]
```

Using `.expand(commutator=True)`, the standard commutator expansion rules can be applied:

```
>>> Commutator(A+B, C).expand(commutator=True)
[A,C] + [B,C]
>>> Commutator(A, B+C).expand(commutator=True)
[A,B] + [A,C]
>>> Commutator(A*B, C).expand(commutator=True)
[A,C]*B + A*[B,C]
>>> Commutator(A, B*C).expand(commutator=True)
[A,B]*C + B*[A,C]
```

Adjoint operations applied to the commutator are properly applied to the arguments:

```
>>> Dagger(Commutator(A, B))
-[Dagger(A),Dagger(B)]
```

## References

[R674]

**doit(\*\*hints)**

Evaluate commutator

## Constants

Constants (like `hbar`) related to quantum mechanics.

**class** `sympy.physics.quantum.constants.HBar`

Reduced Planck's constant in numerical and symbolic form [R675].

## Examples

```
>>> from sympy.physics.quantum.constants import hbar
>>> hbar.evalf()
1.05457162000000e-34
```

## References

[R675]

## Dagger

Hermitian conjugation.

**class** sympy.physics.quantum.dagger.Dagger(arg)

General Hermitian conjugate operation.

### Parameters

**arg** : Expr

The SymPy expression that we want to take the dagger of.

## Explanation

Take the Hermetian conjugate of an argument [R676]. For matrices this operation is equivalent to transpose and complex conjugate [R677].

## Examples

Dagging various quantum objects:

```
>>> from sympy.physics.quantum.dagger import Dagger
>>> from sympy.physics.quantum.state import Ket, Bra
>>> from sympy.physics.quantum.operator import Operator
>>> Dagger(Ket('psi'))
<psi|
>>> Dagger(Bra('phi'))
|phi>
>>> Dagger(Operator('A'))
Dagger(A)
```

Inner and outer products:

```
>>> from sympy.physics.quantum import InnerProduct, OuterProduct
>>> Dagger(InnerProduct(Bra('a'), Ket('b')))
<b|a>
>>> Dagger(OuterProduct(Ket('a'), Bra('b')))
|b><a|
```

Powers, sums and products:

```
>>> A = Operator('A')
>>> B = Operator('B')
>>> Dagger(A*B)
Dagger(B)*Dagger(A)
>>> Dagger(A+B)
Dagger(A) + Dagger(B)
```

(continues on next page)

(continued from previous page)

```
>>> Dagger(A**2)
Dagger(A)**2
```

Dagger also seamlessly handles complex numbers and matrices:

```
>>> from sympy import Matrix, I
>>> m = Matrix([[1,I],[2,I]])
>>> m
Matrix([
[1, I],
[2, I]])
>>> Dagger(m)
Matrix([
[1, 2],
[-I, -I]])
```

## References

[R676], [R677]

## Inner Product

Symbolic inner product.

**class** sympy.physics.quantum.innerproduct.**InnerProduct**(*bra*, *ket*)

An unevaluated inner product between a Bra and a Ket [1].

### Parameters

**bra** : BraBase or subclass

The bra on the left side of the inner product.

**ket** : KetBase or subclass

The ket on the right side of the inner product.

## Examples

Create an InnerProduct and check its properties:

```
>>> from sympy.physics.quantum import Bra, Ket
>>> b = Bra('b')
>>> k = Ket('k')
>>> ip = b*k
>>> ip
<b|k>
>>> ip.bra
<b|
>>> ip.ket
|k>
```

In simple products of kets and bras inner products will be automatically identified and created:

```
>>> b*k
<b|k>
```

But in more complex expressions, there is ambiguity in whether inner or outer products should be created:

```
>>> k*b*k*b
|k><b|*|k>*<b|
```

A user can force the creation of a inner products in a complex expression by using parentheses to group the bra and ket:

```
>>> k*(b*k)*b
<b|k>*<b|k>*<b|
```

Notice how the inner product  $\langle b|k\rangle$  moved to the left of the expression because inner products are commutative complex numbers.

## References

[R683]

## Tensor Product

Abstract tensor product.

**class** sympy.physics.quantum.tensorproduct.**TensorProduct**(\*args)

The tensor product of two or more arguments.

For matrices, this uses `matrix_tensor_product` to compute the Kronecker or tensor product matrix. For other objects a symbolic `TensorProduct` instance is returned. The tensor product is a non-commutative multiplication that is used primarily with operators and states in quantum mechanics.

Currently, the tensor product distinguishes between commutative and non-commutative arguments. Commutative arguments are assumed to be scalars and are pulled out in front of the `TensorProduct`. Non-commutative arguments remain in the resulting `TensorProduct`.

### Parameters

**args** : tuple

A sequence of the objects to take the tensor product of.

## Examples

Start with a simple tensor product of SymPy matrices:

```
>>> from sympy import Matrix
>>> from sympy.physics.quantum import TensorProduct

>>> m1 = Matrix([[1,2],[3,4]])
>>> m2 = Matrix([[1,0],[0,1]])
>>> TensorProduct(m1, m2)
Matrix([
[1, 0, 2, 0],
[0, 1, 0, 2],
[3, 0, 4, 0],
[0, 3, 0, 4]])
>>> TensorProduct(m2, m1)
Matrix([
[1, 2, 0, 0],
[3, 4, 0, 0],
[0, 0, 1, 2],
[0, 0, 3, 4]])
```

We can also construct tensor products of non-commutative symbols:

```
>>> from sympy import Symbol
>>> A = Symbol('A',commutative=False)
>>> B = Symbol('B',commutative=False)
>>> tp = TensorProduct(A, B)
>>> tp
AxB
```

We can take the dagger of a tensor product (note the order does NOT reverse like the dagger of a normal product):

```
>>> from sympy.physics.quantum import Dagger
>>> Dagger(tp)
Dagger(A)xDagger(B)
```

Expand can be used to distribute a tensor product across addition:

```
>>> C = Symbol('C',commutative=False)
>>> tp = TensorProduct(A+B,C)
>>> tp
(A + B)xC
>>> tp.expand(tensorproduct=True)
AxC + BxC
```

`sympy.physics.quantum.tensorproduct.tensor_product_simp(e, **hints)`

Try to simplify and combine TensorProducts.

In general this will try to pull expressions inside of TensorProducts. It currently only works for relatively simple cases where the products have only scalars, raw TensorProducts, not Add, Pow, Commutators of TensorProducts. It is best to see what it does by showing examples.



## Examples

```
>>> from sympy.physics.quantum import tensor_product_simp
>>> from sympy.physics.quantum import TensorProduct
>>> from sympy import Symbol
>>> A = Symbol('A', commutative=False)
>>> B = Symbol('B', commutative=False)
>>> C = Symbol('C', commutative=False)
>>> D = Symbol('D', commutative=False)
```

First see what happens to products of tensor products:

```
>>> e = TensorProduct(A,B)*TensorProduct(C,D)
>>> e
AxB*CxD
>>> tensor_product_simp(e)
(A*C)x(B*D)
```

This is the core logic of this function, and it works inside, powers, sums, commutators and anticommutators as well:

```
>>> tensor_product_simp(e**2)
(A*C)x(B*D)**2
```

## States and Operators

### Cartesian Operators and States

Operators and states for 1D cartesian position and momentum.

TODO:

- Add 3D classes to mappings in operatorset.py

```
class sympy.physics.quantum.cartesian.PositionBra3D(*args, **kwargs)
 3D cartesian position eigenbra
```

```
class sympy.physics.quantum.cartesian.PositionKet3D(*args, **kwargs)
 3D cartesian position eigenket
```

```
class sympy.physics.quantum.cartesian.PositionState3D(*args, **kwargs)
 Base class for 3D cartesian position eigenstates
```

```
 property position_x
 The x coordinate of the state
```

```
 property position_y
 The y coordinate of the state
```

```
 property position_z
 The z coordinate of the state
```

```
class sympy.physics.quantum.cartesian.PxBra(*args, **kwargs)
 1D cartesian momentum eigenbra.
```

**property momentum**

The momentum of the state.

```
class sympy.physics.quantum.cartesian.PxKet(*args, **kwargs)
```

1D cartesian momentum eigenket.

**property momentum**

The momentum of the state.

```
class sympy.physics.quantum.cartesian.PxOp(*args, **kwargs)
```

1D cartesian momentum operator.

```
class sympy.physics.quantum.cartesian.XBra(*args, **kwargs)
```

1D cartesian position eigenbra.

**property position**

The position of the state.

```
class sympy.physics.quantum.cartesian.XKet(*args, **kwargs)
```

1D cartesian position eigenket.

**property position**

The position of the state.

```
class sympy.physics.quantum.cartesian.XOp(*args, **kwargs)
```

1D cartesian position operator.

```
class sympy.physics.quantum.cartesian.YOp(*args, **kwargs)
```

Y cartesian coordinate operator (for 2D or 3D systems)

```
class sympy.physics.quantum.cartesian.ZOp(*args, **kwargs)
```

Z cartesian coordinate operator (for 3D systems)

## Hilbert Space

Hilbert spaces for quantum mechanics.

Authors: \* Brian Granger \* Matt Curry

```
class sympy.physics.quantum.hilbert.ComplexSpace(dimension)
```

Finite dimensional Hilbert space of complex vectors.

The elements of this Hilbert space are n-dimensional complex valued vectors with the usual inner product that takes the complex conjugate of the vector on the right.

A classic example of this type of Hilbert space is spin-1/2, which is `ComplexSpace(2)`. Generalizing to spin-s, the space is `ComplexSpace(2*s+1)`. Quantum computing with N qubits is done with the direct product space `ComplexSpace(2)**N`.

## Examples

```
>>> from sympy import symbols
>>> from sympy.physics.quantum.hilbert import ComplexSpace
>>> c1 = ComplexSpace(2)
>>> c1
C(2)
>>> c1.dimension
2
```

```
>>> n = symbols('n')
>>> c2 = ComplexSpace(n)
>>> c2
C(n)
>>> c2.dimension
n
```

**class** `sympy.physics.quantum.hilbert.DirectSumHilbertSpace(*args)`

A direct sum of Hilbert spaces [R678].

This class uses the + operator to represent direct sums between different Hilbert spaces.

A `DirectSumHilbertSpace` object takes in an arbitrary number of `HilbertSpace` objects as its arguments. Also, addition of `HilbertSpace` objects will automatically return a direct sum object.

## Examples

```
>>> from sympy.physics.quantum.hilbert import ComplexSpace, FockSpace
```

```
>>> c = ComplexSpace(2)
>>> f = FockSpace()
>>> hs = c+f
>>> hs
C(2)+F
>>> hs.dimension
oo
>>> list(hs.spaces)
[C(2), F]
```

## References

[R678]

**classmethod** `eval(args)`

Evaluates the direct product.

**property** `spaces`

A tuple of the Hilbert spaces in this direct sum.

**class** `sympy.physics.quantum.hilbert.FockSpace`

The Hilbert space for second quantization.

Technically, this Hilbert space is a infinite direct sum of direct products of single particle Hilbert spaces [R679]. This is a mess, so we have a class to represent it directly.

**Examples**

```
>>> from sympy.physics.quantum.hilbert import FockSpace
>>> hs = FockSpace()
>>> hs
F
>>> hs.dimension
oo
```

**References**

[R679]

**class** `sympy.physics.quantum.hilbert.HilbertSpace`

An abstract Hilbert space for quantum mechanics.

In short, a Hilbert space is an abstract vector space that is complete with inner products defined [R680].

**Examples**

```
>>> from sympy.physics.quantum.hilbert import HilbertSpace
>>> hs = HilbertSpace()
>>> hs
H
```

**References**

[R680]

**property** `dimension`

Return the Hilbert dimension of the space.

**class** `sympy.physics.quantum.hilbert.L2(interval)`

The Hilbert space of square integrable functions on an interval.

An L2 object takes in a single SymPy Interval argument which represents the interval its functions (vectors) are defined on.

## Examples

```
>>> from sympy import Interval, oo
>>> from sympy.physics.quantum.hilbert import L2
>>> hs = L2(Interval(0,oo))
>>> hs
L2(Interval(0, oo))
>>> hs.dimension
oo
>>> hs.interval
Interval(0, oo)
```

**class** sympy.physics.quantum.hilbert.TensorPowerHilbertSpace(\*args)

An exponentiated Hilbert space [R681].

Tensor powers (repeated tensor products) are represented by the operator `**`. Identical Hilbert spaces that are multiplied together will be automatically combined into a single tensor power object.

Any Hilbert space, product, or sum may be raised to a tensor power. The `TensorPowerHilbertSpace` takes two arguments: the Hilbert space; and the tensor power (number).

## Examples

```
>>> from sympy.physics.quantum.hilbert import ComplexSpace, FockSpace
>>> from sympy import symbols
```

```
>>> n = symbols('n')
>>> c = ComplexSpace(2)
>>> hs = c**n
>>> hs
C(2)**n
>>> hs.dimension
2**n
```

```
>>> c = ComplexSpace(2)
>>> c*c
C(2)**2
>>> f = FockSpace()
>>> c*f*f
C(2)*F**2
```

## References

[R681]

**class** `sympy.physics.quantum.hilbert.TensorProductHilbertSpace(*args)`

A tensor product of Hilbert spaces [R682].

The tensor product between Hilbert spaces is represented by the operator `*`. Products of the same Hilbert space will be combined into tensor powers.

A `TensorProductHilbertSpace` object takes in an arbitrary number of `HilbertSpace` objects as its arguments. In addition, multiplication of `HilbertSpace` objects will automatically return this tensor product object.

## Examples

```
>>> from sympy.physics.quantum.hilbert import ComplexSpace, FockSpace
>>> from sympy import symbols
```

```
>>> c = ComplexSpace(2)
>>> f = FockSpace()
>>> hs = c*f
>>> hs
C(2)*F
>>> hs.dimension
oo
>>> hs.spaces
(C(2), F)
```

```
>>> c1 = ComplexSpace(2)
>>> n = symbols('n')
>>> c2 = ComplexSpace(n)
>>> hs = c1*c2
>>> hs
C(2)*C(n)
>>> hs.dimension
2*n
```

## References

[R682]

**classmethod** `eval(args)`

Evaluates the direct product.

**property** `spaces`

A tuple of the Hilbert spaces in this tensor product.

## Operator

Quantum mechanical operators.

TODO:

- Fix early 0 in `apply_operators`.
- Debug and test `apply_operators`.
- Get cse working with classes in this file.
- Doctests and documentation of special methods for `InnerProduct`, `Commutator`, `Anti-Commutator`, `represent`, `apply_operators`.

**class** `sympy.physics.quantum.operator.DifferentialOperator(*args, **kwargs)`

An operator for representing the differential operator, i.e.  $d/dx$

It is initialized by passing two arguments. The first is an arbitrary expression that involves a function, such as `Derivative(f(x), x)`. The second is the function (e.g. `f(x)`) which we are to replace with the Wavefunction that this `DifferentialOperator` is applied to.

### Parameters

**expr** : Expr

The arbitrary expression which the appropriate Wavefunction is to be substituted into

**func** : Expr

A function (e.g. `f(x)`) which is to be replaced with the appropriate Wavefunction when this `DifferentialOperator` is applied

## Examples

You can define a completely arbitrary expression and specify where the Wavefunction is to be substituted

```
>>> from sympy import Derivative, Function, Symbol
>>> from sympy.physics.quantum.operator import DifferentialOperator
>>> from sympy.physics.quantum.state import Wavefunction
>>> from sympy.physics.quantum.qapply import qapply
>>> f = Function('f')
>>> x = Symbol('x')
>>> d = DifferentialOperator(1/x*Derivative(f(x), x), f(x))
>>> w = Wavefunction(x**2, x)
>>> d.function
f(x)
>>> d.variables
(x,)
>>> qapply(d*w)
Wavefunction(2, x)
```

### property `expr`

Returns the arbitrary expression which is to have the Wavefunction substituted into it

### Examples

```
>>> from sympy.physics.quantum.operator import DifferentialOperator
>>> from sympy import Function, Symbol, Derivative
>>> x = Symbol('x')
>>> f = Function('f')
>>> d = DifferentialOperator(Derivative(f(x), x), f(x))
>>> d.expr
Derivative(f(x), x)
>>> y = Symbol('y')
>>> d = DifferentialOperator(Derivative(f(x, y), x) +
... Derivative(f(x, y), y), f(x, y))
>>> d.expr
Derivative(f(x, y), x) + Derivative(f(x, y), y)
```

#### property `free_symbols`

Return the free symbols of the expression.

#### property `function`

Returns the function which is to be replaced with the Wavefunction

### Examples

```
>>> from sympy.physics.quantum.operator import DifferentialOperator
>>> from sympy import Function, Symbol, Derivative
>>> x = Symbol('x')
>>> f = Function('f')
>>> d = DifferentialOperator(Derivative(f(x), x), f(x))
>>> d.function
f(x)
>>> y = Symbol('y')
>>> d = DifferentialOperator(Derivative(f(x, y), x) +
... Derivative(f(x, y), y), f(x, y))
>>> d.function
f(x, y)
```

#### property `variables`

Returns the variables with which the function in the specified arbitrary expression is evaluated

### Examples

```
>>> from sympy.physics.quantum.operator import DifferentialOperator
>>> from sympy import Symbol, Function, Derivative
>>> x = Symbol('x')
>>> f = Function('f')
>>> d = DifferentialOperator(1/x*Derivative(f(x), x), f(x))
>>> d.variables
(x,)
>>> y = Symbol('y')
```

(continues on next page)



(continued from previous page)

```
>>> d = DifferentialOperator(Derivative(f(x, y), x) +
... Derivative(f(x, y), y), f(x, y))
>>> d.variables
(x, y)
```

**class** `sympy.physics.quantum.operator.HermitianOperator(*args, **kwargs)`

A Hermitian operator that satisfies  $H == \text{Dagger}(H)$ .

#### Parameters

**args** : tuple

The list of numbers or parameters that uniquely specify the operator.  
For time-dependent operators, this will include the time.

#### Examples

```
>>> from sympy.physics.quantum import Dagger, HermitianOperator
>>> H = HermitianOperator('H')
>>> Dagger(H)
H
```

**class** `sympy.physics.quantum.operator.IdentityOperator(*args, **kwargs)`

An identity operator  $I$  that satisfies  $op * I == I * op == op$  for any operator  $op$ .

#### Parameters

**N** : Integer

Optional parameter that specifies the dimension of the Hilbert space of operator. This is used when generating a matrix representation.

#### Examples

```
>>> from sympy.physics.quantum import IdentityOperator
>>> IdentityOperator()
I
```

**class** `sympy.physics.quantum.operator.Operator(*args, **kwargs)`

Base class for non-commuting quantum operators.

An operator maps between quantum states [R684]. In quantum mechanics, observables (including, but not limited to, measured physical values) are represented as Hermitian operators [R685].

#### Parameters

**args** : tuple

The list of numbers or parameters that uniquely specify the operator.  
For time-dependent operators, this will include the time.

## Examples

Create an operator and examine its attributes:

```
>>> from sympy.physics.quantum import Operator
>>> from sympy import I
>>> A = Operator('A')
>>> A
A
>>> A.hilbert_space
H
>>> A.label
(A,)
>>> A.is_commutative
False
```

Create another operator and do some arithmetic operations:

```
>>> B = Operator('B')
>>> C = 2*A*A + I*B
>>> C
2*A**2 + I*B
```

Operators do not commute:

```
>>> A.is_commutative
False
>>> B.is_commutative
False
>>> A*B == B*A
False
```

Polynomials of operators respect the commutation properties:

```
>>> e = (A+B)**3
>>> e.expand()
A*B*A + A*B**2 + A**2*B + A**3 + B*A*B + B*A**2 + B**2*A + B**3
```

Operator inverses are handle symbolically:

```
>>> A.inv()
A**(-1)
>>> A*A.inv()
1
```

## References

[R684], [R685]

**class** sympy.physics.quantum.operator.OuterProduct(\*args, \*\*old\_assumptions)

An unevaluated outer product between a ket and bra.

This constructs an outer product between any subclass of KetBase and BraBase as  $|a\rangle\langle b|$ . An OuterProduct inherits from Operator as they act as operators in quantum expressions. For reference see [R686].

### Parameters

**ket** : KetBase

The ket on the left side of the outer product.

**bra** : BraBase

The bra on the right side of the outer product.

## Examples

Create a simple outer product by hand and take its dagger:

```
>>> from sympy.physics.quantum import Ket, Bra, OuterProduct, Dagger
>>> from sympy.physics.quantum import Operator

>>> k = Ket('k')
>>> b = Bra('b')
>>> op = OuterProduct(k, b)
>>> op
|k><b|
>>> op.hilbert_space
H
>>> op.ket
|k>
>>> op.bra
<b|
>>> Dagger(op)
|b><k|
```

In simple products of kets and bras outer products will be automatically identified and created:

```
>>> k*b
|k><b|
```

But in more complex expressions, outer products are not automatically created:

```
>>> A = Operator('A')
>>> A*k*b
A*|k>*<b|
```

A user can force the creation of an outer product in a complex expression by using parentheses to group the ket and bra:

```
>>> A*(k*b)
A*|k><b|
```

## References

[R686]

### property bra

Return the bra on the right side of the outer product.

### property ket

Return the ket on the left side of the outer product.

**class** sympy.physics.quantum.operator.**UnitaryOperator**(\*args, \*\*kwargs)

A unitary operator that satisfies  $U^\dagger U = 1$ .

#### Parameters

**args** : tuple

The list of numbers or parameters that uniquely specify the operator.  
For time-dependent operators, this will include the time.

## Examples

```
>>> from sympy.physics.quantum import Dagger, UnitaryOperator
>>> U = UnitaryOperator('U')
>>> U*Dagger(U)
1
```

## Operator/State Helper Functions

A module for mapping operators to their corresponding eigenstates and vice versa

It contains a global dictionary with eigenstate-operator pairings. If a new state-operator pair is created, this dictionary should be updated as well.

It also contains functions `operators_to_state` and `state_to_operators` for mapping between the two. These can handle both classes and instances of operators and states. See the individual function descriptions for details.

TODO List: - Update the dictionary with a complete list of state-operator pairs

`sympy.physics.quantum.operatorset.operators_to_state(operators, **options)`

Returns the eigenstate of the given operator or set of operators

A global function for mapping operator classes to their associated states. It takes either an `Operator` or a set of operators and returns the state associated with these.

This function can handle both instances of a given operator or just the class itself (i.e. both `XOp()` and `XOp`)

There are multiple use cases to consider:

1) A class or set of classes is passed: First, we try to instantiate default instances for these operators. If this fails, then the class is simply returned. If we succeed in instantiating default instances, then we try to call `state._operators_to_state` on the operator instances. If this fails, the class is returned. Otherwise, the instance returned by `_operators_to_state` is returned.

2) An instance or set of instances is passed: In this case, `state._operators_to_state` is called on the instances passed. If this fails, a state class is returned. If the method returns an instance, that instance is returned.

In both cases, if the operator class or set does not exist in the `state_mapping` dictionary, `None` is returned.

### Parameters

**arg: Operator or set**

The class or instance of the operator or set of operators to be mapped to a state

### Examples

```
>>> from sympy.physics.quantum.cartesian import XOp, POp
>>> from sympy.physics.quantum.operatorset import operators_to_state
>>> from sympy.physics.quantum.operator import Operator
>>> operators_to_state(XOp)
|x>
>>> operators_to_state(XOp())
|x>
>>> operators_to_state(POp)
|px>
>>> operators_to_state(POp())
|px>
>>> operators_to_state(Operator)
|psi>
>>> operators_to_state(Operator())
|psi>
```

`sympy.physics.quantum.operatorset.state_to_operators(state, **options)`

Returns the operator or set of operators corresponding to the given eigenstate

A global function for mapping state classes to their associated operators or sets of operators. It takes either a state class or instance.

This function can handle both instances of a given state or just the class itself (i.e. both `XXKet()` and `XXKet`)

There are multiple use cases to consider:

1) A state class is passed: In this case, we first try instantiating a default instance of the class. If this succeeds, then we try to call `state._state_to_operators` on that instance. If the creation of the default instance or if the calling of `_state_to_operators` fails, then either an operator class or set of operator classes is returned. Otherwise, the appropriate operator instances are returned.

2) A state instance is returned: Here, `state._state_to_operators` is called for the instance. If this fails, then a class or set of operator classes is returned. Otherwise, the instances are returned.

In either case, if the state's class does not exist in `state_mapping`, `None` is returned.

**Parameters**

**arg: StateBase class or instance (or subclasses)**

The class or instance of the state to be mapped to an operator or set of operators

**Examples**

```
>>> from sympy.physics.quantum.cartesian import XKet, PxKet, XBra, PxBra
>>> from sympy.physics.quantum.operatorset import state_to_operators
>>> from sympy.physics.quantum.state import Ket, Bra
>>> state_to_operators(XKet)
X
>>> state_to_operators(XKet())
X
>>> state_to_operators(PxKet)
Px
>>> state_to_operators(PxKet())
Px
>>> state_to_operators(PxBra)
Px
>>> state_to_operators(XBra)
X
>>> state_to_operators(Ket)
0
>>> state_to_operators(Bra)
0
```

**Qapply**

Logic for applying operators to states.

Todo: \* Sometimes the final result needs to be expanded, we should do this by hand.

`sympy.physics.quantum.qapply.qapply(e, **options)`

Apply operators to states in a quantum expression.

**Parameters**

**e** : Expr

The expression containing operators and states. This expression tree will be walked to find operators acting on states symbolically.

**options** : dict

A dict of key/value pairs that determine how the operator actions are carried out.

The following options are valid:

- **dagger**: try to apply Dagger operators to the left (default: `False`).
- **ip\_doit**: call `.doit()` in inner products when they are encountered (default: `True`).

**Returns****e** : Expr

The original expression, but with the operators applied to states.

**Examples**

```

>>> from sympy.physics.quantum import qapply, Ket, Bra
>>> b = Bra('b')
>>> k = Ket('k')
>>> A = k * b
>>> A
|k><b|
>>> qapply(A * b.dual / (b * b.dual))
|k>
>>> qapply(k.dual * A / (k.dual * k), dagger=True)
<b|
>>> qapply(k.dual * A / (k.dual * k))
<k|*|k><b|/<k|k>

```

**Represent**

Logic for representing operators in state in various bases.

TODO:

- Get represent working with continuous hilbert spaces.
- Document default basis functionality.

`sympy.physics.quantum.represent.enumerate_states(*args, **options)`

Returns instances of the given state with dummy indices appended

Operates in two different modes:

1. Two arguments are passed to it. The first is the base state which is to be indexed, and the second argument is a list of indices to append.
2. Three arguments are passed. The first is again the base state to be indexed. The second is the start index for counting. The final argument is the number of kets you wish to receive.

Tries to call `state._enumerate_state`. If this fails, returns an empty list**Parameters****args** : list

See list of operation modes above for explanation

## Examples

```
>>> from sympy.physics.quantum.cartesian import XBra, XKet
>>> from sympy.physics.quantum.represent import enumerate_states
>>> test = XKet('foo')
>>> enumerate_states(test, 1, 3)
[|foo_1>, |foo_2>, |foo_3>]
>>> test2 = XBra('bar')
>>> enumerate_states(test2, [4, 5, 10])
[<bar_4|, <bar_5|, <bar_10|]
```

`sympy.physics.quantum.represent.get_basis(expr, *, basis=None, replace_none=True, **options)`

Returns a basis state instance corresponding to the basis specified in `options=s`. If no basis is specified, the function tries to form a default basis state of the given expression.

There are three behaviors:

1. The basis specified in options is already an instance of `StateBase`. If this is the case, it is simply returned. If the class is specified but not an instance, a default instance is returned.
2. The basis specified is an operator or set of operators. If this is the case, the `operator_to_state` mapping method is used.
3. No basis is specified. If `expr` is a state, then a default instance of its class is returned. If `expr` is an operator, then it is mapped to the corresponding state. If it is neither, then we cannot obtain the basis state.

If the basis cannot be mapped, then it is not changed.

This will be called from within `represent`, and `represent` will only pass `QExpr`'s.

TODO (?): Support for Muls and other types of expressions?

### Parameters

**expr** : Operator or `StateBase`

Expression whose basis is sought

## Examples

```
>>> from sympy.physics.quantum.represent import get_basis
>>> from sympy.physics.quantum.cartesian import XOp, XKet, PxOp, PxKet
>>> x = XKet()
>>> X = XOp()
>>> get_basis(x)
|x>
>>> get_basis(X)
|x>
>>> get_basis(x, basis=PxOp())
|px>
>>> get_basis(x, basis=PxKet)
|px>
```



`sympy.physics.quantum.represent.integrate_result(orig_expr, result, **options)`

Returns the result of integrating over any unities ( $|x\rangle\langle x|$ ) in the given expression. Intended for integrating over the result of representations in continuous bases.

This function integrates over any unities that may have been inserted into the quantum expression and returns the result. It uses the interval of the Hilbert space of the basis state passed to it in order to figure out the limits of integration. The unities option must be specified for this to work.

Note: This is mostly used internally by `represent()`. Examples are given merely to show the use cases.

#### Parameters

**orig\_expr** : quantum expression

The original expression which was to be represented

**result: Expr**

The resulting representation that we wish to integrate over

#### Examples

```
>>> from sympy import symbols, DiracDelta
>>> from sympy.physics.quantum.represent import integrate_result
>>> from sympy.physics.quantum.cartesian import X0p, XKet
>>> x_ket = XKet()
>>> X_op = X0p()
>>> x, x_1, x_2 = symbols('x, x_1, x_2')
>>> integrate_result(X_op*x_ket, x*DiracDelta(x-x_1)*DiracDelta(x_1-x_2))
x*DiracDelta(x - x_1)*DiracDelta(x_1 - x_2)
>>> integrate_result(X_op*x_ket, x*DiracDelta(x-x_1)*DiracDelta(x_1-x_2),
... unities=[1])
x*DiracDelta(x - x_2)
```

`sympy.physics.quantum.represent.rep_expectation(expr, **options)`

Returns an  $\langle x | A | x \rangle$  type representation for the given operator.

#### Parameters

**expr** : Operator

Operator to be represented in the specified basis

#### Examples

```
>>> from sympy.physics.quantum.cartesian import X0p, Px0p, PxKet
>>> from sympy.physics.quantum.represent import rep_expectation
>>> rep_expectation(X0p())
x_1*DiracDelta(x_1 - x_2)
>>> rep_expectation(X0p(), basis=Px0p())
<px_2|*X*|px_1>
>>> rep_expectation(X0p(), basis=PxKet())
<px_2|*X*|px_1>
```

`sympy.physics.quantum.represent.rep_innerproduct(expr, **options)`

Returns an innerproduct like representation (e.g.  $\langle x' | x \rangle$ ) for the given state.

Attempts to calculate inner product with a bra from the specified basis. Should only be passed an instance of KetBase or BraBase

#### Parameters

**expr** : KetBase or BraBase

The expression to be represented

#### Examples

```
>>> from sympy.physics.quantum.represent import rep_innerproduct
>>> from sympy.physics.quantum.cartesian import X0p, XKet, Px0p, PxKet
>>> rep_innerproduct(XKet())
DiracDelta(x - x_1)
>>> rep_innerproduct(XKet(), basis=Px0p())
sqrt(2)*exp(-I*px_1*x/hbar)/(2*sqrt(hbar)*sqrt(pi))
>>> rep_innerproduct(PxKet(), basis=X0p())
sqrt(2)*exp(I*px*x_1/hbar)/(2*sqrt(hbar)*sqrt(pi))
```

`sympy.physics.quantum.represent.represent(expr, **options)`

Represent the quantum expression in the given basis.

In quantum mechanics abstract states and operators can be represented in various basis sets. Under this operation the follow transforms happen:

- Ket -> column vector or function
- Bra -> row vector of function
- Operator -> matrix or differential operator

This function is the top-level interface for this action.

This function walks the SymPy expression tree looking for QExpr instances that have a `_represent` method. This method is then called and the object is replaced by the representation returned by this method. By default, the `_represent` method will dispatch to other methods that handle the representation logic for a particular basis set. The naming convention for these methods is the following:

```
def _represent_FooBasis(self, e, basis, **options)
```

This function will have the logic for representing instances of its class in the basis set having a class named `FooBasis`.

#### Parameters

**expr** : Expr

The expression to represent.

**basis** : Operator, basis set

An object that contains the information about the basis set. If an operator is used, the basis is assumed to be the orthonormal eigenvectors of that operator. In general though, the basis argument can be any object that contains the basis set information.

**options** : dict

Key/value pairs of options that are passed to the underlying method that finds the representation. These options can be used to control how the representation is done. For example, this is where the size of the basis set would be set.

### Returns

**e** : Expr

The SymPy expression of the represented quantum expression.

## Examples

Here we subclass `Operator` and `Ket` to create the z-spin operator and its spin 1/2 up eigenstate. By defining the `_represent_SzOp` method, the ket can be represented in the z-spin basis.

```
>>> from sympy.physics.quantum import Operator, represent, Ket
>>> from sympy import Matrix
```

```
>>> class SzUpKet(Ket):
... def _represent_SzOp(self, basis, **options):
... return Matrix([1,0])
...
>>> class SzOp(Operator):
... pass
...
>>> sz = SzOp('Sz')
>>> up = SzUpKet('up')
>>> represent(up, basis=sz)
Matrix([
[1],
[0]])
```

Here we see an example of representations in a continuous basis. We see that the result of representing various combinations of cartesian position operators and kets give us continuous expressions involving DiracDelta functions.

```
>>> from sympy.physics.quantum.cartesian import XOp, XKet, XBra
>>> X = XOp()
>>> x = XKet()
>>> y = XBra('y')
>>> represent(X*x)
x*DiracDelta(x - x_2)
>>> represent(X*x*y)
x*DiracDelta(x - x_3)*DiracDelta(x_1 - y)
```

## Spin

Quantum mechanical angular momentum.

**class** sympy.physics.quantum.spin.**J2Op**(\*args, \*\*kwargs)

The  $J^2$  operator.

**class** sympy.physics.quantum.spin.**JxBra**(j, m)

Eigenbra of  $J_x$ .

See JzKet for the usage of spin eigenstates.

**See also:**

[\*\*JzKet\*\* \(page 1821\)](#)

Usage of spin states

**class** sympy.physics.quantum.spin.**JxBraCoupled**(j, m, jn, \*jcoupling)

Coupled eigenbra of  $J_x$ .

See JzKetCoupled for the usage of coupled spin eigenstates.

**See also:**

[\*\*JzKetCoupled\*\* \(page 1823\)](#)

Usage of coupled spin states

**class** sympy.physics.quantum.spin.**JxKet**(j, m)

Eigenket of  $J_x$ .

See JzKet for the usage of spin eigenstates.

**See also:**

[\*\*JzKet\*\* \(page 1821\)](#)

Usage of spin states

**class** sympy.physics.quantum.spin.**JxKetCoupled**(j, m, jn, \*jcoupling)

Coupled eigenket of  $J_x$ .

See JzKetCoupled for the usage of coupled spin eigenstates.

**See also:**

[\*\*JzKetCoupled\*\* \(page 1823\)](#)

Usage of coupled spin states

**class** sympy.physics.quantum.spin.**JyBra**(j, m)

Eigenbra of  $J_y$ .

See JzKet for the usage of spin eigenstates.

**See also:**

[\*\*JzKet\*\* \(page 1821\)](#)

Usage of spin states

**class** `sympy.physics.quantum.spin.JyBraCoupled(j, m, jn, *jcoupling)`

Coupled eigenbra of Jy.

See `JzKetCoupled` for the usage of coupled spin eigenstates.

**See also:**

[\*JzKetCoupled\* \(page 1823\)](#)

Usage of coupled spin states

**class** `sympy.physics.quantum.spin.JyKet(j, m)`

Eigenket of Jy.

See `JzKet` for the usage of spin eigenstates.

**See also:**

[\*JzKet\* \(page 1821\)](#)

Usage of spin states

**class** `sympy.physics.quantum.spin.JyKetCoupled(j, m, jn, *jcoupling)`

Coupled eigenket of Jy.

See `JzKetCoupled` for the usage of coupled spin eigenstates.

**See also:**

[\*JzKetCoupled\* \(page 1823\)](#)

Usage of coupled spin states

**class** `sympy.physics.quantum.spin.JzBra(j, m)`

Eigenbra of Jz.

See the `JzKet` for the usage of spin eigenstates.

**See also:**

[\*JzKet\* \(page 1821\)](#)

Usage of spin states

**class** `sympy.physics.quantum.spin.JzBraCoupled(j, m, jn, *jcoupling)`

Coupled eigenbra of Jz.

See the `JzKetCoupled` for the usage of coupled spin eigenstates.

**See also:**

[\*JzKetCoupled\* \(page 1823\)](#)

Usage of coupled spin states

**class** `sympy.physics.quantum.spin.JzKet(j, m)`

Eigenket of Jz.

Spin state which is an eigenstate of the Jz operator. Uncoupled states, that is states representing the interaction of multiple separate spin states, are defined as a tensor product of states.

**Parameters**

**j** : Number, Symbol

Total spin angular momentum

**m** : Number, Symbol

Eigenvalue of the Jz spin operator

## Examples

*Normal States:*

Defining simple spin states, both numerical and symbolic:

```
>>> from sympy.physics.quantum.spin import JzKet, JxKet
>>> from sympy import symbols
>>> JzKet(1, 0)
|1,0>
>>> j, m = symbols('j m')
>>> JzKet(j, m)
|j,m>
```

Rewriting the JzKet in terms of eigenkets of the Jx operator: Note: that the resulting eigenstates are JxKet's

```
>>> JzKet(1,1).rewrite("Jx")
|1,-1>/2 - sqrt(2)*|1,0>/2 + |1,1>/2
```

Get the vector representation of a state in terms of the basis elements of the Jx operator:

```
>>> from sympy.physics.quantum.represent import represent
>>> from sympy.physics.quantum.spin import Jx, Jz
>>> represent(JzKet(1,-1), basis=Jx)
Matrix([
[1/2],
[sqrt(2)/2],
[1/2]])
```

Apply innerproducts between states:

```
>>> from sympy.physics.quantum.innerproduct import InnerProduct
>>> from sympy.physics.quantum.spin import JxBra
>>> i = InnerProduct(JxBra(1,1), JzKet(1,1))
>>> i
<1,1|1,1>
>>> i.doit()
1/2
```

*Uncoupled States:*

Define an uncoupled state as a TensorProduct between two Jz eigenkets:

```
>>> from sympy.physics.quantum.tensorproduct import TensorProduct
>>> j1,m1,j2,m2 = symbols('j1 m1 j2 m2')
>>> TensorProduct(JzKet(1,0), JzKet(1,1))
|1,0>x|1,1>
>>> TensorProduct(JzKet(j1,m1), JzKet(j2,m2))
|j1,m1>x|j2,m2>
```

A `TensorProduct` can be rewritten, in which case the eigenstates that make up the tensor product is rewritten to the new basis:

```
>>> TensorProduct(JzKet(1,1),JxKet(1,1)).rewrite('Jz')
|1,1>x|1,-1>/2 + sqrt(2)*|1,1>x|1,0>/2 + |1,1>x|1,1>/2
```

The `represent` method for `TensorProduct`'s gives the vector representation of the state. Note that the state in the product basis is the equivalent of the tensor product of the vector representation of the component eigenstates:

```
>>> represent(TensorProduct(JzKet(1,0),JzKet(1,1)))
Matrix([
[0],
[0],
[0],
[1],
[0],
[0],
[0],
[0],
[0],
[0]])
>>> represent(TensorProduct(JzKet(1,1),JxKet(1,1)), basis=Jz)
Matrix([
[1/2],
[sqrt(2)/2],
[1/2],
[0],
[0],
[0],
[0],
[0],
[0],
[0]])
```

**See also:**

**[JzKetCoupled](#) (page 1823)**

Coupled eigenstates

**[sympy.physics.quantum.tensorproduct.TensorProduct](#) (page 1799)**

Used to specify uncoupled states

**[uncouple](#) (page 1830)**

Uncouples states given coupling parameters

**[couple](#) (page 1829)**

Couples uncoupled states

**class** `sympy.physics.quantum.spin.JzKetCoupled(j, m, jn, *jcoupling)`

Coupled eigenket of  $J_z$

Spin state that is an eigenket of  $J_z$  which represents the coupling of separate spin spaces.

The arguments for creating instances of `JzKetCoupled` are `j`, `m`, `jn` and an optional `jcoupling` argument. The `j` and `m` options are the total angular momentum quantum numbers, as used for normal states (e.g. `JzKet`).

The other required parameter in `jn`, which is a tuple defining the  $j_n$  angular momentum quantum numbers of the product spaces. So for example, if a state represented the coupling of the product basis state  $|j_1, m_1\rangle \times |j_2, m_2\rangle$ , the `jn` for this state would be `(j1, j2)`.

The final option is `jcoupling`, which is used to define how the spaces specified by `jn` are coupled, which includes both the order these spaces are coupled together and the quantum numbers that arise from these couplings. The `jcoupling` parameter itself is a list of lists, such that each of the sublists defines a single coupling between the spin spaces. If there are  $N$  coupled angular momentum spaces, that is `jn` has  $N$  elements, then there must be  $N-1$  sublists. Each of these sublists making up the `jcoupling` parameter have length 3. The first two elements are the indices of the product spaces that are considered to be coupled together. For example, if we want to couple  $j_1$  and  $j_4$ , the indices would be 1 and 4. If a state has already been coupled, it is referenced by the smallest index that is coupled, so if  $j_2$  and  $j_4$  has already been coupled to some  $j_{24}$ , then this value can be coupled by referencing it with index 2. The final element of the sublist is the quantum number of the coupled state. So putting everything together, into a valid sublist for `jcoupling`, if  $j_1$  and  $j_2$  are coupled to an angular momentum space with quantum number  $j_{12}$  with the value `j12`, the sublist would be `(1,2,j12)`,  $N-1$  of these sublists are used in the list for `jcoupling`.

Note the `jcoupling` parameter is optional, if it is not specified, the default coupling is taken. This default value is to coupled the spaces in order and take the quantum number of the coupling to be the maximum value. For example, if the spin spaces are  $j_1, j_2, j_3, j_4$ , then the default coupling couples  $j_1$  and  $j_2$  to  $j_{12} = j_1 + j_2$ , then,  $j_{12}$  and  $j_3$  are coupled to  $j_{123} = j_{12} + j_3$ , and finally  $j_{123}$  and  $j_4$  to  $j = j_{123} + j_4$ . The `jcoupling` value that would correspond to this is:

`((1,2,j1+j2),(1,3,j1+j2+j3))`

### Parameters

`args` : tuple

The arguments that must be passed are `j`, `m`, `jn`, and `jcoupling`. The `j` value is the total angular momentum. The `m` value is the eigenvalue of the  $J_z$  spin operator. The `jn` list are the  $j$  values of angular momentum spaces coupled together. The `jcoupling` parameter is an optional parameter defining how the spaces are coupled together. See the above description for how these coupling parameters are defined.

### Examples

Defining simple spin states, both numerical and symbolic:

```
>>> from sympy.physics.quantum.spin import JzKetCoupled
>>> from sympy import symbols
>>> JzKetCoupled(1, 0, (1, 1))
|1,0,j1=1,j2=1>
>>> j, m, j1, j2 = symbols('j m j1 j2')
>>> JzKetCoupled(j, m, (j1, j2))
|j,m,j1=j1,j2=j2>
```

Defining coupled spin states for more than 2 coupled spaces with various coupling parameters:



```
>>> JzKetCoupled(2, 1, (1, 1, 1))
|2,1,j1=1,j2=1,j3=1,j(1,2)=2>
>>> JzKetCoupled(2, 1, (1, 1, 1), ((1,2,2),(1,3,2)))
|2,1,j1=1,j2=1,j3=1,j(1,2)=2>
>>> JzKetCoupled(2, 1, (1, 1, 1), ((2,3,1),(1,2,2)))
|2,1,j1=1,j2=1,j3=1,j(2,3)=1>
```

Rewriting the JzKetCoupled in terms of eigenkets of the Jx operator: Note: that the resulting eigenstates are JxKetCoupled

```
>>> JzKetCoupled(1,1,(1,1)).rewrite("Jx")
|1,-1,j1=1,j2=1>/2 - sqrt(2)*|1,0,j1=1,j2=1>/2 + |1,1,j1=1,j2=1>/2
```

The rewrite method can be used to convert a coupled state to an uncoupled state. This is done by passing coupled=False to the rewrite function:

```
>>> JzKetCoupled(1, 0, (1, 1)).rewrite('Jz', coupled=False)
-sqrt(2)*|1,-1>x|1,1>/2 + sqrt(2)*|1,1>x|1,-1>/2
```

Get the vector representation of a state in terms of the basis elements of the Jx operator:

```
>>> from sympy.physics.quantum.represent import represent
>>> from sympy.physics.quantum.spin import Jx
>>> from sympy import S
>>> represent(JzKetCoupled(1, -1, (S(1)/2, S(1)/2)), basis=Jx)
Matrix([
[0],
[1/2],
[sqrt(2)/2],
[1/2]])
```

**See also:**

**JzKet (page 1821)**

Normal spin eigenstates

**uncouple (page 1830)**

Uncoupling of coupling spin states

**couple (page 1829)**

Coupling of uncoupled spin states

**class** sympy.physics.quantum.spin.**JzOp**(\*args, \*\*kwargs)

The Jz operator.

**class** sympy.physics.quantum.spin.**Rotation**(\*args, \*\*kwargs)

Wigner D operator in terms of Euler angles.

Defines the rotation operator in terms of the Euler angles defined by the z-y-z convention for a passive transformation. That is the coordinate axes are rotated first about the z-axis, giving the new x'-y'-z' axes. Then this new coordinate system is rotated about the new y'-axis, giving new x''-y''-z'' axes. Then this new coordinate system is rotated about the z''-axis. Conventions follow those laid out in [R687].

**Parameters**

**alpha** : Number, Symbol

First Euler Angle

**beta** : Number, Symbol

Second Euler angle

**gamma** : Number, Symbol

Third Euler angle

## Examples

A simple example rotation operator:

```
>>> from sympy import pi
>>> from sympy.physics.quantum.spin import Rotation
>>> Rotation(pi, 0, pi/2)
R(pi,0,pi/2)
```

With symbolic Euler angles and calculating the inverse rotation operator:

```
>>> from sympy import symbols
>>> a, b, c = symbols('a b c')
>>> Rotation(a, b, c)
R(a,b,c)
>>> Rotation(a, b, c).inverse()
R(-c,-b,-a)
```

**See also:**

***WignerD*** (page 1828)

Symbolic Wigner-D function

***D*** (page 1826)

Wigner-D function

***d*** (page 1827)

Wigner small-d function

## References

[R687]

**classmethod** **D**(*j, m, mp, alpha, beta, gamma*)

Wigner D-function.

Returns an instance of the WignerD class corresponding to the Wigner-D function specified by the parameters.

### Parameters

**j** : Number

Total angular momentum

**m** : Number

Eigenvalue of angular momentum along axis after rotation

**mp** : Number

Eigenvalue of angular momentum along rotated axis

**alpha** : Number, Symbol

First Euler angle of rotation

**beta** : Number, Symbol

Second Euler angle of rotation

**gamma** : Number, Symbol

Third Euler angle of rotation

## Examples

Return the Wigner-D matrix element for a defined rotation, both numerical and symbolic:

```
>>> from sympy.physics.quantum.spin import Rotation
>>> from sympy import pi, symbols
>>> alpha, beta, gamma = symbols('alpha beta gamma')
>>> Rotation.D(1, 1, 0, pi, pi/2, -pi)
WignerD(1, 1, 0, pi, pi/2, -pi)
```

**See also:**

[\*WignerD\*](#) (page 1828)

Symbolic Wigner-D function

**classmethod** `d(j, m, mp, beta)`

Wigner small-d function.

Returns an instance of the WignerD class corresponding to the Wigner-D function specified by the parameters with the alpha and gamma angles given as 0.

### Parameters

**j** : Number

Total angular momentum

**m** : Number

Eigenvalue of angular momentum along axis after rotation

**mp** : Number

Eigenvalue of angular momentum along rotated axis

**beta** : Number, Symbol

Second Euler angle of rotation

## Examples

Return the Wigner-D matrix element for a defined rotation, both numerical and symbolic:

```
>>> from sympy.physics.quantum.spin import Rotation
>>> from sympy import pi, symbols
>>> beta = symbols('beta')
>>> Rotation.d(1, 1, 0, pi/2)
WignerD(1, 1, 0, 0, pi/2, 0)
```

**See also:**

***WignerD*** (page 1828)

Symbolic Wigner-D function

**class** sympy.physics.quantum.spin.**WignerD**(\*args, \*\*hints)

Wigner-D function

The Wigner D-function gives the matrix elements of the rotation operator in the jm-representation. For the Euler angles  $\alpha$ ,  $\beta$ ,  $\gamma$ , the D-function is defined such that:

$$\langle j, m | \mathcal{R}(\alpha, \beta, \gamma) | j', m' \rangle = \delta_{jj'} D(j, m, m', \alpha, \beta, \gamma)$$

Where the rotation operator is as defined by the Rotation class [R688].

The Wigner D-function defined in this way gives:

$$D(j, m, m', \alpha, \beta, \gamma) = e^{-im\alpha} d(j, m, m', \beta) e^{-im'\gamma}$$

Where d is the Wigner small-d function, which is given by Rotation.d.

The Wigner small-d function gives the component of the Wigner D-function that is determined by the second Euler angle. That is the Wigner D-function is:

$$D(j, m, m', \alpha, \beta, \gamma) = e^{-im\alpha} d(j, m, m', \beta) e^{-im'\gamma}$$

Where d is the small-d function. The Wigner D-function is given by Rotation.D.

Note that to evaluate the D-function, the j, m and mp parameters must be integer or half integer numbers.

### Parameters

**j** : Number

Total angular momentum

**m** : Number

Eigenvalue of angular momentum along axis after rotation

**mp** : Number

Eigenvalue of angular momentum along rotated axis

**alpha** : Number, Symbol

First Euler angle of rotation

**beta** : Number, Symbol

Second Euler angle of rotation

**gamma** : Number, Symbol  
Third Euler angle of rotation

## Examples

Evaluate the Wigner-D matrix elements of a simple rotation:

```
>>> from sympy.physics.quantum.spin import Rotation
>>> from sympy import pi
>>> rot = Rotation.D(1, 1, 0, pi, pi/2, 0)
>>> rot
WignerD(1, 1, 0, pi, pi/2, 0)
>>> rot.doit()
sqrt(2)/2
```

Evaluate the Wigner-d matrix elements of a simple rotation

```
>>> rot = Rotation.d(1, 1, 0, pi/2)
>>> rot
WignerD(1, 1, 0, 0, pi/2, 0)
>>> rot.doit()
-sqrt(2)/2
```

**See also:**

**[Rotation](#) (page 1825)**  
Rotation operator

## References

[R688]

`sympy.physics.quantum.spin.couple(expr, jcoupling_list=None)`

Couple a tensor product of spin states

This function can be used to couple an uncoupled tensor product of spin states. All of the eigenstates to be coupled must be of the same class. It will return a linear combination of eigenstates that are subclasses of `CoupledSpinState` determined by Clebsch-Gordan angular momentum coupling coefficients.

### Parameters

**expr** : Expr

An expression involving `TensorProducts` of spin states to be coupled. Each state must be a subclass of `SpinState` and they all must be the same class.

**jcoupling\_list** : list or tuple

Elements of this list are sub-lists of length 2 specifying the order of the coupling of the spin spaces. The length of this must be  $N-1$ , where  $N$  is the number of states in the tensor product to be coupled. The elements of this sublist are the same as the first two elements of each sublist in the `jcoupling` parameter defined for `JzKetCoupled`. If this

parameter is not specified, the default value is taken, which couples the first and second product basis spaces, then couples this new coupled space to the third product space, etc

## Examples

Couple a tensor product of numerical states for two spaces:

```
>>> from sympy.physics.quantum.spin import JzKet, couple
>>> from sympy.physics.quantum.tensorproduct import TensorProduct
>>> couple(TensorProduct(JzKet(1,0), JzKet(1,1)))
-sqrt(2)*|1,1,j1=1,j2=1>/2 + sqrt(2)*|2,1,j1=1,j2=1>/2
```

Numerical coupling of three spaces using the default coupling method, i.e. first and second spaces couple, then this couples to the third space:

```
>>> couple(TensorProduct(JzKet(1,1), JzKet(1,1), JzKet(1,0)))
sqrt(6)*|2,2,j1=1,j2=1,j3=1,j(1,2)=2>/3 + sqrt(3)*|3,2,j1=1,j2=1,j3=1,
↪ j(1,2)=2>/3
```

Perform this same coupling, but we define the coupling to first couple the first and third spaces:

```
>>> couple(TensorProduct(JzKet(1,1), JzKet(1,1), JzKet(1,0)), ((1,3),(1,
↪ 2)))
sqrt(2)*|2,2,j1=1,j2=1,j3=1,j(1,3)=1>/2 - sqrt(6)*|2,2,j1=1,j2=1,j3=1,
↪ j(1,3)=2>/6 + sqrt(3)*|3,2,j1=1,j2=1,j3=1,j(1,3)=2>/3
```

Couple a tensor product of symbolic states:

```
>>> from sympy import symbols
>>> j1,m1,j2,m2 = symbols('j1 m1 j2 m2')
>>> couple(TensorProduct(JzKet(j1,m1), JzKet(j2,m2)))
Sum(CG(j1, m1, j2, m2, j, m1 + m2)*|j,m1 + m2,j1=j1,j2=j2>, (j, m1 + m2,
↪ j1 + j2))
```

`sympy.physics.quantum.spin.uncouple(expr, jn=None, jcoupling_list=None)`

Uncouple a coupled spin state

Gives the uncoupled representation of a coupled spin state. Arguments must be either a spin state that is a subclass of `CoupledSpinState` or a spin state that is a subclass of `SpinState` and an array giving the `j` values of the spaces that are to be coupled

### Parameters

**expr** : Expr

The expression containing states that are to be coupled. If the states are a subclass of `SpinState`, the `jn` and `jcoupling` parameters must be defined. If the states are a subclass of `CoupledSpinState`, `jn` and `jcoupling` will be taken from the state.

**jn** : list or tuple

The list of the `j`-values that are coupled. If state is a `CoupledSpinState`, this parameter is ignored. This must be defined if state is not

a subclass of `CoupledSpinState`. The syntax of this parameter is the same as the `jn` parameter of `JzKetCoupled`.

**jcoupling\_list** : list or tuple

The list defining how the *j*-values are coupled together. If state is a `CoupledSpinState`, this parameter is ignored. This must be defined if state is not a subclass of `CoupledSpinState`. The syntax of this parameter is the same as the `jcoupling` parameter of `JzKetCoupled`.

## Examples

Uncouple a numerical state using a `CoupledSpinState` state:

```
>>> from sympy.physics.quantum.spin import JzKetCoupled, uncouple
>>> from sympy import S
>>> uncouple(JzKetCoupled(1, 0, (S(1)/2, S(1)/2)))
sqrt(2)*|1/2, -1/2>x|1/2, 1/2>/2 + sqrt(2)*|1/2, 1/2>x|1/2, -1/2>/2
```

Perform the same calculation using a `SpinState` state:

```
>>> from sympy.physics.quantum.spin import JzKet
>>> uncouple(JzKet(1, 0), (S(1)/2, S(1)/2))
sqrt(2)*|1/2, -1/2>x|1/2, 1/2>/2 + sqrt(2)*|1/2, 1/2>x|1/2, -1/2>/2
```

Uncouple a numerical state of three coupled spaces using a `CoupledSpinState` state:

```
>>> uncouple(JzKetCoupled(1, 1, (1, 1, 1), ((1,3,1),(1,2,1))))
|1, -1>x|1, 1>x|1, 1>/2 - |1, 0>x|1, 0>x|1, 1>/2 + |1, 1>x|1, 0>x|1, 0>/2 - |1, 1>
→x|1, 1>x|1, -1>/2
```

Perform the same calculation using a `SpinState` state:

```
>>> uncouple(JzKet(1, 1), (1, 1, 1), ((1,3,1),(1,2,1)))
|1, -1>x|1, 1>x|1, 1>/2 - |1, 0>x|1, 0>x|1, 1>/2 + |1, 1>x|1, 0>x|1, 0>/2 - |1, 1>
→x|1, 1>x|1, -1>/2
```

Uncouple a symbolic state using a `CoupledSpinState` state:

```
>>> from sympy import symbols
>>> j,m,j1,j2 = symbols('j m j1 j2')
>>> uncouple(JzKetCoupled(j, m, (j1, j2)))
Sum(CG(j1, m1, j2, m2, j, m)*|j1,m1>x|j2,m2>, (m1, -j1, j1), (m2, -j2,
→j2))
```

Perform the same calculation using a `SpinState` state

```
>>> uncouple(JzKet(j, m), (j1, j2))
Sum(CG(j1, m1, j2, m2, j, m)*|j1,m1>x|j2,m2>, (m1, -j1, j1), (m2, -j2,
→j2))
```

## State

Dirac notation for states.

**class** sympy.physics.quantum.state.**Bra**(\*args, \*\*kwargs)

A general time-independent Bra in quantum mechanics.

Inherits from State and BraBase. A Bra is the dual of a Ket [R689]. This class and its subclasses will be the main classes that users will use for expressing Bras in Dirac notation.

### Parameters

**args** : tuple

The list of numbers or parameters that uniquely specify the ket. This will usually be its symbol or its quantum numbers. For time-dependent state, this will include the time.

## Examples

Create a simple Bra and look at its properties:

```
>>> from sympy.physics.quantum import Bra
>>> from sympy import symbols, I
>>> b = Bra('psi')
>>> b
<psi|
>>> b.hilbert_space
H
>>> b.is_commutative
False
```

Bra's know about their dual Ket's:

```
>>> b.dual
|psi>
>>> b.dual_class()
<class 'sympy.physics.quantum.state.Ket'>
```

Like Kets, Bras can have compound labels and be manipulated in a similar manner:

```
>>> n, m = symbols('n,m')
>>> b = Bra(n,m) - I*Bra(m,n)
>>> b
-I*<mn| + <nm|
```

Symbols in a Bra can be substituted using `.subs`:

```
>>> b.subs(n,m)
<mm| - I*<mm|
```



## References

[R689]

**class** sympy.physics.quantum.state.**BraBase**(\*args, \*\*kwargs)

Base class for Bras.

This class defines the dual property and the brackets for printing. This is an abstract base class and you should not instantiate it directly, instead use Bra.

**class** sympy.physics.quantum.state.**Ket**(\*args, \*\*kwargs)

A general time-independent Ket in quantum mechanics.

Inherits from State and KetBase. This class should be used as the base class for all physical, time-independent Kets in a system. This class and its subclasses will be the main classes that users will use for expressing Kets in Dirac notation [R690].

### Parameters

**args** : tuple

The list of numbers or parameters that uniquely specify the ket. This will usually be its symbol or its quantum numbers. For time-dependent state, this will include the time.

## Examples

Create a simple Ket and looking at its properties:

```
>>> from sympy.physics.quantum import Ket
>>> from sympy import symbols, I
>>> k = Ket('psi')
>>> k
|psi>
>>> k.hilbert_space
H
>>> k.is_commutative
False
>>> k.label
(psi,)
```

Ket's know about their associated bra:

```
>>> k.dual
<psi|
>>> k.dual_class()
<class 'sympy.physics.quantum.state.Bra'>
```

Take a linear combination of two kets:

```
>>> k0 = Ket(0)
>>> k1 = Ket(1)
>>> 2*I*k0 - 4*k1
2*I*|0> - 4*|1>
```

Compound labels are passed as tuples:

```
>>> n, m = symbols('n,m')
>>> k = Ket(n,m)
>>> k
|nm>
```

## References

[R690]

**class** sympy.physics.quantum.state.**KetBase**(\*args, \*\*kwargs)

Base class for Kets.

This class defines the dual property and the brackets for printing. This is an abstract base class and you should not instantiate it directly, instead use Ket.

**class** sympy.physics.quantum.state.**OrthogonalBra**(\*args, \*\*kwargs)

Orthogonal Bra in quantum mechanics.

**class** sympy.physics.quantum.state.**OrthogonalKet**(\*args, \*\*kwargs)

Orthogonal Ket in quantum mechanics.

The inner product of two states with different labels will give zero, states with the same label will give one.

```
>>> from sympy.physics.quantum import OrthogonalBra, OrthogonalKet
>>> from sympy.abc import m, n
>>> (OrthogonalBra(n)*OrthogonalKet(n)).doit()
1
>>> (OrthogonalBra(n)*OrthogonalKet(n+1)).doit()
0
>>> (OrthogonalBra(n)*OrthogonalKet(m)).doit()
<n|m>
```

**class** sympy.physics.quantum.state.**OrthogonalState**(\*args, \*\*kwargs)

General abstract quantum state used as a base class for Ket and Bra.

**class** sympy.physics.quantum.state.**State**(\*args, \*\*kwargs)

General abstract quantum state used as a base class for Ket and Bra.

**class** sympy.physics.quantum.state.**StateBase**(\*args, \*\*kwargs)

Abstract base class for general abstract states in quantum mechanics.

All other state classes defined will need to inherit from this class. It carries the basic structure for all other states such as dual, \_eval\_adjoint and label.

This is an abstract base class and you should not instantiate it directly, instead use State.

### property dual

Return the dual state of this one.

**classmethod** dual\_class()

Return the class used to construct the dual.

### property operators

Return the operator(s) that this state is an eigenstate of

**class** sympy.physics.quantum.state.**TimeDepBra**(\*args, \*\*kwargs)

General time-dependent Bra in quantum mechanics.

This inherits from TimeDepState and BraBase and is the main class that should be used for Bras that vary with time. Its dual is a TimeDepBra.

#### Parameters

**args** : tuple

The list of numbers or parameters that uniquely specify the ket. This will usually be its symbol or its quantum numbers. For time-dependent state, this will include the time as the final argument.

#### Examples

```
>>> from sympy.physics.quantum import TimeDepBra
>>> b = TimeDepBra('psi', 't')
>>> b
<psi;t|
>>> b.time
t
>>> b.label
(psi,)
>>> b.hilbert_space
H
>>> b.dual
|psi;t>
```

**class** sympy.physics.quantum.state.**TimeDepKet**(\*args, \*\*kwargs)

General time-dependent Ket in quantum mechanics.

This inherits from TimeDepState and KetBase and is the main class that should be used for Kets that vary with time. Its dual is a TimeDepBra.

#### Parameters

**args** : tuple

The list of numbers or parameters that uniquely specify the ket. This will usually be its symbol or its quantum numbers. For time-dependent state, this will include the time as the final argument.

#### Examples

Create a TimeDepKet and look at its attributes:

```
>>> from sympy.physics.quantum import TimeDepKet
>>> k = TimeDepKet('psi', 't')
>>> k
|psi;t>
>>> k.time
t
>>> k.label
(psi,)
```

(continues on next page)

(continued from previous page)

```
>>> k.hilbert_space
H
```

TimeDepKets know about their dual bra:

```
>>> k.dual
<psi;t|
>>> k.dual_class()
<class 'sympy.physics.quantum.state.TimeDepBra'>
```

**class** sympy.physics.quantum.state.**TimeDepState**(\*args, \*\*kwargs)

Base class for a general time-dependent quantum state.

This class is used as a base class for any time-dependent state. The main difference between this class and the time-independent state is that this class takes a second argument that is the time in addition to the usual label argument.

**Parameters**

**args** : tuple

The list of numbers or parameters that uniquely specify the ket. This will usually be its symbol or its quantum numbers. For time-dependent state, this will include the time as the final argument.

**property label**

The label of the state.

**property time**

The time of the state.

**class** sympy.physics.quantum.state.**Wavefunction**(\*args)

Class for representations in continuous bases

This class takes an expression and coordinates in its constructor. It can be used to easily calculate normalizations and probabilities.

**Parameters**

**expr** : Expr

The expression representing the functional form of the w.f.

**coords** : Symbol or tuple

The coordinates to be integrated over, and their bounds

## Examples

Particle in a box, specifying bounds in the more primitive way of using Piecewise:

```
>>> from sympy import Symbol, Piecewise, pi, N
>>> from sympy.functions import sqrt, sin
>>> from sympy.physics.quantum.state import Wavefunction
>>> x = Symbol('x', real=True)
>>> n = 1
>>> L = 1
>>> g = Piecewise((0, x < 0), (0, x > L), (sqrt(2//L)*sin(n*pi*x/L),
```

(continues on next page)

(continued from previous page)

```

 True))
>>> f = Wavefunction(g, x)
>>> f.norm
1
>>> f.is_normalized
True
>>> p = f.prob()
>>> p(0)
0
>>> p(L)
0
>>> p(0.5)
2
>>> p(0.85*L)
2*sin(0.85*pi)**2
>>> N(p(0.85*L))
0.412214747707527

```

Additionally, you can specify the bounds of the function and the indices in a more compact way:

```

>>> from sympy import symbols, pi, diff
>>> from sympy.functions import sqrt, sin
>>> from sympy.physics.quantum.state import Wavefunction
>>> x, L = symbols('x,L', positive=True)
>>> n = symbols('n', integer=True, positive=True)
>>> g = sqrt(2/L)*sin(n*pi*x/L)
>>> f = Wavefunction(g, (x, 0, L))
>>> f.norm
1
>>> f(L+1)
0
>>> f(L-1)
sqrt(2)*sin(pi*n*(L - 1)/L)/sqrt(L)
>>> f(-1)
0
>>> f(0.85)
sqrt(2)*sin(0.85*pi*n/L)/sqrt(L)
>>> f(0.85, n=1, L=1)
sqrt(2)*sin(0.85*pi)
>>> f.is_commutative
False

```

All arguments are automatically sympified, so you can define the variables as strings rather than symbols:

```

>>> expr = x**2
>>> f = Wavefunction(expr, 'x')
>>> type(f.variables[0])
<class 'sympy.core.symbol.Symbol'>

```

Derivatives of Wavefunctions will return Wavefunctions:

```
>>> diff(f, x)
Wavefunction(2*x, x)
```

**property expr**

Return the expression which is the functional form of the Wavefunction

**Examples**

```
>>> from sympy.physics.quantum.state import Wavefunction
>>> from sympy import symbols
>>> x, y = symbols('x, y')
>>> f = Wavefunction(x**2, x)
>>> f.expr
x**2
```

**property is\_commutative**

Override Function's is\_commutative so that order is preserved in represented expressions

**property is\_normalized**

Returns true if the Wavefunction is properly normalized

**Examples**

```
>>> from sympy import symbols, pi
>>> from sympy.functions import sqrt, sin
>>> from sympy.physics.quantum.state import Wavefunction
>>> x, L = symbols('x, L', positive=True)
>>> n = symbols('n', integer=True, positive=True)
>>> g = sqrt(2/L)*sin(n*pi*x/L)
>>> f = Wavefunction(g, (x, 0, L))
>>> f.is_normalized
True
```

**property limits**

Return the limits of the coordinates which the w.f. depends on If no limits are specified, defaults to  $(-\infty, \infty)$ .

**Examples**

```
>>> from sympy.physics.quantum.state import Wavefunction
>>> from sympy import symbols
>>> x, y = symbols('x, y')
>>> f = Wavefunction(x**2, (x, 0, 1))
>>> f.limits
{x: (0, 1)}
>>> f = Wavefunction(x**2, x)
>>> f.limits
```

(continues on next page)

(continued from previous page)

```
{x: (-oo, oo)}
>>> f = Wavefunction(x**2 + y**2, x, (y, -1, 2))
>>> f.limits
{x: (-oo, oo), y: (-1, 2)}
```

**property norm**

Return the normalization of the specified functional form.

This function integrates over the coordinates of the Wavefunction, with the bounds specified.

**Examples**

```
>>> from sympy import symbols, pi
>>> from sympy.functions import sqrt, sin
>>> from sympy.physics.quantum.state import Wavefunction
>>> x, L = symbols('x,L', positive=True)
>>> n = symbols('n', integer=True, positive=True)
>>> g = sqrt(2/L)*sin(n*pi*x/L)
>>> f = Wavefunction(g, (x, 0, L))
>>> f.norm
1
>>> g = sin(n*pi*x/L)
>>> f = Wavefunction(g, (x, 0, L))
>>> f.norm
sqrt(2)*sqrt(L)/2
```

**normalize()**

Return a normalized version of the Wavefunction

**Examples**

```
>>> from sympy import symbols, pi
>>> from sympy.functions import sin
>>> from sympy.physics.quantum.state import Wavefunction
>>> x = symbols('x', real=True)
>>> L = symbols('L', positive=True)
>>> n = symbols('n', integer=True, positive=True)
>>> g = sin(n*pi*x/L)
>>> f = Wavefunction(g, (x, 0, L))
>>> f.normalize()
Wavefunction(sqrt(2)*sin(pi*n*x/L)/sqrt(L), (x, 0, L))
```

**prob()**

Return the absolute magnitude of the w.f.,  $|\psi(x)|^2$

## Examples

```
>>> from sympy import symbols, pi
>>> from sympy.functions import sin
>>> from sympy.physics.quantum.state import Wavefunction
>>> x, L = symbols('x,L', real=True)
>>> n = symbols('n', integer=True)
>>> g = sin(n*pi*x/L)
>>> f = Wavefunction(g, (x, 0, L))
>>> f.prob()
Wavefunction(sin(pi*n*x/L)**2, x)
```

## property variables

Return the coordinates which the wavefunction depends on

## Examples

```
>>> from sympy.physics.quantum.state import Wavefunction
>>> from sympy import symbols
>>> x,y = symbols('x,y')
>>> f = Wavefunction(x*y, x, y)
>>> f.variables
(x, y)
>>> g = Wavefunction(x*y, x)
>>> g.variables
(x,)
```

## Quantum Computation

### Circuit Plot

Matplotlib based plotting of quantum circuits.

Todo:

- Optimize printing of large circuits.
- Get this to work with single gates.
- Do a better job checking the form of circuits to make sure it is a Mul of Gates.
- Get multi-target gates plotting.
- Get initial and final states to plot.
- Get measurements to plot. Might need to rethink measurement as a gate issue.
- Get scale and figsize to be handled in a better way.
- Write some tests/examples!

**class** sympy.physics.quantum.circuitplot.CircuitPlot(*c*, *nqubits*, *\*\*kwargs*)

A class for managing a circuit plot.



**control\_line**(*gate\_idx*, *min\_wire*, *max\_wire*)

Draw a vertical control line.

**control\_point**(*gate\_idx*, *wire\_idx*)

Draw a control point.

**not\_point**(*gate\_idx*, *wire\_idx*)

Draw a NOT gates as the circle with plus in the middle.

**one\_qubit\_box**(*t*, *gate\_idx*, *wire\_idx*)

Draw a box for a single qubit gate.

**swap\_point**(*gate\_idx*, *wire\_idx*)

Draw a swap point as a cross.

**two\_qubit\_box**(*t*, *gate\_idx*, *wire\_idx*)

Draw a box for a two qubit gate. Does not work yet.

**update**(*kwargs*)

Load the kwargs into the instance dict.

`sympy.physics.quantum.circuitplot.CreateCGate`(*name*, *latexname*=None)

Use a lexical closure to make a controlled gate.

**class** `sympy.physics.quantum.circuitplot.Mx`(\*args, \*\*kwargs)

Mock-up of an x measurement gate.

This is in circuitplot rather than gate.py because it's not a real gate, it just draws one.

**class** `sympy.physics.quantum.circuitplot.Mz`(\*args, \*\*kwargs)

Mock-up of a z measurement gate.

This is in circuitplot rather than gate.py because it's not a real gate, it just draws one.

`sympy.physics.quantum.circuitplot.circuit_plot`(*c*, *nqubits*, \*\*kwargs)

Draw the circuit diagram for the circuit with nqubits.

#### Parameters

**c** : circuit

The circuit to plot. Should be a product of Gate instances.

**nqubits** : int

The number of qubits to include in the circuit. Must be at least as big as the largest `min_qubits` of the gates.

`sympy.physics.quantum.circuitplot.labeller`(*n*, *symbol*='q')

Autogenerate labels for wires of quantum circuits.

#### Parameters

**n** : int

number of qubits in the circuit.

**symbol** : string

A character string to precede all gate labels. E.g. 'q\_0', 'q\_1', etc.

```
>>> from sympy.physics.quantum.circuitplot import labeller
```

```
>>> labeller(2)
```

```
['q_1', 'q_0']
>>> labeller(3,'j')
['j_2', 'j_1', 'j_0']
```

## Gates

An implementation of gates that act on qubits.

Gates are unitary operators that act on the space of qubits.

Medium Term Todo:

- Optimize Gate.\_apply\_operators\_Qubit to remove the creation of many intermediate Qubit objects.
- Add commutation relationships to all operators and use this in gate\_sort.
- Fix gate\_sort and gate\_simp.
- Get multi-target UGates plotting properly.
- Get UGate to work with either sympy/numpy matrices and output either format. This should also use the matrix slots.

**class** sympy.physics.quantum.gate.CGate(\*args, \*\*kwargs)

A general unitary gate with control qubits.

A general control gate applies a target gate to a set of targets if all of the control qubits have a particular values (set by CGate.control\_value).

### Parameters

**label** : tuple

The label in this case has the form (controls, gate), where controls is a tuple/list of control qubits (as ints) and gate is a Gate instance that is the target operator.

### property controls

A tuple of control qubits.

### decompose(\*\*options)

Decompose the controlled gate into CNOT and single qubits gates.

### eval\_controls(qubit)

Return True/False to indicate if the controls are satisfied.

### property gate

The non-controlled gate that will be applied to the targets.

### property min\_qubits

The minimum number of qubits this gate needs to act on.

### property nqubits

The total number of qubits this gate acts on.

For controlled gate subclasses this includes both target and control qubits, so that, for examples the CNOT gate acts on 2 qubits.

**plot\_gate**(*circ\_plot*, *gate\_idx*)

Plot the controlled gate. If *simplify\_cgate* is true, simplify C-X and C-Z gates into their more familiar forms.

**property targets**

A tuple of target qubits.

**class** `sympy.physics.quantum.gate.CGate`(\*args, \*\*kwargs)

Version of CGate that allows gate simplifications. I.e. cnot looks like an oplus, cphase has dots, etc.

`sympy.physics.quantum.gate.CNOT`

alias of [CNotGate](#) (page 1843)

**class** `sympy.physics.quantum.gate.CNotGate`(\*args, \*\*kwargs)

Two qubit controlled-NOT.

This gate performs the NOT or X gate on the target qubit if the control qubits all have the value 1.

**Parameters**

**label** : tuple

A tuple of the form (control, target).

## Examples

```
>>> from sympy.physics.quantum.gate import CNOT
>>> from sympy.physics.quantum.qapply import qapply
>>> from sympy.physics.quantum.qubit import Qubit
>>> c = CNOT(1,0)
>>> qapply(c*Qubit('10')) # note that qubits are indexed from right to
→left
|11>
```

**property controls**

A tuple of control qubits.

**property gate**

The non-controlled gate that will be applied to the targets.

**property min\_qubits**

The minimum number of qubits this gate needs to act on.

**property targets**

A tuple of target qubits.

**class** `sympy.physics.quantum.gate.Gate`(\*args, \*\*kwargs)

Non-controlled unitary gate operator that acts on qubits.

This is a general abstract gate that needs to be subclassed to do anything useful.

**Parameters**

**label** : tuple, int

A list of the target qubits (as ints) that the gate will apply to.

**get\_target\_matrix**(*format*='sympy')

The matrix representation of the target part of the gate.

**Parameters**

**format** : str

The format string ('sympy', 'numpy', etc.)

**property min\_qubits**

The minimum number of qubits this gate needs to act on.

**property nqubits**

The total number of qubits this gate acts on.

For controlled gate subclasses this includes both target and control qubits, so that, for examples the CNOT gate acts on 2 qubits.

**property targets**

A tuple of target qubits.

`sympy.physics.quantum.gate.H`

alias of [HadamardGate](#) (page 1844)

**class** `sympy.physics.quantum.gate.HadamardGate(*args, **kwargs)`

The single qubit Hadamard gate.

**Parameters**

**target** : int

The target qubit this gate will apply to.

## Examples

```
>>> from sympy import sqrt
>>> from sympy.physics.quantum.qubit import Qubit
>>> from sympy.physics.quantum.gate import HadamardGate
>>> from sympy.physics.quantum.qapply import qapply
>>> qapply(HadamardGate(0)*Qubit('1'))
sqrt(2)*|0>/2 - sqrt(2)*|1>/2
>>> # Hadamard on bell state, applied on 2 qubits.
>>> psi = 1/sqrt(2)*(Qubit('00')+Qubit('11'))
>>> qapply(HadamardGate(0)*HadamardGate(1)*psi)
sqrt(2)*|00>/2 + sqrt(2)*|11>/2
```

**class** `sympy.physics.quantum.gate.IdentityGate(*args, **kwargs)`

The single qubit identity gate.

**Parameters**

**target** : int

The target qubit this gate will apply to.

**class** `sympy.physics.quantum.gate.OneQubitGate(*args, **kwargs)`

A single qubit unitary gate base class.

`sympy.physics.quantum.gate.Phase`

alias of [PhaseGate](#) (page 1844)

**class** sympy.physics.quantum.gate.**PhaseGate**(\*args, \*\*kwargs)

The single qubit phase, or S, gate.

This gate rotates the phase of the state by  $\pi/2$  if the state is  $|1\rangle$  and does nothing if the state is  $|0\rangle$ .

**Parameters**

**target** : int

The target qubit this gate will apply to.

sympy.physics.quantum.gate.**S**

alias of [PhaseGate](#) (page 1844)

sympy.physics.quantum.gate.**SWAP**

alias of [SwapGate](#) (page 1845)

**class** sympy.physics.quantum.gate.**SwapGate**(\*args, \*\*kwargs)

Two qubit SWAP gate.

This gate swap the values of the two qubits.

**Parameters**

**label** : tuple

A tuple of the form (target1, target2).

**decompose**(\*options)

Decompose the SWAP gate into CNOT gates.

sympy.physics.quantum.gate.**T**

alias of [TGate](#) (page 1845)

**class** sympy.physics.quantum.gate.**TGate**(\*args, \*\*kwargs)

The single qubit  $\pi/8$  gate.

This gate rotates the phase of the state by  $\pi/4$  if the state is  $|1\rangle$  and does nothing if the state is  $|0\rangle$ .

**Parameters**

**target** : int

The target qubit this gate will apply to.

**class** sympy.physics.quantum.gate.**TwoQubitGate**(\*args, \*\*kwargs)

A two qubit unitary gate base class.

**class** sympy.physics.quantum.gate.**UGate**(\*args, \*\*kwargs)

General gate specified by a set of targets and a target matrix.

**Parameters**

**label** : tuple

A tuple of the form (targets, U), where targets is a tuple of the target qubits and U is a unitary matrix with dimension of len(targets).

**get\_target\_matrix**(format='sympy')

The matrix rep. of the target part of the gate.

**Parameters**

**format** : str

The format string ('sympy', 'numpy', etc.)

**property targets**

A tuple of target qubits.

`sympy.physics.quantum.gate.X`

alias of [XGate](#) (page 1846)

**class** `sympy.physics.quantum.gate.XGate(*args, **kwargs)`

The single qubit X, or NOT, gate.

**Parameters**

**target** : int

The target qubit this gate will apply to.

`sympy.physics.quantum.gate.Y`

alias of [YGate](#) (page 1846)

**class** `sympy.physics.quantum.gate.YGate(*args, **kwargs)`

The single qubit Y gate.

**Parameters**

**target** : int

The target qubit this gate will apply to.

`sympy.physics.quantum.gate.Z`

alias of [ZGate](#) (page 1846)

**class** `sympy.physics.quantum.gate.ZGate(*args, **kwargs)`

The single qubit Z gate.

**Parameters**

**target** : int

The target qubit this gate will apply to.

`sympy.physics.quantum.gate.gate_simp(circuit)`

Simplifies gates symbolically

It first sorts gates using `gate_sort`. It then applies basic simplification rules to the circuit, e.g.,  $XGate^{**2} = Identity$

`sympy.physics.quantum.gate.gate_sort(circuit)`

Sorts the gates while keeping track of commutation relations

This function uses a bubble sort to rearrange the order of gate application. Keeps track of Quantum computations special commutation relations (e.g. things that apply to the same Qubit do not commute with each other)

circuit is the Mul of gates that are to be sorted.

`sympy.physics.quantum.gate.normalized(normalize)`

Set flag controlling normalization of Hadamard gates by  $1/\sqrt{2}$ .

This is a global setting that can be used to simplify the look of various expressions, by leaving off the leading  $1/\sqrt{2}$  of the Hadamard gate.

**Parameters**

**normalize** : bool

Should the Hadamard gate include the  $1/\sqrt{2}$  normalization factor? When True, the Hadamard gate will have the  $1/\sqrt{2}$ . When False, the Hadamard gate will not have this factor.

```
sympy.physics.quantum.gate.random_circuit(ngates, nqubits, gate_space=(<class
 'sympy.physics.quantum.gate.XGate'>,
 <class
 'sympy.physics.quantum.gate.YGate'>,
 <class
 'sympy.physics.quantum.gate.ZGate'>,
 <class
 'sympy.physics.quantum.gate.PhaseGate'>,
 <class
 'sympy.physics.quantum.gate.TGate'>,
 <class
 'sympy.physics.quantum.gate.HadamardGate'>,
 <class
 'sympy.physics.quantum.gate.CNotGate'>,
 <class
 'sympy.physics.quantum.gate.SwapGate'>))
```

Return a random circuit of ngates and nqubits.

This uses an equally weighted sample of (X, Y, Z, S, T, H, CNOT, SWAP) gates.

#### Parameters

**ngates** : int

The number of gates in the circuit.

**nqubits** : int

The number of qubits in the circuit.

**gate\_space** : tuple

A tuple of the gate classes that will be used in the circuit. Repeating gate classes multiple times in this tuple will increase the frequency they appear in the random circuit.

## Grover's Algorithm

Grover's algorithm and helper functions.

Todo:

- W gate construction (or perhaps -W gate based on Mermin's book)
- Generalize the algorithm for an unknown function that returns 1 on multiple qubit states, not just one.
- Implement `_represent_ZGate` in `OracleGate`

```
class sympy.physics.quantum.grover.OracleGate(*args, **kwargs)
```

A black box gate.

The gate marks the desired qubits of an unknown function by flipping the sign of the qubits. The unknown function returns true when it finds its desired qubits and false otherwise.

**Parameters****qubits** : int

Number of qubits.

**oracle** : callable

A callable function that returns a boolean on a computational basis.

**Examples**

Apply an Oracle gate that flips the sign of  $|2\rangle$  on different qubits:

```
>>> from sympy.physics.quantum.qubit import IntQubit
>>> from sympy.physics.quantum.qapply import qapply
>>> from sympy.physics.quantum.grover import OracleGate
>>> f = lambda qubits: qubits == IntQubit(2)
>>> v = OracleGate(2, f)
>>> qapply(v*IntQubit(2))
- |2>
>>> qapply(v*IntQubit(3))
|3>
```

**property search\_function**

The unknown function that helps find the sought after qubits.

**property targets**

A tuple of target qubits.

**class** sympy.physics.quantum.grover.WGate(\*args, \*\*kwargs)

General n qubit W Gate in Grover's algorithm.

The gate performs the operation  $2|\phi\rangle\langle\phi| - 1$  on some qubits.  $|\phi\rangle = (\text{tensor product of } n \text{ Hadamards}) * (|0\rangle \text{ with } n \text{ qubits})$

**Parameters****nqubits** : int

The number of qubits to operate on

sympy.physics.quantum.grover.**apply\_grover**(oracle, nqubits, iterations=None)

Applies grover's algorithm.

**Parameters****oracle** : callable

The unknown callable function that returns true when applied to the desired qubits and false otherwise.

**Returns****state** : Expr

The resulting state after Grover's algorithm has been iterated.



## Examples

Apply grover's algorithm to an even superposition of 2 qubits:

```
>>> from sympy.physics.quantum.qapply import qapply
>>> from sympy.physics.quantum.qubit import IntQubit
>>> from sympy.physics.quantum.grover import apply_grover
>>> f = lambda qubits: qubits == IntQubit(2)
>>> qapply(apply_grover(f, 2))
|2>
```

`sympy.physics.quantum.grover.grover_iteration(qstate, oracle)`

Applies one application of the Oracle and W Gate, WV.

### Parameters

**qstate** : Qubit

A superposition of qubits.

**oracle** : OracleGate

The black box operator that flips the sign of the desired basis qubits.

### Returns

**Qubit** : The qubits after applying the Oracle and W gate.

## Examples

Perform one iteration of grover's algorithm to see a phase change:

```
>>> from sympy.physics.quantum.qapply import qapply
>>> from sympy.physics.quantum.qubit import IntQubit
>>> from sympy.physics.quantum.grover import OracleGate
>>> from sympy.physics.quantum.grover import superposition_basis
>>> from sympy.physics.quantum.grover import grover_iteration
>>> numqubits = 2
>>> basis_states = superposition_basis(numqubits)
>>> f = lambda qubits: qubits == IntQubit(2)
>>> v = OracleGate(numqubits, f)
>>> qapply(grover_iteration(basis_states, v))
|2>
```

`sympy.physics.quantum.grover.superposition_basis(nqubits)`

Creates an equal superposition of the computational basis.

### Parameters

**nqubits** : int

The number of qubits.

### Returns

**state** : Qubit

An equal superposition of the computational basis with nqubits.

## Examples

Create an equal superposition of 2 qubits:

```
>>> from sympy.physics.quantum.grover import superposition_basis
>>> superposition_basis(2)
|0>/2 + |1>/2 + |2>/2 + |3>/2
```

## QFT

An implementation of qubits and gates acting on them.

Todo:

- Update docstrings.
- Update tests.
- Implement apply using decompose.
- Implement represent using decompose or something smarter. For this to work we first have to implement represent for SWAP.
- Decide if we want upper index to be inclusive in the constructor.
- Fix the printing of Rk gates in plotting.

```
class sympy.physics.quantum.qft.IQFT(*args, **kwargs)
```

The inverse quantum Fourier transform.

**decompose()**

Decomposes IQFT into elementary gates.

```
class sympy.physics.quantum.qft.QFT(*args, **kwargs)
```

The forward quantum Fourier transform.

**decompose()**

Decomposes QFT into elementary gates.

```
sympy.physics.quantum.qft.Rk
```

alias of [RkGate](#) (page 1850)

```
class sympy.physics.quantum.qft.RkGate(*args)
```

This is the R<sub>k</sub> gate of the QTF.

## Qubit

Qubits for quantum computing.

Todo: \* Finish implementing measurement logic. This should include POVM. \* Update docstrings. \* Update tests.

```
class sympy.physics.quantum.qubit.IntQubit(*args, **kwargs)
```

A qubit ket that store integers as binary numbers in qubit values.

The differences between this class and Qubit are:

- The form of the constructor.

- The qubit values are printed as their corresponding integer, rather than the raw qubit values. The internal storage format of the qubit values is the same as Qubit.

### Parameters

**values** : int, tuple

If a single argument, the integer we want to represent in the qubit values. This integer will be represented using the fewest possible number of qubits. If a pair of integers and the second value is more than one, the first integer gives the integer to represent in binary form and the second integer gives the number of qubits to use. List of zeros and ones is also accepted to generate qubit by bit pattern.

**nqubits** : int

The integer that represents the number of qubits. This number should be passed with keyword `nqubits=N`. You can use this in order to avoid ambiguity of Qubit-style tuple of bits. Please see the example below for more details.

### Examples

Create a qubit for the integer 5:

```
>>> from sympy.physics.quantum.qubit import IntQubit
>>> from sympy.physics.quantum.qubit import Qubit
>>> q = IntQubit(5)
>>> q
|5>
```

We can also create an IntQubit by passing a Qubit instance.

```
>>> q = IntQubit(Qubit('101'))
>>> q
|5>
>>> q.as_int()
5
>>> q.nqubits
3
>>> q.qubit_values
(1, 0, 1)
```

We can go back to the regular qubit form.

```
>>> Qubit(q)
|101>
```

Please note that IntQubit also accepts a Qubit-style list of bits. So, the code below yields qubits 3, not a single bit 1.

```
>>> IntQubit(1, 1)
|3>
```

To avoid ambiguity, use `nqubits` parameter. Use of this keyword is recommended especially when you provide the values by variables.

```
>>> IntQubit(1, nqubits=1)
|1>
>>> a = 1
>>> IntQubit(a, nqubits=1)
|1>
```

**class** sympy.physics.quantum.qubit.**IntQubitBra**(\*args, \*\*kwargs)

A qubit bra that store integers as binary numbers in qubit values.

**class** sympy.physics.quantum.qubit.**Qubit**(\*args, \*\*kwargs)

A multi-qubit ket in the computational (z) basis.

We use the normal convention that the least significant qubit is on the right, so  $|00001\rangle$  has a 1 in the least significant qubit.

#### Parameters

**values** : list, str

The qubit values as a list of ints ([0,0,0,1,1,]) or a string ('011').

### Examples

Create a qubit in a couple of different ways and look at their attributes:

```
>>> from sympy.physics.quantum.qubit import Qubit
>>> Qubit(0,0,0)
|000>
>>> q = Qubit('0101')
>>> q
|0101>
```

```
>>> q.nqubits
4
>>> len(q)
4
>>> q.dimension
4
>>> q.qubit_values
(0, 1, 0, 1)
```

We can flip the value of an individual qubit:

```
>>> q.flip(1)
|0111>
```

We can take the dagger of a Qubit to get a bra:

```
>>> from sympy.physics.quantum.dagger import Dagger
>>> Dagger(q)
<0101|
>>> type(Dagger(q))
<class 'sympy.physics.quantum.qubit.QubitBra'>
```

Inner products work as expected:

```
>>> ip = Dagger(q)*q
>>> ip
<0101|0101>
>>> ip.doit()
1
```

**class** `sympy.physics.quantum.qubit.QubitBra(*args, **kwargs)`

A multi-qubit bra in the computational (z) basis.

We use the normal convention that the least significant qubit is on the right, so  $|00001\rangle$  has a 1 in the least significant qubit.

**Parameters**

**values** : list, str

The qubit values as a list of ints ([0,0,0,1,1,]) or a string ('011').

**See also:**

[Qubit \(page 1852\)](#)

Examples using qubits

`sympy.physics.quantum.qubit.matrix_to_density(mat)`

Works by finding the eigenvectors and eigenvalues of the matrix. We know we can decompose rho by doing:  $\text{sum}(\text{EigenVal} * |\text{Eigenvect}\rangle \langle \text{Eigenvect}|)$

`sympy.physics.quantum.qubit.matrix_to_qubit(matrix)`

Convert from the matrix repr. to a sum of Qubit objects.

**Parameters**

**matrix** : Matrix, numpy.matrix, scipy.sparse

The matrix to build the Qubit representation of. This works with SymPy matrices, numpy matrices and scipy.sparse sparse matrices.

**Examples**

Represent a state and then go back to its qubit form:

```
>>> from sympy.physics.quantum.qubit import matrix_to_qubit, Qubit
>>> from sympy.physics.quantum.represent import represent
>>> q = Qubit('01')
>>> matrix_to_qubit(represent(q))
|01>
```

`sympy.physics.quantum.qubit.measure_all(qubit, format='sympy', normalize=True)`

Perform an ensemble measurement of all qubits.

**Parameters**

**qubit** : Qubit, Add

The qubit to measure. This can be any Qubit or a linear combination of them.

**format** : str

The format of the intermediate matrices to use. Possible values are ('sympy', 'numpy', 'scipy.sparse'). Currently only 'sympy' is implemented.

**Returns**

**result** : list

A list that consists of primitive states and their probabilities.

**Examples**

```
>>> from sympy.physics.quantum.qubit import Qubit, measure_all
>>> from sympy.physics.quantum.gate import H
>>> from sympy.physics.quantum.qapply import qapply
```

```
>>> c = H(0)*H(1)*Qubit('00')
>>> c
H(0)*H(1)*|00>
>>> q = qapply(c)
>>> measure_all(q)
[(|00>, 1/4), (|01>, 1/4), (|10>, 1/4), (|11>, 1/4)]
```

`sympy.physics.quantum.qubit.measure_all_oneshot(qubit, format='sympy')`

Perform a oneshot ensemble measurement on all qubits.

A oneshot measurement is equivalent to performing a measurement on a quantum system. This type of measurement does not return the probabilities like an ensemble measurement does, but rather returns *one* of the possible resulting states. The exact state that is returned is determined by picking a state randomly according to the ensemble probabilities.

**Parameters**

**qubits** : Qubit

The qubit to measure. This can be any Qubit or a linear combination of them.

**format** : str

The format of the intermediate matrices to use. Possible values are ('sympy', 'numpy', 'scipy.sparse'). Currently only 'sympy' is implemented.

**Returns**

**result** : Qubit

The qubit that the system collapsed to upon measurement.

`sympy.physics.quantum.qubit.measure_partial(qubit, bits, format='sympy',  
 normalize=True)`

Perform a partial ensemble measure on the specified qubits.

**Parameters**

**qubits** : Qubit

The qubit to measure. This can be any Qubit or a linear combination of them.

**bits** : tuple

The qubits to measure.

**format** : str

The format of the intermediate matrices to use. Possible values are ('sympy', 'numpy', 'scipy.sparse'). Currently only 'sympy' is implemented.

### Returns

**result** : list

A list that consists of primitive states and their probabilities.

### Examples

```
>>> from sympy.physics.quantum.qubit import Qubit, measure_partial
>>> from sympy.physics.quantum.gate import H
>>> from sympy.physics.quantum.qapply import qapply
```

```
>>> c = H(0)*H(1)*Qubit('00')
>>> c
H(0)*H(1)*|00>
>>> q = qapply(c)
>>> measure_partial(q, (0,))
[(sqrt(2)*|00>/2 + sqrt(2)*|10>/2, 1/2), (sqrt(2)*|01>/2 + sqrt(2)*|11>/
→ 2, 1/2)]
```

`sympy.physics.quantum.qubit.measure_partial_oneshot(qubit, bits, format='sympy')`

Perform a partial oneshot measurement on the specified qubits.

A oneshot measurement is equivalent to performing a measurement on a quantum system. This type of measurement does not return the probabilities like an ensemble measurement does, but rather returns *one* of the possible resulting states. The exact state that is returned is determined by picking a state randomly according to the ensemble probabilities.

### Parameters

**qubits** : Qubit

The qubit to measure. This can be any Qubit or a linear combination of them.

**bits** : tuple

The qubits to measure.

**format** : str

The format of the intermediate matrices to use. Possible values are ('sympy', 'numpy', 'scipy.sparse'). Currently only 'sympy' is implemented.

### Returns

**result** : Qubit

The qubit that the system collapsed to upon measurement.

`sympy.physics.quantum.qubit.qubit_to_matrix(qubit, format='sympy')`

Converts an Add/Mul of Qubit objects into it's matrix representation

This function is the inverse of `matrix_to_qubit` and is a shorthand for `represent(qubit)`.

## Shor's Algorithm

Shor's algorithm and helper functions.

Todo:

- Get the CMod gate working again using the new Gate API.
- Fix everything.
- Update docstrings and reformat.

**class** `sympy.physics.quantum.shor.CMod(*args, **kwargs)`

A controlled mod gate.

This is black box controlled Mod function for use by shor's algorithm. TODO: implement a decompose property that returns how to do this in terms of elementary gates

**property** `N`

`N` is the type of modular arithmetic we are doing.

**property** `a`

Base of the controlled mod function.

**property** `t`

Size of 1/2 input register. First 1/2 holds output.

`sympy.physics.quantum.shor.period_find(a, N)`

Finds the period of `a` in modulo `N` arithmetic

This is quantum part of Shor's algorithm. It takes two registers, puts first in superposition of states with Hadamards so:  $|k\rangle|0\rangle$  with `k` being all possible choices. It then does a controlled mod and a QFT to determine the order of `a`.

`sympy.physics.quantum.shor.shor(N)`

This function implements Shor's factoring algorithm on the Integer `N`

The algorithm starts by picking a random number (`a`) and seeing if it is coprime with `N`. If it is not, then the gcd of the two numbers is a factor and we are done. Otherwise, it begins the `period_finding` subroutine which finds the period of `a` in modulo `N` arithmetic. This period, if even, can be used to calculate factors by taking  $a^{(r/2)-1}$  and  $a^{(r/2)+1}$ . These values are returned.



## Analytic Solutions

### Particle in a Box

1D quantum particle in a box.

```
class sympy.physics.quantum.piab.PIABBra(*args, **kwargs)
```

Particle in a box eigenbra.

```
class sympy.physics.quantum.piab.PIABHamiltonian(*args, **kwargs)
```

Particle in a box Hamiltonian operator.

```
class sympy.physics.quantum.piab.PIABKet(*args, **kwargs)
```

Particle in a box eigenket.

## Optics Module

### Abstract

Contains docstrings of Physics-Optics module

### Gaussian Optics

Gaussian optics.

The module implements:

- Ray transfer matrices for geometrical and gaussian optics.  
See RayTransferMatrix, GeometricRay and BeamParameter
- Conjugation relations for geometrical and gaussian optics.  
See geometric\_conj\*, gauss\_conj and conjugate\_gauss\_beams

The conventions for the distances are as follows:

#### **focal distance**

positive for convergent lenses

#### **object distance**

positive for real objects

#### **image distance**

positive for real images

```
class sympy.physics.optics.gaussopt.BeamParameter(wavelen, z, z_r=None, w=None, n=1)
```

Representation for a gaussian ray in the Ray Transfer Matrix formalism.

#### **Parameters**

**wavelen** : the wavelength,  
**z** : the distance to waist, and  
**w** : the waist, or

**z\_r** : the rayleigh range.

**n** : the refractive index of medium.

### Examples

```
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.q
1 + 1.88679245283019*I*pi
```

```
>>> p.q.n()
1.0 + 5.92753330865999*I
>>> p.w_0.n()
0.00100000000000000
>>> p.z_r.n()
5.92753330865999
```

```
>>> from sympy.physics.optics import FreeSpace
>>> fs = FreeSpace(10)
>>> p1 = fs*p
>>> p.w.n()
0.00101413072159615
>>> p1.w.n()
0.00210803120913829
```

### See also:

[\*RayTransferMatrix\*](#) (page 1863)

### References

[R660], [R661]

#### property **divergence**

Half of the total angular spread.

### Examples

```
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.divergence
0.00053/pi
```

#### property **gouy**

The Gouy phase.

## Examples

```
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.gouy
atan(0.53/pi)
```

### property q

The complex parameter representing the beam.

## Examples

```
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.q
1 + 1.88679245283019*I*pi
```

### property radius

The radius of curvature of the phase front.

## Examples

```
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.radius
1 + 3.55998576005696*pi**2
```

### property w

The radius of the beam  $w(z)$ , at any position  $z$  along the beam. The beam radius at  $1/e^2$  intensity (axial value).

## Examples

```
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.w
0.001*sqrt(0.2809/pi**2 + 1)
```

### See also:

#### [\$w\_0\$](#) (page 1859)

The minimal radius of beam.

### property $w_0$

The minimal radius of beam at  $1/e^2$  intensity (peak value).

### Examples

```
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.w_0
0.0010000000000000000
```

#### See also:

##### [\*w\*](#) (page 1859)

the beam radius at  $1/e^2$  intensity (axial value).

#### property `waist_approximation_limit`

The minimal waist for which the gauss beam approximation is valid.

### Explanation

The gauss beam is a solution to the paraxial equation. For curvatures that are too great it is not a valid approximation.

### Examples

```
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.waist_approximation_limit
1.06e-6/pi
```

**class** `sympy.physics.optics.gaussopt.CurvedMirror(R)`

Ray Transfer Matrix for reflection from curved surface.

#### Parameters

***R*** : radius of curvature (positive for concave)

### Examples

```
>>> from sympy.physics.optics import CurvedMirror
>>> from sympy import symbols
>>> R = symbols('R')
>>> CurvedMirror(R)
Matrix([
[1, 0],
[-2/R, 1]])
```

#### See also:

[\*RayTransferMatrix\*](#) (page 1863)

**class** `sympy.physics.optics.gaussopt.CurvedRefraction(R, n1, n2)`

Ray Transfer Matrix for refraction on curved interface.

**Parameters****R :**

Radius of curvature (positive for concave).

**n1 :**

Refractive index of one medium.

**n2 :**

Refractive index of other medium.

**Examples**

```
>>> from sympy.physics.optics import CurvedRefraction
>>> from sympy import symbols
>>> R, n1, n2 = symbols('R n1 n2')
>>> CurvedRefraction(R, n1, n2)
Matrix([
[1, 0],
[(n1 - n2)/(R*n2), n1/n2]])
```

**See also:**[\*RayTransferMatrix\*](#) (page 1863)**class** sympy.physics.optics.gaussopt.FlatMirror

Ray Transfer Matrix for reflection.

**Examples**

```
>>> from sympy.physics.optics import FlatMirror
>>> FlatMirror()
Matrix([
[1, 0],
[0, 1]])
```

**See also:**[\*RayTransferMatrix\*](#) (page 1863)**class** sympy.physics.optics.gaussopt.FlatRefraction(*n1*, *n2*)

Ray Transfer Matrix for refraction.

**Parameters****n1 :**

Refractive index of one medium.

**n2 :**

Refractive index of other medium.

### Examples

```
>>> from sympy.physics.optics import FlatRefraction
>>> from sympy import symbols
>>> n1, n2 = symbols('n1 n2')
>>> FlatRefraction(n1, n2)
Matrix([
[1, 0],
[0, n1/n2]])
```

#### See also:

[\*RayTransferMatrix\*](#) (page 1863)

**class** sympy.physics.optics.gaussopt.**FreeSpace**(*d*)

Ray Transfer Matrix for free space.

#### Parameters

**distance**

### Examples

```
>>> from sympy.physics.optics import FreeSpace
>>> from sympy import symbols
>>> d = symbols('d')
>>> FreeSpace(d)
Matrix([
[1, d],
[0, 1]])
```

#### See also:

[\*RayTransferMatrix\*](#) (page 1863)

**class** sympy.physics.optics.gaussopt.**GeometricRay**(\**args*)

Representation for a geometric ray in the Ray Transfer Matrix formalism.

#### Parameters

**h** : height, and

**angle** : angle, or

**matrix** : a 2x1 matrix (Matrix(2, 1, [height, angle]))

### Examples

```
>>> from sympy.physics.optics import GeometricRay, FreeSpace
>>> from sympy import symbols, Matrix
>>> d, h, angle = symbols('d, h, angle')
```

```
>>> GeometricRay(h, angle)
Matrix([
[h],
[angle]])
```

```
>>> FreeSpace(d)*GeometricRay(h, angle)
Matrix([
[angle*d + h],
[angle]])
```

```
>>> GeometricRay(Matrix(((h,), (angle,))))
Matrix([
[h],
[angle]])
```

**See also:**

[\*RayTransferMatrix\*](#) (page 1863)

**property angle**

The angle with the optical axis.

**Examples**

```
>>> from sympy.physics.optics import GeometricRay
>>> from sympy import symbols
>>> h, angle = symbols('h, angle')
>>> gRay = GeometricRay(h, angle)
>>> gRay.angle
angle
```

**property height**

The distance from the optical axis.

**Examples**

```
>>> from sympy.physics.optics import GeometricRay
>>> from sympy import symbols
>>> h, angle = symbols('h, angle')
>>> gRay = GeometricRay(h, angle)
>>> gRay.height
h
```

**class** sympy.physics.optics.gaussopt.**RayTransferMatrix**(\*args)

Base class for a Ray Transfer Matrix.

It should be used if there is not already a more specific subclass mentioned in See Also.

**Parameters****parameters :**

A, B, C and D or 2x2 matrix (Matrix(2, 2, [A, B, C, D]))

## Examples

```
>>> from sympy.physics.optics import RayTransferMatrix, ThinLens
>>> from sympy import Symbol, Matrix
```

```
>>> mat = RayTransferMatrix(1, 2, 3, 4)
>>> mat
Matrix([
[1, 2],
[3, 4]])
```

```
>>> RayTransferMatrix(Matrix([[1, 2], [3, 4]]))
Matrix([
[1, 2],
[3, 4]])
```

```
>>> mat.A
1
```

```
>>> f = Symbol('f')
>>> lens = ThinLens(f)
>>> lens
Matrix([
[1, 0],
[-1/f, 1]])
```

```
>>> lens.C
-1/f
```

### See also:

[GeometricRay](#) (page 1862), [BeamParameter](#) (page 1857), [FreeSpace](#) (page 1862), [FlatRefraction](#) (page 1861), [CurvedRefraction](#) (page 1860), [FlatMirror](#) (page 1861), [CurvedMirror](#) (page 1860), [ThinLens](#) (page 1865)

## References

[R662]

### property A

The A parameter of the Matrix.



### Examples

```
>>> from sympy.physics.optics import RayTransferMatrix
>>> mat = RayTransferMatrix(1, 2, 3, 4)
>>> mat.A
1
```

#### property B

The B parameter of the Matrix.

### Examples

```
>>> from sympy.physics.optics import RayTransferMatrix
>>> mat = RayTransferMatrix(1, 2, 3, 4)
>>> mat.B
2
```

#### property C

The C parameter of the Matrix.

### Examples

```
>>> from sympy.physics.optics import RayTransferMatrix
>>> mat = RayTransferMatrix(1, 2, 3, 4)
>>> mat.C
3
```

#### property D

The D parameter of the Matrix.

### Examples

```
>>> from sympy.physics.optics import RayTransferMatrix
>>> mat = RayTransferMatrix(1, 2, 3, 4)
>>> mat.D
4
```

**class** sympy.physics.optics.gaussopt.**ThinLens**(*f*)

Ray Transfer Matrix for a thin lens.

#### Parameters

**f** :

The focal distance.

## Examples

```
>>> from sympy.physics.optics import ThinLens
>>> from sympy import symbols
>>> f = symbols('f')
>>> ThinLens(f)
Matrix([
[1, 0],
[-1/f, 1]])
```

### See also:

[\*RayTransferMatrix\*](#) (page 1863)

`sympy.physics.optics.gaussopt.conjugate_gauss_beams(wavelen, waist_in, waist_out, **kwargs)`

Find the optical setup conjugating the object/image waists.

### Parameters

#### **wavelen :**

The wavelength of the beam.

#### **waist\_in and waist\_out :**

The waists to be conjugated.

#### **f :**

The focal distance of the element used in the conjugation.

### Returns

a tuple containing (s\_in, s\_out, f)

#### **s\_in :**

The distance before the optical element.

#### **s\_out :**

The distance after the optical element.

#### **f :**

The focal distance of the optical element.

## Examples

```
>>> from sympy.physics.optics import conjugate_gauss_beams
>>> from sympy import symbols, factor
>>> l, w_i, w_o, f = symbols('l w_i w_o f')
```

```
>>> conjugate_gauss_beams(l, w_i, w_o, f=f)[0]
f*(1 - sqrt(w_i**2/w_o**2 - pi**2*w_i**4/(f**2*l**2)))
```

```
>>> factor(conjugate_gauss_beams(l, w_i, w_o, f=f)[1])
f*w_o**2*(w_i**2/w_o**2 - sqrt(w_i**2/w_o**2 -
pi**2*w_i**4/(f**2*l**2)))/w_i**2
```

```
>>> conjugate_gauss_beams(l, w_i, w_o, f=f)[2]
f
```

`sympy.physics.optics.gaussopt.gaussian_conj(s_in, z_r_in, f)`

Conjugation relation for gaussian beams.

#### Parameters

**s\_in :**

The distance to optical element from the waist.

**z\_r\_in :**

The rayleigh range of the incident beam.

**f :**

The focal length of the optical element.

#### Returns

a tuple containing (s\_out, z\_r\_out, m)

**s\_out :**

The distance between the new waist and the optical element.

**z\_r\_out :**

The rayleigh range of the emergent beam.

**m :**

The ration between the new and the old waists.

#### Examples

```
>>> from sympy.physics.optics import gaussian_conj
>>> from sympy import symbols
>>> s_in, z_r_in, f = symbols('s_in z_r_in f')
```

```
>>> gaussian_conj(s_in, z_r_in, f)[0]
1/(-1/(s_in + z_r_in**2/(-f + s_in)) + 1/f)
```

```
>>> gaussian_conj(s_in, z_r_in, f)[1]
z_r_in/(1 - s_in**2/f**2 + z_r_in**2/f**2)
```

```
>>> gaussian_conj(s_in, z_r_in, f)[2]
1/sqrt(1 - s_in**2/f**2 + z_r_in**2/f**2)
```

`sympy.physics.optics.gaussopt.geometric_conj_ab(a, b)`

Conjugation relation for geometrical beams under paraxial conditions.

### Explanation

Takes the distances to the optical element and returns the needed focal distance.

### Examples

```
>>> from sympy.physics.optics import geometric_conj_ab
>>> from sympy import symbols
>>> a, b = symbols('a b')
>>> geometric_conj_ab(a, b)
a*b/(a + b)
```

#### See also:

[\*geometric\\_conj\\_af\*](#) (page 1868), [\*geometric\\_conj\\_bf\*](#) (page 1868)

`sympy.physics.optics.gaussopt.geometric_conj_af(a, f)`

Conjugation relation for geometrical beams under paraxial conditions.

### Explanation

Takes the object distance (for `geometric_conj_af`) or the image distance (for `geometric_conj_bf`) to the optical element and the focal distance. Then it returns the other distance needed for conjugation.

### Examples

```
>>> from sympy.physics.optics.gaussopt import geometric_conj_af, g
↳ geometric_conj_bf
>>> from sympy import symbols
>>> a, b, f = symbols('a b f')
>>> geometric_conj_af(a, f)
a*f/(a - f)
>>> geometric_conj_bf(b, f)
b*f/(b - f)
```

#### See also:

[\*geometric\\_conj\\_ab\*](#) (page 1867)

`sympy.physics.optics.gaussopt.geometric_conj_bf(a, f)`

Conjugation relation for geometrical beams under paraxial conditions.

## Explanation

Takes the object distance (for `geometric_conj_af`) or the image distance (for `geometric_conj_bf`) to the optical element and the focal distance. Then it returns the other distance needed for conjugation.

## Examples

```
>>> from sympy.physics.optics.gaussopt import geometric_conj_af,
 geometric_conj_bf
>>> from sympy import symbols
>>> a, b, f = symbols('a b f')
>>> geometric_conj_af(a, f)
a*f/(a - f)
>>> geometric_conj_bf(b, f)
b*f/(b - f)
```

### See also:

[`geometric\_conj\_ab`](#) (page 1867)

`sympy.physics.optics.gaussopt.rayleigh2waist(z_r, wavelen)`  
Calculate the waist from the rayleigh range of a gaussian beam.

## Examples

```
>>> from sympy.physics.optics import rayleigh2waist
>>> from sympy import symbols
>>> z_r, wavelen = symbols('z_r wavelen')
>>> rayleigh2waist(z_r, wavelen)
sqrt(wavelen*z_r)/sqrt(pi)
```

### See also:

[`waist2rayleigh`](#) (page 1869), [`BeamParameter`](#) (page 1857)

`sympy.physics.optics.gaussopt.waist2rayleigh(w, wavelen, n=1)`  
Calculate the rayleigh range from the waist of a gaussian beam.

## Examples

```
>>> from sympy.physics.optics import waist2rayleigh
>>> from sympy import symbols
>>> w, wavelen = symbols('w wavelen')
>>> waist2rayleigh(w, wavelen)
pi*w**2/wavelen
```

### See also:

[`rayleigh2waist`](#) (page 1869), [`BeamParameter`](#) (page 1857)

## Medium

### Contains

- Medium

**class** sympy.physics.optics.medium.**Medium**(*name*, *permittivity*=None, *permeability*=None, *n*=None)

This class represents an optical medium. The prime reason to implement this is to facilitate refraction, Fermat's principle, etc.

#### Parameters

**name:** string

The display name of the Medium.

**permittivity:** Sympifyable

Electric permittivity of the space.

**permeability:** Sympifyable

Magnetic permeability of the space.

**n:** Sympifyable

Index of refraction of the medium.

### Explanation

An optical medium is a material through which electromagnetic waves propagate. The permittivity and permeability of the medium define how electromagnetic waves propagate in it.

### Examples

```
>>> from sympy.abc import epsilon, mu
>>> from sympy.physics.optics import Medium
>>> m1 = Medium('m1')
>>> m2 = Medium('m2', epsilon, mu)
>>> m1.intrinsic_impedance
149896229*pi*kilogram*meter**2/(1250000*ampere**2*second**3)
>>> m2.refractive_index
299792458*meter*sqrt(epsilon*mu)/second
```

## References

[R663]

### property `refractive_index`

Returns refractive index of the medium.

### Examples

```
>>> from sympy.physics.optics import Medium
>>> m = Medium('m')
>>> m.refractive_index
1
```

### property `speed`

Returns speed of the electromagnetic wave travelling in the medium.

### Examples

```
>>> from sympy.physics.optics import Medium
>>> m = Medium('m')
>>> m.speed
299792458*meter/second
>>> m2 = Medium('m2', n=1)
>>> m.speed == m2.speed
True
```

## Polarization

The module implements routines to model the polarization of optical fields and can be used to calculate the effects of polarization optical elements on the fields.

- Jones vectors.
- Stokes vectors.
- Jones matrices.
- Mueller matrices.

### Examples

We calculate a generic Jones vector:

```
>>> from sympy import symbols, pprint, zeros, simplify
>>> from sympy.physics.optics.polarization import (jones_vector, stokes_
→ vector,
... half_wave_retarder, polarizing_beam_splitter, jones_2_stokes)
```

```
>>> psi, chi, p, I0 = symbols("psi, chi, p, I0", real=True)
>>> x0 = jones_vector(psi, chi)
>>> pprint(x0, use_unicode=True)

$$\begin{bmatrix} -i \cdot \sin(\chi) \cdot \sin(\psi) + \cos(\chi) \cdot \cos(\psi) \\ i \cdot \sin(\chi) \cdot \cos(\psi) + \sin(\psi) \cdot \cos(\chi) \end{bmatrix}$$

```

And the more general Stokes vector: `>>> s0 = stokes_vector(psi, chi, p, I0) >>> pprint(s0, use_unicode=True)`

$$\begin{bmatrix} I_0 \\ I_0 \cdot p \cdot \cos(2 \cdot \chi) \cdot \cos(2 \cdot \psi) \\ I_0 \cdot p \cdot \sin(2 \cdot \psi) \cdot \cos(2 \cdot \chi) \\ I_0 \cdot p \cdot \sin(2 \cdot \chi) \end{bmatrix}$$

We calculate how the Jones vector is modified by a half-wave plate: `>>> alpha = symbols("alpha", real=True) >>> HWP = half_wave_retarder(alpha) >>> x1 = simplify(HWP*x0)`

We calculate the very common operation of passing a beam through a half-wave plate and then through a polarizing beam-splitter. We do this by putting this Jones vector as the first entry of a two-Jones-vector state that is transformed by a 4x4 Jones matrix modelling the polarizing beam-splitter to get the transmitted and reflected Jones vectors:

```
>>> PBS = polarizing_beam_splitter()
>>> X1 = zeros(4, 1)
>>> X1[:2, :] = x1
>>> X2 = PBS*X1
>>> transmitted_port = X2[:2, :]
>>> reflected_port = X2[2:, :]
```

This allows us to calculate how the power in both ports depends on the initial polarization:

```
>>> transmitted_power = jones_2_stokes(transmitted_port)[0]
>>> reflected_power = jones_2_stokes(reflected_port)[0]
>>> print(transmitted_power)
cos(-2*alpha + chi + psi)**2/2 + cos(2*alpha + chi - psi)**2/2
```

```
>>> print(reflected_power)
sin(-2*alpha + chi + psi)**2/2 + sin(2*alpha + chi - psi)**2/2
```

Please see the description of the individual functions for further details and examples.

## References

`sympy.physics.optics.polarization.half_wave_retarder(theta)`

A half-wave retarder Jones matrix at angle *theta*.

### Parameters

`theta` : numeric type or SymPy Symbol

The angle of the fast axis relative to the horizontal plane.

### Returns

SymPy Matrix

A Jones matrix representing the retarder.



## Examples

A generic half-wave plate.

```
>>> from sympy import pprint, symbols
>>> from sympy.physics.optics.polarization import half_wave_retarder
>>> theta= symbols("theta", real=True)
>>> HWP = half_wave_retarder(theta)
>>> pprint(HWP, use_unicode=True)

$$\begin{bmatrix} \begin{pmatrix} 2 & 2 \\ -i \cdot (-\sin(\theta) + \cos(\theta)) & -2 \cdot i \cdot \sin(\theta) \cdot \cos(\theta) \end{pmatrix} \\ -2 \cdot i \cdot \sin(\theta) \cdot \cos(\theta) & -i \cdot \begin{pmatrix} 2 & 2 \\ \sin(\theta) - \cos(\theta) \end{pmatrix} \end{bmatrix}$$

```

`sympy.physics.optics.polarization.jones_2_stokes(e)`

Return the Stokes vector for a Jones vector *e*.

### Parameters

```e``` : SymPy Matrix

A Jones vector.

Returns

SymPy Matrix

A Jones vector.

Examples

The axes on the Poincaré sphere.

```
>>> from sympy import pprint, pi
>>> from sympy.physics.optics.polarization import jones_vector
>>> from sympy.physics.optics.polarization import jones_2_stokes
>>> H = jones_vector(0, 0)
>>> V = jones_vector(pi/2, 0)
>>> D = jones_vector(pi/4, 0)
>>> A = jones_vector(-pi/4, 0)
>>> R = jones_vector(0, pi/4)
>>> L = jones_vector(0, -pi/4)
>>> pprint([jones_2_stokes(e) for e in [H, V, D, A, R, L]],
...         use_unicode=True)

$$\begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ -1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 0 \\ -1 \end{bmatrix}$$

```

`sympy.physics.optics.polarization.jones_vector(psi, chi)`

A Jones vector corresponding to a polarization ellipse with *psi* tilt, and *chi* circularity.

Parameters

`psi` : numeric type or SymPy Symbol

The tilt of the polarization relative to the x axis.

`chi` : numeric type or SymPy Symbol

The angle adjacent to the mayor axis of the polarization ellipse.

Returns

Matrix :

A Jones vector.

Examples

The axes on the Poincaré sphere.

```
>>> from sympy import pprint, symbols, pi
>>> from sympy.physics.optics.polarization import jones_vector
>>> psi, chi = symbols("psi, chi", real=True)
```

A general Jones vector. `>>> pprint(jones_vector(psi, chi), use_unicode=True)` $\begin{bmatrix} -i\sin(\chi)\sin(\psi) + \cos(\chi)\cos(\psi) \\ i\sin(\chi)\cos(\psi) + \sin(\psi)\cos(\chi) \end{bmatrix}$

Horizontal polarization. `>>> pprint(jones_vector(0, 0), use_unicode=True)` $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$

Vertical polarization. `>>> pprint(jones_vector(pi/2, 0), use_unicode=True)` $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$

Diagonal polarization. `>>> pprint(jones_vector(pi/4, 0), use_unicode=True)` $\frac{1}{\sqrt{2}} \begin{bmatrix} \sqrt{2} \\ 1 \end{bmatrix}$

Anti-diagonal polarization. `>>> pprint(jones_vector(-pi/4, 0), use_unicode=True)` $\frac{1}{\sqrt{2}} \begin{bmatrix} \sqrt{2} \\ -1 \end{bmatrix}$

Right-hand circular polarization. `>>> pprint(jones_vector(0, pi/4), use_unicode=True)` $\frac{1}{\sqrt{2}} \begin{bmatrix} \sqrt{2} \\ i \end{bmatrix}$

Left-hand circular polarization. `>>> pprint(jones_vector(0, -pi/4), use_unicode=True)` $\frac{1}{\sqrt{2}} \begin{bmatrix} \sqrt{2} \\ -i \end{bmatrix}$

`sympy.physics.optics.polarization.linear_polarizer(theta=0)`

A linear polarizer Jones matrix with transmission axis at an angle θ .

Parameters

`theta` : numeric type or SymPy Symbol

The angle of the transmission axis relative to the horizontal plane.

Returns

SymPy Matrix

A Jones matrix representing the polarizer.

Examples

A generic polarizer.

```
>>> from sympy import pprint, symbols
>>> from sympy.physics.optics.polarization import linear_polarizer
>>> theta = symbols("theta", real=True)
>>> J = linear_polarizer(theta)
>>> pprint(J, use_unicode=True)

$$\begin{bmatrix} \cos^2(\theta) & \sin(\theta)\cos(\theta) \\ \sin(\theta)\cos(\theta) & \sin^2(\theta) \end{bmatrix}$$

```

`sympy.physics.optics.polarization.mueller_matrix(J)`

The Mueller matrix corresponding to Jones matrix J .

Parameters

`J` : SymPy Matrix

A Jones matrix.

Returns

SymPy Matrix

The corresponding Mueller matrix.

Examples

Generic optical components.

```
>>> from sympy import pprint, symbols
>>> from sympy.physics.optics.polarization import (mueller_matrix,
...         linear_polarizer, half_wave_retarder, quarter_wave_retarder)
>>> theta = symbols("theta", real=True)
```

```
A linear_polarizer >>> pprint(mueller_matrix(linear_polarizer(theta)),
use_unicode=True)

$$\begin{bmatrix} \cos(2\theta) & \sin(2\theta) & 0 & 0 \\ \cos(2\theta)\cos(4\theta) & \sin(2\theta)\cos(4\theta) & 0 & 0 \\ \sin(2\theta)\cos(4\theta) & \sin(2\theta)\sin(4\theta) & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

```

```
A half-wave plate >>> pprint(mueller_matrix(half_wave_retarder(theta)),
use_unicode=True)

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(4\theta) & \sin(4\theta) & 0 \\ 0 & \sin(4\theta) & -\cos(4\theta) & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

```

```
A quarter-wave plate >>> pprint(mueller_matrix(quarter_wave_retarder(theta)),
use_unicode=True)

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(4\theta) & \sin(4\theta) & 0 \\ 0 & \sin(4\theta) & -\cos(4\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```

`sympy.physics.optics.polarization.phase_retarder(theta=0, delta=0)`

A phase retarder Jones matrix with retardance δ at angle θ .

Parameters

```theta``` : numeric type or SymPy Symbol

The angle of the fast axis relative to the horizontal plane.

```delta``` : numeric type or SymPy Symbol

The phase difference between the fast and slow axes of the transmitted light.

Returns

SymPy Matrix :

A Jones matrix representing the retarder.

Examples

A generic retarder.

```
>>> from sympy import pprint, symbols
>>> from sympy.physics.optics.polarization import phase_retarder
>>> theta, delta = symbols("theta, delta", real=True)
>>> R = phase_retarder(theta, delta)
>>> pprint(R, use_unicode=True)
⎡⎛ i⋅δ  2      2      ⎞      -i⋅δ      -i⋅δ
⎛ e  ⋅sin(θ) + cos(θ) ⎟⋅e  ⎛ 1 - e  ⎟⋅e  ⋅sin(θ)⋅cos(θ)
⎛  i⋅δ  2      2      ⎞      -i⋅δ      -i⋅δ
⎛ 1 - e  ⎟⋅e  ⋅sin(θ)⋅cos(θ) ⎛ e  ⋅cos(θ) + sin(θ) ⎟⋅e  ⎡
```

```
sympy.physics.optics.polarization.polarizing_beam_splitter(Tp=1, Rs=1, Ts=0,
                                                             Rp=0, phia=0,
                                                             phib=0)
```

A polarizing beam splitter Jones matrix at angle *theta*.

Parameters

```J``` : SymPy Matrix

A Jones matrix.

```Tp``` : numeric type or SymPy Symbol

The transmissivity of the P-polarized component.

```Rs``` : numeric type or SymPy Symbol

The reflectivity of the S-polarized component.

```Ts``` : numeric type or SymPy Symbol

The transmissivity of the S-polarized component.

```Rp``` : numeric type or SymPy Symbol

The reflectivity of the P-polarized component.

```phia``` : numeric type or SymPy Symbol

The phase difference between transmitted and reflected component for output mode a.

```phib``` : numeric type or SymPy Symbol

The phase difference between transmitted and reflected component for output mode b.

### Returns

SymPy Matrix

A 4x4 matrix representing the PBS. This matrix acts on a 4x1 vector whose first two entries are the Jones vector on one of the PBS ports, and the last two entries the Jones vector on the other port.

### Examples

Generic polarizing beam-splitter.

```
>>> from sympy import pprint, symbols
>>> from sympy.physics.optics.polarization import polarizing_beam_
 ↪ splitter
>>> Ts, Rs, Tp, Rp = symbols(r"Ts, Rs, Tp, Rp", positive=True)
>>> phia, phib = symbols("phi_a, phi_b", real=True)
>>> PBS = polarizing_beam_splitter(Tp, Rs, Ts, Rp, phia, phib)
>>> pprint(PBS, use_unicode=False)
[
[\sqrt{Tp} 0 I*\sqrt{Rp} 0]
[
[0 \sqrt{Ts} 0 -I*\sqrt{Rs}*e^{I*phi_a}
[
[I*\sqrt{Rp} 0 \sqrt{Tp} 0]
[
[0 -I*\sqrt{Rs}*e^{I*phi_b} 0 \sqrt{Ts}
[
```

`sympy.physics.optics.polarization.quarter_wave_retarder(theta)`

A quarter-wave retarder Jones matrix at angle *theta*.

### Parameters

```theta``` : numeric type or SymPy Symbol

The angle of the fast axis relative to the horizontal plane.

Returns

SymPy Matrix

A Jones matrix representing the retarder.

Examples

A generic quarter-wave plate.

```
>>> from sympy import pprint, symbols
>>> from sympy.physics.optics.polarization import quarter_wave_retarder
>>> theta= symbols("theta", real=True)
>>> QWP = quarter_wave_retarder(theta)
>>> pprint(QWP, use_unicode=True)
```

$$\begin{bmatrix} \left(i \sin^2(\theta) + \cos^2(\theta) \right) e^{\frac{-i \cdot \pi}{4}} & (1 - i) e^{\frac{-i \cdot \pi}{4}} \sin(\theta) \cos(\theta) \\ (1 - i) e^{\frac{-i \cdot \pi}{4}} \sin(\theta) \cos(\theta) & \left(\sin^2(\theta) + i \cos^2(\theta) \right) e^{\frac{-i \cdot \pi}{4}} \end{bmatrix}$$

`sympy.physics.optics.polarization.reflective_filter(R)`

A reflective filter Jones matrix with reflectance *R*.

Parameters

```R``` : numeric type or SymPy Symbol

The reflectance of the filter.

### Returns

SymPy Matrix

A Jones matrix representing the filter.

## Examples

A generic filter.

```
>>> from sympy import pprint, symbols
>>> from sympy.physics.optics.polarization import reflective_filter
>>> R = symbols("R", real=True)
>>> pprint(reflective_filter(R), use_unicode=True)
```

$$\begin{bmatrix} \sqrt{R} & 0 \\ 0 & -\sqrt{R} \end{bmatrix}$$

`sympy.physics.optics.polarization.stokes_vector(psi, chi, p=1, I=1)`

A Stokes vector corresponding to a polarization ellipse with *psi* tilt, and *chi* circularity.

### Parameters

```psi``` : numeric type or SymPy Symbol

The tilt of the polarization relative to the x axis.

```chi``` : numeric type or SymPy Symbol

The angle adjacent to the mayor axis of the polarization ellipse.

`p` : numeric type or SymPy Symbol

The degree of polarization.

`I` : numeric type or SymPy Symbol

The intensity of the field.

### Returns

Matrix :

A Stokes vector.

### Examples

The axes on the Poincaré sphere.

```
>>> from sympy import pprint, symbols, pi
>>> from sympy.physics.optics.polarization import stokes_vector
>>> psi, chi, p, I = symbols("psi, chi, p, I", real=True)
>>> pprint(stokes_vector(psi, chi, p, I), use_unicode=True)
```

$$\begin{bmatrix} I \\ I \cdot p \cdot \cos(2 \cdot \chi) \cdot \cos(2 \cdot \psi) \\ I \cdot p \cdot \sin(2 \cdot \psi) \cdot \cos(2 \cdot \chi) \\ I \cdot p \cdot \sin(2 \cdot \chi) \end{bmatrix}$$

```
Horizontal polarization >>> pprint(stokes_vector(0, 0), use_unicode=True) [1] | | | 1 |
| | 0 | | | 0]
```

```
Vertical polarization >>> pprint(stokes_vector(pi/2, 0), use_unicode=True) [1] | | | -1 |
| | 0 | | | 0]
```

```
Diagonal polarization >>> pprint(stokes_vector(pi/4, 0), use_unicode=True) [1] | | | 0 |
| | 1 | | | 0]
```

```
Anti-diagonal polarization >>> pprint(stokes_vector(-pi/4, 0), use_unicode=True) [1] | |
| | 0 | | | -1 | | 0]
```

```
Right-hand circular polarization >>> pprint(stokes_vector(0, pi/4), use_unicode=True)
[1] | | | 0 | | | 0 | | 1]
```

```
Left-hand circular polarization >>> pprint(stokes_vector(0, -pi/4), use_unicode=True)
[1] | | | 0 | | | 0 | | -1]
```

```
Unpolarized light >>> pprint(stokes_vector(0, 0, 0), use_unicode=True) [1] | | | 0 | | |
| 0 | | | 0]
```

`sympy.physics.optics.polarization.transmissive_filter(T)`

An attenuator Jones matrix with transmittance  $T$ .

### Parameters

`T` : numeric type or SymPy Symbol

The transmittance of the attenuator.

### Returns

SymPy Matrix

A Jones matrix representing the filter.

### Examples

A generic filter.

```
>>> from sympy import pprint, symbols
>>> from sympy.physics.optics.polarization import transmissive_filter
>>> T = symbols("T", real=True)
>>> NDF = transmissive_filter(T)
>>> pprint(NDF, use_unicode=True)

$$\begin{bmatrix} \sqrt{T} & 0 \\ 0 & \sqrt{T} \end{bmatrix}$$

```

### Utilities

#### Contains

- `refraction_angle`
- `fresnel_coefficients`
- `deviation`
- `brewster_angle`
- `critical_angle`
- `lens_makers_formula`
- `mirror_formula`
- `lens_formula`
- `hyperfocal_distance`
- `transverse_magnification`

`sympy.physics.optics.utils.brewster_angle(medium1, medium2)`

This function calculates the Brewster's angle of incidence to Medium 2 from Medium 1 in radians.

#### Parameters

**medium 1** : Medium or sympifiable

Refractive index of Medium 1

**medium 2** : Medium or sympifiable

Refractive index of Medium 1



## Examples

```
>>> from sympy.physics.optics import brewster_angle
>>> brewster_angle(1, 1.33)
0.926093295503462
```

`sympy.physics.optics.utils.critical_angle(medium1, medium2)`

This function calculates the critical angle of incidence (marking the onset of total internal) to Medium 2 from Medium 1 in radians.

### Parameters

**medium 1** : Medium or sympifiable

Refractive index of Medium 1.

**medium 2** : Medium or sympifiable

Refractive index of Medium 1.

## Examples

```
>>> from sympy.physics.optics import critical_angle
>>> critical_angle(1.33, 1)
0.850908514477849
```

`sympy.physics.optics.utils.deviation(incident, medium1, medium2, normal=None, plane=None)`

This function calculates the angle of deviation of a ray due to refraction at planar surface.

### Parameters

**incident** : Matrix, Ray3D, sequence or float

Incident vector or angle of incidence

**medium1** : `sympy.physics.optics.medium.Medium` or sympifiable

Medium 1 or its refractive index

**medium2** : `sympy.physics.optics.medium.Medium` or sympifiable

Medium 2 or its refractive index

**normal** : Matrix, Ray3D, or sequence

Normal vector

**plane** : Plane

Plane of separation of the two media.

**Returns angular deviation between incident and refracted rays**

## Examples

```
>>> from sympy.physics.optics import deviation
>>> from sympy.geometry import Point3D, Ray3D, Plane
>>> from sympy.matrices import Matrix
>>> from sympy import symbols
>>> n1, n2 = symbols('n1, n2')
>>> n = Matrix([0, 0, 1])
>>> P = Plane(Point3D(0, 0, 0), normal_vector=[0, 0, 1])
>>> r1 = Ray3D(Point3D(-1, -1, 1), Point3D(0, 0, 0))
>>> deviation(r1, 1, 1, n)
0
>>> deviation(r1, n1, n2, plane=P)
- $\arcsin\left(\frac{-\sqrt{-2n_1^2/(3n_2^2+1)}}{3}\right) + \arcsin\left(\frac{\sqrt{3}}{3}\right)$
>>> round(deviation(0.1, 1.2, 1.5), 5)
-0.02005
```

`sympy.physics.optics.utils.fresnel_coefficients(angle_of_incidence, medium1, medium2)`

This function uses Fresnel equations to calculate reflection and transmission coefficients. Those are obtained for both polarisations when the electric field vector is in the plane of incidence (labelled 'p') and when the electric field vector is perpendicular to the plane of incidence (labelled 's'). There are four real coefficients unless the incident ray reflects in total internal in which case there are two complex ones. Angle of incidence is the angle between the incident ray and the surface normal. *medium1* and *medium2* can be *Medium* or any sympifiable object.

### Parameters

**angle\_of\_incidence** : sympifiable

**medium1** : *Medium* or sympifiable

*Medium 1* or its refractive index

**medium2** : *Medium* or sympifiable

*Medium 2* or its refractive index

### Returns

Returns a list with four real Fresnel coefficients:

[reflection p (TM), reflection s (TE),  
transmission p (TM), transmission s (TE)]

If the ray is undergoes total internal reflection then returns a  
list of two complex Fresnel coefficients:

[reflection p (TM), reflection s (TE)]

## Examples

```
>>> from sympy.physics.optics import fresnel_coefficients
>>> fresnel_coefficients(0.3, 1, 2)
[0.317843553417859, -0.348645229818821,
 0.658921776708929, 0.651354770181179]
>>> fresnel_coefficients(0.6, 2, 1)
[-0.235625382192159 - 0.971843958291041*I,
 0.816477005968898 - 0.577377951366403*I]
```

## References

[R667]

`sympy.physics.optics.utils.hyperfocal_distance(f, N, c)`

### Parameters

**f: sympifiable**

Focal length of a given lens.

**N: sympifiable**

F-number of a given lens.

**c: sympifiable**

Circle of Confusion (CoC) of a given image format.

## Example

```
>>> from sympy.physics.optics import hyperfocal_distance
>>> round(hyperfocal_distance(f = 0.5, N = 8, c = 0.0033), 2)
9.47
```

`sympy.physics.optics.utils.lens_formula(focal_length=None, u=None, v=None)`

This function provides one of the three parameters when two of them are supplied. This is valid only for paraxial rays.

### Parameters

**focal\_length** : sympifiable

Focal length of the mirror.

**u** : sympifiable

Distance of object from the optical center on the principal axis.

**v** : sympifiable

Distance of the image from the optical center on the principal axis.

### Examples

```
>>> from sympy.physics.optics import lens_formula
>>> from sympy.abc import f, u, v
>>> lens_formula(focal_length=f, u=u)
f*u/(f + u)
>>> lens_formula(focal_length=f, v=v)
f*v/(f - v)
>>> lens_formula(u=u, v=v)
u*v/(u - v)
```

`sympy.physics.optics.utils.lens_makers_formula(n_lens, n_surr, r1, r2, d=0)`

This function calculates focal length of a lens. It follows cartesian sign convention.

#### Parameters

- n\_lens*** : Medium or sympifiable  
Index of refraction of lens.
- n\_surr*** : Medium or sympifiable  
Index of reflection of surrounding.
- r1*** : sympifiable  
Radius of curvature of first surface.
- r2*** : sympifiable  
Radius of curvature of second surface.
- d*** : sympifiable, optional  
Thickness of lens, default value is 0.

### Examples

```
>>> from sympy.physics.optics import lens_makers_formula
>>> from sympy import S
>>> lens_makers_formula(1.33, 1, 10, -10)
15.1515151515151
>>> lens_makers_formula(1.2, 1, 10, S.Infinity)
50.0000000000000
>>> lens_makers_formula(1.33, 1, 10, -10, d=1)
15.3418463277618
```

`sympy.physics.optics.utils.mirror_formula(focal_length=None, u=None, v=None)`

This function provides one of the three parameters when two of them are supplied. This is valid only for paraxial rays.

#### Parameters

- focal\_length*** : sympifiable  
Focal length of the mirror.
- u*** : sympifiable  
Distance of object from the pole on the principal axis.

**v** : sympifiable

Distance of the image from the pole on the principal axis.

## Examples

```
>>> from sympy.physics.optics import mirror_formula
>>> from sympy.abc import f, u, v
>>> mirror_formula(focal_length=f, u=u)
f*u/(-f + u)
>>> mirror_formula(focal_length=f, v=v)
f*v/(-f + v)
>>> mirror_formula(u=u, v=v)
u*v/(u + v)
```

`sympy.physics.optics.utils.refraction_angle(incident, medium1, medium2, normal=None, plane=None)`

This function calculates transmitted vector after refraction at planar surface. `medium1` and `medium2` can be `Medium` or any sympifiable object. If `incident` is a number then treated as angle of incidence (in radians) in which case refraction angle is returned.

If `incident` is an object of `Ray3D`, `normal` also has to be an instance of `Ray3D` in order to get the output as a `Ray3D`. Please note that if plane of separation is not provided and `normal` is an instance of `Ray3D`, `normal` will be assumed to be intersecting incident ray at the plane of separation. This will not be the case when `normal` is a `Matrix` or any other sequence. If `incident` is an instance of `Ray3D` and `plane` has not been provided and `normal` is not `Ray3D`, output will be a `Matrix`.

### Parameters

**incident** : Matrix, Ray3D, sequence or a number

Incident vector or angle of incidence

**medium1** : `sympy.physics.optics.medium.Medium` or sympifiable

Medium 1 or its refractive index

**medium2** : `sympy.physics.optics.medium.Medium` or sympifiable

Medium 2 or its refractive index

**normal** : Matrix, Ray3D, or sequence

Normal vector

**plane** : Plane

Plane of separation of the two media.

### Returns

Returns an angle of refraction or a refracted ray depending on inputs.

## Examples

```
>>> from sympy.physics.optics import refraction_angle
>>> from sympy.geometry import Point3D, Ray3D, Plane
>>> from sympy.matrices import Matrix
>>> from sympy import symbols, pi
>>> n = Matrix([0, 0, 1])
>>> P = Plane(Point3D(0, 0, 0), normal_vector=[0, 0, 1])
>>> r1 = Ray3D(Point3D(-1, -1, 1), Point3D(0, 0, 0))
>>> refraction_angle(r1, 1, 1, n)
Matrix([
[1],
[1],
[-1]])
>>> refraction_angle(r1, 1, 1, plane=P)
Ray3D(Point3D(0, 0, 0), Point3D(1, 1, -1))
```

With different index of refraction of the two media

```
>>> n1, n2 = symbols('n1, n2')
>>> refraction_angle(r1, n1, n2, n)
Matrix([
[n1/n2],
[n1/n2],
[-sqrt(3)*sqrt(-2*n1**2/(3*n2**2) + 1)]]])
>>> refraction_angle(r1, n1, n2, plane=P)
Ray3D(Point3D(0, 0, 0), Point3D(n1/n2, n1/n2, -sqrt(3)*sqrt(-2*n1**2/
↪ (3*n2**2) + 1)))
>>> round(refraction_angle(pi/6, 1.2, 1.5), 5)
0.41152
```

`sympy.physics.optics.utils.transverse_magnification(si, so)`

Calculates the transverse magnification, which is the ratio of the image size to the object size.

### Parameters

**so: sympifiable**

Lens-object distance.

**si: sympifiable**

Lens-image distance.

## Example

```
>>> from sympy.physics.optics import transverse_magnification
>>> transverse_magnification(30, 15)
-2
```

## Waves

This module has all the classes and functions related to waves in optics.

### Contains

- TWave

**class** sympy.physics.optics.waves.**TWave**(*amplitude*, *frequency=None*, *phase=0*, *time\_period=None*, *n=n*)

This is a simple transverse sine wave travelling in a one-dimensional space. Basic properties are required at the time of creation of the object, but they can be changed later with respective methods provided.

### Raises

**ValueError** : When neither frequency nor time period is provided or they are not consistent.

**TypeError** : When anything other than TWave objects is added.

### Explanation

It is represented as  $A \times \cos(k * x - \omega \times t + \phi)$ , where  $A$  is the amplitude,  $\omega$  is the angular frequency,  $k$  is the wavenumber (spatial frequency),  $x$  is a spatial variable to represent the position on the dimension on which the wave propagates, and  $\phi$  is the phase angle of the wave.

### Arguments

#### amplitude

[Symplifiable] Amplitude of the wave.

#### frequency

[Symplifiable] Frequency of the wave.

#### phase

[Symplifiable] Phase angle of the wave.

#### time\_period

[Symplifiable] Time period of the wave.

#### n

[Symplifiable] Refractive index of the medium.

### Examples

```
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A1, phi1, A2, phi2, f = symbols('A1, phi1, A2, phi2, f')
>>> w1 = TWave(A1, f, phi1)
>>> w2 = TWave(A2, f, phi2)
>>> w3 = w1 + w2 # Superposition of two waves
>>> w3
```

(continues on next page)

(continued from previous page)

```
Twave(sqrt(A1**2 + 2*A1*A2*cos(phi1 - phi2) + A2**2), f,
 atan2(A1*sin(phi1) + A2*sin(phi2), A1*cos(phi1) + A2*cos(phi2)), 1/f,
 ↪ n)
>>> w3.amplitude
sqrt(A1**2 + 2*A1*A2*cos(phi1 - phi2) + A2**2)
>>> w3.phase
atan2(A1*sin(phi1) + A2*sin(phi2), A1*cos(phi1) + A2*cos(phi2))
>>> w3.speed
299792458*meter/(second*n)
>>> w3.angular_velocity
2*pi*f
```

**property amplitude**

Returns the amplitude of the wave.

**Examples**

```
>>> from sympy import symbols
>>> from sympy.physics.optics import Twave
>>> A, phi, f = symbols('A, phi, f')
>>> w = Twave(A, f, phi)
>>> w.amplitude
A
```

**property angular\_velocity**

Returns the angular velocity of the wave, in radians per second.

**Examples**

```
>>> from sympy import symbols
>>> from sympy.physics.optics import Twave
>>> A, phi, f = symbols('A, phi, f')
>>> w = Twave(A, f, phi)
>>> w.angular_velocity
2*pi*f
```

**property frequency**

Returns the frequency of the wave, in cycles per second.



### Examples

```
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A, phi, f = symbols('A, phi, f')
>>> w = TWave(A, f, phi)
>>> w.frequency
f
```

#### property `n`

Returns the refractive index of the medium

#### property `phase`

Returns the phase angle of the wave, in radians.

### Examples

```
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A, phi, f = symbols('A, phi, f')
>>> w = TWave(A, f, phi)
>>> w.phase
phi
```

#### property `speed`

Returns the propagation speed of the wave, in meters per second. It is dependent on the propagation medium.

### Examples

```
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A, phi, f = symbols('A, phi, f')
>>> w = TWave(A, f, phi)
>>> w.speed
299792458*meter/(second*n)
```

#### property `time_period`

Returns the temporal period of the wave, in seconds per cycle.

### Examples

```
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A, phi, f = symbols('A, phi, f')
>>> w = TWave(A, f, phi)
>>> w.time_period
1/f
```

**property wavelength**

Returns the wavelength (spatial period) of the wave, in meters per cycle. It depends on the medium of the wave.

**Examples**

```
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A, phi, f = symbols('A, phi, f')
>>> w = TWave(A, f, phi)
>>> w.wavelength
299792458*meter/(second*f*n)
```

**property wavenumber**

Returns the wavenumber of the wave, in radians per meter.

**Examples**

```
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A, phi, f = symbols('A, phi, f')
>>> w = TWave(A, f, phi)
>>> w.wavenumber
pi*second*f*n/(149896229*meter)
```

**Control Module****Abstract**

Contains docstrings of Physics-Control module

**Control**

Currently, *sympy.physics.control* (page 1890) is able to deal with LTI (Linear, time-invariant) systems. The *TransferFunction* class is used to represent Continuous-time Transfer functions in the Laplace domain; where Transfer functions are input to output representations of dynamic systems. The additive property is used for transfer functions in the *Parallel* class, and the multiplicative property is used for transfer functions in the *Series* class. Also, there is a *Feedback* class which is used to represent negative feedback interconnection between two input/output systems. MIMO systems are also supported with *TransferFunctionMatrix* as the base class for representing one. *MIMOSeries*, *MIMOParallel* and *MIMOFeedback* are MIMO equivalent of *Series*, *Parallel* and *Feedback* classes.

The advantage of this symbolic Control system package is that the solutions obtained from it are highly accurate and do not rely on numerical methods to approximate the solutions. Symbolic solutions obtained are also in a compact form that can be used for further analysis.

## Control API

### lti

**class** `sympy.physics.control.lti.TransferFunction(num, den, var)`

A class for representing LTI (Linear, time-invariant) systems that can be strictly described by ratio of polynomials in the Laplace transform complex variable. The arguments are `num`, `den`, and `var`, where `num` and `den` are numerator and denominator polynomials of the `TransferFunction` respectively, and the third argument is a complex variable of the Laplace transform used by these polynomials of the transfer function. `num` and `den` can be either polynomials or numbers, whereas `var` has to be a [Symbol](#) (page 976).

#### Parameters

**num** : Expr, Number

The numerator polynomial of the transfer function.

**den** : Expr, Number

The denominator polynomial of the transfer function.

**var** : Symbol

Complex variable of the Laplace transform used by the polynomials of the transfer function.

#### Raises

##### **TypeError**

When `var` is not a `Symbol` or when `num` or `den` is not a number or a polynomial.

##### **ValueError**

When `den` is zero.

## Explanation

Generally, a dynamical system representing a physical model can be described in terms of Linear Ordinary Differential Equations like -

$$b_m y^{(m)} + b_{m-1} y^{(m-1)} + \dots + b_1 y^{(1)} + b_0 y = a_n x^{(n)} + a_{n-1} x^{(n-1)} + \dots + a_1 x^{(1)} + a_0 x$$

Here,  $x$  is the input signal and  $y$  is the output signal and superscript on both is the order of derivative (not exponent). Derivative is taken with respect to the independent variable,  $t$ . Also, generally  $m$  is greater than  $n$ .

It is not feasible to analyse the properties of such systems in their native form therefore, we use mathematical tools like Laplace transform to get a better perspective. Taking the Laplace transform of both the sides in the equation (at zero initial conditions), we get -

$$\mathcal{L}[b_m y^{(m)} + b_{m-1} y^{(m-1)} + \dots + b_1 y^{(1)} + b_0 y] = \mathcal{L}[a_n x^{(n)} + a_{n-1} x^{(n-1)} + \dots + a_1 x^{(1)} + a_0 x]$$

Using the linearity property of Laplace transform and also considering zero initial conditions (i.e.  $y(0^-) = 0$ ,  $y'(0^-) = 0$  and so on), the equation above gets translated to -

$$b_m \mathcal{L}[y^{(m)}] + \dots + b_1 \mathcal{L}[y^{(1)}] + b_0 \mathcal{L}[y] = a_n \mathcal{L}[x^{(n)}] + \dots + a_1 \mathcal{L}[x^{(1)}] + a_0 \mathcal{L}[x]$$

Now, applying Derivative property of Laplace transform,

$$b_m s^m \mathcal{L}[y] + \cdots + b_1 s \mathcal{L}[y] + b_0 \mathcal{L}[y] = a_n s^n \mathcal{L}[x] + \cdots + a_1 s \mathcal{L}[x] + a_0 \mathcal{L}[x]$$

Here, the superscript on  $s$  is **exponent**. Note that the zero initial conditions assumption, mentioned above, is very important and cannot be ignored otherwise the dynamical system cannot be considered time-independent and the simplified equation above cannot be reached.

Collecting  $\mathcal{L}[y]$  and  $\mathcal{L}[x]$  terms from both the sides and taking the ratio  $\frac{\mathcal{L}\{y\}}{\mathcal{L}\{x\}}$ , we get the typical rational form of transfer function.

The numerator of the transfer function is, therefore, the Laplace transform of the output signal (The signals are represented as functions of time) and similarly, the denominator of the transfer function is the Laplace transform of the input signal. It is also a convention to denote the input and output signal's Laplace transform with capital alphabets like shown below.

$$H(s) = \frac{Y(s)}{X(s)} = \frac{\mathcal{L}\{y(t)\}}{\mathcal{L}\{x(t)\}}$$

$s$ , also known as complex frequency, is a complex variable in the Laplace domain. It corresponds to the equivalent variable  $t$ , in the time domain. Transfer functions are sometimes also referred to as the Laplace transform of the system's impulse response. Transfer function,  $H$ , is represented as a rational function in  $s$  like,

$$H(s) = \frac{a_n s^n + a_{n-1} s^{n-1} + \cdots + a_1 s + a_0}{b_m s^m + b_{m-1} s^{m-1} + \cdots + b_1 s + b_0}$$

## Examples

```
>>> from sympy.abc import s, p, a
>>> from sympy.physics.control.lti import TransferFunction
>>> tf1 = TransferFunction(s + a, s**2 + s + 1, s)
>>> tf1
TransferFunction(a + s, s**2 + s + 1, s)
>>> tf1.num
a + s
>>> tf1.den
s**2 + s + 1
>>> tf1.var
s
>>> tf1.args
(a + s, s**2 + s + 1, s)
```

Any complex variable can be used for var.

```
>>> tf2 = TransferFunction(a*p**3 - a*p**2 + s*p, p + a**2, p)
>>> tf2
TransferFunction(a*p**3 - a*p**2 + p*s, a**2 + p, p)
>>> tf3 = TransferFunction((p + 3)*(p - 1), (p - 1)*(p + 5), p)
>>> tf3
TransferFunction((p - 1)*(p + 3), (p - 1)*(p + 5), p)
```

To negate a transfer function the `-` operator can be prepended:

```
>>> tf4 = TransferFunction(-a + s, p**2 + s, p)
>>> -tf4
```

(continues on next page)

(continued from previous page)

```

TransferFunction(a - s, p**2 + s, p)
>>> tf5 = TransferFunction(s**4 - 2*s**3 + 5*s + 4, s + 4, s)
>>> -tf5
TransferFunction(-s**4 + 2*s**3 - 5*s - 4, s + 4, s)

```

You can use a float or an integer (or other constants) as numerator and denominator:

```

>>> tf6 = TransferFunction(1/2, 4, s)
>>> tf6.num
0.5000000000000000
>>> tf6.den
4
>>> tf6.var
s
>>> tf6.args
(0.5, 4, s)

```

You can take the integer power of a transfer function using the `**` operator:

```

>>> tf7 = TransferFunction(s + a, s - a, s)
>>> tf7**3
TransferFunction((a + s)**3, (-a + s)**3, s)
>>> tf7**0
TransferFunction(1, 1, s)
>>> tf8 = TransferFunction(p + 4, p - 3, p)
>>> tf8**-1
TransferFunction(p - 3, p + 4, p)

```

Addition, subtraction, and multiplication of transfer functions can form unevaluated Series or Parallel objects.

```

>>> tf9 = TransferFunction(s + 1, s**2 + s + 1, s)
>>> tf10 = TransferFunction(s - p, s + 3, s)
>>> tf11 = TransferFunction(4*s**2 + 2*s - 4, s - 1, s)
>>> tf12 = TransferFunction(1 - s, s**2 + 4, s)
>>> tf9 + tf10
Parallel(TransferFunction(s + 1, s**2 + s + 1, s), TransferFunction(-p +
→ s, s + 3, s))
>>> tf10 - tf11
Parallel(TransferFunction(-p + s, s + 3, s), TransferFunction(-4*s**2 -
→ 2*s + 4, s - 1, s))
>>> tf9 * tf10
Series(TransferFunction(s + 1, s**2 + s + 1, s), TransferFunction(-p + s,
→ s + 3, s))
>>> tf10 - (tf9 + tf12)
Parallel(TransferFunction(-p + s, s + 3, s), TransferFunction(-s - 1,
→ s**2 + s + 1, s), TransferFunction(s - 1, s**2 + 4, s))
>>> tf10 - (tf9 * tf12)
Parallel(TransferFunction(-p + s, s + 3, s), Series(TransferFunction(-1,
→ 1, s), TransferFunction(s + 1, s**2 + s + 1, s), TransferFunction(1 -
→ s, s**2 + 4, s)))
>>> tf11 * tf10 * tf9

```

(continues on next page)

(continued from previous page)

```

Series(TransferFunction(4*s**2 + 2*s - 4, s - 1, s), TransferFunction(-p
→ + s, s + 3, s), TransferFunction(s + 1, s**2 + s + 1, s))
>>> tf9 * tf11 + tf10 * tf12
Parallel(Series(TransferFunction(s + 1, s**2 + s + 1, s),
→ TransferFunction(4*s**2 + 2*s - 4, s - 1, s)),
→ Series(TransferFunction(-p + s, s + 3, s), TransferFunction(1 - s,
→ s**2 + 4, s)))
>>> (tf9 + tf12) * (tf10 + tf11)
Series(Parallel(TransferFunction(s + 1, s**2 + s + 1, s),
→ TransferFunction(1 - s, s**2 + 4, s)), Parallel(TransferFunction(-p +
→ s, s + 3, s), TransferFunction(4*s**2 + 2*s - 4, s - 1, s)))

```

These unevaluated Series or Parallel objects can convert into the resultant transfer function using `.doit()` method or by `.rewrite(TransferFunction)`.

```

>>> ((tf9 + tf10) * tf12).doit()
TransferFunction((1 - s)*((-p + s)*(s**2 + s + 1) + (s + 1)*(s + 3)), (s
→ + 3)*(s**2 + 4)*(s**2 + s + 1), s)
>>> (tf9 * tf10 - tf11 * tf12).rewrite(TransferFunction)
TransferFunction(-(1 - s)*(s + 3)*(s**2 + s + 1)*(4*s**2 + 2*s - 4) + (-
→ p + s)*(s - 1)*(s + 1)*(s**2 + 4), (s - 1)*(s + 3)*(s**2 + 4)*(s**2 +
→ s + 1), s)

```

**See also:**

[Feedback](#) (page 1906), [Series](#) (page 1899), [Parallel](#) (page 1903)

**References**

[R656], [R657]

**dc\_gain()**

Computes the gain of the response as the frequency approaches zero.

The DC gain is infinite for systems with pure integrators.

**Examples**

```

>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction
>>> tf1 = TransferFunction(s + 3, s**2 - 9, s)
>>> tf1.dc_gain()
-1/3
>>> tf2 = TransferFunction(p**2, p - 3 + p**3, p)
>>> tf2.dc_gain()
0
>>> tf3 = TransferFunction(a*p**2 - b, s + b, s)
>>> tf3.dc_gain()
(a*p**2 - b)/b
>>> tf4 = TransferFunction(1, s, s)

```

(continues on next page)

(continued from previous page)

```
>>> tf4.dc_gain()
00
```

**property den**

Returns the denominator polynomial of the transfer function.

**Examples**

```
>>> from sympy.abc import s, p
>>> from sympy.physics.control.lti import TransferFunction
>>> G1 = TransferFunction(s + 4, p**3 - 2*p + 4, s)
>>> G1.den
p**3 - 2*p + 4
>>> G2 = TransferFunction(3, 4, s)
>>> G2.den
4
```

**expand()**

Returns the transfer function with numerator and denominator in expanded form.

**Examples**

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction
>>> G1 = TransferFunction((a - s)**2, (s**2 + a)**2, s)
>>> G1.expand()
TransferFunction(a**2 - 2*a*s + s**2, a**2 + 2*a*s**2 + s**4, s)
>>> G2 = TransferFunction((p + 3*b)*(p - b), (p - b)*(p + 2*b), p)
>>> G2.expand()
TransferFunction(-3*b**2 + 2*b*p + p**2, -2*b**2 + b*p + p**2, p)
```

**classmethod from\_rational\_expression(expr, var=None)**

Creates a new TransferFunction efficiently from a rational expression.

**Parameters**

**expr** : Expr, Number

The rational expression representing the TransferFunction.

**var** : Symbol, optional

Complex variable of the Laplace transform used by the polynomials of the transfer function.

**Raises****ValueError**

When expr is of type Number and optional parameter var is not passed.

When expr has more than one variables and an optional parameter var is not passed.

### ZeroDivisionError

When denominator of `expr` is zero or it has `ComplexInfinity` in its numerator.

### Examples

```
>>> from sympy.abc import s, p, a
>>> from sympy.physics.control.lti import TransferFunction
>>> expr1 = (s + 5)/(3*s**2 + 2*s + 1)
>>> tf1 = TransferFunction.from_rational_expression(expr1)
>>> tf1
TransferFunction(s + 5, 3*s**2 + 2*s + 1, s)
>>> expr2 = (a*p**3 - a*p**2 + s*p)/(p + a**2) # Expr with more than
↳one variables
>>> tf2 = TransferFunction.from_rational_expression(expr2, p)
>>> tf2
TransferFunction(a*p**3 - a*p**2 + p*s, a**2 + p, p)
```

In case of conflict between two or more variables in a expression, SymPy will raise a `ValueError`, if `var` is not passed by the user.

```
>>> tf = TransferFunction.from_rational_expression((a + a*s)/(s**2 +
↳s + 1))
Traceback (most recent call last):
...
ValueError: Conflicting values found for positional argument `var` (
↳{a, s}). Specify it manually.
```

This can be corrected by specifying the `var` parameter manually.

```
>>> tf = TransferFunction.from_rational_expression((a + a*s)/(s**2 +
↳s + 1), s)
>>> tf
TransferFunction(a*s + a, s**2 + s + 1, s)
```

`var` also need to be specified when `expr` is a `Number`

```
>>> tf3 = TransferFunction.from_rational_expression(10, s)
>>> tf3
TransferFunction(10, 1, s)
```

### `property is_biproper`

Returns `True` if degree of the numerator polynomial is equal to degree of the denominator polynomial, else `False`.



## Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf1.is_biproper
True
>>> tf2 = TransferFunction(p**2, p + a, p)
>>> tf2.is_biproper
False
```

### property is\_proper

Returns True if degree of the numerator polynomial is less than or equal to degree of the denominator polynomial, else False.

## Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction
>>> tf1 = TransferFunction(b*s**2 + p**2 - a*p + s, b - p**2, s)
>>> tf1.is_proper
False
>>> tf2 = TransferFunction(p**2 - 4*p, p**3 + 3*p + 2, p)
>>> tf2.is_proper
True
```

### is\_stable()

Returns True if the transfer function is asymptotically stable; else False.  
This would not check the marginal or conditional stability of the system.

## Examples

```
>>> from sympy.abc import s, p, a
>>> from sympy import symbols
>>> from sympy.physics.control.lti import TransferFunction
>>> q, r = symbols('q, r', negative=True)
>>> tf1 = TransferFunction((1 - s)**2, (s + 1)**2, s)
>>> tf1.is_stable()
True
>>> tf2 = TransferFunction((1 - p)**2, (s**2 + 1)**2, s)
>>> tf2.is_stable()
False
>>> tf3 = TransferFunction(4, q*s - r, s)
>>> tf3.is_stable()
False
>>> tf4 = TransferFunction(p + 1, a*p - s**2, p)
>>> tf4.is_stable() is None # Not enough info about the symbols to
 ↪ determine stability
True
```

**property is\_strictly\_proper**

Returns True if degree of the numerator polynomial is strictly less than degree of the denominator polynomial, else False.

**Examples**

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf1.is_strictly_proper
False
>>> tf2 = TransferFunction(s**3 - 2, s**4 + 5*s + 6, s)
>>> tf2.is_strictly_proper
True
```

**property num**

Returns the numerator polynomial of the transfer function.

**Examples**

```
>>> from sympy.abc import s, p
>>> from sympy.physics.control.lti import TransferFunction
>>> G1 = TransferFunction(s**2 + p*s + 3, s - 4, s)
>>> G1.num
p*s + s**2 + 3
>>> G2 = TransferFunction((p + 5)*(p - 3), (p - 3)*(p + 1), p)
>>> G2.num
(p - 3)*(p + 5)
```

**poles()**

Returns the poles of a transfer function.

**Examples**

```
>>> from sympy.abc import s, p, a
>>> from sympy.physics.control.lti import TransferFunction
>>> tf1 = TransferFunction((p + 3)*(p - 1), (p - 1)*(p + 5), p)
>>> tf1.poles()
[-5, 1]
>>> tf2 = TransferFunction((1 - s)**2, (s**2 + 1)**2, s)
>>> tf2.poles()
[I, I, -I, -I]
>>> tf3 = TransferFunction(s**2, a*s + p, s)
>>> tf3.poles()
[-p/a]
```

**to\_expr()**

Converts a TransferFunction object to SymPy Expr.

## Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction
>>> from sympy import Expr
>>> tf1 = TransferFunction(s, a*s**2 + 1, s)
>>> tf1.to_expr()
s/(a*s**2 + 1)
>>> isinstance(_, Expr)
True
>>> tf2 = TransferFunction(1, (p + 3*b)*(b - p), p)
>>> tf2.to_expr()
1/((b - p)*(3*b + p))
>>> tf3 = TransferFunction((s - 2)*(s - 3), (s - 1)*(s - 2)*(s - 3),
↪ s)
>>> tf3.to_expr()
((s - 3)*(s - 2))/((s - 3)*(s - 2)*(s - 1))
```

### property var

Returns the complex variable of the Laplace transform used by the polynomials of the transfer function.

## Examples

```
>>> from sympy.abc import s, p
>>> from sympy.physics.control.lti import TransferFunction
>>> G1 = TransferFunction(p**2 + 2*p + 4, p - 6, p)
>>> G1.var
p
>>> G2 = TransferFunction(0, s - 5, s)
>>> G2.var
s
```

### zeros()

Returns the zeros of a transfer function.

## Examples

```
>>> from sympy.abc import s, p, a
>>> from sympy.physics.control.lti import TransferFunction
>>> tf1 = TransferFunction((p + 3)*(p - 1), (p - 1)*(p + 5), p)
>>> tf1.zeros()
[-3, 1]
>>> tf2 = TransferFunction((1 - s)**2, (s**2 + 1)**2, s)
>>> tf2.zeros()
[1, 1]
>>> tf3 = TransferFunction(s**2, a*s + p, s)
>>> tf3.zeros()
[0, 0]
```

**class** `sympy.physics.control.lti.Series(*args, evaluate=False)`

A class for representing a series configuration of SISO systems.

**Parameters**

**args** : SISOLinearTimeInvariant

SISO systems in a series configuration.

**evaluate** : Boolean, Keyword

When passed True, returns the equivalent `Series(*args).doit()`.  
Set to False by default.

**Raises**

**ValueError**

When no argument is passed.

var attribute is not same for every system.

**TypeError**

Any of the passed `*args` has unsupported type

A combination of SISO and MIMO systems is passed. There should be homogeneity in the type of systems passed, SISO in this case.

## Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction, Series, Parallel
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf2 = TransferFunction(s**3 - 2, s**4 + 5*s + 6, s)
>>> tf3 = TransferFunction(p**2, p + s, s)
>>> S1 = Series(tf1, tf2)
>>> S1
Series(TransferFunction(a*p**2 + b*s, -p + s, s), TransferFunction(s**3 -
 2, s**4 + 5*s + 6, s))
>>> S1.var
s
>>> S2 = Series(tf2, Parallel(tf3, -tf1))
>>> S2
Series(TransferFunction(s**3 - 2, s**4 + 5*s + 6, s),
 Parallel(TransferFunction(p**2, p + s, s), TransferFunction(-a*p**2 -
 b*s, -p + s, s)))
>>> S2.var
s
>>> S3 = Series(Parallel(tf1, tf2), Parallel(tf2, tf3))
>>> S3
Series(Parallel(TransferFunction(a*p**2 + b*s, -p + s, s),
 TransferFunction(s**3 - 2, s**4 + 5*s + 6, s)),
 Parallel(TransferFunction(s**3 - 2, s**4 + 5*s + 6, s),
 TransferFunction(p**2, p + s, s)))
>>> S3.var
s
```

You can get the resultant transfer function by using `.doit()` method:

```
>>> S3 = Series(tf1, tf2, -tf3)
>>> S3.doit()
TransferFunction(-p**2*(s**3 - 2)*(a*p**2 + b*s), (-p + s)*(p + s)*(s**4
↪ + 5*s + 6), s)
>>> S4 = Series(tf2, Parallel(tf1, -tf3))
>>> S4.doit()
TransferFunction((s**3 - 2)*(-p**2*(-p + s) + (p + s)*(a*p**2 + b*s)), (-
↪ p + s)*(p + s)*(s**4 + 5*s + 6), s)
```

## Notes

All the transfer functions should use the same complex variable `var` of the Laplace transform.

### See also:

*MIMOSeries* (page 1922), *Parallel* (page 1903), *TransferFunction* (page 1891), *Feedback* (page 1906)

### `doit(**hints)`

Returns the resultant transfer function obtained after evaluating the transfer functions in series configuration.

## Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction, Series
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf2 = TransferFunction(s**3 - 2, s**4 + 5*s + 6, s)
>>> Series(tf2, tf1).doit()
TransferFunction((s**3 - 2)*(a*p**2 + b*s), (-p + s)*(s**4 + 5*s + 6),
↪ s)
>>> Series(-tf1, -tf2).doit()
TransferFunction((2 - s**3)*(-a*p**2 - b*s), (-p + s)*(s**4 + 5*s +
↪ 6), s)
```

### `property is_biproper`

Returns True if degree of the numerator polynomial of the resultant transfer function is equal to degree of the denominator polynomial of the same, else False.

## Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction, Series
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf2 = TransferFunction(p, s**2, s)
>>> tf3 = TransferFunction(s**2, 1, s)
>>> S1 = Series(tf1, -tf2)
>>> S1.is_biproper
False
>>> S2 = Series(tf2, tf3)
>>> S2.is_biproper
True
```

### property is\_proper

Returns True if degree of the numerator polynomial of the resultant transfer function is less than or equal to degree of the denominator polynomial of the same, else False.

## Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction, Series
>>> tf1 = TransferFunction(b*s**2 + p**2 - a*p + s, b - p**2, s)
>>> tf2 = TransferFunction(p**2 - 4*p, p**3 + 3*s + 2, s)
>>> tf3 = TransferFunction(s, s**2 + s + 1, s)
>>> S1 = Series(-tf2, tf1)
>>> S1.is_proper
False
>>> S2 = Series(tf1, tf2, tf3)
>>> S2.is_proper
True
```

### property is\_strictly\_proper

Returns True if degree of the numerator polynomial of the resultant transfer function is strictly less than degree of the denominator polynomial of the same, else False.

## Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction, Series
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf2 = TransferFunction(s**3 - 2, s**2 + 5*s + 6, s)
>>> tf3 = TransferFunction(1, s**2 + s + 1, s)
>>> S1 = Series(tf1, tf2)
>>> S1.is_strictly_proper
False
>>> S2 = Series(tf1, tf2, tf3)
>>> S2.is_strictly_proper
True
```

**to\_expr()**

Returns the equivalent Expr object.

**property var**

Returns the complex variable used by all the transfer functions.

**Examples**

```
>>> from sympy.abc import p
>>> from sympy.physics.control.lti import TransferFunction, Series, Parallel
>>> G1 = TransferFunction(p**2 + 2*p + 4, p - 6, p)
>>> G2 = TransferFunction(p, 4 - p, p)
>>> G3 = TransferFunction(0, p**4 - 1, p)
>>> Series(G1, G2).var
p
>>> Series(-G3, Parallel(G1, G2)).var
p
```

**class** sympy.physics.control.lti.**Parallel**(\*args, evaluate=False)

A class for representing a parallel configuration of SISO systems.

**Parameters**

**args** : SISOLinearTimeInvariant

SISO systems in a parallel arrangement.

**evaluate** : Boolean, Keyword

When passed True, returns the equivalent `Parallel(*args).doit()`.  
Set to False by default.

**Raises****ValueError**

When no argument is passed.

var attribute is not same for every system.

**TypeError**

Any of the passed \*args has unsupported type

A combination of SISO and MIMO systems is passed. There should be homogeneity in the type of systems passed.

**Examples**

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction, Parallel, Series
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf2 = TransferFunction(s**3 - 2, s**4 + 5*s + 6, s)
>>> tf3 = TransferFunction(p**2, p + s, s)
>>> P1 = Parallel(tf1, tf2)
```

(continues on next page)

(continued from previous page)

```

>>> P1
Parallel(TransferFunction(a*p**2 + b*s, -p + s, s),
↳ TransferFunction(s**3 - 2, s**4 + 5*s + 6, s))
>>> P1.var
s
>>> P2 = Parallel(tf2, Series(tf3, -tf1))
>>> P2
Parallel(TransferFunction(s**3 - 2, s**4 + 5*s + 6, s),
↳ Series(TransferFunction(p**2, p + s, s), TransferFunction(-a*p**2 -
↳ b*s, -p + s, s)))
>>> P2.var
s
>>> P3 = Parallel(Series(tf1, tf2), Series(tf2, tf3))
>>> P3
Parallel(Series(TransferFunction(a*p**2 + b*s, -p + s, s),
↳ TransferFunction(s**3 - 2, s**4 + 5*s + 6, s)),
↳ Series(TransferFunction(s**3 - 2, s**4 + 5*s + 6, s),
↳ TransferFunction(p**2, p + s, s)))
>>> P3.var
s

```

You can get the resultant transfer function by using `.doit()` method:

```

>>> Parallel(tf1, tf2, -tf3).doit()
TransferFunction(-p**2*(-p + s)*(s**4 + 5*s + 6) + (-p + s)*(p +
↳ s)*(s**3 - 2) + (p + s)*(a*p**2 + b*s)*(s**4 + 5*s + 6), (-p + s)*(p +
↳ s)*(s**4 + 5*s + 6), s)
>>> Parallel(tf2, Series(tf1, -tf3)).doit()
TransferFunction(-p**2*(a*p**2 + b*s)*(s**4 + 5*s + 6) + (-p + s)*(p +
↳ s)*(s**3 - 2), (-p + s)*(p + s)*(s**4 + 5*s + 6), s)

```

## Notes

All the transfer functions should use the same complex variable `var` of the Laplace transform.

### See also:

[Series](#) (page 1899), [TransferFunction](#) (page 1891), [Feedback](#) (page 1906)

### `doit(**hints)`

Returns the resultant transfer function obtained after evaluating the transfer functions in parallel configuration.



## Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction, Parallel
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf2 = TransferFunction(s**3 - 2, s**4 + 5*s + 6, s)
>>> Parallel(tf2, tf1).doit()
TransferFunction((-p + s)*(s**3 - 2) + (a*p**2 + b*s)*(s**4 + 5*s +
↳ 6), (-p + s)*(s**4 + 5*s + 6), s)
>>> Parallel(-tf1, -tf2).doit()
TransferFunction((2 - s**3)*(-p + s) + (-a*p**2 - b*s)*(s**4 + 5*s +
↳ 6), (-p + s)*(s**4 + 5*s + 6), s)
```

### property is\_biproper

Returns True if degree of the numerator polynomial of the resultant transfer function is equal to degree of the denominator polynomial of the same, else False.

## Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction, Parallel
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf2 = TransferFunction(p**2, p + s, s)
>>> tf3 = TransferFunction(s, s**2 + s + 1, s)
>>> P1 = Parallel(tf1, -tf2)
>>> P1.is_biproper
True
>>> P2 = Parallel(tf2, tf3)
>>> P2.is_biproper
False
```

### property is\_proper

Returns True if degree of the numerator polynomial of the resultant transfer function is less than or equal to degree of the denominator polynomial of the same, else False.

## Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction, Parallel
>>> tf1 = TransferFunction(b*s**2 + p**2 - a*p + s, b - p**2, s)
>>> tf2 = TransferFunction(p**2 - 4*p, p**3 + 3*s + 2, s)
>>> tf3 = TransferFunction(s, s**2 + s + 1, s)
>>> P1 = Parallel(-tf2, tf1)
>>> P1.is_proper
False
>>> P2 = Parallel(tf2, tf3)
>>> P2.is_proper
True
```

**property is\_strictly\_proper**

Returns True if degree of the numerator polynomial of the resultant transfer function is strictly less than degree of the denominator polynomial of the same, else False.

**Examples**

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction, Parallel
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf2 = TransferFunction(s**3 - 2, s**4 + 5*s + 6, s)
>>> tf3 = TransferFunction(s, s**2 + s + 1, s)
>>> P1 = Parallel(tf1, tf2)
>>> P1.is_strictly_proper
False
>>> P2 = Parallel(tf2, tf3)
>>> P2.is_strictly_proper
True
```

**to\_expr()**

Returns the equivalent Expr object.

**property var**

Returns the complex variable used by all the transfer functions.

**Examples**

```
>>> from sympy.abc import p
>>> from sympy.physics.control.lti import TransferFunction, Parallel, Series
>>> G1 = TransferFunction(p**2 + 2*p + 4, p - 6, p)
>>> G2 = TransferFunction(p, 4 - p, p)
>>> G3 = TransferFunction(0, p**4 - 1, p)
>>> Parallel(G1, G2).var
p
>>> Parallel(-G3, Series(G1, G2)).var
p
```

**class** sympy.physics.control.lti.**Feedback**(sys1, sys2=None, sign=-1)

A class for representing closed-loop feedback interconnection between two SISO input/output systems.

The first argument, sys1, is the feedforward part of the closed-loop system or in simple words, the dynamical model representing the process to be controlled. The second argument, sys2, is the feedback system and controls the fed back signal to sys1. Both sys1 and sys2 can either be Series or TransferFunction objects.

**Parameters**

**sys1** : Series, TransferFunction

The feedforward path system.

**sys2** : Series, TransferFunction, optional

The feedback path system (often a feedback controller). It is the model sitting on the feedback path.

If not specified explicitly, the sys2 is assumed to be unit (1.0) transfer function.

**sign** : int, optional

The sign of feedback. Can either be 1 (for positive feedback) or -1 (for negative feedback). Default value is -1.

### Raises

#### ValueError

When sys1 and sys2 are not using the same complex variable of the Laplace transform.

When a combination of sys1 and sys2 yields zero denominator.

#### TypeError

When either sys1 or sys2 is not a Series or a TransferFunction object.

## Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction, Feedback
>>> plant = TransferFunction(3*s**2 + 7*s - 3, s**2 - 4*s + 2, s)
>>> controller = TransferFunction(5*s - 10, s + 7, s)
>>> F1 = Feedback(plant, controller)
>>> F1
Feedback(TransferFunction(3*s**2 + 7*s - 3, s**2 - 4*s + 2, s),
↳ TransferFunction(5*s - 10, s + 7, s), -1)
>>> F1.var
s
>>> F1.args
(TransferFunction(3*s**2 + 7*s - 3, s**2 - 4*s + 2, s),
↳ TransferFunction(5*s - 10, s + 7, s), -1)
```

You can get the feedforward and feedback path systems by using `.sys1` and `.sys2` respectively.

```
>>> F1.sys1
TransferFunction(3*s**2 + 7*s - 3, s**2 - 4*s + 2, s)
>>> F1.sys2
TransferFunction(5*s - 10, s + 7, s)
```

You can get the resultant closed loop transfer function obtained by negative feedback interconnection using `.doit()` method.

```
>>> F1.doit()
TransferFunction((s + 7)*(s**2 - 4*s + 2)*(3*s**2 + 7*s - 3), ((s +
↳ 7)*(s**2 - 4*s + 2) + (5*s - 10)*(3*s**2 + 7*s - 3))*(s**2 - 4*s + 2),
↳ s)
>>> G = TransferFunction(2*s**2 + 5*s + 1, s**2 + 2*s + 3, s)
```

(continues on next page)

(continued from previous page)

```
>>> C = TransferFunction(5*s + 10, s + 10, s)
>>> F2 = Feedback(G*C, TransferFunction(1, 1, s))
>>> F2.doit()
TransferFunction((s + 10)*(5*s + 10)*(s**2 + 2*s + 3)*(2*s**2 + 5*s + 1),
↪ (s + 10)*((s + 10)*(s**2 + 2*s + 3) + (5*s + 10)*(2*s**2 + 5*s +
↪ 1))*(s**2 + 2*s + 3), s)
```

To negate a Feedback object, the - operator can be prepended:

```
>>> -F1
Feedback(TransferFunction(-3*s**2 - 7*s + 3, s**2 - 4*s + 2, s),
↪ TransferFunction(10 - 5*s, s + 7, s), -1)
>>> -F2
Feedback(Series(TransferFunction(-1, 1, s), TransferFunction(2*s**2 +
↪ 5*s + 1, s**2 + 2*s + 3, s), TransferFunction(5*s + 10, s + 10, s)),
↪ TransferFunction(-1, 1, s), -1)
```

### See also:

[MIMOFeedback](#) (page 1927), [Series](#) (page 1899), [Parallel](#) (page 1903)

**doit**(*cancel=False, expand=False, \*\*hints*)

Returns the resultant transfer function obtained by the feedback interconnection.

### Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction, Feedback
>>> plant = TransferFunction(3*s**2 + 7*s - 3, s**2 - 4*s + 2, s)
>>> controller = TransferFunction(5*s - 10, s + 7, s)
>>> F1 = Feedback(plant, controller)
>>> F1.doit()
TransferFunction((s + 7)*(s**2 - 4*s + 2)*(3*s**2 + 7*s - 3), ((s +
↪ 7)*(s**2 - 4*s + 2) + (5*s - 10)*(3*s**2 + 7*s - 3))*(s**2 - 4*s +
↪ 2), s)
>>> G = TransferFunction(2*s**2 + 5*s + 1, s**2 + 2*s + 3, s)
>>> F2 = Feedback(G, TransferFunction(1, 1, s))
>>> F2.doit()
TransferFunction((s**2 + 2*s + 3)*(2*s**2 + 5*s + 1), (s**2 + 2*s +
↪ 3)*(3*s**2 + 7*s + 4), s)
```

Use kwarg `expand=True` to expand the resultant transfer function. Use `cancel=True` to cancel out the common terms in numerator and denominator.

```
>>> F2.doit(cancel=True, expand=True)
TransferFunction(2*s**2 + 5*s + 1, 3*s**2 + 7*s + 4, s)
>>> F2.doit(expand=True)
TransferFunction(2*s**4 + 9*s**3 + 17*s**2 + 17*s + 3, 3*s**4 +
↪ 13*s**3 + 27*s**2 + 29*s + 12, s)
```

### property sensitivity

Returns the sensitivity function of the feedback loop.

Sensitivity of a Feedback system is the ratio of change in the open loop gain to the change in the closed loop gain.

---

**Note:** This method would not return the complementary sensitivity function.

---

### Examples

```
>>> from sympy.abc import p
>>> from sympy.physics.control.lti import TransferFunction, Feedback
>>> C = TransferFunction(5*p + 10, p + 10, p)
>>> P = TransferFunction(1 - p, p + 2, p)
>>> F_1 = Feedback(P, C)
>>> F_1.sensitivity
1/((1 - p)*(5*p + 10)/((p + 2)*(p + 10)) + 1)
```

#### property sign

Returns the type of MIMO Feedback model. 1 for Positive and -1 for Negative.

#### property sys1

Returns the feedforward system of the feedback interconnection.

### Examples

```
>>> from sympy.abc import s, p
>>> from sympy.physics.control.lti import TransferFunction, Feedback
>>> plant = TransferFunction(3*s**2 + 7*s - 3, s**2 - 4*s + 2, s)
>>> controller = TransferFunction(5*s - 10, s + 7, s)
>>> F1 = Feedback(plant, controller)
>>> F1.sys1
TransferFunction(3*s**2 + 7*s - 3, s**2 - 4*s + 2, s)
>>> G = TransferFunction(2*s**2 + 5*s + 1, p**2 + 2*p + 3, p)
>>> C = TransferFunction(5*p + 10, p + 10, p)
>>> P = TransferFunction(1 - s, p + 2, p)
>>> F2 = Feedback(TransferFunction(1, 1, p), G*C*P)
>>> F2.sys1
TransferFunction(1, 1, p)
```

#### property sys2

Returns the feedback controller of the feedback interconnection.

## Examples

```
>>> from sympy.abc import s, p
>>> from sympy.physics.control.lti import TransferFunction, Feedback
>>> plant = TransferFunction(3*s**2 + 7*s - 3, s**2 - 4*s + 2, s)
>>> controller = TransferFunction(5*s - 10, s + 7, s)
>>> F1 = Feedback(plant, controller)
>>> F1.sys2
TransferFunction(5*s - 10, s + 7, s)
>>> G = TransferFunction(2*s**2 + 5*s + 1, p**2 + 2*p + 3, p)
>>> C = TransferFunction(5*p + 10, p + 10, p)
>>> P = TransferFunction(1 - s, p + 2, p)
>>> F2 = Feedback(TransferFunction(1, 1, p), G*C*P)
>>> F2.sys2
Series(TransferFunction(2*s**2 + 5*s + 1, p**2 + 2*p + 3, p),
↳ TransferFunction(5*p + 10, p + 10, p), TransferFunction(1 - s, p +
↳ 2, p))
```

### property var

Returns the complex variable of the Laplace transform used by all the transfer functions involved in the feedback interconnection.

## Examples

```
>>> from sympy.abc import s, p
>>> from sympy.physics.control.lti import TransferFunction, Feedback
>>> plant = TransferFunction(3*s**2 + 7*s - 3, s**2 - 4*s + 2, s)
>>> controller = TransferFunction(5*s - 10, s + 7, s)
>>> F1 = Feedback(plant, controller)
>>> F1.var
s
>>> G = TransferFunction(2*s**2 + 5*s + 1, p**2 + 2*p + 3, p)
>>> C = TransferFunction(5*p + 10, p + 10, p)
>>> P = TransferFunction(1 - s, p + 2, p)
>>> F2 = Feedback(TransferFunction(1, 1, p), G*C*P)
>>> F2.var
p
```

**class** sympy.physics.control.lti.TransferFunctionMatrix(*arg*)

A class for representing the MIMO (multiple-input and multiple-output) generalization of the SISO (single-input and single-output) transfer function.

It is a matrix of transfer functions (TransferFunction, SISO-Series or SISO-Parallel). There is only one argument, *arg* which is also the compulsory argument. *arg* is expected to be strictly of the type list of lists which holds the transfer functions or reducible to transfer functions.

### Parameters

**arg** : Nested List (strictly).

Users are expected to input a nested list of TransferFunction, Series and/or Parallel objects.

## Examples

**Note:** `pprint()` can be used for better visualization of `TransferFunctionMatrix` objects.

```
>>> from sympy.abc import s, p, a
>>> from sympy import pprint
>>> from sympy.physics.control.lti import TransferFunction,
↳ TransferFunctionMatrix, Series, Parallel
>>> tf_1 = TransferFunction(s + a, s**2 + s + 1, s)
>>> tf_2 = TransferFunction(p**4 - 3*p + 2, s + p, s)
>>> tf_3 = TransferFunction(3, s + 2, s)
>>> tf_4 = TransferFunction(-a + p, 9*s - 9, s)
>>> tfm_1 = TransferFunctionMatrix([[tf_1], [tf_2], [tf_3]])
>>> tfm_1
TransferFunctionMatrix(((TransferFunction(a + s, s**2 + s + 1, s)),
↳ (TransferFunction(p**4 - 3*p + 2, p + s, s)), (TransferFunction(3, s
↳ + 2, s))),)
>>> tfm_1.var
s
>>> tfm_1.num_inputs
1
>>> tfm_1.num_outputs
3
>>> tfm_1.shape
(3, 1)
>>> tfm_1.args
(((TransferFunction(a + s, s**2 + s + 1, s)), (TransferFunction(p**4 -
↳ 3*p + 2, p + s, s)), (TransferFunction(3, s + 2, s))),)
>>> tfm_2 = TransferFunctionMatrix([[tf_1, -tf_3], [tf_2, -tf_1], [tf_3,
↳ -tf_2]])
>>> tfm_2
TransferFunctionMatrix(((TransferFunction(a + s, s**2 + s + 1, s)),
↳ TransferFunction(-3, s + 2, s)), (TransferFunction(p**4 - 3*p + 2, p +
↳ s, s), TransferFunction(-a - s, s**2 + s + 1, s)), (TransferFunction(3,
↳ s + 2, s), TransferFunction(-p**4 + 3*p - 2, p + s, s)))
>>> pprint(tfm_2, use_unicode=False) # pretty-printing for better
↳ visualization
[a + s -3]
[----- -----]
[2 s + 2]
[s + s + 1]
[]
[4]
[p - 3*p + 2 -a - s]
[----- -----]
[p + s 2]
[]
[]
[]
[]
[]
[3 - p + 3*p - 2]
```

(continues on next page)

(continued from previous page)

```
[----- -----]
[s + 2 p + s]{t}
```

TransferFunctionMatrix can be transposed, if user wants to switch the input and output transfer functions

```
>>> tfm_2.transpose()
TransferFunctionMatrix(((TransferFunction(a + s, s**2 + s + 1, s),
→ TransferFunction(p**4 - 3*p + 2, p + s, s), TransferFunction(3, s + 2,
→ s)), (TransferFunction(-3, s + 2, s), TransferFunction(-a - s, s**2 +
→ s + 1, s), TransferFunction(-p**4 + 3*p - 2, p + s, s))))
>>> pprint(_, use_unicode=False)
[4]
[a + s p - 3*p + 2 3]
[----- ----- -----]
[2 p + s s + 2]
[s + s + 1]
[]
[4]
[-3 -a - s - p + 3*p - 2]
[----- ----- -----]
[s + 2 2 p + s]
[s + s + 1]{t}
```

```
>>> tf_5 = TransferFunction(5, s, s)
>>> tf_6 = TransferFunction(5*s, (2 + s**2), s)
>>> tf_7 = TransferFunction(5, (s*(2 + s**2)), s)
>>> tf_8 = TransferFunction(5, 1, s)
>>> tfm_3 = TransferFunctionMatrix([[tf_5, tf_6], [tf_7, tf_8]])
>>> tfm_3
TransferFunctionMatrix(((TransferFunction(5, s, s), TransferFunction(5*s,
→ s**2 + 2, s)), (TransferFunction(5, s*(s**2 + 2), s),
→ TransferFunction(5, 1, s))))
>>> pprint(tfm_3, use_unicode=False)
[5 5*s]
[- -----]
[s 2]
[s + 2]
[]
[5 5]
[----- -]
[/ 2 \ 1]
[s*\s + 2/]{t}
>>> tfm_3.var
s
>>> tfm_3.shape
(2, 2)
>>> tfm_3.num_outputs
2
>>> tfm_3.num_inputs
2
>>> tfm_3.args
```

(continues on next page)



(continued from previous page)

```
((TransferFunction(5, s, s), TransferFunction(5*s, s**2 + 2, s)),
→(TransferFunction(5, s*(s**2 + 2), s), TransferFunction(5, 1, s))),)
```

To access the TransferFunction at any index in the TransferFunctionMatrix, use the index notation.

```
>>> tfm_3[1, 0] # gives the TransferFunction present at 2nd Row and 1st
→Col. Similar to that in Matrix classes
TransferFunction(5, s*(s**2 + 2), s)
>>> tfm_3[0, 0] # gives the TransferFunction present at 1st Row and 1st
→Col.
TransferFunction(5, s, s)
>>> tfm_3[:, 0] # gives the first column
TransferFunctionMatrix(((TransferFunction(5, s, s)),
→(TransferFunction(5, s*(s**2 + 2), s))),)
>>> pprint(_, use_unicode=False)
[5]
[-]
[s]
[]
[5]
[-----]
[/ 2 \]
[s*\s + 2/]{t}
>>> tfm_3[0, :] # gives the first row
TransferFunctionMatrix(((TransferFunction(5, s, s), TransferFunction(5*s,
→ s**2 + 2, s))),)
>>> pprint(_, use_unicode=False)
[5 5*s]
[- -----]
[s 2]
[s + 2]{t}
```

To negate a transfer function matrix, - operator can be prepended:

```
>>> tfm_4 = TransferFunctionMatrix([[tf_2], [-tf_1], [tf_3]])
>>> -tfm_4
TransferFunctionMatrix(((TransferFunction(-p**4 + 3*p - 2, p + s, s)),
→(TransferFunction(a + s, s**2 + s + 1, s)), (TransferFunction(-3, s +
→2, s))),)
>>> tfm_5 = TransferFunctionMatrix([[tf_1, tf_2], [tf_3, -tf_1]])
>>> -tfm_5
TransferFunctionMatrix(((TransferFunction(-a - s, s**2 + s + 1, s)),
→TransferFunction(-p**4 + 3*p - 2, p + s, s)), (TransferFunction(-3, s
→+ 2, s), TransferFunction(a + s, s**2 + s + 1, s)))
```

subs() returns the TransferFunctionMatrix object with the value substituted in the expression. This will not mutate your original TransferFunctionMatrix.

```
>>> tfm_2.subs(p, 2) # substituting p everywhere in tfm_2 with 2.
TransferFunctionMatrix(((TransferFunction(a + s, s**2 + s + 1, s)),
→TransferFunction(-3, s + 2, s)), (TransferFunction(12, s + 2, s)),
```

(continues on next page)

(continued from previous page)

```

→TransferFunction(-a - s, s**2 + s + 1, s)), (TransferFunction(3, s + 2,
→ s), TransferFunction(-12, s + 2, s)))
>>> pprint(_, use_unicode=False)
[a + s -3]
[-----]
[2 s + 2]
[s + s + 1]
[]
[12 -a - s]
[-----]
[s + 2 2]
[s + s + 1]
[]
[3 -12]
[-----]
[s + 2 s + 2]{t}
>>> pprint(tfm_2, use_unicode=False) # State of tfm_2 is unchanged after
→substitution
[a + s -3]
[-----]
[2 s + 2]
[s + s + 1]
[]
[4]
[p - 3*p + 2 -a - s]
[-----]
[p + s 2]
[s + s + 1]
[]
[4]
[3 - p + 3*p - 2]
[-----]
[s + 2 p + s]{t}

```

`subs()` also supports multiple substitutions.

```

>>> tfm_2.subs({p: 2, a: 1}) # substituting p with 2 and a with 1
TransferFunctionMatrix(((TransferFunction(s + 1, s**2 + s + 1, s),
→TransferFunction(-3, s + 2, s)), (TransferFunction(12, s + 2, s),
→TransferFunction(-s - 1, s**2 + s + 1, s)), (TransferFunction(3, s + 2,
→ s), TransferFunction(-12, s + 2, s))))
>>> pprint(_, use_unicode=False)
[s + 1 -3]
[-----]
[2 s + 2]
[s + s + 1]
[]
[12 -s - 1]
[-----]
[s + 2 2]
[s + s + 1]
[]

```

(continues on next page)

(continued from previous page)

```
[3 -12]
[---- ----]
[s + 2 s + 2]{t}
```

Users can reduce the Series and Parallel elements of the matrix to TransferFunction by using `doit()`.

```
>>> tfm_6 = TransferFunctionMatrix([[Series(tf_3, tf_4), Parallel(tf_3,
→tf_4)]])
>>> tfm_6
TransferFunctionMatrix(((Series(TransferFunction(3, s + 2, s),
→TransferFunction(-a + p, 9*s - 9, s)), Parallel(TransferFunction(3, s
→+ 2, s), TransferFunction(-a + p, 9*s - 9, s))),))
>>> pprint(tfm_6, use_unicode=False)
[-a + p 3 -a + p 3]
[-----*----- + -----]
[9*s - 9 s + 2 9*s - 9 s + 2]{t}
>>> tfm_6.doit()
TransferFunctionMatrix(((TransferFunction(-3*a + 3*p, (s + 2)*(9*s - 9),
→s), TransferFunction(27*s + (-a + p)*(s + 2) - 27, (s + 2)*(9*s - 9),
→s))),))
>>> pprint(_, use_unicode=False)
[-3*a + 3*p 27*s + (-a + p)*(s + 2) - 27]
[-----]
[(s + 2)*(9*s - 9) (s + 2)*(9*s - 9)]{t}
>>> tf_9 = TransferFunction(1, s, s)
>>> tf_10 = TransferFunction(1, s**2, s)
>>> tfm_7 = TransferFunctionMatrix([[Series(tf_9, tf_10), tf_9], [tf_10,
→Parallel(tf_9, tf_10)]])
>>> tfm_7
TransferFunctionMatrix(((Series(TransferFunction(1, s, s),
→TransferFunction(1, s**2, s)), TransferFunction(1, s, s)),
→(TransferFunction(1, s**2, s), Parallel(TransferFunction(1, s, s),
→TransferFunction(1, s**2, s)))))
>>> pprint(tfm_7, use_unicode=False)
[1 1]
[---- -]
[2 s]
[s*s]
[]
[1 1 1]
[-- -- + -]
[2 2 s]
[s s]{t}
>>> tfm_7.doit()
TransferFunctionMatrix(((TransferFunction(1, s**3, s),
→TransferFunction(1, s, s)), (TransferFunction(1, s**2, s),
→TransferFunction(s**2 + s, s**3, s))))
>>> pprint(_, use_unicode=False)
[1 1]
[-- -]
[3 s]
```

(continues on next page)

(continued from previous page)

```
[s]
[]
[2]
[1 s + s]
[-- -----]
[2 3]
[s s]{t}
```

Addition, subtraction, and multiplication of transfer function matrices can form unevaluated `Series` or `Parallel` objects.

- For addition and subtraction: All the transfer function matrices must have the same shape.
- For multiplication ( $C = A * B$ ): The number of inputs of the first transfer function matrix (A) must be equal to the number of outputs of the second transfer function matrix (B).

Also, use pretty-printing (`pprint`) to analyse better.

```
>>> tfm_8 = TransferFunctionMatrix([[tf_3], [tf_2], [-tf_1]])
>>> tfm_9 = TransferFunctionMatrix([[-tf_3]])
>>> tfm_10 = TransferFunctionMatrix([[tf_1], [tf_2], [tf_4]])
>>> tfm_11 = TransferFunctionMatrix([[tf_4], [-tf_1]])
>>> tfm_12 = TransferFunctionMatrix([[tf_4, -tf_1, tf_3], [-tf_2, -tf_4,
↳ -tf_3]])
>>> tfm_8 + tfm_10
MIMOParallel(TransferFunctionMatrix(((TransferFunction(3, s + 2, s)),
↳ (TransferFunction(p**4 - 3*p + 2, p + s, s)), (TransferFunction(-a -
↳ s, s**2 + s + 1, s))), TransferFunctionMatrix(((TransferFunction(a +
↳ s, s**2 + s + 1, s)), (TransferFunction(p**4 - 3*p + 2, p + s, s)),
↳ (TransferFunction(-a + p, 9*s - 9, s))))))
>>> pprint(_, use_unicode=False)
[3] [a + s]
[----] [-----]
[s + 2] [2]
[] [s + s + 1]
[4] []
[p - 3*p + 2] [4]
[-----] + [p - 3*p + 2]
[p + s] [-----]
[] [p + s]
[-a - s] []
[-----] [-a + p]
[2] [-----]
[s + s + 1]{t} [9*s - 9]{t}
>>> -tfm_10 - tfm_8
MIMOParallel(TransferFunctionMatrix(((TransferFunction(-a - s, s**2 + s
↳ + 1, s)), (TransferFunction(-p**4 + 3*p - 2, p + s, s)),
↳ (TransferFunction(a - p, 9*s - 9, s))),
↳ TransferFunctionMatrix(((TransferFunction(-3, s + 2, s)),
↳ (TransferFunction(-p**4 + 3*p - 2, p + s, s)), (TransferFunction(a +
↳ s, s**2 + s + 1, s))))))
```

(continues on next page)

(continued from previous page)

```

>>> pprint(_, use_unicode=False)
[-a - s] [-3]
[-----] [-----]
[2] [s + 2]
[s + s + 1] []
[] [4]
[4] [- p + 3*p - 2]
[- p + 3*p - 2] + [-----]
[-----] [p + s]
[p + s] []
[a - p] [-----]
[-----] [2]
[9*s - 9]{t} [s + s + 1]{t}
>>> tfm_12 * tfm_8
MIMOSeries(TransferFunctionMatrix(((TransferFunction(3, s + 2, s)),
↳ (TransferFunction(p**4 - 3*p + 2, p + s, s)), (TransferFunction(-a -
↳ s, s**2 + s + 1, s))), TransferFunctionMatrix(((TransferFunction(-a
↳ + p, 9*s - 9, s), TransferFunction(-a - s, s**2 + s + 1, s)),
↳ TransferFunction(3, s + 2, s)), (TransferFunction(-p**4 + 3*p - 2, p +
↳ s, s), TransferFunction(a - p, 9*s - 9, s), TransferFunction(-3, s + 2,
↳ s)))))
>>> pprint(_, use_unicode=False)
[-a + p -a - s 3] [3]
[----- ----- -----] [-----]
[9*s - 9 2 s + 2] [s + 2]
[s + s + 1] [4]
[] [p - 3*p + 2]
[] * [-----]
[4] [p + s]
[- p + 3*p - 2 a - p -3] []
[-----] [-----]
[p + s 9*s - 9 s + 2]{t} [-a - s]
[] [-----]
[] [2]
[] [s + s + 1]{t}
>>> tfm_12 * tfm_8 * tfm_9
MIMOSeries(TransferFunctionMatrix(((TransferFunction(-3, s + 2, s)),
↳ TransferFunctionMatrix(((TransferFunction(3, s + 2, s)),
↳ (TransferFunction(p**4 - 3*p + 2, p + s, s)), (TransferFunction(-a -
↳ s, s**2 + s + 1, s))), TransferFunctionMatrix(((TransferFunction(-a
↳ + p, 9*s - 9, s), TransferFunction(-a - s, s**2 + s + 1, s)),
↳ TransferFunction(3, s + 2, s)), (TransferFunction(-p**4 + 3*p - 2, p +
↳ s, s), TransferFunction(a - p, 9*s - 9, s), TransferFunction(-3, s + 2,
↳ s)))))
>>> pprint(_, use_unicode=False)
[-a + p -a - s 3] [3]
[----- ----- -----] [-----]
[9*s - 9 2 s + 2] [s + 2]
[s + s + 1] [4]
[] [p - 3*p + 2] [-3]

```

(continues on next page)

(continued from previous page)

```

[4] * [-----] * [-----]
[] [p + s] [s + 2]{t}
[- p + 3*p - 2 a - p -3] []
[-----] [-a - s]
[p + s 9*s - 9 s + 2]{t} [-----]
[] [2]
[] [s + s + 1]{t}

>>> tfm_10 + tfm_8*tfm_9
MIMOParallel(TransferFunctionMatrix(((TransferFunction(a + s, s**2 + s +
↳ 1, s),), (TransferFunction(p**4 - 3*p + 2, p + s, s),),
↳ (TransferFunction(-a + p, 9*s - 9, s),))),
↳ MIMOSeries(TransferFunctionMatrix(((TransferFunction(-3, s + 2, s),),
↳ (TransferFunction(3, s + 2, s),),
↳ (TransferFunction(p**4 - 3*p + 2, p + s, s),), (TransferFunction(-a -
↳ s, s**2 + s + 1, s),)))))
>>> pprint(_, use_unicode=False)
[a + s] [3]
[-----] [-----]
[2] [s + 2]
[s + s + 1] []
[] [4]
[] [p - 3*p + 2] [-3]
[p - 3*p + 2] + [-----] * [-----]
[-----] [p + s] [s + 2]{t}
[p + s] []
[] [-a - s]
[-a + p] [-----]
[-----] [2]
[9*s - 9]{t} [s + s + 1]{t}

```

These unevaluated Series or Parallel objects can convert into the resultant transfer function matrix using `.doit()` method or by `.rewrite(TransferFunctionMatrix)`.

```

>>> (-tfm_8 + tfm_10 + tfm_8*tfm_9).doit()
TransferFunctionMatrix(((TransferFunction((a + s)*(s + 2)**3 - 3*(s +
↳ 2)**2*(s**2 + s + 1) - 9*(s + 2)*(s**2 + s + 1), (s + 2)**3*(s**2 + s
↳ + 1), s),), (TransferFunction((p + s)*(-3*p**4 + 9*p - 6), (p +
↳ s)**2*(s + 2), s),), (TransferFunction((-a + p)*(s + 2)*(s**2 + s +
↳ 1)**2 + (a + s)*(s + 2)*(9*s - 9)*(s**2 + s + 1) + (3*a + 3*s)*(9*s -
↳ 9)*(s**2 + s + 1), (s + 2)*(9*s - 9)*(s**2 + s + 1)**2, s),)))
>>> (-tfm_12 * -tfm_8 * -tfm_9).rewrite(TransferFunctionMatrix)
TransferFunctionMatrix(((TransferFunction(3*(-3*a + 3*p)*(p + s)*(s +
↳ 2)*(s**2 + s + 1)**2 + 3*(-3*a - 3*s)*(p + s)*(s + 2)*(9*s - 9)*(s**2
↳ + s + 1) + 3*(a + s)*(s + 2)**2*(9*s - 9)*(-p**4 + 3*p - 2)*(s**2 + s
↳ + 1), (p + s)*(s + 2)**3*(9*s - 9)*(s**2 + s + 1)**2, s),),
↳ (TransferFunction(3*(-a + p)*(p + s)*(s + 2)**2*(-p**4 + 3*p -
↳ 2)*(s**2 + s + 1) + 3*(3*a + 3*s)*(p + s)**2*(s + 2)*(9*s - 9) + 3*(p
↳ + s)*(s + 2)*(9*s - 9)*(-3*p**4 + 9*p - 6)*(s**2 + s + 1), (p +
↳ s)**2*(s + 2)**3*(9*s - 9)*(s**2 + s + 1), s),)))

```

**See also:**

[TransferFunction](#) (page 1891), [MIMOSeries](#) (page 1922), [MIMOParallel](#) (page 1925),

[Feedback](#) (page 1906)

### elem\_poles()

Returns the poles of each element of the TransferFunctionMatrix.

---

**Note:** Actual poles of a MIMO system are NOT the poles of individual elements.

---

### Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction,
↳ TransferFunctionMatrix
>>> tf_1 = TransferFunction(3, (s + 1), s)
>>> tf_2 = TransferFunction(s + 6, (s + 1)*(s + 2), s)
>>> tf_3 = TransferFunction(s + 3, s**2 + 3*s + 2, s)
>>> tf_4 = TransferFunction(s + 2, s**2 + 5*s - 10, s)
>>> tfm_1 = TransferFunctionMatrix([[tf_1, tf_2], [tf_3, tf_4]])
>>> tfm_1
TransferFunctionMatrix(((TransferFunction(3, s + 1, s),
↳ TransferFunction(s + 6, (s + 1)*(s + 2), s)), (TransferFunction(s +
↳ 3, s**2 + 3*s + 2, s), TransferFunction(s + 2, s**2 + 5*s - 10,
↳ s))))
>>> tfm_1.elem_poles()
[[[-1], [-2, -1]], [[-2, -1], [-5/2 + sqrt(65)/2, -sqrt(65)/2 - 5/2]]]
```

**See also:**

[elem\\_zeros](#) (page 1919)

### elem\_zeros()

Returns the zeros of each element of the TransferFunctionMatrix.

---

**Note:** Actual zeros of a MIMO system are NOT the zeros of individual elements.

---

### Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction,
↳ TransferFunctionMatrix
>>> tf_1 = TransferFunction(3, (s + 1), s)
>>> tf_2 = TransferFunction(s + 6, (s + 1)*(s + 2), s)
>>> tf_3 = TransferFunction(s + 3, s**2 + 3*s + 2, s)
>>> tf_4 = TransferFunction(s**2 - 9*s + 20, s**2 + 5*s - 10, s)
>>> tfm_1 = TransferFunctionMatrix([[tf_1, tf_2], [tf_3, tf_4]])
>>> tfm_1
TransferFunctionMatrix(((TransferFunction(3, s + 1, s),
↳ TransferFunction(s + 6, (s + 1)*(s + 2), s)), (TransferFunction(s +
↳ 3, s**2 + 3*s + 2, s), TransferFunction(s**2 - 9*s + 20, s**2 + 5*s
```

(continues on next page)

(continued from previous page)

```
↪ - 10, s))))
>>> tfm_1.elem_zeros()
[[[]], [-6]], [[-3], [4, 5]]]
```

**See also:**[\*elem\\_poles\*](#) (page 1919)**expand(\*\*hints)**

Expands the transfer function matrix

**classmethod from\_Matrix(matrix, var)**

Creates a new TransferFunctionMatrix efficiently from a SymPy Matrix of Expr objects.

**Parameters****matrix** : ImmutableMatrix having Expr/Number elements.**var** : Symbol

Complex variable of the Laplace transform which will be used by the all the TransferFunction objects in the TransferFunctionMatrix.

**Examples**

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunctionMatrix
>>> from sympy import Matrix, pprint
>>> M = Matrix([[s, 1/s], [1/(s+1), s]])
>>> M_tf = TransferFunctionMatrix.from_Matrix(M, s)
>>> pprint(M_tf, use_unicode=False)
[s 1]
[- -]
[1 s]
[]
[1 s]
[-----]
[s + 1 1]{t}
>>> M_tf.elem_poles()
[[[]], [0]], [[-1], []]]
>>> M_tf.elem_zeros()
[[[0], []], [[], [0]]]
```

**property num\_inputs**

Returns the number of inputs of the system.



## Examples

```
>>> from sympy.abc import s, p
>>> from sympy.physics.control.lti import TransferFunction, _
↳ TransferFunctionMatrix
>>> G1 = TransferFunction(s + 3, s**2 - 3, s)
>>> G2 = TransferFunction(4, s**2, s)
>>> G3 = TransferFunction(p**2 + s**2, p - 3, s)
>>> tfm_1 = TransferFunctionMatrix([[G2, -G1, G3], [-G2, -G1, -G3]])
>>> tfm_1.num_inputs
3
```

### See also:

[num\\_outputs](#) (page 1921)

### property num\_outputs

Returns the number of outputs of the system.

## Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunctionMatrix
>>> from sympy import Matrix
>>> M_1 = Matrix([[s], [1/s]])
>>> TFM = TransferFunctionMatrix.from_Matrix(M_1, s)
>>> print(TFM)
TransferFunctionMatrix(((TransferFunction(s, 1, s)), _
↳ (TransferFunction(1, s, s))))
>>> TFM.num_outputs
2
```

### See also:

[num\\_inputs](#) (page 1920)

### property shape

Returns the shape of the transfer function matrix, that is, (# of outputs, # of inputs).

## Examples

```
>>> from sympy.abc import s, p
>>> from sympy.physics.control.lti import TransferFunction, _
↳ TransferFunctionMatrix
>>> tf1 = TransferFunction(p**2 - 1, s**4 + s**3 - p, p)
>>> tf2 = TransferFunction(1 - p, p**2 - 3*p + 7, p)
>>> tf3 = TransferFunction(3, 4, p)
>>> tfm1 = TransferFunctionMatrix([[tf1, -tf2]])
>>> tfm1.shape
(1, 2)
```

(continues on next page)

(continued from previous page)

```
>>> tfm2 = TransferFunctionMatrix([[-tf2, tf3], [tf1, -tf1]])
>>> tfm2.shape
(2, 2)
```

**transpose()**

Returns the transpose of the TransferFunctionMatrix (switched input and output layers).

**property var**

Returns the complex variable used by all the transfer functions or Series/Parallel objects in a transfer function matrix.

**Examples**

```
>>> from sympy.abc import p, s
>>> from sympy.physics.control.lti import TransferFunction,
↳ TransferFunctionMatrix, Series, Parallel
>>> G1 = TransferFunction(p**2 + 2*p + 4, p - 6, p)
>>> G2 = TransferFunction(p, 4 - p, p)
>>> G3 = TransferFunction(0, p**4 - 1, p)
>>> G4 = TransferFunction(s + 1, s**2 + s + 1, s)
>>> S1 = Series(G1, G2)
>>> S2 = Series(-G3, Parallel(G2, -G1))
>>> tfm1 = TransferFunctionMatrix([[G1], [G2], [G3]])
>>> tfm1.var
p
>>> tfm2 = TransferFunctionMatrix([[-S1, -S2], [S1, S2]])
>>> tfm2.var
p
>>> tfm3 = TransferFunctionMatrix([[G4]])
>>> tfm3.var
s
```

**class** sympy.physics.control.lti.MIMOSeries(\*args, evaluate=False)

A class for representing a series configuration of MIMO systems.

**Parameters**

**args** : MIMOLinearTimeInvariant

MIMO systems in a series configuration.

**evaluate** : Boolean, Keyword

When passed True, returns the equivalent MIMOSeries(\*args).  
doit(). Set to False by default.

**Raises****ValueError**

When no argument is passed.

var attribute is not same for every system.

num\_outputs of the MIMO system is not equal to the num\_inputs of its adjacent MIMO system. (Matrix multiplication constraint, basically)

**TypeError**

Any of the passed \*args has unsupported type

A combination of SISO and MIMO systems is passed. There should be homogeneity in the type of systems passed, MIMO in this case.

**Examples**

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import MIMOSeries,
↳ TransferFunctionMatrix
>>> from sympy import Matrix, pprint
>>> mat_a = Matrix([[5*s], [5]]) # 2 Outputs 1 Input
>>> mat_b = Matrix([[5, 1/(6*s**2)]]) # 1 Output 2 Inputs
>>> mat_c = Matrix([[1, s], [5/s, 1]]) # 2 Outputs 2 Inputs
>>> tfm_a = TransferFunctionMatrix.from_Matrix(mat_a, s)
>>> tfm_b = TransferFunctionMatrix.from_Matrix(mat_b, s)
>>> tfm_c = TransferFunctionMatrix.from_Matrix(mat_c, s)
>>> MIMOSeries(tfm_c, tfm_b, tfm_a)
MIMOSeries(TransferFunctionMatrix(((TransferFunction(1, 1, s),
↳ TransferFunction(s, 1, s)), (TransferFunction(5, s, s),
↳ TransferFunction(1, 1, s)))),
↳ TransferFunctionMatrix(((TransferFunction(5, 1, s), TransferFunction(1,
↳ 6*s**2, s))), TransferFunctionMatrix(((TransferFunction(5*s, 1, s),
↳), (TransferFunction(5, 1, s),))))
>>> pprint(_, use_unicode=False) # For Better Visualization
[5*s] [1 s]
[---] [5 1] [- -]
[1] [- ----] [1 1]
[] *[1 2] *[]
[5] [6*s]{t} [5 1]
[-] [- -]
[1]{t} [s 1]{t}
>>> MIMOSeries(tfm_c, tfm_b, tfm_a).doit()
TransferFunctionMatrix(((TransferFunction(150*s**4 + 25*s, 6*s**3, s),
↳ TransferFunction(150*s**4 + 5*s, 6*s**2, s)),
↳ (TransferFunction(150*s**3 + 25, 6*s**3, s), TransferFunction(150*s**3
↳ + 5, 6*s**2, s))))
>>> pprint(_, use_unicode=False) # (2 Inputs -A-> 2 Outputs) -> (2
↳ Inputs -B-> 1 Output) -> (1 Input -C-> 2 Outputs) is equivalent to (2
↳ Inputs -Series Equivalent-> 2 Outputs).
[4 4]
[150*s + 25*s 150*s + 5*s]
[-----]
[3 2]
[6*s 6*s]
[]
[3 3]
[150*s + 25 150*s + 5]
[-----]
[3 2]
[6*s 6*s]{t}
```

## Notes

All the transfer function matrices should use the same complex variable `var` of the Laplace transform.

`MIMOSeries(A, B)` is not equivalent to  $A*B$ . It is always in the reverse order, that is  $B*A$ .

### See also:

[`Series`](#) (page 1899), [`MIMOParallel`](#) (page 1925)

**doit**(*cancel=False*, *\*\*kwargs*)

Returns the resultant transfer function matrix obtained after evaluating the MIMO systems arranged in a series configuration.

## Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction,
↳ MIMOSeries, TransferFunctionMatrix
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf2 = TransferFunction(s**3 - 2, s**4 + 5*s + 6, s)
>>> tfm1 = TransferFunctionMatrix([[tf1, tf2], [tf2, tf2]])
>>> tfm2 = TransferFunctionMatrix([[tf2, tf1], [tf1, tf1]])
>>> MIMOSeries(tfm2, tfm1).doit()
TransferFunctionMatrix(((TransferFunction(2*(-p + s)*(s**3 -
↳ 2)*(a*p**2 + b*s)*(s**4 + 5*s + 6), (-p + s)**2*(s**4 + 5*s + 6)**2,
↳ s), TransferFunction((-p + s)**2*(s**3 - 2)*(a*p**2 + b*s) + (-p +
↳ s)*(a*p**2 + b*s)**2*(s**4 + 5*s + 6), (-p + s)**3*(s**4 + 5*s + 6),
↳ s)), (TransferFunction((-p + s)*(s**3 - 2)**2*(s**4 + 5*s + 6) +
↳ (s**3 - 2)*(a*p**2 + b*s)*(s**4 + 5*s + 6)**2, (-p + s)*(s**4 + 5*s
↳ + 6)**3, s), TransferFunction(2*(s**3 - 2)*(a*p**2 + b*s), (-p +
↳ s)*(s**4 + 5*s + 6), s))))
```

### property num\_inputs

Returns the number of input signals of the series system.

### property num\_outputs

Returns the number of output signals of the series system.

### property shape

Returns the shape of the equivalent MIMO system.

### property var

Returns the complex variable used by all the transfer functions.

## Examples

```
>>> from sympy.abc import p
>>> from sympy.physics.control.lti import TransferFunction, \
↳ MIMOSeries, TransferFunctionMatrix
>>> G1 = TransferFunction(p**2 + 2*p + 4, p - 6, p)
>>> G2 = TransferFunction(p, 4 - p, p)
>>> G3 = TransferFunction(0, p**4 - 1, p)
>>> tfm_1 = TransferFunctionMatrix([[G1, G2, G3]])
>>> tfm_2 = TransferFunctionMatrix([[G1], [G2], [G3]])
>>> MIMOSeries(tfm_2, tfm_1).var
p
```

**class** sympy.physics.control.lti.**MIMOParallel**(\*args, evaluate=False)

A class for representing a parallel configuration of MIMO systems.

### Parameters

**args** : MIMOLinearTimeInvariant

MIMO Systems in a parallel arrangement.

**evaluate** : Boolean, Keyword

When passed True, returns the equivalent MIMOParallel(\*args).  
doit(). Set to False by default.

### Raises

#### ValueError

When no argument is passed.

var attribute is not same for every system.

All MIMO systems passed do not have same shape.

#### TypeError

Any of the passed \*args has unsupported type

A combination of SISO and MIMO systems is passed. There should  
be homogeneity in the type of systems passed, MIMO in this case.

## Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunctionMatrix, \
↳ MIMOParallel
>>> from sympy import Matrix, pprint
>>> expr_1 = 1/s
>>> expr_2 = s/(s**2-1)
>>> expr_3 = (2 + s)/(s**2 - 1)
>>> expr_4 = 5
>>> tfm_a = TransferFunctionMatrix.from_Matrix(Matrix([[expr_1, expr_2], \
↳ [expr_3, expr_4]]), s)
>>> tfm_b = TransferFunctionMatrix.from_Matrix(Matrix([[expr_2, expr_1], \
↳ [expr_4, expr_3]]), s)
>>> tfm_c = TransferFunctionMatrix.from_Matrix(Matrix([[expr_3, expr_4], \
↳ [expr_1, expr_2]]), s)
```

(continues on next page)

(continued from previous page)

```

→[expr_1, expr_2]]), s)
>>> MIMOParallel(tfm_a, tfm_b, tfm_c)
MIMOParallel(TransferFunctionMatrix(((TransferFunction(1, s, s),
→TransferFunction(s, s**2 - 1, s)), (TransferFunction(s + 2, s**2 - 1,
→s), TransferFunction(5, 1, s)))),
→TransferFunctionMatrix(((TransferFunction(s, s**2 - 1, s),
→TransferFunction(1, s, s)), (TransferFunction(5, 1, s),
→TransferFunction(s + 2, s**2 - 1, s)))),
→TransferFunctionMatrix(((TransferFunction(s + 2, s**2 - 1, s),
→TransferFunction(5, 1, s)), (TransferFunction(1, s, s),
→TransferFunction(s, s**2 - 1, s)))))
>>> pprint(_, use_unicode=False) # For Better Visualization
[1 s] [s 1] [s + 2 5]
[- - - - -] [- - - - - -] [- - - - - -]
[s 2] [2 s] [2 1]
[s - 1] [s - 1] [s - 1]
[] + [] + []
[s + 2 5] [5 s + 2] [1 s]
[- - - - - -] [- - - - - -] [- - - - - -]
[2 1] [1 2] [s 2]
[s - 1]{t} [s - 1]{t} [s - 1]{t}
>>> MIMOParallel(tfm_a, tfm_b, tfm_c).doit()
TransferFunctionMatrix(((TransferFunction(s**2 + s*(2*s + 2) - 1,
→s*(s**2 - 1), s), TransferFunction(2*s**2 + 5*s*(s**2 - 1) - 1,
→s*(s**2 - 1), s)), (TransferFunction(s**2 + s*(s + 2) + 5*s*(s**2 - 1)
→- 1, s*(s**2 - 1), s), TransferFunction(5*s**2 + 2*s - 3, s**2 - 1,
→s))))
>>> pprint(_, use_unicode=False)
[2] 2 / 2 \]
[s + s*(2*s + 2) - 1 2*s + 5*s*\s - 1/ - 1]
[- - - - -] - - - - -]
[/ 2 \] / 2 \]
[s*\s - 1/] s*\s - 1/]
[]]
[2 / 2 \] 2]
[s + s*(s + 2) + 5*s*\s - 1/ - 1 5*s + 2*s - 3]
[- - - - -] - - - - -]
[/ 2 \] 2]
[s*\s - 1/] s - 1]{t}

```

## Notes

All the transfer function matrices should use the same complex variable `var` of the Laplace transform.

### See also:

[Parallel](#) (page 1903), [MIMOSeries](#) (page 1922)

### `doit(**hints)`

Returns the resultant transfer function matrix obtained after evaluating the MIMO systems arranged in a parallel configuration.

## Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction,
↳ MIMOParallel, TransferFunctionMatrix
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf2 = TransferFunction(s**3 - 2, s**4 + 5*s + 6, s)
>>> tfm_1 = TransferFunctionMatrix([[tf1, tf2], [tf2, tf1]])
>>> tfm_2 = TransferFunctionMatrix([[tf2, tf1], [tf1, tf2]])
>>> MIMOParallel(tfm_1, tfm_2).doit()
TransferFunctionMatrix(((TransferFunction((-p + s)*(s**3 - 2) +
↳ (a*p**2 + b*s)*(s**4 + 5*s + 6), (-p + s)*(s**4 + 5*s + 6), s),
↳ TransferFunction((-p + s)*(s**3 - 2) + (a*p**2 + b*s)*(s**4 + 5*s +
↳ 6), (-p + s)*(s**4 + 5*s + 6), s)), (TransferFunction((-p +
↳ s)*(s**3 - 2) + (a*p**2 + b*s)*(s**4 + 5*s + 6), (-p + s)*(s**4 +
↳ 5*s + 6), s), TransferFunction((-p + s)*(s**3 - 2) + (a*p**2 +
↳ b*s)*(s**4 + 5*s + 6), (-p + s)*(s**4 + 5*s + 6), s))))
```

### property num\_inputs

Returns the number of input signals of the parallel system.

### property num\_outputs

Returns the number of output signals of the parallel system.

### property shape

Returns the shape of the equivalent MIMO system.

### property var

Returns the complex variable used by all the systems.

## Examples

```
>>> from sympy.abc import p
>>> from sympy.physics.control.lti import TransferFunction,
↳ TransferFunctionMatrix, MIMOParallel
>>> G1 = TransferFunction(p**2 + 2*p + 4, p - 6, p)
>>> G2 = TransferFunction(p, 4 - p, p)
>>> G3 = TransferFunction(0, p**4 - 1, p)
>>> G4 = TransferFunction(p**2, p**2 - 1, p)
>>> tfm_a = TransferFunctionMatrix([[G1, G2], [G3, G4]])
>>> tfm_b = TransferFunctionMatrix([[G2, G1], [G4, G3]])
>>> MIMOParallel(tfm_a, tfm_b).var
p
```

**class** sympy.physics.control.lti.**MIMOFeedback**(sys1, sys2, sign=-1)

A class for representing closed-loop feedback interconnection between two MIMO input/output systems.

### Parameters

**sys1** : MIMOSeries, TransferFunctionMatrix

The MIMO system placed on the feedforward path.

**sys2** : MIMOSeries, TransferFunctionMatrix

The system placed on the feedback path (often a feedback controller).

**sign** : int, optional

The sign of feedback. Can either be 1 (for positive feedback) or -1 (for negative feedback). Default value is -1.

### Raises

#### ValueError

When sys1 and sys2 are not using the same complex variable of the Laplace transform.

Forward path model should have an equal number of inputs/outputs to the feedback path outputs/inputs.

When product of sys1 and sys2 is not a square matrix.

When the equivalent MIMO system is not invertible.

#### TypeError

When either sys1 or sys2 is not a MIMOSeries or a TransferFunctionMatrix object.

### Examples

```
>>> from sympy import Matrix, pprint
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunctionMatrix, MIMOFeedback
>>> plant_mat = Matrix([[1, 1/s], [0, 1]])
>>> controller_mat = Matrix([[10, 0], [0, 10]]) # Constant Gain
>>> plant = TransferFunctionMatrix.from_Matrix(plant_mat, s)
>>> controller = TransferFunctionMatrix.from_Matrix(controller_mat, s)
>>> feedback = MIMOFeedback(plant, controller) # Negative Feedback
>>> pprint(feedback, use_unicode=False)
/ [1 1] [10 0] \ -1 [1 1]
| [- -] [-- -] | [- -]
| [1 s] [1 1] | [1 s]
| I + [] * [] | * []
| [0 1] [0 10] | [0 1]
| [- -] [- --] | [- -]
\ [1 1]{t} [1 1]{t}/ [1 1]{t}
```

To get the equivalent system matrix, use either doit or rewrite method.

```
>>> pprint(feedback.doit(), use_unicode=False)
[1 1]
[-- -----]
[11 121*s]
[]
[0 1]
[- -]
[1 11]{t}
```



To negate the MIMOFeedback object, use - operator.

```
>>> neg_feedback = -feedback
>>> pprint(neg_feedback.doit(), use_unicode=False)
[-1 -1]
[--- ----]
[11 121*s]
[]
[0 -1]
[- ---]
[1 11]{t}
```

#### See also:

[Feedback](#) (page 1906), [MIMOSeries](#) (page 1922), [MIMOParallel](#) (page 1925)

**doit**(cancel=True, expand=False, \*\*hints)

Returns the resultant transfer function matrix obtained by the feedback interconnection.

#### Examples

```
>>> from sympy import pprint
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction,
↳ TransferFunctionMatrix, MIMOFeedback
>>> tf1 = TransferFunction(s, 1 - s, s)
>>> tf2 = TransferFunction(1, s, s)
>>> tf3 = TransferFunction(5, 1, s)
>>> tf4 = TransferFunction(s - 1, s, s)
>>> tf5 = TransferFunction(0, 1, s)
>>> sys1 = TransferFunctionMatrix([[tf1, tf2], [tf3, tf4]])
>>> sys2 = TransferFunctionMatrix([[tf3, tf5], [tf5, tf5]])
>>> F_1 = MIMOFeedback(sys1, sys2, 1)
>>> pprint(F_1, use_unicode=False)
/ [s 1] [5 0] \ -1 [s 1]
| [- - - - - -] [- -] | [- - - - - -]
| [1 - s s] [1 1] | [1 - s s]
| I - [] * [] | * []
| [5 s - 1] [0 0] | [5 s - 1]
| [- - - - -] [- -] | [- - - - -]
\ [1 s]{t} [1 1]{t} / [1 s]{t}
>>> pprint(F_1.doit(), use_unicode=False)
[-s s - 1]
[- - - - - -]
[6*s - 1 s*(6*s - 1)]
[]
[5*s - 5 (s - 1)*(6*s + 24)]
[- - - - - -]
[6*s - 1 s*(6*s - 1)]{t}
```

If the user wants the resultant TransferFunctionMatrix object without canceling the common factors then the cancel kwarg should be passed False.

```

>>> pprint(F_1.doit(cancel=False), use_unicode=False)
[25*s*(1 - s) 25 - 25*s]
[]
[----- -----]
[]
[25*(1 - 6*s)*(1 - s) 25*s*(1 - 6*s)]
[]
[]
[]
[s*(25*s - 25) + 5*(1 - s)*(6*s - 1) s*(s - 1)*(6*s - 1) + s*(25*s - 25)]
[]
[----- -----]
[]
[(1 - s)*(6*s - 1) 2]
[]
[s *(6*s - 1)]
[]{t}

```

If the user wants the expanded form of the resultant transfer function matrix, the `expand` kwarg should be passed as `True`.

```

>>> pprint(F_1.doit(expand=True), use_unicode=False)
[-s s - 1]
[----- -----]
[6*s - 1 2]
[6*s - s]
[]
[2]
[5*s - 5 6*s + 18*s - 24]
[----- -----]
[6*s - 1 2]
[6*s - s]{t}

```

### property sensitivity

Returns the sensitivity function matrix of the feedback loop.

Sensitivity of a closed-loop system is the ratio of change in the open loop gain to the change in the closed loop gain.

---

**Note:** This method would not return the complementary sensitivity function.

---

### Examples

```

>>> from sympy import pprint
>>> from sympy.abc import p
>>> from sympy.physics.control.lti import TransferFunction,
↳ TransferFunctionMatrix, MIMOFeedback
>>> tf1 = TransferFunction(p, 1 - p, p)
>>> tf2 = TransferFunction(1, p, p)
>>> tf3 = TransferFunction(1, 1, p)

```

(continues on next page)

(continued from previous page)

```

>>> sys1 = TransferFunctionMatrix([[tf1, tf2], [tf2, tf1]])
>>> sys2 = TransferFunctionMatrix([[tf1, tf3], [tf3, tf2]])
>>> F_1 = MIMOFeedback(sys1, sys2, 1) # Positive feedback
>>> F_2 = MIMOFeedback(sys1, sys2) # Negative feedback
>>> pprint(F_1.sensitivity, use_unicode=False)
[4 3 2 5 4 2]
[- p + 3*p - 4*p + 3*p - 1 p - 2*p + 3*p - 3*p + 1]

[4 3 2 5 4 3 2]
[p + 3*p - 8*p + 8*p - 3 p + 3*p - 8*p + 8*p - 3*p]

[4 3 2 3 2]
[p - p - p + p 3*p - 6*p + 4*p - 1]

[4 3 2 4 3 2]
[p + 3*p - 8*p + 8*p - 3 p + 3*p - 8*p + 8*p - 3]
>>> pprint(F_2.sensitivity, use_unicode=False)
[4 3 2 5 4 2]
[p - 3*p + 2*p + p - 1 p - 2*p + 3*p - 3*p + 1]

[4 3 5 4 2]
[p - 3*p + 2*p - 1 p - 3*p + 2*p - p]

[4 3 4 3]
[p - p - p + p 2*p - 3*p + 2*p - 1]

[4 3 4 3]
[p - 3*p + 2*p - 1 p - 3*p + 2*p - 1]

```

**property sign**

Returns the type of feedback interconnection of two models. 1 for Positive and -1 for Negative.

**property sys1**

Returns the system placed on the feedforward path of the MIMO feedback interconnection.

**Examples**

```

>>> from sympy import pprint
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction,
↳ TransferFunctionMatrix, MIMOFeedback
>>> tf1 = TransferFunction(s**2 + s + 1, s**2 - s + 1, s)
>>> tf2 = TransferFunction(1, s, s)
>>> tf3 = TransferFunction(1, 1, s)
>>> sys1 = TransferFunctionMatrix([[tf1, tf2], [tf2, tf1]])
>>> sys2 = TransferFunctionMatrix([[tf3, tf3], [tf3, tf2]])
>>> F_1 = MIMOFeedback(sys1, sys2, 1)
>>> F_1.sys1
TransferFunctionMatrix(((TransferFunction(s**2 + s + 1, s**2 - s + 1,

```

(continues on next page)

(continued from previous page)

```

↪s), TransferFunction(1, s, s)), (TransferFunction(1, s, s),
↪TransferFunction(s**2 + s + 1, s**2 - s + 1, s)))
>>> pprint(_, use_unicode=False)
[2
[s + s + 1 1]
[----- -]
[2 s]
[s - s + 1]
[
[2]
[1 s + s + 1]
[- -----]
[s 2]
[s - s + 1]{t}

```

**property sys2**

Returns the feedback controller of the MIMO feedback interconnection.

**Examples**

```

>>> from sympy import pprint
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction,
↪TransferFunctionMatrix, MIMOFeedback
>>> tf1 = TransferFunction(s**2, s**3 - s + 1, s)
>>> tf2 = TransferFunction(1, s, s)
>>> tf3 = TransferFunction(1, 1, s)
>>> sys1 = TransferFunctionMatrix([[tf1, tf2], [tf2, tf1]])
>>> sys2 = TransferFunctionMatrix([[tf1, tf3], [tf3, tf2]])
>>> F_1 = MIMOFeedback(sys1, sys2)
>>> F_1.sys2
TransferFunctionMatrix(((TransferFunction(s**2, s**3 - s + 1, s),
↪TransferFunction(1, 1, s)), (TransferFunction(1, 1, s),
↪TransferFunction(1, s, s))))
>>> pprint(_, use_unicode=False)
[2
[s 1]
[----- -]
[3 1]
[s - s + 1]
[
[1 1]
[- -]
[1 s]{t}

```

**property var**

Returns the complex variable of the Laplace transform used by all the transfer functions involved in the MIMO feedback loop.

## Examples

```
>>> from sympy.abc import p
>>> from sympy.physics.control.lti import TransferFunction, \
↳ TransferFunctionMatrix, MIMOFeedback
>>> tf1 = TransferFunction(p, 1 - p, p)
>>> tf2 = TransferFunction(1, p, p)
>>> tf3 = TransferFunction(1, 1, p)
>>> sys1 = TransferFunctionMatrix([[tf1, tf2], [tf2, tf1]])
>>> sys2 = TransferFunctionMatrix([[tf1, tf3], [tf3, tf2]])
>>> F_1 = MIMOFeedback(sys1, sys2, 1) # Positive feedback
>>> F_1.var
p
```

## Control System Plots

This module contains plotting functions for some of the common plots used in control system. Matplotlib will be required as an external dependency if the user wants the plots. To get only the numerical data of the plots, NumPy will be required as external dependency.

### Pole-Zero Plot

```
control_plots.pole_zero_plot(pole_color='blue', pole_markersize=10,
 zero_color='orange', zero_markersize=7, grid=True,
 show_axes=True, show=True, **kwargs)
```

Returns the Pole-Zero plot (also known as PZ Plot or PZ Map) of a system.

A Pole-Zero plot is a graphical representation of a system's poles and zeros. It is plotted on a complex plane, with circular markers representing the system's zeros and 'x' shaped markers representing the system's poles.

#### Parameters

**system** : SISOLinearTimeInvariant type systems

The system for which the pole-zero plot is to be computed.

**pole\_color** : str, tuple, optional

The color of the pole points on the plot. Default color is blue. The color can be provided as a matplotlib color string, or a 3-tuple of floats each in the 0-1 range.

**pole\_markersize** : Number, optional

The size of the markers used to mark the poles in the plot. Default pole markersize is 10.

**zero\_color** : str, tuple, optional

The color of the zero points on the plot. Default color is orange. The color can be provided as a matplotlib color string, or a 3-tuple of floats each in the 0-1 range.

**zero\_markersize** : Number, optional

The size of the markers used to mark the zeros in the plot. Default zero markersize is 7.

**grid** : boolean, optional

If True, the plot will have a grid. Defaults to True.

**show\_axes** : boolean, optional

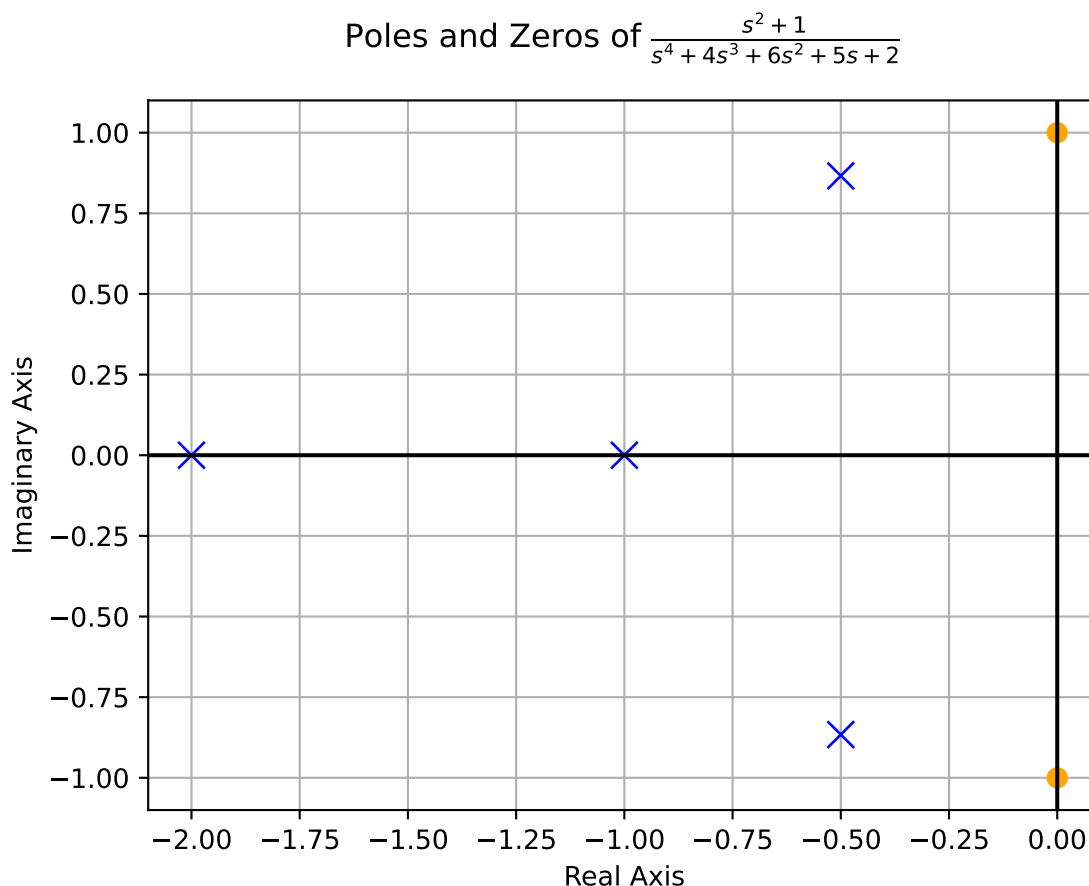
If True, the coordinate axes will be shown. Defaults to False.

**show** : boolean, optional

If True, the plot will be displayed otherwise the equivalent matplotlib plot object will be returned. Defaults to True.

### Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction
>>> from sympy.physics.control.control_plots import pole_zero_plot
>>> tf1 = TransferFunction(s**2 + 1, s**4 + 4*s**3 + 6*s**2 + 5*s + 2, s)
>>> pole_zero_plot(tf1)
```



**See also:**

[\*pole\\_zero\\_numerical\\_data\*](#) (page 1935)

**References**

[R652]

`control_plots.pole_zero_numerical_data()`

Returns the numerical data of poles and zeros of the system. It is internally used by `pole_zero_plot` to get the data for plotting poles and zeros. Users can use this data to further analyse the dynamics of the system or plot using a different backend/plotting-module.

**Parameters**

**system** : SISOLinearTimeInvariant

The system for which the pole-zero data is to be computed.

**Returns**

**tuple** : (zeros, poles)

zeros = Zeros of the system. NumPy array of complex numbers. poles  
= Poles of the system. NumPy array of complex numbers.

**Raises****NotImplementedError**

When a SISO LTI system is not passed.

When time delay terms are present in the system.

**ValueError**

When more than one free symbol is present in the system. The only variable in the transfer function should be the variable of the Laplace transform.

**Examples**

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction
>>> from sympy.physics.control.control_plots import pole_zero_numerical_
→ data
>>> tf1 = TransferFunction(s**2 + 1, s**4 + 4*s**3 + 6*s**2 + 5*s + 2, s)
>>> pole_zero_numerical_data(tf1)
([(-0.+1.j 0.-1.j], [-2. +0.j -0.5+0.8660254j -0.5-0.8660254j -1.+
→ +0.j])
```

**See also:**

[\*pole\\_zero\\_plot\*](#) (page 1933)

## Bode Plot

`control_plots.bode_plot(initial_exp=-5, final_exp=5, grid=True, show_axes=False, show=True, freq_unit='rad/sec', phase_unit='rad', **kwargs)`

Returns the Bode phase and magnitude plots of a continuous-time system.

### Parameters

**system** : SISOLinearTimeInvariant type

The LTI SISO system for which the Bode Plot is to be computed.

**initial\_exp** : Number, optional

The initial exponent of 10 of the semilog plot. Defaults to -5.

**final\_exp** : Number, optional

The final exponent of 10 of the semilog plot. Defaults to 5.

**show** : boolean, optional

If True, the plot will be displayed otherwise the equivalent matplotlib plot object will be returned. Defaults to True.

**prec** : int, optional

The decimal point precision for the point coordinate values. Defaults to 8.

**grid** : boolean, optional

If True, the plot will have a grid. Defaults to True.

**show\_axes** : boolean, optional

If True, the coordinate axes will be shown. Defaults to False.

**freq\_unit** : string, optional

User can choose between 'rad/sec' (radians/second) and 'Hz' (Hertz) as frequency units.

**phase\_unit** : string, optional

User can choose between 'rad' (radians) and 'deg' (degree) as phase units.

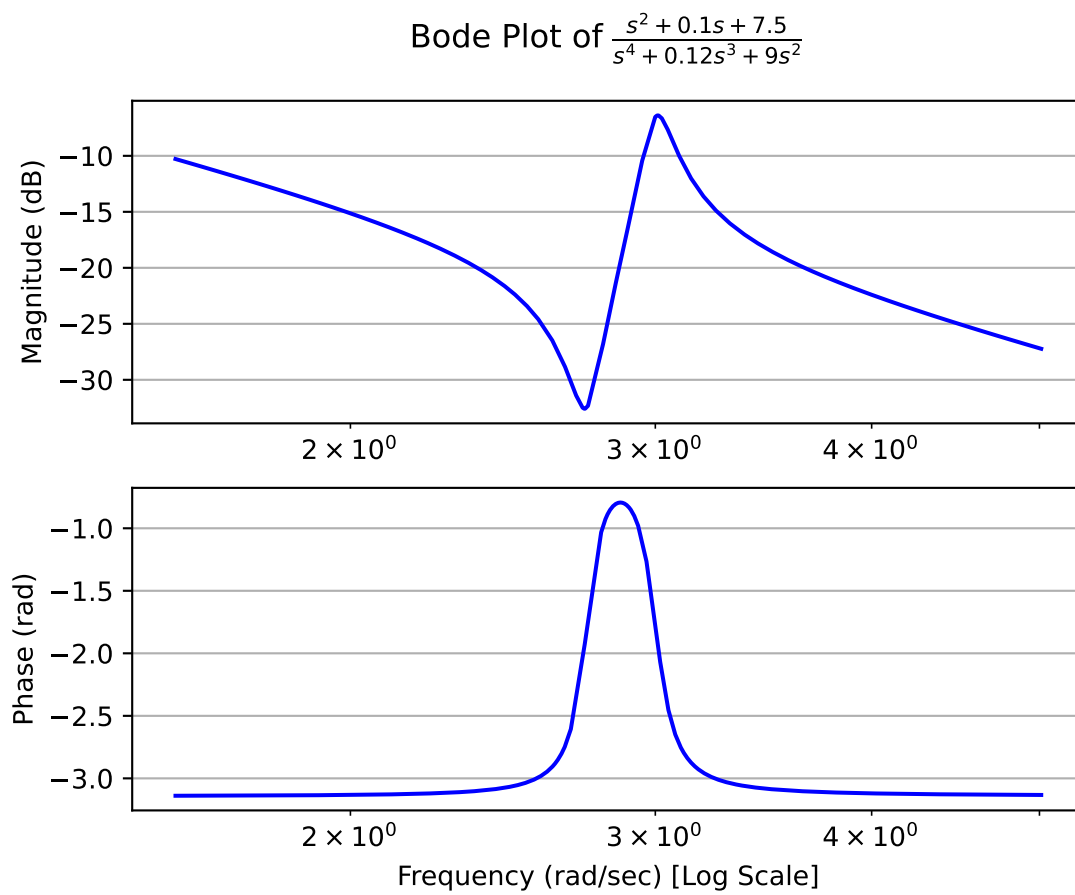
## Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction
>>> from sympy.physics.control.control_plots import bode_plot
>>> tf1 = TransferFunction(1*s**2 + 0.1*s + 7.5, 1*s**4 + 0.12*s**3 +
→ 9*s**2, s)
>>> bode_plot(tf1, initial_exp=0.2, final_exp=0.7)
```

### See also:

[`bode\_magnitude\_plot`](#) (page 1936), [`bode\_phase\_plot`](#) (page 1938)





```
control_plots.bode_magnitude_plot(initial_exp=-5, final_exp=5, color='b',
 show_axes=False, grid=True, show=True,
 freq_unit='rad/sec', **kwargs)
```

Returns the Bode magnitude plot of a continuous-time system.

See `bode_plot` for all the parameters.

```
control_plots.bode_phase_plot(initial_exp=-5, final_exp=5, color='b', show_axes=False,
 grid=True, show=True, freq_unit='rad/sec',
 phase_unit='rad', **kwargs)
```

Returns the Bode phase plot of a continuous-time system.

See `bode_plot` for all the parameters.

```
control_plots.bode_magnitude_numerical_data(initial_exp=-5, final_exp=5,
 freq_unit='rad/sec', **kwargs)
```

Returns the numerical data of the Bode magnitude plot of the system. It is internally used by `bode_magnitude_plot` to get the data for plotting Bode magnitude plot. Users can use this data to further analyse the dynamics of the system or plot using a different backend/plotting-module.

#### Parameters

**system** : SISOLinearTimeInvariant

The system for which the data is to be computed.

**initial\_exp** : Number, optional

The initial exponent of 10 of the semilog plot. Defaults to -5.

**final\_exp** : Number, optional

The final exponent of 10 of the semilog plot. Defaults to 5.

**freq\_unit** : string, optional

User can choose between 'rad/sec' (radians/second) and 'Hz' (Hertz) as frequency units.

#### Returns

**tuple** : (x, y)

x = x-axis values of the Bode magnitude plot. y = y-axis values of the Bode magnitude plot.

#### Raises

**NotImplementedError**

When a SISO LTI system is not passed.

When time delay terms are present in the system.

**ValueError**

When more than one free symbol is present in the system. The only variable in the transfer function should be the variable of the Laplace transform.

When incorrect frequency units are given as input.

## Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction
>>> from sympy.physics.control.control_plots import bode_magnitude_
 numerical_data
>>> tf1 = TransferFunction(s**2 + 1, s**4 + 4*s**3 + 6*s**2 + 5*s + 2, s)
>>> bode_magnitude_numerical_data(tf1)
([1e-05, 1.5148378120533502e-05, ..., 68437.36188804005, 100000.0],
 [-6.020599914256786, -6.0205999155219505, ..., -193.4117304087953, -200.
 -000000000260573])
```

### See also:

[\*bode\\_magnitude\\_plot\*](#) (page 1936), [\*bode\\_phase\\_numerical\\_data\*](#) (page 1939)

```
control_plots.bode_phase_numerical_data(initial_exp=-5, final_exp=5,
 freq_unit='rad/sec', phase_unit='rad',
 **kwargs)
```

Returns the numerical data of the Bode phase plot of the system. It is internally used by `bode_phase_plot` to get the data for plotting Bode phase plot. Users can use this data to further analyse the dynamics of the system or plot using a different backend/plotting-module.

### Parameters

**system** : SISOLinearTimeInvariant

The system for which the Bode phase plot data is to be computed.

**initial\_exp** : Number, optional

The initial exponent of 10 of the semilog plot. Defaults to -5.

**final\_exp** : Number, optional

The final exponent of 10 of the semilog plot. Defaults to 5.

**freq\_unit** : string, optional

User can choose between 'rad/sec' (radians/second) and ``'Hz'`` (Hertz) as frequency units.

**phase\_unit** : string, optional

User can choose between 'rad' (radians) and 'deg' (degree) as phase units.

### Returns

**tuple** : (x, y)

x = x-axis values of the Bode phase plot. y = y-axis values of the Bode phase plot.

### Raises

**NotImplementedError**

When a SISO LTI system is not passed.

When time delay terms are present in the system.

**ValueError**

When more than one free symbol is present in the system. The only variable in the transfer function should be the variable of the Laplace transform.

When incorrect frequency or phase units are given as input.

### Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction
>>> from sympy.physics.control.control_plots import bode_phase_numerical_
 data
>>> tf1 = TransferFunction(s**2 + 1, s**4 + 4*s**3 + 6*s**2 + 5*s + 2, s)
>>> bode_phase_numerical_data(tf1)
([1e-05, 1.4472354033813751e-05, 2.035581932165858e-05, ..., 47577.
 3248186011, 67884.09326036123, 100000.0],
 [-2.5000000000291665e-05, -3.6180885085e-05, -5.08895483066e-05, ..., -3.
 1415085799262523, -3.14155265358979])
```

#### See also:

[\*bode\\_magnitude\\_plot\*](#) (page 1936), [\*bode\\_phase\\_numerical\\_data\*](#) (page 1939)

### Impulse-Response Plot

```
control_plots.impulse_response_plot(color='b', prec=8, lower_limit=0,
 upper_limit=10, show_axes=False, grid=True,
 show=True, **kwargs)
```

Returns the unit impulse response (Input is the Dirac-Delta Function) of a continuous-time system.

#### Parameters

**system** : SISOLinearTimeInvariant type

The LTI SISO system for which the Impulse Response is to be computed.

**color** : str, tuple, optional

The color of the line. Default is Blue.

**show** : boolean, optional

If True, the plot will be displayed otherwise the equivalent matplotlib plot object will be returned. Defaults to True.

**lower\_limit** : Number, optional

The lower limit of the plot range. Defaults to 0.

**upper\_limit** : Number, optional

The upper limit of the plot range. Defaults to 10.

**prec** : int, optional

The decimal point precision for the point coordinate values. Defaults to 8.

**show\_axes** : boolean, optional

If True, the coordinate axes will be shown. Defaults to False.

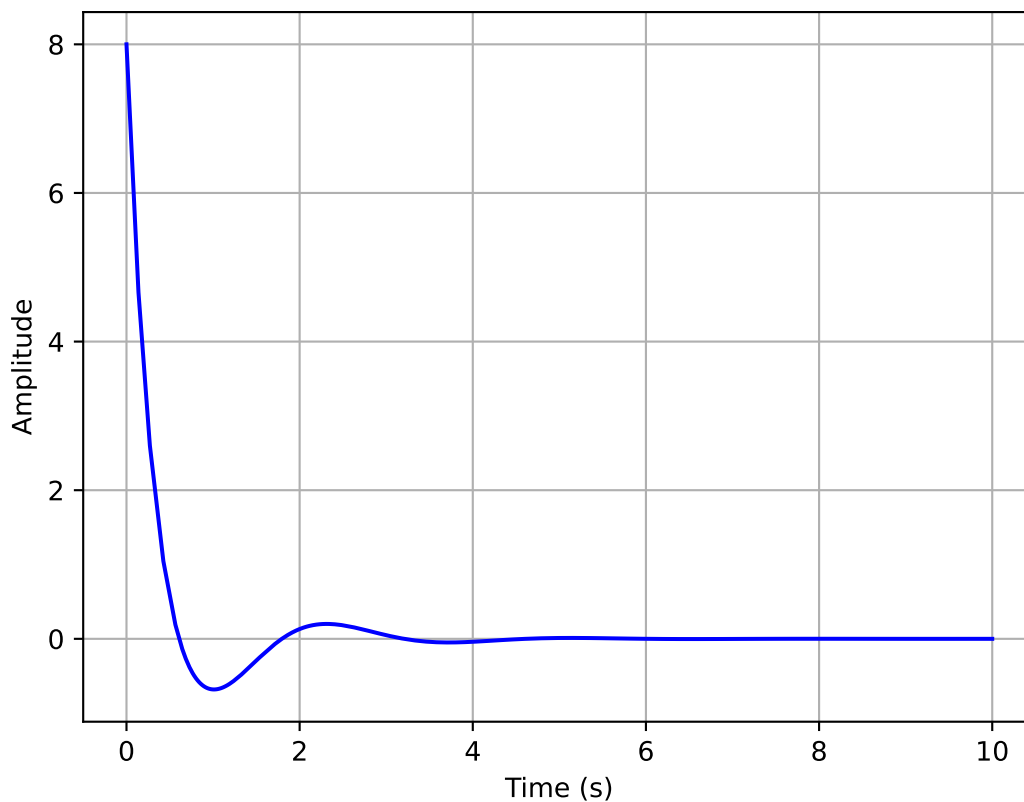
**grid** : boolean, optional

If True, the plot will have a grid. Defaults to True.

## Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction
>>> from sympy.physics.control.control_plots import impulse_response_plot
>>> tf1 = TransferFunction(8*s**2 + 18*s + 32, s**3 + 6*s**2 + 14*s + 24,
↪ s)
>>> impulse_response_plot(tf1)
```

Impulse Response of  $\frac{8s^2 + 18s + 32}{s^3 + 6s^2 + 14s + 24}$



**See also:**

[\*step\\_response\\_plot\*](#) (page 1943), [\*ramp\\_response\\_plot\*](#) (page 1946)

## References

[R653]

`control_plots.impulse_response_numerical_data(prec=8, lower_limit=0,  
upper_limit=10, **kwargs)`

Returns the numerical values of the points in the impulse response plot of a SISO continuous-time system. By default, adaptive sampling is used. If the user wants to instead get an uniformly sampled response, then `adaptive` kwarg should be passed `False` and `nb_of_points` must be passed as additional kwargs. Refer to the parameters of class [`sympy.plotting.plot.LineOver1DRangeSeries`](#) (page 2866) for more details.

### Parameters

**system** : `SISOLinearTimeInvariant`

The system for which the impulse response data is to be computed.

**prec** : int, optional

The decimal point precision for the point coordinate values. Defaults to 8.

**lower\_limit** : Number, optional

The lower limit of the plot range. Defaults to 0.

**upper\_limit** : Number, optional

The upper limit of the plot range. Defaults to 10.

**kwargs** :

Additional keyword arguments are passed to the underlying [`sympy.plotting.plot.LineOver1DRangeSeries`](#) (page 2866) class.

### Returns

**tuple** : (x, y)

x = Time-axis values of the points in the impulse response. NumPy array. y = Amplitude-axis values of the points in the impulse response. NumPy array.

### Raises

**NotImplementedError**

When a SISO LTI system is not passed.

When time delay terms are present in the system.

**ValueError**

When more than one free symbol is present in the system. The only variable in the transfer function should be the variable of the Laplace transform.

When `lower_limit` parameter is less than 0.

## Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction
>>> from sympy.physics.control.control_plots import impulse_response_
 ↪ numerical_data
>>> tf1 = TransferFunction(s, s**2 + 5*s + 8, s)
>>> impulse_response_numerical_data(tf1)
([0.0, 0.06616480200395854,..., 9.854500743565858, 10.0],
 [0.9999999799999999, 0.7042848373025861,..., 7.170748906965121e-13, -5.
 ↪ 1901263495547205e-12])
```

### See also:

[\*impulse\\_response\\_plot\*](#) (page 1940)

## Step-Response Plot

`control_plots.step_response_plot(color='b', prec=8, lower_limit=0, upper_limit=10, show_axes=False, grid=True, show=True, **kwargs)`

Returns the unit step response of a continuous-time system. It is the response of the system when the input signal is a step function.

### Parameters

**system** : SISOLinearTimeInvariant type

The LTI SISO system for which the Step Response is to be computed.

**color** : str, tuple, optional

The color of the line. Default is Blue.

**show** : boolean, optional

If True, the plot will be displayed otherwise the equivalent matplotlib plot object will be returned. Defaults to True.

**lower\_limit** : Number, optional

The lower limit of the plot range. Defaults to 0.

**upper\_limit** : Number, optional

The upper limit of the plot range. Defaults to 10.

**prec** : int, optional

The decimal point precision for the point coordinate values. Defaults to 8.

**show\_axes** : boolean, optional

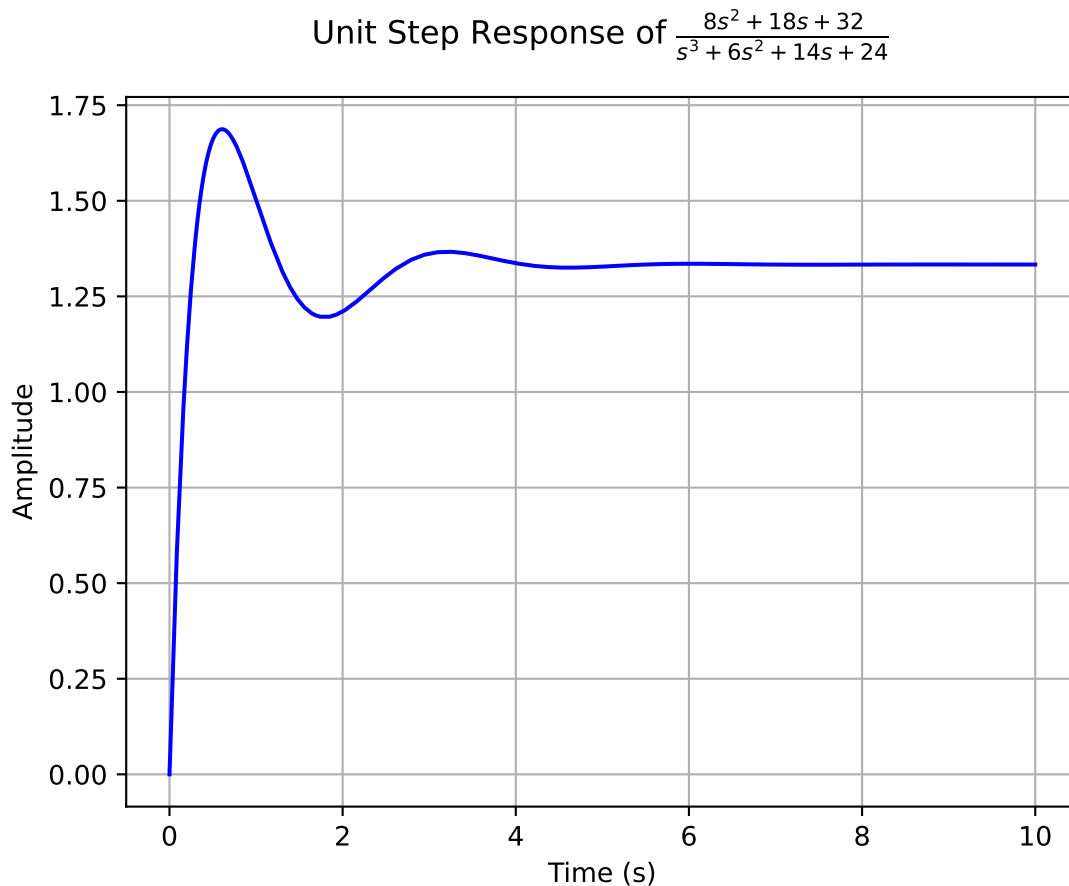
If True, the coordinate axes will be shown. Defaults to False.

**grid** : boolean, optional

If True, the plot will have a grid. Defaults to True.

## Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction
>>> from sympy.physics.control.control_plots import step_response_plot
>>> tf1 = TransferFunction(8*s**2 + 18*s + 32, s**3 + 6*s**2 + 14*s + 24,
 ↪ s)
>>> step_response_plot(tf1)
```



### See also:

[\*impulse\\_response\\_plot\*](#) (page 1940), [\*ramp\\_response\\_plot\*](#) (page 1946)

## References

[R654]

`control_plots.step_response_numerical_data`(*prec*=8, *lower\_limit*=0, *upper\_limit*=10, *\*\*kwargs*)

Returns the numerical values of the points in the step response plot of a SISO continuous-time system. By default, adaptive sampling is used. If the user wants to instead get an uniformly sampled response, then adaptive kwarg should be passed False and



`nb_of_points` must be passed as additional kwargs. Refer to the parameters of class `sympy.plotting.plot.LineOver1DRangeSeries` (page 2866) for more details.

### Parameters

**system** : SISOLinearTimeInvariant

The system for which the unit step response data is to be computed.

**prec** : int, optional

The decimal point precision for the point coordinate values. Defaults to 8.

**lower\_limit** : Number, optional

The lower limit of the plot range. Defaults to 0.

**upper\_limit** : Number, optional

The upper limit of the plot range. Defaults to 10.

**kwargs** :

Additional keyword arguments are passed to the underlying `sympy.plotting.plot.LineOver1DRangeSeries` (page 2866) class.

### Returns

**tuple** : (x, y)

x = Time-axis values of the points in the step response. NumPy array.  
y = Amplitude-axis values of the points in the step response. NumPy array.

### Raises

**NotImplementedError**

When a SISO LTI system is not passed.

When time delay terms are present in the system.

**ValueError**

When more than one free symbol is present in the system. The only variable in the transfer function should be the variable of the Laplace transform.

When `lower_limit` parameter is less than 0.

### Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction
>>> from sympy.physics.control.control_plots import step_response_
 ↪ numerical_data
>>> tf1 = TransferFunction(s, s**2 + 5*s + 8, s)
>>> step_response_numerical_data(tf1)
([0.0, 0.025413462339411542, 0.0484508722725343, ... , 9.670250533855183,
 ↪ 9.844291913708725, 10.0],
 [0.0, 0.023844582399907256, 0.042894276802320226, ..., 6.
 ↪ 828770759094287e-12, 6.456457160755703e-12])
```

See also:

[\*step\\_response\\_plot\*](#) (page 1943)

## Ramp-Response Plot

```
control_plots.ramp_response_plot(slope=1, color='b', prec=8, lower_limit=0,
 upper_limit=10, show_axes=False, grid=True,
 show=True, **kwargs)
```

Returns the ramp response of a continuous-time system.

Ramp function is defined as the straight line passing through origin ( $f(x) = mx$ ). The slope of the ramp function can be varied by the user and the default value is 1.

### Parameters

**system** : SISOLinearTimeInvariant type

The LTI SISO system for which the Ramp Response is to be computed.

**slope** : Number, optional

The slope of the input ramp function. Defaults to 1.

**color** : str, tuple, optional

The color of the line. Default is Blue.

**show** : boolean, optional

If True, the plot will be displayed otherwise the equivalent matplotlib plot object will be returned. Defaults to True.

**lower\_limit** : Number, optional

The lower limit of the plot range. Defaults to 0.

**upper\_limit** : Number, optional

The upper limit of the plot range. Defaults to 10.

**prec** : int, optional

The decimal point precision for the point coordinate values. Defaults to 8.

**show\_axes** : boolean, optional

If True, the coordinate axes will be shown. Defaults to False.

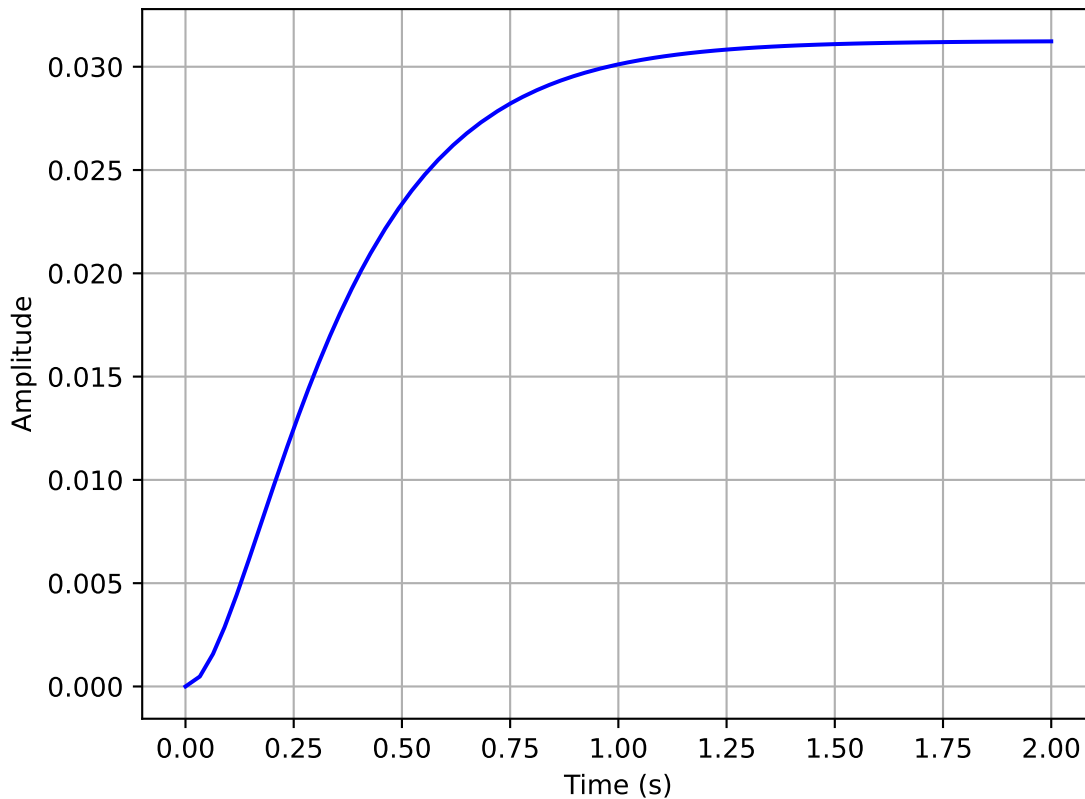
**grid** : boolean, optional

If True, the plot will have a grid. Defaults to True.

## Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction
>>> from sympy.physics.control.control_plots import ramp_response_plot
>>> tf1 = TransferFunction(s, (s+4)*(s+8), s)
>>> ramp_response_plot(tf1, upper_limit=2)
```

Ramp Response of  $\frac{s}{(s+4)(s+8)}$  [Slope = 1]



### See also:

[step\\_response\\_plot](#) (page 1943), [ramp\\_response\\_plot](#) (page 1946)

### References

[R655]

```
control_plots.ramp_response_numerical_data(slope=1, prec=8, lower_limit=0,
 upper_limit=10, **kwargs)
```

Returns the numerical values of the points in the ramp response plot of a SISO continuous-time system. By default, adaptive sampling is used. If the user wants to instead get an uniformly sampled response, then adaptive kwarg should be passed False and nb\_of\_points must be passed as additional kwargs. Refer to the parameters of class [sympy.plotting.plot.LineOver1DRangeSeries](#) (page 2866) for more details.

**Parameters**

**system** : SISOLinearTimeInvariant

The system for which the ramp response data is to be computed.

**slope** : Number, optional

The slope of the input ramp function. Defaults to 1.

**prec** : int, optional

The decimal point precision for the point coordinate values. Defaults to 8.

**lower\_limit** : Number, optional

The lower limit of the plot range. Defaults to 0.

**upper\_limit** : Number, optional

The upper limit of the plot range. Defaults to 10.

**kwargs** :

Additional keyword arguments are passed to the underlying [sympy.plotting.plot.LineOver1DRangeSeries](#) (page 2866) class.

**Returns**

**tuple** : (x, y)

x = Time-axis values of the points in the ramp response plot. NumPy array. y = Amplitude-axis values of the points in the ramp response plot. NumPy array.

**Raises**

**NotImplementedError**

When a SISO LTI system is not passed.

When time delay terms are present in the system.

**ValueError**

When more than one free symbol is present in the system. The only variable in the transfer function should be the variable of the Laplace transform.

When lower\_limit parameter is less than 0.

When slope is negative.

**Examples**

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction
>>> from sympy.physics.control.control_plots import ramp_response_
 numerical_data
>>> tf1 = TransferFunction(s, s**2 + 5*s + 8, s)
>>> ramp_response_numerical_data(tf1)
([([0.0, 0.12166980856813935,..., 9.861246379582118, 10.0],
 [1.4504508011325967e-09, 0.006046440489058766,..., 0.12499999999568202,
 0.12499999999661349]))
```

**See also:**

[\*ramp\\_response\\_plot\*](#) (page 1946)

## Continuum Mechanics

### Abstract

Contains docstrings for methods in continuum mechanics module.

## Beam

### Beam (Docstrings)

## Beam

This module can be used to solve 2D beam bending problems with singularity functions in mechanics.

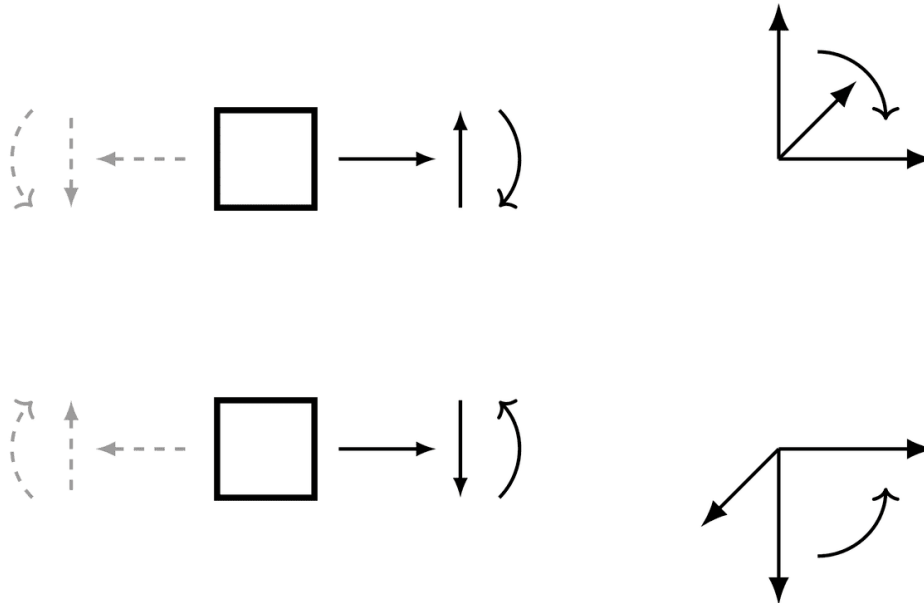
```
class sympy.physics.continuum_mechanics.beam.Beam(length, elastic_modulus,
 second_moment, area=A,
 variable=x, base_char='C')
```

A Beam is a structural element that is capable of withstanding load primarily by resisting against bending. Beams are characterized by their cross sectional profile(Second moment of area), their length and their material.

---

**Note:** A consistent sign convention must be used while solving a beam bending problem; the results will automatically follow the chosen sign convention. However, the chosen sign convention must respect the rule that, on the positive side of beam's axis (in respect to current section), a loading force giving positive shear yields a negative moment, as below (the curved arrow shows the positive moment and rotation):

---



## Examples

There is a beam of length 4 meters. A constant distributed load of 6 N/m is applied from half of the beam till the end. There are two simple supports below the beam, one at the starting point and another at the ending point of the beam. The deflection of the beam at the end is restricted.

Using the sign convention of downwards forces being positive.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols, Piecewise
>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(4, E, I)
>>> b.apply_load(R1, 0, -1)
>>> b.apply_load(6, 2, 0)
>>> b.apply_load(R2, 4, -1)
>>> b.bc_deflection = [(0, 0), (4, 0)]
>>> b.boundary_conditions
{'deflection': [(0, 0), (4, 0)], 'slope': []}
>>> b.load
R1*SingularityFunction(x, 0, -1) + R2*SingularityFunction(x, 4, -1) +
→ 6*SingularityFunction(x, 2, 0)
>>> b.solve_for_reaction_loads(R1, R2)
```

(continues on next page)

(continued from previous page)

```

>>> b.load
-3*SingularityFunction(x, 0, -1) + 6*SingularityFunction(x, 2, 0) -
↳ 9*SingularityFunction(x, 4, -1)
>>> b.shear_force()
3*SingularityFunction(x, 0, 0) - 6*SingularityFunction(x, 2, 1) +
↳ 9*SingularityFunction(x, 4, 0)
>>> b.bending_moment()
3*SingularityFunction(x, 0, 1) - 3*SingularityFunction(x, 2, 2) +
↳ 9*SingularityFunction(x, 4, 1)
>>> b.slope()
(-3*SingularityFunction(x, 0, 2)/2 + SingularityFunction(x, 2, 3) -
↳ 9*SingularityFunction(x, 4, 2)/2 + 7)/(E*I)
>>> b.deflection()
(7*x - SingularityFunction(x, 0, 3)/2 + SingularityFunction(x, 2, 4)/4 -
↳ 3*SingularityFunction(x, 4, 3)/2)/(E*I)
>>> b.deflection().rewrite(Piecewise)
(7*x - Piecewise((x**3, x > 0), (0, True))/2
 - 3*Piecewise((x - 4)**3, x > 4), (0, True))/2
 + Piecewise((x - 2)**4, x > 2), (0, True))/4)/(E*I)

```

**property `applied_loads`**

Returns a list of all loads applied on the beam object. Each load in the list is a tuple of form (value, start, order, end).

**Examples**

There is a beam of length 4 meters. A moment of magnitude 3 Nm is applied in the clockwise direction at the starting point of the beam. A pointload of magnitude 4 N is applied from the top of the beam at 2 meters from the starting point. Another pointload of magnitude 5 N is applied at same position.

```

>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> b = Beam(4, E, I)
>>> b.apply_load(-3, 0, -2)
>>> b.apply_load(4, 2, -1)
>>> b.apply_load(5, 2, -1)
>>> b.load
-3*SingularityFunction(x, 0, -2) + 9*SingularityFunction(x, 2, -1)
>>> b.applied_loads
[(-3, 0, -2, None), (4, 2, -1, None), (5, 2, -1, None)]

```

**`apply_load(value, start, order, end=None)`**

This method adds up the loads given to a particular beam object.

**Parameters**

**value** : Sympifyable

The value inserted should have the units  $[\text{Force}/(\text{Distance}^{n+1})]$  where  $n$  is the order of applied load. Units for applied loads:

- For moments, unit = kN\*m

- For point loads, unit = kN
- For constant distributed load, unit = kN/m
- For ramp loads, unit = kN/m/m
- For parabolic ramp loads, unit = kN/m/m/m
- ... so on.

**start** : Sympifyable

The starting point of the applied load. For point moments and point forces this is the location of application.

**order** : Integer

The order of the applied load.

- For moments, order = -2
- For point loads, order = -1
- For constant distributed load, order = 0
- For ramp loads, order = 1
- For parabolic ramp loads, order = 2
- ... so on.

**end** : Sympifyable, optional

An optional argument that can be used if the load has an end point within the length of the beam.

## Examples

There is a beam of length 4 meters. A moment of magnitude 3 Nm is applied in the clockwise direction at the starting point of the beam. A point load of magnitude 4 N is applied from the top of the beam at 2 meters from the starting point and a parabolic ramp load of magnitude 2 N/m is applied below the beam starting from 2 meters to 3 meters away from the starting point of the beam.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> b = Beam(4, E, I)
>>> b.apply_load(-3, 0, -2)
>>> b.apply_load(4, 2, -1)
>>> b.apply_load(-2, 2, 2, end=3)
>>> b.load
-3*SingularityFunction(x, 0, -2) + 4*SingularityFunction(x, 2, -1) -
↳ 2*SingularityFunction(x, 2, 2) + 2*SingularityFunction(x, 3, 0) +
↳ 4*SingularityFunction(x, 3, 1) + 2*SingularityFunction(x, 3, 2)
```

**apply\_support**(*loc*, *type*='fixed')

This method applies support to a particular beam object.

### Parameters

**loc** : Sympifyable



Location of point at which support is applied.

**type** : String

Determines type of Beam support applied. To apply support structure with - zero degree of freedom, type = "fixed" - one degree of freedom, type = "pin" - two degrees of freedom, type = "roller"

## Examples

There is a beam of length 30 meters. A moment of magnitude 120 Nm is applied in the clockwise direction at the end of the beam. A pointload of magnitude 8 N is applied from the top of the beam at the starting point. There are two simple supports below the beam. One at the end and another one at a distance of 10 meters from the start. The deflection is restricted at both the supports.

Using the sign convention of upward forces and clockwise moment being positive.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> b = Beam(30, E, I)
>>> b.apply_support(10, 'roller')
>>> b.apply_support(30, 'roller')
>>> b.apply_load(-8, 0, -1)
>>> b.apply_load(120, 30, -2)
>>> R_10, R_30 = symbols('R_10, R_30')
>>> b.solve_for_reaction_loads(R_10, R_30)
>>> b.load
-8*SingularityFunction(x, 0, -1) + 6*SingularityFunction(x, 10, -1)
+ 120*SingularityFunction(x, 30, -2) + 2*SingularityFunction(x, 30, -
↪ 1)
>>> b.slope()
(-4*SingularityFunction(x, 0, 2) + 3*SingularityFunction(x, 10, 2)
+ 120*SingularityFunction(x, 30, 1) + SingularityFunction(x, 30,
↪ 2) + 4000/3)/(E*I)
```

## property area

Cross-sectional area of the Beam.

## bending\_moment()

Returns a Singularity Function expression which represents the bending moment curve of the Beam object.

## Examples

There is a beam of length 30 meters. A moment of magnitude 120 Nm is applied in the clockwise direction at the end of the beam. A pointload of magnitude 8 N is applied from the top of the beam at the starting point. There are two simple supports below the beam. One at the end and another one at a distance of 10 meters from the start. The deflection is restricted at both the supports.

Using the sign convention of upward forces and clockwise moment being positive.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(30, E, I)
>>> b.apply_load(-8, 0, -1)
>>> b.apply_load(R1, 10, -1)
>>> b.apply_load(R2, 30, -1)
>>> b.apply_load(120, 30, -2)
>>> b.bc_deflection = [(10, 0), (30, 0)]
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.bending_moment()
8*SingularityFunction(x, 0, 1) - 6*SingularityFunction(x, 10, 1) -
↪ 120*SingularityFunction(x, 30, 0) - 2*SingularityFunction(x, 30, 1)
```

### property boundary\_conditions

Returns a dictionary of boundary conditions applied on the beam. The dictionary has three keywords namely moment, slope and deflection. The value of each keyword is a list of tuple, where each tuple contains location and value of a boundary condition in the format (location, value).

## Examples

There is a beam of length 4 meters. The bending moment at 0 should be 4 and at 4 it should be 0. The slope of the beam should be 1 at 0. The deflection should be 2 at 0.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> b = Beam(4, E, I)
>>> b.bc_deflection = [(0, 2)]
>>> b.bc_slope = [(0, 1)]
>>> b.boundary_conditions
{'deflection': [(0, 2)], 'slope': [(0, 1)]}
```

Here the deflection of the beam should be 2 at 0. Similarly, the slope of the beam should be 1 at 0.

### property cross\_section

Cross-section of the beam

**deflection()**

Returns a Singularity Function expression which represents the elastic curve or deflection of the Beam object.

**Examples**

There is a beam of length 30 meters. A moment of magnitude 120 Nm is applied in the clockwise direction at the end of the beam. A pointload of magnitude 8 N is applied from the top of the beam at the starting point. There are two simple supports below the beam. One at the end and another one at a distance of 10 meters from the start. The deflection is restricted at both the supports.

Using the sign convention of upward forces and clockwise moment being positive.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(30, E, I)
>>> b.apply_load(-8, 0, -1)
>>> b.apply_load(R1, 10, -1)
>>> b.apply_load(R2, 30, -1)
>>> b.apply_load(120, 30, -2)
>>> b.bc_deflection = [(10, 0), (30, 0)]
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.deflection()
(4000*x/3 - 4*SingularityFunction(x, 0, 3)/3 + SingularityFunction(x, 10, 3)
+ 60*SingularityFunction(x, 30, 2) + SingularityFunction(x, 30, 3)/3 - 12000)/(E*I)
```

**draw(pictorial=True)**

Returns a plot object representing the beam diagram of the beam.

**Note:** The user must be careful while entering load values. The draw function assumes a sign convention which is used for plotting loads. Given a right handed coordinate system with XYZ coordinates, the beam's length is assumed to be along the positive X axis. The draw function recognizes positive loads (with  $n > -2$ ) as loads acting along negative Y direction and positive moments acting along positive Z direction.

**Parameters****pictorial: Boolean (default=True)**

Setting `pictorial=True` would simply create a pictorial (scaled) view of the beam diagram not with the exact dimensions. Although setting `pictorial=False` would create a beam diagram with the exact dimensions on the plot.

## Examples

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> R1, R2 = symbols('R1, R2')
>>> E, I = symbols('E, I')
>>> b = Beam(50, 20, 30)
>>> b.apply_load(10, 2, -1)
>>> b.apply_load(R1, 10, -1)
>>> b.apply_load(R2, 30, -1)
>>> b.apply_load(90, 5, 0, 23)
>>> b.apply_load(10, 30, 1, 50)
>>> b.apply_support(50, "pin")
>>> b.apply_support(0, "fixed")
>>> b.apply_support(20, "roller")
>>> p = b.draw()
>>> p
Plot object containing:
[0]: cartesian line: 25*SingularityFunction(x, 5, 0) -
↳ 25*SingularityFunction(x, 23, 0)
+ SingularityFunction(x, 30, 1) - 20*SingularityFunction(x, 50, 0)
- SingularityFunction(x, 50, 1) + 5 for x over (0.0, 50.0)
[1]: cartesian line: 5 for x over (0.0, 50.0)
>>> p.show()
```

### property `elastic_modulus`

Young's Modulus of the Beam.

### property `ild_moment`

Returns the I.L.D. moment equation.

### property `ild_reactions`

Returns the I.L.D. reaction forces in a dictionary.

### property `ild_shear`

Returns the I.L.D. shear equation.

### `join(beam, via='fixed')`

This method joins two beams to make a new composite beam system. Passed Beam class instance is attached to the right end of calling object. This method can be used to form beams having Discontinuous values of Elastic modulus or Second moment.

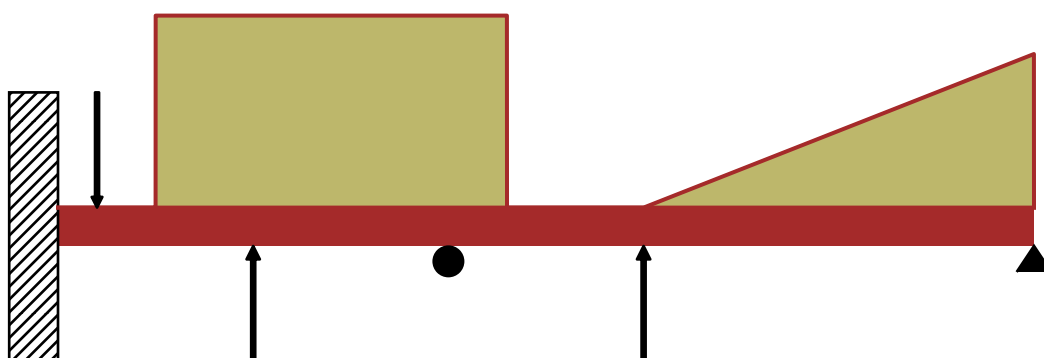
#### Parameters

**beam** : Beam class object

The Beam object which would be connected to the right of calling object.

**via** : String

States the way two Beam object would get connected - For axially fixed Beams, `via="fixed"` - For Beams connected via hinge, `via="hinge"`



## Examples

There is a cantilever beam of length 4 meters. For first 2 meters its moment of inertia is  $1.5*I$  and  $I$  for the other end. A pointload of magnitude 4 N is applied from the top at its free end.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')
>>> b1 = Beam(2, E, 1.5*I)
>>> b2 = Beam(2, E, I)
>>> b = b1.join(b2, "fixed")
>>> b.apply_load(20, 4, -1)
>>> b.apply_load(R1, 0, -1)
>>> b.apply_load(R2, 0, -2)
>>> b.bc_slope = [(0, 0)]
>>> b.bc_deflection = [(0, 0)]
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.load
80*SingularityFunction(x, 0, -2) - 20*SingularityFunction(x, 0, -1) +
↳20*SingularityFunction(x, 4, -1)
>>> b.slope()
(-((-80*SingularityFunction(x, 0, 1) + 10*SingularityFunction(x, 0,
↳2) - 10*SingularityFunction(x, 4, 2))/I + 120/I)/E + 80.0/
↳(E*I))*SingularityFunction(x, 2, 0)
- 0.6666666666666667*(-80*SingularityFunction(x, 0, 1) +
↳10*SingularityFunction(x, 0, 2) - 10*SingularityFunction(x, 4,
↳2))*SingularityFunction(x, 0, 0)/(E*I)
+ 0.6666666666666667*(-80*SingularityFunction(x, 0, 1) +
↳10*SingularityFunction(x, 0, 2) - 10*SingularityFunction(x, 4,
↳2))*SingularityFunction(x, 2, 0)/(E*I)
```

### property length

Length of the Beam.

### property load

Returns a Singularity Function expression which represents the load distribution curve of the Beam object.

## Examples

There is a beam of length 4 meters. A moment of magnitude 3 Nm is applied in the clockwise direction at the starting point of the beam. A point load of magnitude 4 N is applied from the top of the beam at 2 meters from the starting point and a parabolic ramp load of magnitude 2 N/m is applied below the beam starting from 3 meters away from the starting point of the beam.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> b = Beam(4, E, I)
```

(continues on next page)

(continued from previous page)

```

>>> b.apply_load(-3, 0, -2)
>>> b.apply_load(4, 2, -1)
>>> b.apply_load(-2, 3, 2)
>>> b.load
-3*SingularityFunction(x, 0, -2) + 4*SingularityFunction(x, 2, -1) -
↳ 2*SingularityFunction(x, 3, 2)

```

**max\_bmoment()**

Returns maximum Shear force and its coordinate in the Beam object.

**max\_deflection()**

Returns point of max deflection and its corresponding deflection value in a Beam object.

**max\_shear\_force()**

Returns maximum Shear force and its coordinate in the Beam object.

**plot\_bending\_moment(subs=None)**

Returns a plot for Bending moment present in the Beam object.

#### Parameters

**subs** : dictionary

Python dictionary containing Symbols as key and their corresponding values.

## Examples

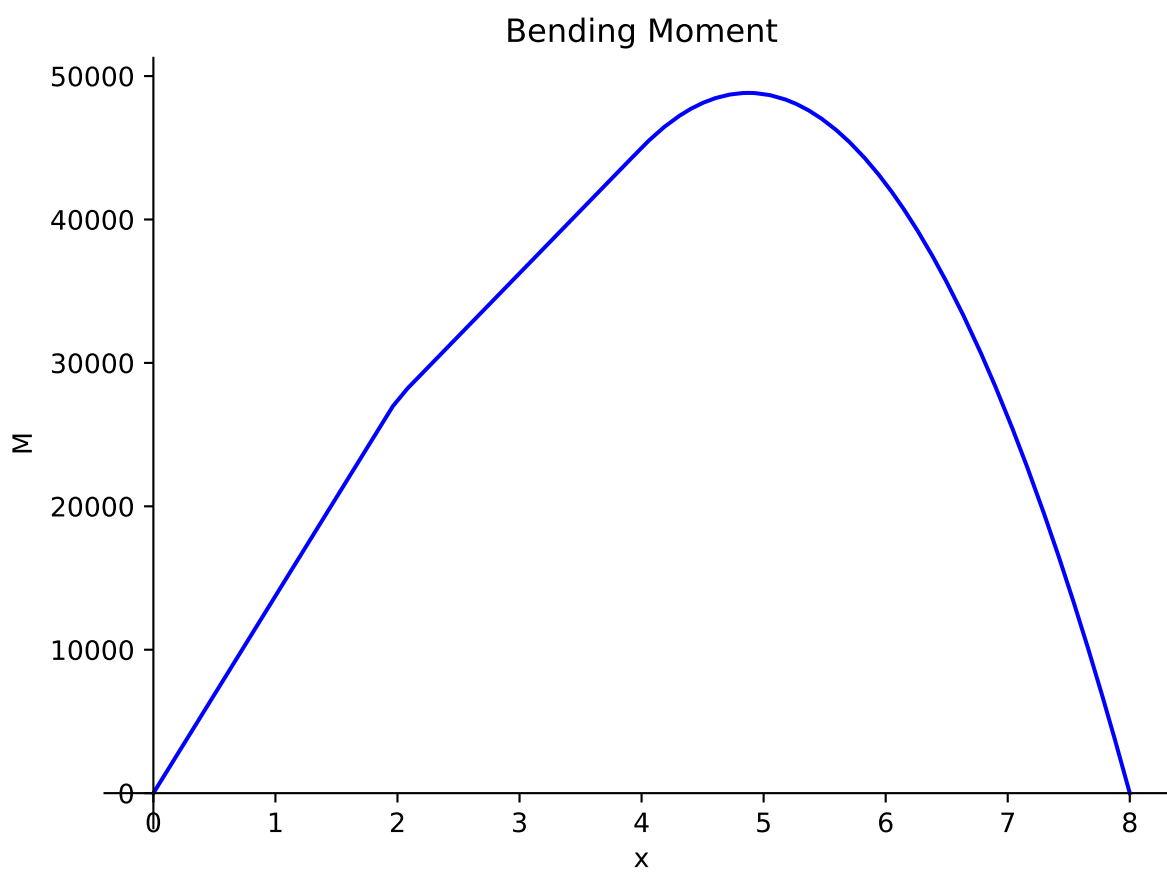
There is a beam of length 8 meters. A constant distributed load of 10 KN/m is applied from half of the beam till the end. There are two simple supports below the beam, one at the starting point and another at the ending point of the beam. A pointload of magnitude 5 KN is also applied from top of the beam, at a distance of 4 meters from the starting point. Take  $E = 200$  GPa and  $I = 400 \cdot (10^{-6})$  meter<sup>4</sup>.

Using the sign convention of downwards forces being positive.

```

>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(8, 200*(10**9), 400*(10**-6))
>>> b.apply_load(5000, 2, -1)
>>> b.apply_load(R1, 0, -1)
>>> b.apply_load(R2, 8, -1)
>>> b.apply_load(10000, 4, 0, end=8)
>>> b.bc_deflection = [(0, 0), (8, 0)]
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.plot_bending_moment()
Plot object containing:
[0]: cartesian line: 13750*SingularityFunction(x, 0, 1) -
↳ 5000*SingularityFunction(x, 2, 1)
- 5000*SingularityFunction(x, 4, 2) + 31250*SingularityFunction(x, 8,
↳ 1)
+ 5000*SingularityFunction(x, 8, 2) for x over (0.0, 8.0)

```





**plot\_deflection**(subs=None)

Returns a plot for deflection curve of the Beam object.

**Parameters**

**subs** : dictionary

Python dictionary containing Symbols as key and their corresponding values.

**Examples**

There is a beam of length 8 meters. A constant distributed load of 10 KN/m is applied from half of the beam till the end. There are two simple supports below the beam, one at the starting point and another at the ending point of the beam. A pointload of magnitude 5 KN is also applied from top of the beam, at a distance of 4 meters from the starting point. Take  $E = 200$  GPa and  $I = 400 \times (10^{-6})$  meter<sup>4</sup>.

Using the sign convention of downwards forces being positive.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(8, 200*(10**9), 400*(10**-6))
>>> b.apply_load(5000, 2, -1)
>>> b.apply_load(R1, 0, -1)
>>> b.apply_load(R2, 8, -1)
>>> b.apply_load(10000, 4, 0, end=8)
>>> b.bc_deflection = [(0, 0), (8, 0)]
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.plot_deflection()
Plot object containing:
[0]: cartesian line: 0.00138541666666667*x - 2.86458333333333e-
 5*SingularityFunction(x, 0, 3)
+ 1.04166666666667e-5*SingularityFunction(x, 2, 3) + 5.
 208333333333333e-6*SingularityFunction(x, 4, 4)
- 6.51041666666667e-5*SingularityFunction(x, 8, 3) - 5.
 208333333333333e-6*SingularityFunction(x, 8, 4)
for x over (0.0, 8.0)
```

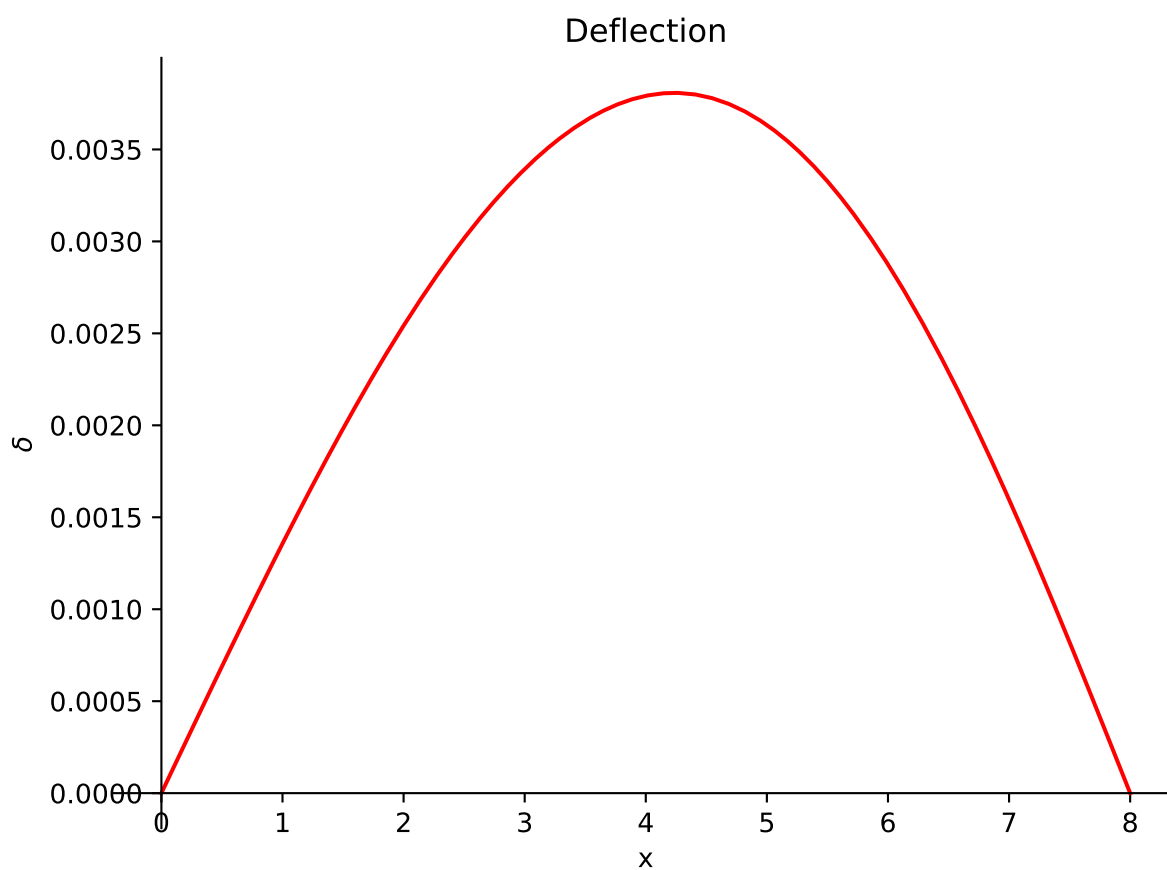
**plot\_ild\_moment**(subs=None)

Plots the Influence Line Diagram for Moment under the effect of a moving load. This function should be called after calling solve\_for\_ild\_moment().

**Parameters**

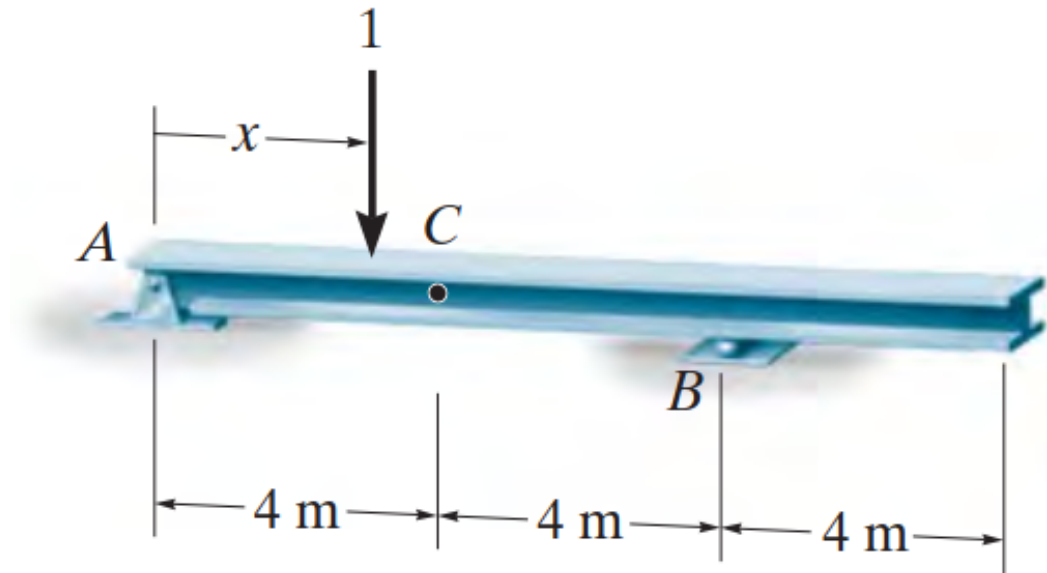
**subs** : dictionary

Python dictionary containing Symbols as key and their corresponding values.



## Examples

There is a beam of length 12 meters. There are two simple supports below the beam, one at the starting point and another at a distance of 8 meters. Plot the I.L.D. for Moment at a distance of 4 meters under the effect of a moving load of magnitude 1kN.



Using the sign convention of downwards forces being positive.

```
>>> from sympy import symbols
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> E, I = symbols('E, I')
>>> R_0, R_8 = symbols('R_0, R_8')
>>> b = Beam(12, E, I)
>>> b.apply_support(0, 'roller')
>>> b.apply_support(8, 'roller')
>>> b.solve_for_ild_reactions(1, R_0, R_8)
>>> b.solve_for_ild_moment(4, 1, R_0, R_8)
>>> b.ild_moment
Piecewise((-x/2, x < 4), (x/2 - 4, x > 4))
>>> b.plot_ild_moment()
Plot object containing:
[0]: cartesian line: Piecewise((-x/2, x < 4), (x/2 - 4, x > 4)) for x_␣
↪ over (0.0, 12.0)
```

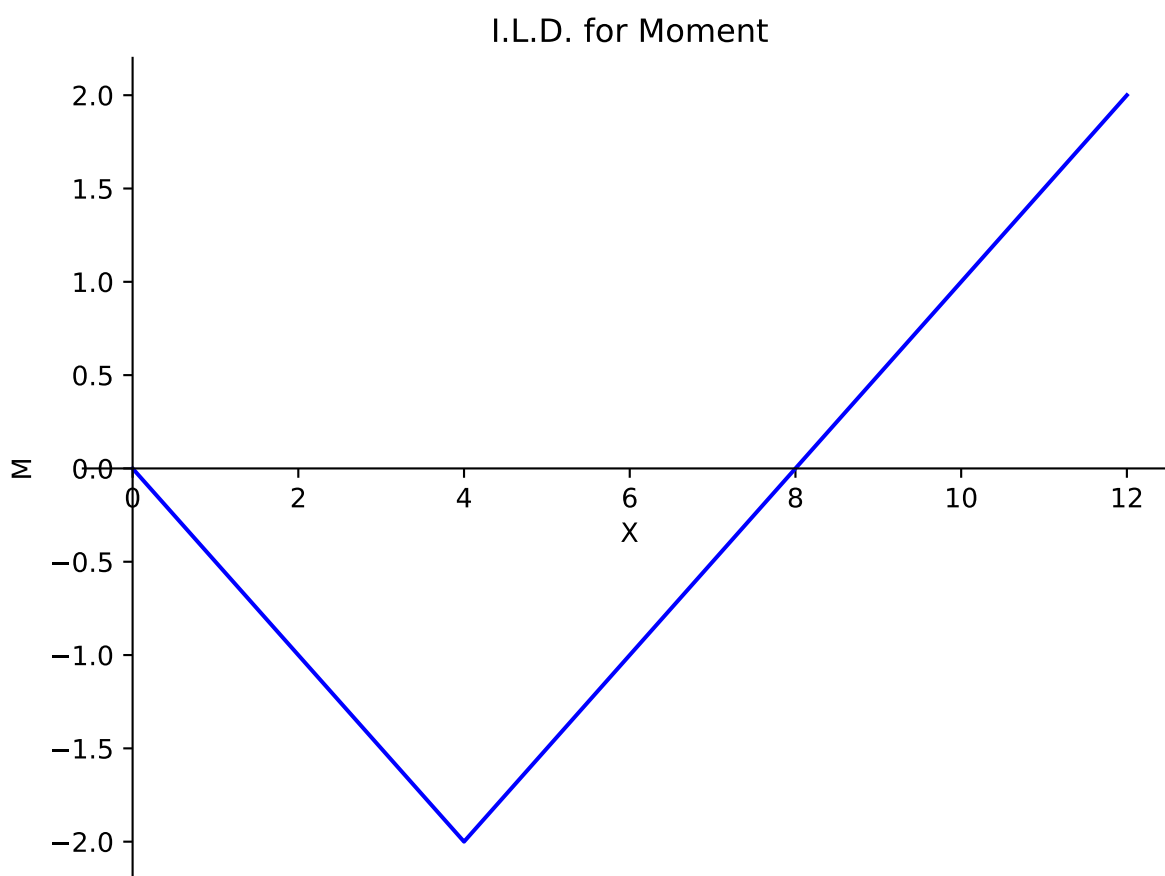
**plot\_ild\_reactions(subs=None)**

Plots the Influence Line Diagram of Reaction Forces under the effect of a moving load. This function should be called after calling solve\_for\_ild\_reactions().

### Parameters

**subs** : dictionary

Python dictionary containing Symbols as key and their corresponding values.



## Examples

There is a beam of length 10 meters. A point load of magnitude 5kN is also applied from top of the beam, at a distance of 4 meters from the starting point. There are two simple supports below the beam, located at the starting point and at a distance of 7 meters from the starting point. Plot the I.L.D. equations for reactions at both support points under the effect of a moving load of magnitude 1kN.

Using the sign convention of downwards forces being positive.

```
>>> from sympy import symbols
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> E, I = symbols('E, I')
>>> R_0, R_7 = symbols('R_0, R_7')
>>> b = Beam(10, E, I)
>>> b.apply_support(0, 'roller')
>>> b.apply_support(7, 'roller')
>>> b.apply_load(5,4,-1)
>>> b.solve_for_ild_reactions(1,R_0,R_7)
>>> b.ild_reactions
{R_0: x/7 - 22/7, R_7: -x/7 - 20/7}
>>> b.plot_ild_reactions()
PlotGrid object containing:
Plot[0]:Plot object containing:
[0]: cartesian line: x/7 - 22/7 for x over (0.0, 10.0)
Plot[1]:Plot object containing:
[0]: cartesian line: -x/7 - 20/7 for x over (0.0, 10.0)
```

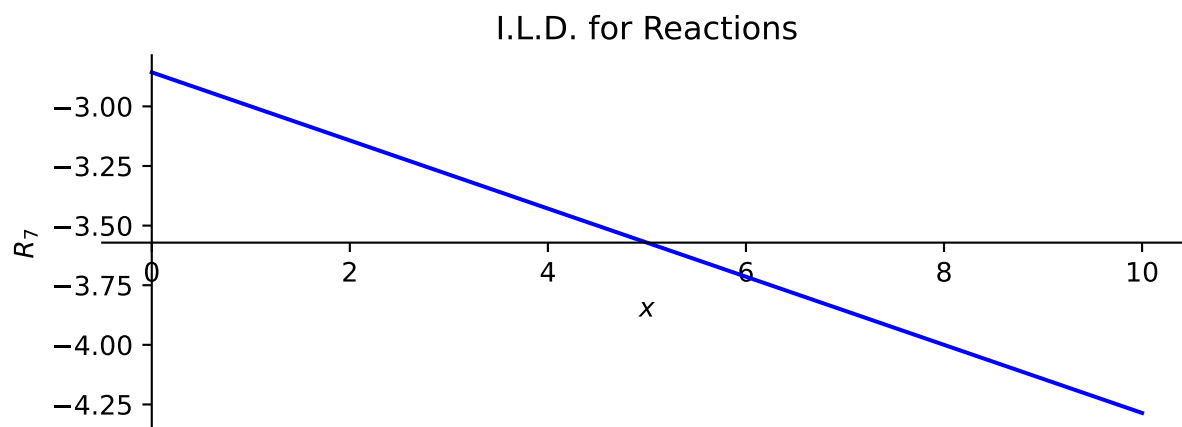
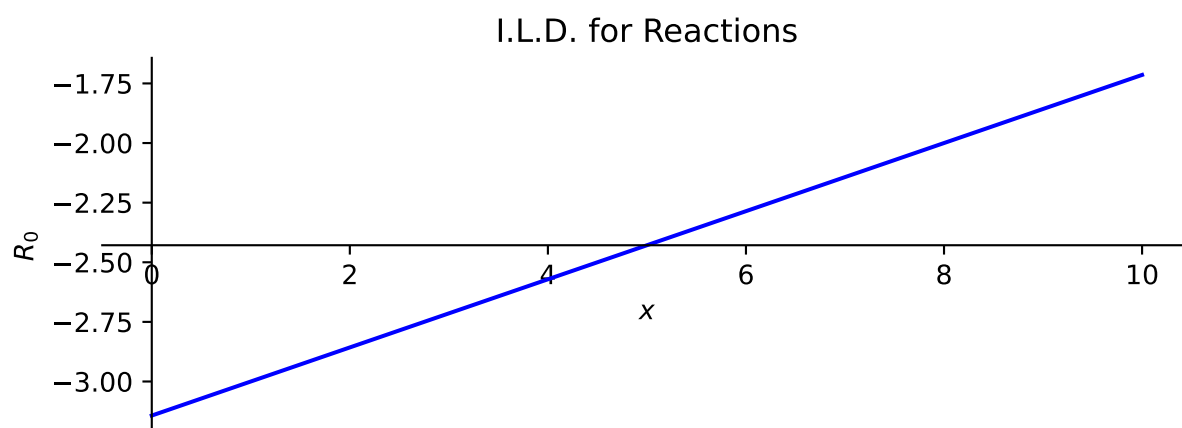
### `plot_ild_shear(subs=None)`

Plots the Influence Line Diagram for Shear under the effect of a moving load. This function should be called after calling `solve_for_ild_shear()`.

#### Parameters

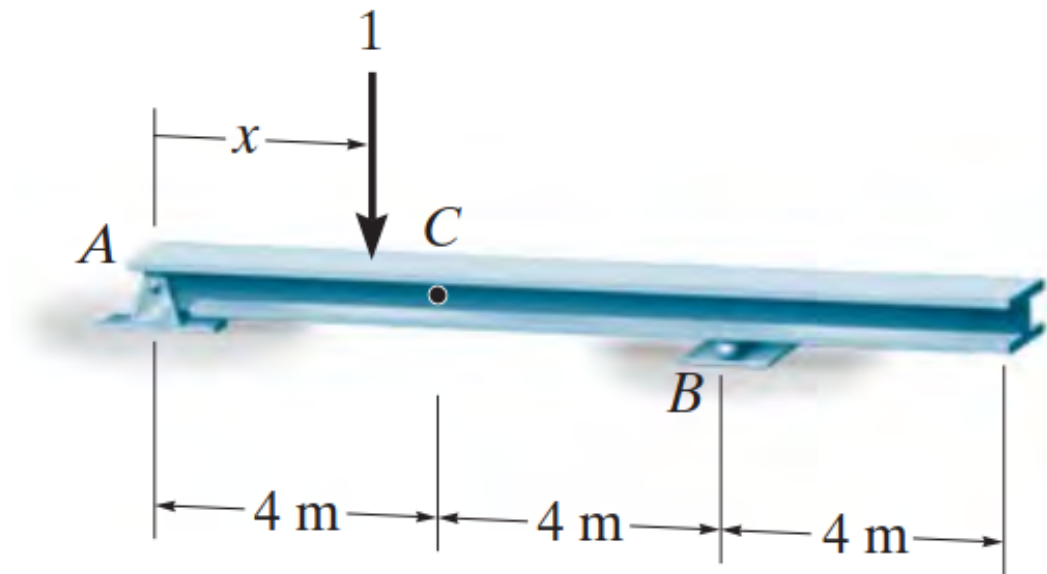
**subs** : dictionary

Python dictionary containing Symbols as key and their corresponding values.



## Examples

There is a beam of length 12 meters. There are two simple supports below the beam, one at the starting point and another at a distance of 8 meters. Plot the I.L.D. for Shear at a distance of 4 meters under the effect of a moving load of magnitude 1kN.



Using the sign convention of downwards forces being positive.

```
>>> from sympy import symbols
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> E, I = symbols('E, I')
>>> R_0, R_8 = symbols('R_0, R_8')
>>> b = Beam(12, E, I)
>>> b.apply_support(0, 'roller')
>>> b.apply_support(8, 'roller')
>>> b.solve_for_ild_reactions(1, R_0, R_8)
>>> b.solve_for_ild_shear(4, 1, R_0, R_8)
>>> b.ild_shear
Piecewise((x/8, x < 4), (x/8 - 1, x > 4))
>>> b.plot_ild_shear()
Plot object containing:
[0]: cartesian line: Piecewise((x/8, x < 4), (x/8 - 1, x > 4)) for x_
→over (0.0, 12.0)
```

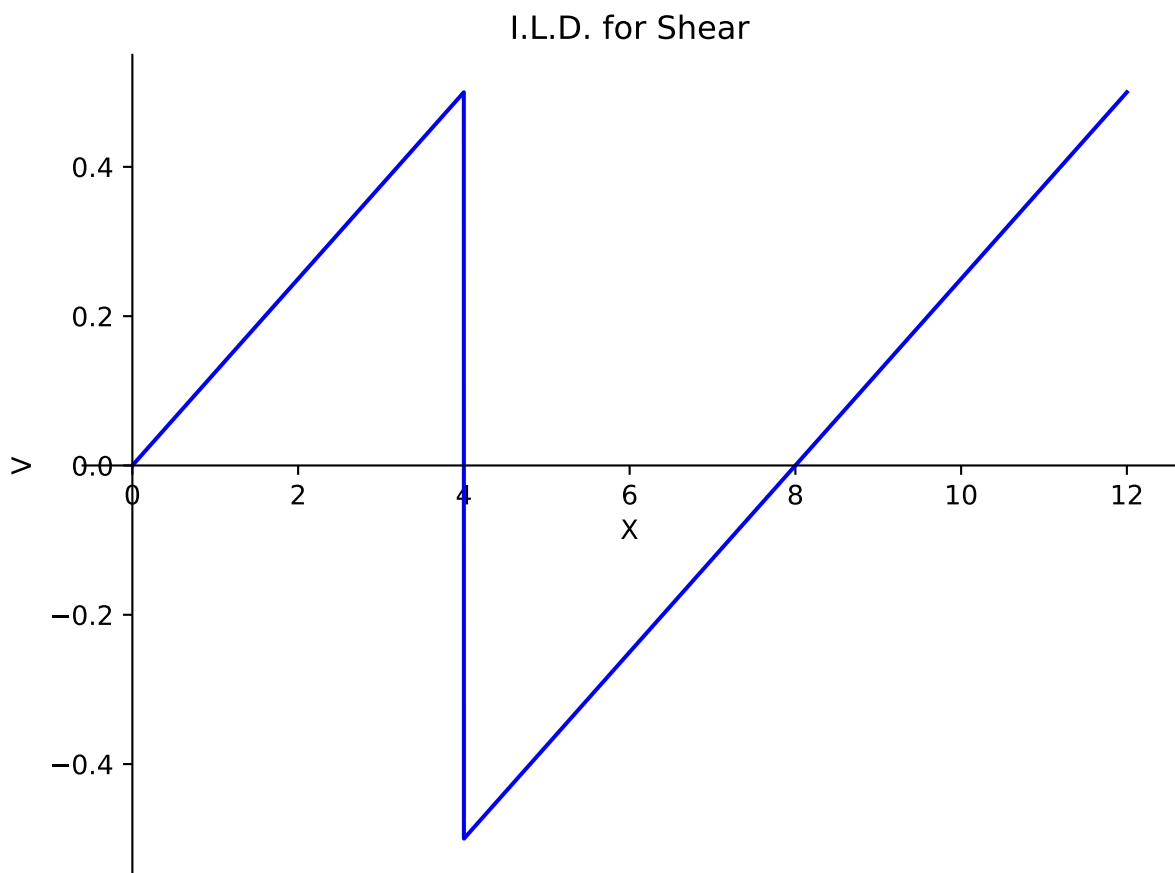
**plot\_loading\_results**(subs=None)

Returns a subplot of Shear Force, Bending Moment, Slope and Deflection of the Beam object.

### Parameters

**subs** : dictionary

Python dictionary containing Symbols as key and their corresponding values.



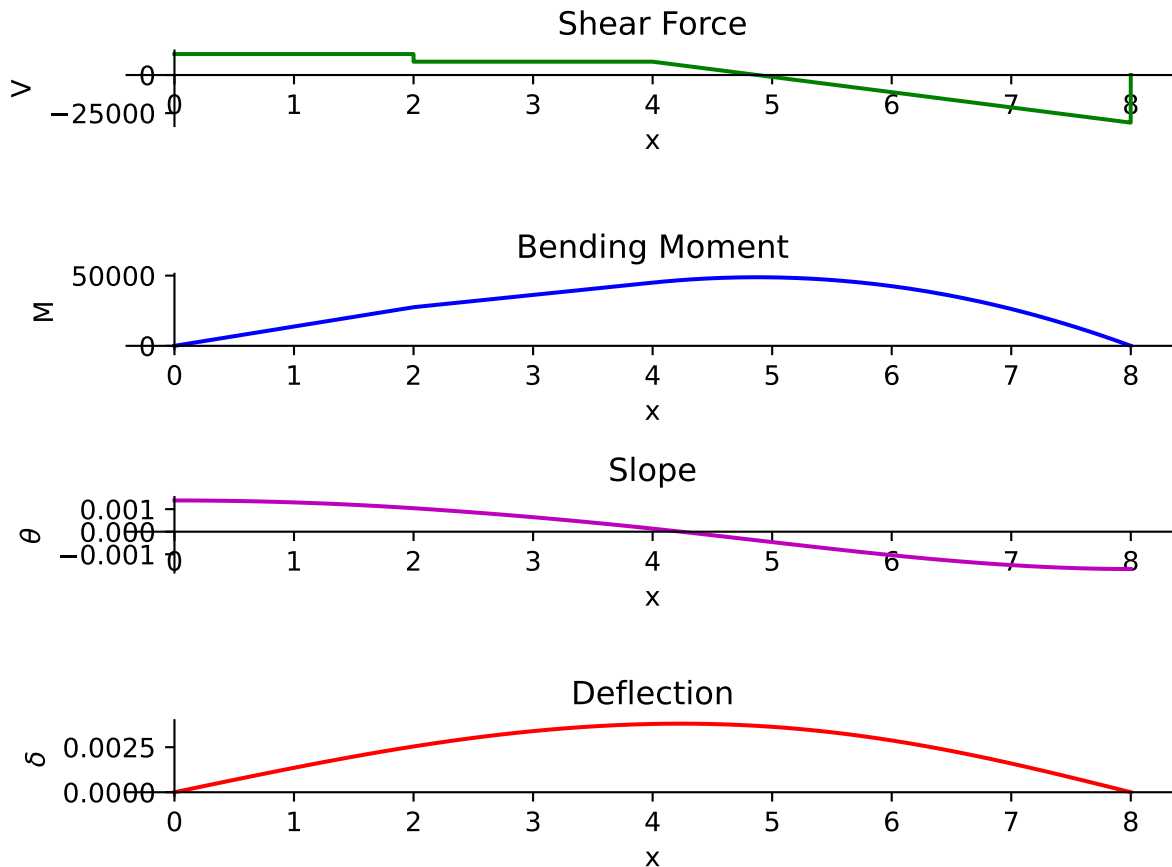


## Examples

There is a beam of length 8 meters. A constant distributed load of 10 KN/m is applied from half of the beam till the end. There are two simple supports below the beam, one at the starting point and another at the ending point of the beam. A pointload of magnitude 5 KN is also applied from top of the beam, at a distance of 4 meters from the starting point. Take  $E = 200$  GPa and  $I = 400 \times (10^{-6})$  meter<sup>4</sup>.

Using the sign convention of downwards forces being positive.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(8, 200*(10**9), 400*(10**-6))
>>> b.apply_load(5000, 2, -1)
>>> b.apply_load(R1, 0, -1)
>>> b.apply_load(R2, 8, -1)
>>> b.apply_load(10000, 4, 0, end=8)
>>> b.bc_deflection = [(0, 0), (8, 0)]
>>> b.solve_for_reaction_loads(R1, R2)
>>> axes = b.plot_loading_results()
```



`plot_shear_force(subs=None)`

Returns a plot for Shear force present in the Beam object.

**Parameters****subs** : dictionary

Python dictionary containing Symbols as key and their corresponding values.

**Examples**

There is a beam of length 8 meters. A constant distributed load of 10 KN/m is applied from half of the beam till the end. There are two simple supports below the beam, one at the starting point and another at the ending point of the beam. A pointload of magnitude 5 KN is also applied from top of the beam, at a distance of 4 meters from the starting point. Take  $E = 200$  GPa and  $I = 400 \cdot (10^{-6}) \text{ meter}^4$ .

Using the sign convention of downwards forces being positive.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(8, 200*(10**9), 400*(10**-6))
>>> b.apply_load(5000, 2, -1)
>>> b.apply_load(R1, 0, -1)
>>> b.apply_load(R2, 8, -1)
>>> b.apply_load(10000, 4, 0, end=8)
>>> b.bc_deflection = [(0, 0), (8, 0)]
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.plot_shear_force()
Plot object containing:
[0]: cartesian line: 13750*SingularityFunction(x, 0, 0) -
↳ 5000*SingularityFunction(x, 2, 0)
- 10000*SingularityFunction(x, 4, 1) + 31250*SingularityFunction(x, 8,
↳ 0)
+ 10000*SingularityFunction(x, 8, 1) for x over (0.0, 8.0)
```

**plot\_shear\_stress(subs=None)**

Returns a plot of shear stress present in the beam object.

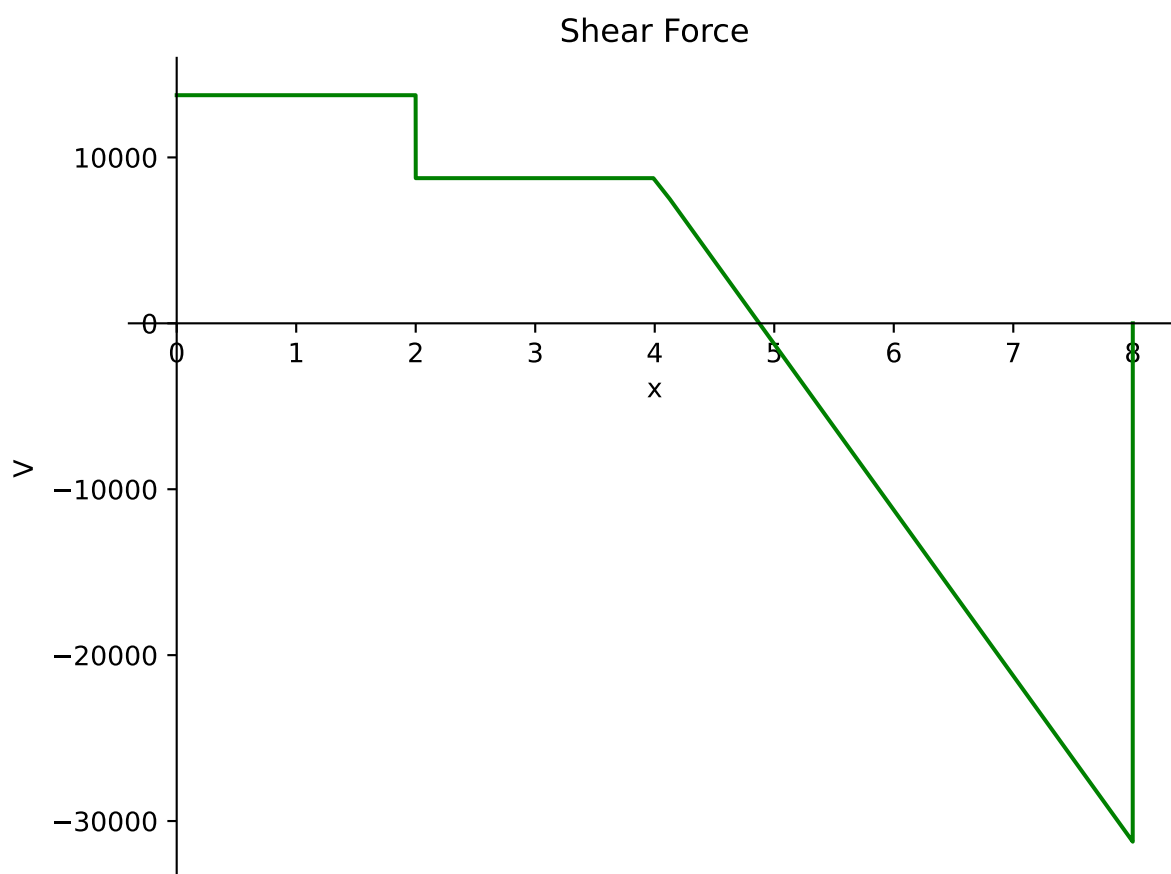
**Parameters****subs** : dictionary

Python dictionary containing Symbols as key and their corresponding values.

**Examples**

There is a beam of length 8 meters and area of cross section 2 square meters. A constant distributed load of 10 KN/m is applied from half of the beam till the end. There are two simple supports below the beam, one at the starting point and another at the ending point of the beam. A pointload of magnitude 5 KN is also applied from top of the beam, at a distance of 4 meters from the starting point. Take  $E = 200$  GPa and  $I = 400 \cdot (10^{-6}) \text{ meter}^4$ .

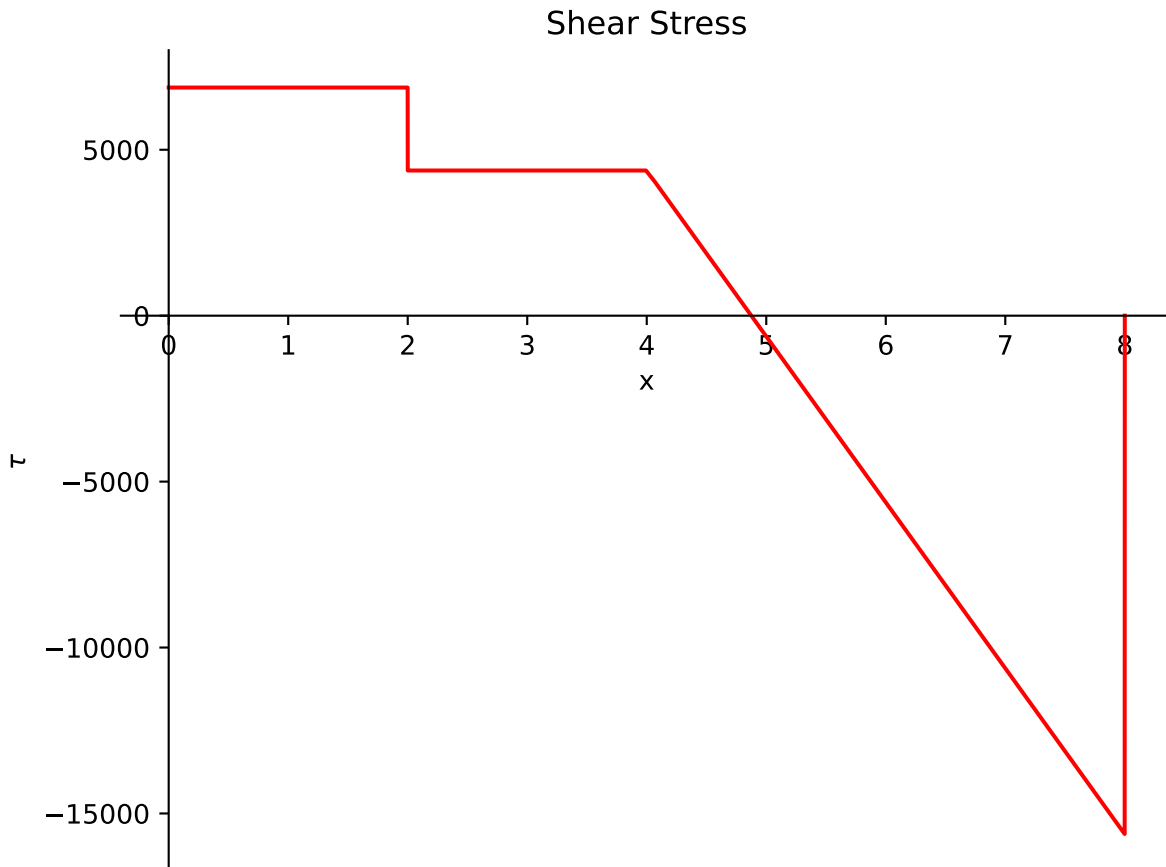
Using the sign convention of downwards forces being positive.



```

>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(8, 200*(10**9), 400*(10**-6), 2)
>>> b.apply_load(5000, 2, -1)
>>> b.apply_load(R1, 0, -1)
>>> b.apply_load(R2, 8, -1)
>>> b.apply_load(10000, 4, 0, end=8)
>>> b.bc_deflection = [(0, 0), (8, 0)]
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.plot_shear_stress()
Plot object containing:
[0]: cartesian line: 6875*SingularityFunction(x, 0, 0) -
 2500*SingularityFunction(x, 2, 0)
- 5000*SingularityFunction(x, 4, 1) + 15625*SingularityFunction(x, 8,
 0)
+ 5000*SingularityFunction(x, 8, 1) for x over (0.0, 8.0)

```



**plot\_slope**(subs=None)

Returns a plot for slope of deflection curve of the Beam object.

**Parameters**

**subs** : dictionary

Python dictionary containing Symbols as key and their corresponding values.

## Examples

There is a beam of length 8 meters. A constant distributed load of 10 KN/m is applied from half of the beam till the end. There are two simple supports below the beam, one at the starting point and another at the ending point of the beam. A pointload of magnitude 5 KN is also applied from top of the beam, at a distance of 4 meters from the starting point. Take  $E = 200$  GPa and  $I = 400 \cdot (10^{-6})$  meter<sup>4</sup>.

Using the sign convention of downwards forces being positive.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(8, 200*(10**9), 400*(10**-6))
>>> b.apply_load(5000, 2, -1)
>>> b.apply_load(R1, 0, -1)
>>> b.apply_load(R2, 8, -1)
>>> b.apply_load(10000, 4, 0, end=8)
>>> b.bc_deflection = [(0, 0), (8, 0)]
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.plot_slope()
Plot object containing:
[0]: cartesian line: -8.59375e-5*SingularityFunction(x, 0, 2) + 3.
 ↪ 125e-5*SingularityFunction(x, 2, 2)
+ 2.08333333333333e-5*SingularityFunction(x, 4, 3) - 0.
 ↪ 0001953125*SingularityFunction(x, 8, 2)
- 2.08333333333333e-5*SingularityFunction(x, 8, 3) + 0.
 ↪ 00138541666666667 for x over (0.0, 8.0)
```

### point\_cflexture()

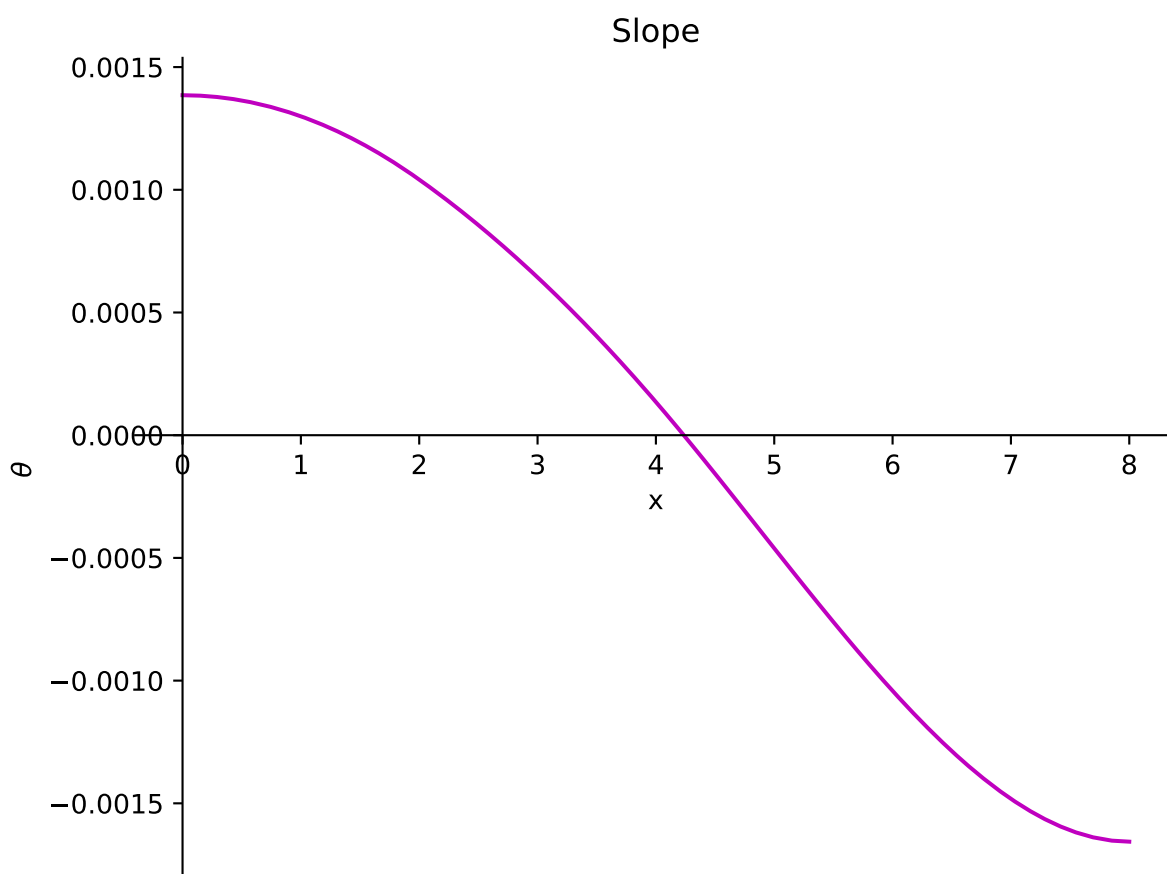
Returns a Set of point(s) with zero bending moment and where bending moment curve of the beam object changes its sign from negative to positive or vice versa.

## Examples

There is 10 meter long overhanging beam. There are two simple supports below the beam. One at the start and another one at a distance of 6 meters from the start. Point loads of magnitude 10KN and 20KN are applied at 2 meters and 4 meters from start respectively. A Uniformly distribute load of magnitude of magnitude 3KN/m is also applied on top starting from 6 meters away from starting point till end. Using the sign convention of upward forces and clockwise moment being positive.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> b = Beam(10, E, I)
>>> b.apply_load(-4, 0, -1)
>>> b.apply_load(-46, 6, -1)
>>> b.apply_load(10, 2, -1)
```

(continues on next page)



(continued from previous page)

```
>>> b.apply_load(20, 4, -1)
>>> b.apply_load(3, 6, 0)
>>> b.point_cflexure()
[10/3]
```

**property reaction\_loads**

Returns the reaction forces in a dictionary.

**remove\_load**(*value, start, order, end=None*)

This method removes a particular load present on the beam object. Returns a ValueError if the load passed as an argument is not present on the beam.

**Parameters**

**value** : Sympifyable

The magnitude of an applied load.

**start** : Sympifyable

The starting point of the applied load. For point moments and point forces this is the location of application.

**order** : Integer

The order of the applied load. - For moments, order= -2 - For point loads, order=-1 - For constant distributed load, order=0 - For ramp loads, order=1 - For parabolic ramp loads, order=2 - ... so on.

**end** : Sympifyable, optional

An optional argument that can be used if the load has an end point within the length of the beam.

**Examples**

There is a beam of length 4 meters. A moment of magnitude 3 Nm is applied in the clockwise direction at the starting point of the beam. A pointload of magnitude 4 N is applied from the top of the beam at 2 meters from the starting point and a parabolic ramp load of magnitude 2 N/m is applied below the beam starting from 2 meters to 3 meters away from the starting point of the beam.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> b = Beam(4, E, I)
>>> b.apply_load(-3, 0, -2)
>>> b.apply_load(4, 2, -1)
>>> b.apply_load(-2, 2, 2, end=3)
>>> b.load
-3*SingularityFunction(x, 0, -2) + 4*SingularityFunction(x, 2, -1) -
↳ 2*SingularityFunction(x, 2, 2) + 2*SingularityFunction(x, 3, 0) +
↳ 4*SingularityFunction(x, 3, 1) + 2*SingularityFunction(x, 3, 2)
>>> b.remove_load(-2, 2, 2, end = 3)
>>> b.load
-3*SingularityFunction(x, 0, -2) + 4*SingularityFunction(x, 2, -1)
```

**property second\_moment**

Second moment of area of the Beam.

**shear\_force()**

Returns a Singularity Function expression which represents the shear force curve of the Beam object.

**Examples**

There is a beam of length 30 meters. A moment of magnitude 120 Nm is applied in the clockwise direction at the end of the beam. A pointload of magnitude 8 N is applied from the top of the beam at the starting point. There are two simple supports below the beam. One at the end and another one at a distance of 10 meters from the start. The deflection is restricted at both the supports.

Using the sign convention of upward forces and clockwise moment being positive.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(30, E, I)
>>> b.apply_load(-8, 0, -1)
>>> b.apply_load(R1, 10, -1)
>>> b.apply_load(R2, 30, -1)
>>> b.apply_load(120, 30, -2)
>>> b.bc_deflection = [(10, 0), (30, 0)]
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.shear_force()
8*SingularityFunction(x, 0, 0) - 6*SingularityFunction(x, 10, 0) -
↪ 120*SingularityFunction(x, 30, -1) - 2*SingularityFunction(x, 30, 0)
```

**shear\_stress()**

Returns an expression representing the Shear Stress curve of the Beam object.

**slope()**

Returns a Singularity Function expression which represents the slope the elastic curve of the Beam object.

**Examples**

There is a beam of length 30 meters. A moment of magnitude 120 Nm is applied in the clockwise direction at the end of the beam. A pointload of magnitude 8 N is applied from the top of the beam at the starting point. There are two simple supports below the beam. One at the end and another one at a distance of 10 meters from the start. The deflection is restricted at both the supports.

Using the sign convention of upward forces and clockwise moment being positive.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
```

(continues on next page)



(continued from previous page)

```

>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(30, E, I)
>>> b.apply_load(-8, 0, -1)
>>> b.apply_load(R1, 10, -1)
>>> b.apply_load(R2, 30, -1)
>>> b.apply_load(120, 30, -2)
>>> b.bc_deflection = [(10, 0), (30, 0)]
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.slope()
(-4*SingularityFunction(x, 0, 2) + 3*SingularityFunction(x, 10, 2)
 + 120*SingularityFunction(x, 30, 1) + SingularityFunction(x, 30,
↪ 2) + 4000/3)/(E*I)

```

**solve\_for\_ild\_moment(*distance, value, \*reactions*)**

Determines the Influence Line Diagram equations for moment at a specified point under the effect of a moving load.

**Parameters**

**distance** : Integer

Distance of the point from the start of the beam for which equations are to be determined

**value** : Integer

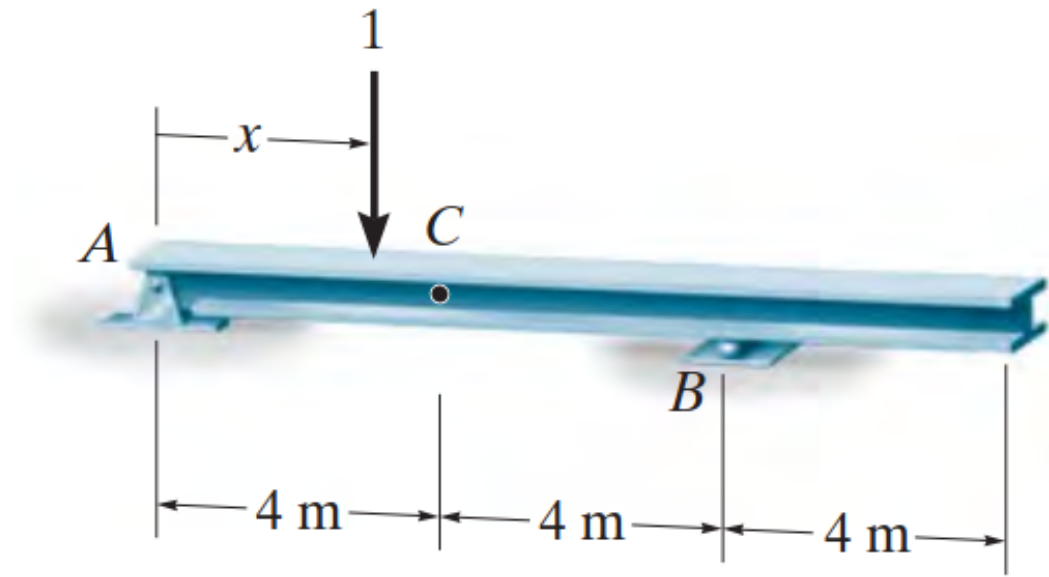
Magnitude of moving load

**reactions** :

The reaction forces applied on the beam.

**Examples**

There is a beam of length 12 meters. There are two simple supports below the beam, one at the starting point and another at a distance of 8 meters. Calculate the I.L.D. equations for Moment at a distance of 4 meters under the effect of a moving load of magnitude 1kN.



Using the sign convention of downwards forces being positive.

```
>>> from sympy import symbols
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> E, I = symbols('E, I')
>>> R_0, R_8 = symbols('R_0, R_8')
>>> b = Beam(12, E, I)
>>> b.apply_support(0, 'roller')
>>> b.apply_support(8, 'roller')
>>> b.solve_for_ild_reactions(1, R_0, R_8)
>>> b.solve_for_ild_moment(4, 1, R_0, R_8)
>>> b.ild_moment
Piecewise((-x/2, x < 4), (x/2 - 4, x > 4))
```

**solve\_for\_ild\_reactions(value, \*reactions)**

Determines the Influence Line Diagram equations for reaction forces under the effect of a moving load.

**Parameters**

**value** : Integer

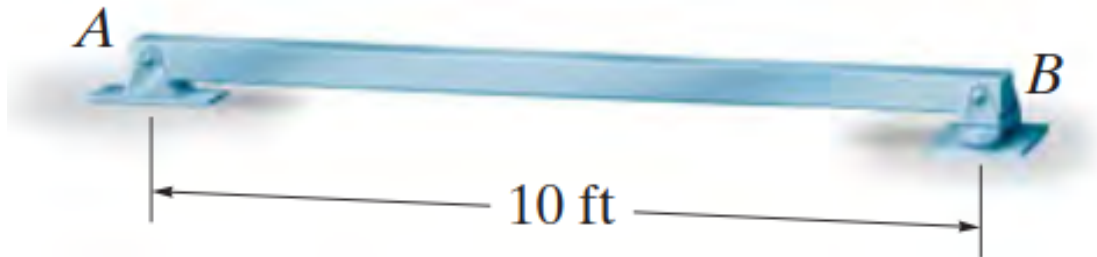
Magnitude of moving load

**reactions** :

The reaction forces applied on the beam.

## Examples

There is a beam of length 10 meters. There are two simple supports below the beam, one at the starting point and another at the ending point of the beam. Calculate the I.L.D. equations for reaction forces under the effect of a moving load of magnitude 1kN.



Using the sign convention of downwards forces being positive.

```
>>> from sympy import symbols
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> E, I = symbols('E, I')
>>> R_0, R_10 = symbols('R_0, R_10')
>>> b = Beam(10, E, I)
>>> b.apply_support(0, 'roller')
>>> b.apply_support(10, 'roller')
>>> b.solve_for_ild_reactions(1, R_0, R_10)
>>> b.ild_reactions
{R_0: x/10 - 1, R_10: -x/10}
```

### `solve_for_ild_shear(distance, value, *reactions)`

Determines the Influence Line Diagram equations for shear at a specified point under the effect of a moving load.

#### Parameters

**distance** : Integer

Distance of the point from the start of the beam for which equations are to be determined

**value** : Integer

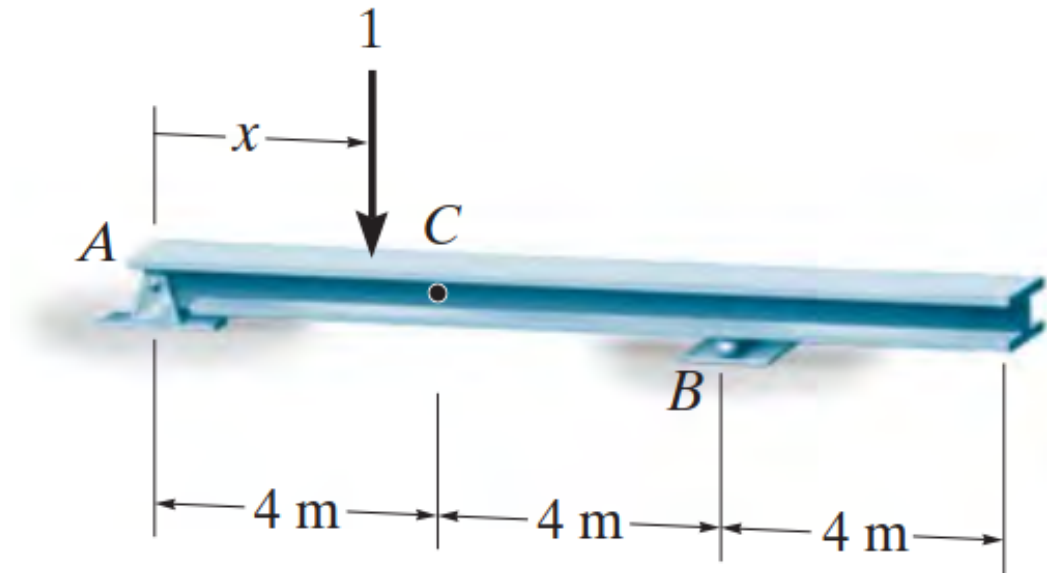
Magnitude of moving load

**reactions** :

The reaction forces applied on the beam.

## Examples

There is a beam of length 12 meters. There are two simple supports below the beam, one at the starting point and another at a distance of 8 meters. Calculate the I.L.D. equations for Shear at a distance of 4 meters under the effect of a moving load of magnitude 1kN.



Using the sign convention of downwards forces being positive.

```
>>> from sympy import symbols
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> E, I = symbols('E, I')
>>> R_0, R_8 = symbols('R_0, R_8')
>>> b = Beam(12, E, I)
>>> b.apply_support(0, 'roller')
>>> b.apply_support(8, 'roller')
>>> b.solve_for_ild_reactions(1, R_0, R_8)
>>> b.solve_for_ild_shear(4, 1, R_0, R_8)
>>> b.ild_shear
Piecewise((x/8, x < 4), (x/8 - 1, x > 4))
```

**solve\_for\_reaction\_loads(\*reactions)**

Solves for the reaction forces.

## Examples

There is a beam of length 30 meters. A moment of magnitude 120 Nm is applied in the clockwise direction at the end of the beam. A pointload of magnitude 8 N is applied from the top of the beam at the starting point. There are two simple supports below the beam. One at the end and another one at a distance of 10 meters from the start. The deflection is restricted at both the supports.

Using the sign convention of upward forces and clockwise moment being positive.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(30, E, I)
>>> b.apply_load(-8, 0, -1)
>>> b.apply_load(R1, 10, -1) # Reaction force at x = 10
>>> b.apply_load(R2, 30, -1) # Reaction force at x = 30
>>> b.apply_load(120, 30, -2)
>>> b.bc_deflection = [(10, 0), (30, 0)]
>>> b.load
R1*SingularityFunction(x, 10, -1) + R2*SingularityFunction(x, 30, -1)
- 8*SingularityFunction(x, 0, -1) + 120*SingularityFunction(x, 30,
↪ -2)
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.reaction_loads
{R1: 6, R2: 2}
>>> b.load
-8*SingularityFunction(x, 0, -1) + 6*SingularityFunction(x, 10, -1)
+ 120*SingularityFunction(x, 30, -2) + 2*SingularityFunction(x,
↪ 30, -1)
```

## property variable

A symbol that can be used as a variable along the length of the beam while representing load distribution, shear force curve, bending moment, slope curve and the deflection curve. By default, it is set to `Symbol('x')`, but this property is mutable.

## Examples

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I, A = symbols('E, I, A')
>>> x, y, z = symbols('x, y, z')
>>> b = Beam(4, E, I)
>>> b.variable
x
>>> b.variable = y
>>> b.variable
y
>>> b = Beam(4, E, I, A, z)
>>> b.variable
z
```

```
class sympy.physics.continuum_mechanics.beam.Beam3D(length, elastic_modulus,
 shear_modulus,
 second_moment, area,
 variable=x)
```

This class handles loads applied in any direction of a 3D space along with unequal values of Second moment along different axes.

---

**Note:** A consistent sign convention must be used while solving a beam bending problem; the results will automatically follow the chosen sign convention. This class assumes that any kind of distributed load/moment is applied through out the span of a beam.

---

## Examples

There is a beam of  $l$  meters long. A constant distributed load of magnitude  $q$  is applied along  $y$ -axis from start till the end of beam. A constant distributed moment of magnitude  $m$  is also applied along  $z$ -axis from start till the end of beam. Beam is fixed at both of its end. So, deflection of the beam at the both ends is restricted.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam3D
>>> from sympy import symbols, simplify, collect, factor
>>> l, E, G, I, A = symbols('l, E, G, I, A')
>>> b = Beam3D(l, E, G, I, A)
>>> x, q, m = symbols('x, q, m')
>>> b.apply_load(q, 0, 0, dir="y")
>>> b.apply_moment_load(m, 0, -1, dir="z")
>>> b.shear_force()
[0, -q*x, 0]
>>> b.bending_moment()
[0, 0, -m*x + q*x**2/2]
>>> b.bc_slope = [(0, [0, 0, 0]), (l, [0, 0, 0])]
>>> b.bc_deflection = [(0, [0, 0, 0]), (l, [0, 0, 0])]
>>> b.solve_slope_deflection()
>>> factor(b.slope())
[0, 0, x*(-l + x)*(-A*G*l**3*q + 2*A*G*l**2*q*x - 12*E*I*l*q
- 72*E*I*m + 24*E*I*q*x)/(12*E*I*(A*G*l**2 + 12*E*I))]
>>> dx, dy, dz = b.deflection()
>>> dy = collect(simplify(dy), x)
>>> dx == dz == 0
True
>>> dy == (x*(12*E*I*l*(A*G*l**2*q - 2*A*G*l*m + 12*E*I*q)
... + x*(A*G*l*(3*l*(A*G*l**2*q - 2*A*G*l*m + 12*E*I*q) + x*(-
↪ 2*A*G*l**2*q + 4*A*G*l*m - 24*E*I*q))
... + A*G*(A*G*l**2 + 12*E*I)*(-2*l**2*q + 6*l*m - 4*m*x + q*x**2)
... - 12*E*I*q*(A*G*l**2 + 12*E*I)))/(24*A*E*G*I*(A*G*l**2 + 12*E*I)))
True
```

## References

[R651]

**apply\_load**(*value*, *start*, *order*, *dir*='y')

This method adds up the force load to a particular beam object.

**Parameters**

**value** : Sympifyable

The magnitude of an applied load.

**dir** : String

Axis along which load is applied.

**order** : Integer

The order of the applied load. - For point loads, order=-1 - For constant distributed load, order=0 - For ramp loads, order=1 - For parabolic ramp loads, order=2 - ... so on.

**apply\_moment\_load**(*value*, *start*, *order*, *dir*='y')

This method adds up the moment loads to a particular beam object.

**Parameters**

**value** : Sympifyable

The magnitude of an applied moment.

**dir** : String

Axis along which moment is applied.

**order** : Integer

The order of the applied load. - For point moments, order=-2 - For constant distributed moment, order=-1 - For ramp moments, order=0 - For parabolic ramp moments, order=1 - ... so on.

**property area**

Cross-sectional area of the Beam.

**axial\_force()**

Returns expression of Axial shear force present inside the Beam object.

**axial\_stress()**

Returns expression of Axial stress present inside the Beam object.

**bending\_moment()**

Returns a list of three expressions which represents the bending moment curve of the Beam object along all three axes.

**property boundary\_conditions**

Returns a dictionary of boundary conditions applied on the beam. The dictionary has two keywords namely slope and deflection. The value of each keyword is a list of tuple, where each tuple contains location and value of a boundary condition in the format (location, value). Further each value is a list corresponding to slope or deflection(s) values along three axes at that location.

## Examples

There is a beam of length 4 meters. The slope at 0 should be 4 along the x-axis and 0 along others. At the other end of beam, deflection along all the three axes should be zero.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam3D
>>> from sympy import symbols
>>> l, E, G, I, A, x = symbols('l, E, G, I, A, x')
>>> b = Beam3D(30, E, G, I, A, x)
>>> b.bc_slope = [(0, (4, 0, 0))]
>>> b.bc_deflection = [(4, [0, 0, 0])]
>>> b.boundary_conditions
{'deflection': [(4, [0, 0, 0])], 'slope': [(0, (4, 0, 0))]}
```

Here the deflection of the beam should be 0 along all the three axes at 4. Similarly, the slope of the beam should be 4 along x-axis and 0 along y and z axis at 0.

### **deflection()**

Returns a three element list representing deflection curve along all the three axes.

### **property load\_vector**

Returns a three element list representing the load vector.

### **max\_bending\_moment()**

Returns point of max bending moment and its corresponding bending moment value along all directions in a Beam object as a list. `solve_for_reaction_loads()` must be called before using this function.

## Examples

There is a beam of length 20 meters. It is supported by rollers at both ends. A linear load having slope equal to 12 is applied along y-axis. A constant distributed load of magnitude 15 N is applied from start till its end along z-axis.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam3D
>>> from sympy import symbols
>>> l, E, G, I, A, x = symbols('l, E, G, I, A, x')
>>> b = Beam3D(20, 40, 21, 100, 25, x)
>>> b.apply_load(15, start=0, order=0, dir="z")
>>> b.apply_load(12*x, start=0, order=0, dir="y")
>>> b.bc_deflection = [(0, [0, 0, 0]), (20, [0, 0, 0])]
>>> R1, R2, R3, R4 = symbols('R1, R2, R3, R4')
>>> b.apply_load(R1, start=0, order=-1, dir="z")
>>> b.apply_load(R2, start=20, order=-1, dir="z")
>>> b.apply_load(R3, start=0, order=-1, dir="y")
>>> b.apply_load(R4, start=20, order=-1, dir="y")
>>> b.solve_for_reaction_loads(R1, R2, R3, R4)
>>> b.max_bending_moment()
[(0, 0), (20, 3000), (20, 16000)]
```

### **max\_bmoment()**

Returns point of max bending moment and its corresponding bending moment value



along all directions in a Beam object as a list. `solve_for_reaction_loads()` must be called before using this function.

### Examples

There is a beam of length 20 meters. It is supported by rollers at both ends. A linear load having slope equal to 12 is applied along y-axis. A constant distributed load of magnitude 15 N is applied from start till its end along z-axis.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam3D
>>> from sympy import symbols
>>> l, E, G, I, A, x = symbols('l, E, G, I, A, x')
>>> b = Beam3D(20, 40, 21, 100, 25, x)
>>> b.apply_load(15, start=0, order=0, dir="z")
>>> b.apply_load(12*x, start=0, order=0, dir="y")
>>> b.bc_deflection = [(0, [0, 0, 0]), (20, [0, 0, 0])]
>>> R1, R2, R3, R4 = symbols('R1, R2, R3, R4')
>>> b.apply_load(R1, start=0, order=-1, dir="z")
>>> b.apply_load(R2, start=20, order=-1, dir="z")
>>> b.apply_load(R3, start=0, order=-1, dir="y")
>>> b.apply_load(R4, start=20, order=-1, dir="y")
>>> b.solve_for_reaction_loads(R1, R2, R3, R4)
>>> b.max_bending_moment()
[(0, 0), (20, 3000), (20, 16000)]
```

#### `max_deflection()`

Returns point of max deflection and its corresponding deflection value along all directions in a Beam object as a list. `solve_for_reaction_loads()` and `solve_slope_deflection()` must be called before using this function.

### Examples

There is a beam of length 20 meters. It is supported by rollers at both ends. A linear load having slope equal to 12 is applied along y-axis. A constant distributed load of magnitude 15 N is applied from start till its end along z-axis.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam3D
>>> from sympy import symbols
>>> l, E, G, I, A, x = symbols('l, E, G, I, A, x')
>>> b = Beam3D(20, 40, 21, 100, 25, x)
>>> b.apply_load(15, start=0, order=0, dir="z")
>>> b.apply_load(12*x, start=0, order=0, dir="y")
>>> b.bc_deflection = [(0, [0, 0, 0]), (20, [0, 0, 0])]
>>> R1, R2, R3, R4 = symbols('R1, R2, R3, R4')
>>> b.apply_load(R1, start=0, order=-1, dir="z")
>>> b.apply_load(R2, start=20, order=-1, dir="z")
>>> b.apply_load(R3, start=0, order=-1, dir="y")
>>> b.apply_load(R4, start=20, order=-1, dir="y")
>>> b.solve_for_reaction_loads(R1, R2, R3, R4)
>>> b.solve_slope_deflection()
>>> b.max_deflection()
```

(continues on next page)

(continued from previous page)

```
[(0, 0), (10, 495/14), (-10 + 10*sqrt(10793)/43, (10 - 10*sqrt(10793)/
↳ 43)**3/160 - 20/7 + (10 - 10*sqrt(10793)/43)**4/6400 +
↳ 20*sqrt(10793)/301 + 27*(10 - 10*sqrt(10793)/43)**2/560)]
```

**max\_shear\_force()**

Returns point of max shear force and its corresponding shear value along all directions in a Beam object as a list. `solve_for_reaction_loads()` must be called before using this function.

**Examples**

There is a beam of length 20 meters. It is supported by rollers at both ends. A linear load having slope equal to 12 is applied along y-axis. A constant distributed load of magnitude 15 N is applied from start till its end along z-axis.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam3D
>>> from sympy import symbols
>>> l, E, G, I, A, x = symbols('l, E, G, I, A, x')
>>> b = Beam3D(20, 40, 21, 100, 25, x)
>>> b.apply_load(15, start=0, order=0, dir="z")
>>> b.apply_load(12*x, start=0, order=0, dir="y")
>>> b.bc_deflection = [(0, [0, 0, 0]), (20, [0, 0, 0])]
>>> R1, R2, R3, R4 = symbols('R1, R2, R3, R4')
>>> b.apply_load(R1, start=0, order=-1, dir="z")
>>> b.apply_load(R2, start=20, order=-1, dir="z")
>>> b.apply_load(R3, start=0, order=-1, dir="y")
>>> b.apply_load(R4, start=20, order=-1, dir="y")
>>> b.solve_for_reaction_loads(R1, R2, R3, R4)
>>> b.max_shear_force()
[(0, 0), (20, 2400), (20, 300)]
```

**property moment\_load\_vector**

Returns a three element list representing moment loads on Beam.

**plot\_bending\_moment(*dir*='all', *subs*=None)**

Returns a plot for bending moment along all three directions present in the Beam object.

**Parameters**

**dir** : string (default

Direction along which bending moment plot is required. If no direction is specified, all plots are displayed.

**subs** : dictionary

Python dictionary containing Symbols as key and their corresponding values.

## Examples

There is a beam of length 20 meters. It is supported by rollers at both ends. A linear load having slope equal to 12 is applied along y-axis. A constant distributed load of magnitude 15 N is applied from start till its end along z-axis.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam3D
>>> from sympy import symbols
>>> l, E, G, I, A, x = symbols('l, E, G, I, A, x')
>>> b = Beam3D(20, E, G, I, A, x)
>>> b.apply_load(15, start=0, order=0, dir="z")
>>> b.apply_load(12*x, start=0, order=0, dir="y")
>>> b.bc_deflection = [(0, [0, 0, 0]), (20, [0, 0, 0])]
>>> R1, R2, R3, R4 = symbols('R1, R2, R3, R4')
>>> b.apply_load(R1, start=0, order=-1, dir="z")
>>> b.apply_load(R2, start=20, order=-1, dir="z")
>>> b.apply_load(R3, start=0, order=-1, dir="y")
>>> b.apply_load(R4, start=20, order=-1, dir="y")
>>> b.solve_for_reaction_loads(R1, R2, R3, R4)
>>> b.plot_bending_moment()
PlotGrid object containing:
Plot[0]:Plot object containing:
[0]: cartesian line: 0 for x over (0.0, 20.0)
Plot[1]:Plot object containing:
[0]: cartesian line: -15*x**2/2 for x over (0.0, 20.0)
Plot[2]:Plot object containing:
[0]: cartesian line: 2*x**3 for x over (0.0, 20.0)
```

**plot\_deflection**(*dir*='all', *subs*=None)

Returns a plot for Deflection along all three directions present in the Beam object.

### Parameters

**dir** : string (default

Direction along which deflection plot is required. If no direction is specified, all plots are displayed.

**subs** : dictionary

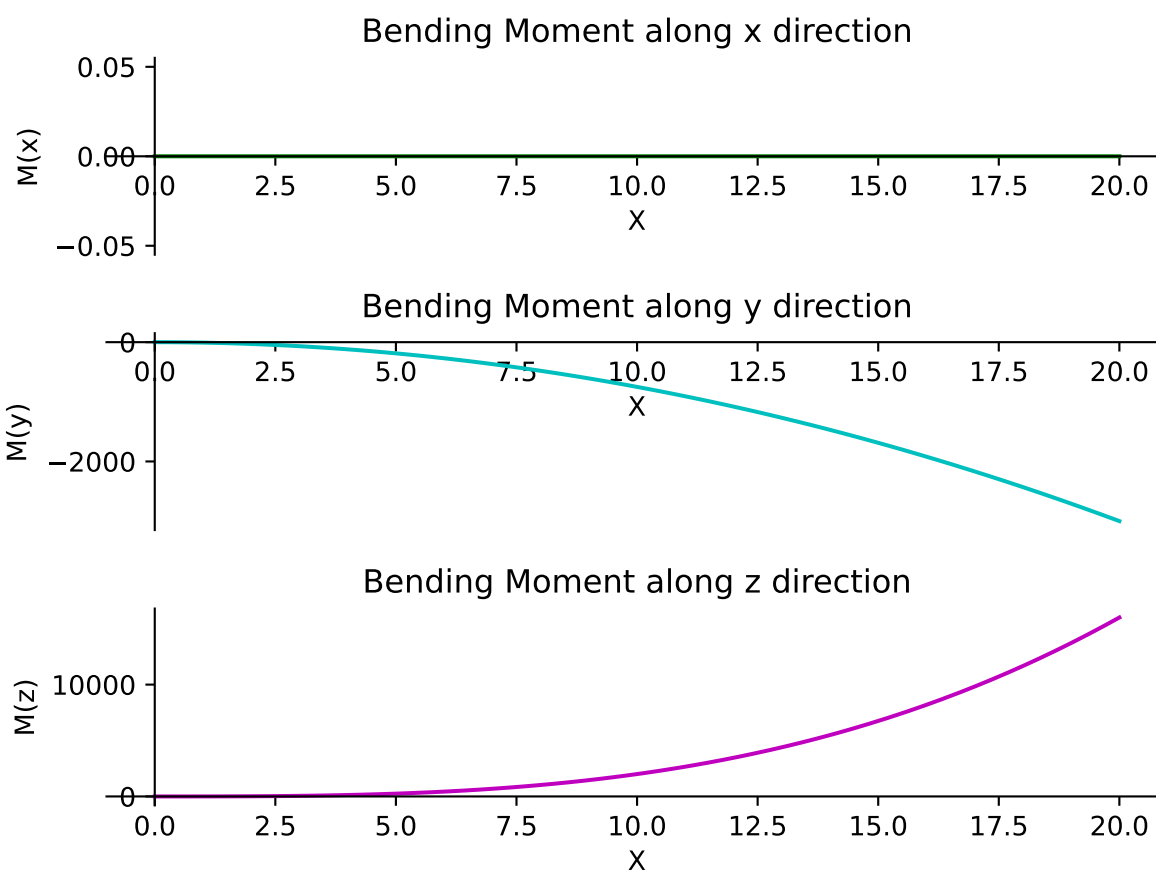
Python dictionary containing Symbols as keys and their corresponding values.

## Examples

There is a beam of length 20 meters. It is supported by rollers at both ends. A linear load having slope equal to 12 is applied along y-axis. A constant distributed load of magnitude 15 N is applied from start till its end along z-axis.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam3D
>>> from sympy import symbols
>>> l, E, G, I, A, x = symbols('l, E, G, I, A, x')
>>> b = Beam3D(20, 40, 21, 100, 25, x)
>>> b.apply_load(15, start=0, order=0, dir="z")
>>> b.apply_load(12*x, start=0, order=0, dir="y")
```

(continues on next page)

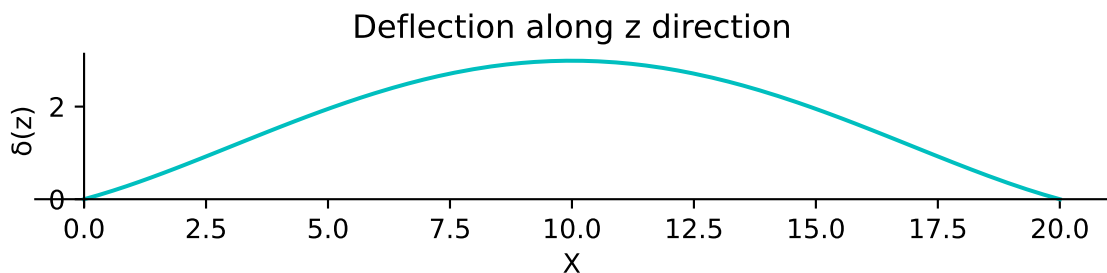
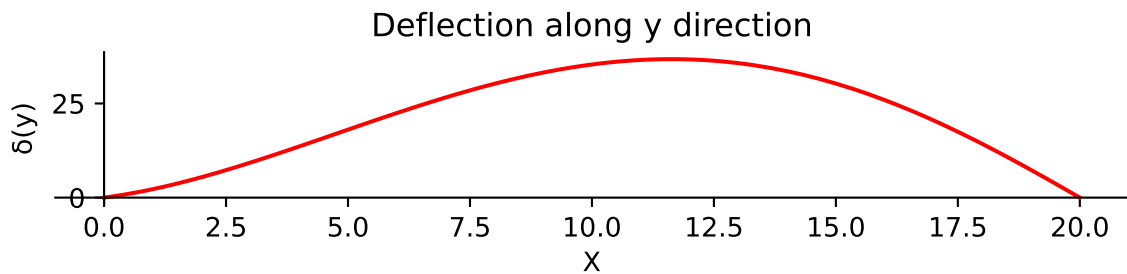
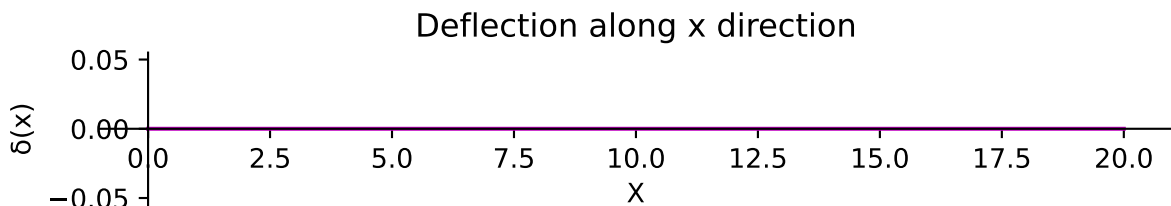


(continued from previous page)

```

>>> b.bc_deflection = [(0, [0, 0, 0]), (20, [0, 0, 0])]
>>> R1, R2, R3, R4 = symbols('R1, R2, R3, R4')
>>> b.apply_load(R1, start=0, order=-1, dir="z")
>>> b.apply_load(R2, start=20, order=-1, dir="z")
>>> b.apply_load(R3, start=0, order=-1, dir="y")
>>> b.apply_load(R4, start=20, order=-1, dir="y")
>>> b.solve_for_reaction_loads(R1, R2, R3, R4)
>>> b.solve_slope_deflection()
>>> b.plot_deflection()
PlotGrid object containing:
Plot[0]:Plot object containing:
[0]: cartesian line: 0 for x over (0.0, 20.0)
Plot[1]:Plot object containing:
[0]: cartesian line: $x^5/40000 - 4013x^3/90300 + 26x^2/43 + 1520x/903$ for x over (0.0, 20.0)
Plot[2]:Plot object containing:
[0]: cartesian line: $x^4/6400 - x^3/160 + 27x^2/560 + 2x/7$ for x over (0.0, 20.0)

```



**plot\_loading\_results**(*dir*='x', *subs*=None)

Returns a subplot of Shear Force, Bending Moment, Slope and Deflection of the Beam object along the direction specified.

**Parameters****dir** : string (default

Direction along which plots are required. If no direction is specified, plots along x-axis are displayed.

**subs** : dictionary

Python dictionary containing Symbols as key and their corresponding values.

**Examples**

There is a beam of length 20 meters. It is supported by rollers at both ends. A linear load having slope equal to 12 is applied along y-axis. A constant distributed load of magnitude 15 N is applied from start till its end along z-axis.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam3D
>>> from sympy import symbols
>>> l, E, G, I, A, x = symbols('l, E, G, I, A, x')
>>> b = Beam3D(20, E, G, I, A, x)
>>> subs = {E:40, G:21, I:100, A:25}
>>> b.apply_load(15, start=0, order=0, dir="z")
>>> b.apply_load(12*x, start=0, order=0, dir="y")
>>> b.bc_deflection = [(0, [0, 0, 0]), (20, [0, 0, 0])]
>>> R1, R2, R3, R4 = symbols('R1, R2, R3, R4')
>>> b.apply_load(R1, start=0, order=-1, dir="z")
>>> b.apply_load(R2, start=20, order=-1, dir="z")
>>> b.apply_load(R3, start=0, order=-1, dir="y")
>>> b.apply_load(R4, start=20, order=-1, dir="y")
>>> b.solve_for_reaction_loads(R1, R2, R3, R4)
>>> b.solve_slope_deflection()
>>> b.plot_loading_results('y', subs)
PlotGrid object containing:
Plot[0]:Plot object containing:
[0]: cartesian line: -6*x**2 for x over (0.0, 20.0)
Plot[1]:Plot object containing:
[0]: cartesian line: -15*x**2/2 for x over (0.0, 20.0)
Plot[2]:Plot object containing:
[0]: cartesian line: -x**3/1600 + 3*x**2/160 - x/8 for x over (0.0, 20.0)
Plot[3]:Plot object containing:
[0]: cartesian line: x**5/40000 - 4013*x**3/90300 + 26*x**2/43 + 1520*x/903 for x over (0.0, 20.0)
```

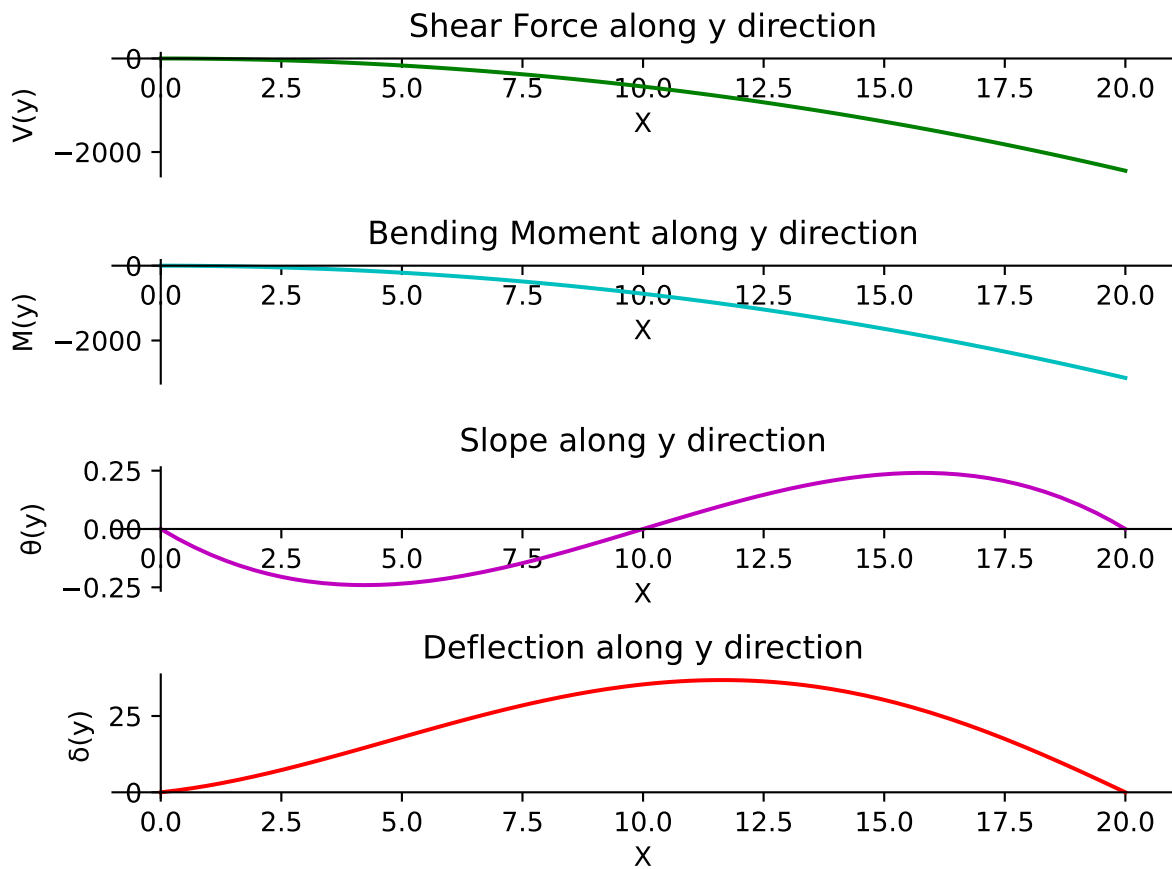
**plot\_shear\_force**(*dir*='all', *subs*=None)

Returns a plot for Shear force along all three directions present in the Beam object.

**Parameters****dir** : string (default

Direction along which shear force plot is required. If no direction is specified, all plots are displayed.

**subs** : dictionary



Python dictionary containing Symbols as key and their corresponding values.

### Examples

There is a beam of length 20 meters. It is supported by rollers at its end. A linear load having slope equal to 12 is applied along y-axis. A constant distributed load of magnitude 15 N is applied from start till its end along z-axis.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam3D
>>> from sympy import symbols
>>> l, E, G, I, A, x = symbols('l, E, G, I, A, x')
>>> b = Beam3D(20, E, G, I, A, x)
>>> b.apply_load(15, start=0, order=0, dir="z")
>>> b.apply_load(12*x, start=0, order=0, dir="y")
>>> b.bc_deflection = [(0, [0, 0, 0]), (20, [0, 0, 0])]
>>> R1, R2, R3, R4 = symbols('R1, R2, R3, R4')
>>> b.apply_load(R1, start=0, order=-1, dir="z")
>>> b.apply_load(R2, start=20, order=-1, dir="z")
>>> b.apply_load(R3, start=0, order=-1, dir="y")
>>> b.apply_load(R4, start=20, order=-1, dir="y")
>>> b.solve_for_reaction_loads(R1, R2, R3, R4)
>>> b.plot_shear_force()
PlotGrid object containing:
Plot[0]:Plot object containing:
[0]: cartesian line: 0 for x over (0.0, 20.0)
Plot[1]:Plot object containing:
[0]: cartesian line: -6*x**2 for x over (0.0, 20.0)
Plot[2]:Plot object containing:
[0]: cartesian line: -15*x for x over (0.0, 20.0)
```

**plot\_shear\_stress**(*dir*='all', *subs*=None)

Returns a plot for Shear Stress along all three directions present in the Beam object.

#### Parameters

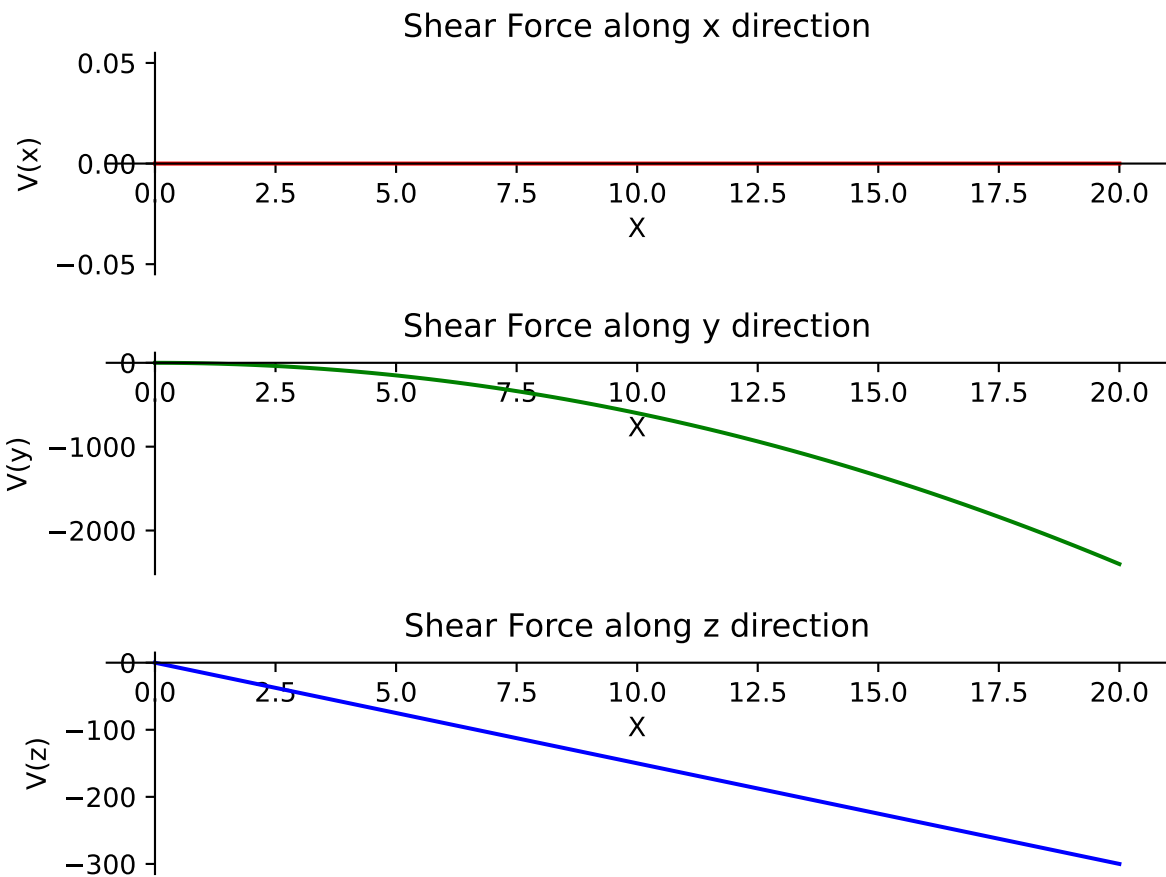
**dir** : string (default

Direction along which shear stress plot is required. If no direction is specified, all plots are displayed.

**subs** : dictionary

Python dictionary containing Symbols as key and their corresponding values.





## Examples

There is a beam of length 20 meters and area of cross section 2 square meters. It is supported by rollers at its end. A linear load having slope equal to 12 is applied along y-axis. A constant distributed load of magnitude 15 N is applied from start till its end along z-axis.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam3D
>>> from sympy import symbols
>>> l, E, G, I, A, x = symbols('l, E, G, I, A, x')
>>> b = Beam3D(20, E, G, I, 2, x)
>>> b.apply_load(15, start=0, order=0, dir="z")
>>> b.apply_load(12*x, start=0, order=0, dir="y")
>>> b.bc_deflection = [(0, [0, 0, 0]), (20, [0, 0, 0])]
>>> R1, R2, R3, R4 = symbols('R1, R2, R3, R4')
>>> b.apply_load(R1, start=0, order=-1, dir="z")
>>> b.apply_load(R2, start=20, order=-1, dir="z")
>>> b.apply_load(R3, start=0, order=-1, dir="y")
>>> b.apply_load(R4, start=20, order=-1, dir="y")
>>> b.solve_for_reaction_loads(R1, R2, R3, R4)
>>> b.plot_shear_stress()
PlotGrid object containing:
Plot[0]:Plot object containing:
[0]: cartesian line: 0 for x over (0.0, 20.0)
Plot[1]:Plot object containing:
[0]: cartesian line: -3*x**2 for x over (0.0, 20.0)
Plot[2]:Plot object containing:
[0]: cartesian line: -15*x/2 for x over (0.0, 20.0)
```

**plot\_slope**(*dir*='all', *subs*=None)

Returns a plot for Slope along all three directions present in the Beam object.

### Parameters

**dir** : string (default

Direction along which Slope plot is required. If no direction is specified, all plots are displayed.

**subs** : dictionary

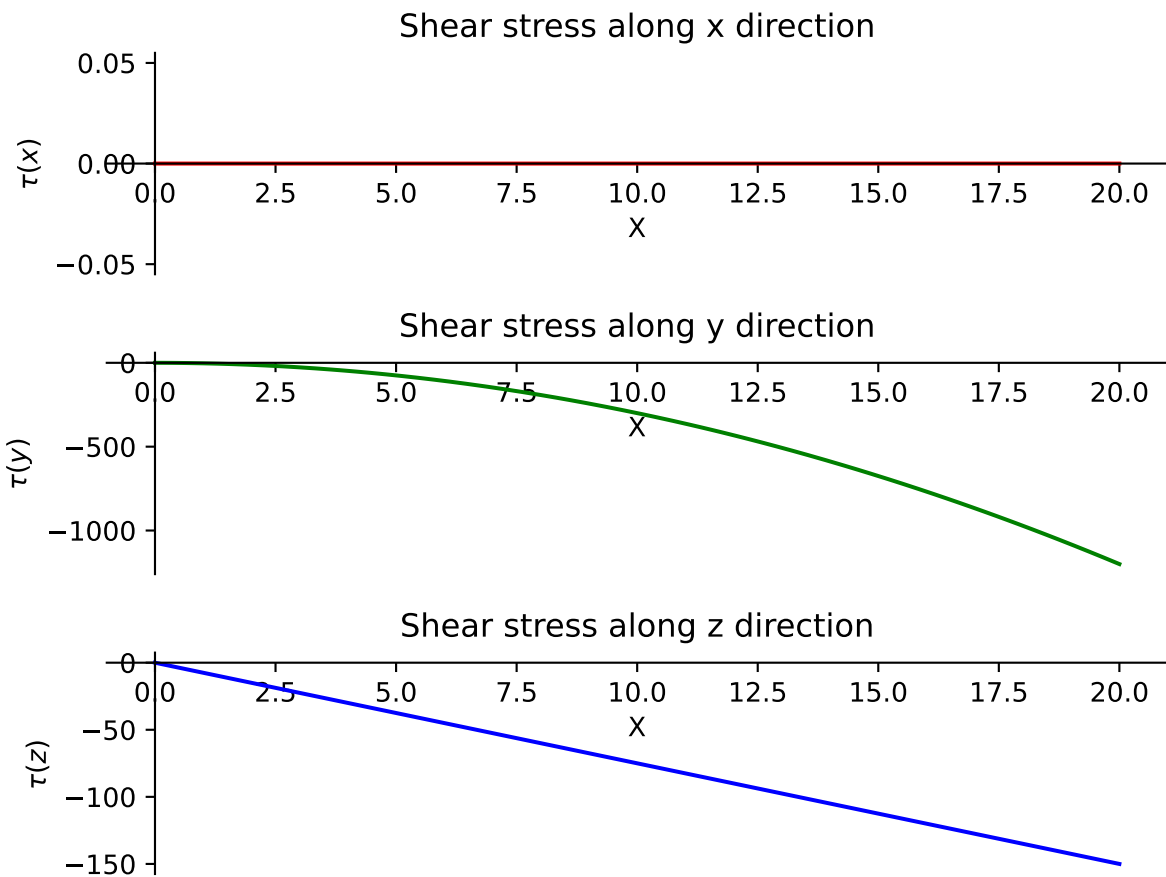
Python dictionary containing Symbols as keys and their corresponding values.

## Examples

There is a beam of length 20 meters. It is supported by rollers at its end. A linear load having slope equal to 12 is applied along y-axis. A constant distributed load of magnitude 15 N is applied from start till its end along z-axis.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam3D
>>> from sympy import symbols
>>> l, E, G, I, A, x = symbols('l, E, G, I, A, x')
>>> b = Beam3D(20, 40, 21, 100, 25, x)
>>> b.apply_load(15, start=0, order=0, dir="z")
```

(continues on next page)

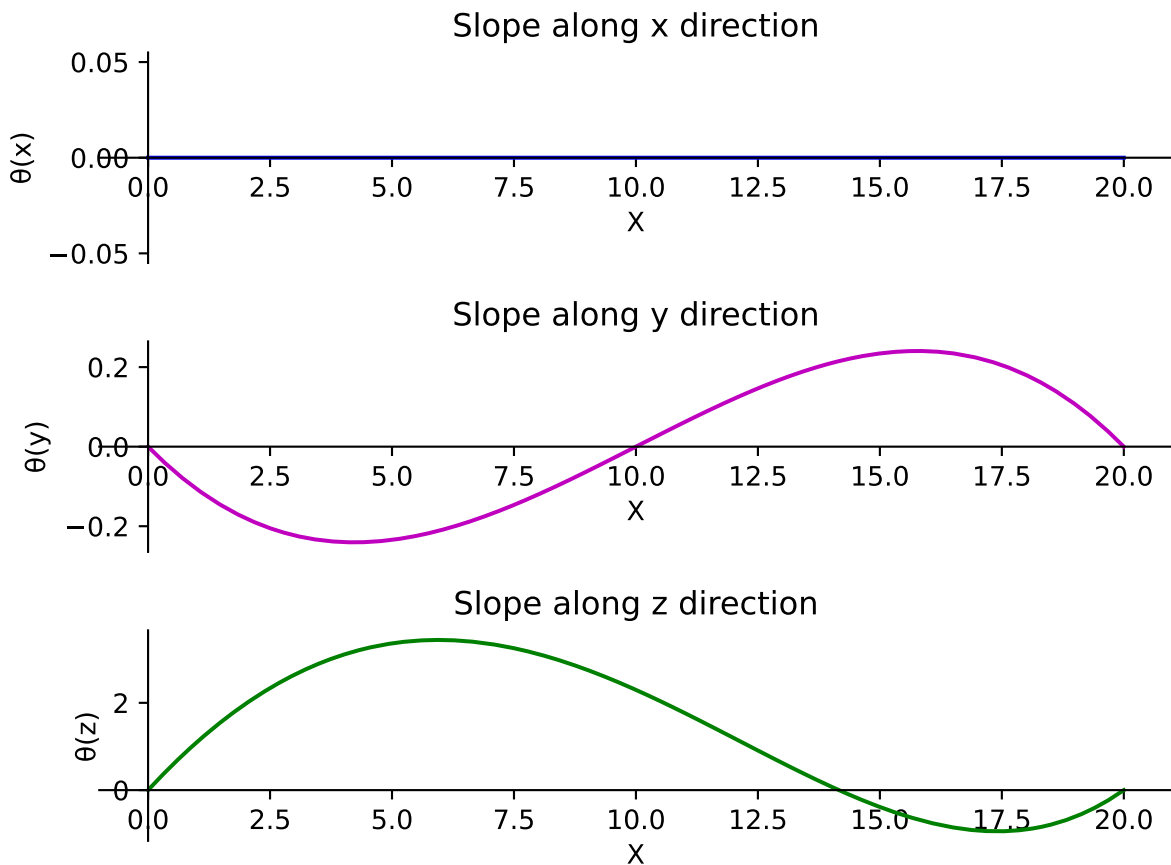


(continued from previous page)

```

>>> b.apply_load(12*x, start=0, order=0, dir="y")
>>> b.bc_deflection = [(0, [0, 0, 0]), (20, [0, 0, 0])]
>>> R1, R2, R3, R4 = symbols('R1, R2, R3, R4')
>>> b.apply_load(R1, start=0, order=-1, dir="z")
>>> b.apply_load(R2, start=20, order=-1, dir="z")
>>> b.apply_load(R3, start=0, order=-1, dir="y")
>>> b.apply_load(R4, start=20, order=-1, dir="y")
>>> b.solve_for_reaction_loads(R1, R2, R3, R4)
>>> b.solve_slope_deflection()
>>> b.plot_slope()
PlotGrid object containing:
Plot[0]:Plot object containing:
[0]: cartesian line: 0 for x over (0.0, 20.0)
Plot[1]:Plot object containing:
[0]: cartesian line: -x**3/1600 + 3*x**2/160 - x/8 for x over (0.0, 20.0)
Plot[2]:Plot object containing:
[0]: cartesian line: x**4/8000 - 19*x**2/172 + 52*x/43 for x over (0.0, 20.0)

```

**polar\_moment()**

Returns the polar moment of area of the beam about the X axis with respect to the

centroid.

## Examples

```
>>> from sympy.physics.continuum_mechanics.beam import Beam3D
>>> from sympy import symbols
>>> l, E, G, I, A = symbols('l, E, G, I, A')
>>> b = Beam3D(l, E, G, I, A)
>>> b.polar_moment()
2*I
>>> I1 = [9, 15]
>>> b = Beam3D(l, E, G, I1, A)
>>> b.polar_moment()
24
```

### property second\_moment

Second moment of area of the Beam.

### shear\_force()

Returns a list of three expressions which represents the shear force curve of the Beam object along all three axes.

### property shear\_modulus

Young's Modulus of the Beam.

### shear\_stress()

Returns a list of three expressions which represents the shear stress curve of the Beam object along all three axes.

### slope()

Returns a three element list representing slope of deflection curve along all the three axes.

### solve\_for\_reaction\_loads(\*reaction)

Solves for the reaction forces.

## Examples

There is a beam of length 30 meters. It is supported by rollers at both ends. A constant distributed load of magnitude 8 N is applied from start till its end along y-axis. Another linear load having slope equal to 9 is applied along z-axis.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam3D
>>> from sympy import symbols
>>> l, E, G, I, A, x = symbols('l, E, G, I, A, x')
>>> b = Beam3D(30, E, G, I, A, x)
>>> b.apply_load(8, start=0, order=0, dir="y")
>>> b.apply_load(9*x, start=0, order=0, dir="z")
>>> b.bc_deflection = [(0, [0, 0, 0]), (30, [0, 0, 0])]
>>> R1, R2, R3, R4 = symbols('R1, R2, R3, R4')
>>> b.apply_load(R1, start=0, order=-1, dir="y")
>>> b.apply_load(R2, start=30, order=-1, dir="y")
```

(continues on next page)

(continued from previous page)

```
>>> b.apply_load(R3, start=0, order=-1, dir="z")
>>> b.apply_load(R4, start=30, order=-1, dir="z")
>>> b.solve_for_reaction_loads(R1, R2, R3, R4)
>>> b.reaction_loads
{R1: -120, R2: -120, R3: -1350, R4: -2700}
```

**torsional\_moment()**

Returns expression of Torsional moment present inside the Beam object.

## Solving Beam Bending Problems using Singularity Functions

To make this document easier to read, enable pretty printing:

```
>>> from sympy import *
>>> x, y, z = symbols('x y z')
>>> init_printing(use_unicode=True, wrap_line=False)
```

## Beam

A planar beam is a structural element that is capable of withstanding load through resistance to internal shear and bending. Beams are characterized by their length, constraints, cross-sectional second moment of area, and elastic modulus. In SymPy, 2D beam objects are constructed by specifying the following properties:

- Length
- Elastic Modulus
- Second Moment of Area
- Variable : A symbol representing the location along the beam's length. By default, this is set to `Symbol(x)`.
- **Boundary Conditions**
  - `bc_slope` : Boundary conditions for slope.
  - `bc_deflection` : Boundary conditions for deflection.
- Load Distribution

Once the above are specified, the following methods are used to compute useful information about the loaded beam:

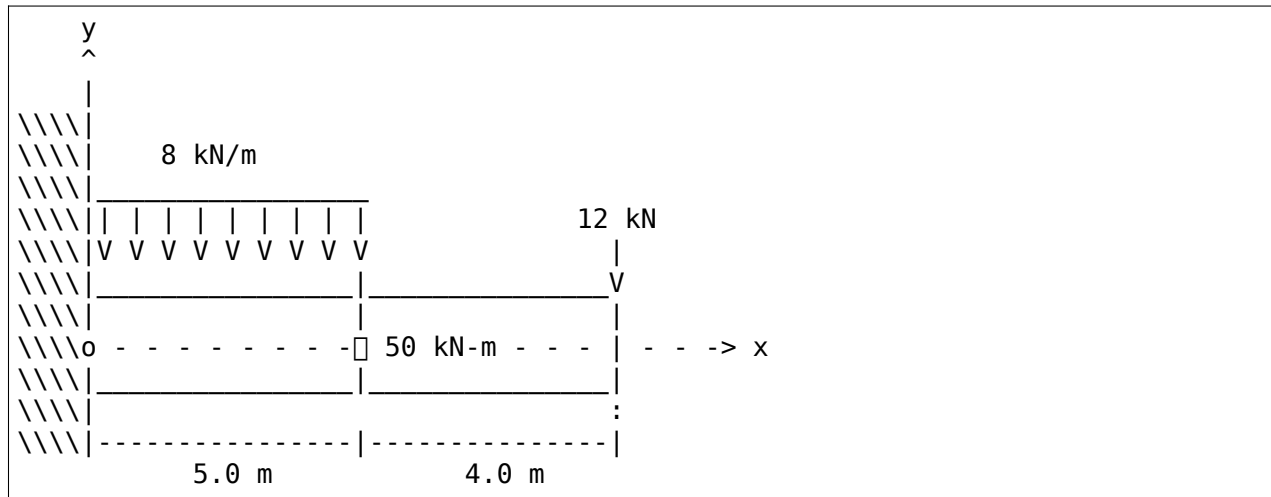
- `solve_for_reaction_loads()`
- `shear_force()`
- `bending_moment()`
- `slope()`

## Examples

Below are examples of a variety two dimensional beam bending problems.

### Example 1

A cantilever beam 9 meters in length has a distributed constant load of 8 kN/m applied downward from the fixed end over a 5 meter distance. A counterclockwise moment of 50 kN-m is applied 5 meters from the fixed end. Lastly, a downward point load of 12 kN is applied at the free end of the beam.



**Note:** The user is free to choose their own sign convention. In this case the downward forces and counterclockwise bending moment being positive.

The beam must be initialized with the length, modulus of elasticity, and the second moment of area. These quantities can be symbols or numbers.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> E, I = symbols('E, I')
>>> b = Beam(9, E, I)
```

The three loads are applied to the beam using the `apply_load()` method. This method supports point forces, point moments, and polynomial distributed loads of any order, i.e.  $c$ ,  $cx$ ,  $cx^2$ ,  $cx^3$ , ...

The 12 kN point load is in the negative direction, at the location of 9 meters, and the polynomial order is specified as -1:

```
>>> b.apply_load(12, 9, -1)
```

The load attribute can then be used to access the loading function in singularity function form:

```
>>> b.load
-1
12.<x - 9>
```

Similarly, the positive moment can be applied with a polynomial order -2:

```
>>> b.apply_load(50, 5, -2)
```

The distributed load is of order 0 and spans  $x=0$  to  $x=5$ :

```
>>> b.apply_load(8, 0, 0, end=5)
```

The fixed end imposes two boundary conditions: 1) no vertical deflection and 2) no rotation. These are specified by appending tuples of  $x$  values and the corresponding deflection or slope values:

```
>>> b.bc_deflection.append((0, 0))
>>> b.bc_slope.append((0, 0))
```

These boundary conditions introduce an unknown reaction force and moment which need to be applied to the beam to maintain static equilibrium:

```
>>> R, M = symbols('R, M')
>>> b.apply_load(R, 0, -1)
>>> b.apply_load(M, 0, -2)
>>> b.load
M·<x>-2 + R·<x>-1 + 8·<x>0 + 50·<x - 5>-2 - 8·<x - 5>0 + 12·<x - 9>-1
```

These two variables can be solved for in terms of the applied loads and the final loading can be displayed:

```
>>> b.solve_for_reaction_loads(R, M)
>>> b.reaction_loads
{M: 158, R: -52}
>>> b.load
158·<x>-2 - 52·<x>-1 + 8·<x>0 + 50·<x - 5>-2 - 8·<x - 5>0 + 12·<x - 9>-1
```

At this point, the beam is fully defined and the internal shear and bending moments are calculated:

```
>>> b.shear_force()
- 158·<x>-1 + 52·<x>0 - 8·<x>1 - 50·<x - 5>-1 + 8·<x - 5>1 - 12·<x - 9>0
```

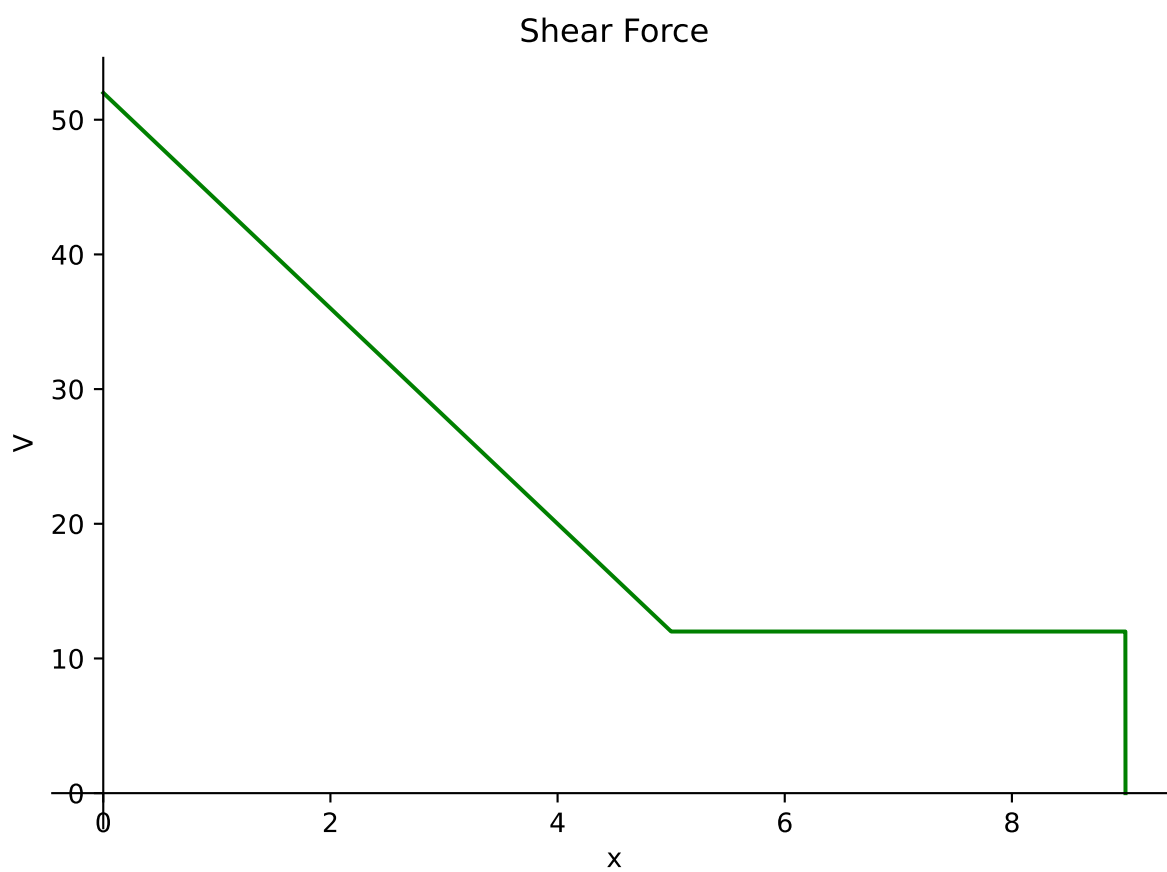
```
>>> b.bending_moment()
- 158·<x>0 + 52·<x>1 - 4·<x>2 - 50·<x - 5>0 + 4·<x - 5>2 - 12·<x - 9>1
```

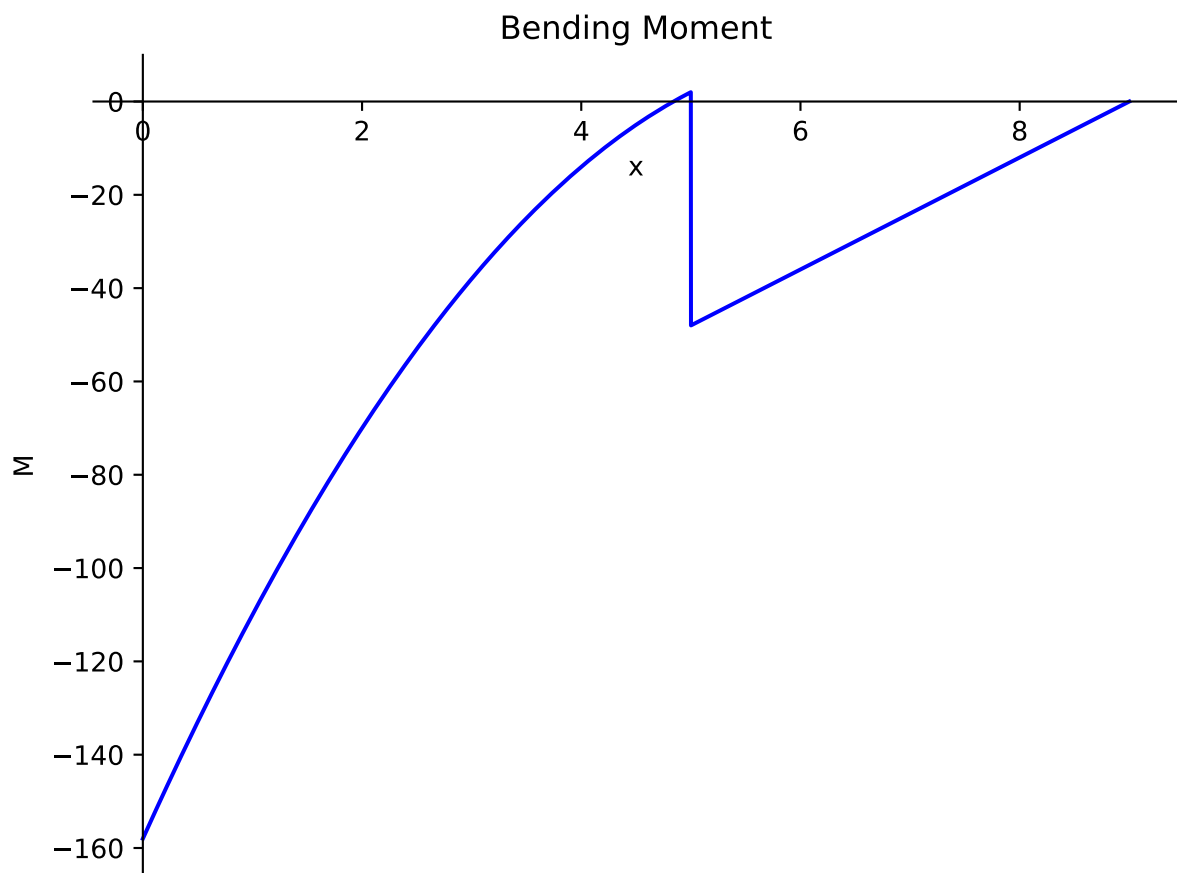
These can be visualized by calling the respective plot methods:

```
>>> b.plot_shear_force()
>>> b.plot_bending_moment()
```

The beam will deform under load and the slope and deflection can be determined with:





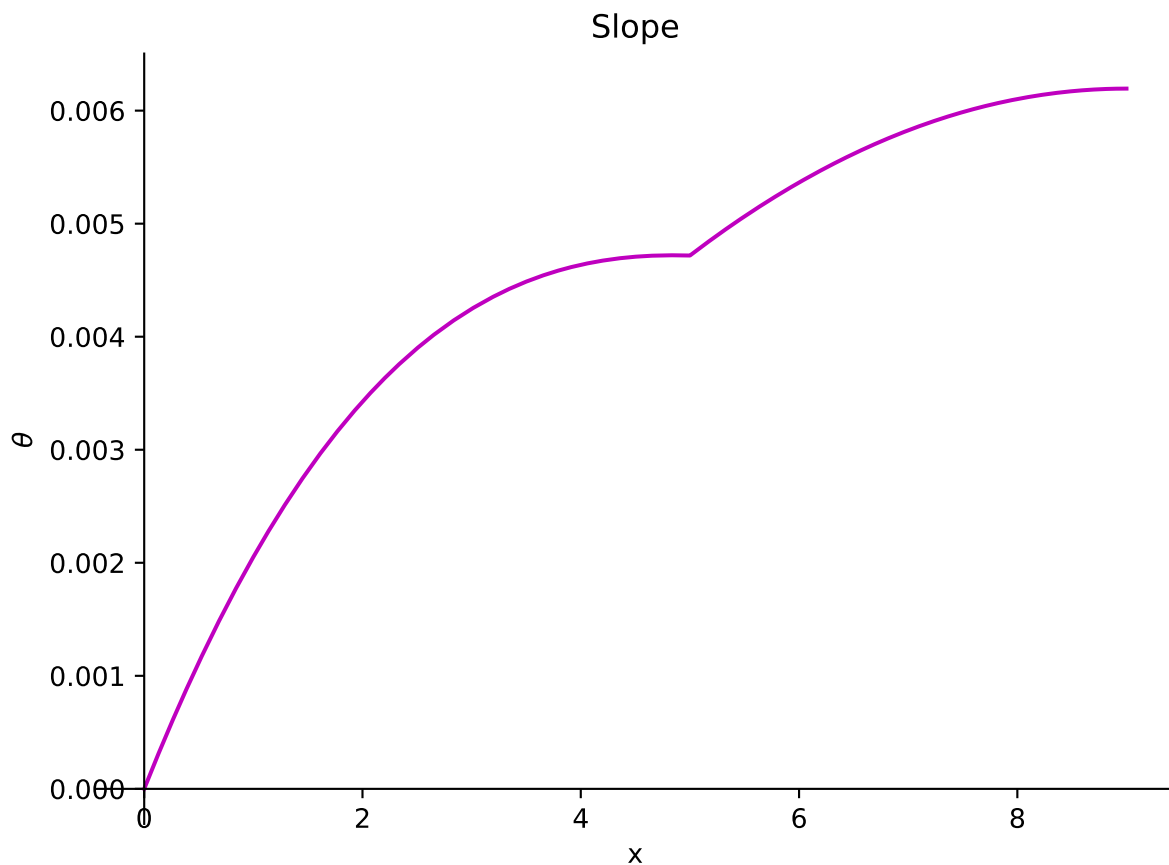


```
>>> b.slope()
- \left(- \frac{158 \cdot \langle x \rangle^1 + 26 \cdot \langle x \rangle^2 - \frac{4 \cdot \langle x \rangle^3}{3} - 50 \cdot \langle x - 5 \rangle^1 + \frac{4 \cdot \langle x - 5 \rangle^3}{3} - 6 \cdot \langle x - 9 \rangle^2}{E \cdot I} \right)
```

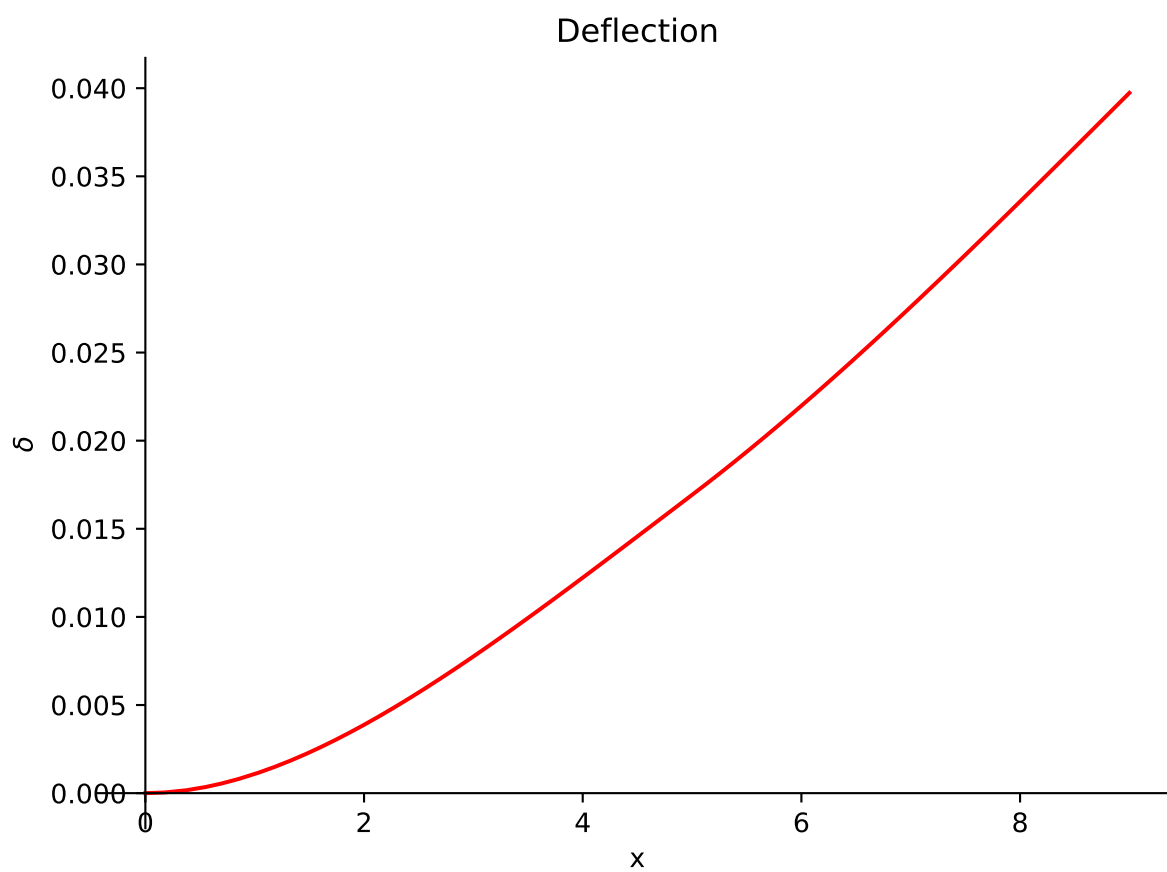
```
>>> b.deflection()
- \left(- \frac{79 \cdot \langle x \rangle^2 + \frac{26 \cdot \langle x \rangle^3}{3} - \frac{\langle x \rangle^4}{3} - 25 \cdot \langle x - 5 \rangle^2 + \frac{\langle x - 5 \rangle^4}{3} - 2 \cdot \langle x - 9 \rangle^3}{E \cdot I} \right)
```

The slope and deflection of the beam can be plotted so long as numbers are provided for the modulus and second moment:

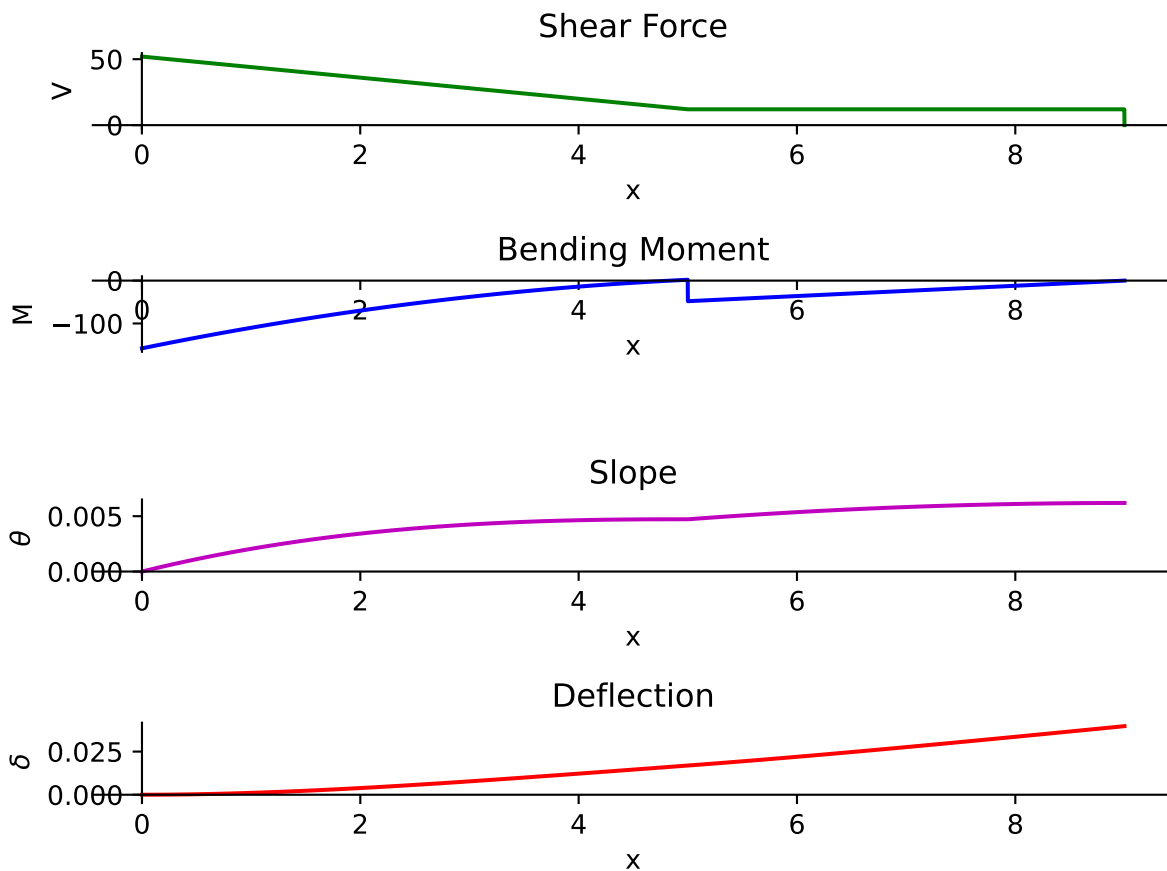
```
>>> b.plot_slope(subs={E: 20E9, I: 3.25E-6})
>>> b.plot_deflection(subs={E: 20E9, I: 3.25E-6})
```



All of the plots can be shown in one figure with:

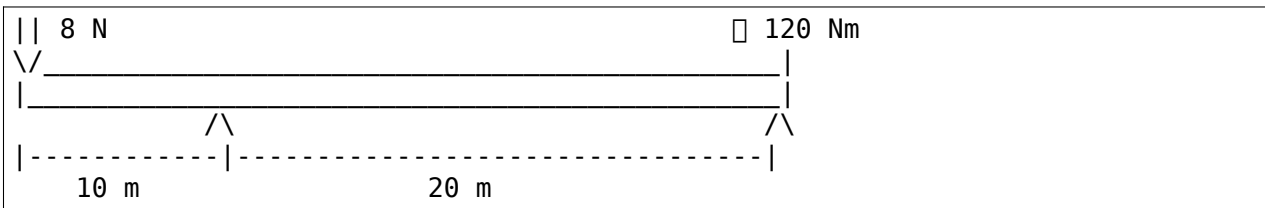


```
>>> b.plot_loading_results(subs={E: 20E9, I: 3.25E-6})
```



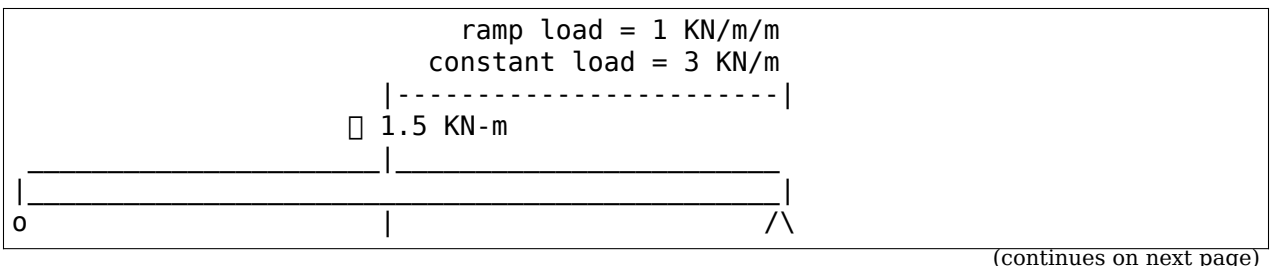
### Example 2

There is a beam of length 30 meters. A moment of magnitude 120 Nm is applied in the counter-clockwise direction at the end of the beam. A point load of magnitude 8 N is applied from the top of the beam at the starting point. There are two simple supports below the beam. One at the end and another one at a distance of 10 meters from the start. The deflection is restricted at both the supports.

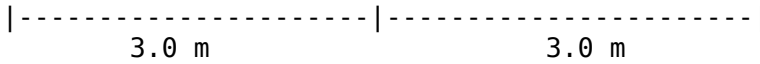


**Note:** Using the sign convention of downward forces and counterclockwise moment being positive.

A beam of length 6 meters is having a roller support at the start and a hinged support at the end. A counterclockwise moment of 1.5 kN-m is applied at the mid of the beam. A constant distributed load of 3 kN/m and a ramp load of 1 kN/m/m is applied from the mid till the end of the beam.



(continued from previous page)



**Note:** Using the sign convention of downward forces and counterclockwise moment being positive.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols, plot, S
>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(6, E, I)
>>> b.apply_load(R1, 0, -1)
>>> b.apply_load(-S(3)/2, 3, -2)
>>> b.apply_load(3, 3, 0)
>>> b.apply_load(1, 3, 1)
>>> b.apply_load(R2, 6, -1)
>>> b.bc_deflection.append((0, 0))
>>> b.bc_deflection.append((6, 0))
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.reaction_loads
{R1: -11/4, R2: -43/4}
```

```
>>> b.load
```

$$-\frac{11 \cdot \langle x \rangle^{-1}}{4} - \frac{3 \cdot \langle x - 3 \rangle^{-2}}{2} + 3 \cdot \langle x - 3 \rangle^0 + \langle x - 3 \rangle^1 - \frac{43 \cdot \langle x - 6 \rangle^{-1}}{4}$$

```
>>> plot(b.load)
```

```
>>> b.shear_force()
```

$$\frac{11 \cdot \langle x \rangle^0}{4} + \frac{3 \cdot \langle x - 3 \rangle^{-1}}{2} - 3 \cdot \langle x - 3 \rangle^1 - \frac{\langle x - 3 \rangle^2}{2} + \frac{43 \cdot \langle x - 6 \rangle^0}{4}$$

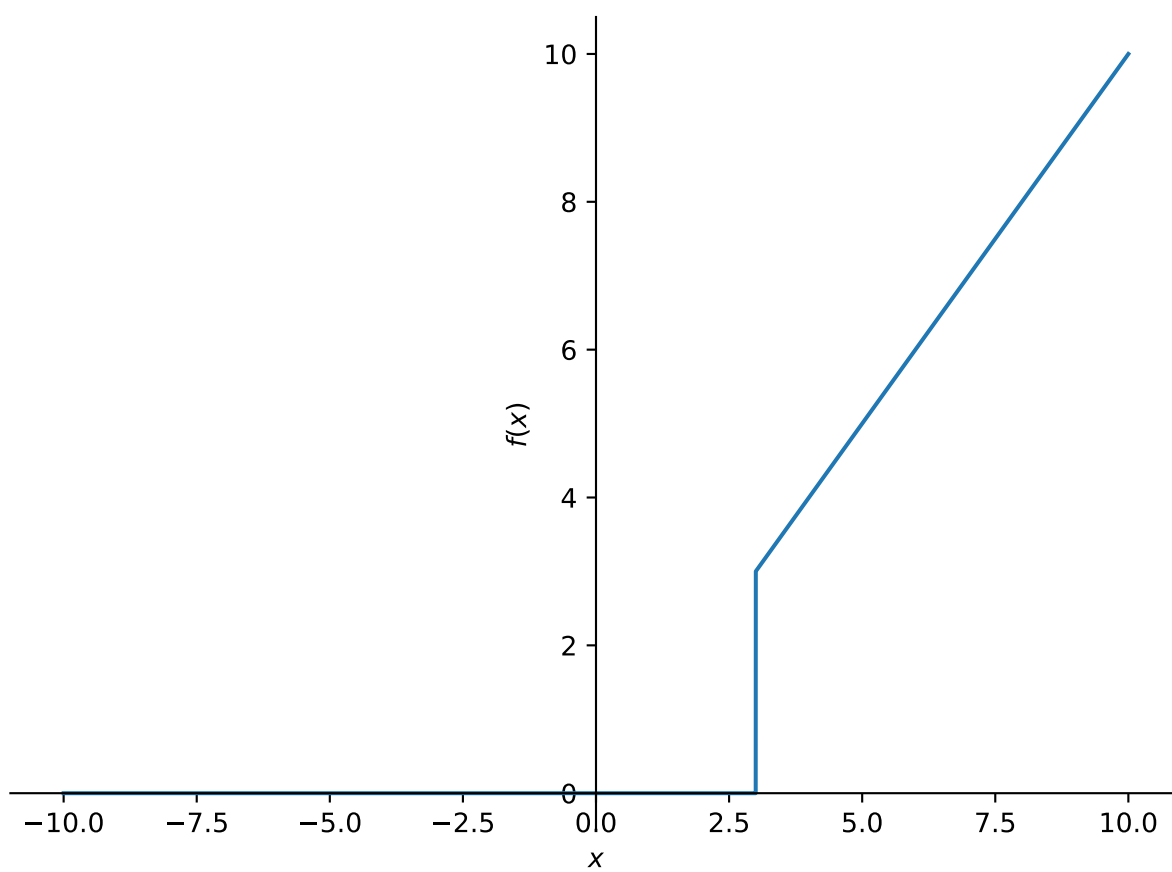
```
>>> b.bending_moment()
```

$$\frac{11 \cdot \langle x \rangle^1}{4} + \frac{3 \cdot \langle x - 3 \rangle^0}{2} - \frac{3 \cdot \langle x - 3 \rangle^2}{2} - \frac{\langle x - 3 \rangle^3}{6} + \frac{43 \cdot \langle x - 6 \rangle^1}{4}$$

```
>>> b.slope()
```

$$-\frac{11 \cdot \langle x \rangle^2}{8} - \frac{3 \cdot \langle x - 3 \rangle^1}{2} + \frac{\langle x - 3 \rangle^3}{2} + \frac{\langle x - 3 \rangle^4}{24} - \frac{43 \cdot \langle x - 6 \rangle^2}{8} + \frac{78}{5}$$

$E \cdot I$





```
>>> b.deflection()

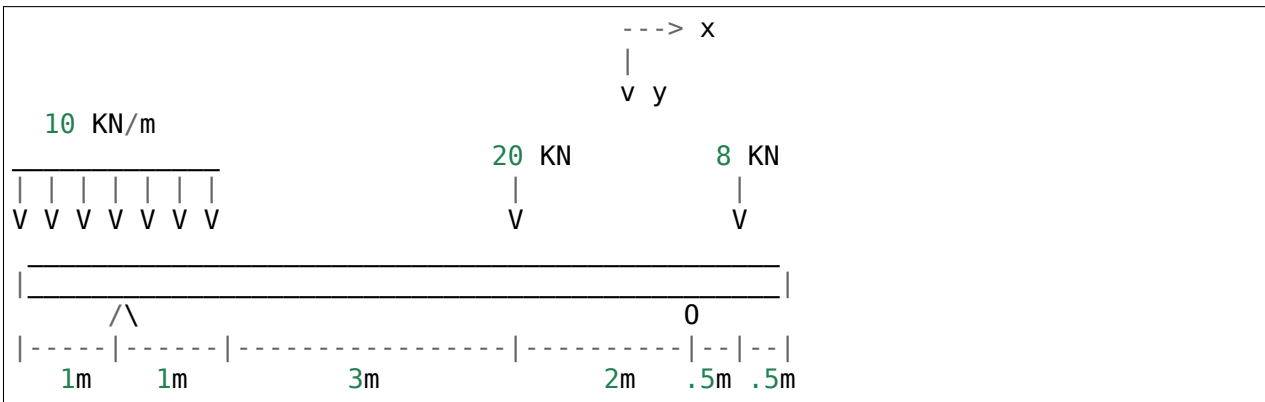
$$\frac{78 \cdot x^5}{5} - \frac{11 \cdot \langle x \rangle^3}{24} - \frac{3 \cdot \langle x - 3 \rangle^2}{4} + \frac{\langle x - 3 \rangle^4}{8} + \frac{\langle x - 3 \rangle^5}{120} - \frac{43 \cdot \langle x - 6 \rangle^3}{24}$$

$$E \cdot I$$

```

#### Example 4

An overhanging beam of length 8 meters is pinned at 1 meter from starting point and supported by a roller 1 meter before the other end. It is subjected to a distributed constant load of 10 KN/m from the starting point till 2 meters away from it. Two point loads of 20KN and 8KN are applied at 5 meters and 7.5 meters away from the starting point respectively.



```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E,I,M,V = symbols('E I M V')
>>> b = Beam(8, E, I)
>>> E,I,R1,R2 = symbols('E I R1 R2')
>>> b.apply_load(R1, 1, -1)
>>> b.apply_load(R2, 7, -1)
>>> b.apply_load(10, 0, 0, end=2)
>>> b.apply_load(20, 5, -1)
>>> b.apply_load(8, 7.5, -1)
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.reaction_loads
{R1: -26, R2: -22}
>>> b.load
0 -1 0 -1 -1
-1
10·<x> - 26·<x - 1> - 10·<x - 2> + 20·<x - 5> - 22·<x - 7> + 8·<x - 7.5>
5>

>>> b.shear_force()
1 0 1 0 0
0
- 10·<x> + 26·<x - 1> + 10·<x - 2> - 20·<x - 5> + 22·<x - 7> - 8·<x - 7.5>
```

(continues on next page)

(continued from previous page)

```

->5>

>>> b.bending_moment()
 2 1 2 1 1 1
- 5·<x> + 26·<x - 1> + 5·<x - 2> - 20·<x - 5> + 22·<x - 7> - 8·<x - 7.5>

>>> b.bc_deflection = [(1, 0), (7, 0)]
>>> b.slope()
 3 2 3 2 2 2
5·<x> - 13·<x - 1> - 5·<x - 2> + 10·<x - 5> - 11·<x - 7> + 4·<x - 7.5>
-> 679

 3 3
-> + -----
 3 3
-> 24

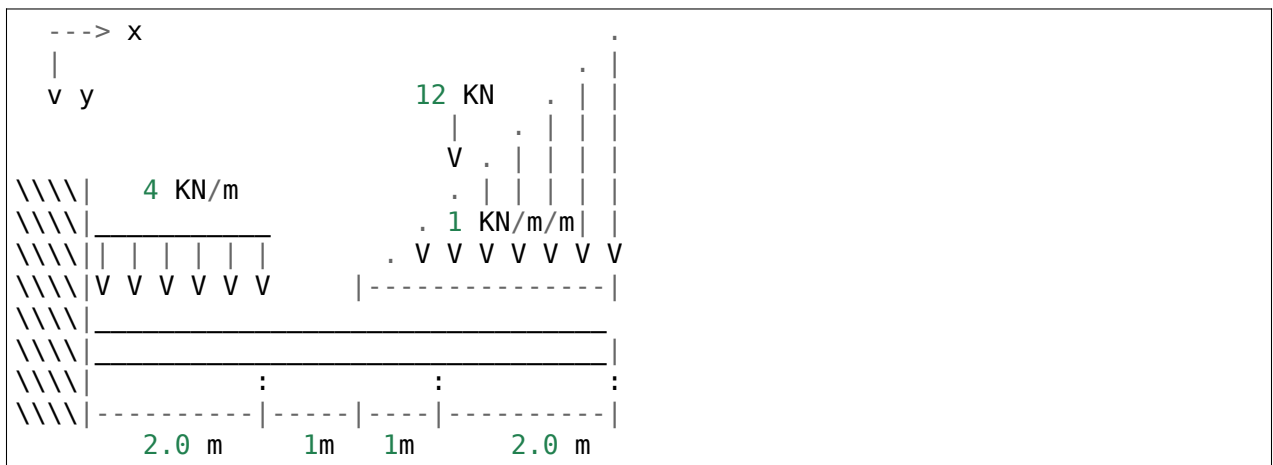
>>> b.deflection()
 4 3 4 3 3
E·I
679·x - 5·<x> - 13·<x - 1> - 5·<x - 2> - 10·<x - 5> - 11·<x - 7> - 4·<x -
-> 7.5>

 24 12 3 12 3 3
-> 3 24 3 12 3 3
E·I

```

### Example 5

A cantilever beam of length 6 meters is under downward distributed constant load with magnitude of 4.0 kN/m from starting point till 2 meters away from it. A ramp load of 1 kN/m/m applied from the mid till the end of the beam. A point load of 12kN is also applied in same direction 4 meters away from start.



```

>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E,I,M,V = symbols('E I M V')
>>> b = Beam(6, E, I)
>>> b.apply_load(V, 0, -1)
>>> b.apply_load(M, 0, -2)
>>> b.apply_load(4, 0, 0, end=2)
>>> b.apply_load(12, 4, -1)
>>> b.apply_load(1, 3, 1, end=6)
>>> b.solve_for_reaction_loads(V, M)
>>> b.reaction_loads
{M: 157/2, V: -49/2}
>>> b.load

```

$$\frac{157 \cdot \langle x \rangle^{-2}}{0} - \frac{49 \cdot \langle x \rangle^{-1}}{1} + 4 \cdot \langle x \rangle^0 - 4 \cdot \langle x - 2 \rangle^0 + \langle x - 3 \rangle^1 + 12 \cdot \langle x - 4 \rangle^{-1} - 3 \cdot \langle x - 6 \rangle^{-1}$$

```

>>> b.shear_force()

```

$$-\frac{157 \cdot \langle x \rangle^{-1}}{1} + \frac{49 \cdot \langle x \rangle^0}{\langle x - 6 \rangle} - 4 \cdot \langle x \rangle^1 + 4 \cdot \langle x - 2 \rangle^1 - \frac{\langle x - 3 \rangle^2}{2} - 12 \cdot \langle x - 4 \rangle^2 + 3 \cdot \langle x - 6 \rangle^2$$

```

>>> b.bending_moment()

```

$$-\frac{157 \cdot \langle x \rangle^2}{2} + \frac{49 \cdot \langle x \rangle^3}{2} - 2 \cdot \langle x \rangle^2 + 2 \cdot \langle x - 2 \rangle^2 - \frac{\langle x - 3 \rangle^3}{6} - 12 \cdot \langle x - 4 \rangle^3 + \frac{3 \cdot \langle x - 6 \rangle^3}{2}$$

```

>>> b.bc_deflection = [(0, 0)]
>>> b.bc_slope = [(0, 0)]
>>> b.slope()

```

$$\left( \frac{157 \cdot \langle x \rangle^3}{24} - \frac{49 \cdot \langle x \rangle^4}{4} - \frac{2 \cdot \langle x \rangle^3}{3} + \frac{2 \cdot \langle x - 2 \rangle^3}{3} - \frac{\langle x - 3 \rangle^4}{24} - 6 \cdot \langle x - 4 \rangle^4 + \frac{\langle x - 6 \rangle^4}{2} \right)$$

```

>>> b.deflection()

```

$$\left( \frac{\dots}{E \cdot I} \right)$$

(continues on next page)

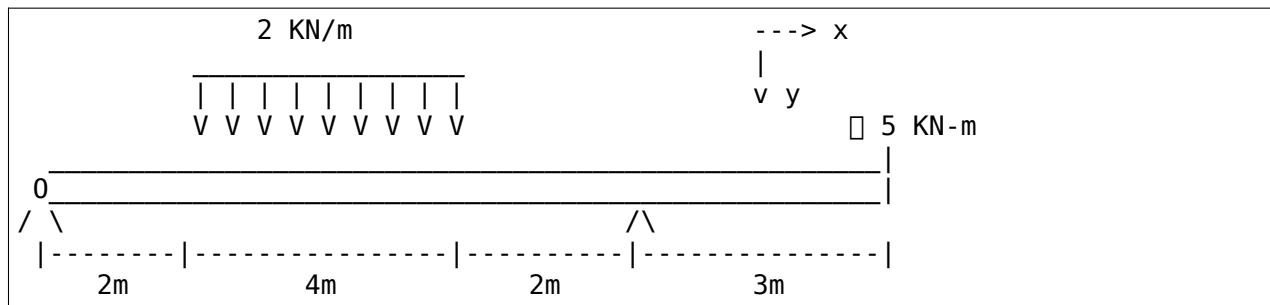
(continued from previous page)

$$\begin{array}{r} \overline{\overbrace{157 \cdot \langle x \rangle}^{5)}} + \frac{49 \cdot \langle x \rangle}{12} - \frac{\langle x \rangle}{6} + \frac{\langle x - 2 \rangle}{6} - \frac{\langle x - 3 \rangle}{120} - 2 \cdot \langle x - 4 \rangle + \frac{3 \cdot \langle x - 6 \rangle}{8} \\ - \left( \frac{\langle x - 6 \rangle}{4} - \frac{\langle x - 6 \rangle^2}{120} \right) \end{array}$$

E · I

### Example 6

An overhanging beam of length 11 meters is subjected to a distributed constant load of 2 KN/m from 2 meters away from the starting point till 6 meters away from it. It is pinned at the starting point and is resting over a roller 8 meters away from that end. Also a counter-clockwise moment of 5 KN-m is applied at the overhanging end.



```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> R1, R2 = symbols('R1, R2')
>>> E, I = symbols('E, I')
>>> b = Beam(11, E, I)
>>> b.apply_load(R1, 0, -1)
>>> b.apply_load(2, 2, 0, end=6)
>>> b.apply_load(R2, 8, -1)
>>> b.apply_load(5, 11, -2)
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.reaction_loads
{R1: -37/8, R2: -27/8}
>>> b.load
-1
37·<x>0 + 2·<x - 2>0 - 2·<x - 6>0 - $\frac{27 \cdot \langle x - 8 \rangle^{-1}}{8}$ + 5·<x - 11>-2
- $\frac{37 \cdot \langle x \rangle^0}{8}$
>>> b.shear_force()
0
37·<x>0 - 2·<x - 2>1 + 2·<x - 6>1 + $\frac{27 \cdot \langle x - 8 \rangle^0}{8}$ - 5·<x - 11>-1
 $\frac{37 \cdot \langle x \rangle^0}{8}$
```

(continues on next page)

(continued from previous page)

```
>>> b.bending_moment()
 1
 37·<x>
 8
 2
 - <x - 2>
 2
 + <x - 6>
 2
 + 27·<x - 8>
 8
 1
 - 5·<x - 11>
 0

>>> b.bc_deflection = [(0, 0), (8, 0)]
>>> b.slope()
 2
 37·<x>
 16
 +
 3
 <x - 2>
 3
 -
 3
 <x - 6>
 3
 -
 2
 27·<x - 8>
 16
 +
 1
 5·<x - 11>
 + 36

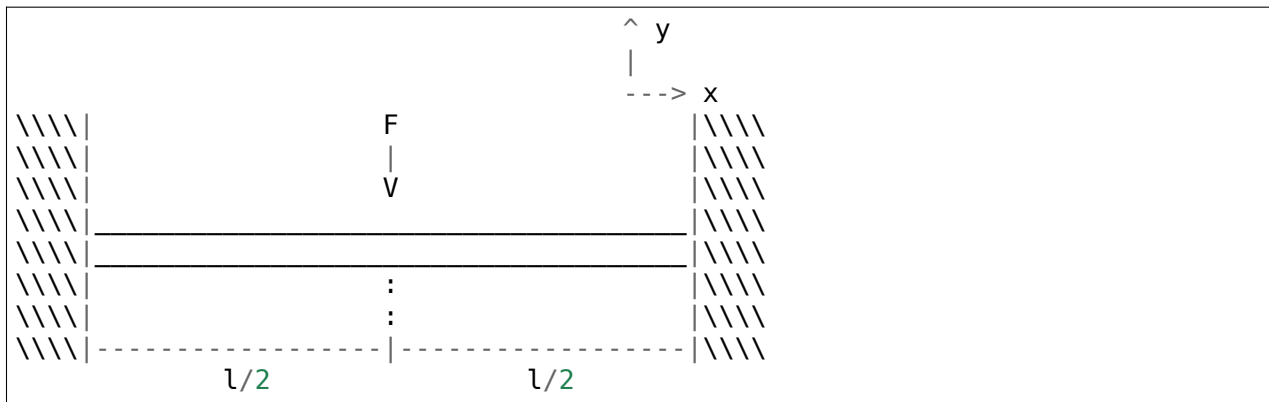
 E·I

>>> b.deflection()
 3
 37·<x>
 48
 +
 4
 <x - 2>
 12
 -
 4
 <x - 6>
 12
 -
 3
 9·<x - 8>
 16
 +
 2
 5·<x - 11>
 2

 E·I
```

### Example 7

There is a beam of length  $l$ , fixed at both ends. A concentrated point load of magnitude  $F$  is applied in downward direction at mid-point of the beam.



```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I, F = symbols('E I F')
>>> l = symbols('l', positive=True)
>>> b = Beam(l, E, I)
>>> R1, R2 = symbols('R1 R2')
>>> M1, M2 = symbols('M1, M2')
>>> b.apply_load(R1, 0, -1)
>>> b.apply_load(M1, 0, -2)
>>> b.apply_load(R2, l, -1)
>>> b.apply_load(M2, l, -2)
```

(continues on next page)

(continued from previous page)

```

>>> b.apply_load(-F, l/2, -1)
>>> b.bc_deflection = [(0, 0), (l, 0)]
>>> b.bc_slope = [(0, 0), (l, 0)]
>>> b.solve_for_reaction_loads(R1, R2, M1, M2)
>>> b.reaction_loads
{M1: $-\frac{F \cdot l}{8}$, M2: $\frac{F \cdot l}{8}$, R1: $-\frac{F}{2}$, R2: $-\frac{F}{2}$ }

>>> b.load

$$-\frac{F \cdot l \cdot \langle x \rangle^{-2}}{8} + \frac{F \cdot l \cdot \langle -l + x \rangle^{-2}}{8} + \frac{F \cdot \langle x \rangle^{-1}}{2} - F \cdot \langle -\frac{l}{2} + x \rangle^{-1} + \frac{F \cdot \langle -l + x \rangle^{-1}}{2}$$

>>> b.shear_force()

$$\frac{F \cdot l \cdot \langle x \rangle^{-1}}{8} - \frac{F \cdot l \cdot \langle -l + x \rangle^{-1}}{8} - \frac{F \cdot \langle x \rangle^0}{2} + F \cdot \langle -\frac{l}{2} + x \rangle^0 - \frac{F \cdot \langle -l + x \rangle^0}{2}$$

>>> b.bending_moment()

$$\frac{F \cdot l \cdot \langle x \rangle^0}{8} - \frac{F \cdot l \cdot \langle -l + x \rangle^0}{8} - \frac{F \cdot \langle x \rangle^1}{2} + F \cdot \langle -\frac{l}{2} + x \rangle^1 - \frac{F \cdot \langle -l + x \rangle^1}{2}$$

>>> b.slope()

$$-\left(\frac{F \cdot l \cdot \langle x \rangle^1}{8} - \frac{F \cdot l \cdot \langle -l + x \rangle^1}{8} - \frac{F \cdot \langle x \rangle^2}{4} + \frac{F \cdot \langle -\frac{l}{2} + x \rangle^2}{2} - \frac{F \cdot \langle -l + x \rangle^2}{4} \right)$$

$E \cdot I$

>>> b.deflection()

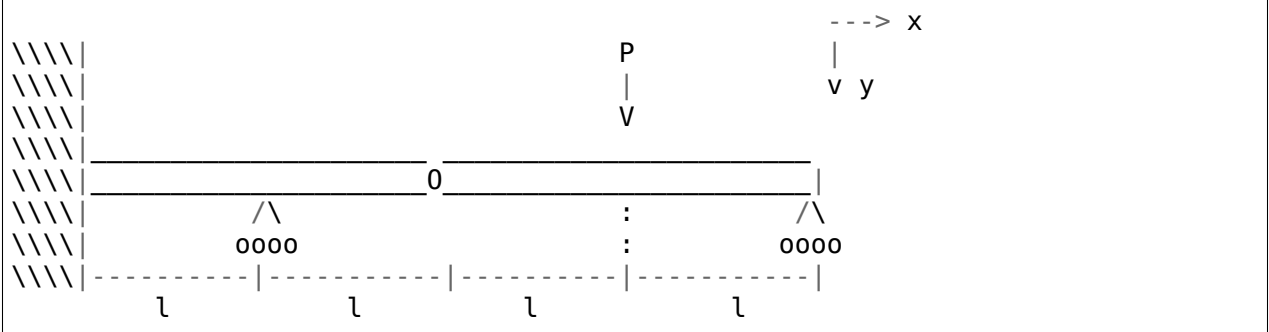
$$-\left(\frac{F \cdot l \cdot \langle x \rangle^2}{16} - \frac{F \cdot l \cdot \langle -l + x \rangle^2}{16} - \frac{F \cdot \langle x \rangle^3}{12} + \frac{F \cdot \langle -\frac{l}{2} + x \rangle^3}{6} - \frac{F \cdot \langle -l + x \rangle^3}{12} \right)$$

$E \cdot I$


```

### Example 8

There is a beam of length  $4 \cdot l$ , having a hinge connector at the middle. It is having a fixed support at the start and also has two rollers at a distance of  $l$  and  $4 \cdot l$  from the starting point. A concentrated point load  $P$  is also applied at a distance of  $3 \cdot l$  from the starting point.



```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E I')
>>> l = symbols('l', positive=True)
>>> R1, M1, R2, R3, P = symbols('R1 M1 R2 R3 P')
>>> b1 = Beam(2*l, E, I)
>>> b2 = Beam(2*l, E, I)
>>> b = b1.join(b2, "hinge")
>>> b.apply_load(M1, 0, -2)
>>> b.apply_load(R1, 0, -1)
>>> b.apply_load(R2, l, -1)
>>> b.apply_load(R3, 4*l, -1)
>>> b.apply_load(P, 3*l, -1)
>>> b.bc_slope = [(0, 0)]
>>> b.bc_deflection = [(0, 0), (l, 0), (4*l, 0)]
>>> b.solve_for_reaction_loads(M1, R1, R2, R3)
>>> b.reaction_loads
{M1: -P*l/4, R1: 3*P/4, R2: -5*P/4, R3: -P/2}
```

$$b.load = -\frac{P \cdot l \cdot \langle x \rangle^{-2}}{4} + \frac{3 \cdot P \cdot \langle x \rangle^{-1}}{4} - \frac{5 \cdot P \cdot \langle -l + x \rangle^{-1}}{4} + P \cdot \langle -3 \cdot l + x \rangle^{-1} - \frac{P \cdot \langle -4 \cdot l + x \rangle^{-1}}{2}$$

```
>>> b.shear_force()
P*l*⟨x⟩-1/4 - 3*P*⟨x⟩0/4 + 5*P*⟨-l+x⟩0/4 - P*⟨-3l+x⟩0 + P*⟨-4l+x⟩0/2
```

```
>>> b.bending_moment()
0 1 1 1
```

(continues on next page)

## Chapter 5. API Reference

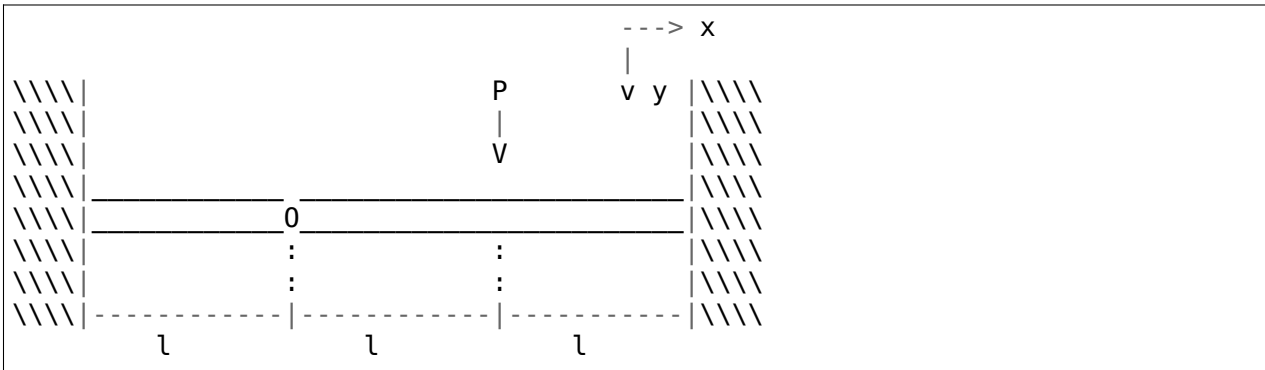




(continued from previous page)

 $\rightarrow E \cdot I$ **Example 10**

A combined beam, with constant flexural rigidity  $E \cdot I$ , is formed by joining a Beam of length  $2 \cdot l$  to the right of another Beam of length  $l$ . The whole beam is fixed at both of its ends. A point load of magnitude  $P$  is also applied from the top at a distance of  $2 \cdot l$  from starting point.



```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> l = symbols('l', positive=True)
>>> b1 = Beam(l, E, I)
>>> b2 = Beam(2*l, E, I)
>>> b = b1.join(b2, "hinge")
>>> M1, A1, M2, A2, P = symbols('M1 A1 M2 A2 P')
>>> b.apply_load(A1, 0, -1)
>>> b.apply_load(M1, 0, -2)
>>> b.apply_load(P, 2*l, -1)
>>> b.apply_load(A2, 3*l, -1)
>>> b.apply_load(M2, 3*l, -2)
>>> b.bc_slope=[(0, 0), (3*l, 0)]
>>> b.bc_deflection=[(0, 0), (3*l, 0)]
>>> b.solve_for_reaction_loads(M1, A1, M2, A2)
>>> b.reaction_loads
{A1: -5*P/18, A2: -13*P/18, M1: 5*P*l/18, M2: -4*P*l/9}
```

$$b.load = \frac{5 \cdot P \cdot l \cdot \langle x \rangle^{-2}}{18} - \frac{4 \cdot P \cdot l \cdot \langle -3 \cdot l + x \rangle^{-2}}{9} - \frac{5 \cdot P \cdot \langle x \rangle^{-1}}{18} + P \cdot \langle -2 \cdot l + x \rangle^{-1} - \frac{13 \cdot P \cdot \langle -3 \cdot l + x \rangle^{-1}}{18}$$

(continues on next page)

(continued from previous page)

```

>>> b.shear_force()
 -1 -1 0
 0
 5·P·l·<x> 4·P·l·<-3·l + x> 5·P·<x> 0 13·P·<-3·l + x>
 x>
 - ----- + ----- + ----- - P·<-2·l + x> +
 18 9 18 18

>>> b.bending_moment()
 0 0 1 1 1
 5·P·l·<x> 4·P·l·<-3·l + x> 5·P·<x> P·<-2·l + x> 13·P·<-3·l + x>
 - ----- + ----- + ----- - P·<-2·l + x> + -----
 18 9 18 18 18

>>> b.slope()
 1 2 2 1 1 2 2
 2) 2) 2) 1 1 2 2
 2) 2) 2) 1 1 2 2
 | 5·P·l·<x> 5·P·<x> 5·P·<-l + x> | 0 | 5·P·l·<x> 5·P·<x> 5·P·<-
 l + x> | 0 | P·l 4·P·l·<-3·l + x> 5·P·<-l + x> P·<-2·l +
 x> 13·P·<-3·l + x> | 0
 | ----- - ----- + ----- |·<x> | ----- - ----- +
 <-l + x> | ----- - ----- +
 <-l + x>
 (18 36 36 18 36 36 2
 36) 36) 9 (18 36 36
 36) 36)
 ----- -
 ----- +

 E·I E·I E·I

>>> b.deflection()
 2 3 3 2 2 3 3
 3) 3) 3) 2 2 3 3
 3) 3) 3) 2 2 3 3
 | 5·P·l·<x> 5·P·<x> 5·P·<-l + x> | 0 | 5·P·l·<x> 5·P·<x> 5·P·<-
 l + x> | 0 | 5·P·l P·l·<(-l + x)> 2·P·l·<-3·l + x> 5·P·<-l
 + x> P·<-2·l + x> 13·P·<-3·l + x> | 0
 | ----- - ----- + ----- |·<x> | ----- - ----- +
 <-l + x> | ----- + ----- - -----
 <-l + x>
 (36 108 108 54 108 18 36 108
 108) 6 (54 108 18) 9 108
 108) 6 (54 108 18)
 ----- -
 ----- +

 E·I E·I E·I

```

(continues on next page)

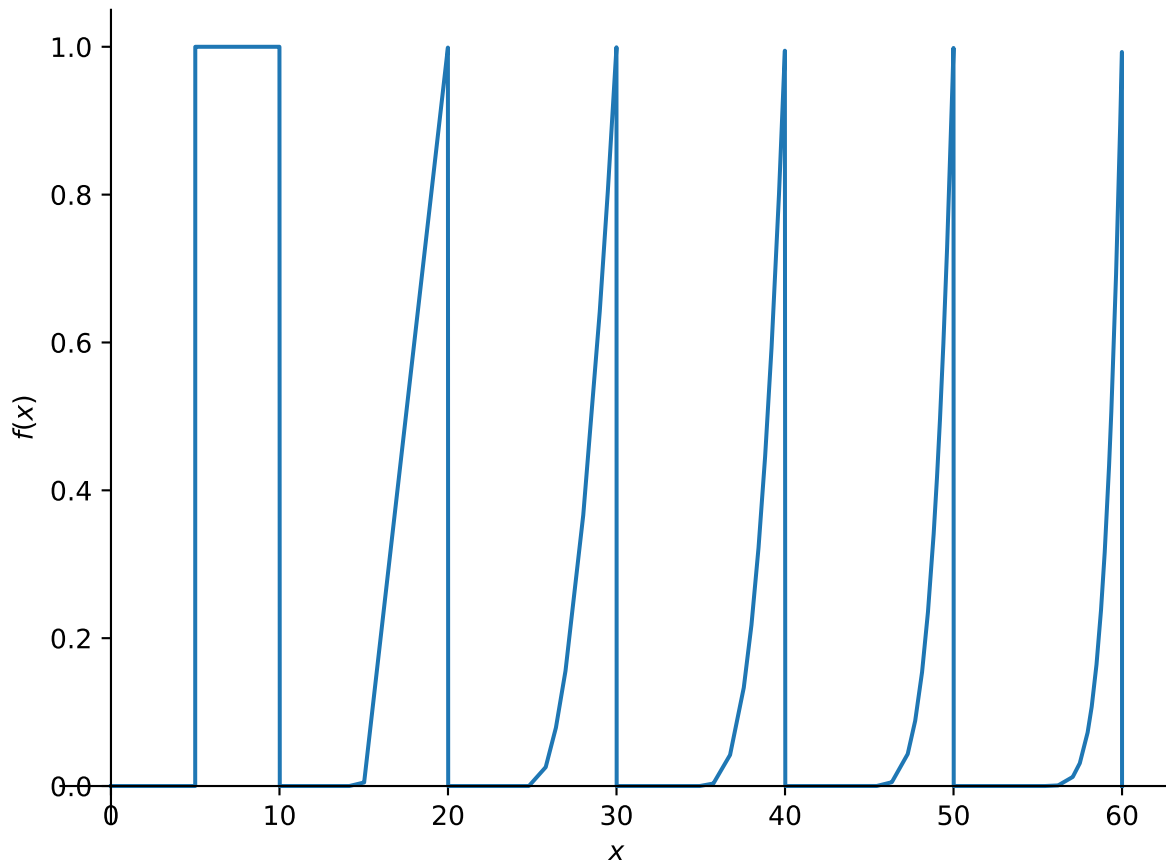
(continued from previous page)

|             |             |             |
|-------------|-------------|-------------|
| $E \cdot I$ | $E \cdot I$ | $E \cdot I$ |
|-------------|-------------|-------------|

**Example 11**

Any type of load defined by a polynomial can be applied to the beam. This allows approximation of arbitrary load distributions. The following example shows six truncated polynomial loads across the surface of a beam.

```
>>> n = 6
>>> b = Beam(10*n, E, I)
>>> for i in range(n):
... b.apply_load(1 / (5**i), 10*i + 5, i, end=10*i + 10)
>>> plot(b.load, (x, 0, 10*n))
```



## Truss

### Truss (Docstrings)

## Truss

This module can be used to solve problems related to 2D Trusses.

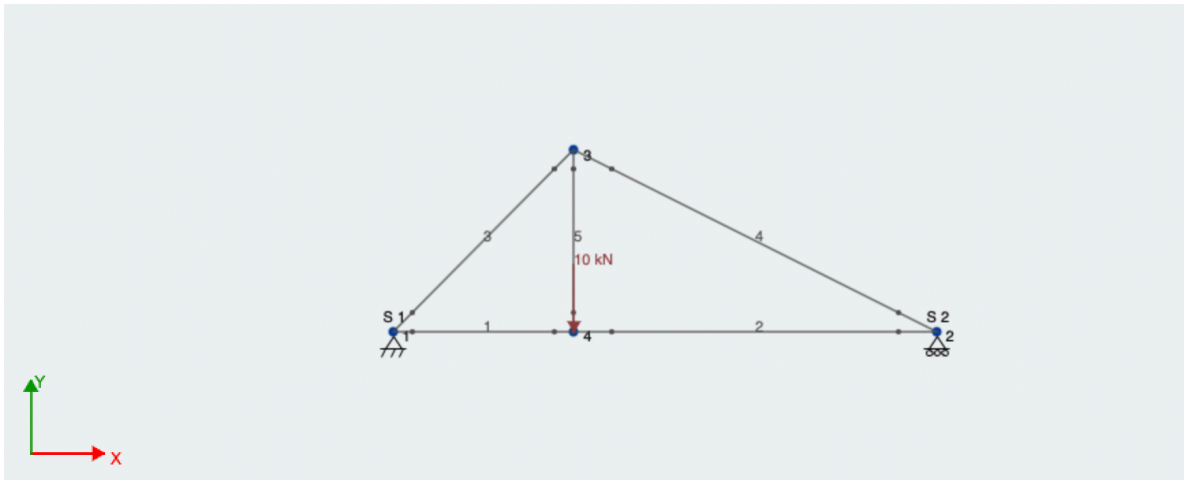
**class** `sympy.physics.continuum_mechanics.truss.Truss`

A Truss is an assembly of members such as beams, connected by nodes, that create a rigid structure. In engineering, a truss is a structure that consists of two-force members only.

Trusses are extremely important in engineering applications and can be seen in numerous real-world applications like bridges.

### Examples

There is a Truss consisting of four nodes and five members connecting the nodes. A force  $P$  acts downward on the node D and there also exist pinned and roller joints on the nodes A and B respectively.



```
>>> from sympy.physics.continuum_mechanics.truss import Truss
>>> t = Truss()
>>> t.add_node("node_1", 0, 0)
>>> t.add_node("node_2", 6, 0)
>>> t.add_node("node_3", 2, 2)
>>> t.add_node("node_4", 2, 0)
>>> t.add_member("member_1", "node_1", "node_4")
>>> t.add_member("member_2", "node_2", "node_4")
>>> t.add_member("member_3", "node_1", "node_3")
>>> t.add_member("member_4", "node_2", "node_3")
>>> t.add_member("member_5", "node_3", "node_4")
>>> t.apply_load("node_4", magnitude=10, direction=270)
>>> t.apply_support("node_1", type="fixed")
>>> t.apply_support("node_2", type="roller")
```

**add\_member**(*label, start, end*)

This method adds a member between any two nodes in the given truss.

**Parameters**

**label: String or Symbol**

The label for a member. It is the only way to identify a particular member.

**start: String or Symbol**

The label of the starting point/node of the member.

**end: String or Symbol**

The label of the ending point/node of the member.

**Examples**

```
>>> from sympy.physics.continuum_mechanics.truss import Truss
>>> t = Truss()
>>> t.add_node('A', 0, 0)
>>> t.add_node('B', 3, 0)
>>> t.add_node('C', 2, 2)
>>> t.add_member('AB', 'A', 'B')
>>> t.members
{'AB': ['A', 'B']}
```

**add\_node**(*label, x, y*)

This method adds a node to the truss along with its name/label and its location.

**Parameters**

**label: String or a Symbol**

The label for a node. It is the only way to identify a particular node.

**x: Sympifyable**

The x-coordinate of the position of the node.

**y: Sympifyable**

The y-coordinate of the position of the node.

**Examples**

```
>>> from sympy.physics.continuum_mechanics.truss import Truss
>>> t = Truss()
>>> t.add_node('A', 0, 0)
>>> t.nodes
[('A', 0, 0)]
>>> t.add_node('B', 3, 0)
>>> t.nodes
[('A', 0, 0), ('B', 3, 0)]
```

**apply\_load**(*location, magnitude, direction*)

This method applies an external load at a particular node

**Parameters**

**location: String or Symbol**

Label of the Node at which load is applied.

**magnitude: Sympifyable**

Magnitude of the load applied. It must always be positive and any changes in the direction of the load are not reflected here.

**direction: Sympifyable**

The angle, in degrees, that the load vector makes with the horizontal in the counter-clockwise direction. It takes the values 0 to 360, inclusive.

**Examples**

```
>>> from sympy.physics.continuum_mechanics.truss import Truss
>>> from sympy import symbols
>>> t = Truss()
>>> t.add_node('A', 0, 0)
>>> t.add_node('B', 3, 0)
>>> P = symbols('P')
>>> t.apply_load('A', P, 90)
>>> t.apply_load('A', P/2, 45)
>>> t.apply_load('A', P/4, 90)
>>> t.loads
{'A': [[P, 90], [P/2, 45], [P/4, 90]]}
```

**apply\_support**(*location, type*)

This method adds a pinned or roller support at a particular node

**Parameters**

**location: String or Symbol**

Label of the Node at which support is added.

**type: String**

Type of the support being provided at the node.

**Examples**

```
>>> from sympy.physics.continuum_mechanics.truss import Truss
>>> t = Truss()
>>> t.add_node('A', 0, 0)
>>> t.add_node('B', 3, 0)
>>> t.apply_support('A', 'pinned')
>>> t.supports
{'A': 'pinned'}
```

**change\_member\_label**(*label*, *new\_label*)

This method changes the label of a member.

**Parameters**

**label:** String or Symbol

The label of the member for which the label has to be changed.

**new\_label:** String or Symbol

The new label of the member.

**Examples**

```
>>> from sympy.physics.continuum_mechanics.truss import Truss
>>> t = Truss()
>>> t.add_node('A', 0, 0)
>>> t.add_node('B', 3, 0)
>>> t.nodes
[('A', 0, 0), ('B', 3, 0)]
>>> t.change_node_label('A', 'C')
>>> t.nodes
[('C', 0, 0), ('B', 3, 0)]
>>> t.add_member('BC', 'B', 'C')
>>> t.members
{'BC': ['B', 'C']}
>>> t.change_member_label('BC', 'BC_new')
>>> t.members
{'BC_new': ['B', 'C']}
```

**change\_node\_label**(*label*, *new\_label*)

This method changes the label of a node.

**Parameters**

**label:** String or Symbol

The label of the node for which the label has to be changed.

**new\_label:** String or Symbol

The new label of the node.

**Examples**

```
>>> from sympy.physics.continuum_mechanics.truss import Truss
>>> t = Truss()
>>> t.add_node('A', 0, 0)
>>> t.add_node('B', 3, 0)
>>> t.nodes
[('A', 0, 0), ('B', 3, 0)]
>>> t.change_node_label('A', 'C')
>>> t.nodes
[('C', 0, 0), ('B', 3, 0)]
```



**property internal\_forces**

Returns the internal forces for all members which are all initialized to 0.

**property loads**

Returns the loads acting on the truss.

**property member\_labels**

Returns the members of the truss along with the start and end points.

**property members**

Returns the members of the truss along with the start and end points.

**property node\_labels**

Returns the node labels of the truss.

**property node\_positions**

Returns the positions of the nodes of the truss.

**property nodes**

Returns the nodes of the truss along with their positions.

**property reaction\_loads**

Returns the reaction forces for all supports which are all initialized to 0.

**remove\_load(*location, magnitude, direction*)**

This method removes an already present external load at a particular node

**Parameters****location: String or Symbol**

Label of the Node at which load is applied and is to be removed.

**magnitude: Sympifyable**

Magnitude of the load applied.

**direction: Sympifyable**

The angle, in degrees, that the load vector makes with the horizontal in the counter-clockwise direction. It takes the values 0 to 360, inclusive.

**Examples**

```
>>> from sympy.physics.continuum_mechanics.truss import Truss
>>> from sympy import symbols
>>> t = Truss()
>>> t.add_node('A', 0, 0)
>>> t.add_node('B', 3, 0)
>>> P = symbols('P')
>>> t.apply_load('A', P, 90)
>>> t.apply_load('A', P/2, 45)
>>> t.apply_load('A', P/4, 90)
>>> t.loads
{'A': [[P, 90], [P/2, 45], [P/4, 90]]}
>>> t.remove_load('A', P/4, 90)
```

(continues on next page)

(continued from previous page)

```
>>> t.loads
{'A': [[P, 90], [P/2, 45]]}
```

**remove\_member(*label*)**

This method removes a member from the given truss.

**Parameters****label: String or Symbol**

The label for the member to be removed.

**Examples**

```
>>> from sympy.physics.continuum_mechanics.truss import Truss
>>> t = Truss()
>>> t.add_node('A', 0, 0)
>>> t.add_node('B', 3, 0)
>>> t.add_node('C', 2, 2)
>>> t.add_member('AB', 'A', 'B')
>>> t.add_member('AC', 'A', 'C')
>>> t.add_member('BC', 'B', 'C')
>>> t.members
{'AB': ['A', 'B'], 'AC': ['A', 'C'], 'BC': ['B', 'C']}
>>> t.remove_member('AC')
>>> t.members
{'AB': ['A', 'B'], 'BC': ['B', 'C']}
```

**remove\_node(*label*)**

This method removes a node from the truss.

**Parameters****label: String or Symbol**

The label of the node to be removed.

**Examples**

```
>>> from sympy.physics.continuum_mechanics.truss import Truss
>>> t = Truss()
>>> t.add_node('A', 0, 0)
>>> t.nodes
[('A', 0, 0)]
>>> t.add_node('B', 3, 0)
>>> t.nodes
[('A', 0, 0), ('B', 3, 0)]
>>> t.remove_node('A')
>>> t.nodes
[('B', 3, 0)]
```

**remove\_support(*location*)**

This method removes support from a particular node

**Parameters****location: String or Symbol**

Label of the Node at which support is to be removed.

**Examples**

```
>>> from sympy.physics.continuum_mechanics.truss import Truss
>>> t = Truss()
>>> t.add_node('A', 0, 0)
>>> t.add_node('B', 3, 0)
>>> t.apply_support('A', 'pinned')
>>> t.supports
{'A': 'pinned'}
>>> t.remove_support('A')
>>> t.supports
{}
```

**property supports**

Returns the nodes with provided supports along with the kind of support provided i.e. pinned or roller.

## 5.8.7 Utilities

**Contents****Testing**

This module contains code for running the tests in SymPy.

Contents:

**pytest**

py.test hacks to support XFAIL/XPASS

`sympy.testing.pytest.SKIP(reason)`

Similar to `skip()`, but this is a decorator.

`sympy.testing.pytest.nocache_fail(func)`

Dummy decorator for marking tests that fail when cache is disabled

`sympy.testing.pytest.raises(expectedException, code=None)`

Tests that code raises the exception `expectedException`.

`code` may be a callable, such as a lambda expression or function name.

If `code` is not given or `None`, `raises` will return a context manager for use in `with` statements; the code to execute then comes from the scope of the `with`.

`raises()` does nothing if the callable raises the expected exception, otherwise it raises an `AssertionError`.

## Examples

```
>>> from sympy.testing.pytest import raises
```

```
>>> raises(ZeroDivisionError, lambda: 1/0)
<ExceptionInfo ZeroDivisionError(...)>
>>> raises(ZeroDivisionError, lambda: 1/2)
Traceback (most recent call last):
...
Failed: DID NOT RAISE
```

```
>>> with raises(ZeroDivisionError):
... n = 1/0
>>> with raises(ZeroDivisionError):
... n = 1/2
Traceback (most recent call last):
...
Failed: DID NOT RAISE
```

Note that you cannot test multiple statements via `with raises`:

```
>>> with raises(ZeroDivisionError):
... n = 1/0 # will execute and raise, aborting the ``with``
... n = 9999/0 # never executed
```

This is just what `with` is supposed to do: abort the contained statement sequence at the first exception and let the context manager deal with the exception.

To test multiple statements, you'll need a separate `with` for each:

```
>>> with raises(ZeroDivisionError):
... n = 1/0 # will execute and raise
>>> with raises(ZeroDivisionError):
... n = 9999/0 # will also execute and raise
```

`sympy.testing.pytest.warns(warningcls, *, match="", test_stacklevel=True)`

Like `raises` but tests that warnings are emitted.

```
>>> from sympy.testing.pytest import warns
>>> import warnings
```

```
>>> with warns(UserWarning):
... warnings.warn('deprecated', UserWarning, stacklevel=2)
```

```
>>> with warns(UserWarning):
... pass
Traceback (most recent call last):
...
Failed: DID NOT WARN. No warnings of type UserWarning was emitted.
↳ The list of emitted warnings is: [].
```

`test_stacklevel` makes it check that the `stacklevel` parameter to `warn()` is set so that the warning shows the user line of code (the code under the `warns()` context manager).

Set this to False if this is ambiguous or if the context manager does not test the direct user code that emits the warning.

If the warning is a `SymPyDeprecationWarning`, this additionally tests that the `active_deprecations_target` is a real target in the `active-deprecations.md` file.

`sympy.testing.pytest.warns_deprecated_sympy()`

Shorthand for `warns(SymPyDeprecationWarning)`

This is the recommended way to test that `SymPyDeprecationWarning` is emitted for deprecated features in SymPy. To test for other warnings use `warns`. To suppress warnings without asserting that they are emitted use `ignore_warnings`.

---

**Note:** `warns_deprecated_sympy()` is only intended for internal use in the SymPy test suite to test that a deprecation warning triggers properly. All other code in the SymPy codebase, including documentation examples, should not use deprecated behavior.

If you are a user of SymPy and you want to disable `SymPyDeprecationWarnings`, use warnings filters (see [Silencing SymPy Deprecation Warnings](#) (page 164)).

---

```
>>> from sympy.testing.pytest import warns_deprecated_sympy
>>> from sympy.utilities.exceptions import sympy_deprecation_warning
>>> with warns_deprecated_sympy():
... sympy_deprecation_warning("Don't use",
... deprecated_since_version="1.0",
... active_deprecations_target="active-deprecations")
```

```
>>> with warns_deprecated_sympy():
... pass
Traceback (most recent call last):
...
Failed: DID NOT WARN. No warnings of type SymPyDeprecationWarning
↳ was emitted. The list of emitted warnings is: [].
```

---

**Note:** Sometimes the stacklevel test will fail because the same warning is emitted multiple times. In this case, you can use `sympy.utilities.exceptions.ignore_warnings()` (page 2067) in the code to prevent the `SymPyDeprecationWarning` from being emitted again recursively. In rare cases it is impossible to have a consistent stacklevel for deprecation warnings because different ways of calling a function will produce different call stacks.. In those cases, use `warns(SymPyDeprecationWarning)` instead.

---

**See also:**

[sympy.utilities.exceptions.SymPyDeprecationWarning](#) (page 2066), [sympy.utilities.exceptions.sympy\\_deprecation\\_warning](#) (page 2067), [sympy.utilities.decorator.deprecated](#) (page 2057)

## Randomised Testing

Deprecated since version 1.10: `sympy.testing.randtest` functions have been moved to [\*sympy.core.random\*](#) (page 1077).

## Run Tests

This is our testing framework.

Goals:

- it should be compatible with `py.test` and operate very similarly (or identically)
- does not require any external dependencies
- preferably all the functionality should be in this file only
- no magic, just import the test file and execute the test functions, that's it
- portable

**class** `sympy.testing.runtests.PyTestReporter`(*verbose=False, tb='short', colors=True, force\_colors=False, split=None*)

Py.test like reporter. Should produce output identical to `py.test`.

**write**(*text, color="", align='left', width=None, force\_colors=False*)

Prints a text on the screen.

It uses `sys.stdout.write()`, so no `readline` library is necessary.

### Parameters

**color** : choose from the colors below, "" means default color

**align** : "left"/"right", "left" is a normal print, "right" is aligned on the right-hand side of the screen, filled with spaces if necessary

**width** : the screen width

**class** `sympy.testing.runtests.Reporter`

Parent class for all reporters.

**class** `sympy.testing.runtests.SymPyDocTestFinder`(*verbose=False, parser=<doctest.DocTestParser object>, recurse=True, exclude\_empty=True*)

A class used to extract the DocTests that are relevant to a given object, from its docstring and the docstrings of its contained objects. Doctests can currently be extracted from the following object types: modules, functions, classes, methods, staticmethods, classmethods, and properties.

Modified from `doctest`'s version to look harder for code that appears comes from a different module. For example, the `@vectorize` decorator makes it look like functions come from `multidimensional.py` even though their code exists elsewhere.

**class** `sympy.testing.runtests.SymPyDocTestRunner`(*checker=None, verbose=None, optionflags=0*)

A class used to run DocTest test cases, and accumulate statistics. The `run` method is used to process a single DocTest case. It returns a tuple (`f`, `t`), where `t` is the number of test cases tried, and `f` is the number of test cases that failed.

Modified from the doctest version to not reset the `sys.displayhook` (see issue 5140).

See the docstring of the original `DocTestRunner` for more information.

**run**(*test*, *compileflags=None*, *out=None*, *clear\_globs=True*)

Run the examples in *test*, and display the results using the writer function *out*.

The examples are run in the namespace `test.globs`. If *clear\_globs* is true (the default), then this namespace will be cleared after the test runs, to help with garbage collection. If you would like to examine the namespace after the test completes, then use *clear\_globs=False*.

*compileflags* gives the set of flags that should be used by the Python compiler when running the examples. If not specified, then it will default to the set of future-import flags that apply to *globs*.

The output of each example is checked using `SymPyDocTestRunner.check_output`, and the results are formatted by the `SymPyDocTestRunner.report_*` methods.

**class** `sympy.testing.runtests.SymPyOutputChecker`

Compared to the `OutputChecker` from the `stdlib` our `OutputChecker` class supports numerical comparison of floats occurring in the output of the doctest examples

**check\_output**(*want*, *got*, *optionflags*)

Return True iff the actual output from an example (*got*) matches the expected output (*want*). These strings are always considered to match if they are identical; but depending on what option flags the test runner is using, several non-exact match types are also possible. See the documentation for *TestRunner* for more information about option flags.

**class** `sympy.testing.runtests.SymPyTestResults`(*failed*, *attempted*)

**attempted**

Alias for field number 1

**failed**

Alias for field number 0

`sympy.testing.runtests.convert_to_native_paths`(*lst*)

Converts a list of '/' separated paths into a list of native (os.sep separated) paths and converts to lowercase if the system is case insensitive.

`sympy.testing.runtests.doctest`(\**paths*, *subprocess=True*, *rerun=0*, \*\**kwargs*)

Runs doctests in all \*.py files in the SymPy directory which match any of the given strings in *paths* or all tests if *paths=[]*.

Notes:

- Paths can be entered in native system format or in unix, forward-slash format.
- Files that are on the blacklist can be tested by providing their path; they are only excluded if no paths are given.

## Examples

```
>>> import sympy
```

Run all tests:

```
>>> sympy.doctest()
```

Run one file:

```
>>> sympy.doctest("sympy/core/basic.py")
>>> sympy.doctest("polynomial.rst")
```

Run all tests in sympy/functions/ and some particular file:

```
>>> sympy.doctest("/functions", "basic.py")
```

Run any file having polynomial in its name, doc/src/modules/polynomial.rst, sympy/functions/special/polynomials.py, and sympy/polys/polynomial.py:

```
>>> sympy.doctest("polynomial")
```

The `split` option can be passed to split the test run into parts. The split currently only splits the test files, though this may change in the future. `split` should be a string of the form 'a/b', which will run part a of b. Note that the regular doctests and the Sphinx doctests are split independently. For instance, to run the first half of the test suite:

```
>>> sympy.doctest(split='1/2')
```

The `subprocess` and `verbose` options are the same as with the function `test()`. See the docstring of that function for more information.

`sympy.testing.runtests.get_sympy_dir()`

Returns the root SymPy directory and set the global value indicating whether the system is case sensitive or not.

`sympy.testing.runtests.raise_on_deprecated()`

Context manager to make DeprecationWarning raise an error

This is to catch SymPyDeprecationWarning from library code while running tests and doctests. It is important to use this context manager around each individual test/doctest in case some tests modify the warning filters.

`sympy.testing.runtests.run_all_tests(test_args=(), test_kwargs=None, doctest_args=(), doctest_kwargs=None, examples_args=(), examples_kwargs=None)`

Run all tests.

Right now, this runs the regular tests (bin/test), the doctests (bin/doctest), and the examples (examples/all.py).

This is what `setup.py test` uses.

You can pass arguments and keyword arguments to the test functions that support them (for now, `test`, `doctest`, and the examples). See the docstrings of those functions for a description of the available options.

For example, to run the solvers tests with colors turned off:



```
>>> from sympy.testing.runtests import run_all_tests
>>> run_all_tests(test_args=("solvers",),
... test_kwargs={"colors:False"})
```

`sympy.testing.runtests.run_in_subprocess_with_hash_randomization`(*function*, *function\_args*=(), *function\_kwargs*=None, *command*='/opt/hostedtoolcache/Python/3.7.3/x64/python.exe', *module*='sympy.testing.runtests', *force*=False)

Run a function in a Python subprocess with hash randomization enabled.

If hash randomization is not supported by the version of Python given, it returns False. Otherwise, it returns the exit value of the command. The function is passed to `sys.exit()`, so the return value of the function will be the return value.

The environment variable `PYTHONHASHSEED` is used to seed Python's hash randomization. If it is set, this function will return False, because starting a new subprocess is unnecessary in that case. If it is not set, one is set at random, and the tests are run. Note that if this environment variable is set when Python starts, hash randomization is automatically enabled. To force a subprocess to be created even if `PYTHONHASHSEED` is set, pass `force=True`. This flag will not force a subprocess in Python versions that do not support hash randomization (see below), because those versions of Python do not support the `-R` flag.

*function* should be a string name of a function that is importable from the module *module*, like `"_test"`. The default for *module* is `"sympy.testing.runtests"`. *function\_args* and *function\_kwargs* should be a repr-able tuple and dict, respectively. The default Python command is `sys.executable`, which is the currently running Python command.

This function is necessary because the seed for hash randomization must be set by the environment variable before Python starts. Hence, in order to use a predetermined seed for tests, we must start Python in a separate subprocess.

Hash randomization was added in the minor Python versions 2.6.8, 2.7.3, 3.1.5, and 3.2.3, and is enabled by default in all Python versions after and including 3.3.0.

## Examples

```
>>> from sympy.testing.runtests import (
... run_in_subprocess_with_hash_randomization)
>>> # run the core tests in verbose mode
>>> run_in_subprocess_with_hash_randomization("_test",
... function_args=("core",),
... function_kwargs={'verbose': True})
Will return 0 if sys.executable supports hash randomization and tests
pass, 1 if they fail, and False if it does not support hash
randomization.
```

`sympy.testing.runtests.split_list`(*l*, *split*, *density*=None)

Splits a list into part a of b

split should be a string of the form 'a/b'. For instance, '1/3' would give the split one of three.

If the length of the list is not divisible by the number of splits, the last split will have more items.

*density* may be specified as a list. If specified, tests will be balanced so that each split has as equal-as-possible amount of mass according to *density*.

```
>>> from sympy.testing.runtests import split_list
>>> a = list(range(10))
>>> split_list(a, '1/3')
[0, 1, 2]
>>> split_list(a, '2/3')
[3, 4, 5]
>>> split_list(a, '3/3')
[6, 7, 8, 9]
```

`sympy.testing.runtests.sympytestfile(filename, module_relative=True, name=None, package=None, globs=None, verbose=None, report=True, optionflags=0, extraglobs=None, raise_on_error=False, parser=<doctest.DocTestParser object>, encoding=None)`

Test examples in the given file. Return (#failures, #tests).

Optional keyword arg `module_relative` specifies how filenames should be interpreted:

- If `module_relative` is `True` (the default), then `filename` specifies a module-relative path. By default, this path is relative to the calling module's directory; but if the `package` argument is specified, then it is relative to that package. To ensure os-independence, `filename` should use "/" characters to separate path segments, and should not be an absolute path (i.e., it may not begin with "/").
- If `module_relative` is `False`, then `filename` specifies an os-specific path. The path may be absolute or relative (to the current working directory).

Optional keyword arg `name` gives the name of the test; by default use the file's basename.

Optional keyword argument `package` is a Python package or the name of a Python package whose directory should be used as the base directory for a module relative filename. If no `package` is specified, then the calling module's directory is used as the base directory for module relative filenames. It is an error to specify `package` if `module_relative` is `False`.

Optional keyword arg `globs` gives a dict to be used as the globals when executing examples; by default, use `{}`. A copy of this dict is actually used for each docstring, so that each docstring's examples start with a clean slate.

Optional keyword arg `extraglobs` gives a dictionary that should be merged into the globals that are used to execute examples. By default, no extra globals are used.

Optional keyword arg `verbose` prints lots of stuff if true, prints only failures if false; by default, it's true iff "-v" is in `sys.argv`.

Optional keyword arg `report` prints a summary at the end when true, else prints nothing at the end. In verbose mode, the summary is detailed, else very brief (in fact, empty if all tests passed).

Optional keyword arg `optionflags` or's together module constants, and defaults to 0. Possible values (see the docs for details):

- `DONT_ACCEPT_TRUE_FOR_1`
- `DONT_ACCEPT_BLANKLINE`
- `NORMALIZE_WHITESPACE`
- `ELLIPSIS`
- `SKIP`
- `IGNORE_EXCEPTION_DETAIL`
- `REPORT_UDIFF`
- `REPORT_CDIF`
- `REPORT_NDIFF`
- `REPORT_ONLY_FIRST_FAILURE`

Optional keyword arg `raise_on_error` raises an exception on the first unexpected exception or failure. This allows failures to be post-mortem debugged.

Optional keyword arg `parser` specifies a `DocTestParser` (or subclass) that should be used to extract tests from the files.

Optional keyword arg `encoding` specifies an encoding that should be used to convert the file to unicode.

Advanced tomfoolery: `testmod` runs methods of a local instance of class `doctest.Tester`, then merges the results into (or creates) global `Tester` instance `doctest.master`. Methods of `doctest.master` can be called directly too, if you want to do something unusual. Passing `report=0` to `testmod` is especially useful then, to delay displaying a summary. Invoke `doctest.master.summarize(verbose)` when you're done fiddling.

`sympy.testing.runtests.test(*paths, subprocess=True, rerun=0, **kwargs)`

Run tests in the specified `test_*.py` files.

Tests in a particular `test_*.py` file are run if any of the given strings in `paths` matches a part of the test file's path. If `paths=[]`, tests in all `test_*.py` files are run.

Notes:

- If `sort=False`, tests are run in random order (not default).
- Paths can be entered in native system format or in unix, forward-slash format.
- Files that are on the blacklist can be tested by providing their path; they are only excluded if no paths are given.

### Explanation of test results

| Out-put | Meaning                                                                                                                                  |
|---------|------------------------------------------------------------------------------------------------------------------------------------------|
| .       | passed                                                                                                                                   |
| F       | failed                                                                                                                                   |
| X       | XPassed (expected to fail but passed)                                                                                                    |
| f       | XFAILed (expected to fail and indeed failed)                                                                                             |
| s       | skipped                                                                                                                                  |
| w       | slow                                                                                                                                     |
| T       | timeout (e.g., when <code>--timeout</code> is used)                                                                                      |
| K       | KeyboardInterrupt (when running the slow tests with <code>--slow</code> , you can interrupt one of them without killing the test runner) |

Colors have no additional meaning and are used just to facilitate interpreting the output.

## Examples

```
>>> import sympy
```

Run all tests:

```
>>> sympy.test()
```

Run one file:

```
>>> sympy.test("sympy/core/tests/test_basic.py")
>>> sympy.test("_basic")
```

Run all tests in sympy/functions/ and some particular file:

```
>>> sympy.test("sympy/core/tests/test_basic.py",
... "sympy/functions")
```

Run all tests in sympy/core and sympy/utilities:

```
>>> sympy.test("/core", "/util")
```

Run specific test from a file:

```
>>> sympy.test("sympy/core/tests/test_basic.py",
... kw="test_equality")
```

Run specific test from any file:

```
>>> sympy.test(kw="subs")
```

Run the tests with verbose mode on:

```
>>> sympy.test(verbose=True)
```

Do not sort the test output:

```
>>> sympy.test(sort=False)
```

Turn on post-mortem pdb:

```
>>> sympy.test(pdb=True)
```

Turn off colors:

```
>>> sympy.test(colors=False)
```

Force colors, even when the output is not to a terminal (this is useful, e.g., if you are piping to `less -r` and you still want colors)

```
>>> sympy.test(force_colors=False)
```

The traceback verbosity can be set to “short” or “no” (default is “short”)

```
>>> sympy.test(tb='no')
```

The `split` option can be passed to split the test run into parts. The split currently only splits the test files, though this may change in the future. `split` should be a string of the form ‘a/b’, which will run part a of b. For instance, to run the first half of the test suite:

```
>>> sympy.test(split='1/2')
```

The `time_balance` option can be passed in conjunction with `split`. If `time_balance=True` (the default for `sympy.test`), SymPy will attempt to split the tests such that each split takes equal time. This heuristic for balancing is based on pre-recorded test data.

```
>>> sympy.test(split='1/2', time_balance=True)
```

You can disable running the tests in a separate subprocess using `subprocess=False`. This is done to support seeding hash randomization, which is enabled by default in the Python versions where it is supported. If `subprocess=False`, hash randomization is enabled/disabled according to whether it has been enabled or not in the calling Python process. However, even if it is enabled, the seed cannot be printed unless it is called from a new Python process.

Hash randomization was added in the minor Python versions 2.6.8, 2.7.3, 3.1.5, and 3.2.3, and is enabled by default in all Python versions after and including 3.3.0.

If hash randomization is not supported `subprocess=False` is used automatically.

```
>>> sympy.test(subprocess=False)
```

To set the hash randomization seed, set the environment variable `PYTHONHASHSEED` before running the tests. This can be done from within Python using

```
>>> import os
>>> os.environ['PYTHONHASHSEED'] = '42'
```

Or from the command line using

```
$ PYTHONHASHSEED=42 ./bin/test
```

If the seed is not set, a random seed will be chosen.

Note that to reproduce the same hash values, you must use both the same seed as well as the same architecture (32-bit vs. 64-bit).

## Utilities

This module contains some general purpose utilities that are used across SymPy.

Contents:

### Autowrap Module

The autowrap module works very well in tandem with the Indexed classes of the *Tensor* (page 1387). Here is a simple example that shows how to setup a binary routine that calculates a matrix-vector product.

```
>>> from sympy.utilities.autowrap import autowrap
>>> from sympy import symbols, IndexedBase, Idx, Eq
>>> A, x, y = map(IndexedBase, ['A', 'x', 'y'])
>>> m, n = symbols('m n', integer=True)
>>> i = Idx('i', m)
>>> j = Idx('j', n)
>>> instruction = Eq(y[i], A[i, j]*x[j]); instruction
Eq(y[i], A[i, j]*x[j])
```

Because the code printers treat Indexed objects with repeated indices as a summation, the above equality instance will be translated to low-level code for a matrix vector product. This is how you tell SymPy to generate the code, compile it and wrap it as a python function:

```
>>> matvec = autowrap(instruction)
```

That's it. Now let's test it with some numpy arrays. The default wrapper backend is f2py. The wrapper function it provides is set up to accept python lists, which it will silently convert to numpy arrays. So we can test the matrix vector product like this:

```
>>> M = [[0, 1],
... [1, 0]]
>>> matvec(M, [2, 3])
[3. 2.]
```

### Implementation details

The autowrap module is implemented with a backend consisting of CodeWrapper objects. The base class CodeWrapper takes care of details about module name, filenames and options. It also contains the driver routine, which runs through all steps in the correct order, and also takes care of setting up and removing the temporary working directory.

The actual compilation and wrapping is done by external resources, such as the system installed f2py command. The Cython backend runs a distutils setup script in a subprocess. Subclasses of CodeWrapper takes care of these backend-dependent details.

## API Reference

Module for compiling codegen output, and wrap the binary for use in python.

**Note:** To use the autowrap module it must first be imported

```
>>> from sympy.utilities.autowrap import autowrap
```

This module provides a common interface for different external backends, such as f2py, fwrap, Cython, SWIG(?) etc. (Currently only f2py and Cython are implemented) The goal is to provide access to compiled binaries of acceptable performance with a one-button user interface, e.g.,

```
>>> from sympy.abc import x,y
>>> expr = (x - y)**25
>>> flat = expr.expand()
>>> binary_callable = autowrap(flat)
>>> binary_callable(2, 3)
-1.0
```

Although a SymPy user might primarily be interested in working with mathematical expressions and not in the details of wrapping tools needed to evaluate such expressions efficiently in numerical form, the user cannot do so without some understanding of the limits in the target language. For example, the expanded expression contains large coefficients which result in loss of precision when computing the expression:

```
>>> binary_callable(3, 2)
0.0
>>> binary_callable(4, 5), binary_callable(5, 4)
(-22925376.0, 25165824.0)
```

Wrapping the unexpanded expression gives the expected behavior:

```
>>> e = autowrap(expr)
>>> e(4, 5), e(5, 4)
(-1.0, 1.0)
```

The callable returned from `autowrap()` is a binary Python function, not a SymPy object. If it is desired to use the compiled function in symbolic expressions, it is better to use `binary_function()` which returns a SymPy Function object. The binary callable is attached as the `_imp_` attribute and invoked when a numerical evaluation is requested with `evalf()`, or with `lambdify()`.

```
>>> from sympy.utilities.autowrap import binary_function
>>> f = binary_function('f', expr)
>>> 2*f(x, y) + y
y + 2*f(x, y)
>>> (2*f(x, y) + y).evalf(2, subs={x: 1, y:2})
0.e-110
```

When is this useful?

- 1) For computations on large arrays, Python iterations may be too slow, and depending on the mathematical expression, it may be difficult to exploit the advanced index operations provided by NumPy.

- 2) For *really* long expressions that will be called repeatedly, the compiled binary should be significantly faster than SymPy's `.evalf()`
- 3) If you are generating code with the codegen utility in order to use it in another project, the automatic Python wrappers let you test the binaries immediately from within SymPy.
- 4) To create customized ufuncs for use with numpy arrays. See *ufuncify*.

When is this module NOT the best approach?

- 1) If you are really concerned about speed or memory optimizations, you will probably get better results by working directly with the wrapper tools and the low level code. However, the files generated by this utility may provide a useful starting point and reference code. Temporary files will be left intact if you supply the keyword `tempdir="path/to/files/"`.
- 2) If the array computation can be handled easily by numpy, and you do not need the binaries for another project.

**class** `sympy.utilities.autowrap.CodeWrapper`(*generator*, *filepath=None*, *flags=[]*,  
*verbose=False*)

Base Class for code wrappers

**class** `sympy.utilities.autowrap.CythonCodeWrapper`(\*args, \*\*kwargs)

Wrapper that uses Cython

**dump\_pyx**(*routines*, *f*, *prefix*)

Write a Cython file with Python wrappers

This file contains all the definitions of the routines in c code and refers to the header file.

## Arguments

### **routines**

List of Routine instances

### **f**

File-like object to write the file to

### **prefix**

The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

**class** `sympy.utilities.autowrap.DummyWrapper`(*generator*, *filepath=None*, *flags=[]*,  
*verbose=False*)

Class used for testing independent of backends

**class** `sympy.utilities.autowrap.F2PyCodeWrapper`(\*args, \*\*kwargs)

Wrapper that uses f2py

**class** `sympy.utilities.autowrap.UfuncifyCodeWrapper`(\*args, \*\*kwargs)

Wrapper for Ufuncify

**dump\_c**(*routines*, *f*, *prefix*, *funcname=None*)

Write a C file with Python wrappers

This file contains all the definitions of the routines in c code.



## Arguments

### **routines**

List of Routine instances

### **f**

File-like object to write the file to

### **prefix**

The filename prefix, used to name the imported module.

### **funcname**

Name of the main function to be returned.

```
sympy.utilities.autowrap.autowrap(expr, language=None, backend='f2py',
 tempdir=None, args=None, flags=None,
 verbose=False, helpers=None, code_gen=None,
 **kwargs)
```

Generates Python callable binaries based on the math expression.

### **Parameters**

#### **expr**

The SymPy expression that should be wrapped as a binary routine.

#### **language** : string, optional

If supplied, (options: 'C' or 'F95'), specifies the language of the generated code. If None [default], the language is inferred based upon the specified backend.

#### **backend** : string, optional

Backend used to wrap the generated code. Either 'f2py' [default], or 'cython'.

#### **tempdir** : string, optional

Path to directory for temporary files. If this argument is supplied, the generated code and the wrapper input files are left intact in the specified path.

#### **args** : iterable, optional

An ordered iterable of symbols. Specifies the argument sequence for the function.

#### **flags** : iterable, optional

Additional option flags that will be passed to the backend.

#### **verbose** : bool, optional

If True, autowrap will not mute the command line backends. This can be helpful for debugging.

#### **helpers** : 3-tuple or iterable of 3-tuples, optional

Used to define auxiliary expressions needed for the main expr. If the main expression needs to call a specialized function it should be passed in via helpers. Autowrap will then make sure that the compiled main expression can link to the helper routine. Items should

be 3-tuples with (<function\_name>, <sympy\_expression>, <argument\_tuple>). It is mandatory to supply an argument sequence to helper routines.

**code\_gen** : CodeGen instance

An instance of a CodeGen subclass. Overrides language.

**include\_dirs** : [string]

A list of directories to search for C/C++ header files (in Unix form for portability).

**library\_dirs** : [string]

A list of directories to search for C/C++ libraries at link time.

**libraries** : [string]

A list of library names (not filenames or paths) to link against.

**extra\_compile\_args** : [string]

Any extra platform- and compiler-specific information to use when compiling the source files in 'sources'. For platforms and compilers where "command line" makes sense, this is typically a list of command-line arguments, but for other platforms it could be anything.

**extra\_link\_args** : [string]

Any extra platform- and compiler-specific information to use when linking object files together to create the extension (or to create a new static Python interpreter). Similar interpretation as for 'extra\_compile\_args'.

## Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.utilities.autowrap import autowrap
>>> expr = ((x - y + z)**(13)).expand()
>>> binary_func = autowrap(expr)
>>> binary_func(1, 4, 2)
-1.0
```

`sympy.utilities.autowrap.binary_function(symfunc, expr, **kwargs)`

Returns a SymPy function with `expr` as binary implementation

This is a convenience function that automates the steps needed to autowrap the SymPy expression and attaching it to a Function object with `implemented_function()`.

### Parameters

**symfunc** : SymPy Function

The function to bind the callable to.

**expr** : SymPy Expression

The expression used to generate the function.

**kwargs** : dict

Any kwargs accepted by autowrap.

## Examples

```
>>> from sympy.abc import x, y
>>> from sympy.utilities.autowrap import binary_function
>>> expr = ((x - y)**(25)).expand()
>>> f = binary_function('f', expr)
>>> type(f)
<class 'sympy.core.function.UndefinedFunction'>
>>> 2*f(x, y)
2*f(x, y)
>>> f(x, y).evalf(2, subs={x: 1, y: 2})
-1.0
```

```
sympy.utilities.autowrap.ufuncify(args, expr, language=None, backend='numpy',
 tempdir=None, flags=None, verbose=False,
 helpers=None, **kwargs)
```

Generates a binary function that supports broadcasting on numpy arrays.

### Parameters

**args** : iterable

Either a Symbol or an iterable of symbols. Specifies the argument sequence for the function.

**expr**

A SymPy expression that defines the element wise operation.

**language** : string, optional

If supplied, (options: 'C' or 'F95'), specifies the language of the generated code. If None [default], the language is inferred based upon the specified backend.

**backend** : string, optional

Backend used to wrap the generated code. Either 'numpy' [default], 'cython', or 'f2py'.

**tempdir** : string, optional

Path to directory for temporary files. If this argument is supplied, the generated code and the wrapper input files are left intact in the specified path.

**flags** : iterable, optional

Additional option flags that will be passed to the backend.

**verbose** : bool, optional

If True, autowrap will not mute the command line backends. This can be helpful for debugging.

**helpers** : iterable, optional

Used to define auxiliary expressions needed for the main expr. If the main expression needs to call a specialized function it should be put in

the helpers iterable. Autowrap will then make sure that the compiled main expression can link to the helper routine. Items should be tuples with (<function\_name>, <sympy\_expression>, <arguments>). It is mandatory to supply an argument sequence to helper routines.

**kwargs** : dict

These kwargs will be passed to autowrap if the *f2py* or *cython* backend is used and ignored if the *numpy* backend is used.

## Notes

The default backend ('numpy') will create actual instances of `numpy.ufunc`. These support ndimensional broadcasting, and implicit type conversion. Use of the other backends will result in a "ufunc-like" function, which requires equal length 1-dimensional arrays for all arguments, and will not perform any type conversions.

## Examples

```
>>> from sympy.utilities.autowrap import ufuncify
>>> from sympy.abc import x, y
>>> import numpy as np
>>> f = ufuncify((x, y), y + x**2)
>>> type(f)
<class 'numpy.ufunc'>
>>> f([1, 2, 3], 2)
array([3., 6., 11.])
>>> f(np.arange(5), 3)
array([3., 4., 7., 12., 19.])
```

For the 'f2py' and 'cython' backends, inputs are required to be equal length 1-dimensional arrays. The 'f2py' backend will perform type conversion, but the Cython backend will error if the inputs are not of the expected type.

```
>>> f_fortran = ufuncify((x, y), y + x**2, backend='f2py')
>>> f_fortran(1, 2)
array([3.])
>>> f_fortran(np.array([1, 2, 3]), np.array([1.0, 2.0, 3.0]))
array([2., 6., 12.])
>>> f_cython = ufuncify((x, y), y + x**2, backend='Cython')
>>> f_cython(1, 2)
Traceback (most recent call last):
...
TypeError: Argument '_x' has incorrect type (expected numpy.ndarray, got
→int)
>>> f_cython(np.array([1.0]), np.array([2.0]))
array([3.])
```

## References

[R957]

## Codegen

This module provides functionality to generate directly compilable code from SymPy expressions. The codegen function is the user interface to the code generation functionality in SymPy. Some details of the implementation is given below for advanced users that may want to use the framework directly.

---

**Note:** The codegen callable is not in the sympy namespace automatically, to use it you must first execute

```
>>> from sympy.utilities.codegen import codegen
```

---

## Implementation Details

Here we present the most important pieces of the internal structure, as advanced users may want to use it directly, for instance by subclassing a code generator for a specialized application. **It is very likely that you would prefer to use the codegen() function documented above.**

Basic assumptions:

- A generic Routine data structure describes the routine that must be translated into C/Fortran/... code. This data structure covers all features present in one or more of the supported languages.
- Descendants from the CodeGen class transform multiple Routine instances into compilable code. Each derived class translates into a specific language.
- In many cases, one wants a simple workflow. The friendly functions in the last part are a simple api on top of the Routine/CodeGen stuff. They are easier to use, but are less powerful.

## Routine

The Routine class is a very important piece of the codegen module. Viewing the codegen utility as a translator of mathematical expressions into a set of statements in a programming language, the Routine instances are responsible for extracting and storing information about how the math can be encapsulated in a function call. Thus, it is the Routine constructor that decides what arguments the routine will need and if there should be a return value.

## API Reference

module for generating C, C++, Fortran77, Fortran90, Julia, Rust and Octave/Matlab routines that evaluate SymPy expressions. This module is work in progress. Only the milestones with a '+' character in the list below have been completed.

— How is `sympy.utilities.codegen` different from `sympy.printing.ccode`? —

We considered the idea to extend the printing routines for SymPy functions in such a way that it prints complete compilable code, but this leads to a few unsurmountable issues that can only be tackled with dedicated code generator:

- For C, one needs both a code and a header file, while the printing routines generate just one string. This code generator can be extended to support `.pyf` files for `f2py`.
- SymPy functions are not concerned with programming-technical issues, such as input, output and input-output arguments. Other examples are contiguous or non-contiguous arrays, including headers of other libraries such as `gsl` or others.
- It is highly interesting to evaluate several SymPy functions in one C routine, eventually sharing common intermediate results with the help of the `cse` routine. This is more than just printing.
- From the programming perspective, expressions with constants should be evaluated in the code generator as much as possible. This is different for printing.

— Basic assumptions —

- A generic Routine data structure describes the routine that must be translated into C/Fortran/... code. This data structure covers all features present in one or more of the supported languages.
- Descendants from the `CodeGen` class transform multiple Routine instances into compilable code. Each derived class translates into a specific language.
- In many cases, one wants a simple workflow. The friendly functions in the last part are a simple api on top of the Routine/CodeGen stuff. They are easier to use, but are less powerful.

— Milestones —

- First working version with scalar input arguments, generating C code, tests
- Friendly functions that are easier to use than the rigorous Routine/CodeGen workflow.
- Integer and Real numbers as input and output
- Output arguments
- InputOutput arguments
- Sort input/output arguments properly
- Contiguous array arguments (numpy matrices)
- Also generate `.pyf` code for `f2py` (in `autowrap` module)
- Isolate constants and evaluate them beforehand in double precision
- Fortran 90
- Octave/Matlab
- Common Subexpression Elimination
- User defined comments in the generated code

- Optional extra include lines for libraries/objects that can eval special functions
- Test other C compilers and libraries: gcc, tcc, libtcc, gcc+gsl, ...
- Contiguous array arguments (SymPy matrices)
- Non-contiguous array arguments (SymPy matrices)
- ccode must raise an error when it encounters something that cannot be translated into c. ccode(integrate(sin(x)/x, x)) does not make sense.
- Complex numbers as input and output
- A default complex datatype
- Include extra information in the header: date, user, hostname, sha1 hash, ...
- Fortran 77
- C++
- Python
- Julia
- Rust
- ...

**class** sympy.utilities.codegen.**Argument**(*name, datatype=None, dimensions=None, precision=None*)

An abstract Argument data structure: a name and a data type.

This structure is refined in the descendants below.

**class** sympy.utilities.codegen.**CCodeGen**(*project='project', printer=None, preprocessor\_statements=None, cse=False*)

Generator for C code.

The .write() method inherited from CodeGen will output a code file and an interface file, <prefix>.c and <prefix>.h respectively.

**dump\_c**(*routines, f, prefix, header=True, empty=True*)

Write the code by calling language specific methods.

The generated file contains all the definitions of the routines in low-level code and refers to the header file if appropriate.

#### Parameters

**routines** : list

A list of Routine instances.

**f** : file-like

Where to write the file.

**prefix** : string

The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

**header** : bool, optional

When True, a header comment is included on top of each source file.  
[default : True]

**empty** : bool, optional

When True, empty lines are included to structure the source files.  
[default : True]

**dump\_h**(*routines*, *f*, *prefix*, *header=True*, *empty=True*)

Writes the C header file.

This file contains all the function declarations.

**Parameters**

**routines** : list

A list of Routine instances.

**f** : file-like

Where to write the file.

**prefix** : string

The filename prefix, used to construct the include guards. Only the  
basename of the prefix is used.

**header** : bool, optional

When True, a header comment is included on top of each source file.  
[default : True]

**empty** : bool, optional

When True, empty lines are included to structure the source files.  
[default : True]

**get\_prototype**(*routine*)

Returns a string for the function prototype of the routine.

If the routine has multiple result objects, an `CodeGenError` is raised.

See: [https://en.wikipedia.org/wiki/Function\\_prototype](https://en.wikipedia.org/wiki/Function_prototype)

**class** `sympy.utilities.codegen.CodeGen`(*project='project'*, *cse=False*)

Abstract class for the code generators.

**dump\_code**(*routines*, *f*, *prefix*, *header=True*, *empty=True*)

Write the code by calling language specific methods.

The generated file contains all the definitions of the routines in low-level code and  
refers to the header file if appropriate.

**Parameters**

**routines** : list

A list of Routine instances.

**f** : file-like

Where to write the file.

**prefix** : string

The filename prefix, used to refer to the proper header file. Only the  
basename of the prefix is used.

**header** : bool, optional



When True, a header comment is included on top of each source file.  
[default : True]

**empty** : bool, optional

When True, empty lines are included to structure the source files.  
[default : True]

**routine**(*name, expr, argument\_sequence=None, global\_vars=None*)

Creates an Routine object that is appropriate for this language.

This implementation is appropriate for at least C/Fortran. Subclasses can override this if necessary.

Here, we assume at most one return value (the l-value) which must be scalar. Additional outputs are OutputArguments (e.g., pointers on right-hand-side or pass-by-reference). Matrices are always returned via OutputArguments. If *argument\_sequence* is None, arguments will be ordered alphabetically, but with all InputArguments first, and then OutputArgument and InOutArguments.

**write**(*routines, prefix, to\_files=False, header=True, empty=True*)

Writes all the source code files for the given routines.

The generated source is returned as a list of (filename, contents) tuples, or is written to files (see below). Each filename consists of the given prefix, appended with an appropriate extension.

#### Parameters

**routines** : list

A list of Routine instances to be written

**prefix** : string

The prefix for the output files

**to\_files** : bool, optional

When True, the output is written to files. Otherwise, a list of (filename, contents) tuples is returned. [default: False]

**header** : bool, optional

When True, a header comment is included on top of each source file.  
[default: True]

**empty** : bool, optional

When True, empty lines are included to structure the source files.  
[default: True]

**class** sympy.utilities.codegen.**DataType**(*cname, fname, pyname, jlname, octname, rsname*)

Holds strings for a certain datatype in different languages.

**class** sympy.utilities.codegen.**FCodeGen**(*project='project', printer=None*)

Generator for Fortran 95 code

The .write() method inherited from CodeGen will output a code file and an interface file, <prefix>.f90 and <prefix>.h respectively.

**dump\_f95**(*routines*, *f*, *prefix*, *header=True*, *empty=True*)

Write the code by calling language specific methods.

The generated file contains all the definitions of the routines in low-level code and refers to the header file if appropriate.

**Parameters**

**routines** : list

A list of Routine instances.

**f** : file-like

Where to write the file.

**prefix** : string

The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

**header** : bool, optional

When True, a header comment is included on top of each source file.  
[default : True]

**empty** : bool, optional

When True, empty lines are included to structure the source files.  
[default : True]

**dump\_h**(*routines*, *f*, *prefix*, *header=True*, *empty=True*)

Writes the interface to a header file.

This file contains all the function declarations.

**Parameters**

**routines** : list

A list of Routine instances.

**f** : file-like

Where to write the file.

**prefix** : string

The filename prefix.

**header** : bool, optional

When True, a header comment is included on top of each source file.  
[default : True]

**empty** : bool, optional

When True, empty lines are included to structure the source files.  
[default : True]

**get\_interface**(*routine*)

Returns a string for the function interface.

The routine should have a single result object, which can be None. If the routine has multiple result objects, a CodeGenError is raised.

See: [https://en.wikipedia.org/wiki/Function\\_prototype](https://en.wikipedia.org/wiki/Function_prototype)

**class** sympy.utilities.codegen.**JuliaCodeGen**(*project='project', printer=None*)

Generator for Julia code.

The .write() method inherited from CodeGen will output a code file <prefix>.jl.

**dump\_jl**(*routines, f, prefix, header=True, empty=True*)

Write the code by calling language specific methods.

The generated file contains all the definitions of the routines in low-level code and refers to the header file if appropriate.

**Parameters**

**routines** : list

A list of Routine instances.

**f** : file-like

Where to write the file.

**prefix** : string

The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

**header** : bool, optional

When True, a header comment is included on top of each source file.  
[default : True]

**empty** : bool, optional

When True, empty lines are included to structure the source files.  
[default : True]

**routine**(*name, expr, argument\_sequence, global\_vars*)

Specialized Routine creation for Julia.

**class** sympy.utilities.codegen.**OctaveCodeGen**(*project='project', printer=None*)

Generator for Octave code.

The .write() method inherited from CodeGen will output a code file <prefix>.m.

Octave .m files usually contain one function. That function name should match the file-name (prefix). If you pass multiple name\_expr pairs, the latter ones are presumed to be private functions accessed by the primary function.

You should only pass inputs to argument\_sequence: outputs are ordered according to their order in name\_expr.

**dump\_m**(*routines, f, prefix, header=True, empty=True, inline=True*)

Write the code by calling language specific methods.

The generated file contains all the definitions of the routines in low-level code and refers to the header file if appropriate.

**Parameters**

**routines** : list

A list of Routine instances.

**f** : file-like

Where to write the file.

**prefix** : string

The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

**header** : bool, optional

When True, a header comment is included on top of each source file.  
[default : True]

**empty** : bool, optional

When True, empty lines are included to structure the source files.  
[default : True]

**routine**(*name, expr, argument\_sequence, global\_vars*)

Specialized Routine creation for Octave.

**class** sympy.utilities.codegen.**OutputArgument**(*name, result\_var, expr,*  
*datatype=None, dimensions=None,*  
*precision=None*)

OutputArgument are always initialized in the routine.

**class** sympy.utilities.codegen.**Result**(*expr, name=None, result\_var=None,*  
*datatype=None, dimensions=None,*  
*precision=None*)

An expression for a return value.

The name result is used to avoid conflicts with the reserved word “return” in the Python language. It is also shorter than ReturnValue.

These may or may not need a name in the destination (e.g., “return(x\*y)” might return a value without ever naming it).

**class** sympy.utilities.codegen.**Routine**(*name, arguments, results, local\_vars,*  
*global\_vars*)

Generic description of evaluation routine for set of expressions.

A CodeGen class can translate instances of this class into code in a particular language. The routine specification covers all the features present in these languages. The CodeGen part must raise an exception when certain features are not present in the target language. For example, multiple return values are possible in Python, but not in C or Fortran. Another example: Fortran and Python support complex numbers, while C does not.

**property result\_variables**

Returns a list of OutputArgument, InOutArgument and Result.

If return values are present, they are at the end of the list.

**property variables**

Returns a set of all variables possibly used in the routine.

For routines with unnamed return values, the dummies that may or may not be used will be included in the set.

**class** sympy.utilities.codegen.**RustCodeGen**(*project='project', printer=None*)

Generator for Rust code.

The .write() method inherited from CodeGen will output a code file <prefix>.rs

**dump\_rs**(*routines*, *f*, *prefix*, *header=True*, *empty=True*)

Write the code by calling language specific methods.

The generated file contains all the definitions of the routines in low-level code and refers to the header file if appropriate.

#### Parameters

**routines** : list

A list of Routine instances.

**f** : file-like

Where to write the file.

**prefix** : string

The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

**header** : bool, optional

When True, a header comment is included on top of each source file.  
[default : True]

**empty** : bool, optional

When True, empty lines are included to structure the source files.  
[default : True]

**get\_prototype**(*routine*)

Returns a string for the function prototype of the routine.

If the routine has multiple result objects, an CodeGenError is raised.

See: [https://en.wikipedia.org/wiki/Function\\_prototype](https://en.wikipedia.org/wiki/Function_prototype)

**routine**(*name*, *expr*, *argument\_sequence*, *global\_vars*)

Specialized Routine creation for Rust.

`sympy.utilities.codegen.codegen`(*name\_expr*, *language=None*, *prefix=None*,  
*project='project'*, *to\_files=False*, *header=True*,  
*empty=True*, *argument\_sequence=None*,  
*global\_vars=None*, *standard=None*, *code\_gen=None*,  
*printer=None*)

Generate source code for expressions in a given language.

#### Parameters

**name\_expr** : tuple, or list of tuples

A single (name, expression) tuple or a list of (name, expression) tuples. Each tuple corresponds to a routine. If the expression is an equality (an instance of class Equality) the left hand side is considered an output argument. If expression is an iterable, then the routine will have multiple outputs.

**language** : string,

A string that indicates the source code language. This is case insensitive. Currently, 'C', 'F95' and 'Octave' are supported. 'Octave' generates code compatible with both Octave and Matlab.

**prefix** : string, optional

A prefix for the names of the files that contain the source code. Language-dependent suffixes will be appended. If omitted, the name of the first `name_expr` tuple is used.

**project** : string, optional

A project name, used for making unique preprocessor instructions. [default: "project"]

**to\_files** : bool, optional

When True, the code will be written to one or more files with the given prefix, otherwise strings with the names and contents of these files are returned. [default: False]

**header** : bool, optional

When True, a header is written on top of each source file. [default: True]

**empty** : bool, optional

When True, empty lines are used to structure the code. [default: True]

**argument\_sequence** : iterable, optional

Sequence of arguments for the routine in a preferred order. A `CodeGenError` is raised if required arguments are missing. Redundant arguments are used without warning. If omitted, arguments will be ordered alphabetically, but with all input arguments first, and then output or in-out arguments.

**global\_vars** : iterable, optional

Sequence of global variables used by the routine. Variables listed here will not show up as function arguments.

**standard** : string

**code\_gen** : CodeGen instance

An instance of a CodeGen subclass. Overrides `language`.

## Examples

```
>>> from sympy.utilities.codegen import codegen
>>> from sympy.abc import x, y, z
>>> [(c_name, c_code), (h_name, c_header)] = codegen(
... ("f", x+y*z), "C89", "test", header=False, empty=False)
>>> print(c_name)
test.c
>>> print(c_code)
#include "test.h"
#include <math.h>
double f(double x, double y, double z) {
 double f_result;
 f_result = x + y*z;
 return f_result;
```

(continues on next page)

(continued from previous page)

```

}

>>> print(h_name)
test.h
>>> print(c_header)
#ifndef PROJECT__TEST__H
#define PROJECT__TEST__H
double f(double x, double y, double z);
#endif

```

Another example using Equality objects to give named outputs. Here the filename (prefix) is taken from the first (name, expr) pair.

```

>>> from sympy.abc import f, g
>>> from sympy import Eq
>>> [(c_name, c_code), (h_name, c_header)] = codegen(
... [("myfcn", x + y), ("fcn2", [Eq(f, 2*x), Eq(g, y)])],
... "C99", header=False, empty=False)
>>> print(c_name)
myfcn.c
>>> print(c_code)
#include "myfcn.h"
#include <math.h>
double myfcn(double x, double y) {
 double myfcn_result;
 myfcn_result = x + y;
 return myfcn_result;
}
void fcn2(double x, double y, double *f, double *g) {
 (*f) = 2*x;
 (*g) = y;
}

```

If the generated function(s) will be part of a larger project where various global variables have been defined, the 'global\_vars' option can be used to remove the specified variables from the function signature

```

>>> from sympy.utilities.codegen import codegen
>>> from sympy.abc import x, y, z
>>> [(f_name, f_code), header] = codegen(
... ("f", x+y*z), "F95", header=False, empty=False,
... argument_sequence=(x, y), global_vars=(z,))
>>> print(f_code)
REAL*8 function f(x, y)
implicit none
REAL*8, intent(in) :: x
REAL*8, intent(in) :: y
f = x + y*z
end function

```

`sympy.utilities.codegen.get_default_datatype(expr, complex_allowed=None)`

Derives an appropriate datatype based on the expression.

`sympy.utilities.codegen.make_routine(name, expr, argument_sequence=None, global_vars=None, language='F95')`

A factory that makes an appropriate Routine from an expression.

#### Parameters

**name** : string

The name of this routine in the generated code.

**expr** : expression or list/tuple of expressions

A SymPy expression that the Routine instance will represent. If given a list or tuple of expressions, the routine will be considered to have multiple return values and/or output arguments.

**argument\_sequence** : list or tuple, optional

List arguments for the routine in a preferred order. If omitted, the results are language dependent, for example, alphabetical order or in the same order as the given expressions.

**global\_vars** : iterable, optional

Sequence of global variables used by the routine. Variables listed here will not show up as function arguments.

**language** : string, optional

Specify a target language. The Routine itself should be language-agnostic but the precise way one is created, error checking, etc depend on the language. [default: "F95"].

#### Notes

A decision about whether to use output arguments or return values is made depending on both the language and the particular mathematical expressions. For an expression of type Equality, the left hand side is typically made into an OutputArgument (or perhaps an InOutArgument if appropriate). Otherwise, typically, the calculated expression is made a return values of the routine.

#### Examples

```
>>> from sympy.utilities.codegen import make_routine
>>> from sympy.abc import x, y, f, g
>>> from sympy import Eq
>>> r = make_routine('test', [Eq(f, 2*x), Eq(g, x + y)])
>>> [arg.result_var for arg in r.results]
[]
>>> [arg.name for arg in r.arguments]
[x, y, f, g]
>>> [arg.name for arg in r.result_variables]
[f, g]
>>> r.local_vars
set()
```



Another more complicated example with a mixture of specified and automatically-assigned names. Also has Matrix output.

```
>>> from sympy import Matrix
>>> r = make_routine('fcn', [x*y, Eq(f, 1), Eq(g, x + g), Matrix([[x,
↪ 2]])])
>>> [arg.result_var for arg in r.results]
[result_5397460570204848505]
>>> [arg.expr for arg in r.results]
[x*y]
>>> [arg.name for arg in r.arguments]
[x, y, f, g, out_8598435338387848786]
```

We can examine the various arguments more closely:

```
>>> from sympy.utilities.codegen import (InputArgument, OutputArgument,
... InOutArgument)
>>> [a.name for a in r.arguments if isinstance(a, InputArgument)]
[x, y]
```

```
>>> [a.name for a in r.arguments if isinstance(a, OutputArgument)]
[f, out_8598435338387848786]
>>> [a.expr for a in r.arguments if isinstance(a, OutputArgument)]
[1, Matrix([[x, 2]])]
```

```
>>> [a.name for a in r.arguments if isinstance(a, InOutArgument)]
[g]
>>> [a.expr for a in r.arguments if isinstance(a, InOutArgument)]
[g + x]
```

## Decorator

Useful utility decorators.

`@sympy.utilities.decorator.deprecated`(*message*, \*, *deprecated\_since\_version*,  
*active\_deprecations\_target*, *stacklevel*=3)

Mark a function as deprecated.

This decorator should be used if an entire function or class is deprecated. If only a certain functionality is deprecated, you should use `warns_deprecated_sympy()` (page 2029) directly. This decorator is just a convenience. There is no functional difference between using this decorator and calling `warns_deprecated_sympy()` at the top of the function.

The decorator takes the same arguments as `warns_deprecated_sympy()` (page 2029). See its documentation for details on what the keywords to this decorator do.

See the [Deprecation Policy](#) (page 3017) document for details on when and how things should be deprecated in SymPy.

## Examples

```
>>> from sympy.utilities.decorator import deprecated
>>> from sympy import simplify
>>> @deprecated(""" ... The simplify_this(expr) function is
↳ deprecated. Use simplify(expr)
... instead.""", deprecated_since_version="1.1",
... active_deprecations_target='simplify-this-deprecation')
... def simplify_this(expr):
... """
... Simplify ``expr``.
...
... .. deprecated:: 1.1
...
... The ``simplify_this`` function is deprecated. Use
↳ :func:`simplify`
... instead. See its documentation for more information. See
... :ref:`simplify-this-deprecation` for details.
...
... """
... return simplify(expr)
>>> from sympy.abc import x
>>> simplify_this(x*(x + 1) - x**2)
<stdin>:1: SymPyDeprecationWarning:
```

The `simplify_this(expr)` function is deprecated. Use `simplify(expr)` instead.

See <https://docs.sympy.org/latest/explanation/active-deprecations.html#simplify-this-deprecation> for details.

This has been deprecated since SymPy version 1.1. It will be removed in a future version of SymPy.

```
simplify_this(x)
x
```

### See also:

[`sympy.utilities.exceptions.SymPyDeprecationWarning`](#) (page 2066), [`sympy.utilities.exceptions.sympy\_deprecation\_warning`](#) (page 2067), [`sympy.utilities.exceptions.ignore\_warnings`](#) (page 2067), [`sympy.testing.pytest.warns\_deprecated\_sympy`](#) (page 2029)

`sympy.utilities.decorator.conserve_mpmath_dps(func)`

After the function finishes, resets the value of `mpmath.mp.dps` to the value it had before the function was run.

`sympy.utilities.decorator.doctest_depends_on(exe=None, modules=None, disable_viewers=None, python_version=None)`

Adds metadata about the dependencies which need to be met for doctesting the docstrings of the decorated objects.

`exe` should be a list of executables

`modules` should be a list of modules

`disable_viewers` should be a list of viewers for `preview()` to disable

`python_version` should be the minimum Python version required, as a tuple (like (3, 0))

`sympy.utilities.decorator.memoize_property(propfunc)`

Property decorator that caches the value of potentially expensive *propfunc* after the first evaluation. The cached value is stored in the corresponding property name with an attached underscore.

`class sympy.utilities.decorator.no_attrs_in_subclass(cls, f)`

Don't 'inherit' certain attributes from a base class

```
>>> from sympy.utilities.decorator import no_attrs_in_subclass
```

```
>>> class A(object):
... x = 'test'
```

```
>>> A.x = no_attrs_in_subclass(A, A.x)
```

```
>>> class B(A):
... pass
```

```
>>> hasattr(A, 'x')
True
>>> hasattr(B, 'x')
False
```

`sympy.utilities.decorator.public(obj)`

Append `obj`'s name to global `__all__` variable (call site).

By using this decorator on functions or classes you achieve the same goal as by filling `__all__` variables manually, you just do not have to repeat yourself (object's name). You also know if object is public at definition site, not at some random location (where `__all__` was set).

Note that in multiple decorator setup (in almost all cases) `@public` decorator must be applied before any other decorators, because it relies on the pointer to object's global namespace. If you apply other decorators first, `@public` may end up modifying the wrong namespace.

## Examples

```
>>> from sympy.utilities.decorator import public
```

```
>>> __all__ # noqa: F821
Traceback (most recent call last):
...
NameError: name '__all__' is not defined
```

```
>>> @public
... def some_function():
... pass
```

```
>>> __all__ # noqa: F821
['some_function']
```

`sympy.utilities.decorator.threaded(func)`

Apply `func` to sub-elements of an object, including [Add](#) (page 1013).

This decorator is intended to make it uniformly possible to apply a function to all elements of composite objects, e.g. matrices, lists, tuples and other iterable containers, or just expressions.

This version of `threaded()` (page 2060) decorator allows threading over elements of [Add](#) (page 1013) class. If this behavior is not desirable use `xthreaded()` (page 2060) decorator.

Functions using this decorator must have the following signature:

```
@threaded
def function(expr, *args, **kwargs):
```

`sympy.utilities.decorator.threaded_factory(func, use_add)`

A factory for threaded decorators.

`sympy.utilities.decorator.xthreaded(func)`

Apply `func` to sub-elements of an object, excluding [Add](#) (page 1013).

This decorator is intended to make it uniformly possible to apply a function to all elements of composite objects, e.g. matrices, lists, tuples and other iterable containers, or just expressions.

This version of `threaded()` (page 2060) decorator disallows threading over elements of [Add](#) (page 1013) class. If this behavior is not desirable use `threaded()` (page 2060) decorator.

Functions using this decorator must have the following signature:

```
@xthreaded
def function(expr, *args, **kwargs):
```

## Enumerative

This module includes functions and classes for enumerating and counting multiset partitions.

`sympy.utilities.enumerative.multiset_partitions_taocp(multiplicities)`

Enumerates partitions of a multiset.

### Parameters

#### **multiplicities**

list of integer multiplicities of the components of the multiset.

### Yields

state

Internal data structure which encodes a particular partition. This output is then usually processed by a visitor function which combines the information from this data structure with the components themselves to produce an actual partition.

Unless they wish to create their own visitor function, users will have little need to look inside this data structure. But, for reference, it is a 3-element list with components:

**f**

is a frame array, which is used to divide pstack into parts.

**lpart**

points to the base of the topmost part.

**pstack**

is an array of PartComponent objects.

The state output offers a peek into the internal data structures of the enumeration function. The client should treat this as read-only; any modification of the data structure will cause unpredictable (and almost certainly incorrect) results. Also, the components of state are modified in place at each iteration. Hence, the visitor must be called at each loop iteration. Accumulating the state instances and processing them later will not work.

## Examples

```
>>> from sympy.utilities.enumerative import list_visitor
>>> from sympy.utilities.enumerative import multiset_partitions_taocp
>>> # variables components and multiplicities represent the multiset 'abb'
↪
>>> components = 'ab'
>>> multiplicities = [1, 2]
>>> states = multiset_partitions_taocp(multiplicities)
>>> list(list_visitor(state, components) for state in states)
[[['a', 'b', 'b']],
 [['a', 'b'], ['b']],
 [['a'], ['b', 'b']],
 [['a'], ['b'], ['b']]]
```

**See also:**

[\*sympy.utilities.iterables.multiset\\_partitions\*](#) (page 2084)

Takes a multiset as input and directly yields multiset partitions. It dispatches to a number of functions, including this one, for implementation. Most users will find it more convenient to use than multiset\_partitions\_taocp.

`sympy.utilities.enumerative.factoring_visitor(state, primes)`

Use with multiset\_partitions\_taocp to enumerate the ways a number can be expressed as a product of factors. For this usage, the exponents of the prime factors of a number are arguments to the partition enumerator, while the corresponding prime factors are input here.

## Examples

To enumerate the factorings of a number we can think of the elements of the partition as being the prime factors and the multiplicities as being their exponents.

```
>>> from sympy.utilities.enumerative import factoring_visitor
>>> from sympy.utilities.enumerative import multiset_partitions_taocp
>>> from sympy import factorint
>>> primes, multiplicities = zip(*factorint(24).items())
>>> primes
(2, 3)
>>> multiplicities
(3, 1)
>>> states = multiset_partitions_taocp(multiplicities)
>>> list(factoring_visitor(state, primes) for state in states)
[[24], [8, 3], [12, 2], [4, 6], [4, 2, 3], [6, 2, 2], [2, 2, 2, 3]]
```

`sympy.utilities.enumerative.list_visitor(state, components)`

Return a list of lists to represent the partition.

## Examples

```
>>> from sympy.utilities.enumerative import list_visitor
>>> from sympy.utilities.enumerative import multiset_partitions_taocp
>>> states = multiset_partitions_taocp([1, 2, 1])
>>> s = next(states)
>>> list_visitor(s, 'abc') # for multiset 'a b b c'
[['a', 'b', 'b', 'c']]
>>> s = next(states)
>>> list_visitor(s, [1, 2, 3]) # for multiset '1 2 2 3'
[[1, 2, 2], [3]]
```

The approach of the function `multiset_partitions_taocp` is extended and generalized by the class `MultisetPartitionTraverser`.

**class** `sympy.utilities.enumerative.MultisetPartitionTraverser`

Has methods to enumerate and count the partitions of a multiset.

This implements a refactored and extended version of Knuth's algorithm 7.1.2.5M [AOCP]."

The enumeration methods of this class are generators and return data structures which can be interpreted by the same visitor functions used for the output of `multiset_partitions_taocp`.

## Examples

```
>>> from sympy.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> m.count_partitions([4,4,4,2])
127750
>>> m.count_partitions([3,3,3])
686
```

### See also:

[\*multiset\\_partitions\\_taocp\*](#) (page 2060), [\*sympy.utilities.iterables.multiset\\_partitions\*](#) (page 2084)

## References

[AOCP], [Factorisatio], [Yorgey]

### `count_partitions(multiplicities)`

Returns the number of partitions of a multiset whose components have the multiplicities given in *multiplicities*.

For larger counts, this method is much faster than calling one of the enumerators and counting the result. Uses dynamic programming to cut down on the number of nodes actually explored. The dictionary used in order to accelerate the counting process is stored in the `MultisetPartitionTraverser` object and persists across calls. If the user does not expect to call `count_partitions` for any additional multisets, the object should be cleared to save memory. On the other hand, the cache built up from one count run can significantly speed up subsequent calls to `count_partitions`, so it may be advantageous not to clear the object.

## Examples

```
>>> from sympy.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> m.count_partitions([9,8,2])
288716
>>> m.count_partitions([2,2])
9
>>> del m
```

## Notes

If one looks at the workings of Knuth's algorithm M [AOCP], it can be viewed as a traversal of a binary tree of parts. A part has (up to) two children, the left child resulting from the spread operation, and the right child from the decrement operation. The ordinary enumeration of multiset partitions is an in-order traversal of this tree, and with the partitions corresponding to paths from the root to the leaves. The mapping from paths to partitions is a little complicated, since the partition would contain only those parts which are leaves or the parents of a spread link, not those which are parents of a decrement link.

For counting purposes, it is sufficient to count leaves, and this can be done with a recursive in-order traversal. The number of leaves of a subtree rooted at a particular part is a function only of that part itself, so memoizing has the potential to speed up the counting dramatically.

This method follows a computational approach which is similar to the hypothetical memoized recursive function, but with two differences:

- 1) This method is iterative, borrowing its structure from the other enumerations and maintaining an explicit stack of parts which are in the process of being counted. (There may be multisets which can be counted reasonably quickly by this implementation, but which would overflow the default Python recursion limit with a recursive implementation.)
- 2) Instead of using the part data structure directly, a more compact key is constructed. This saves space, but more importantly coalesces some parts which would remain separate with physical keys.

Unlike the enumeration functions, there is currently no `_range` version of `count_partitions`. If someone wants to stretch their brain, it should be possible to construct one by memoizing with a histogram of counts rather than a single count, and combining the histograms.

**enum\_all**(*multiplicities*)

Enumerate the partitions of a multiset.

### Examples

```
>>> from sympy.utilities.enumerative import list_visitor
>>> from sympy.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> states = m.enum_all([2,2])
>>> list(list_visitor(state, 'ab') for state in states)
[[['a', 'a', 'b', 'b']],
 [['a', 'a', 'b'], ['b']],
 [['a', 'a'], ['b', 'b']],
 [['a', 'a'], ['b'], ['b']],
 [['a', 'b', 'b'], ['a']],
 [['a', 'b'], ['a', 'b']],
 [['a', 'b'], ['a'], ['b']],
 [['a'], ['a'], ['b', 'b']],
 [['a'], ['a'], ['b'], ['b']]]
```

**See also:**

***multiset\_partitions\_taocp*** (page 2060)

which provides the same result as this method, but is about twice as fast. Hence, `enum_all` is primarily useful for testing. Also see the function for a discussion of states and visitors.

**enum\_large**(*multiplicities*, *lb*)

Enumerate the partitions of a multiset with `lb < num(parts)`

Equivalent to `enum_range(multiplicities, lb, sum(multiplicities))`



**Parameters****multiplicities**

list of multiplicities of the components of the multiset.

**lb**

Number of parts in the partition must be greater than this lower bound.

**Examples**

```
>>> from sympy.utilities.enumerative import list_visitor
>>> from sympy.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> states = m.enum_large([2,2], 2)
>>> list(list_visitor(state, 'ab') for state in states)
[[['a', 'a'], ['b'], ['b']],
 [['a', 'b'], ['a'], ['b']],
 [['a'], ['a'], ['b', 'b']],
 [['a'], ['a'], ['b'], ['b']]]
```

**See also:**

[\*enum\\_all\*](#) (page 2064), [\*enum\\_small\*](#) (page 2065), [\*enum\\_range\*](#) (page 2065)

**enum\_range**(*multiplicities*, *lb*, *ub*)

Enumerate the partitions of a multiset with  $lb < \text{num}(\text{parts}) \leq ub$ .

In particular, if partitions with exactly  $k$  parts are desired, call with  $(\text{multiplicities}, k - 1, k)$ . This method generalizes `enum_all`, `enum_small`, and `enum_large`.

**Examples**

```
>>> from sympy.utilities.enumerative import list_visitor
>>> from sympy.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> states = m.enum_range([2,2], 1, 2)
>>> list(list_visitor(state, 'ab') for state in states)
[[['a', 'a', 'b'], ['b']],
 [['a', 'a'], ['b', 'b']],
 [['a', 'b', 'b'], ['a']],
 [['a', 'b'], ['a', 'b']]]
```

**enum\_small**(*multiplicities*, *ub*)

Enumerate multiset partitions with no more than  $ub$  parts.

Equivalent to `enum_range(multiplicities, 0, ub)`

**Parameters****multiplicities**

list of multiplicities of the components of the multiset.

**ub**

Maximum number of parts

### Examples

```
>>> from sympy.utilities.enumerative import list_visitor
>>> from sympy.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> states = m.enum_small([2,2], 2)
>>> list(list_visitor(state, 'ab') for state in states)
[[['a', 'a', 'b', 'b']],
 [['a', 'a', 'b'], ['b']],
 [['a', 'a'], ['b', 'b']],
 [['a', 'b', 'b'], ['a']],
 [['a', 'b'], ['a', 'b']]]
```

The implementation is based, in part, on the answer given to exercise 69, in Knuth [AOCP].

#### See also:

[enum\\_all](#) (page 2064), [enum\\_large](#) (page 2064), [enum\\_range](#) (page 2065)

## Exceptions and Warnings

General SymPy exceptions and warnings.

**exception** `sympy.utilities.exceptions.SymPyDeprecationWarning`(*message*, \*, *depre-*  
*cated\_since\_version*,  
*ac-*  
*tive\_deprecations\_target*)

A warning for deprecated features of SymPy.

See the [Deprecation Policy](#) (page 3017) document for details on when and how things should be deprecated in SymPy.

Note that simply constructing this class will not cause a warning to be issued. To do that, you must call the `:func`sympy_deprecation_warning`` function. For this reason, it is not recommended to ever construct this class directly.

### Explanation

The `SymPyDeprecationWarning` class is a subclass of `DeprecationWarning` that is used for all deprecations in SymPy. A special subclass is used so that we can automatically augment the warning message with additional metadata about the version the deprecation was introduced in and a link to the documentation. This also allows users to explicitly filter deprecation warnings from SymPy using warnings filters (see [Silencing SymPy Deprecation Warnings](#) (page 164)).

Additionally, `SymPyDeprecationWarning` is enabled to be shown by default, unlike normal `DeprecationWarnings`, which are only shown by default in interactive sessions. This ensures that deprecation warnings in SymPy will actually be seen by users.

See the documentation of `sympy_deprecation_warning()` (page 2067) for a description of the parameters to this function.

To mark a function as deprecated, you can use the `@deprecated` (page 2057) decorator.

**See also:**

`sympy.utilities.exceptions.sympy_deprecation_warning` (page 2067), `sympy.utilities.exceptions.ignore_warnings` (page 2067), `sympy.utilities.decorator.deprecated` (page 2057), `sympy.testing.pytest.warns_deprecated_sympy` (page 2029)

`sympy.utilities.exceptions.ignore_warnings(warningcls)`

Context manager to suppress warnings during tests.

---

**Note:** Do not use this with `SymPyDeprecationWarning` in the tests. `warns_deprecated_sympy()` should be used instead.

---

This function is useful for suppressing warnings during tests. The `warns` function should be used to assert that a warning is raised. The `ignore_warnings` function is useful in situation when the warning is not guaranteed to be raised (e.g. on importing a module) or if the warning comes from third-party code.

This function is also useful to prevent the same or similar warnings from being issue twice due to recursive calls.

When the warning is coming (reliably) from SymPy the `warns` function should be preferred to `ignore_warnings`.

```
>>> from sympy.utilities.exceptions import ignore_warnings
>>> import warnings
```

Here's a warning:

```
>>> with warnings.catch_warnings(): # reset warnings in doctest
... warnings.simplefilter('error')
... warnings.warn('deprecated', UserWarning)
Traceback (most recent call last):
...
UserWarning: deprecated
```

Let's suppress it with `ignore_warnings`:

```
>>> with warnings.catch_warnings(): # reset warnings in doctest
... warnings.simplefilter('error')
... with ignore_warnings(UserWarning):
... warnings.warn('deprecated', UserWarning)
```

(No warning emitted)

**See also:**

`sympy.utilities.exceptions.SymPyDeprecationWarning` (page 2066), `sympy.utilities.exceptions.sympy_deprecation_warning` (page 2067), `sympy.utilities.decorator.deprecated` (page 2057), `sympy.testing.pytest.warns_deprecated_sympy` (page 2029)

`sympy.utilities.exceptions.sympy_deprecation_warning(message, *, deprecated_since_version, active_deprecations_target, stacklevel=3)`

Warn that a feature is deprecated in SymPy.

See the [Deprecation Policy](#) (page 3017) document for details on when and how things should be deprecated in SymPy.

To mark an entire function or class as deprecated, you can use the [@deprecated](#) (page 2057) decorator.

#### Parameters

##### **message: str**

The deprecation message. This may span multiple lines and contain code examples. Messages should be wrapped to 80 characters. The message is automatically dedented and leading and trailing whitespace stripped. Messages may include dynamic content based on the user input, but avoid using `str(expression)` if an expression can be arbitrary, as it might be huge and make the warning message unreadable.

##### **deprecated\_since\_version: str**

The version of SymPy the feature has been deprecated since. For new deprecations, this should be the version in [sympy/release.py](#) without the `.dev`. If the next SymPy version ends up being different from this, the release manager will need to update any `SymPyDeprecationWarnings` using the incorrect version. This argument is required and must be passed as a keyword argument. (example: `deprecated_since_version="1.10"`).

##### **active\_deprecations\_target: str**

The Sphinx target corresponding to the section for the deprecation in the [List of active deprecations](#) (page 163) document (see `doc/src/explanation/active-deprecations.md`). This is used to automatically generate a URL to the page in the warning message. This argument is required and must be passed as a keyword argument. (example: `active_deprecations_target="deprecated-feature-abc"`)

##### **stacklevel: int (default: 3)**

The `stacklevel` parameter that is passed to `warnings.warn`. If you create a wrapper that calls this function, this should be increased so that the warning message shows the user line of code that produced the warning. Note that in some cases there will be multiple possible different user code paths that could result in the warning. In that case, just choose the smallest common stacklevel.

#### Examples

```
>>> from sympy.utilities.exceptions import sympy_deprecation_warning
>>> def is_this_zero(x, y=0):
... """
... Determine if x = 0.
...
... Parameters
... =====
```

(continues on next page)

(continued from previous page)

```

...
... x : Expr
... The expression to check.
...
... y : Expr, optional
... If provided, check if x = y.
...
... .. deprecated:: 1.1
...
... The ``y`` argument to ``is_this_zero`` is deprecated. Use
... ``is_this_zero(x - y)`` instead.
...
... """
... from sympy import simplify
...
... if y != 0:
... sympy_deprecation_warning("""
... The y argument to is_zero() is deprecated. Use is_zero(x - y)
→instead.""",
... deprecated_since_version="1.1",
... active_deprecations_target='is-this-zero-y-deprecation')
... return simplify(x - y) == 0
>>> is_this_zero(0)
True
>>> is_this_zero(1, 1)
<stdin>:1: SymPyDeprecationWarning:

The y argument to is_zero() is deprecated. Use is_zero(x - y) instead.

See https://docs.sympy.org/latest/explanation/active-deprecations.html
→#is-this-zero-y-deprecation
for details.

This has been deprecated since SymPy version 1.1. It
will be removed in a future version of SymPy.

 is_this_zero(1, 1)
True

```

**See also:**

`sympy.utilities.exceptions.SymPyDeprecationWarning` (page 2066), `sympy.utilities.exceptions.ignore_warnings` (page 2067), `sympy.utilities.decorator.deprecated` (page 2057), `sympy.testing.pytest.warns_deprecated_sympy` (page 2029)

## Iterables

### variations

`variations(seq, n)` Returns all the variations of the list of size `n`.

Has an optional third argument. Must be a boolean value and makes the method return the variations with repetition if set to `True`, or the variations without repetition if set to `False`.

#### Examples::

```
>>> from sympy.utilities.iterables import variations
>>> list(variations([1,2,3], 2))
[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
>>> list(variations([1,2,3], 2, True))
[(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)]
```

### partitions

Although the combinatorics module contains `Partition` and `IntegerPartition` classes for investigation and manipulation of partitions, there are a few functions to generate partitions that can be used as low-level tools for routines: `partitions` and `multiset_partitions`. The former gives integer partitions, and the latter gives enumerated partitions of elements. There is also a routine `kbins` that will give a variety of permutations of partitions.

`partitions`:

```
>>> from sympy.utilities.iterables import partitions
>>> [p.copy() for s, p in partitions(7, m=2, size=True) if s == 2]
[{1: 1, 6: 1}, {2: 1, 5: 1}, {3: 1, 4: 1}]
```

`multiset_partitions`:

```
>>> from sympy.utilities.iterables import multiset_partitions
>>> [p for p in multiset_partitions(3, 2)]
[[[0, 1], [2]], [[0, 2], [1]], [[0], [1, 2]]]
>>> [p for p in multiset_partitions([1, 1, 1, 2], 2)]
[[[1, 1, 1], [2]], [[1, 1, 2], [1]], [[1, 1], [1, 2]]]
```

`kbins`:

```
>>> from sympy.utilities.iterables import kbins
>>> def show(k):
... rv = []
... for p in k:
... rv.append(','.join([''.join(j) for j in p]))
... return sorted(rv)
...
>>> show(kbins("ABCD", 2))
['A,BCD', 'AB,CD', 'ABC,D']
>>> show(kbins("ABC", 2))
['A,BC', 'AB,C']
>>> show(kbins("ABC", 2, ordered=0)) # same as multiset_partitions
```

(continues on next page)

(continued from previous page)

```

['A,BC', 'AB,C', 'AC,B']
>>> show(kbins("ABC", 2, ordered=1))
['A,BC', 'A,CB',
 'B,AC', 'B,CA',
 'C,AB', 'C,BA']
>>> show(kbins("ABC", 2, ordered=10))
['A,BC', 'AB,C', 'AC,B',
 'B,AC', 'BC,A',
 'C,AB']
>>> show(kbins("ABC", 2, ordered=11))
['A,BC', 'A,CB', 'AB,C', 'AC,B',
 'B,AC', 'B,CA', 'BA,C', 'BC,A',
 'C,AB', 'C,BA', 'CA,B', 'CB,A']

```

## Docstring

### `class sympy.utilities.iterables.NotIterable`

Use this as mixin when creating a class which is not supposed to return true when `iterable()` is called on its instances because calling `list()` on the instance, for example, would result in an infinite loop.

### `sympy.utilities.iterables.binary_partitions(n)`

Generates the binary partition of  $n$ .

A binary partition consists only of numbers that are powers of two. Each step reduces a  $2^{k+1}$  to  $2^k$  and  $2^k$ . Thus 16 is converted to 8 and 8.

## Examples

```

>>> from sympy.utilities.iterables import binary_partitions
>>> for i in binary_partitions(5):
... print(i)
...
[4, 1]
[2, 2, 1]
[2, 1, 1, 1]
[1, 1, 1, 1, 1]

```

## References

[R958]

### `sympy.utilities.iterables.bracelets(n, k)`

Wrapper to necklaces to return a free (unrestricted) necklace.

### `sympy.utilities.iterables.capture(func)`

Return the printed output of `func()`.

`func` should be a function without arguments that produces output with print statements.

```
>>> from sympy.utilities.iterables import capture
>>> from sympy import pprint
>>> from sympy.abc import x
>>> def foo():
... print('hello world!')
...
>>> 'hello' in capture(foo) # foo, not foo()
True
>>> capture(lambda: pprint(2/x))
'2\n-\nx\n'
```

`sympy.utilities.iterables.common_prefix(*seqs)`

Return the subsequence that is a common start of sequences in `seqs`.

```
>>> from sympy.utilities.iterables import common_prefix
>>> common_prefix(list(range(3)))
[0, 1, 2]
>>> common_prefix(list(range(3)), list(range(4)))
[0, 1, 2]
>>> common_prefix([1, 2, 3], [1, 2, 5])
[1, 2]
>>> common_prefix([1, 2, 3], [1, 3, 5])
[1]
```

`sympy.utilities.iterables.common_suffix(*seqs)`

Return the subsequence that is a common ending of sequences in `seqs`.

```
>>> from sympy.utilities.iterables import common_suffix
>>> common_suffix(list(range(3)))
[0, 1, 2]
>>> common_suffix(list(range(3)), list(range(4)))
[]
>>> common_suffix([1, 2, 3], [9, 2, 3])
[2, 3]
>>> common_suffix([1, 2, 3], [9, 7, 3])
[3]
```

`sympy.utilities.iterables.connected_components(G)`

Connected components of an undirected graph or weakly connected components of a directed graph.

**Parameters**

**graph** : tuple[list, list[tuple[T, T]]

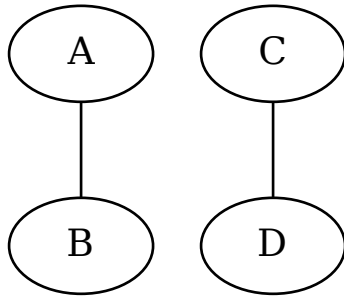
A tuple consisting of a list of vertices and a list of edges of a graph whose connected components are to be found.



## Examples

Given an undirected graph:

```
graph {
 A -- B
 C -- D
}
```



We can find the connected components using this function if we include each edge in both directions:

```
>>> from sympy.utilities.iterables import connected_components

>>> V = ['A', 'B', 'C', 'D']
>>> E = [('A', 'B'), ('B', 'A'), ('C', 'D'), ('D', 'C')]
>>> connected_components((V, E))
[['A', 'B'], ['C', 'D']]
```

The weakly connected components of a directed graph can found the same way.

## Notes

The vertices of the graph must be hashable for the data structures used. If the vertices are unhashable replace them with integer indices.

This function uses Tarjan's algorithm to compute the connected components in  $O(|V|+|E|)$  (linear) time.

### See also:

[\*sympy.utilities.iterables.strongly\\_connected\\_components\*](#) (page 2094)

## References

[R959], [R960]

`sympy.utilities.iterables.dict_merge(*dicts)`

Merge dictionaries into a single dictionary.

`sympy.utilities.iterables.filter_symbols(iterator, exclude)`

Only yield elements from *iterator* that do not occur in *exclude*.

### Parameters

**iterator** : iterable

iterator to take elements from

**exclude** : iterable

elements to exclude

### Returns

**iterator** : iterator

filtered iterator

`sympy.utilities.iterables.flatten(iterable, levels=None, cls=None)`

Recursively denest iterable containers.

```
>>> from sympy import flatten
```

```
>>> flatten([1, 2, 3])
[1, 2, 3]
>>> flatten([1, 2, [3]])
[1, 2, 3]
>>> flatten([1, [2, 3], [4, 5]])
[1, 2, 3, 4, 5]
>>> flatten([1.0, 2, (1, None)])
[1.0, 2, 1, None]
```

If you want to denest only a specified number of levels of nested containers, then set `levels` flag to the desired number of levels:

```
>>> ls = [((-2, -1), (1, 2)), [(0, 0)]]
```

```
>>> flatten(ls, levels=1)
[(-2, -1), (1, 2), (0, 0)]
```

If `cls` argument is specified, it will only flatten instances of that class, for example:

```
>>> from sympy import Basic, S
>>> class MyOp(Basic):
... pass
...
>>> flatten([MyOp(S(1), MyOp(S(2), S(3)))], cls=MyOp)
[1, 2, 3]
```

adapted from [https://kogs-www.informatik.uni-hamburg.de/~meine/python\\_tricks](https://kogs-www.informatik.uni-hamburg.de/~meine/python_tricks)

`sympy.utilities.iterables.generate_bell(n)`

Return permutations of  $[0, 1, \dots, n - 1]$  such that each permutation differs from the last by the exchange of a single pair of neighbors. The  $n!$  permutations are returned as an iterator. In order to obtain the next permutation from a random starting permutation, use the `next_trotterjohnson` method of the `Permutation` class (which generates the same sequence in a different manner).

### Examples

```
>>> from itertools import permutations
>>> from sympy.utilities.iterables import generate_bell
>>> from sympy import zeros, Matrix
```

This is the sort of permutation used in the ringing of physical bells, and does not produce permutations in lexicographical order. Rather, the permutations differ from each other by exactly one inversion, and the position at which the swapping occurs varies periodically in a simple fashion. Consider the first few permutations of 4 elements generated by `permutations` and `generate_bell`:

```
>>> list(permutations(range(4))[:5])
[(0, 1, 2, 3), (0, 1, 3, 2), (0, 2, 1, 3), (0, 2, 3, 1), (0, 3, 1, 2)]
>>> list(generate_bell(4)[:5])
[(0, 1, 2, 3), (0, 1, 3, 2), (0, 3, 1, 2), (3, 0, 1, 2), (3, 0, 2, 1)]
```

Notice how the 2nd and 3rd lexicographical permutations have 3 elements out of place whereas each “bell” permutation always has only two elements out of place relative to the previous permutation (and so the signature  $(+/-1)$  of a permutation is opposite of the signature of the previous permutation).

How the position of inversion varies across the elements can be seen by tracing out where the largest number appears in the permutations:

```
>>> m = zeros(4, 24)
>>> for i, p in enumerate(generate_bell(4)):
... m[:, i] = Matrix([j - 3 for j in list(p)]) # make largest zero
>>> m.print_nonzero('X')
[XXX XXXXXX XXXXXX XXX]
[XX XX XXXX XX XXXX XX XX]
[X XXXX XX XXXX XX XXXX X]
[XXXXXX XXXXXX XXXXXX]
```

**See also:**

[`sympy.combinatorics.permutations.Permutation.next\_trotterjohnson`](#) (page 277)

## References

[R961], [R962], [R963], [R964], [R965]

`sympy.utilities.iterables.generate_derangements(s)`

Return unique derangements of the elements of iterable *s*.

## Examples

```
>>> from sympy.utilities.iterables import generate_derangements
>>> list(generate_derangements([0, 1, 2]))
[[1, 2, 0], [2, 0, 1]]
>>> list(generate_derangements([0, 1, 2, 2]))
[[2, 2, 0, 1], [2, 2, 1, 0]]
>>> list(generate_derangements([0, 1, 1]))
[]
```

### See also:

[\*sympy.functions.combinatorial.factorials.subfactorial\*](#) (page 435)

`sympy.utilities.iterables.generate_involutions(n)`

Generates involutions.

An involution is a permutation that when multiplied by itself equals the identity permutation. In this implementation the involutions are generated using Fixed Points.

Alternatively, an involution can be considered as a permutation that does not contain any cycles with a length that is greater than two.

## Examples

```
>>> from sympy.utilities.iterables import generate_involutions
>>> list(generate_involutions(3))
[(0, 1, 2), (0, 2, 1), (1, 0, 2), (2, 1, 0)]
>>> len(list(generate_involutions(4)))
10
```

## References

[R966]

`sympy.utilities.iterables.generate_oriented_forest(n)`

This algorithm generates oriented forests.

An oriented graph is a directed graph having no symmetric pair of directed edges. A forest is an acyclic graph, i.e., it has no cycles. A forest can also be described as a disjoint union of trees, which are graphs in which any two vertices are connected by exactly one simple path.

## Examples

```
>>> from sympy.utilities.iterables import generate_oriented_forest
>>> list(generate_oriented_forest(4))
[[0, 1, 2, 3], [0, 1, 2, 2], [0, 1, 2, 1], [0, 1, 2, 0], [0, 1, 1, 0],
→ [1], [0, 1, 1, 0], [0, 1, 0, 1], [0, 1, 0, 0], [0, 0, 0, 0]]
```

## References

[R967], [R968]

`sympy.utilities.iterables.group(seq, multiple=True)`  
 Splits a sequence into a list of lists of equal, adjacent elements.

## Examples

```
>>> from sympy import group
```

```
>>> group([1, 1, 1, 2, 2, 3])
[[1, 1, 1], [2, 2], [3]]
>>> group([1, 1, 1, 2, 2, 3], multiple=False)
[(1, 3), (2, 2), (3, 1)]
>>> group([1, 1, 3, 2, 2, 1], multiple=False)
[(1, 2), (3, 1), (2, 2), (1, 1)]
```

## See also:

[`multiset`](#) (page 2083)

`sympy.utilities.iterables.has_dups(seq)`  
 Return True if there are any duplicate elements in seq.

## Examples

```
>>> from sympy import has_dups, Dict, Set
>>> has_dups((1, 2, 1))
True
>>> has_dups(range(3))
False
>>> all(has_dups(c) is False for c in (set(), Set(), dict(), Dict()))
True
```

`sympy.utilities.iterables.has_variety(seq)`  
 Return True if there are any different elements in seq.

## Examples

```
>>> from sympy import has_variety
```

```
>>> has_variety((1, 2, 1))
True
>>> has_variety((1, 1, 1))
False
```

`sympy.utilities.iterables.ibin(n, bits=None, str=False)`

Return a list of length *bits* corresponding to the binary value of *n* with small bits to the right (last). If *bits* is omitted, the length will be the number required to represent *n*. If the bits are desired in reversed order, use the `[::-1]` slice of the returned list.

If a sequence of all bits-length lists starting from `[0, 0, ..., 0]` through `[1, 1, ..., 1]` are desired, pass a non-integer for *bits*, e.g. `'all'`.

If the bit *string* is desired pass *str=True*.

## Examples

```
>>> from sympy.utilities.iterables import ibin
>>> ibin(2)
[1, 0]
>>> ibin(2, 4)
[0, 0, 1, 0]
```

If all lists corresponding to 0 to  $2^n - 1$ , pass a non-integer for *bits*:

```
>>> bits = 2
>>> for i in ibin(2, 'all'):
... print(i)
(0, 0)
(0, 1)
(1, 0)
(1, 1)
```

If a bit string is desired of a given length, use *str=True*:

```
>>> n = 123
>>> bits = 10
>>> ibin(n, bits, str=True)
'0001111011'
>>> ibin(n, bits, str=True)[::-1] # small bits left
'1101111000'
>>> list(ibin(3, 'all', str=True))
['000', '001', '010', '011', '100', '101', '110', '111']
```

`sympy.utilities.iterables.iproduct(*iterables)`

Cartesian product of iterables.

Generator of the Cartesian product of iterables. This is analogous to `itertools.product` except that it works with infinite iterables and will yield any item from the infinite product eventually.

## Examples

```
>>> from sympy.utilities.iterables import iproduct
>>> sorted(iproduct([1,2], [3,4]))
[(1, 3), (1, 4), (2, 3), (2, 4)]
```

With an infinite iterator:

```
>>> from sympy import S
>>> (3,) in iproduct(S.Integers)
True
>>> (3, 4) in iproduct(S.Integers, S.Integers)
True
```

### See also:

[itertools.product](#)

`sympy.utilities.iterables.is_palindromic(s, i=0, j=None)`

Return True if the sequence is the same from left to right as it is from right to left in the whole sequence (default) or in the Python slice `s[i: j]`; else False.

## Examples

```
>>> from sympy.utilities.iterables import is_palindromic
>>> is_palindromic([1, 0, 1])
True
>>> is_palindromic('abcbb')
False
>>> is_palindromic('abcbb', 1)
False
```

Normal Python slicing is performed in place so there is no need to create a slice of the sequence for testing:

```
>>> is_palindromic('abcbb', 1, -1)
True
>>> is_palindromic('abcbb', -4, -1)
True
```

### See also:

[sympy.ntheory.digits.is\\_palindromic](#) (page 1528)  
tests integers

`sympy.utilities.iterables.is_sequence(i, include=None)`

Return a boolean indicating whether `i` is a sequence in the SymPy sense. If anything that fails the test below should be included as being a sequence for your application, set 'include' to that object's type; multiple types should be passed as a tuple of types.

Note: although generators can generate a sequence, they often need special handling to make sure their elements are captured before the generator is exhausted, so these are not included by default in the definition of a sequence.

See also: `iterable`

### Examples

```
>>> from sympy.utilities.iterables import is_sequence
>>> from types import GeneratorType
>>> is_sequence([])
True
>>> is_sequence(set())
False
>>> is_sequence('abc')
False
>>> is_sequence('abc', include=str)
True
>>> generator = (c for c in 'abc')
>>> is_sequence(generator)
False
>>> is_sequence(generator, include=(str, GeneratorType))
True
```

`sympy.utilities.iterables.iterable(i, exclude=(<class 'str'>, <class 'dict'>, <class 'sympy.utilities.iterables.NotIterable'>))`

Return a boolean indicating whether `i` is SymPy iterable. True also indicates that the iterator is finite, e.g. you can call `list(...)` on the instance.

When SymPy is working with iterables, it is almost always assuming that the iterable is not a string or a mapping, so those are excluded by default. If you want a pure Python definition, make `exclude=None`. To exclude multiple items, pass them as a tuple.

You can also set the `_iterable` attribute to True or False on your class, which will override the checks here, including the `exclude` test.

As a rule of thumb, some SymPy functions use this to check if they should recursively map over an object. If an object is technically iterable in the Python sense but does not desire this behavior (e.g., because its iteration is not finite, or because iteration might induce an unwanted computation), it should disable it by setting the `_iterable` attribute to False.

See also: `is_sequence`

### Examples

```
>>> from sympy.utilities.iterables import iterable
>>> from sympy import Tuple
>>> things = [[1], (1,), set([1]), Tuple(1), (j for j in [1, 2]), {1:2},
↳ '1', 1]
>>> for i in things:
... print('%s %s' % (iterable(i), type(i)))
True <... 'list'>
True <... 'tuple'>
True <... 'set'>
True <class 'sympy.core.containers.Tuple'>
```

(continues on next page)



(continued from previous page)

```
True <... 'generator'>
False <... 'dict'>
False <... 'str'>
False <... 'int'>
```

```
>>> iterable({}, exclude=None)
True
>>> iterable({}, exclude=str)
True
>>> iterable("no", exclude=str)
False
```

`sympy.utilities.iterables.kbins(l, k, ordered=None)`

Return sequence `l` partitioned into `k` bins.

### Examples

The default is to give the items in the same order, but grouped into `k` partitions without any reordering:

```
>>> from sympy.utilities.iterables import kbins
>>> for p in kbins(list(range(5)), 2):
... print(p)
...
[[0], [1, 2, 3, 4]]
[[0, 1], [2, 3, 4]]
[[0, 1, 2], [3, 4]]
[[0, 1, 2, 3], [4]]
```

The `ordered` flag is either `None` (to give the simple partition of the elements) or is a 2 digit integer indicating whether the order of the bins and the order of the items in the bins matters. Given:

```
A = [[0], [1, 2]]
B = [[1, 2], [0]]
C = [[2, 1], [0]]
D = [[0], [2, 1]]
```

the following values for `ordered` have the shown meanings:

```
00 means A == B == C == D
01 means A == B
10 means A == D
11 means A == A
```

```
>>> for ordered_flag in [None, 0, 1, 10, 11]:
... print('ordered = %s' % ordered_flag)
... for p in kbins(list(range(3)), 2, ordered=ordered_flag):
... print(' %s' % p)
...
ordered = None
```

(continues on next page)

(continued from previous page)

```

[[0], [1, 2]]
[[0, 1], [2]]
ordered = 0
[[0, 1], [2]]
[[0, 2], [1]]
[[0], [1, 2]]
ordered = 1
[[0], [1, 2]]
[[0], [2, 1]]
[[1], [0, 2]]
[[1], [2, 0]]
[[2], [0, 1]]
[[2], [1, 0]]
ordered = 10
[[0, 1], [2]]
[[2], [0, 1]]
[[0, 2], [1]]
[[1], [0, 2]]
[[0], [1, 2]]
[[1, 2], [0]]
ordered = 11
[[0], [1, 2]]
[[0, 1], [2]]
[[0], [2, 1]]
[[0, 2], [1]]
[[1], [0, 2]]
[[1, 0], [2]]
[[1], [2, 0]]
[[1, 2], [0]]
[[2], [0, 1]]
[[2, 0], [1]]
[[2], [1, 0]]
[[2, 1], [0]]

```

**See also:**

[\*partitions\*](#) (page 2089), [\*multiset\\_partitions\*](#) (page 2084)

`sympy.utilities.iterables.least_rotation(x, key=None)`

Returns the number of steps of left rotation required to obtain lexicographically minimal string/list/tuple, etc.

**Examples**

```

>>> from sympy.utilities.iterables import least_rotation, rotate_left
>>> a = [3, 1, 5, 1, 2]
>>> least_rotation(a)
3
>>> rotate_left(a, _)
[1, 2, 3, 1, 5]

```

## References

[R969]

`sympy.utilities.iterables.minlex(seq, directed=True, key=None)`

Return the rotation of the sequence in which the lexically smallest elements appear first, e.g.  $cba \rightarrow acb$ .

The sequence returned is a tuple, unless the input sequence is a string in which case a string is returned.

If `directed` is `False` then the smaller of the sequence and the reversed sequence is returned, e.g.  $cba \rightarrow abc$ .

If `key` is not `None` then it is used to extract a comparison key from each element in iterable.

## Examples

```
>>> from sympy.combinatorics.polyhedron import minlex
>>> minlex((1, 2, 0))
(0, 1, 2)
>>> minlex((1, 0, 2))
(0, 2, 1)
>>> minlex((1, 0, 2), directed=False)
(0, 1, 2)
```

```
>>> minlex('11010011000', directed=True)
'00011010011'
>>> minlex('11010011000', directed=False)
'00011001011'
```

```
>>> minlex(('bb', 'aaa', 'c', 'a'))
('a', 'bb', 'aaa', 'c')
>>> minlex(('bb', 'aaa', 'c', 'a'), key=len)
('c', 'a', 'bb', 'aaa')
```

`sympy.utilities.iterables.multiset(seq)`

Return the hashable sequence in multiset form with values being the multiplicity of the item in the sequence.

## Examples

```
>>> from sympy.utilities.iterables import multiset
>>> multiset('mississippi')
{'i': 4, 'm': 1, 'p': 2, 's': 4}
```

**See also:**

[group](#) (page 2077)

`sympy.utilities.iterables.multiset_combinations(m, n, g=None)`

Return the unique combinations of size `n` from multiset `m`.

### Examples

```
>>> from sympy.utilities.iterables import multiset_combinations
>>> from itertools import combinations
>>> [''.join(i) for i in multiset_combinations('baby', 3)]
['abb', 'aby', 'bby']
```

```
>>> def count(f, s): return len(list(f(s, 3)))
```

The number of combinations depends on the number of letters; the number of unique combinations depends on how the letters are repeated.

```
>>> s1 = 'abracadabra'
>>> s2 = 'banana tree'
>>> count(combinations, s1), count(multiset_combinations, s1)
(165, 23)
>>> count(combinations, s2), count(multiset_combinations, s2)
(165, 54)
```

`sympy.utilities.iterables.multiset_derangements(s)`

Generate derangements of the elements of *s in place*.

### Examples

```
>>> from sympy.utilities.iterables import multiset_derangements, uniq
```

Because the derangements of multisets (not sets) are generated in place, copies of the return value must be made if a collection of derangements is desired or else all values will be the same:

```
>>> list(uniq([i for i in multiset_derangements('1233')]))
[[None, None, None, None]]
>>> [i.copy() for i in multiset_derangements('1233')]
[['3', '3', '1', '2'], ['3', '3', '2', '1']]
>>> [''.join(i) for i in multiset_derangements('1233')]
['3312', '3321']
```

`sympy.utilities.iterables.multiset_partitions(multiset, m=None)`

Return unique partitions of the given multiset (in list form). If *m* is *None*, all multisets will be returned, otherwise only partitions with *m* parts will be returned.

If *multiset* is an integer, a range `[0, 1, ..., multiset - 1]` will be supplied.

## Examples

```
>>> from sympy.utilities.iterables import multiset_partitions
>>> list(multiset_partitions([1, 2, 3, 4], 2))
[[[1, 2, 3], [4]], [[1, 2, 4], [3]], [[1, 2], [3, 4]],
 [[1, 3, 4], [2]], [[1, 3], [2, 4]], [[1, 4], [2, 3]],
 [[1], [2, 3, 4]]]
>>> list(multiset_partitions([1, 2, 3, 4], 1))
[[[1, 2, 3, 4]]]
```

Only unique partitions are returned and these will be returned in a canonical order regardless of the order of the input:

```
>>> a = [1, 2, 2, 1]
>>> ans = list(multiset_partitions(a, 2))
>>> a.sort()
>>> list(multiset_partitions(a, 2)) == ans
True
>>> a = range(3, 1, -1)
>>> (list(multiset_partitions(a)) ==
... list(multiset_partitions(sorted(a))))
True
```

If *m* is omitted then all partitions will be returned:

```
>>> list(multiset_partitions([1, 1, 2]))
[[[1, 1, 2]], [[1, 1], [2]], [[1, 2], [1]], [[1], [1], [2]]]
>>> list(multiset_partitions([1]*3))
[[[1, 1, 1]], [[1], [1, 1]], [[1], [1], [1]]]
```

## Counting

The number of partitions of a set is given by the bell number:

```
>>> from sympy import bell
>>> len(list(multiset_partitions(5))) == bell(5) == 52
True
```

The number of partitions of length *k* from a set of size *n* is given by the Stirling Number of the 2nd kind:

```
>>> from sympy.functions.combinatorial.numbers import stirling
>>> stirling(5, 2) == len(list(multiset_partitions(5, 2))) == 15
True
```

These comments on counting apply to *sets*, not multisets.

## Notes

When all the elements are the same in the multiset, the order of the returned partitions is determined by the `partitions` routine. If one is counting partitions then it is better to use the `nT` function.

### See also:

[\*partitions\*](#) (page 2089), [\*sympy.combinatorics.partitions.Partition\*](#) (page 251), [\*sympy.combinatorics.partitions.IntegerPartition\*](#) (page 253), [\*sympy.functions.combinatorial.numbers.nT\*](#) (page 448)

`sympy.utilities.iterables.multiset_permutations(m, size=None, g=None)`

Return the unique permutations of multiset `m`.

## Examples

```
>>> from sympy.utilities.iterables import multiset_permutations
>>> from sympy import factorial
>>> [''.join(i) for i in multiset_permutations('aab')]
['aab', 'aba', 'baa']
>>> factorial(len('banana'))
720
>>> len(list(multiset_permutations('banana')))
60
```

`sympy.utilities.iterables.necklaces(n, k, free=False)`

A routine to generate necklaces that may (`free=True`) or may not (`free=False`) be turned over to be viewed. The “necklaces” returned are comprised of `n` integers (beads) with `k` different values (colors). Only unique necklaces are returned.

## Examples

```
>>> from sympy.utilities.iterables import necklaces, bracelets
>>> def show(s, i):
... return ''.join(s[j] for j in i)
```

The “unrestricted necklace” is sometimes also referred to as a “bracelet” (an object that can be turned over, a sequence that can be reversed) and the term “necklace” is used to imply a sequence that cannot be reversed. So `ACB == ABC` for a bracelet (rotate and reverse) while the two are different for a necklace since rotation alone cannot make the two sequences the same.

(mnemonic: Bracelets can be viewed Backwards, but Not Necklaces.)

```
>>> B = [show('ABC', i) for i in bracelets(3, 3)]
>>> N = [show('ABC', i) for i in necklaces(3, 3)]
>>> set(N) - set(B)
{'ACB'}
```

```
>>> list(necklaces(4, 2))
[(0, 0, 0, 0), (0, 0, 0, 1), (0, 0, 1, 1),
 (0, 1, 0, 1), (0, 1, 1, 1), (1, 1, 1, 1)]
```

```
>>> [show('.o', i) for i in bracelets(4, 2)]
['.....', '....o', '..oo', '.o.o', '.ooo', 'oooo']
```

## References

[R970]

`sympy.utilities.iterables.numbered_symbols(prefix='x', cls=None, start=0, exclude=(), *args, **assumptions)`

Generate an infinite stream of Symbols consisting of a prefix and increasing subscripts provided that they do not occur in `exclude`.

### Parameters

**prefix** : str, optional

The prefix to use. By default, this function will generate symbols of the form “x0”, “x1”, etc.

**cls** : class, optional

The class to use. By default, it uses `Symbol`, but you can also use `Wild` or `Dummy`.

**start** : int, optional

The start number. By default, it is 0.

### Returns

**sym** : Symbol

The subscripted symbols.

`sympy.utilities.iterables.ordered_partitions(n, m=None, sort=True)`

Generates ordered partitions of integer `n`.

### Parameters

**m** : integer (default None)

The default value gives partitions of all sizes else only those with size `m`. In addition, if `m` is not None then partitions are generated *in place* (see examples).

**sort** : bool (default True)

Controls whether partitions are returned in sorted order when `m` is not None; when False, the partitions are returned as fast as possible with elements sorted, but when `m|n` the partitions will not be in ascending lexicographical order.

## Examples

```
>>> from sympy.utilities.iterables import ordered_partitions
```

All partitions of 5 in ascending lexicographical:

```
>>> for p in ordered_partitions(5):
... print(p)
[1, 1, 1, 1, 1]
[1, 1, 1, 2]
[1, 1, 3]
[1, 2, 2]
[1, 4]
[2, 3]
[5]
```

Only partitions of 5 with two parts:

```
>>> for p in ordered_partitions(5, 2):
... print(p)
[1, 4]
[2, 3]
```

When *m* is given, a given list objects will be used more than once for speed reasons so you will not see the correct partitions unless you make a copy of each as it is generated:

```
>>> [p for p in ordered_partitions(7, 3)]
[[1, 1, 1], [1, 1, 1], [1, 1, 1], [2, 2, 2]]
>>> [list(p) for p in ordered_partitions(7, 3)]
[[1, 1, 5], [1, 2, 4], [1, 3, 3], [2, 2, 3]]
```

When *n* is a multiple of *m*, the elements are still sorted but the partitions themselves will be *unordered* if *sort* is *False*; the default is to return them in ascending lexicographical order.

```
>>> for p in ordered_partitions(6, 2):
... print(p)
[1, 5]
[2, 4]
[3, 3]
```

But if speed is more important than ordering, *sort* can be set to *False*:

```
>>> for p in ordered_partitions(6, 2, sort=False):
... print(p)
[1, 5]
[3, 3]
[2, 4]
```



## References

[R971], [R972]

`sympy.utilities.iterables.partitions(n, m=None, k=None, size=False)`

Generate all partitions of positive integer, *n*.

### Parameters

**m** : integer (default gives partitions of all sizes)

limits number of parts in partition (mnemonic: *m*, maximum parts)

**k** : integer (default gives partitions number from 1 through *n*)

limits the numbers that are kept in the partition (mnemonic: *k*, keys)

**size** : bool (default False, only partition is returned)

when True then (*M*, *P*) is returned where *M* is the sum of the multiplicities and *P* is the generated partition.

**Each partition is represented as a dictionary, mapping an integer to the number of copies of that integer in the partition. For example, the first partition of 4 returned is {4: 1}, “4: one of them”.**

## Examples

```
>>> from sympy.utilities.iterables import partitions
```

The numbers appearing in the partition (the key of the returned dict) are limited with *k*:

```
>>> for p in partitions(6, k=2):
... print(p)
{2: 3}
{1: 2, 2: 2}
{1: 4, 2: 1}
{1: 6}
```

The maximum number of parts in the partition (the sum of the values in the returned dict) are limited with *m* (default value, None, gives partitions from 1 through *n*):

```
>>> for p in partitions(6, m=2):
... print(p)
...
{6: 1}
{1: 1, 5: 1}
{2: 1, 4: 1}
{3: 2}
```

### See also:

[sympy.combinatorics.partitions.Partition](#) (page 251), [sympy.combinatorics.partitions.IntegerPartition](#) (page 253)

## References

[R973]

`sympy.utilities.iterables.permute_signs(t)`

Return iterator in which the signs of non-zero elements of `t` are permuted.

## Examples

```
>>> from sympy.utilities.iterables import permute_signs
>>> list(permute_signs((0, 1, 2)))
[(0, 1, 2), (0, -1, 2), (0, 1, -2), (0, -1, -2)]
```

`sympy.utilities.iterables.postfixes(seq)`

Generate all postfixes of a sequence.

## Examples

```
>>> from sympy.utilities.iterables import postfixes
```

```
>>> list(postfixes([1,2,3,4]))
[[4], [3, 4], [2, 3, 4], [1, 2, 3, 4]]
```

`sympy.utilities.iterables.prefixes(seq)`

Generate all prefixes of a sequence.

## Examples

```
>>> from sympy.utilities.iterables import prefixes
```

```
>>> list(prefixes([1,2,3,4]))
[[1], [1, 2], [1, 2, 3], [1, 2, 3, 4]]
```

`sympy.utilities.iterables.random_derangement(t, choice=None, strict=True)`

Return a list of elements in which none are in the same positions as they were originally. If an element fills more than half of the positions then an error will be raised since no derangement is possible. To obtain a derangement of as many items as possible-with some of the most numerous remaining in their original positions-pass `strict = False`. To produce a pseudorandom derangement, pass a pseudorandom selector like `choice` (see below).

## Examples

```
>>> from sympy.utilities.iterables import random_derangement
>>> t = 'SymPy: a CAS in pure Python'
>>> d = random_derangement(t)
>>> all(i != j for i, j in zip(d, t))
True
```

A predictable result can be obtained by using a pseudorandom generator for the choice:

```
>>> from sympy.core.random import seed, choice as c
>>> seed(1)
>>> d = [''.join(random_derangement(t, c)) for i in range(5)]
>>> assert len(set(d)) != 1 # we got different values
```

By reseeding, the same sequence can be obtained:

```
>>> seed(1)
>>> d2 = [''.join(random_derangement(t, c)) for i in range(5)]
>>> assert d == d2
```

`sympy.utilities.iterables.reshape(seq, how)`

Reshape the sequence according to the template in how.

## Examples

```
>>> from sympy.utilities import reshape
>>> seq = list(range(1, 9))
```

```
>>> reshape(seq, [4]) # lists of 4
[[1, 2, 3, 4], [5, 6, 7, 8]]
```

```
>>> reshape(seq, (4,)) # tuples of 4
[(1, 2, 3, 4), (5, 6, 7, 8)]
```

```
>>> reshape(seq, (2, 2)) # tuples of 4
[(1, 2, 3, 4), (5, 6, 7, 8)]
```

```
>>> reshape(seq, (2, [2])) # (i, i, [i, i])
[(1, 2, [3, 4]), (5, 6, [7, 8])]
```

```
>>> reshape(seq, ((2,), [2])) # etc....
[((1, 2), [3, 4]), ((5, 6), [7, 8])]
```

```
>>> reshape(seq, (1, [2], 1))
[(1, [2, 3], 4), (5, [6, 7], 8)]
```

```
>>> reshape(tuple(seq), ([[1], 1, (2,)]))
([([1], 2, (3, 4)), ([5], 6, (7, 8))])
```

```
>>> reshape(tuple(seq), ([1], 1, (2,)))
([1], 2, (3, 4)), ([5], 6, (7, 8)))
```

```
>>> reshape(list(range(12)), [2, [3], {2}, (1, (3,), 1)])
[[0, 1, [2, 3, 4], {5, 6}, (7, (8, 9, 10), 11)]]
```

`sympy.utilities.iterables.rotate_left(x, y)`

Left rotates a list x by the number of steps specified in y.

### Examples

```
>>> from sympy.utilities.iterables import rotate_left
>>> a = [0, 1, 2]
>>> rotate_left(a, 1)
[1, 2, 0]
```

`sympy.utilities.iterables.rotate_right(x, y)`

Right rotates a list x by the number of steps specified in y.

### Examples

```
>>> from sympy.utilities.iterables import rotate_right
>>> a = [0, 1, 2]
>>> rotate_right(a, 1)
[2, 0, 1]
```

`sympy.utilities.iterables.rotations(s, dir=1)`

Return a generator giving the items in s as list where each subsequent list has the items rotated to the left (default) or right (`dir=-1`) relative to the previous list.

### Examples

```
>>> from sympy import rotations
>>> list(rotations([1,2,3]))
[[1, 2, 3], [2, 3, 1], [3, 1, 2]]
>>> list(rotations([1,2,3], -1))
[[1, 2, 3], [3, 1, 2], [2, 3, 1]]
```

`sympy.utilities.iterables.roundrobin(*iterables)`

roundrobin recipe taken from itertools documentation: <https://docs.python.org/3/library/itertools.html#recipes>

`roundrobin('ABC', 'D', 'EF')` -> A D E B F C

Recipe credited to George Sakkis

`sympy.utilities.iterables.runs(seq, op=<built-in function gt>)`

Group the sequence into lists in which successive elements all compare the same with the comparison operator, op: `op(seq[i + 1], seq[i])` is True from all elements in a run.

## Examples

```
>>> from sympy.utilities.iterables import runs
>>> from operator import ge
>>> runs([0, 1, 2, 2, 1, 4, 3, 2, 2])
[[0, 1, 2], [2], [1, 4], [3], [2], [2]]
>>> runs([0, 1, 2, 2, 1, 4, 3, 2, 2], op=ge)
[[0, 1, 2, 2], [1, 4], [3], [2, 2]]
```

`sympy.utilities.iterables.sift(seq, keyfunc, binary=False)`  
Sift the sequence, seq according to keyfunc.

### Returns

When binary is False (default), the output is a dictionary where elements of seq are stored in a list keyed to the value of keyfunc for that element. If binary is True then a tuple with lists T and F are returned where T is a list containing elements of seq for which keyfunc was True and F containing those elements for which keyfunc was False; a ValueError is raised if the keyfunc is not binary.

## Examples

```
>>> from sympy.utilities import sift
>>> from sympy.abc import x, y
>>> from sympy import sqrt, exp, pi, Tuple
```

```
>>> sift(range(5), lambda x: x % 2)
{0: [0, 2, 4], 1: [1, 3]}
```

`sift()` returns a `defaultdict()` object, so any key that has no matches will give `[]`.

```
>>> sift([x], lambda x: x.is_commutative)
{True: [x]}
>>> _[False]
[]
```

Sometimes you will not know how many keys you will get:

```
>>> sift([sqrt(x), exp(x), (y**x)**2],
... lambda x: x.as_base_exp()[0])
{E: [exp(x)], x: [sqrt(x)], y: [y**(2*x)]}
```

Sometimes you expect the results to be binary; the results can be unpacked by setting binary to True:

```
>>> sift(range(4), lambda x: x % 2, binary=True)
([1, 3], [0, 2])
>>> sift(Tuple(1, pi), lambda x: x.is_rational, binary=True)
([1], [pi])
```

A `ValueError` is raised if the predicate was not actually binary (which is a good test for the logic where sifting is used and binary results were expected):

```
>>> unknown = exp(1) - pi # the rationality of this is unknown
>>> args = Tuple(1, pi, unknown)
>>> sift(args, lambda x: x.is_rational, binary=True)
Traceback (most recent call last):
...
ValueError: keyfunc gave non-binary output
```

The non-binary sifting shows that there were 3 keys generated:

```
>>> set(sift(args, lambda x: x.is_rational).keys())
{None, False, True}
```

If you need to sort the sifted items it might be better to use `ordered` which can economically apply multiple sort keys to a sequence while sorting.

**See also:**

[`ordered`](#) (page 1076)

`sympy.utilities.iterables.signed_permutations(t)`

Return iterator in which the signs of non-zero elements of *t* and the order of the elements are permuted.

## Examples

```
>>> from sympy.utilities.iterables import signed_permutations
>>> list(signed_permutations((0, 1, 2)))
[(0, 1, 2), (0, -1, 2), (0, 1, -2), (0, -1, -2), (0, 2, 1),
(0, -2, 1), (0, 2, -1), (0, -2, -1), (1, 0, 2), (-1, 0, 2),
(1, 0, -2), (-1, 0, -2), (1, 2, 0), (-1, 2, 0), (1, -2, 0),
(-1, -2, 0), (2, 0, 1), (-2, 0, 1), (2, 0, -1), (-2, 0, -1),
(2, 1, 0), (-2, 1, 0), (2, -1, 0), (-2, -1, 0)]
```

`sympy.utilities.iterables.strongly_connected_components(G)`

Strongly connected components of a directed graph in reverse topological order.

**Parameters**

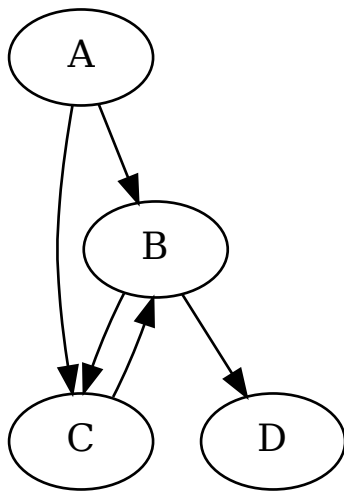
**graph** : tuple[list, list[tuple[T, T]]

A tuple consisting of a list of vertices and a list of edges of a graph whose strongly connected components are to be found.

## Examples

Consider a directed graph (in dot notation):

```
digraph {
 A -> B
 A -> C
 B -> C
 C -> B
 B -> D
}
```



where vertices are the letters A, B, C and D. This graph can be encoded using Python's elementary data structures as follows:

```
>>> V = ['A', 'B', 'C', 'D']
>>> E = [('A', 'B'), ('A', 'C'), ('B', 'C'), ('C', 'B'), ('B', 'D')]
```

The strongly connected components of this graph can be computed as

```
>>> from sympy.utilities.iterables import strongly_connected_components
```

```
>>> strongly_connected_components((V, E))
[['D'], ['B', 'C'], ['A']]
```

This also gives the components in reverse topological order.

Since the subgraph containing B and C has a cycle they must be together in a strongly connected component. A and D are connected to the rest of the graph but not in a cyclic manner so they appear as their own strongly connected components.

## Notes

The vertices of the graph must be hashable for the data structures used. If the vertices are unhashable replace them with integer indices.

This function uses Tarjan's algorithm to compute the strongly connected components in  $O(|V| + |E|)$  (linear) time.

### See also:

[`sympy.utilities.iterables.connected\_components`](#) (page 2072)

## References

[R974], [R975]

`sympy.utilities.iterables.subsets(seq, k=None, repetition=False)`

Generates all  $k$ -subsets (combinations) from an  $n$ -element set, `seq`.

A  $k$ -subset of an  $n$ -element set is any subset of length exactly  $k$ . The number of  $k$ -subsets of an  $n$ -element set is given by  $\text{binomial}(n, k)$ , whereas there are  $2^n$  subsets all together. If  $k$  is `None` then all  $2^n$  subsets will be returned from shortest to longest.

## Examples

```
>>> from sympy import subsets
```

`subsets(seq, k)` will return the  $\frac{n!}{k!(n-k)!}$   $k$ -subsets (combinations) without repetition, i.e. once an item has been removed, it can no longer be "taken":

```
>>> list(subsets([1, 2], 2))
[(1, 2)]
>>> list(subsets([1, 2]))
[(), (1,), (2,), (1, 2)]
>>> list(subsets([1, 2, 3], 2))
[(1, 2), (1, 3), (2, 3)]
```

`subsets(seq, k, repetition=True)` will return the  $\frac{(n-1+k)!}{k!(n-1)!}$  combinations *with* repetition:

```
>>> list(subsets([1, 2], 2, repetition=True))
[(1, 1), (1, 2), (2, 2)]
```

If you ask for more items than are in the set you get the empty set unless you allow repetitions:

```
>>> list(subsets([0, 1], 3, repetition=False))
[]
>>> list(subsets([0, 1], 3, repetition=True))
[(0, 0, 0), (0, 0, 1), (0, 1, 1), (1, 1, 1)]
```

`sympy.utilities.iterables.take(iter, n)`

Return  $n$  items from `iter` iterator.



`sympy.utilities.iterables.topological_sort(graph, key=None)`

Topological sort of graph's vertices.

#### Parameters

**graph** : tuple[list, list[tuple[T, T]]

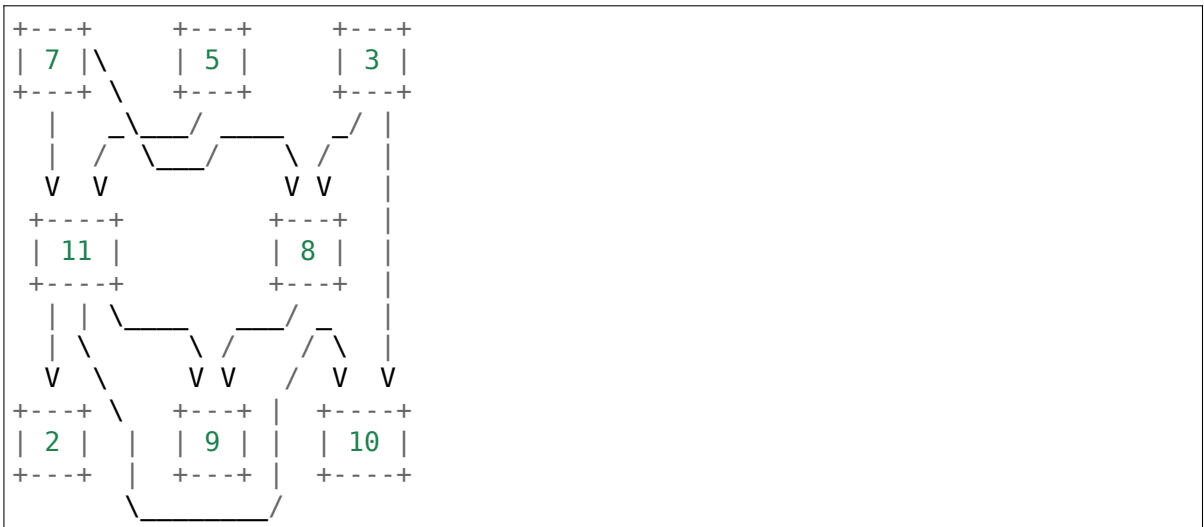
A tuple consisting of a list of vertices and a list of edges of a graph to be sorted topologically.

**key** : callable[T] (optional)

Ordering key for vertices on the same level. By default the natural (e.g. lexicographic) ordering is used (in this case the base type must implement ordering relations).

#### Examples

Consider a graph:



where vertices are integers. This graph can be encoded using elementary Python's data structures as follows:

```
>>> V = [2, 3, 5, 7, 8, 9, 10, 11]
>>> E = [(7, 11), (7, 8), (5, 11), (3, 8), (3, 10),
... (11, 2), (11, 9), (11, 10), (8, 9)]
```

To compute a topological sort for graph (V, E) issue:

```
>>> from sympy.utilities.iterables import topological_sort

>>> topological_sort((V, E))
[3, 5, 7, 8, 11, 2, 9, 10]
```

If specific tie breaking approach is needed, use key parameter:

```
>>> topological_sort((V, E), key=lambda v: -v)
[7, 5, 11, 3, 10, 8, 9, 2]
```

Only acyclic graphs can be sorted. If the input graph has a cycle, then `ValueError` will be raised:

```
>>> topological_sort((V, E + [(10, 7)]))
Traceback (most recent call last):
...
ValueError: cycle detected
```

## References

[R976]

`sympy.utilities.iterables.unflatten(iter, n=2)`

Group `iter` into tuples of length `n`. Raise an error if the length of `iter` is not a multiple of `n`.

`sympy.utilities.iterables.uniq(seq, result=None)`

Yield unique elements from `seq` as an iterator. The second parameter `result` is used internally; it is not necessary to pass anything for this.

Note: changing the sequence during iteration will raise a `RuntimeError` if the size of the sequence is known; if you pass an iterator and advance the iterator you will change the output of this routine but there will be no warning.

## Examples

```
>>> from sympy.utilities.iterables import uniq
>>> dat = [1, 4, 1, 5, 4, 2, 1, 2]
>>> type(uniq(dat)) in (list, tuple)
False
```

```
>>> list(uniq(dat))
[1, 4, 5, 2]
>>> list(uniq(x for x in dat))
[1, 4, 5, 2]
>>> list(uniq([[1], [2, 1], [1]]))
[[1], [2, 1]]
```

`sympy.utilities.iterables.variations(seq, n, repetition=False)`

Returns an iterator over the `n`-sized variations of `seq` (size `N`). `repetition` controls whether items in `seq` can appear more than once;

## Examples

`variations(seq, n)` will return  $\frac{N!}{(N-n)!}$  permutations without repetition of `seq`'s elements:

```
>>> from sympy import variations
>>> list(variations([1, 2], 2))
[(1, 2), (2, 1)]
```

`variations(seq, n, True)` will return the  $N^n$  permutations obtained by allowing repetition of elements:

```
>>> list(variations([1, 2], 2, repetition=True))
[(1, 1), (1, 2), (2, 1), (2, 2)]
```

If you ask for more items than are in the set you get the empty set unless you allow repetitions:

```
>>> list(variations([0, 1], 3, repetition=False))
[]
>>> list(variations([0, 1], 3, repetition=True))[:4]
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1)]
```

**See also:**

[itertools.permutations](#), [itertools.product](#)

## Lambdify

This module provides convenient functions to transform SymPy expressions to lambda functions which can be used to calculate numerical values very fast.

`sympy.utilities.lambdify.implemented_function(symfunc, implementation)`

Add numerical implementation to function `symfunc`.

`symfunc` can be an `UndefinedFunction` instance, or a name string. In the latter case we create an `UndefinedFunction` instance with that name.

Be aware that this is a quick workaround, not a general method to create special symbolic functions. If you want to create a symbolic function to be used by all the machinery of SymPy you should subclass the `Function` class.

### Parameters

**`symfunc`** : str or `UndefinedFunction` instance

If str, then create new `UndefinedFunction` with this as name. If `symfunc` is an `UndefinedFunction` instance, create a new function with the same name and the implemented function attached.

**`implementation`** : callable

numerical implementation to be called by `evalf()` or `lambdify`

### Returns

**`afunc`** : `sympy.FunctionClass` instance

function with attached implementation

### Examples

```
>>> from sympy.abc import x
>>> from sympy.utilities.lambdify import implemented_function
>>> from sympy import lambdify
>>> f = implemented_function('f', lambda x: x+1)
>>> lam_f = lambdify(x, f(x))
>>> lam_f(4)
5
```

`sympy.utilities.lambdify.lambdastr(args, expr, printer=None, dummify=None)`

Returns a string that can be evaluated to a lambda function.

### Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.utilities.lambdify import lambdastr
>>> lambdastr(x, x**2)
'lambda x: (x**2)'
>>> lambdastr((x,y,z), [z,y,x])
'lambda x,y,z: ([z, y, x])'
```

Although tuples may not appear as arguments to lambda in Python 3, lambdastr will create a lambda function that will unpack the original arguments so that nested arguments can be handled:

```
>>> lambdastr((x, (y, z)), x + y)
'lambda _0,_1: (lambda x,y,z: (x + y))(_0,_1[0],_1[1])'
```

`sympy.utilities.lambdify.lambdify(args, expr, modules=None, printer=None, use_ims=True, dummify=False, cse=False)`

Convert a SymPy expression into a function that allows for fast numeric evaluation.

**Warning:** This function uses `exec`, and thus should not be used on unsanitized input.

Deprecated since version 1.7: Passing a set for the *args* parameter is deprecated as sets are unordered. Use an ordered iterable such as a list or tuple.

#### Parameters

**args** : List[Symbol]

A variable or a list of variables whose nesting represents the nesting of the arguments that will be passed to the function.

Variables can be symbols, undefined functions, or matrix symbols.

```
>>> from sympy import Eq
>>> from sympy.abc import x, y, z
```

The list of variables should match the structure of how the arguments will be passed to the function. Simply enclose the parameters as they will be passed in a list.

To call a function like  $f(x)$  then  $[x]$  should be the first argument to `lambdify`; for this case a single  $x$  can also be used:

```
>>> f = lambdify(x, x + 1)
>>> f(1)
2
>>> f = lambdify([x], x + 1)
>>> f(1)
2
```

To call a function like  $f(x, y)$  then  $[x, y]$  will be the first argument of the `lambdify`:

```
>>> f = lambdify([x, y], x + y)
>>> f(1, 1)
2
```

To call a function with a single 3-element tuple like  $f((x, y, z))$  then  $[(x, y, z)]$  will be the first argument of the `lambdify`:

```
>>> f = lambdify([(x, y, z)], Eq(z**2, x**2 + y**2))
>>> f((3, 4, 5))
True
```

If two args will be passed and the first is a scalar but the second is a tuple with two arguments then the items in the list should match that structure:

```
>>> f = lambdify([x, (y, z)], x + y + z)
>>> f(1, (2, 3))
6
```

**expr** : Expr

An expression, list of expressions, or matrix to be evaluated.

Lists may be nested. If the expression is a list, the output will also be a list.

```
>>> f = lambdify(x, [x, [x + 1, x + 2]])
>>> f(1)
[1, [2, 3]]
```

If it is a matrix, an array will be returned (for the NumPy module).

```
>>> from sympy import Matrix
>>> f = lambdify(x, Matrix([x, x + 1]))
>>> f(1)
[[1]
 [2]]
```

Note that the argument order here (variables then expression) is used to emulate the Python `lambda` keyword. `lambdify(x, expr)` works (roughly) like `lambda x: expr` (see [How It Works](#) (page 2106) below).

**modules** : str, optional

Specifies the numeric library to use.

If not specified, *modules* defaults to:

- ["scipy", "numpy"] if SciPy is installed
- ["numpy"] if only NumPy is installed
- ["math", "mpmath", "sympy"] if neither is installed.

That is, SymPy functions are replaced as far as possible by either *scipy* or *numpy* functions if available, and Python's standard library *math*, or *mpmath* functions otherwise.

*modules* can be one of the following types:

- The strings "math", "mpmath", "numpy", "numexpr", "scipy", "sympy", or "tensorflow" or "jax". This uses the corresponding printer and namespace mapping for that module.
- A module (e.g., *math*). This uses the global namespace of the module. If the module is one of the above known modules, it will also use the corresponding printer and namespace mapping (i.e., *modules=numpy* is equivalent to *modules="numpy"*).
- A dictionary that maps names of SymPy functions to arbitrary functions (e.g., {'sin': custom\_sin}).
- A list that contains a mix of the arguments above, with higher priority given to entries appearing first (e.g., to use the NumPy module but override the *sin* function with a custom version, you can use [{'sin': custom\_sin}, 'numpy']).

**dummify** : bool, optional

Whether or not the variables in the provided expression that are not valid Python identifiers are substituted with dummy symbols.

This allows for undefined functions like `Function('f')(t)` to be supplied as arguments. By default, the variables are only dummified if they are not valid Python identifiers.

Set *dummify*=True to replace all arguments with dummy symbols (if *args* is not a string) - for example, to ensure that the arguments do not redefine any built-in names.

**cse** : bool, or callable, optional

Large expressions can be computed more efficiently when common subexpressions are identified and precomputed before being used multiple time. Finding the subexpressions will make creation of the 'lambdify' function slower, however.

When True, `sympy.simplify.cse` is used, otherwise (the default) the user may pass a function matching the *cse* signature.

## Explanation

For example, to convert the SymPy expression  $\sin(x) + \cos(x)$  to an equivalent NumPy function that numerically evaluates it:

```
>>> from sympy import sin, cos, symbols, lambdify
>>> import numpy as np
>>> x = symbols('x')
>>> expr = sin(x) + cos(x)
>>> expr
sin(x) + cos(x)
>>> f = lambdify(x, expr, 'numpy')
>>> a = np.array([1, 2])
>>> f(a)
[1.38177329 0.49315059]
```

The primary purpose of this function is to provide a bridge from SymPy expressions to numerical libraries such as NumPy, SciPy, NumExpr, mpmath, and tensorflow. In general, SymPy functions do not work with objects from other libraries, such as NumPy arrays, and functions from numeric libraries like NumPy or mpmath do not work on SymPy expressions. `lambdify` bridges the two by converting a SymPy expression to an equivalent numeric function.

The basic workflow with `lambdify` is to first create a SymPy expression representing whatever mathematical function you wish to evaluate. This should be done using only SymPy functions and expressions. Then, use `lambdify` to convert this to an equivalent function for numerical evaluation. For instance, above we created `expr` using the SymPy symbol `x` and SymPy functions `sin` and `cos`, then converted it to an equivalent NumPy function `f`, and called it on a NumPy array `a`.

## Examples

```
>>> from sympy.utilities.lambdify import implemented_function
>>> from sympy import sqrt, sin, Matrix
>>> from sympy import Function
>>> from sympy.abc import w, x, y, z
```

```
>>> f = lambdify(x, x**2)
>>> f(2)
4
>>> f = lambdify((x, y, z), [z, y, x])
>>> f(1,2,3)
[3, 2, 1]
>>> f = lambdify(x, sqrt(x))
>>> f(4)
2.0
>>> f = lambdify((x, y), sin(x*y)**2)
>>> f(0, 5)
0.0
>>> row = lambdify((x, y), Matrix((x, x + y)).T, modules='sympy')
>>> row(1, 2)
Matrix([[1, 3]])
```

lambdify can be used to translate SymPy expressions into mpmath functions. This may be preferable to using evalf (which uses mpmath on the backend) in some cases.

```
>>> f = lambdify(x, sin(x), 'mpmath')
>>> f(1)
0.8414709848078965
```

Tuple arguments are handled and the lambdified function should be called with the same type of arguments as were used to create the function:

```
>>> f = lambdify((x, (y, z)), x + y)
>>> f(1, (2, 4))
3
```

The flatten function can be used to always work with flattened arguments:

```
>>> from sympy.utilities.iterables import flatten
>>> args = w, (x, (y, z))
>>> vals = 1, (2, (3, 4))
>>> f = lambdify(flatten(args), w + x + y + z)
>>> f(*flatten(vals))
10
```

Functions present in expr can also carry their own numerical implementations, in a callable attached to the `_imp_` attribute. This can be used with undefined functions using the `implemented_function` factory:

```
>>> f = implemented_function(Function('f'), lambda x: x+1)
>>> func = lambdify(x, f(x))
>>> func(4)
5
```

lambdify always prefers `_imp_` implementations to implementations in other namespaces, unless the `use_ims` input parameter is False.

Usage with Tensorflow:

```
>>> import tensorflow as tf
>>> from sympy import Max, sin, lambdify
>>> from sympy.abc import x
```

```
>>> f = Max(x, sin(x))
>>> func = lambdify(x, f, 'tensorflow')
```

After tensorflow v2, eager execution is enabled by default. If you want to get the compatible result across tensorflow v1 and v2 as same as this tutorial, run this line.

```
>>> tf.compat.v1.enable_eager_execution()
```

If you have eager execution enabled, you can get the result out immediately as you can use numpy.

If you pass tensorflow objects, you may get an EagerTensor object instead of value.



```
>>> result = func(tf.constant(1.0))
>>> print(result)
tf.Tensor(1.0, shape=(), dtype=float32)
>>> print(result.__class__)
<class 'tensorflow.python.framework.ops.EagerTensor'>
```

You can use `.numpy()` to get the numpy value of the tensor.

```
>>> result.numpy()
1.0
```

```
>>> var = tf.Variable(2.0)
>>> result = func(var) # also works for tf.Variable and tf.Placeholder
>>> result.numpy()
2.0
```

And it works with any shape array.

```
>>> tensor = tf.constant([[1.0, 2.0], [3.0, 4.0]])
>>> result = func(tensor)
>>> result.numpy()
[[1. 2.]
 [3. 4.]]
```

## Notes

- For functions involving large array calculations, numexpr can provide a significant speedup over numpy. Please note that the available functions for numexpr are more limited than numpy but can be expanded with `implemented_function` and user defined subclasses of `Function`. If specified, numexpr may be the only option in modules. The official list of numexpr functions can be found at: [https://numexpr.readthedocs.io/en/latest/user\\_guide.html#supported-functions](https://numexpr.readthedocs.io/en/latest/user_guide.html#supported-functions)
- In the above examples, the generated functions can accept scalar values or numpy arrays as arguments. However, in some cases the generated function relies on the input being a numpy array:

```
>>> import numpy
>>> from sympy import Piecewise
>>> from sympy.testing.pytest import ignore_warnings
>>> f = lambdify(x, Piecewise((x, x <= 1), (1/x, x > 1)), "numpy")
```

```
>>> with ignore_warnings(RuntimeWarning):
... f(numpy.array([-1, 0, 1, 2]))
[-1. 0. 1. 0.5]
```

```
>>> f(0)
Traceback (most recent call last):
...
ZeroDivisionError: division by zero
```

In such cases, the input should be wrapped in a numpy array:

```
>>> with ignore_warnings(RuntimeWarning):
... float(f(numpy.array([0])))
0.0
```

Or if numpy functionality is not required another module can be used:

```
>>> f = lambdify(x, Piecewise((x, x <= 1), (1/x, x > 1)), "math")
>>> f(0)
0
```

## How It Works

When using this function, it helps a great deal to have an idea of what it is doing. At its core, `lambdify` is nothing more than a namespace translation, on top of a special printer that makes some corner cases work properly.

To understand `lambdify`, first we must properly understand how Python namespaces work. Say we had two files. One called `sin_cos_sympy.py`, with

```
sin_cos_sympy.py

from sympy.functions.elementary.trigonometric import (cos, sin)

def sin_cos(x):
 return sin(x) + cos(x)
```

and one called `sin_cos_numpy.py` with

```
sin_cos_numpy.py

from numpy import sin, cos

def sin_cos(x):
 return sin(x) + cos(x)
```

The two files define an identical function `sin_cos`. However, in the first file, `sin` and `cos` are defined as the SymPy `sin` and `cos`. In the second, they are defined as the NumPy versions.

If we were to import the first file and use the `sin_cos` function, we would get something like

```
>>> from sin_cos_sympy import sin_cos
>>> sin_cos(1)
cos(1) + sin(1)
```

On the other hand, if we imported `sin_cos` from the second file, we would get

```
>>> from sin_cos_numpy import sin_cos
>>> sin_cos(1)
1.38177329068
```

In the first case we got a symbolic output, because it used the symbolic `sin` and `cos` functions from SymPy. In the second, we got a numeric result, because `sin_cos` used the numeric `sin` and `cos` functions from NumPy. But notice that the versions of `sin` and `cos` that were used was not inherent to the `sin_cos` function definition. Both `sin_cos` definitions are exactly the same. Rather, it was based on the names defined at the module where the `sin_cos` function was defined.

The key point here is that when function in Python references a name that is not defined in the function, that name is looked up in the “global” namespace of the module where that function is defined.

Now, in Python, we can emulate this behavior without actually writing a file to disk using the `exec` function. `exec` takes a string containing a block of Python code, and a dictionary that should contain the global variables of the module. It then executes the code “in” that dictionary, as if it were the module globals. The following is equivalent to the `sin_cos` defined in `sin_cos_sympy.py`:

```
>>> import sympy
>>> module_dictionary = {'sin': sympy.sin, 'cos': sympy.cos}
>>> exec('''
... def sin_cos(x):
... return sin(x) + cos(x)
... ''', module_dictionary)
>>> sin_cos = module_dictionary['sin_cos']
>>> sin_cos(1)
cos(1) + sin(1)
```

and similarly with `sin_cos_numpy`:

```
>>> import numpy
>>> module_dictionary = {'sin': numpy.sin, 'cos': numpy.cos}
>>> exec('''
... def sin_cos(x):
... return sin(x) + cos(x)
... ''', module_dictionary)
>>> sin_cos = module_dictionary['sin_cos']
>>> sin_cos(1)
1.38177329068
```

So now we can get an idea of how `lambdify` works. The name “`lambdify`” comes from the fact that we can think of something like `lambdify(x, sin(x) + cos(x), 'numpy')` as `lambda x: sin(x) + cos(x)`, where `sin` and `cos` come from the `numpy` namespace. This is also why the symbols argument is first in `lambdify`, as opposed to most SymPy functions where it comes after the expression: to better mimic the `lambda` keyword.

`lambdify` takes the input expression (like `sin(x) + cos(x)`) and

1. Converts it to a string
2. Creates a module globals dictionary based on the modules that are passed in (by default, it uses the NumPy module)
3. Creates the string “`def func({vars}): return {expr}`”, where `{vars}` is the list of variables separated by commas, and `{expr}` is the string created in step 1., then `exec``s that string with the module globals namespace and returns ``func`.

In fact, functions returned by `lambdify` support inspection. So you can see exactly how they are defined by using `inspect.getsource`, or `??` if you are using IPython or the Jupyter notebook.

```
>>> f = lambdify(x, sin(x) + cos(x))
>>> import inspect
>>> print(inspect.getsource(f))
def _lambdifygenerated(x):
 return sin(x) + cos(x)
```

This shows us the source code of the function, but not the namespace it was defined in. We can inspect that by looking at the `__globals__` attribute of `f`:

```
>>> f.__globals__['sin']
<ufunc 'sin'>
>>> f.__globals__['cos']
<ufunc 'cos'>
>>> f.__globals__['sin'] is numpy.sin
True
```

This shows us that `sin` and `cos` in the namespace of `f` will be `numpy.sin` and `numpy.cos`.

Note that there are some convenience layers in each of these steps, but at the core, this is how `lambdify` works. Step 1 is done using the `LambdaPrinter` printers defined in the printing module (see [sympy.printing.lambdarepr](#) (page 2170)). This allows different SymPy expressions to define how they should be converted to a string for different modules. You can change which printer `lambdify` uses by passing a custom printer in to the printer argument.

Step 2 is augmented by certain translations. There are default translations for each module, but you can provide your own by passing a list to the `modules` argument. For instance,

```
>>> def mysin(x):
... print('taking the sin of', x)
... return numpy.sin(x)
...
>>> f = lambdify(x, sin(x), [{'sin': mysin}, 'numpy'])
>>> f(1)
taking the sin of 1
0.8414709848078965
```

The `globals` dictionary is generated from the list by merging the dictionary `{'sin': mysin}` and the module dictionary for NumPy. The merging is done so that earlier items take precedence, which is why `mysin` is used above instead of `numpy.sin`.

If you want to modify the way `lambdify` works for a given function, it is usually easiest to do so by modifying the `globals` dictionary as such. In more complicated cases, it may be necessary to create and pass in a custom printer.

Finally, step 3 is augmented with certain convenience operations, such as the addition of a docstring.

Understanding how `lambdify` works can make it easier to avoid certain gotchas when using it. For instance, a common mistake is to create a `lambdified` function for one module (say, NumPy), and pass it objects from another (say, a SymPy expression).

For instance, say we create

```
>>> from sympy.abc import x
>>> f = lambdify(x, x + 1, 'numpy')
```

Now if we pass in a NumPy array, we get that array plus 1

```
>>> import numpy
>>> a = numpy.array([1, 2])
>>> f(a)
[2 3]
```

But what happens if you make the mistake of passing in a SymPy expression instead of a NumPy array:

```
>>> f(x + 1)
x + 2
```

This worked, but it was only by accident. Now take a different lambdified function:

```
>>> from sympy import sin
>>> g = lambdify(x, x + sin(x), 'numpy')
```

This works as expected on NumPy arrays:

```
>>> g(a)
[1.84147098 2.90929743]
```

But if we try to pass in a SymPy expression, it fails

```
>>> try:
... g(x + 1)
... # NumPy release after 1.17 raises TypeError instead of
... # AttributeError
... except (AttributeError, TypeError):
... raise AttributeError()
Traceback (most recent call last):
...
AttributeError:
```

Now, let's look at what happened. The reason this fails is that `g` calls `numpy.sin` on the input expression, and `numpy.sin` does not know how to operate on a SymPy object. **As a general rule, NumPy functions do not know how to operate on SymPy expressions, and SymPy functions do not know how to operate on NumPy arrays. This is why lambdify exists: to provide a bridge between SymPy and NumPy.**

However, why is it that `f` did work? That's because `f` does not call any functions, it only adds 1. So the resulting function that is created, `def _lambdifygenerated(x): return x + 1` does not depend on the globals namespace it is defined in. Thus it works, but only by accident. A future version of `lambdify` may remove this behavior.

Be aware that certain implementation details described here may change in future versions of SymPy. The API of passing in custom modules and printers will not change, but the details of how a lambda function is created may change. However, the basic idea will remain the same, and understanding it will be helpful to understanding the behavior of `lambdify`.

**In general: you should create lambdified functions for one module (say, NumPy), and only pass it input types that are compatible with that module (say, NumPy arrays).** Remember that by default, if the module argument is not provided, `lambdify` creates functions using the NumPy and SciPy namespaces.

## Memoization

`sympy.utilities.memoization.assoc_recurrence_memo(base_seq)`

Memo decorator for associated sequences defined by recurrence starting from base  
`base_seq(n)` - callable to get base sequence elements

XXX works only for  $P_n0 = \text{base\_seq}(0)$  cases XXX works only for  $m \leq n$  cases

`sympy.utilities.memoization.recurrence_memo(initial)`

Memo decorator for sequences defined by recurrence

See usage examples e.g. in the `specfun/combinatorial` module

## Miscellaneous

Miscellaneous stuff that does not really fit anywhere else.

`sympy.utilities.misc.as_int(n, strict=True)`

Convert the argument to a builtin integer.

The return value is guaranteed to be equal to the input. `ValueError` is raised if the input has a non-integral value. When `strict` is `True`, this uses `__index__` and when it is `False` it uses `int`.

## Examples

```
>>> from sympy.utilities.misc import as_int
>>> from sympy import sqrt, S
```

The function is primarily concerned with sanitizing input for functions that need to work with builtin integers, so anything that is unambiguously an integer should be returned as an int:

```
>>> as_int(S(3))
3
```

Floats, being of limited precision, are not assumed to be exact and will raise an error unless the `strict` flag is `False`. This precision issue becomes apparent for large floating point numbers:

```
>>> big = 1e23
>>> type(big) is float
True
>>> big == int(big)
True
>>> as_int(big)
```

(continues on next page)

[illegible]

```
>>> one = sqrt(3 + 2*sqrt(2)) - sqrt(2)
>>> int(one) == 1
True
>>> as_int(one)
Traceback (most recent call last):
...
ValueError: ... is not an integer
```

Print `*args` if `SYMPY_DEBUG` is `True`, else do nothing.

If SYMPY\_DEBUG is True, it will print a nice execution tree with arguments and results of all decorated functions, else do nothing.

Strips leading and trailing empty lines from a copy of `s`, then dedents, fills and returns it.

Additional keyword arguments will be passed to `textwrap.fill()`.

*strlines* (page 2114), *rawlines* (page 2112)

Try to find ‘executable’ in the directories listed in ‘path’ (a string listing directories separated by ‘os.pathsep’; defaults to os.environ[‘PATH’]). Returns the complete filename or None if not found

Return function name of  $x$  (if defined) else the  $type(x)$ . If `short` is `True` and there is a shorter alias for the result, return the alias.

## Examples

```
>>> from sympy.utilities.misc import func_name
>>> from sympy import Matrix
>>> from sympy.abc import x
>>> func_name(Matrix.eye(3))
'MutableDenseMatrix'
>>> func_name(x < 1)
'StrictLessThan'
>>> func_name(x < 1, short=True)
'Lt'
```

`sympy.utilities.misc.ordinal(num)`

Return ordinal number string of num, e.g. 1 becomes 1st.

`sympy.utilities.misc.rawlines(s)`

Return a cut-and-pastable string that, when printed, is equivalent to the input. Use this when there is more than one line in the string. The string returned is formatted so it can be indented nicely within tests; in some cases it is wrapped in the `dedent` function which has to be imported from `textwrap`.

## Examples

Note: because there are characters in the examples below that need to be escaped because they are themselves within a triple quoted docstring, expressions below look more complicated than they would be if they were printed in an interpreter window.

```
>>> from sympy.utilities.misc import rawlines
>>> from sympy import TableForm
>>> s = str(TableForm([[1, 10]], headings=(None, ['a', 'bee'])))
>>> print(rawlines(s))
(
 'a bee\n'
 '-----\n'
 '1 10 '
)
>>> print(rawlines('''this
... that'''))
dedent('''\
 this
 that''')
```

```
>>> print(rawlines('''this
... that
... '''))
dedent('''\
 this
 that
 ''')
```



```
>>> s = """this
... is a triple '''
... """
>>> print(rawlines(s))
dedent("""\
 this
 is a triple '''
 """)
```

```
>>> print(rawlines(''''this
... that
... '''))
(
 'this\n'
 'that\n'
 ' '
)
```

**See also:**

[\*filldedent\*](#) (page 2111), [\*strlines\*](#) (page 2114)

`sympy.utilities.misc.replace(string, *reps)`

Return string with all keys in `reps` replaced with their corresponding values, longer strings first, irrespective of the order they are given. `reps` may be passed as tuples or a single mapping.

**Examples**

```
>>> from sympy.utilities.misc import replace
>>> replace('foo', {'oo': 'ar', 'f': 'b'})
'bar'
>>> replace("spamham sha", ("spam", "eggs"), ("sha", "md5"))
'eggsham md5'
```

There is no guarantee that a unique answer will be obtained if keys in a mapping overlap (i.e. are the same length and have some identical sequence at the beginning/end):

```
>>> reps = [
... ('ab', 'x'),
... ('bc', 'y')]
>>> replace('abc', *reps) in ('xc', 'ay')
True
```

## References

[R977]

`sympy.utilities.misc.strlines(s, c=64, short=False)`

Return a cut-and-pastable string that, when printed, is equivalent to the input. The lines will be surrounded by parentheses and no line will be longer than `c` (default 64) characters. If the line contains newlines characters, the *rawlines* result will be returned. If `short` is `True` (default is `False`) then if there is one line it will be returned without bounding parentheses.

## Examples

```
>>> from sympy.utilities.misc import strlines
>>> q = 'this is a long string that should be broken into shorter lines'
>>> print(strlines(q, 40))
(
'this is a long string that should be b'
'roken into shorter lines'
)
>>> q == (
... 'this is a long string that should be b'
... 'roken into shorter lines'
...)
True
```

**See also:**

*filldedent* (page 2111), *rawlines* (page 2112)

`sympy.utilities.misc.translate(s, a, b=None, c=None)`

Return `s` where characters have been replaced or deleted.

## Syntax

**`translate(s, None, deletechars):`**

all characters in `deletechars` are deleted

**`translate(s, map [,deletechars]):`**

all characters in `deletechars` (if provided) are deleted then the replacements defined by `map` are made; if the keys of `map` are strings then the longer ones are handled first. Multicharacter deletions should have a value of `''`.

**`translate(s, oldchars, newchars, deletechars)`**

all characters in `deletechars` are deleted then each character in `oldchars` is replaced with the corresponding character in `newchars`

## Examples

```
>>> from sympy.utilities.misc import translate
>>> abc = 'abc'
>>> translate(abc, None, 'a')
'bc'
>>> translate(abc, {'a': 'x'}, 'c')
'xb'
>>> translate(abc, {'abc': 'x', 'a': 'y'})
'x'
```

```
>>> translate('abcd', 'ac', 'AC', 'd')
'AbC'
```

There is no guarantee that a unique answer will be obtained if keys in a mapping overlap are the same length and have some identical sequences at the beginning/end:

```
>>> translate(abc, {'ab': 'x', 'bc': 'y'}) in ('xc', 'ay')
True
```

## PKGDATA

pkgdata is a simple, extensible way for a package to acquire data file resources.

The getResource function is equivalent to the standard idioms, such as the following minimal implementation:

```
import sys, os

def getResource(identifier, pkgname=__name__):
 pkgpath = os.path.dirname(sys.modules[pkgname].__file__)
 path = os.path.join(pkgpath, identifier)
 return open(os.path.normpath(path), mode='rb')
```

When a `__loader__` is present on the module given by `__name__`, it will defer getResource to its `get_data` implementation and return it as a file-like object (such as StringIO).

`sympy.utilities.pkgdata.get_resource(identifier, pkgname='sympy.utilities.pkgdata')`

Acquire a readable object for a given package name and identifier. An IOError will be raised if the resource cannot be found.

For example:

```
mydata = get_resource('mypkgdata.jpg').read()
```

Note that the package name must be fully qualified, if given, such that it would be found in `sys.modules`.

In some cases, getResource will return a real file object. In that case, it may be useful to use its name attribute to get the path rather than use it as a file-like object. For example, you may be handing data off to a C API.

## Source Code Inspection

This module adds several functions for interactive source code inspection.

`sympy.utilities.source.get_class(lookup_view)`

Convert a string version of a class name to the object.

For example, `get_class('sympy.core.Basic')` will return class `Basic` located in module `sympy.core`

`sympy.utilities.source.get_mod_func(callback)`

splits the string path to a class into a string path to the module and the name of the class.

## Examples

```
>>> from sympy.utilities.source import get_mod_func
>>> get_mod_func('sympy.core.basic.Basic')
('sympy.core.basic', 'Basic')
```

`sympy.utilities.source.source(object)`

Prints the source code of a given object.

Deprecated since version 1.3: The `source()` function is deprecated. Use `inspect.getsource()` or `??` in IPython/Jupyter instead.

## Timing Utilities

Simple tools for timing functions' execution, when IPython is not available.

`sympy.utilities.timeutils.timed(func, setup='pass', limit=None)`

Adaptively measure execution time of a function.

## Interactive

Helper module for setting up interactive SymPy sessions.

## Session

Tools for setting up interactive sessions.

`sympy.interactive.session.enable_automatic_int_sympification(shell)`

Allow IPython to automatically convert integer literals to Integer.

`sympy.interactive.session.enable_automatic_symbols(shell)`

Allow IPython to automatically create symbols (`isympy -a`).

`sympy.interactive.session.init_ipython_session(shell=None, argv=[],  
auto_symbols=False,  
auto_int_to_Integer=False)`

Construct new IPython session.

```
sympy.interactive.session.init_python_session()
```

Construct new Python session.

```
sympy.interactive.session.init_session(ipython=None, pretty_print=True,
 order=None, use_unicode=None,
 use_latex=None, quiet=False,
 auto_symbols=False,
 auto_int_to_Integer=False, str_printer=None,
 pretty_printer=None, latex_printer=None,
 argv=[])
```

Initialize an embedded IPython or Python session. The IPython session is initiated with the `-pylab` option, without the numpy imports, so that matplotlib plotting can be interactive.

#### Parameters

**pretty\_print: boolean**

If True, use `pretty_print` to stringify; if False, use `sstrrepr` to stringify.

**order: string or None**

There are a few different settings for this parameter: `lex` (default), which is lexicographic order; `grlex`, which is graded lexicographic order; `grevlex`, which is reversed graded lexicographic order; `old`, which is used for compatibility reasons and for long expressions; `None`, which sets it to `lex`.

**use\_unicode: boolean or None**

If True, use unicode characters; if False, do not use unicode characters.

**use\_latex: boolean or None**

If True, use latex rendering if IPython GUI's; if False, do not use latex rendering.

**quiet: boolean**

If True, `init_session` will not print messages regarding its status; if False, `init_session` will print messages regarding its status.

**auto\_symbols: boolean**

If True, IPython will automatically create symbols for you. If False, it will not. The default is False.

**auto\_int\_to\_Integer: boolean**

If True, IPython will automatically wrap int literals with `Integer`, so that things like `1/2` give `Rational(1, 2)`. If False, it will not. The default is False.

**ipython: boolean or None**

If True, printing will initialize for an IPython console; if False, printing will initialize for a normal console; The default is `None`, which automatically determines whether we are in an ipython instance or not.

**str\_printer: function, optional, default=None**

A custom string printer function. This should mimic `sympy.printing.sstrrepr()`.

**pretty\_printer: function, optional, default=None**

A custom pretty printer. This should mimic `sympy.printing.pretty()`.

**latex\_printer: function, optional, default=None**

A custom LaTeX printer. This should mimic `sympy.printing.latex()`  
This should mimic `sympy.printing.latex()`.

**argv: list of arguments for IPython**

See `sympy.bin.isympy` for options that can be used to initialize IPython.

## Examples

```
>>> from sympy import init_session, Symbol, sin, sqrt
>>> sin(x)
NameError: name 'x' is not defined
>>> init_session()
>>> sin(x)
sin(x)
>>> sqrt(5)
 $\sqrt{5}$
>>> init_session(pretty_print=False)
>>> sqrt(5)
sqrt(5)
>>> y + x + y**2 + x**2
x**2 + x + y**2 + y
>>> init_session(order='grlex')
>>> y + x + y**2 + x**2
x**2 + y**2 + x + y
>>> init_session(order='grevlex')
>>> y * x**2 + x * y**2
x**2*y + x*y**2
>>> init_session(order='old')
>>> x**2 + y**2 + x + y
x + y + x**2 + y**2
>>> theta = Symbol('theta')
>>> theta
theta
>>> init_session(use_unicode=True)
>>> theta
θ
```

**See also:**

[\*sympy.interactive.printing.init\\_printing\*](#) (page 2119)  
for examples and the rest of the parameters.

`sympy.interactive.session.int_to_Integer(s)`

Wrap integer literals with Integer.

This is based on the decistmt example from <http://docs.python.org/library/tokenize.html>.

Only integer literals are converted. Float literals are left alone.

## Examples

```
>>> from sympy import Integer # noqa: F401
>>> from sympy.interactive.session import int_to_Integer
>>> s = '1.2 + 1/2 - 0x12 + a1'
>>> int_to_Integer(s)
'1.2 +Integer (1)/Integer (2)-Integer (0x12)+a1 '
>>> s = 'print (1/2)'
>>> int_to_Integer(s)
'print (Integer (1)/Integer (2))'
>>> exec(s)
0.5
>>> exec(int_to_Integer(s))
1/2
```

## Printing

Tools for setting up printing in interactive sessions.

`sympy.interactive.printing.init_printing(pretty_print=True, order=None, use_unicode=None, use_latex=None, wrap_line=None, num_columns=None, no_global=False, ip=None, euler=False, forecolor=None, backcolor='Transparent', fontsize='10pt', latex_mode='plain', print_builtin=True, str_printer=None, pretty_printer=None, latex_printer=None, scale=1.0, **settings)`

Initializes pretty-printer depending on the environment.

### Parameters

**pretty\_print** : bool, default=True

If True, use `pretty_print()` (page 2139) to stringify or the provided pretty printer; if False, use `sstrrepr()` (page 2178) to stringify or the provided string printer.

**order** : string or None, default='lex'

There are a few different settings for this parameter: 'lex' (default), which is lexographic order; 'grlex', which is graded lexographic order; 'grevlex', which is reversed graded lexographic order; 'old', which is used for compatibility reasons and for long expressions; None, which sets it to lex.

**use\_unicode** : bool or None, default=None

If True, use unicode characters; if False, do not use unicode characters; if None, make a guess based on the environment.

**use\_latex** : string, bool, or None, default=None

If True, use default LaTeX rendering in GUI interfaces (png and mathjax); if False, do not use LaTeX rendering; if None, make a guess based on the environment; if 'png', enable LaTeX rendering with an external LaTeX compiler, falling back to matplotlib if external compilation fails; if 'matplotlib', enable LaTeX rendering with matplotlib; if 'mathjax', enable LaTeX text generation, for example MathJax rendering in IPython notebook or text rendering in LaTeX documents; if 'svg', enable LaTeX rendering with an external latex compiler, no fallback

**wrap\_line** : bool

If True, lines will wrap at the end; if False, they will not wrap but continue as one line. This is only relevant if `pretty_print` is True.

**num\_columns** : int or None, default=None

If int, number of columns before wrapping is set to `num_columns`; if None, number of columns before wrapping is set to terminal width. This is only relevant if `pretty_print` is True.

**no\_global** : bool, default=False

If True, the settings become system wide; if False, use just for this console/session.

**ip** : An interactive console

This can either be an instance of IPython, or a class that derives from `code.InteractiveConsole`.

**euler** : bool, optional, default=False

Loads the euler package in the LaTeX preamble for handwritten style fonts (<http://www.ctan.org/pkg/euler>).

**forecolor** : string or None, optional, default=None

DVI setting for foreground color. None means that either 'Black', 'White', or 'Gray' will be selected based on a guess of the IPython terminal color setting. See notes.

**backcolor** : string, optional, default='Transparent'

DVI setting for background color. See notes.

**fontsize** : string or int, optional, default='10pt'

A font size to pass to the LaTeX documentclass function in the preamble. Note that the options are limited by the documentclass. Consider using `scale` instead.

**latex\_mode** : string, optional, default='plain'

The mode used in the LaTeX printer. Can be one of: {'inline'|'plain'|'equation'|'equation\*'}.

**print\_builtint** : boolean, optional, default=True



If True then floats and integers will be printed. If False the printer will only print SymPy types.

**str\_printer** : function, optional, default=None

A custom string printer function. This should mimic `sstrrepr()` (page 2178).

**pretty\_printer** : function, optional, default=None

A custom pretty printer. This should mimic `pretty()` (page 2139).

**latex\_printer** : function, optional, default=None

A custom LaTeX printer. This should mimic `latex()` (page 2170).

**scale** : float, optional, default=1.0

Scale the LaTeX output when using the 'png' or 'svg' backends. Useful for high dpi screens.

**settings** :

Any additional settings for the latex and pretty commands can be used to fine-tune the output.

## Examples

```
>>> from sympy.interactive import init_printing
>>> from sympy import Symbol, sqrt
>>> from sympy.abc import x, y
>>> sqrt(5)
sqrt(5)
>>> init_printing(pretty_print=True)
>>> sqrt(5)

$$\sqrt{5}$$

>>> theta = Symbol('theta')
>>> init_printing(use_unicode=True)
>>> theta
 θ
>>> init_printing(use_unicode=False)
>>> theta
theta
>>> init_printing(order='lex')
>>> str(y + x + y**2 + x**2)
x**2 + x + y**2 + y
>>> init_printing(order='grlex')
>>> str(y + x + y**2 + x**2)
x**2 + x + y**2 + y
>>> init_printing(order='grevlex')
>>> str(y * x**2 + x * y**2)
x**2*y + x*y**2
>>> init_printing(order='old')
>>> str(x**2 + y**2 + x + y)
x**2 + x + y**2 + y
>>> init_printing(num_columns=10)
```

(continues on next page)

(continued from previous page)

```
>>> x**2 + x + y**2 + y
x + y +
x**2 + y**2
```

## Notes

The foreground and background colors can be selected when using 'png' or 'svg' LaTeX rendering. Note that before the `init_printing` command is executed, the LaTeX rendering is handled by the IPython console and not SymPy.

The colors can be selected among the 68 standard colors known to dvips, for a list see [R553]. In addition, the background color can be set to 'Transparent' (which is the default value).

When using the 'Auto' foreground color, the guess is based on the `colors` variable in the IPython console, see [R554]. Hence, if that variable is set correctly in your IPython console, there is a high chance that the output will be readable, although manual settings may be needed.

### See also:

[\*sympy.printing.latex\*](#) (page 2170), [\*sympy.printing.pretty\*](#) (page 2139)

## References

[R553], [R554]

## Parsing

### Parsing Functions Reference

`sympy.parsing.sympy_parser.parse_expr(s: str, local_dict: ~typing.Optional[~typing.Dict[str, ~typing.Any]] = None, transformations: ~typing.Union[~typing.Tuple[~typing.Callable[[~typing.List[~typing.List[~typing.Dict[str, ~typing.Any]], ~typing.Dict[str, ~typing.Any]], ~typing.Dict[str, ~typing.Any]], ~typing.List[~typing.Tuple[int, str]]], ...], str] = (<function lambda_notation>, <function auto_symbol>, <function repeated_decimals>, <function auto_number>, <function factorial_notation>), global_dict: ~typing.Optional[~typing.Dict[str, ~typing.Any]] = None, evaluate=True)`

Converts the string `s` to a SymPy expression, in `local_dict`

#### Parameters

**s** : str

The string to parse.

**local\_dict** : dict, optional

A dictionary of local variables to use when parsing.

**global\_dict** : dict, optional

A dictionary of global variables. By default, this is initialized with `from sympy import *`; provide this parameter to override this behavior (for instance, to parse "Q & S").

**transformations** : tuple or str

A tuple of transformation functions used to modify the tokens of the parsed expression before evaluation. The default transformations convert numeric literals into their SymPy equivalents, convert undefined variables into SymPy symbols, and allow the use of standard mathematical factorial notation (e.g.  $x!$ ). Selection via string is available (see below).

**evaluate** : bool, optional

When False, the order of the arguments will remain as they were in the string and automatic simplification that would normally occur is suppressed. (see examples)

## Examples

```
>>> from sympy.parsing.sympy_parser import parse_expr
>>> parse_expr("1/2")
1/2
>>> type(_)
<class 'sympy.core.numbers.Half'>
>>> from sympy.parsing.sympy_parser import standard_transformations,\
... implicit_multiplication_application
>>> transformations = (standard_transformations +
... (implicit_multiplication_application,))
>>> parse_expr("2x", transformations=transformations)
2*x
```

When `evaluate=False`, some automatic simplifications will not occur:

```
>>> parse_expr("2**3"), parse_expr("2**3", evaluate=False)
(8, 2**3)
```

In addition the order of the arguments will not be made canonical. This feature allows one to tell exactly how the expression was entered:

```
>>> a = parse_expr('1 + x', evaluate=False)
>>> b = parse_expr('x + 1', evaluate=0)
>>> a == b
False
>>> a.args
(1, x)
>>> b.args
(x, 1)
```

Note, however, that when these expressions are printed they will appear the same:

```
>>> assert str(a) == str(b)
```

As a convenience, transformations can be seen by printing transformations:

```
>>> from sympy.parsing.sympy_parser import transformations
```

```
>>> print(transformations)
0: lambda_notation
1: auto_symbol
2: repeated_decimals
3: auto_number
4: factorial_notation
5: implicit_multiplication_application
6: convert_xor
7: implicit_application
8: implicit_multiplication
9: convert_equals_signs
10: function_exponentiation
11: rationalize
```

The T object provides a way to select these transformations:

```
>>> from sympy.parsing.sympy_parser import T
```

If you print it, you will see the same list as shown above.

```
>>> str(T) == str(transformations)
True
```

Standard slicing will return a tuple of transformations:

```
>>> T[:5] == standard_transformations
True
```

So T can be used to specify the parsing transformations:

```
>>> parse_expr("2x", transformations=T[:5])
Traceback (most recent call last):
...
SyntaxError: invalid syntax
>>> parse_expr("2x", transformations=T[:6])
2*x
>>> parse_expr('.3', transformations=T[3, 11])
3/10
>>> parse_expr('.3x', transformations=T[:])
3*x/10
```

As a further convenience, strings 'implicit' and 'all' can be used to select 0-5 and all the transformations, respectively.

```
>>> parse_expr('.3x', transformations='all')
3*x/10
```

**See also:**

[stringify\\_expr](#) (page 2125), [eval\\_expr](#) (page 2125), [standard\\_transformations](#) (page 2126), [implicit\\_multiplication\\_application](#) (page 2128)

```
sympy.parsing.sympy_parser.stringify_expr(s: str, local_dict: Dict[str, Any],
 global_dict: Dict[str, Any],
 transformations:
 Tuple[Callable[[List[Tuple[int, str]],
 Dict[str, Any], Dict[str, Any]],
 List[Tuple[int, str]]], ...]) → str
```

Converts the string *s* to Python code, in *local\_dict*

Generally, `parse_expr` should be used.

```
sympy.parsing.sympy_parser.eval_expr(code, local_dict: Dict[str, Any], global_dict:
 Dict[str, Any])
```

Evaluate Python code generated by `stringify_expr`.

Generally, `parse_expr` should be used.

```
sympy.parsing.maxima.parse_maxima(str, globals=None, name_dict={})
```

```
sympy.parsing.mathematica.parse_mathematica(s)
```

Translate a string containing a Wolfram Mathematica expression to a SymPy expression.

If the translator is unable to find a suitable SymPy expression, the FullForm of the Mathematica expression will be output, using SymPy Function objects as nodes of the syntax tree.

**Examples**

```
>>> from sympy.parsing.mathematica import parse_mathematica
>>> parse_mathematica("Sin[x]^2 Tan[y]")
sin(x)**2*tan(y)
>>> e = parse_mathematica("F[7,5,3]")
>>> e
F(7, 5, 3)
>>> from sympy import Function, Max, Min
>>> e.replace(Function("F"), lambda *x: Max(*x)*Min(*x))
21
```

Both standard input form and Mathematica full form are supported:

```
>>> parse_mathematica("x*(a + b)")
x*(a + b)
>>> parse_mathematica("Times[x, Plus[a, b]]")
x*(a + b)
```

To get a matrix from Wolfram's code:

```
>>> m = parse_mathematica("{ {a, b}, {c, d} }")
>>> m
((a, b), (c, d))
>>> from sympy import Matrix
```

(continues on next page)

(continued from previous page)

```
>>> Matrix(m)
Matrix([
[a, b],
[c, d]])
```

If the translation into equivalent SymPy expressions fails, an SymPy expression equivalent to Wolfram Mathematica's "FullForm" will be created:

```
>>> parse_mathematica("x_.")
Optional(Pattern(x, Blank()))
>>> parse_mathematica("Plus @@ {x, y, z}")
Apply(Plus, (x, y, z))
>>> parse_mathematica("f[x_, 3] := x^3 /; x > 0")
SetDelayed(f(Pattern(x, Blank()), 3), Condition(x**3, x > 0))
```

## Parsing Transformations Reference

A transformation is a function that accepts the arguments `tokens`, `local_dict`, `global_dict` and returns a list of transformed tokens. They can be used by passing a list of functions to `parse_expr()` (page 2122) and are applied in the order given.

`sympy.parsing.sympy_parser.standard_transformations:`  
**Tuple[Callable[[List[Tuple[int, str]], Dict[str, Any], Dict[str, Any]], List[Tuple[int, str]], ...] = (<function lambda\_notation>, <function auto\_symbol>, <function repeated\_decimals>, <function auto\_number>, <function factorial\_notation>)**

Standard transformations for `parse_expr()` (page 2122). Inserts calls to `Symbol` (page 976), `Integer` (page 987), and other SymPy datatypes and allows the use of standard factorial notation (e.g. `x!`).

`sympy.parsing.sympy_parser.split_symbols(tokens: List[Tuple[int, str]], local_dict: Dict[str, Any], global_dict: Dict[str, Any])`

Splits symbol names for implicit multiplication.

Intended to let expressions like `xyz` be parsed as `x*y*z`. Does not split Greek character names, so `theta` will *not* become `t*h*e*t*a`. Generally this should be used with `implicit_multiplication`.

`sympy.parsing.sympy_parser.split_symbols_custom(predicate: Callable[[str], bool])`  
 Creates a transformation that splits symbol names.

`predicate` should return `True` if the symbol name is to be split.

For instance, to retain the default behavior but avoid splitting certain symbol names, a predicate like this would work:

```
>>> from sympy.parsing.sympy_parser import (parse_expr, _token_
->splittable,
... standard_transformations, implicit_multiplication,
... split_symbols_custom)
>>> def can_split(symbol):
... if symbol not in ('list', 'of', 'unsplittable', 'names'):
... return _token_splittable(symbol)
```

(continues on next page)

(continued from previous page)

```

... return False
...
>>> transformation = split_symbols_custom(can_split)
>>> parse_expr('unsplittable', transformations=standard_transformations +
... (transformation, implicit_multiplication))
unsplittable

```

`sympy.parsing.sympy_parser.implicit_multiplication(tokens: List[Tuple[int, str]], local_dict: Dict[str, Any], global_dict: Dict[str, Any]) → List[Tuple[int, str]]`

Makes the multiplication operator optional in most cases.

Use this before `implicit_application()` (page 2127), otherwise expressions like `sin 2x` will be parsed as `x * sin(2)` rather than `sin(2*x)`.

### Examples

```

>>> from sympy.parsing.sympy_parser import (parse_expr,
... standard_transformations, implicit_multiplication)
>>> transformations = standard_transformations + (implicit_
... multiplication,)
>>> parse_expr('3 x y', transformations=transformations)
3*x*y

```

`sympy.parsing.sympy_parser.implicit_application(tokens: List[Tuple[int, str]], local_dict: Dict[str, Any], global_dict: Dict[str, Any]) → List[Tuple[int, str]]`

Makes parentheses optional in some cases for function calls.

Use this after `implicit_multiplication()` (page 2127), otherwise expressions like `sin 2x` will be parsed as `x * sin(2)` rather than `sin(2*x)`.

### Examples

```

>>> from sympy.parsing.sympy_parser import (parse_expr,
... standard_transformations, implicit_application)
>>> transformations = standard_transformations + (implicit_application,)
>>> parse_expr('cot z + csc z', transformations=transformations)
cot(z) + csc(z)

```

`sympy.parsing.sympy_parser.function_exponentiation(tokens: List[Tuple[int, str]], local_dict: Dict[str, Any], global_dict: Dict[str, Any])`

Allows functions to be exponentiated, e.g. `cos**2(x)`.

## Examples

```
>>> from sympy.parsing.sympy_parser import (parse_expr,
... standard_transformations, function_exponentiation)
>>> transformations = standard_transformations + (function_
 ↪ exponentiation,)
>>> parse_expr('sin**4(x)', transformations=transformations)
sin(x)**4
```

`sympy.parsing.sympy_parser.implicit_multiplication_application`(*result:*  
*List[Tuple[int, str]], local\_dict:*  
*Dict[str, Any], global\_dict:*  
*Dict[str, Any]) →*  
*List[Tuple[int, str]]*

Allows a slightly relaxed syntax.

- Parentheses for single-argument method calls are optional.
- Multiplication is implicit.
- Symbol names can be split (i.e. spaces are not needed between symbols).
- Functions can be exponentiated.

## Examples

```
>>> from sympy.parsing.sympy_parser import (parse_expr,
... standard_transformations, implicit_multiplication_application)
>>> parse_expr("10sin**2 x**2 + 3xyz + tan theta",
... transformations=(standard_transformations +
... (implicit_multiplication_application,)))
3*x*y*z + 10*sin(x**2)**2 + tan(theta)
```

`sympy.parsing.sympy_parser.rationalize`(*tokens: List[Tuple[int, str]], local\_dict:*  
*Dict[str, Any], global\_dict: Dict[str, Any])*

Converts floats into Rational. Run AFTER auto\_number.

`sympy.parsing.sympy_parser.convert_xor`(*tokens: List[Tuple[int, str]], local\_dict:*  
*Dict[str, Any], global\_dict: Dict[str, Any])*

Treats XOR, ^, as exponentiation, \*\*.

These are included in `sympy.parsing.sympy_parser.standard_transformations` (page 2126) and generally don't need to be manually added by the user.

`sympy.parsing.sympy_parser.lambda_notation`(*tokens: List[Tuple[int, str]], local\_dict:*  
*Dict[str, Any], global\_dict: Dict[str, Any])*

Substitutes "lambda" with its SymPy equivalent Lambda(). However, the conversion does not take place if only "lambda" is passed because that is a syntax error.

`sympy.parsing.sympy_parser.auto_symbol`(*tokens: List[Tuple[int, str]], local\_dict:*  
*Dict[str, Any], global\_dict: Dict[str, Any])*

Inserts calls to Symbol/Function for undefined variables.



```
sympy.parsing.sympy_parser.repeated_decimals(tokens: List[Tuple[int, str]], local_dict:
 Dict[str, Any], global_dict: Dict[str,
 Any])
```

Allows 0.2[1] notation to represent the repeated decimal 0.2111... (19/90)

Run this before `auto_number`.

```
sympy.parsing.sympy_parser.auto_number(tokens: List[Tuple[int, str]], local_dict:
 Dict[str, Any], global_dict: Dict[str, Any])
```

Converts numeric literals to use SymPy equivalents.

Complex numbers use `I`, integer literals use `Integer`, and float literals use `Float`.

```
sympy.parsing.sympy_parser.factorial_notation(tokens: List[Tuple[int, str]],
 local_dict: Dict[str, Any], global_dict:
 Dict[str, Any])
```

Allows standard notation for factorial.

## Experimental $\text{\LaTeX}$ Parsing

$\text{\LaTeX}$  parsing was ported from `latex2sympy`. While functional and its API should remain stable, the parsing behavior or backend may change in future releases.

### $\text{\LaTeX}$ Parsing Caveats

The current implementation is experimental. The behavior, parser backend and API might change in the future. Unlike some of the other parsers,  $\text{\LaTeX}$  is designed as a *type-setting* language, not a *computer algebra system* and so can contain typographical conventions that might be interpreted multiple ways.

In its current definition, the parser will at times will fail to fully parse the expression, but not throw a warning:

```
parse_latex(r'x -')
```

Will simply find `x`. What is covered by this behavior will almost certainly change between releases, and become stricter, more relaxed, or some mix.

## $\text{\LaTeX}$ Parsing Functions Reference

```
sympy.parsing.latex.parse_latex(s)
```

Converts the string `s` to a SymPy Expr

### Parameters

`s` : str

The LaTeX string to parse. In Python source containing LaTeX, *raw strings* (denoted with `r''`, like this one) are preferred, as LaTeX makes liberal use of the `\` character, which would trigger escaping in normal Python strings.

## Examples

```
>>> from sympy.parsing.latex import parse_latex
>>> expr = parse_latex(r"\frac {1 + \sqrt {\a}} {\b}")
>>> expr
(sqrt(a) + 1)/b
>>> expr.evalf(4, subs=dict(a=5, b=2))
1.618
```

## LaTeX Parsing Exceptions Reference

`class sympy.parsing.latex.LaTeXParsingError`

## SymPy Expression Reference

`class sympy.parsing.sym_expr.SymPyExpression(source_code=None, mode=None)`

Class to store and handle SymPy expressions

This class will hold SymPy Expressions and handle the API for the conversion to and from different languages.

It works with the C and the Fortran Parser to generate SymPy expressions which are stored here and which can be converted to multiple language's source code.

## Notes

The module and its API are currently under development and experimental and can be changed during development.

The Fortran parser does not support numeric assignments, so all the variables have been Initialized to zero.

The module also depends on external dependencies:

- LFortran which is required to use the Fortran parser
- Clang which is required for the C parser

## Examples

Example of parsing C code:

```
>>> from sympy.parsing.sym_expr import SymPyExpression
>>> src = '''
... int a,b;
... float c = 2, d =4;
... '''
>>> a = SymPyExpression(src, 'c')
>>> a.return_expr()
[Declaration(Variable(a, type=intc)),
```

(continues on next page)

(continued from previous page)

```
Declaration(Variable(b, type=intc)),
Declaration(Variable(c, type=float32, value=2.0)),
Declaration(Variable(d, type=float32, value=4.0))]
```

An example of variable definition:

```
>>> from sympy.parsing.sym_expr import SymPyExpression
>>> src2 = '''
... integer :: a, b, c, d
... real :: p, q, r, s
... '''
>>> p = SymPyExpression()
>>> p.convert_to_expr(src2, 'f')
>>> p.convert_to_c()
['int a = 0', 'int b = 0', 'int c = 0', 'int d = 0', 'double p = 0.0',
 → 'double q = 0.0', 'double r = 0.0', 'double s = 0.0']
```

An example of Assignment:

```
>>> from sympy.parsing.sym_expr import SymPyExpression
>>> src3 = '''
... integer :: a, b, c, d, e
... d = a + b - c
... e = b * d + c * e / a
... '''
>>> p = SymPyExpression(src3, 'f')
>>> p.convert_to_python()
['a = 0', 'b = 0', 'c = 0', 'd = 0', 'e = 0', 'd = a + b - c', 'e = b*d_
 → + c*e/a']
```

An example of function definition:

```
>>> from sympy.parsing.sym_expr import SymPyExpression
>>> src = '''
... integer function f(a,b)
... integer, intent(in) :: a, b
... integer :: r
... end function
... '''
>>> a = SymPyExpression(src, 'f')
>>> a.convert_to_python()
['def f(a, b):\n f = 0\n r = 0\n return f']
```

### convert\_to\_c()

Returns a list with the c source code for the SymPy expressions

## Examples

```
>>> from sympy.parsing.sym_expr import SymPyExpression
>>> src2 = '''
... integer :: a, b, c, d
... real :: p, q, r, s
... c = a/b
... d = c/a
... s = p/q
... r = q/p
... '''
>>> p = SymPyExpression()
>>> p.convert_to_expr(src2, 'f')
>>> p.convert_to_c()
['int a = 0', 'int b = 0', 'int c = 0', 'int d = 0', 'double p = 0.0',
↪ 'double q = 0.0', 'double r = 0.0', 'double s = 0.0', 'c = a/b;',
↪ 'd = c/a;', 's = p/q;', 'r = q/p;']
```

**convert\_to\_expr**(src\_code, mode)

Converts the given source code to SymPy Expressions

## Examples

```
>>> from sympy.parsing.sym_expr import SymPyExpression
>>> src3 = '''
... integer function f(a,b) result(r)
... integer, intent(in) :: a, b
... integer :: x
... r = a + b -x
... end function
... '''
>>> p = SymPyExpression()
>>> p.convert_to_expr(src3, 'f')
>>> p.return_expr()
[FunctionDefinition(integer, name=f, parameters=(Variable(a),
↪ Variable(b)), body=CodeBlock(
Declaration(Variable(r, type=integer, value=0)),
Declaration(Variable(x, type=integer, value=0)),
Assignment(Variable(r), a + b - x),
Return(Variable(r))
))]
```

## Attributes

|          |                                                                                                                                   |
|----------|-----------------------------------------------------------------------------------------------------------------------------------|
| src_code | (String) the source code or filename of the source code that is to be converted                                                   |
| mode:    | the mode to determine which parser is to be used according to the language of the source code f or F for Fortran c or C for C/C++ |

### convert\_to\_fortran()

Returns a list with the fortran source code for the SymPy expressions

## Examples

```
>>> from sympy.parsing.sym_expr import SymPyExpression
>>> src2 = '''
... integer :: a, b, c, d
... real :: p, q, r, s
... c = a/b
... d = c/a
... s = p/q
... r = q/p
... '''
>>> p = SymPyExpression(src2, 'f')
>>> p.convert_to_fortran()
[' integer*4 a', ' integer*4 b', ' integer*4 c', '
↳ integer*4 d', ' real*8 p', ' real*8 q', ' real*8 r',
↳ ' real*8 s', ' c = a/b', ' d = c/a', ' s = p/q
↳ ', ' r = q/p']
```

### convert\_to\_python()

Returns a list with Python code for the SymPy expressions

## Examples

```
>>> from sympy.parsing.sym_expr import SymPyExpression
>>> src2 = '''
... integer :: a, b, c, d
... real :: p, q, r, s
... c = a/b
... d = c/a
... s = p/q
... r = q/p
... '''
>>> p = SymPyExpression(src2, 'f')
>>> p.convert_to_python()
['a = 0', 'b = 0', 'c = 0', 'd = 0', 'p = 0.0', 'q = 0.0', 'r = 0.0',
↳ 's = 0.0', 'c = a/b', 'd = c/a', 's = p/q', 'r = q/p']
```

### return\_expr()

Returns the expression list

## Examples

```
>>> from sympy.parsing.sym_expr import SymPyExpression
>>> src3 = '''
... integer function f(a,b)
... integer, intent(in) :: a, b
... integer :: r
... r = a+b
... f = r
... end function
... '''
>>> p = SymPyExpression()
>>> p.convert_to_expr(src3, 'f')
>>> p.return_expr()
[FunctionDefinition(integer, name=f, parameters=(Variable(a),
↳Variable(b)), body=CodeBlock(
Declaration(Variable(f, type=integer, value=0)),
Declaration(Variable(r, type=integer, value=0)),
Assignment(Variable(f), Variable(r)),
Return(Variable(f))
))]
```

## Runtime Installation

The currently-packaged LaTeX parser backend is partially generated with [ANTLR4](#), but to use the parser, you only need the antlr4 Python package available.

Depending on your package manager, you can install the right package with, for example, pip:

```
$ pip install antlr4-python3-runtime==4.10
```

or conda:

```
$ conda install -c conda-forge antlr-python-runtime==4.10
```

The C parser depends on clang and the Fortran parser depends on LFortran. You can install these packages using:

```
$ conda install -c conda-forge lfortran clang
```

## Printing

See the [Printing](#) (page 19) section in tutorial for introduction into printing.

This guide documents the printing system in SymPy and how it works internally.

## Printer Class

Printing subsystem driver

SymPy's printing system works the following way: Any expression can be passed to a designated Printer who then is responsible to return an adequate representation of that expression.

**The basic concept is the following:**

1. Let the object print itself if it knows how.
2. Take the best fitting method defined in the printer.
3. As fall-back use the emptyPrinter method for the printer.

## Which Method is Responsible for Printing?

The whole printing process is started by calling `.doprint(expr)` on the printer which you want to use. This method looks for an appropriate method which can print the given expression in the given style that the printer defines. While looking for the method, it follows these steps:

1. **Let the object print itself if it knows how.**

The printer looks for a specific method in every object. The name of that method depends on the specific printer and is defined under `Printer.printmethod`. For example, `StrPrinter` calls `_sympystr` and `LatexPrinter` calls `_latex`. Look at the documentation of the printer that you want to use. The name of the method is specified there.

This was the original way of doing printing in sympy. Every class had its own `latex`, `mathml`, `str` and `repr` methods, but it turned out that it is hard to produce a high quality printer, if all the methods are spread out that far. Therefore all printing code was combined into the different printers, which works great for built-in SymPy objects, but not that good for user defined classes where it is inconvenient to patch the printers.

2. **Take the best fitting method defined in the printer.**

The printer loops through `expr` classes (class + its bases), and tries to dispatch the work to `_print_<EXPR_CLASS>`

e.g., suppose we have the following class hierarchy:

```

Basic
|
Atom
|
Number
|
Rational

```

then, for `expr=Rational(...)`, the Printer will try to call printer methods in the order as shown in the figure below:

```

p._print(expr)
|
|-- p._print_Rational(expr)
|

```

(continues on next page)

(continued from previous page)

```
|-- p._print_Number(expr)
|
|-- p._print_Atom(expr)
|
|-- p._print_Basic(expr)
```

if `._print_Rational` method exists in the printer, then it is called, and the result is returned back. Otherwise, the printer tries to call `._print_Number` and so on.

### 3. As a fall-back use the `emptyPrinter` method for the printer.

As fall-back `self.emptyPrinter` will be called with the expression. If not defined in the `Printer` subclass this will be the same as `str(expr)`.

## Example of Custom Printer

In the example below, we have a printer which prints the derivative of a function in a shorter form.

```
from sympy.core.symbol import Symbol
from sympy.printing.latex import LatexPrinter, print_latex
from sympy.core.function import UndefinedFunction, Function

class MyLatexPrinter(LatexPrinter):
 """Print derivative of a function of symbols in a shorter form.
 """
 def _print_Derivative(self, expr):
 function, *vars = expr.args
 if not isinstance(type(function), UndefinedFunction) or \
 not all(isinstance(i, Symbol) for i in vars):
 return super()._print_Derivative(expr)

 # If you want the printer to work correctly for nested
 # expressions then use self._print() instead of str() or latex().
 # See the example of nested modulo below in the custom printing
 # method section.
 return "{}_{{{}}}".format(
 self._print(Symbol(function.func.__name__)),
 ''.join(self._print(i) for i in vars))

def print_my_latex(expr):
 """ Most of the printers define their own wrappers for print().
 These wrappers usually take printer settings. Our printer does not have
 any settings.
 """
 print(MyLatexPrinter().doprint(expr))

y = Symbol("y")
x = Symbol("x")
```

(continues on next page)



(continued from previous page)

```
f = Function("f")
expr = f(x, y).diff(x, y)

Print the expression using the normal latex printer and our custom
printer.
print_latex(expr)
print_my_latex(expr)
```

The output of the code above is:

```
\frac{\partial^{\textcolor{teal}{2}}}{\partial x \partial y} f(\textcolor{teal}{\left(x,y \right)})
f_{xy}
```

### Example of Custom Printing Method

In the example below, the latex printing of the modulo operator is modified. This is done by overriding the method `_latex` of `Mod`.

```
>>> from sympy import Symbol, Mod, Integer, print_latex
```

```
>>> # Always use printer._print()
>>> class ModOp(Mod):
... def _latex(self, printer):
... a, b = [printer._print(i) for i in self.args]
... return r"\operatorname{Mod}\textcolor{teal}{\left(}%s, %s\right)}" % (a, b)
```

Comparing the output of our custom operator to the builtin one:

```
>>> x = Symbol('x')
>>> m = Symbol('m')
>>> print_latex(Mod(x, m))
x \bmod m
>>> print_latex(ModOp(x, m))
\operatorname{Mod}\textcolor{teal}{\left(x, m\right)}
```

### Common mistakes

It's important to always use `self._print(obj)` to print subcomponents of an expression when customizing a printer. Mistakes include:

1. Using `self.doprint(obj)` instead:

```
>>> # This example does not work properly, as only the outermost call
 ↪ may use
>>> # doprint.
>>> class ModOpModeWrong(Mod):
... def _latex(self, printer):
... a, b = [printer.doprint(i) for i in self.args]
... return r"\operatorname{Mod}\textcolor{teal}{\left(}%s, %s\right)}" % (a, b)
```

This fails when the mode argument is passed to the printer:

```
>>> print_latex(ModOp(x, m), mode='inline') # ok
\operatorname{Mod}\left(x, m\right)\$
>>> print_latex(ModOpModeWrong(x, m), mode='inline') # bad
\operatorname{Mod}\left(\$x\$, \$m\$\right)\$
```

2. Using `str(obj)` instead:

```
>>> class ModOpNestedWrong(Mod):
... def _latex(self, printer):
... a, b = [str(i) for i in self.args]
... return r"\operatorname{Mod}\left(%s, %s\right)" % (a, b)
```

This fails on nested objects:

```
>>> # Nested modulo.
>>> print_latex(ModOp(ModOp(x, m), Integer(7))) # ok
\operatorname{Mod}\left(\operatorname{Mod}\left(x, m\right), 7\right)\$
>>> print_latex(ModOpNestedWrong(ModOpNestedWrong(x, m), Integer(7))) #
↳bad
\operatorname{Mod}\left(\operatorname{ModOpNestedWrong}\left(x, m\right), 7\right)\$
```

3. Using `LatexPrinter()._print(obj)` instead.

```
>>> from sympy.printing.latex import LatexPrinter
>>> class ModOpSettingsWrong(Mod):
... def _latex(self, printer):
... a, b = [LatexPrinter()._print(i) for i in self.args]
... return r"\operatorname{Mod}\left(%s, %s\right)" % (a, b)
```

This causes all the settings to be discarded in the subobjects. As an example, the `full_prec` setting which shows floats to full precision is ignored:

```
>>> from sympy import Float
>>> print_latex(ModOp(Float(1) * x, m), full_prec=True) # ok
\operatorname{Mod}\left(1.0000000000000000 x, m\right)\$
>>> print_latex(ModOpSettingsWrong(Float(1) * x, m), full_prec=True) #
↳bad
\operatorname{Mod}\left(1.0 x, m\right)\$
```

The main class responsible for printing is `Printer` (see also its [source code](#)):

```
class sympy.printing.printer.Printer(settings=None)
```

Generic printer

Its job is to provide infrastructure for implementing new printers easily.

If you want to define your custom `Printer` or your custom printing method for your custom class then see the example above: [printer\\_example](#) (page 2136) .

**printmethod:** `str = None`

**\_print**(*expr*, *\*\*kwargs*) → `str`

Internal dispatcher

**Tries the following concepts to print an expression:**

1. Let the object print itself if it knows how.
2. Take the best fitting method defined in the printer.
3. As fall-back use the emptyPrinter method for the printer.

**doprint**(*expr*)

Returns printer's representation for *expr* (as a string)

**classmethod set\_global\_settings**(*\*\*settings*)

Set system-wide printing settings.

## PrettyPrinter Class

The pretty printing subsystem is implemented in `sympy.printing.pretty.pretty` by the `PrettyPrinter` class deriving from `Printer`. It relies on the modules `sympy.printing.pretty.stringPict`, and `sympy.printing.pretty.pretty_symbology` for rendering nice-looking formulas.

The module `stringPict` provides a base class `stringPict` and a derived class `prettyForm` that ease the creation and manipulation of formulas that span across multiple lines.

The module `pretty_symbology` provides primitives to construct 2D shapes (hline, vline, etc) together with a technique to use unicode automatically when possible.

**class** `sympy.printing.pretty.pretty.PrettyPrinter`(*settings=None*)

Printer, which converts an expression into 2D ASCII-art figure.

**printmethod:** `str = '_pretty'`

`sympy.printing.pretty.pretty.pretty`(*expr*, \*, *order=None*, *full\_prec='auto'*,  
*use\_unicode=True*, *wrap\_line=False*,  
*num\_columns=None*,  
*use\_unicode\_sqrt\_char=True*,  
*root\_notation=True*, *mat\_symbol\_style='plain'*,  
*imaginary\_unit='i'*, *perm\_cyclic=True*)

Returns a string containing the prettified form of *expr*.

For information on keyword arguments see `pretty_print` function.

`sympy.printing.pretty.pretty.pretty.pretty_print`(*expr*, *\*\*kwargs*)

Prints *expr* in pretty form.

`pprint` is just a shortcut for this function.

### Parameters

**expr** : expression

The expression to print.

**wrap\_line** : bool, optional (default=True)

Line wrapping enabled/disabled.

**num\_columns** : int or None, optional (default=None)

Number of columns before line breaking (default to None which reads the terminal width), useful when using SymPy without terminal.

**use\_unicode** : bool or None, optional (default=None)

Use unicode characters, such as the Greek letter pi instead of the string pi.

**full\_prec** : bool or string, optional (default="auto")

Use full precision.

**order** : bool or string, optional (default=None)

Set to 'none' for long expressions if slow; default is None.

**use\_unicode\_sqrt\_char** : bool, optional (default=True)

Use compact single-character square root symbol (when unambiguous).

**root\_notation** : bool, optional (default=True)

Set to 'False' for printing exponents of the form 1/n in fractional form. By default exponent is printed in root form.

**mat\_symbol\_style** : string, optional (default="plain")

Set to "bold" for printing MatrixSymbols using a bold mathematical symbol face. By default the standard face is used.

**imaginary\_unit** : string, optional (default="i")

Letter to use for imaginary unit when use\_unicode is True. Can be "i" (default) or "j".

## C code printers

This class implements C code printing, i.e. it converts Python expressions to strings of C code (see also C89CodePrinter).

Usage:

```
>>> from sympy.printing import print_ccode
>>> from sympy.functions import sin, cos, Abs, gamma
>>> from sympy.abc import x
>>> print_ccode(sin(x)**2 + cos(x)**2, standard='C89')
pow(sin(x), 2) + pow(cos(x), 2)
>>> print_ccode(2*x + cos(x), assign_to="result", standard='C89')
result = 2*x + cos(x);
>>> print_ccode(Abs(x**2), standard='C89')
fabs(pow(x, 2))
>>> print_ccode(gamma(x**2), standard='C99')
tgamma(pow(x, 2))
```

```
sympy.printing.c.known_functions_C89 = {'Abs': [(<function <lambda>>, 'fabs'),
(<function <lambda>>, 'abs')], 'acos': 'acos', 'asin': 'asin', 'atan': 'atan',
'atan2': 'atan2', 'ceiling': 'ceil', 'cos': 'cos', 'cosh': 'cosh', 'exp':
'exp', 'floor': 'floor', 'log': 'log', 'sin': 'sin', 'sinh': 'sinh', 'sqrt':
'sqrt', 'tan': 'tan', 'tanh': 'tanh'}
```

```
sympy.printing.c.known_functions_C99 = {'Abs': [(<function <lambda>>, 'fabs'),
(<function <lambda>>, 'abs')], 'Cbrt': 'cbrt', 'Max': 'fmax', 'Min': 'fmin',
'acos': 'acos', 'acosh': 'acosh', 'asin': 'asin', 'asinh': 'asinh', 'atan':
'atan', 'atan2': 'atan2', 'atanh': 'atanh', 'ceiling': 'ceil', 'cos': 'cos',
'cosh': 'cosh', 'erf': 'erf', 'erfc': 'erfc', 'exp': 'exp', 'exp2': 'exp2',
'expm1': 'expm1', 'floor': 'floor', 'fma': 'fma', 'gamma': 'tgamma', 'hypot':
'hypot', 'log': 'log', 'log10': 'log10', 'log1p': 'log1p', 'log2': 'log2',
'loggamma': 'lgamma', 'sin': 'sin', 'sinh': 'sinh', 'sqrt': 'sqrt', 'tan':
'tan', 'tanh': 'tanh'}
```

```
class sympy.printing.c.C89CodePrinter(settings=None)
```

A printer to convert Python expressions to strings of C code

```
printmethod: str = '_ccode'
```

```
indent_code(code)
```

Accepts a string of code or a list of code lines

```
class sympy.printing.c.C99CodePrinter(settings=None)
```

```
printmethod: str = '_ccode'
```

```
sympy.printing.c.ccode(expr, assign_to=None, standard='c99', **settings)
```

Converts an expr to a string of c code

#### Parameters

**expr** : Expr

A SymPy expression to be converted.

**assign\_to** : optional

When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, Symbol, MatrixSymbol, or Indexed type. This is helpful in case of line-wrapping, or for expressions that generate multi-line statements.

**standard** : str, optional

String specifying the standard. If your compiler supports a more modern standard you may set this to 'c99' to allow the printer to use more math functions. [default='c89'].

**precision** : integer, optional

The precision for numbers such as pi [default=17].

**user\_functions** : dict, optional

A dictionary where the keys are string representations of either FunctionClass or UndefinedFunction instances and the values are their desired C string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument\_test, cfunction\_string)] or [(argument\_test, cfunction\_formatter)]. See below for examples.

**dereference** : iterable, optional

An iterable of symbols that should be dereferenced in the printed code expression. These would be values passed by address to the function. For example, if dereference=[a], the resulting code would print (\*a) instead of a.

**human** : bool, optional

If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the same information is returned in a tuple of (symbols\_to\_declare, not\_supported\_functions, code\_text). [default=True].

**contract**: bool, optional

If True, Indexed instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting contract=False will not generate loops, instead the user is responsible to provide values for the indices in the code. [default=True].

## Examples

```
>>> from sympy import ccode, symbols, Rational, sin, ceiling, Abs, Function
>>> x, tau = symbols("x, tau")
>>> expr = (2*tau)**Rational(7, 2)
>>> ccode(expr)
'8*M_SQRT2*pow(tau, 7.0/2.0)'
>>> ccode(expr, math_macros={})
'8*sqrt(2)*pow(tau, 7.0/2.0)'
>>> ccode(sin(x), assign_to="s")
's = sin(x);'
>>> from sympy.codegen.ast import real, float80
>>> ccode(expr, type_aliases={real: float80})
'8*M_SQRT2l*powl(tau, 7.0L/2.0L)'
```

Simple custom printing can be defined for certain types by passing a dictionary of {"type": "function"} to the user\_functions kwarg. Alternatively, the dictionary value can be a list of tuples i.e. [(argument\_test, cfunction\_string)].

```
>>> custom_functions = {
... "ceiling": "CEIL",
... "Abs": [(lambda x: not x.is_integer, "fabs"),
... (lambda x: x.is_integer, "ABS")],
... "func": "f"
... }
>>> func = Function('func')
>>> ccode(func(Abs(x) + ceiling(x)), standard='C89', user_
... functions=custom_functions)
'f(fabs(x) + CEIL(x))'
```

or if the C-function takes a subset of the original arguments:

```
>>> ccode(2**x + 3**x, standard='C99', user_functions={'Pow': [
... (lambda b, e: b == 2, lambda b, e: 'exp2(%s)' % e),
... (lambda b, e: b != 2, 'pow')]])
'exp2(x) + pow(3, x)'
```

Piecewise expressions are converted into conditionals. If an assign\_to variable is provided an if statement is created, otherwise the ternary operator is used. Note that if

the Piecewise lacks a default term, represented by (expr, True) then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.

```
>>> from sympy import Piecewise
>>> expr = Piecewise((x + 1, x > 0), (x, True))
>>> print(ccode(expr, tau, standard='C89'))
if (x > 0) {
tau = x + 1;
}
else {
tau = x;
}
```

Support for loops is provided through Indexed types. With contract=True these expressions will be turned into loops, whereas contract=False will just print the assignment expression that should be looped over:

```
>>> from sympy import Eq, IndexedBase, Idx
>>> len_y = 5
>>> y = IndexedBase('y', shape=(len_y,))
>>> t = IndexedBase('t', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y-1,))
>>> i = Idx('i', len_y-1)
>>> e=Eq(Dy[i], (y[i+1]-y[i])/(t[i+1]-t[i]))
>>> ccode(e.rhs, assign_to=e.lhs, contract=False, standard='C89')
'Dy[i] = (y[i + 1] - y[i])/(t[i + 1] - t[i]);'
```

Matrices are also supported, but a MatrixSymbol of the same dimensions must be provided to assign\_to. Note that any expression that can be generated normally can also exist inside a Matrix:

```
>>> from sympy import Matrix, MatrixSymbol
>>> mat = Matrix([x**2, Piecewise((x + 1, x > 0), (x, True)), sin(x)])
>>> A = MatrixSymbol('A', 3, 1)
>>> print(ccode(mat, A, standard='C89'))
A[0] = pow(x, 2);
if (x > 0) {
 A[1] = x + 1;
}
else {
 A[1] = x;
}
A[2] = sin(x);
```

`sympy.printing.c.print_ccode(expr, **settings)`  
 Prints C representation of the given expression.

## C++ code printers

This module contains printers for C++ code, i.e. functions to convert SymPy expressions to strings of C++ code.

Usage:

```
>>> from sympy.printing import cxxcode
>>> from sympy.functions import Min, gamma
>>> from sympy.abc import x
>>> print(cxxcode(Min(gamma(x) - 1, x), standard='C++11'))
std::min(x, std::tgamma(x) - 1)
```

```
class sympy.printing.cxx.CXX98CodePrinter(settings=None)
```

```
 printmethod: str = '_cxxcode'
```

```
class sympy.printing.cxx.CXX11CodePrinter(settings=None)
```

```
 printmethod: str = '_cxxcode'
```

```
sympy.printing.codeprinter.cxxcode(expr, assign_to=None, standard='c++11',
 **settings)
```

C++ equivalent of `ccode()` (page 2141).

## RCodePrinter

This class implements R code printing (i.e. it converts Python expressions to strings of R code).

Usage:

```
>>> from sympy.printing import print_rcode
>>> from sympy.functions import sin, cos, Abs
>>> from sympy.abc import x
>>> print_rcode(sin(x)**2 + cos(x)**2)
sin(x)^2 + cos(x)^2
>>> print_rcode(2*x + cos(x), assign_to="result")
result = 2*x + cos(x);
>>> print_rcode(Abs(x**2))
abs(x^2)
```

```
sympy.printing.rcode.known_functions = {'Abs': 'abs', 'Max': 'max', 'Min':
'min', 'acos': 'acos', 'acosh': 'acosh', 'asin': 'asin', 'asinh': 'asinh',
'atan': 'atan', 'atan2': 'atan2', 'atanh': 'atanh', 'beta': 'beta', 'ceiling':
'ceiling', 'cos': 'cos', 'cosh': 'cosh', 'digamma': 'digamma', 'erf': 'erf',
'exp': 'exp', 'factorial': 'factorial', 'floor': 'floor', 'gamma': 'gamma',
'log': 'log', 'sign': 'sign', 'sin': 'sin', 'sinh': 'sinh', 'sqrt': 'sqrt',
'tan': 'tan', 'tanh': 'tanh', 'trigamma': 'trigamma'}
```

```
class sympy.printing.rcode.RCodePrinter(settings={})
```

A printer to convert SymPy expressions to strings of R code

```
 printmethod: str = '_rcode'
```



**indent\_code**(code)

Accepts a string of code or a list of code lines

`sympy.printing.rcode.rcode(expr, assign_to=None, **settings)`

Converts an expr to a string of r code

#### Parameters

**expr** : Expr

A SymPy expression to be converted.

**assign\_to** : optional

When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, Symbol, MatrixSymbol, or Indexed type. This is helpful in case of line-wrapping, or for expressions that generate multi-line statements.

**precision** : integer, optional

The precision for numbers such as pi [default=15].

**user\_functions** : dict, optional

A dictionary where the keys are string representations of either FunctionClass or UndefinedFunction instances and the values are their desired R string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument\_test, rfunction\_string)] or [(argument\_test, rfunction\_formatter)]. See below for examples.

**human** : bool, optional

If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the same information is returned in a tuple of (symbols\_to\_declare, not\_supported\_functions, code\_text). [default=True].

**contract**: bool, optional

If True, Indexed instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting contract=False will not generate loops, instead the user is responsible to provide values for the indices in the code. [default=True].

## Examples

```
>>> from sympy import rcode, symbols, Rational, sin, ceiling, Abs, u
↳ Function
>>> x, tau = symbols("x, tau")
>>> rcode((2*tau)**Rational(7, 2))
'8*sqrt(2)*tau^(7.0/2.0)'
>>> rcode(sin(x), assign_to="s")
's = sin(x);'
```

Simple custom printing can be defined for certain types by passing a dictionary of {"type": "function"} to the user\_functions kwarg. Alternatively, the dictionary value can be a list of tuples i.e. [(argument\_test, cfunction\_string)].

```
>>> custom_functions = {
... "ceiling": "CEIL",
... "Abs": [(lambda x: not x.is_integer, "fabs"),
... (lambda x: x.is_integer, "ABS")],
... "func": "f"
... }
>>> func = Function('func')
>>> rcode(func(Abs(x) + ceiling(x)), user_functions=custom_functions)
'f(fabs(x) + CEIL(x))'
```

or if the R-function takes a subset of the original arguments:

```
>>> rcode(2**x + 3**x, user_functions={'Pow': [
... (lambda b, e: b == 2, lambda b, e: 'exp2(%s)' % e),
... (lambda b, e: b != 2, 'pow')])})
'exp2(x) + pow(3, x)'
```

Piecewise expressions are converted into conditionals. If an `assign_to` variable is provided an if statement is created, otherwise the ternary operator is used. Note that if the Piecewise lacks a default term, represented by `(expr, True)` then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.

```
>>> from sympy import Piecewise
>>> expr = Piecewise((x + 1, x > 0), (x, True))
>>> print(rcode(expr, assign_to=tau))
tau = ifelse(x > 0,x + 1,x);
```

Support for loops is provided through Indexed types. With `contract=True` these expressions will be turned into loops, whereas `contract=False` will just print the assignment expression that should be looped over:

```
>>> from sympy import Eq, IndexedBase, Idx
>>> len_y = 5
>>> y = IndexedBase('y', shape=(len_y,))
>>> t = IndexedBase('t', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y-1,))
>>> i = Idx('i', len_y-1)
>>> e=Eq(Dy[i], (y[i+1]-y[i])/(t[i+1]-t[i]))
>>> rcode(e.rhs, assign_to=e.lhs, contract=False)
'Dy[i] = (y[i + 1] - y[i])/(t[i + 1] - t[i]);'
```

Matrices are also supported, but a `MatrixSymbol` of the same dimensions must be provided to `assign_to`. Note that any expression that can be generated normally can also exist inside a Matrix:

```
>>> from sympy import Matrix, MatrixSymbol
>>> mat = Matrix([x**2, Piecewise((x + 1, x > 0), (x, True)), sin(x)])
>>> A = MatrixSymbol('A', 3, 1)
>>> print(rcode(mat, A))
A[0] = x^2;
A[1] = ifelse(x > 0,x + 1,x);
A[2] = sin(x);
```

`sympy.printing.rcode.print_rcode(expr, **settings)`

Prints R representation of the given expression.

## Fortran Printing

The `fcode` function translates a sympy expression into Fortran code. The main purpose is to take away the burden of manually translating long mathematical expressions. Therefore the resulting expression should also require no (or very little) manual tweaking to make it compilable. The optional arguments of `fcode` can be used to fine-tune the behavior of `fcode` in such a way that manual changes in the result are no longer needed.

```
sympy.printing.fortran.fcode(expr, assign_to=None, **settings)
```

Converts an `expr` to a string of fortran code

### Parameters

**expr** : Expr

A SymPy expression to be converted.

**assign\_to** : optional

When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, `Symbol`, `MatrixSymbol`, or `Indexed` type. This is helpful in case of line-wrapping, or for expressions that generate multi-line statements.

**precision** : integer, optional

DEPRECATED. Use `type_mappings` instead. The precision for numbers such as `pi` [default=17].

**user\_functions** : dict, optional

A dictionary where keys are `FunctionClass` instances and values are their string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument\_test, cfunction\_string)]. See below for examples.

**human** : bool, optional

If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the same information is returned in a tuple of (`symbols_to_declare`, `not_supported_functions`, `code_text`). [default=True].

**contract**: bool, optional

If True, `Indexed` instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting `contract=False` will not generate loops, instead the user is responsible to provide values for the indices in the code. [default=True].

**source\_format** : optional

The source format can be either 'fixed' or 'free'. [default='fixed']

**standard** : integer, optional

The Fortran standard to be followed. This is specified as an integer. Acceptable standards are 66, 77, 90, 95, 2003, and 2008. Default

is 77. Note that currently the only distinction internally is between standards before 95, and those 95 and after. This may change later as more features are added.

**name\_mangling** : bool, optional

If True, then the variables that would become identical in case-insensitive Fortran are mangled by appending different number of `_` at the end. If False, SymPy Will not interfere with naming of variables. [default=True]

## Examples

```
>>> from sympy import fcode, symbols, Rational, sin, ceiling, floor
>>> x, tau = symbols("x, tau")
>>> fcode((2*tau)**Rational(7, 2))
' 8*sqrt(2.0d0)*tau**(7.0d0/2.0d0)'
>>> fcode(sin(x), assign_to="s")
' s = sin(x)'
```

Custom printing can be defined for certain types by passing a dictionary of "type" : "function" to the `user_functions` kwarg. Alternatively, the dictionary value can be a list of tuples i.e. [(argument\_test, cfunction\_string)].

```
>>> custom_functions = {
... "ceiling": "CEIL",
... "floor": [(lambda x: not x.is_integer, "FLOOR1"),
... (lambda x: x.is_integer, "FLOOR2")]
... }
>>> fcode(floor(x) + ceiling(x), user_functions=custom_functions)
' CEIL(x) + FLOOR1(x)'
```

Piecewise expressions are converted into conditionals. If an `assign_to` variable is provided an if statement is created, otherwise the ternary operator is used. Note that if the Piecewise lacks a default term, represented by (expr, True) then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.

```
>>> from sympy import Piecewise
>>> expr = Piecewise((x + 1, x > 0), (x, True))
>>> print(fcode(expr, tau))
 if (x > 0) then
 tau = x + 1
 else
 tau = x
 end if
```

Support for loops is provided through Indexed types. With `contract=True` these expressions will be turned into loops, whereas `contract=False` will just print the assignment expression that should be looped over:

```
>>> from sympy import Eq, IndexedBase, Idx
>>> len_y = 5
>>> y = IndexedBase('y', shape=(len_y,))
```

(continues on next page)

(continued from previous page)

```

>>> t = IndexedBase('t', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y-1,))
>>> i = Idx('i', len_y-1)
>>> e=Eq(Dy[i], (y[i+1]-y[i])/(t[i+1]-t[i]))
>>> fcode(e.rhs, assign_to=e.lhs, contract=False)
' Dy(i) = (y(i + 1) - y(i))/(t(i + 1) - t(i))'

```

Matrices are also supported, but a `MatrixSymbol` of the same dimensions must be provided to `assign_to`. Note that any expression that can be generated normally can also exist inside a `Matrix`:

```

>>> from sympy import Matrix, MatrixSymbol
>>> mat = Matrix([x**2, Piecewise((x + 1, x > 0), (x, True)), sin(x)])
>>> A = MatrixSymbol('A', 3, 1)
>>> print(fcode(mat, A))
 A(1, 1) = x**2
 if (x > 0) then
 A(2, 1) = x + 1
 else
 A(2, 1) = x
 end if
 A(3, 1) = sin(x)

```

`sympy.printing.fortran.print_fcode(expr, **settings)`

Prints the Fortran representation of the given expression.

See `fcode` for the meaning of the optional arguments.

**class** `sympy.printing.fortran.FCodePrinter(settings=None)`

A printer to convert SymPy expressions to strings of Fortran code

**printmethod:** `str = '_fcode'`

**indent\_code**(code)

Accepts a string of code or a list of code lines

Two basic examples:

```

>>> from sympy import *
>>> x = symbols("x")
>>> fcode(sqrt(1-x**2))
' sqrt(1 - x**2)'
>>> fcode((3 + 4*I)/(1 - conjugate(x)))
' (cmplx(3,4))/(1 - conjg(x))'

```

An example where line wrapping is required:

```

>>> expr = sqrt(1-x**2).series(x,n=20).removeO()
>>> print(fcode(expr))
-715.0d0/65536.0d0*x**18 - 429.0d0/32768.0d0*x**16 - 33.0d0/
@ 2048.0d0*x**14 - 21.0d0/1024.0d0*x**12 - 7.0d0/256.0d0*x**10 -
@ 5.0d0/128.0d0*x**8 - 1.0d0/16.0d0*x**6 - 1.0d0/8.0d0*x**4 - 1.0d0
@ /2.0d0*x**2 + 1

```

In case of line wrapping, it is handy to include the assignment so that lines are wrapped properly when the assignment part is added.

```
>>> print(fcode(expr, assign_to="var"))
var = -715.0d0/65536.0d0*x**18 - 429.0d0/32768.0d0*x**16 - 33.0d0/
@ 2048.0d0*x**14 - 21.0d0/1024.0d0*x**12 - 7.0d0/256.0d0*x**10 -
@ 5.0d0/128.0d0*x**8 - 1.0d0/16.0d0*x**6 - 1.0d0/8.0d0*x**4 - 1.0d0
@ /2.0d0*x**2 + 1
```

For piecewise functions, the `assign_to` option is mandatory:

```
>>> print(fcode(Piecewise((x,x<1),(x**2,True)), assign_to="var"))
if (x < 1) then
 var = x
else
 var = x**2
end if
```

Note that by default only top-level piecewise functions are supported due to the lack of a conditional operator in Fortran 77. Inline conditionals can be supported using the `merge` function introduced in Fortran 95 by setting of the `kwargs` `standard=95`:

```
>>> print(fcode(Piecewise((x,x<1),(x**2,True)), standard=95))
merge(x, x**2, x < 1)
```

Loops are generated if there are Indexed objects in the expression. This also requires use of the `assign_to` option.

```
>>> A, B = map(IndexedBase, ['A', 'B'])
>>> m = Symbol('m', integer=True)
>>> i = Idx('i', m)
>>> print(fcode(2*B[i], assign_to=A[i]))
do i = 1, m
 A(i) = 2*B(i)
end do
```

Repeated indices in an expression with Indexed objects are interpreted as summation. For instance, code for the trace of a matrix can be generated with

```
>>> print(fcode(A[i, i], assign_to=x))
x = 0
do i = 1, m
 x = x + A(i, i)
end do
```

By default, number symbols such as `pi` and `E` are detected and defined as Fortran parameters. The precision of the constants can be tuned with the `precision` argument. Parameter definitions are easily avoided using the `N` function.

```
>>> print(fcode(x - pi**2 - E))
parameter (E = 2.7182818284590452d0)
parameter (pi = 3.1415926535897932d0)
x - pi**2 - E
>>> print(fcode(x - pi**2 - E, precision=25))
```

(continues on next page)

(continued from previous page)

```

parameter (E = 2.718281828459045235360287d0)
parameter (pi = 3.141592653589793238462643d0)
x - pi**2 - E
>>> print(fcode(N(x - pi**2, 25)))
x - 9.869604401089358618834491d0

```

When some functions are not part of the Fortran standard, it might be desirable to introduce the names of user-defined functions in the Fortran expression.

```

>>> print(fcode(1 - gamma(x)**2, user_functions={'gamma': 'mygamma'}))
1 - mygamma(x)**2

```

However, when the `user_functions` argument is not provided, `fcode` will generate code which assumes that a function of the same name will be provided by the user. A comment will be added to inform the user of the issue:

```

>>> print(fcode(1 - gamma(x)**2))
C Not supported in Fortran:
C gamma
1 - gamma(x)**2

```

The printer can be configured to omit these comments:

```

>>> print(fcode(1 - gamma(x)**2, allow_unknown_functions=True))
1 - gamma(x)**2

```

By default the output is human readable code, ready for copy and paste. With the option `human=False`, the return value is suitable for post-processing with source code generators that write routines with multiple instructions. The return value is a three-tuple containing: (i) a set of number symbols that must be defined as 'Fortran parameters', (ii) a list functions that cannot be translated in pure Fortran and (iii) a string of Fortran code. A few examples:

```

>>> fcode(1 - gamma(x)**2, human=False)
(set(), {gamma(x)}, ' 1 - gamma(x)**2')
>>> fcode(1 - sin(x)**2, human=False)
(set(), set(), ' 1 - sin(x)**2')
>>> fcode(x - pi**2, human=False)
(({pi, '3.1415926535897932d0'}), set(), ' x - pi**2')

```

## Mathematica code printing





**class** sympy.printing.mathematica.MCodePrinter(*settings*={})

A printer to convert Python expressions to strings of the Wolfram's Mathematica code

**printmethod:** *str* = '\_mcode'

sympy.printing.mathematica.mathematica\_code(*expr*, **\*\*settings**)

Converts an *expr* to a string of the Wolfram Mathematica code

### Examples

```
>>> from sympy import mathematica_code as mcode, symbols, sin
>>> x = symbols('x')
>>> mcode(sin(x).series(x).removeO())
'(1/120)*x^5 - 1/6*x^3 + x'
```

### Maple code printing

**class** sympy.printing.maple.MapleCodePrinter(*settings*=None)

Printer which converts a SymPy expression into a maple code.

**printmethod:** *str* = '\_maple'

sympy.printing.maple.maple\_code(*expr*, *assign\_to*=None, **\*\*settings**)

Converts *expr* to a string of Maple code.

#### Parameters

**expr** : Expr

A SymPy expression to be converted.

**assign\_to** : optional

When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, Symbol, MatrixSymbol, or Indexed type. This can be helpful for expressions that generate multi-line statements.

**precision** : integer, optional

The precision for numbers such as pi [default=16].

**user\_functions** : dict, optional

A dictionary where keys are FunctionClass instances and values are their string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument\_test, cfunction\_string)]. See below for examples.

**human** : bool, optional

If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the same information is returned in a tuple of (symbols\_to\_declare, not\_supported\_functions, code\_text). [default=True].

**contract: bool, optional**

If True, Indexed instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting `contract=False` will not generate loops, instead the user is responsible to provide values for the indices in the code. [default=True].

**inline: bool, optional**

If True, we try to create single-statement code instead of multiple statements. [default=True].

`sympy.printing.maple.print_maple_code(expr, **settings)`

Prints the Maple representation of the given expression.

See `maple_code()` (page 2154) for the meaning of the optional arguments.

**Examples**

```
>>> from sympy import print_maple_code, symbols
>>> x, y = symbols('x y')
>>> print_maple_code(x, assign_to=y)
y := x
```

**Javascript Code printing**

```
sympy.printing.jrcode.known_functions = {'Abs': 'Math.abs', 'Max': 'Math.max',
'Min': 'Math.min', 'acos': 'Math.acos', 'acosh': 'Math.acosh', 'asin':
'Math.asin', 'asinh': 'Math.asinh', 'atan': 'Math.atan', 'atan2':
'Math.atan2', 'atanh': 'Math.atanh', 'ceiling': 'Math.ceil', 'cos':
'Math.cos', 'cosh': 'Math.cosh', 'exp': 'Math.exp', 'floor': 'Math.floor',
'log': 'Math.log', 'sign': 'Math.sign', 'sin': 'Math.sin', 'sinh':
'Math.sinh', 'tan': 'Math.tan', 'tanh': 'Math.tanh'}
```

`class sympy.printing.jrcode.JavascriptCodePrinter(settings={})`

“A Printer to convert Python expressions to strings of JavaScript code

**printmethod:** `str = '_javascript'`

**indent\_code**(code)

Accepts a string of code or a list of code lines

`sympy.printing.jrcode.jrcode(expr, assign_to=None, **settings)`

Converts an `expr` to a string of javascript code

**Parameters**

**expr** : Expr

A SymPy expression to be converted.

**assign\_to** : optional

When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, Symbol,

MatrixSymbol, or Indexed type. This is helpful in case of line-wrapping, or for expressions that generate multi-line statements.

**precision** : integer, optional

The precision for numbers such as pi [default=15].

**user\_functions** : dict, optional

A dictionary where keys are FunctionClass instances and values are their string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument\_test, js\_function\_string)]. See below for examples.

**human** : bool, optional

If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the same information is returned in a tuple of (symbols\_to\_declare, not\_supported\_functions, code\_text). [default=True].

**contract**: bool, optional

If True, Indexed instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting contract=False will not generate loops, instead the user is responsible to provide values for the indices in the code. [default=True].

## Examples

```
>>> from sympy import jscode, symbols, Rational, sin, ceiling, Abs
>>> x, tau = symbols("x, tau")
>>> jscode((2*tau)**Rational(7, 2))
'8*Math.sqrt(2)*Math.pow(tau, 7/2)'
>>> jscode(sin(x), assign_to="s")
's = Math.sin(x);'
```

Custom printing can be defined for certain types by passing a dictionary of “type” : “function” to the user\_functions kwarg. Alternatively, the dictionary value can be a list of tuples i.e. [(argument\_test, js\_function\_string)].

```
>>> custom_functions = {
... "ceiling": "CEIL",
... "Abs": [(lambda x: not x.is_integer, "fabs"),
... (lambda x: x.is_integer, "ABS")]
... }
>>> jscode(Abs(x) + ceiling(x), user_functions=custom_functions)
'fabs(x) + CEIL(x)'
```

Piecewise expressions are converted into conditionals. If an assign\_to variable is provided an if statement is created, otherwise the ternary operator is used. Note that if the Piecewise lacks a default term, represented by (expr, True) then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.

```
>>> from sympy import Piecewise
>>> expr = Piecewise((x + 1, x > 0), (x, True))
>>> print(jrcode(expr, tau))
if (x > 0) {
 tau = x + 1;
}
else {
 tau = x;
}
```

Support for loops is provided through Indexed types. With `contract=True` these expressions will be turned into loops, whereas `contract=False` will just print the assignment expression that should be looped over:

```
>>> from sympy import Eq, IndexedBase, Idx
>>> len_y = 5
>>> y = IndexedBase('y', shape=(len_y,))
>>> t = IndexedBase('t', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y-1,))
>>> i = Idx('i', len_y-1)
>>> e=Eq(Dy[i], (y[i+1]-y[i])/(t[i+1]-t[i]))
>>> jrcode(e.rhs, assign_to=e.lhs, contract=False)
'Dy[i] = (y[i + 1] - y[i])/(t[i + 1] - t[i]);'
```

Matrices are also supported, but a `MatrixSymbol` of the same dimensions must be provided to `assign_to`. Note that any expression that can be generated normally can also exist inside a Matrix:

```
>>> from sympy import Matrix, MatrixSymbol
>>> mat = Matrix([x**2, Piecewise((x + 1, x > 0), (x, True)), sin(x)])
>>> A = MatrixSymbol('A', 3, 1)
>>> print(jrcode(mat, A))
A[0] = Math.pow(x, 2);
if (x > 0) {
 A[1] = x + 1;
}
else {
 A[1] = x;
}
A[2] = Math.sin(x);
```

## Julia code printing

```
sympy.printing.julia.known_fcns_src1 = ['sin', 'cos', 'tan', 'cot', 'sec',
'csc', 'asin', 'acos', 'atan', 'acot', 'asec', 'acsc', 'sinh', 'cosh', 'tanh',
'coth', 'sech', 'csch', 'asinh', 'acosh', 'atanh', 'acoth', 'asech', 'acsch',
'sinc', 'atan2', 'sign', 'floor', 'log', 'exp', 'cbrt', 'sqrt', 'erf', 'erfc',
'erfi', 'factorial', 'gamma', 'digamma', 'trigamma', 'polygamma', 'beta',
'airyai', 'airyaiprime', 'airybi', 'airybiprime', 'besselj', 'bessely',
'besseli', 'besselk', 'erfinv', 'erfcinv']
```

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

```
sympy.printing.julia.known_fcns_src2 = {'Abs': 'abs', 'ceiling': 'ceil',
'conjugate': 'conj', 'hankel1': 'hankelh1', 'hankel2': 'hankelh2', 'im':
'imag', 're': 'real'}
```

```
class sympy.printing.julia.JuliaCodePrinter(settings={})
```

A printer to convert expressions to strings of Julia code.

**printmethod:** `str = '_julia'`

**indent\_code**(*code*)

Accepts a string of code or a list of code lines

```
sympy.printing.julia.julia_code(expr, assign_to=None, **settings)
```

Converts *expr* to a string of Julia code.

#### Parameters

**expr** : Expr

A SymPy expression to be converted.

**assign\_to** : optional

When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, `Symbol`, `MatrixSymbol`, or `Indexed` type. This can be helpful for expressions that generate multi-line statements.

**precision** : integer, optional

The precision for numbers such as pi [default=16].

**user\_functions** : dict, optional

A dictionary where keys are `FunctionClass` instances and values are their string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument\_test, cfunction\_string)]. See below for examples.

**human** : bool, optional

If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the same information is returned in a tuple of (symbols\_to\_declare, not\_supported\_functions, code\_text). [default=True].

**contract:** bool, optional

If True, `Indexed` instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting `contract=False` will not generate loops, instead the user is responsible to provide values for the indices in the code. [default=True].

**inline:** bool, optional

If True, we try to create single-statement code instead of multiple statements. [default=True].

## Examples

```
>>> from sympy import julia_code, symbols, sin, pi
>>> x = symbols('x')
>>> julia_code(sin(x).series(x).remove0())
'x .^ 5 / 120 - x .^ 3 / 6 + x'
```

```
>>> from sympy import Rational, ceiling
>>> x, y, tau = symbols("x, y, tau")
>>> julia_code((2*tau)**Rational(7, 2))
'8 * sqrt(2) * tau .^ (7 // 2)'
```

Note that element-wise (Hadamard) operations are used by default between symbols. This is because its possible in Julia to write “vectorized” code. It is harmless if the values are scalars.

```
>>> julia_code(sin(pi*x*y), assign_to="s")
's = sin(pi * x .* y)'
```

If you need a matrix product “\*” or matrix power “^”, you can specify the symbol as a `MatrixSymbol`.

```
>>> from sympy import Symbol, MatrixSymbol
>>> n = Symbol('n', integer=True, positive=True)
>>> A = MatrixSymbol('A', n, n)
>>> julia_code(3*pi*A**3)
'(3 * pi) * A ^ 3'
```

This class uses several rules to decide which symbol to use a product. Pure numbers use “\*”, Symbols use “.” and MatrixSymbols use “\*”. A `HadamardProduct` can be used to specify componentwise multiplication “.” of two `MatrixSymbols`. There is currently there is no easy way to specify scalar symbols, so sometimes the code might have some minor cosmetic issues. For example, suppose  $x$  and  $y$  are scalars and  $A$  is a Matrix, then while a human programmer might write “ $(x^2 y) A^3$ ”, we generate:

```
>>> julia_code(x**2*y*A**3)
'(x .^ 2 .* y) * A ^ 3'
```

Matrices are supported using Julia inline notation. When using `assign_to` with matrices, the name can be specified either as a string or as a `MatrixSymbol`. The dimensions must align in the latter case.

```
>>> from sympy import Matrix, MatrixSymbol
>>> mat = Matrix([[x**2, sin(x), ceiling(x)]])
>>> julia_code(mat, assign_to='A')
'A = [x .^ 2 sin(x) ceil(x)]'
```

Piecewise expressions are implemented with logical masking by default. Alternatively, you can pass “`inline=False`” to use if-else conditionals. Note that if the `Piecewise` lacks a default term, represented by `(expr, True)` then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.

```
>>> from sympy import Piecewise
>>> pw = Piecewise((x + 1, x > 0), (x, True))
```

(continues on next page)

(continued from previous page)

```
>>> julia_code(pw, assign_to=tau)
'tau = ((x > 0) ? (x + 1) : (x))'
```

Note that any expression that can be generated normally can also exist inside a Matrix:

```
>>> mat = Matrix([[x**2, pw, sin(x)]])
>>> julia_code(mat, assign_to='A')
'A = [x .^ 2 ((x > 0) ? (x + 1) : (x)) sin(x)]'
```

Custom printing can be defined for certain types by passing a dictionary of “type” : “function” to the `user_functions` kwarg. Alternatively, the dictionary value can be a list of tuples i.e., [(argument\_test, cfunction\_string)]. This can be used to call a custom Julia function.

```
>>> from sympy import Function
>>> f = Function('f')
>>> g = Function('g')
>>> custom_functions = {
... "f": "existing_julia_fcn",
... "g": [(lambda x: x.is_Matrix, "my_mat_fcn"),
... (lambda x: not x.is_Matrix, "my_fcn")]
... }
>>> mat = Matrix([[1, x]])
>>> julia_code(f(x) + g(x) + g(mat), user_functions=custom_functions)
'existing_julia_fcn(x) + my_fcn(x) + my_mat_fcn([1 x])'
```

Support for loops is provided through Indexed types. With `contract=True` these expressions will be turned into loops, whereas `contract=False` will just print the assignment expression that should be looped over:

```
>>> from sympy import Eq, IndexedBase, Idx
>>> len_y = 5
>>> y = IndexedBase('y', shape=(len_y,))
>>> t = IndexedBase('t', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y-1,))
>>> i = Idx('i', len_y-1)
>>> e = Eq(Dy[i], (y[i+1]-y[i])/(t[i+1]-t[i]))
>>> julia_code(e.rhs, assign_to=e.lhs, contract=False)
'Dy[i] = (y[i + 1] - y[i]) ./ (t[i + 1] - t[i])'
```

## Octave (and Matlab) Code printing

```
sympy.printing.octave.known_fcns_src1 = ['sin', 'cos', 'tan', 'cot', 'sec',
'csc', 'asin', 'acos', 'acot', 'atan', 'atan2', 'asec', 'acsc', 'sinh',
'cosh', 'tanh', 'coth', 'csch', 'sech', 'asinh', 'acosh', 'atanh', 'acoth',
'asech', 'acsch', 'erfc', 'erfi', 'erf', 'erfinv', 'erfcinv', 'besseli',
'besselj', 'besselk', 'bessely', 'bernoulli', 'beta', 'euler', 'exp',
'factorial', 'floor', 'fresnelc', 'fresnels', 'gamma', 'harmonic', 'log',
'polylog', 'sign', 'zeta', 'legendre']
```

Built-in mutable sequence.



If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

```
sympy.printing.octave.known_fcns_src2 = {'Abs': 'abs', 'Chi': 'coshint', 'Ci': 'cosint', 'DiracDelta': 'dirac', 'Heaviside': 'heaviside', 'LambertW': 'lambertw', 'Max': 'max', 'Min': 'min', 'Mod': 'mod', 'RisingFactorial': 'pochhammer', 'Shi': 'sinhint', 'Si': 'sinint', 'arg': 'angle', 'binomial': 'bincoeff', 'ceiling': 'ceil', 'chebyshevt': 'chebyshevT', 'chebyshevu': 'chebyshevU', 'conjugate': 'conj', 'im': 'imag', 'laguerre': 'laguerreL', 'li': 'logint', 'loggamma': 'gammaln', 'polygamma': 'psi', 're': 'real'}
```

```
class sympy.printing.octave.OctaveCodePrinter(settings={})
```

A printer to convert expressions to strings of Octave/Matlab code.

**printmethod:** str = '\_octave'

**indent\_code**(code)

Accepts a string of code or a list of code lines

```
sympy.printing.octave.octave_code(expr, assign_to=None, **settings)
```

Converts *expr* to a string of Octave (or Matlab) code.

The string uses a subset of the Octave language for Matlab compatibility.

#### Parameters

**expr** : Expr

A SymPy expression to be converted.

**assign\_to** : optional

When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, Symbol, MatrixSymbol, or Indexed type. This can be helpful for expressions that generate multi-line statements.

**precision** : integer, optional

The precision for numbers such as pi [default=16].

**user\_functions** : dict, optional

A dictionary where keys are FunctionClass instances and values are their string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument\_test, cfunction\_string)]. See below for examples.

**human** : bool, optional

If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the same information is returned in a tuple of (symbols\_to\_declare, not\_supported\_functions, code\_text). [default=True].

**contract**: bool, optional

If True, Indexed instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting contract=False will not generate loops, instead the user is responsible to provide values for the indices in the code. [default=True].

**inline: bool, optional**

If True, we try to create single-statement code instead of multiple statements. [default=True].

**Examples**

```
>>> from sympy import octave_code, symbols, sin, pi
>>> x = symbols('x')
>>> octave_code(sin(x).series(x).removeO())
'x.^5/120 - x.^3/6 + x'
```

```
>>> from sympy import Rational, ceiling
>>> x, y, tau = symbols("x, y, tau")
>>> octave_code((2*tau)**Rational(7, 2))
'8*sqrt(2)*tau.^(7/2)'
```

Note that element-wise (Hadamard) operations are used by default between symbols. This is because its very common in Octave to write “vectorized” code. It is harmless if the values are scalars.

```
>>> octave_code(sin(pi*x*y), assign_to="s")
's = sin(pi*x.y);'
```

If you need a matrix product “\*” or matrix power “^”, you can specify the symbol as a MatrixSymbol.

```
>>> from sympy import Symbol, MatrixSymbol
>>> n = Symbol('n', integer=True, positive=True)
>>> A = MatrixSymbol('A', n, n)
>>> octave_code(3*pi*A**3)
'(3*pi)*A^3'
```

This class uses several rules to decide which symbol to use a product. Pure numbers use “\*”, Symbols use “.” and MatrixSymbols use “\*”. A HadamardProduct can be used to specify componentwise multiplication “.” of two MatrixSymbols. There is currently there is no easy way to specify scalar symbols, so sometimes the code might have some minor cosmetic issues. For example, suppose x and y are scalars and A is a Matrix, then while a human programmer might write “(x^2\*y)\*A^3”, we generate:

```
>>> octave_code(x**2*y*A**3)
'(x.^2.*y)*A^3'
```

Matrices are supported using Octave inline notation. When using assign\_to with matrices, the name can be specified either as a string or as a MatrixSymbol. The dimensions must align in the latter case.

```
>>> from sympy import Matrix, MatrixSymbol
>>> mat = Matrix([[x**2, sin(x), ceiling(x)]])
>>> octave_code(mat, assign_to='A')
'A = [x.^2 sin(x) ceil(x)];'
```

Piecewise expressions are implemented with logical masking by default. Alternatively, you can pass “inline=False” to use if-else conditionals. Note that if the Piecewise lacks

a default term, represented by `(expr, True)` then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.

```
>>> from sympy import Piecewise
>>> pw = Piecewise((x + 1, x > 0), (x, True))
>>> octave_code(pw, assign_to=tau)
'tau = ((x > 0).*(x + 1) + (~(x > 0)).*(x));'
```

Note that any expression that can be generated normally can also exist inside a Matrix:

```
>>> mat = Matrix([[x**2, pw, sin(x)]])
>>> octave_code(mat, assign_to='A')
'A = [x.^2 ((x > 0).*(x + 1) + (~(x > 0)).*(x)) sin(x)];'
```

Custom printing can be defined for certain types by passing a dictionary of “type” : “function” to the `user_functions` kwarg. Alternatively, the dictionary value can be a list of tuples i.e., `[(argument_test, cfunction_string)]`. This can be used to call a custom Octave function.

```
>>> from sympy import Function
>>> f = Function('f')
>>> g = Function('g')
>>> custom_functions = {
... "f": "existing_octave_fcn",
... "g": [(lambda x: x.is_Matrix, "my_mat_fcn"),
... (lambda x: not x.is_Matrix, "my_fcn")]
... }
>>> mat = Matrix([[1, x]])
>>> octave_code(f(x) + g(x) + g(mat), user_functions=custom_functions)
'existing_octave_fcn(x) + my_fcn(x) + my_mat_fcn([1 x])'
```

Support for loops is provided through Indexed types. With `contract=True` these expressions will be turned into loops, whereas `contract=False` will just print the assignment expression that should be looped over:

```
>>> from sympy import Eq, IndexedBase, Idx
>>> len_y = 5
>>> y = IndexedBase('y', shape=(len_y,))
>>> t = IndexedBase('t', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y-1,))
>>> i = Idx('i', len_y-1)
>>> e = Eq(Dy[i], (y[i+1]-y[i])/(t[i+1]-t[i]))
>>> octave_code(e.rhs, assign_to=e.lhs, contract=False)
'Dy(i) = (y(i + 1) - y(i))./(t(i + 1) - t(i));'
```

## Rust code printing

```
sympy.printing.rust.known_functions = {'Abs': 'abs', 'Max': 'max', 'Min':
'min', 'Pow': [(lambda: '<function <lambda>>', 'recip', 2), (lambda: '<function <lambda>>',
'sqrt', 2), (lambda: '<function <lambda>>', 'sqrt().recip', 2), (lambda: '<function <lambda>>',
'cbrt', 2), (lambda: '<function <lambda>>', 'exp2', 3), (lambda: '<function <lambda>>', 'powi',
1), (lambda: '<function <lambda>>', 'powf', 1)], 'acos': 'acos', 'acosh': 'acosh',
'asin': 'asin', 'asinh': 'asinh', 'atan': 'atan', 'atan2': 'atan2', 'atanh':
'atanh', 'ceiling': 'ceil', 'cos': 'cos', 'cosh': 'cosh', 'exp': [(lambda: '<function
<lambda>>', 'exp', 2)], 'floor': 'floor', 'log': 'ln', 'sign': 'signum', 'sin':
'sin', 'sinh': 'sinh', 'sqrt': 'sqrt', 'tan': 'tan', 'tanh': 'tanh'}
```

```
class sympy.printing.rust.RustCodePrinter(settings={})
```

A printer to convert SymPy expressions to strings of Rust code

**printmethod:** str = '\_rust\_code'

**indent\_code**(code)

Accepts a string of code or a list of code lines

```
sympy.printing.rust.rust_code(expr, assign_to=None, **settings)
```

Converts an expr to a string of Rust code

### Parameters

**expr** : Expr

A SymPy expression to be converted.

**assign\_to** : optional

When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, Symbol, MatrixSymbol, or Indexed type. This is helpful in case of line-wrapping, or for expressions that generate multi-line statements.

**precision** : integer, optional

The precision for numbers such as pi [default=15].

**user\_functions** : dict, optional

A dictionary where the keys are string representations of either FunctionClass or UndefinedFunction instances and the values are their desired C string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument\_test, cfunction\_string)]. See below for examples.

**dereference** : iterable, optional

An iterable of symbols that should be dereferenced in the printed code expression. These would be values passed by address to the function. For example, if dereference=[a], the resulting code would print (\*a) instead of a.

**human** : bool, optional

If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the same information is returned in a tuple of (symbols\_to\_declare, not\_supported\_functions, code\_text). [default=True].

**contract: bool, optional**

If True, Indexed instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting `contract=False` will not generate loops, instead the user is responsible to provide values for the indices in the code. [default=True].

**Examples**

```
>>> from sympy import rust_code, symbols, Rational, sin, ceiling, Abs, _
_Function
>>> x, tau = symbols("x, tau")
>>> rust_code((2*tau)**Rational(7, 2))
'8*1.4142135623731*tau.powf(7_f64/2.0)'
>>> rust_code(sin(x), assign_to="s")
's = x.sin();'
```

Simple custom printing can be defined for certain types by passing a dictionary of {"type": "function"} to the `user_functions` kwarg. Alternatively, the dictionary value can be a list of tuples i.e. [(argument\_test, cfunction\_string)].

```
>>> custom_functions = {
... "ceiling": "CEIL",
... "Abs": [(lambda x: not x.is_integer, "fabs", 4),
... (lambda x: x.is_integer, "ABS", 4)],
... "func": "f"
... }
>>> func = Function('func')
>>> rust_code(func(Abs(x) + ceiling(x)), user_functions=custom_functions)
'(fabs(x) + x.CEIL()).f()'
```

Piecewise expressions are converted into conditionals. If an `assign_to` variable is provided an if statement is created, otherwise the ternary operator is used. Note that if the Piecewise lacks a default term, represented by (expr, True) then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.

```
>>> from sympy import Piecewise
>>> expr = Piecewise((x + 1, x > 0), (x, True))
>>> print(rust_code(expr, tau))
tau = if (x > 0) {
 x + 1
} else {
 x
};
```

Support for loops is provided through Indexed types. With `contract=True` these expressions will be turned into loops, whereas `contract=False` will just print the assignment expression that should be looped over:

```
>>> from sympy import Eq, IndexedBase, Idx
>>> len_y = 5
>>> y = IndexedBase('y', shape=(len_y,))
```

(continues on next page)

(continued from previous page)

```
>>> t = IndexedBase('t', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y-1,))
>>> i = Idx('i', len_y-1)
>>> e=Eq(Dy[i], (y[i+1]-y[i])/(t[i+1]-t[i]))
>>> rust_code(e.rhs, assign_to=e.lhs, contract=False)
'Dy[i] = (y[i + 1] - y[i])/(t[i + 1] - t[i]);'
```

Matrices are also supported, but a `MatrixSymbol` of the same dimensions must be provided to `assign_to`. Note that any expression that can be generated normally can also exist inside a `Matrix`:

```
>>> from sympy import Matrix, MatrixSymbol
>>> mat = Matrix([x**2, Piecewise((x + 1, x > 0), (x, True)), sin(x)])
>>> A = MatrixSymbol('A', 3, 1)
>>> print(rust_code(mat, A))
A = [x.powi(2), if (x > 0) {
 x + 1
 } else {
 x
 }, x.sin()];
```

## Aesara Code printing

**class** `sympy.printing.aesaracode.AesaraPrinter(*args, **kwargs)`

Code printer which creates Aesara symbolic expression graphs.

### Parameters

**cache** : dict

Cache dictionary to use. If `None` (default) will use the global cache. To create a printer which does not depend on or alter global state pass an empty dictionary. Note: the dictionary is not copied on initialization of the printer and will be updated in-place, so using the same dict object when creating multiple printers or making multiple calls to [aesara\\_code\(\)](#) (page 2167) or [aesara\\_function\(\)](#) (page 2167) means the cache is shared between all these applications.

### Attributes

|              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|--------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>cache</b> | (dict) A cache of Aesara variables which have been created for SymPy symbol-like objects (e.g. <a href="#">sympy.core.symbol.Symbol</a> (page 976) or <a href="#">sympy.matrices.expressions.MatrixSymbol</a> (page 1372)). This is used to ensure that all references to a given symbol in an expression (or multiple expressions) are printed as the same Aesara variable, which is created only once. Symbols are differentiated only by name and type. The format of the cache's contents should be considered opaque to the user. |
|--------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

**printmethod:** `str = '_aesara'`

**doprint**(*expr*, *dtypes*=None, *broadcastables*=None)

Convert a SymPy expression to a Aesara graph variable.

The *dtypes* and *broadcastables* arguments are used to specify the data type, dimension, and broadcasting behavior of the Aesara variables corresponding to the free symbols in *expr*. Each is a mapping from SymPy symbols to the value of the corresponding argument to `aesara.tensor.var.TensorVariable`.

See the corresponding [documentation page](#) for more information on broadcasting in Aesara.

#### Parameters

**expr** : `sympy.core.expr.Expr`

SymPy expression to print.

**dtypes** : dict

Mapping from SymPy symbols to Aesara datatypes to use when creating new Aesara variables for those symbols. Corresponds to the *dtype* argument to `aesara.tensor.var.TensorVariable`. Defaults to 'floatX' for symbols not included in the mapping.

**broadcastables** : dict

Mapping from SymPy symbols to the value of the *broadcastable* argument to `aesara.tensor.var.TensorVariable` to use when creating Aesara variables for those symbols. Defaults to the empty tuple for symbols not included in the mapping (resulting in a scalar).

#### Returns

`aesara.graph.basic.Variable`

A variable corresponding to the expression's value in a Aesara symbolic expression graph.

`sympy.printing.aesaracode.aesara_code`(*expr*, *cache*=None, *\*\*kwargs*)

Convert a SymPy expression into a Aesara graph variable.

#### Parameters

**expr** : `sympy.core.expr.Expr`

SymPy expression object to convert.

**cache** : dict

Cached Aesara variables (see [AesaraPrinter.cache](#) (page 2166)). Defaults to the module-level global cache.

**dtypes** : dict

Passed to [AesaraPrinter.doprint\(\)](#) (page 2166).

**broadcastables** : dict

Passed to [AesaraPrinter.doprint\(\)](#) (page 2166).

#### Returns

`aesara.graph.basic.Variable`

A variable corresponding to the expression's value in a Aesara symbolic expression graph.

```
sympy.printing.aesaracode.aesara_function(inputs, outputs, scalar=False, *,
 dim=None, dims=None,
 broadcastables=None, **kwargs)
```

Create a Aesara function from SymPy expressions.

The inputs and outputs are converted to Aesara variables using [aesara\\_code\(\)](#) (page 2167) and then passed to `aesara.function`.

### Parameters

#### **inputs**

Sequence of symbols which constitute the inputs of the function.

#### **outputs**

Sequence of expressions which constitute the outputs(s) of the function. The free symbols of each expression must be a subset of inputs.

#### **scalar** : bool

Convert 0-dimensional arrays in output to scalars. This will return a Python wrapper function around the Aesara function object.

#### **cache** : dict

Cached Aesara variables (see [AesaraPrinter.cache](#) (page 2166)). Defaults to the module-level global cache.

#### **dtypes** : dict

Passed to [AesaraPrinter.doprint\(\)](#) (page 2166).

#### **broadcastables** : dict

Passed to [AesaraPrinter.doprint\(\)](#) (page 2166).

#### **dims** : dict

Alternative to `broadcastables` argument. Mapping from elements of inputs to integers indicating the dimension of their associated arrays/tensors. Overrides `broadcastables` argument if given.

#### **dim** : int

Another alternative to the `broadcastables` argument. Common number of dimensions to use for all arrays/tensors. `aesara_function([x, y], [...], dim=2)` is equivalent to using `broadcastables={x: (False, False), y: (False, False)}`.

### Returns

callable

A callable object which takes values of inputs as positional arguments and returns an output array for each of the expressions in outputs. If outputs is a single expression the function will return a Numpy array, if it is a list of multiple expressions the function will return a list of arrays. See description of the `squeeze` argument above for the behavior when a single output is passed in a list. The returned object will either be an instance of `aesara.compile.function.types.Function` or a Python wrapper function around one. In both cases, the returned value will have a `aesara_function` attribute which points to the return value of `aesara.function`.



## Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.printing.aesaracode import aesara_function
```

A simple function with one input and one output:

```
>>> f1 = aesara_function([x], [x**2 - 1], scalar=True)
>>> f1(3)
8.0
```

A function with multiple inputs and one output:

```
>>> f2 = aesara_function([x, y, z], [(x**z + y**z)**(1/z)], scalar=True)
>>> f2(3, 4, 2)
5.0
```

A function with multiple inputs and multiple outputs:

```
>>> f3 = aesara_function([x, y], [x**2 + y**2, x**2 - y**2], scalar=True)
>>> f3(2, 3)
[13.0, -5.0]
```

### See also:

[dim\\_handling](#) (page 2169)

`sympy.printing.aesaracode.dim_handling(inputs, dim=None, dims=None, broadcastables=None)`

Get value of `broadcastables` argument to [aesara\\_code\(\)](#) (page 2167) from keyword arguments to [aesara\\_function\(\)](#) (page 2167).

Included for backwards compatibility.

### Parameters

#### inputs

Sequence of input symbols.

#### dim : int

Common number of dimensions for all inputs. Overrides other arguments if given.

#### dims : dict

Mapping from input symbols to number of dimensions. Overrides `broadcastables` argument if given.

#### broadcastables : dict

Explicit value of `broadcastables` argument to [AesaraPrinter.doprint\(\)](#) (page 2166). If not None function will return this value unchanged.

### Returns

dict

Dictionary mapping elements of inputs to their “broadcastable” values (tuple of bools).

## Gtk

You can print to a `gtkmathview` widget using the function `print_gtk` located in `sympy.printing.gtk` (it requires to have installed `gtkmathview` and `libgtkmathview-bin` in some systems).

`GtkMathView` accepts MathML, so this rendering depends on the MathML representation of the expression.

Usage:

```
from sympy import *
print_gtk(x**2 + 2*exp(x**3))
```

`sympy.printing.gtk.print_gtk(x, start_viewer=True)`

Print to `Gtkmathview`, a gtk widget capable of rendering MathML.

Needs `libgtkmathview-bin`

## LambdaPrinter

This classes implements printing to strings that can be used by the `sympy.utilities.lambdify.lambdify()` (page 2100) function.

**class** `sympy.printing.lambdarepr.LambdaPrinter(settings=None)`

This printer converts expressions into strings that can be used by `lambdify`.

**printmethod:** `str = '_lambdacode'`

`sympy.printing.lambdarepr.lambdarepr(expr, **settings)`

Returns a string usable for `lambdifying`.

## LatexPrinter

This class implements LaTeX printing. See `sympy.printing.latex`.

`sympy.printing.latex.accepted_latex_functions = ['arcsin', 'arccos', 'arctan', 'sin', 'cos', 'tan', 'sinh', 'cosh', 'tanh', 'sqrt', 'ln', 'log', 'sec', 'csc', 'cot', 'coth', 're', 'im', 'frac', 'root', 'arg']`

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

**class** `sympy.printing.latex.LatexPrinter(settings=None)`

**printmethod:** `str = '_latex'`

**parenthesize\_super(s)**

Protect superscripts in `s`

If the `parenthesize_super` option is set, protect with parentheses, else wrap in braces.

```
sympy.printing.latex.latex(expr, *, full_prec=False, fold_frac_powers=False,
 fold_func_brackets=False, fold_short_frac=None,
 inv_trig_style='abbreviated', itex=False, ln_notation=False,
 long_frac_ratio=None, mat_delim='[', mat_str=None,
 mode='plain', mul_symbol=None, order=None,
 symbol_names={}, root_notation=True,
 mat_symbol_style='plain', imaginary_unit='i',
 gothic_re_im=False, decimal_separator='period',
 perm_cyclic=True, parenthesize_super=True, min=None,
 max=None, diff_operator='d')
```

Convert the given expression to LaTeX string representation.

#### Parameters

**full\_prec**: boolean, optional

If set to True, a floating point number is printed with full precision.

**fold\_frac\_powers**: boolean, optional

Emit  $\{p/q\}$  instead of  $\{\frac{p}{q}\}$  for fractional powers.

**fold\_func\_brackets**: boolean, optional

Fold function brackets where applicable.

**fold\_short\_frac**: boolean, optional

Emit  $p / q$  instead of  $\frac{p}{q}$  when the denominator is simple enough (at most two terms and no powers). The default value is True for inline mode, False otherwise.

**inv\_trig\_style**: string, optional

How inverse trig functions should be displayed. Can be one of 'abbreviated', 'full', or 'power'. Defaults to 'abbreviated'.

**itex**: boolean, optional

Specifies if itex-specific syntax is used, including emitting  $\$ \dots \$$ .

**ln\_notation**: boolean, optional

If set to True,  $\ln$  is used instead of default  $\log$ .

**long\_frac\_ratio**: float or None, optional

The allowed ratio of the width of the numerator to the width of the denominator before the printer breaks off long fractions. If None (the default value), long fractions are not broken up.

**mat\_delim**: string, optional

The delimiter to wrap around matrices. Can be one of '[', '(', or the empty string ''. Defaults to '['.

**mat\_str**: string, optional

Which matrix environment string to emit. 'smallmatrix', 'matrix', 'array', etc. Defaults to 'smallmatrix' for inline mode, 'matrix' for matrices of no more than 10 columns, and 'array' otherwise.

**mode**: string, optional

Specifies how the generated code will be delimited. mode can be one of 'plain', 'inline', 'equation' or 'equation\*'. If mode is set to 'plain', then the resulting code will not be delimited at all (this is the default). If mode is set to 'inline' then inline LaTeX  $\dots$  will be used. If mode is set to 'equation' or 'equation\*', the resulting code will be enclosed in the equation or equation\* environment (remember to import amsmath for equation\*), unless the itex option is set. In the latter case, the  $\dots$  syntax is used.

**mul\_symbol** : string or None, optional

The symbol to use for multiplication. Can be one of None, 'ldot', 'dot', or 'times'.

**order**: string, optional

Any of the supported monomial orderings (currently 'lex', 'grlex', or 'grevlex'), 'old', and 'none'. This parameter does nothing for *.Mul* objects. Setting order to 'old' uses the compatibility ordering for *.Add* defined in Printer. For very large expressions, set the order keyword to 'none' if speed is a concern.

**symbol\_names** : dictionary of strings mapped to symbols, optional

Dictionary of symbols and the custom strings they should be emitted as.

**root\_notation** : boolean, optional

If set to False, exponents of the form  $1/n$  are printed in fractional form. Default is True, to print exponent in root form.

**mat\_symbol\_style** : string, optional

Can be either 'plain' (default) or 'bold'. If set to 'bold', a *.MatrixSymbol* A will be printed as  $\mathbf{A}$ , otherwise as A.

**imaginary\_unit** : string, optional

String to use for the imaginary unit. Defined options are 'i' (default) and 'j'. Adding r or t in front gives  $\mathrm{r}$  or  $\mathrm{t}$ , so 'ri' leads to  $\mathrm{i}$  which gives i.

**gothic\_re\_im** : boolean, optional

If set to True,  $\Re$  and  $\Im$  is used for re and im, respectively. The default is False leading to re and im.

**decimal\_separator** : string, optional

Specifies what separator to use to separate the whole and fractional parts of a floating point number as in 2.5 for the default, period or 2,5 when comma is specified. Lists, sets, and tuple are printed with semi-colon separating the elements when comma is chosen. For example, [1; 2; 3] when comma is chosen and [1,2,3] for when period is chosen.

**parenthesize\_super** : boolean, optional

If set to False, superscripted expressions will not be parenthesized when powered. Default is True, which parenthesizes the expression when powered.

**min**: Integer or None, optional

Sets the lower bound for the exponent to print floating point numbers in fixed-point format.

**max: Integer or None, optional**

Sets the upper bound for the exponent to print floating point numbers in fixed-point format.

**diff\_operator: string, optional**

String to use for differential operator. Default is 'd', to print in italic form. 'rd', 'td' are shortcuts for  $\mathrm{d}$  and  $\text{d}$ .

## Notes

Not using a print statement for printing, results in double backslashes for latex commands since that's the way Python escapes backslashes in strings.

```
>>> from sympy import latex, Rational
>>> from sympy.abc import tau
>>> latex((2*tau)**Rational(7,2))
'8 \sqrt{2} \tau^{\frac{7}{2}}'
>>> print(latex((2*tau)**Rational(7,2)))
8 \sqrt{2} \tau^{\frac{7}{2}}
```

## Examples

```
>>> from sympy import latex, pi, sin, asin, Integral, Matrix, Rational, log
>>> from sympy.abc import x, y, mu, r, tau
```

Basic usage:

```
>>> print(latex((2*tau)**Rational(7,2)))
8 \sqrt{2} \tau^{\frac{7}{2}}
```

mode and itex options:

```
>>> print(latex((2*mu)**Rational(7,2), mode='plain'))
8 \sqrt{2} \mu^{\frac{7}{2}}
>>> print(latex((2*tau)**Rational(7,2), mode='inline'))
$8 \sqrt{2} \tau^{7 / 2}$
>>> print(latex((2*mu)**Rational(7,2), mode='equation*'))
\begin{equation*}8 \sqrt{2} \mu^{\frac{7}{2}}\end{equation*}
>>> print(latex((2*mu)**Rational(7,2), mode='equation'))
\begin{equation}8 \sqrt{2} \mu^{\frac{7}{2}}\end{equation}
>>> print(latex((2*mu)**Rational(7,2), mode='equation', itex=True))
$$8 \sqrt{2} \mu^{\frac{7}{2}}$$
>>> print(latex((2*mu)**Rational(7,2), mode='plain'))
8 \sqrt{2} \mu^{\frac{7}{2}}
>>> print(latex((2*tau)**Rational(7,2), mode='inline'))
$8 \sqrt{2} \tau^{7 / 2}$
>>> print(latex((2*mu)**Rational(7,2), mode='equation*'))
```

(continues on next page)

(continued from previous page)

```

\begin{equation*}8 \sqrt{2} \mu^{\frac{7}{2}}\end{equation*}
>>> print(latex((2*mu)**Rational(7,2), mode='equation'))
\begin{equation}8 \sqrt{2} \mu^{\frac{7}{2}}\end{equation}
>>> print(latex((2*mu)**Rational(7,2), mode='equation', itex=True))

$$8 \sqrt{2} \mu^{\frac{7}{2}}$$


```

Fraction options:

```

>>> print(latex((2*tau)**Rational(7,2), fold_frac_powers=True))
8 \sqrt{2} \tau^{\frac{7}{2}}
>>> print(latex((2*tau)**sin(Rational(7,2))))
\left(2 \tau\right)^{\sin\left(\frac{7}{2}\right)}
>>> print(latex((2*tau)**sin(Rational(7,2)), fold_func_brackets=True))
\left(2 \tau\right)^{\sin \frac{7}{2}}
>>> print(latex(3*x**2/y))
\frac{3 x^2}{y}
>>> print(latex(3*x**2/y, fold_short_frac=True))
3 x^2 / y
>>> print(latex(Integral(r, r)/2/pi, long_frac_ratio=2))
\frac{\int r, dr}{2 \pi}
>>> print(latex(Integral(r, r)/2/pi, long_frac_ratio=0))
\frac{1}{2 \pi} \int r, dr

```

Multiplication options:

```

>>> print(latex((2*tau)**sin(Rational(7,2)), mul_symbol="times"))
\left(2 \times \tau\right)^{\sin\left(\frac{7}{2}\right)}

```

Trig options:

```

>>> print(latex(asin(Rational(7,2))))
\operatorname{asin}\left(\frac{7}{2}\right)
>>> print(latex(asin(Rational(7,2)), inv_trig_style="full"))
\arcsin\left(\frac{7}{2}\right)
>>> print(latex(asin(Rational(7,2)), inv_trig_style="power"))
\sin^{-1}\left(\frac{7}{2}\right)

```

Matrix options:

```

>>> print(latex(Matrix(2, 1, [x, y])))
\left[\begin{matrix}x\\y\end{matrix}\right]
>>> print(latex(Matrix(2, 1, [x, y]), mat_str = "array"))
\left[\begin{array}{c}x\\y\end{array}\right]
>>> print(latex(Matrix(2, 1, [x, y]), mat_delim="("))
\left(\begin{matrix}x\\y\end{matrix}\right)

```

Custom printing of symbols:

```

>>> print(latex(x**2, symbol_names={x: 'x_i'}))
x_i^2

```

Logarithms:

```
>>> print(latex(log(10)))
\log{\left(10 \right)}
>>> print(latex(log(10), ln_notation=True))
\ln{\left(10 \right)}
```

latex() also supports the builtin container types list, tuple, and dict:

```
>>> print(latex([2/x, y], mode='inline'))
$\left[2 / x, \ y \right]$
```

Unsupported types are rendered as monospaced plaintext:

```
>>> print(latex(int))
\mathtt{\text{<class 'int'>}}
>>> print(latex("plain % text"))
\mathtt{\text{plain \% text}}
```

See [Example of Custom Printing Method](#) (page 2137) for an example of how to override this behavior for your own types by implementing `_latex`.

Changed in version 1.7.0: Unsupported types no longer have their `str` representation treated as valid latex.

`sympy.printing.latex.print_latex(expr, **settings)`

Prints LaTeX representation of the given expression. Takes the same settings as `latex()`.

## MathMLPrinter

This class is responsible for MathML printing. See `sympy.printing.mathml`.

More info on mathml : <http://www.w3.org/TR/MathML2>

**class** `sympy.printing.mathml.MathMLPrinterBase(settings=None)`

Contains common code required for MathMLContentPrinter and MathMLPresentationPrinter.

**doprint**(*expr*)

Prints the expression as MathML.

**class** `sympy.printing.mathml.MathMLContentPrinter(settings=None)`

Prints an expression to the Content MathML markup language.

References: <https://www.w3.org/TR/MathML2/chapter4.html>

**printmethod:** `str = '_mathml_content'`

**mathml\_tag**(*e*)

Returns the MathML tag for an expression.

**class** `sympy.printing.mathml.MathMLPresentationPrinter(settings=None)`

Prints an expression to the Presentation MathML markup language.

References: <https://www.w3.org/TR/MathML2/chapter3.html>

**printmethod:** `str = '_mathml_presentation'`

**mathml\_tag(*e*)**

Returns the MathML tag for an expression.

`sympy.printing.mathml.mathml(expr, printer='content', *, order=None, encoding='utf-8',  
fold_frac_powers=False, fold_func_brackets=False,  
fold_short_frac=None, inv_trig_style='abbreviated',  
ln_notation=False, long_frac_ratio=None, mat_delim='[',  
mat_symbol_style='plain', mul_symbol=None,  
root_notation=True, symbol_names={},  
mul_symbol_mathml_numbers='&#xB7;')`

Returns the MathML representation of *expr*. If *printer* is *presentation* then prints Presentation MathML else prints content MathML.

`sympy.printing.mathml.print_mathml(expr, printer='content', **settings)`

Prints a pretty representation of the MathML code for *expr*. If *printer* is *presentation* then prints Presentation MathML else prints content MathML.

## Examples

```
>>> ##
>>> from sympy import print_mathml
>>> from sympy.abc import x
>>> print_mathml(x+1)
<apply>
 <plus/>
 <ci>x</ci>
 <cn>1</cn>
</apply>
>>> print_mathml(x+1, printer='presentation')
<mrow>
 <mi>x</mi>
 <mo>+</mo>
 <mn>1</mn>
</mrow>
```

## PythonCodePrinter

Python code printers

This module contains Python code printers for plain Python as well as NumPy & SciPy enabled code.

**class** `sympy.printing.pycode.MpmathPrinter(settings=None)`

Lambda printer for mpmath which maintains precision for floats

`sympy.printing.pycode.pycode(expr, **settings)`

Converts an *expr* to a string of Python code

### Parameters

**expr** : Expr

A SymPy expression.

**fully\_qualified\_modules** : bool



Whether or not to write out full module names of functions (`math.sin` vs. `sin`). default: `True`.

**standard** : str or None, optional

Only 'python3' (default) is supported. This parameter may be removed in the future.

## Examples

```
>>> from sympy import pycode, tan, Symbol
>>> pycode(tan(Symbol('x')) + 1)
'math.tan(x) + 1'
```

## PythonPrinter

This class implements Python printing. Usage:

```
>>> from sympy import print_python, sin
>>> from sympy.abc import x

>>> print_python(5*x**3 + sin(x))
x = Symbol('x')
e = 5*x**3 + sin(x)
```

## srepr

This printer generates executable code. This code satisfies the identity `eval(srepr(expr)) == expr`.

`srepr()` gives more low level textual output than `repr()`

Example:

```
>>> repr(5*x**3 + sin(x))
'5*x**3 + sin(x)'
```

```
>>> srepr(5*x**3 + sin(x))
"Add(Mul(Integer(5), Pow(Symbol('x'), Integer(3))), sin(Symbol('x')))"
```

`srepr()` gives the `repr` form, which is what `repr()` would normally give but for SymPy we don't actually use `srepr()` for `__repr__` because it's so verbose, it is unlikely that anyone would want it called by default. Another reason is that lists call `repr` on their elements, like `print([a, b, c])` calls `repr(a)`, `repr(b)`, `repr(c)`. So if we used `srepr` for `__repr__` any list with SymPy objects would include the `srepr` form, even if we used `str()` or `print()`.

**class** `sympy.printing.repr.ReprPrinter(settings=None)`

**printmethod:** `str = '_sympyrepr'`

**emptyPrinter**(*expr*)

The fallback printer.

**reprify**(*args*, *sep*)

Prints each item in *args* and joins them with *sep*.

`sympy.printing.repr.srepr(expr, *, order=None, perm_cyclic=True)`

return *expr* in repr form

## StrPrinter

This module generates readable representations of SymPy expressions.

**class** `sympy.printing.str.StrPrinter(settings=None)`

**printmethod:** `str = '_sympystr'`

`sympy.printing.str.sstr(expr, *, order=None, full_prec='auto', sympy_integers=False, abbrev=False, perm_cyclic=True, min=None, max=None)`

Returns the expression as a string.

For large expressions where speed is a concern, use the setting `order='none'`. If `abbrev=True` setting is used then units are printed in abbreviated form.

## Examples

```
>>> from sympy import symbols, Eq, sstr
>>> a, b = symbols('a b')
>>> sstr(Eq(a + b, 0))
'Eq(a + b, 0)'
```

`sympy.printing.str.sstrrepr(expr, *, order=None, full_prec='auto', sympy_integers=False, abbrev=False, perm_cyclic=True, min=None, max=None)`

return *expr* in mixed str/repr form

i.e. strings are returned in repr form with quotes, and everything else is returned in str form.

This function could be useful for hooking into `sys.displayhook`

## Tree Printing

The functions in this module create a representation of an expression as a tree.

`sympy.printing.tree.pprint_nodes(subtrees)`

Prettyprints systems of nodes.

## Examples

```
>>> from sympy.printing.tree import pprint_nodes
>>> print(pprint_nodes(["a", "b1\nb2", "c"]))
+-a
+-b1
| b2
+-c
```

`sympy.printing.tree.print_node(node, assumptions=True)`

Returns information about the “node”.

This includes class name, string representation and assumptions.

### Parameters

**assumptions** : bool, optional

See the assumptions keyword in `tree`

`sympy.printing.tree.tree(node, assumptions=True)`

Returns a tree representation of “node” as a string.

It uses `print_node()` together with `pprint_nodes()` on `node.args` recursively.

### Parameters

**assumptions** : bool, optional

The flag to decide whether to print out all the assumption data (such as `is_integer``, `is_real`) associated with the expression or not.

Enabling the flag makes the result verbose, and the printed result may not be deterministic because of the randomness used in back-tracing the assumptions.

### See also:

[`print\_tree`](#) (page 2179)

`sympy.printing.tree.print_tree(node, assumptions=True)`

Prints a tree representation of “node”.

### Parameters

**assumptions** : bool, optional

The flag to decide whether to print out all the assumption data (such as `is_integer``, `is_real`) associated with the expression or not.

Enabling the flag makes the result verbose, and the printed result may not be deterministic because of the randomness used in back-tracing the assumptions.

## Examples

```
>>> from sympy.printing import print_tree
>>> from sympy import Symbol
>>> x = Symbol('x', odd=True)
>>> y = Symbol('y', even=True)
```

Printing with full assumptions information:

```
>>> print_tree(y**x)
Pow: y**x
+-Symbol: y
| algebraic: True
| commutative: True
| complex: True
| even: True
| extended_real: True
| finite: True
| hermitian: True
| imaginary: False
| infinite: False
| integer: True
| irrational: False
| noninteger: False
| odd: False
| rational: True
| real: True
| transcendental: False
+-Symbol: x
| algebraic: True
| commutative: True
| complex: True
| even: False
| extended_nonzero: True
| extended_real: True
| finite: True
| hermitian: True
| imaginary: False
| infinite: False
| integer: True
| irrational: False
| noninteger: False
| nonzero: True
| odd: True
| rational: True
| real: True
| transcendental: False
| zero: False
```

Hiding the assumptions:

```
>>> print_tree(y**x, assumptions=False)
Pow: y**x
```

(continues on next page)

(continued from previous page)

```
+-Symbol: y
+-Symbol: x
```

**See also:**

[tree](#) (page 2179)

**Preview**

A useful function is `preview`:

```
sympy.printing.preview.preview(expr, output='png', viewer=None, euler=True,
 packages=(), filename=None, outputbuffer=None,
 preamble=None, dvioptions=None,
 outputTexFile=None, extra_preamble=None,
 fontsize=None, **latex_settings)
```

View expression or LaTeX markup in PNG, DVI, PostScript or PDF form.

If the `expr` argument is an expression, it will be exported to LaTeX and then compiled using the available TeX distribution. The first argument, `'expr'`, may also be a LaTeX string. The function will then run the appropriate viewer for the given output format or use the user defined one. By default png output is generated.

By default pretty Euler fonts are used for typesetting (they were used to typeset the well known “Concrete Mathematics” book). For that to work, you need the `'eulervm.sty'` LaTeX style (in Debian/Ubuntu, install the `texlive-fonts-extra` package). If you prefer default AMS fonts or your system lacks `'eulervm'` LaTeX package then unset the `'euler'` keyword argument.

To use viewer auto-detection, lets say for `'png'` output, issue

```
>>> from sympy import symbols, preview, Symbol
>>> x, y = symbols("x,y")
```

```
>>> preview(x + y, output='png')
```

This will choose `'pyglet'` by default. To select a different one, do

```
>>> preview(x + y, output='png', viewer='gimp')
```

The `'png'` format is considered special. For all other formats the rules are slightly different. As an example we will take `'dvi'` output format. If you would run

```
>>> preview(x + y, output='dvi')
```

then `'view'` will look for available `'dvi'` viewers on your system (predefined in the function, so it will try `evince`, first, then `kdvi` and `xdvi`). If nothing is found, it will fall back to using a system file association (via `open` and `xdg-open`). To always use your system file association without searching for the above readers, use

```
>>> from sympy.printing.preview import system_default_viewer
>>> preview(x + y, output='dvi', viewer=system_default_viewer)
```

If this still does not find the viewer you want, it can be set explicitly.

```
>>> preview(x + y, output='dvi', viewer='superior-dvi-viewer')
```

This will skip auto-detection and will run user specified 'superior-dvi-viewer'. If view fails to find it on your system it will gracefully raise an exception.

You may also enter 'file' for the viewer argument. Doing so will cause this function to return a file object in read-only mode, if filename is unset. However, if it was set, then 'preview' writes the generated file to this filename instead.

There is also support for writing to a `io.BytesIO` like object, which needs to be passed to the `outputbuffer` argument.

```
>>> from io import BytesIO
>>> obj = BytesIO()
>>> preview(x + y, output='png', viewer='BytesIO',
... outputbuffer=obj)
```

The LaTeX preamble can be customized by setting the 'preamble' keyword argument. This can be used, e.g., to set a different font size, use a custom documentclass or import certain set of LaTeX packages.

```
>>> preamble = "\\documentclass[10pt]{article}\\n" \
... "\\usepackage{amsmath,amsfonts}\\begin{document}"
>>> preview(x + y, output='png', preamble=preamble)
```

It is also possible to use the standard preamble and provide additional information to the preamble using the `extra_preamble` keyword argument.

```
>>> from sympy import sin
>>> extra_preamble = "\\renewcommand{\\sin}{\\cos}"
>>> preview(sin(x), output='png', extra_preamble=extra_preamble)
```

If the value of 'output' is different from 'dvi' then command line options can be set ('dvioptions' argument) for the execution of the 'dvi'+output conversion tool. These options have to be in the form of a list of strings (see `subprocess.Popen`).

Additional keyword args will be passed to the `latex()` (page 2170) call, e.g., the `symbol_names` flag.

```
>>> phidd = Symbol('phidd')
>>> preview(phidd, symbol_names={phidd: r'\ddot{\varphi}'})
```

For post-processing the generated TeX File can be written to a file by passing the desired filename to the 'outputTexFile' keyword argument. To write the TeX code to a file named "sample.tex" and run the default png viewer to display the resulting bitmap, do

```
>>> preview(x + y, outputTexFile="sample.tex")
```

## Implementation - Helper Classes/Functions

`sympy.printing.conventions.split_super_sub(text)`

Split a symbol name into a name, superscripts and subscripts

The first part of the symbol name is considered to be its actual 'name', followed by super- and subscripts. Each superscript is preceded with a "^" character or by "\_". Each subscript is preceded by a "\_" character. The three return values are the actual name, a list with superscripts and a list with subscripts.

### Examples

```
>>> from sympy.printing.conventions import split_super_sub
>>> split_super_sub('a_x^1')
('a', ['1'], ['x'])
>>> split_super_sub('var_sub1__sup_sub2')
('var', ['sup'], ['sub1', 'sub2'])
```

## CodePrinter

This class is a base class for other classes that implement code-printing functionality, and additionally lists a number of functions that cannot be easily translated to C or Fortran.

**class** `sympy.printing.codeprinter.CodePrinter(settings=None)`

The base class for code-printing subclasses.

**printmethod:** `str = '_sympystr'`

**doprint**(*expr*, *assign\_to=None*)

Print the expression as code.

### Parameters

**expr** : Expression

The expression to be printed.

**assign\_to** : Symbol, string, MatrixSymbol, list of strings or Symbols (optional)

If provided, the printed code will set the expression to a variable or multiple variables with the name or names given in `assign_to`.

**exception** `sympy.printing.codeprinter.AssignmentError`

Raised if an assignment variable for a loop is missing.

## Precedence

```
sympy.printing.precedence.PRECEDENCE = {'Add': 40, 'And': 30, 'Atom': 1000,
'BitwiseAnd': 38, 'BitwiseOr': 36, 'BitwiseXor': 37, 'Func': 70, 'Lambda': 1,
'Mul': 50, 'Not': 100, 'Or': 20, 'Pow': 60, 'Relational': 35, 'Xor': 10}
```

Default precedence values for some basic types.

```
sympy.printing.precedence.PRECEDENCE_VALUES = {'Add': 40, 'And': 30,
'Equality': 50, 'Equivalent': 10, 'Function': 70, 'HadamardPower': 60,
'HadamardProduct': 50, 'Implies': 10, 'KroneckerProduct': 50, 'MatAdd': 40,
'MatPow': 60, 'MatrixSolve': 50, 'Mod': 50, 'NegativeInfinity': 40, 'Not':
100, 'Or': 20, 'Pow': 60, 'Relational': 35, 'Sub': 40, 'TensAdd': 40,
'TensMul': 50, 'Unequality': 50, 'Xor': 10}
```

A dictionary assigning precedence values to certain classes. These values are treated like they were inherited, so not every single class has to be named here.

```
sympy.printing.precedence.PRECEDENCE_FUNCTIONS = {'Float': <function
precedence_Float>, 'FracElement': <function precedence_FracElement>,
'Integer': <function precedence_Integer>, 'Mul': <function precedence_Mul>,
'PolyElement': <function precedence_PolyElement>, 'Rational': <function
precedence_Rational>, 'UnevaluatedExpr': <function
precedence_UnevaluatedExpr>}
```

Sometimes it's not enough to assign a fixed precedence value to a class. Then a function can be inserted in this dictionary that takes an instance of this class as argument and returns the appropriate precedence value.

```
sympy.printing.precedence.precedence(item)
```

Returns the precedence of a given object.

This is the precedence for StrPrinter.

## Pretty-Printing Implementation Helpers

```
sympy.printing.pretty.pretty_symbology.U(name)
```

Get a unicode character by name or, None if not found.

This exists because older versions of Python use older unicode databases.

```
sympy.printing.pretty.pretty_symbology.pretty_use_unicode(flag=None)
```

Set whether pretty-printer should use unicode by default

```
sympy.printing.pretty.pretty_symbology.pretty_try_use_unicode()
```

See if unicode output is available and leverage it if possible

```
sympy.printing.pretty.pretty_symbology.xstr(*args)
```

The following two functions return the Unicode version of the inputted Greek letter.

```
sympy.printing.pretty.pretty_symbology.g(l)
```

```
sympy.printing.pretty.pretty_symbology.G(l)
```

```
sympy.printing.pretty.pretty_symbology.greek_letters = ['alpha', 'beta',
'gamma', 'delta', 'epsilon', 'zeta', 'eta', 'theta', 'iota', 'kappa', 'lamda',
'mu', 'nu', 'xi', 'omicron', 'pi', 'rho', 'sigma', 'tau', 'upsilon', 'phi',
'chi', 'psi', 'omega']
```



Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

```
sympy.printing.pretty.pretty_symbology.digit_2txt = {'0': 'ZERO', '1': 'ONE',
'2': 'TWO', '3': 'THREE', '4': 'FOUR', '5': 'FIVE', '6': 'SIX', '7': 'SEVEN',
'8': 'EIGHT', '9': 'NINE'}
```

```
sympy.printing.pretty.pretty_symbology.symb_2txt = {'(': 'LEFT PARENTHESIS',
')': 'RIGHT PARENTHESIS', '+': 'PLUS SIGN', '-': 'MINUS', '=': 'EQUALS SIGN',
 '[': 'LEFT SQUARE BRACKET', ']': 'RIGHT SQUARE BRACKET', 'int': 'INTEGRAL',
 'sum': 'SUMMATION', '{': 'LEFT CURLY BRACKET', '}': 'CURLY BRACKET', '}'':
 'RIGHT CURLY BRACKET'}
```

The following functions return the Unicode subscript/superscript version of the character.

```
sympy.printing.pretty.pretty_symbology.sub = {'(': '(', ')': ')', '+': '+',
'-': '-', '0': '₀', '1': '₁', '2': '₂', '3': '₃', '4': '₄', '5': '₅', '6': '₆',
'7': '₇', '8': '₈', '9': '₉', '=': '=', 'a': 'ₐ', 'beta': 'β', 'chi': 'χ',
'e': 'ₑ', 'gamma': 'γ', 'h': 'ₕ', 'i': 'ᵢ', 'k': 'ₖ', 'l': 'ₗ', 'm': 'ₘ', 'n':
'ₙ', 'o': 'ₒ', 'p': 'ₚ', 'phi': 'φ', 'r': 'ᵣ', 'rho': 'ρ', 's': 'ₛ', 't': 'ₜ',
'u': 'ᵤ', 'v': 'ᵥ', 'x': 'ₓ'}
```

```
sympy.printing.pretty.pretty_symbology.sup = {'(': '(', ')': ')', '+': '+',
'-': '-', '0': '⁰', '1': '¹', '2': '²', '3': '³', '4': '⁴', '5': '⁵', '6': '⁶',
'7': '⁷', '8': '⁸', '9': '⁹', '=': '=', 'i': 'ⁱ', 'n': 'ⁿ'}
```

The following functions return Unicode vertical objects.

```
sympy.printing.pretty.pretty_symbology.xobj(symb, length)
```

Construct spatial object of given length.

return: [] of equal-length strings

```
sympy.printing.pretty.pretty_symbology.vobj(symb, height)
```

Construct vertical object of a given height

see: xobj

```
sympy.printing.pretty.pretty_symbology.hobj(symb, width)
```

Construct horizontal object of a given width

see: xobj

The following constants are for rendering roots and fractions.

```
sympy.printing.pretty.pretty_symbology.root = {2: '√', 3: '³√', 4: '⁴√'}
```

```
sympy.printing.pretty.pretty_symbology.VF(txt)
```

```
sympy.printing.pretty.pretty_symbology.frac = {(1, 2): '½', (1, 3): '⅓', (1,
4): '¼', (1, 5): '⅕', (1, 6): '⅙', (1, 8): '⅙', (2, 3): '⅔', (2, 5): '⅖', (3,
4): '¾', (3, 5): '⅗', (3, 8): '⅜', (4, 5): '⅘', (5, 6): '⅚', (5, 8): '⅝', (7,
8): '⅞'}
```

The following constants/functions are for rendering atoms and symbols.

```
sympy.printing.pretty.pretty_symbology.xsym(sym)
```

get symbology for a 'character'

```
sympy.printing.pretty.pretty_symbology.atoms_table = {'Complexes': 'C',
'EmptySequence': 'EmptySequence', 'EmptySet': '∅', 'Exp1': 'e',
'ImaginaryUnit': 'i', 'Infinity': '∞', 'Integers': 'Z', 'Intersection': '∩',
'Modifier Letter Low Ring': '◌', 'Naturals': 'N', 'Naturals0': 'ℕ₀',
'NegativeInfinity': '-∞', 'Pi': 'π', 'Rationals': 'Q', 'Reals': 'R', 'Ring':
'◊', 'SymmetricDifference': 'Δ', 'Union': '∪'}
```

```
sympy.printing.pretty.pretty_symbology.pretty_atom(atom_name, default=None,
 printer=None)
```

return pretty representation of an atom

```
sympy.printing.pretty.pretty_symbology.pretty_symbol(symb_name,
 bold_name=False)
```

return pretty representation of a symbol

```
sympy.printing.pretty.pretty_symbology.annotated(letter)
```

Return a stylised drawing of the letter letter, together with information on how to put annotations (super- and subscripts to the left and to the right) on it.

See pretty.py functions `_print_meijerg`, `_print_hyper` on how to use this information.

Prettyprinter by Jurjen Bos. (I hate spammers: mail me at pietjepuk314 at the reverse of ku.oc.oohay). All objects have a method that create a "stringPict", that can be used in the `str` method for pretty printing.

### Updates by Jason Gedge (email <my last name> at cs mun ca)

- `terminal_string()` method
- minor fixes and changes (mostly to `prettyForm`)

### TODO:

- Allow left/center/right alignment options for above/below and top/center/bottom alignment options for left/right

```
class sympy.printing.pretty.stringpict.stringPict(s, baseline=0)
```

An ASCII picture. The pictures are represented as a list of equal length strings.

**above**(\*args)

Put pictures above this picture. Returns string, baseline arguments for `stringPict`. Baseline is baseline of bottom picture.

**below**(\*args)

Put pictures under this picture. Returns string, baseline arguments for `stringPict`. Baseline is baseline of top picture

## Examples

```
>>> from sympy.printing.pretty.stringpict import stringPict
>>> print(stringPict("x+3").below(
... stringPict.LINE, '3')[0])
x+3

3
```

### **height()**

The height of the picture in characters.

### **left(\*args)**

Put pictures (left to right) at left. Returns string, baseline arguments for stringPict.

### **leftslash()**

Precede object by a slash of the proper size.

### **static next(\*args)**

Put a string of stringPicts next to each other. Returns string, baseline arguments for stringPict.

### **parens(left='(', right=')', ifascii\_nougly=False)**

Put parentheses around self. Returns string, baseline arguments for stringPict.

left or right can be None or empty string which means 'no paren from that side'

### **render(\*args, \*\*kwargs)**

Return the string form of self.

Unless the argument `line_break` is set to False, it will break the expression in a form that can be printed on the terminal without being broken up.

### **right(\*args)**

Put pictures next to this one. Returns string, baseline arguments for stringPict. (Multiline) strings are allowed, and are given a baseline of 0.

## Examples

```
>>> from sympy.printing.pretty.stringpict import stringPict
>>> print(stringPict("10").right(" + ", stringPict("1\r-\r2", 1))[0])
1
10 + -
2
```

### **root(n=None)**

Produce a nice root symbol. Produces ugly results for big n inserts.

### **static stack(\*args)**

Put pictures on top of each other, from top to bottom. Returns string, baseline arguments for stringPict. The baseline is the baseline of the second picture. Everything is centered. Baseline is the baseline of the second picture. Strings are allowed. The special value `stringPict.LINE` is a row of '-' extended to the width.

**terminal\_width()**

Return the terminal width if possible, otherwise return 0.

**width()**

The width of the picture in characters.

**class** sympy.printing.pretty.stringpict.**prettyForm**(*s*, *baseline*=0, *binding*=0, *unicode*=None)

Extension of the stringPict class that knows about basic math applications, optimizing double minus signs.

“Binding” is interpreted as follows:

```
ATOM this is an atom: never needs to be parenthesized
FUNC this is a function application: parenthesize if added (?)
DIV this is a division: make wider division if divided
POW this is a power: only parenthesize if exponent
MUL this is a multiplication: parenthesize if powered
ADD this is an addition: parenthesize if multiplied or powered
NEG this is a negative number: optimize if added, parenthesize if
 multiplied or powered
OPEN this is an open object: parenthesize if added, multiplied, or
 powered (example: Piecewise)
```

**static apply**(*function*, \**args*)

Functions of one or more variables.

## dotprint

sympy.printing.dot.**dotprint**(*expr*, *styles*=((<class 'sympy.core.basic.Basic'>, {'color': 'blue', 'shape': 'ellipse'}), (<class 'sympy.core.expr.Expr'>, {'color': 'black'})), *atom*=<function <lambda>>, *maxdepth*=None, *repeat*=True, *labelfunc*=<class 'str'>, \*\**kwargs*)

DOT description of a SymPy expression tree

### Parameters

**styles** : list of lists composed of (Class, mapping), optional

Styles for different classes.

The default is

```
(
 (Basic, {'color': 'blue', 'shape': 'ellipse'}),
 (Expr, {'color': 'black'})
)
```

**atom** : function, optional

Function used to determine if an arg is an atom.

A good choice is `lambda x: not x.args`.

The default is `lambda x: not isinstance(x, Basic)`.

**maxdepth** : integer, optional

The maximum depth.

The default is None, meaning no limit.

**repeat** : boolean, optional

Whether to use different nodes for common subexpressions.

The default is True.

For example, for  $x + x*y$  with `repeat=True`, it will have two nodes for  $x$ ; with `repeat=False`, it will have one node.

**Warning:** Even if a node appears twice in the same object like  $x$  in  $\text{Pow}(x, x)$ , it will still only appear once. Hence, with `repeat=False`, the number of arrows out of an object might not equal the number of args it has.

**labelfunc** : function, optional

A function to create a label for a given leaf node.

The default is `str`.

Another good option is `srepr`.

For example with `str`, the leaf nodes of  $x + 1$  are labeled,  $x$  and  $1$ .

With `srepr`, they are labeled `Symbol('x')` and `Integer(1)`.

**\*\*kwargs** : optional

Additional keyword arguments are included as styles for the graph.

## Examples

```
>>> from sympy import dotprint
>>> from sympy.abc import x
>>> print(dotprint(x+2))
digraph{

Graph style
"ordering"="out"
"rankdir"="TD"

#####
Nodes
#####

"Add(Integer(2), Symbol('x'))_()" ["color"="black", "label"="Add", "shape"
→ "ellipse"];
"Integer(2)_(0,)" ["color"="black", "label"="2", "shape"="ellipse"];
"Symbol('x')_(1,)" ["color"="black", "label"="x", "shape"="ellipse"];

#####
Edges
#####
```

(continues on next page)

(continued from previous page)

```
"Add(Integer(2), Symbol('x'))_()" -> "Integer(2)_(0,)"
"Add(Integer(2), Symbol('x'))_()" -> "Symbol('x')_(1,)"
}
```

## 5.8.8 Topics

### Contents

### Geometry

#### Introduction

The geometry module for SymPy allows one to create two-dimensional geometrical entities, such as lines and circles, and query for information about these entities. This could include asking the area of an ellipse, checking for collinearity of a set of points, or finding the intersection between two lines. The primary use case of the module involves entities with numerical values, but it is possible to also use symbolic representations.

#### Available Entities

The following entities are currently available in the geometry module:

- [\*Point\*](#) (page 2202)
- [\*Line\*](#) (page 2227), [\*Segment\*](#) (page 2233), [\*Ray\*](#) (page 2230)
- [\*Ellipse\*](#) (page 2253), [\*Circle\*](#) (page 2269)
- [\*Polygon\*](#) (page 2273), [\*RegularPolygon\*](#) (page 2284), [\*Triangle\*](#) (page 2293)

Most of the work one will do will be through the properties and methods of these entities, but several global methods exist:

- `intersection(entity1, entity2)`
- `are_similar(entity1, entity2)`
- `convex_hull(points)`

For a full API listing and an explanation of the methods and their return values please see the list of classes at the end of this document.

## Example Usage

The following Python session gives one an idea of how to work with some of the geometry module.

```
>>> from sympy import *
>>> from sympy.geometry import *
>>> x = Point(0, 0)
>>> y = Point(1, 1)
>>> z = Point(2, 2)
>>> zp = Point(1, 0)
>>> Point.is_collinear(x, y, z)
True
>>> Point.is_collinear(x, y, zp)
False
>>> t = Triangle(zp, y, x)
>>> t.area
1/2
>>> t.medians[x]
Segment2D(Point2D(0, 0), Point2D(1, 1/2))
>>> m = t.medians
>>> intersection(m[x], m[y], m[zp])
[Point2D(2/3, 1/3)]
>>> c = Circle(x, 5)
>>> l = Line(Point(5, -5), Point(5, 5))
>>> c.is_tangent(l) # is l tangent to c?
True
>>> l = Line(x, y)
>>> c.is_tangent(l) # is l tangent to c?
False
>>> intersection(c, l)
[Point2D(-5*sqrt(2)/2, -5*sqrt(2)/2), Point2D(5*sqrt(2)/2, 5*sqrt(2)/2)]
```

## Intersection of medians

```
>>> from sympy import symbols
>>> from sympy.geometry import Point, Triangle, intersection

>>> a, b = symbols("a,b", positive=True)

>>> x = Point(0, 0)
>>> y = Point(a, 0)
>>> z = Point(2*a, b)
>>> t = Triangle(x, y, z)

>>> t.area
a*b/2

>>> t.medians[x]
Segment2D(Point2D(0, 0), Point2D(3*a/2, b/2))
```

(continues on next page)

(continued from previous page)

```
>>> intersection(t.medians[x], t.medians[y], t.medians[z])
[Point2D(a, b/3)]
```

### An in-depth example: Pappus' Hexagon Theorem

From Wikipedia ([\[WikiPappus\]](#)):

Given one set of collinear points  $A, B, C$ , and another set of collinear points  $a, b, c$ , then the intersection points  $X, Y, Z$  of line pairs  $Ab$  and  $aB$ ,  $Ac$  and  $aC$ ,  $Bc$  and  $bC$  are collinear.

```
>>> from sympy import *
>>> from sympy.geometry import *
>>>
>>> l1 = Line(Point(0, 0), Point(5, 6))
>>> l2 = Line(Point(0, 0), Point(2, -2))
>>>
>>> def subs_point(l, val):
... """Take an arbitrary point and make it a fixed point."""
... t = Symbol('t', real=True)
... ap = l.arbitrary_point()
... return Point(ap.x.subs(t, val), ap.y.subs(t, val))
...
>>> p11 = subs_point(l1, 5)
>>> p12 = subs_point(l1, 6)
>>> p13 = subs_point(l1, 11)
>>>
>>> p21 = subs_point(l2, -1)
>>> p22 = subs_point(l2, 2)
>>> p23 = subs_point(l2, 13)
>>>
>>> ll1 = Line(p11, p22)
>>> ll2 = Line(p11, p23)
>>> ll3 = Line(p12, p21)
>>> ll4 = Line(p12, p23)
>>> ll5 = Line(p13, p21)
>>> ll6 = Line(p13, p22)
>>>
>>> pp1 = intersection(ll1, ll3)[0]
>>> pp2 = intersection(ll2, ll5)[0]
>>> pp3 = intersection(ll4, ll6)[0]
>>>
>>> Point.is_collinear(pp1, pp2, pp3)
True
```



## References

### Miscellaneous Notes

- The area property of Polygon and Triangle may return a positive or negative value, depending on whether or not the points are oriented counter-clockwise or clockwise, respectively. If you always want a positive value be sure to use the abs function.
- Although Polygon can refer to any type of polygon, the code has been written for simple polygons. Hence, expect potential problems if dealing with complex polygons (overlapping sides).
- Since SymPy is still in its infancy some things may not simplify properly and hence some things that should return True (e.g., Point.is\_collinear) may not actually do so. Similarly, attempting to find the intersection of entities that do intersect may result in an empty result.

### Future Work

#### Truth Setting Expressions

When one deals with symbolic entities, it often happens that an assertion cannot be guaranteed. For example, consider the following code:

```
>>> from sympy import *
>>> from sympy.geometry import *
>>> x,y,z = map(Symbol, 'xyz')
>>> p1,p2,p3 = Point(x, y), Point(y, z), Point(2*x*y, y)
>>> Point.is_collinear(p1, p2, p3)
False
```

Even though the result is currently False, this is not *always* true. If the quantity  $z - y - 2 * y * z + 2 * y * *2 == 0$  then the points will be collinear. It would be really nice to inform the user of this because such a quantity may be useful to a user for further calculation and, at the very least, being nice to know. This could be potentially done by returning an object (e.g., GeometryResult) that the user could use. This actually would not involve an extensive amount of work.

### Three Dimensions and Beyond

Currently a limited subset of the geometry module has been extended to three dimensions, but it certainly would be a good addition to extend more. This would probably involve a fair amount of work since many of the algorithms used are specific to two dimensions.

## Geometry Visualization

The plotting module is capable of plotting geometric entities. See [Plotting Geometric Entities](#) (page 2872) in the plotting module entry.

### Submodules

### Entities

**class** `sympy.geometry.entity.GeometryEntity(*args, **kwargs)`

The base class for all geometrical entities.

This class does not represent any particular geometric entity, it only provides the implementation of some methods common to all subclasses.

**property** `ambient_dimension`

What is the dimension of the space that the object is contained in?

**property** `bounds`

Return a tuple (xmin, ymin, xmax, ymax) representing the bounding rectangle for the geometric figure.

**encloses**(*o*)

Return True if *o* is inside (not on or outside) the boundaries of self.

The object will be decomposed into Points and individual Entities need only define an `encloses_point` method for their class.

### Examples

```
>>> from sympy import RegularPolygon, Point, Polygon
>>> t = Polygon(*RegularPolygon(Point(0, 0), 1, 3).vertices)
>>> t2 = Polygon(*RegularPolygon(Point(0, 0), 2, 3).vertices)
>>> t2.encloses(t)
True
>>> t.encloses(t2)
False
```

**See also:**

[sympy.geometry.ellipse.Ellipse.encloses\\_point](#) (page 2257), [sympy.geometry.polygon.Polygon.encloses\\_point](#) (page 2278)

**intersection**(*o*)

Returns a list of all of the intersections of self with *o*.

## Notes

An entity is not required to implement this method.

If two different types of entities can intersect, the item with higher index in `ordering_of_classes` should implement intersections with anything having a lower index.

### See also:

[`sympy.geometry.util.intersection`](#) (page 2197)

## `is_similar(other)`

Is this geometrical entity similar to another geometrical entity?

Two entities are similar if a uniform scaling (enlarging or shrinking) of one of the entities will allow one to obtain the other.

## Notes

This method is not intended to be used directly but rather through the `are_similar` function found in `util.py`. An entity is not required to implement this method. If two different types of entities can be similar, it is only required that one of them be able to determine this.

### See also:

[`scale`](#) (page 2196)

## `parameter_value(other, t)`

Return the parameter corresponding to the given point. Evaluating an arbitrary point of the entity at this parameter value will return the given point.

## Examples

```

>>> from sympy import Line, Point
>>> from sympy.abc import t
>>> a = Point(0, 0)
>>> b = Point(2, 2)
>>> Line(a, b).parameter_value((1, 1), t)
{t: 1/2}
>>> Line(a, b).arbitrary_point(t).subs(_)
Point2D(1, 1)

```

## `reflect(line)`

Reflects an object across a line.

### Parameters

**line:** `Line`

## Examples

```
>>> from sympy import pi, sqrt, Line, RegularPolygon
>>> l = Line((0, pi), slope=sqrt(2))
>>> pent = RegularPolygon((1, 2), 1, 5)
>>> rpent = pent.reflect(l)
>>> rpent
RegularPolygon(Point2D(-2*sqrt(2)*pi/3 - 1/3 + 4*sqrt(2)/3, 2/3 +
↳ 2*sqrt(2)/3 + 2*pi/3), -1, 5, -atan(2*sqrt(2)) + 3*pi/5)
```

```
>>> from sympy import pi, Line, Circle, Point
>>> l = Line((0, pi), slope=1)
>>> circ = Circle(Point(0, 0), 5)
>>> rcirc = circ.reflect(l)
>>> rcirc
Circle(Point2D(-pi, pi), -5)
```

**rotate**(*angle*, *pt=None*)

Rotate *angle* radians counterclockwise about Point *pt*.

The default *pt* is the origin, Point(0, 0)

## Examples

```
>>> from sympy import Point, RegularPolygon, Polygon, pi
>>> t = Polygon(*RegularPolygon(Point(0, 0), 1, 3).vertices)
>>> t # vertex on x axis
Triangle(Point2D(1, 0), Point2D(-1/2, sqrt(3)/2), Point2D(-1/2, -
↳ sqrt(3)/2))
>>> t.rotate(pi/2) # vertex on y axis now
Triangle(Point2D(0, 1), Point2D(-sqrt(3)/2, -1/2), Point2D(sqrt(3)/2,
↳ -1/2))
```

**See also:**

[scale](#) (page 2196), [translate](#) (page 2197)

**scale**(*x=1*, *y=1*, *pt=None*)

Scale the object by multiplying the x,y-coordinates by *x* and *y*.

If *pt* is given, the scaling is done relative to that point; the object is shifted by -*pt*, scaled, and shifted by *pt*.

## Examples

```
>>> from sympy import RegularPolygon, Point, Polygon
>>> t = Polygon(*RegularPolygon(Point(0, 0), 1, 3).vertices)
>>> t
Triangle(Point2D(1, 0), Point2D(-1/2, sqrt(3)/2), Point2D(-1/2, -
↳sqrt(3)/2))
>>> t.scale(2)
Triangle(Point2D(2, 0), Point2D(-1, sqrt(3)/2), Point2D(-1, -sqrt(3)/
↳2))
>>> t.scale(2, 2)
Triangle(Point2D(2, 0), Point2D(-1, sqrt(3)), Point2D(-1, -sqrt(3)))
```

### See also:

[rotate](#) (page 2196), [translate](#) (page 2197)

**translate**(*x=0, y=0*)

Shift the object by adding to the x,y-coordinates the values x and y.

## Examples

```
>>> from sympy import RegularPolygon, Point, Polygon
>>> t = Polygon(*RegularPolygon(Point(0, 0), 1, 3).vertices)
>>> t
Triangle(Point2D(1, 0), Point2D(-1/2, sqrt(3)/2), Point2D(-1/2, -
↳sqrt(3)/2))
>>> t.translate(2)
Triangle(Point2D(3, 0), Point2D(3/2, sqrt(3)/2), Point2D(3/2, -
↳sqrt(3)/2))
>>> t.translate(2, 2)
Triangle(Point2D(3, 2), Point2D(3/2, sqrt(3)/2 + 2), Point2D(3/2, 2 -
↳sqrt(3)/2))
```

### See also:

[rotate](#) (page 2196), [scale](#) (page 2196)

## Utils

`sympy.geometry.util.intersection(*entities, pairwise=False, **kwargs)`

The intersection of a collection of GeometryEntity instances.

### Parameters

**entities** : sequence of GeometryEntity

**pairwise (keyword argument)** : Can be either True or False

### Returns

**intersection** : list of GeometryEntity

### Raises

**NotImplementedError**

When unable to calculate intersection.

## Notes

The intersection of any geometrical entity with itself should return a list with one item: the entity in question. An intersection requires two or more entities. If only a single entity is given then the function will return an empty list. It is possible for *intersection* to miss intersections that one knows exists because the required quantities were not fully simplified internally. Reals should be converted to Rationals, e.g. `Rational(str(real_num))` or else failures due to floating point issues may result.

Case 1: When the keyword argument 'pairwise' is False (default value): In this case, the function returns a list of intersections common to all entities.

Case 2: When the keyword argument 'pairwise' is True: In this case, the functions returns a list intersections that occur between any pair of entities.

## Examples

```
>>> from sympy import Ray, Circle, intersection
>>> c = Circle((0, 1), 1)
>>> intersection(c, c.center)
[]
>>> right = Ray((0, 0), (1, 0))
>>> up = Ray((0, 0), (0, 1))
>>> intersection(c, right, up)
[Point2D(0, 0)]
>>> intersection(c, right, up, pairwise=True)
[Point2D(0, 0), Point2D(0, 2)]
>>> left = Ray((1, 0), (0, 0))
>>> intersection(right, left)
[Segment2D(Point2D(0, 0), Point2D(1, 0))]
```

### See also:

[`sympy.geometry.entity.GeometryEntity.intersection`](#) (page 2194)

`sympy.geometry.util.convex_hull(*args, polygon=True)`

The convex hull surrounding the Points contained in the list of entities.

### Parameters

**args** : a collection of Points, Segments and/or Polygons

### Returns

**convex\_hull** : Polygon if `polygon` is True else as a tuple (*U*, *L*) where *L* and *U* are the lower and upper hulls, respectively.

## Optional Parameters

### **polygon**

[Boolean. If True, returns a Polygon, if false a tuple, see below.] Default is True.

## Notes

This can only be performed on a set of points whose coordinates can be ordered on the number line.

## Examples

```
>>> from sympy import convex_hull
>>> points = [(1, 1), (1, 2), (3, 1), (-5, 2), (15, 4)]
>>> convex_hull(*points)
Polygon(Point2D(-5, 2), Point2D(1, 1), Point2D(3, 1), Point2D(15, 4))
>>> convex_hull(*points, **dict(polygon=False))
([Point2D(-5, 2), Point2D(15, 4)],
 [Point2D(-5, 2), Point2D(1, 1), Point2D(3, 1), Point2D(15, 4)])
```

### See also:

[\*sympy.geometry.point.Point\*](#) (page 2202), [\*sympy.geometry.polygon.Polygon\*](#) (page 2273)

## References

[R521], [R522]

`sympy.geometry.util.are_similar(e1, e2)`

Are two geometrical entities similar.

Can one geometrical entity be uniformly scaled to the other?

### Parameters

**e1** : GeometryEntity

**e2** : GeometryEntity

### Returns

**are\_similar** : boolean

### Raises

**GeometryError**

When *e1* and *e2* cannot be compared.

## Notes

If the two objects are equal then they are similar.

## Examples

```
>>> from sympy import Point, Circle, Triangle, are_similar
>>> c1, c2 = Circle(Point(0, 0), 4), Circle(Point(1, 4), 3)
>>> t1 = Triangle(Point(0, 0), Point(1, 0), Point(0, 1))
>>> t2 = Triangle(Point(0, 0), Point(2, 0), Point(0, 2))
>>> t3 = Triangle(Point(0, 0), Point(3, 0), Point(0, 1))
>>> are_similar(t1, t2)
True
>>> are_similar(t1, t3)
False
```

## See also:

[\*sympy.geometry.entity.GeometryEntity.is\\_similar\*](#) (page 2195)

`sympy.geometry.util.centroid(*args)`

Find the centroid (center of mass) of the collection containing only Points, Segments or Polygons. The centroid is the weighted average of the individual centroid where the weights are the lengths (of segments) or areas (of polygons). Overlapping regions will add to the weight of that region.

If there are no objects (or a mixture of objects) then None is returned.

## Examples

```
>>> from sympy import Point, Segment, Polygon
>>> from sympy.geometry.util import centroid
>>> p = Polygon((0, 0), (10, 0), (10, 10))
>>> q = p.translate(0, 20)
>>> p.centroid, q.centroid
(Point2D(20/3, 10/3), Point2D(20/3, 70/3))
>>> centroid(p, q)
Point2D(20/3, 40/3)
>>> p, q = Segment((0, 0), (2, 0)), Segment((0, 0), (2, 2))
>>> centroid(p, q)
Point2D(1, 2 - sqrt(2))
>>> centroid(Point(0, 0), Point(2, 0))
Point2D(1, 0)
```

Stacking 3 polygons on top of each other effectively triples the weight of that polygon:

```
>>> p = Polygon((0, 0), (1, 0), (1, 1), (0, 1))
>>> q = Polygon((1, 0), (3, 0), (3, 1), (1, 1))
>>> centroid(p, q)
Point2D(3/2, 1/2)
>>> centroid(p, p, p, q) # centroid x-coord shifts left
Point2D(11/10, 1/2)
```



Stacking the squares vertically above and below  $p$  has the same effect:

```
>>> centroid(p, p.translate(0, 1), p.translate(0, -1), q)
Point2D(11/10, 1/2)
```

**See also:**

[\*sympy.geometry.point.Point\*](#) (page 2202), [\*sympy.geometry.line.Segment\*](#) (page 2233), [\*sympy.geometry.polygon.Polygon\*](#) (page 2273)

`sympy.geometry.util.idiff(eq, y, x, n=1)`

Return  $dy/dx$  assuming that  $eq == 0$ .

#### Parameters

**y** : the dependent variable or a list of dependent variables (with  $y$  first)

**x** : the variable that the derivative is being taken with respect to

**n** : the order of the derivative (default is 1)

#### Examples

```
>>> from sympy.abc import x, y, a
>>> from sympy.geometry.util import idiff
```

```
>>> circ = x**2 + y**2 - 4
>>> idiff(circ, y, x)
-x/y
>>> idiff(circ, y, x, 2).simplify()
(-x**2 - y**2)/y**3
```

Here,  $a$  is assumed to be independent of  $x$ :

```
>>> idiff(x + a + y, y, x)
-1
```

Now the  $x$ -dependence of  $a$  is made explicit by listing  $a$  after  $y$  in a list.

```
>>> idiff(x + a + y, [y, a], x)
-Derivative(a, x) - 1
```

**See also:**

[\*sympy.core.function.Derivative\*](#) (page 1042)  
represents unevaluated derivatives

[\*sympy.core.function.diff\*](#) (page 1048)  
explicitly differentiates wrt symbols

## Points

**class** `sympy.geometry.point.Point(*args, **kwargs)`

A point in a n-dimensional Euclidean space.

### Parameters

**coords** : sequence of n-coordinate values. In the special case where  $n=2$  or  $3$ , a `Point2D` or `Point3D` will be created as appropriate.

**evaluate** : if `True` (default), all floats are turn into exact types.

**dim** : number of coordinates the point should have. If coordinates are unspecified, they are padded with zeros.

**on\_morph** : indicates what should happen when the number of coordinates of a point need to be changed by adding or removing zeros. Possible values are `'warn'`, `'error'`, or `ignore` (default). No warning or error is given when `*args` is empty and `dim` is given. An error is always raised when trying to remove nonzero coordinates.

### Raises

**TypeError** : When instantiating with anything but a `Point` or sequence

**ValueError** : when instantiating with a sequence with length  $< 2$  or when trying to reduce dimensions if keyword `on_morph = 'error'` is set.

## Examples

```
>>> from sympy import Point
>>> from sympy.abc import x
>>> Point(1, 2, 3)
Point3D(1, 2, 3)
>>> Point([1, 2])
Point2D(1, 2)
>>> Point(0, x)
Point2D(0, x)
>>> Point(dim=4)
Point(0, 0, 0, 0)
```

Floats are automatically converted to Rational unless the `evaluate` flag is `False`:

```
>>> Point(0.5, 0.25)
Point2D(1/2, 1/4)
>>> Point(0.5, 0.25, evaluate=False)
Point2D(0.5, 0.25)
```

**See also:**

[\*sympy.geometry.line.Segment\*](#) (page 2233)

Connects two Points

## Attributes

|                                                       |                                  |
|-------------------------------------------------------|----------------------------------|
| length                                                |                                  |
| origin: A <i>Point</i> representing the origin of the | appropriately-dimensioned space. |

### static `affine_rank(*args)`

The affine rank of a set of points is the dimension of the smallest affine space containing all the points. For example, if the points lie on a line (and are not all the same) their affine rank is 1. If the points lie on a plane but not a line, their affine rank is 2. By convention, the empty set has affine rank -1.

### property `ambient_dimension`

Number of components this point has.

### classmethod `are_coplanar(*points)`

Return True if there exists a plane in which all the points lie. A trivial True value is returned if  $\text{len}(\text{points}) < 3$  or all Points are 2-dimensional.

#### Parameters

**A set of points**

#### Returns

boolean

#### Raises

**ValueError** : if less than 3 unique points are given

## Examples

```
>>> from sympy import Point3D
>>> p1 = Point3D(1, 2, 2)
>>> p2 = Point3D(2, 7, 2)
>>> p3 = Point3D(0, 0, 2)
>>> p4 = Point3D(1, 1, 2)
>>> Point3D.are_coplanar(p1, p2, p3, p4)
True
>>> p5 = Point3D(0, 1, 3)
>>> Point3D.are_coplanar(p1, p2, p3, p5)
False
```

### `canberra_distance(p)`

The Canberra Distance from self to point p.

Returns the weighted sum of horizontal and vertical distances to point p.

#### Parameters

**p** : Point

#### Returns

**canberra\_distance** : The weighted sum of horizontal and vertical distances to point p. The weight used is the sum of absolute values of the coordinates.

**Raises**

**ValueError** when both vectors are zero.

**Examples**

```
>>> from sympy import Point
>>> p1, p2 = Point(1, 1), Point(3, 3)
>>> p1.canberra_distance(p2)
1
>>> p1, p2 = Point(0, 0), Point(3, 3)
>>> p1.canberra_distance(p2)
2
```

**See also:**

[\*sympy.geometry.point.Point.distance\*](#) (page 2204)

**distance(*other*)**

The Euclidean distance between self and another GeometricEntity.

**Returns**

**distance** : number or symbolic expression.

**Raises**

**TypeError** : if *other* is not recognized as a GeometricEntity or is a GeometricEntity for which distance is not defined.

**Examples**

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(1, 1), Point(4, 5)
>>> l = Line((3, 1), (2, 2))
>>> p1.distance(p2)
5
>>> p1.distance(l)
sqrt(2)
```

The computed distance may be symbolic, too:

```
>>> from sympy.abc import x, y
>>> p3 = Point(x, y)
>>> p3.distance((0, 0))
sqrt(x**2 + y**2)
```

**See also:**

[\*sympy.geometry.line.Segment.length\*](#) (page 2235), [\*sympy.geometry.point.Point.taxicab\\_distance\*](#) (page 2208)

**dot(*p*)**

Return dot product of self with another Point.

**equals**(*other*)

Returns whether the coordinates of self and other agree.

**intersection**(*other*)

The intersection between this point and another GeometryEntity.

**Parameters**

**other** : GeometryEntity or sequence of coordinates

**Returns**

**intersection** : list of Points

## Notes

The return value will either be an empty list if there is no intersection, otherwise it will contain this point.

## Examples

```
>>> from sympy import Point
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(0, 0)
>>> p1.intersection(p2)
[]
>>> p1.intersection(p3)
[Point2D(0, 0)]
```

**is\_collinear**(\*args)

Returns *True* if there exists a line that contains *self* and *points*. Returns *False* otherwise. A trivially True value is returned if no points are given.

**Parameters**

**args** : sequence of Points

**Returns**

**is\_collinear** : boolean

## Examples

```
>>> from sympy import Point
>>> from sympy.abc import x
>>> p1, p2 = Point(0, 0), Point(1, 1)
>>> p3, p4, p5 = Point(2, 2), Point(x, x), Point(1, 2)
>>> Point.is_collinear(p1, p2, p3, p4)
True
>>> Point.is_collinear(p1, p2, p3, p5)
False
```

**See also:**

[\*sympy.geometry.line.Line\*](#) (page 2227)

**is\_concyclic(\*args)**

Do *self* and the given sequence of points lie in a circle?

Returns True if the set of points are concyclic and False otherwise. A trivial value of True is returned if there are fewer than 2 other points.

**Parameters**

**args** : sequence of Points

**Returns**

**is\_concyclic** : boolean

**Examples**

```
>>> from sympy import Point
```

Define 4 points that are on the unit circle:

```
>>> p1, p2, p3, p4 = Point(1, 0), (0, 1), (-1, 0), (0, -1)
```

```
>>> p1.is_concyclic() == p1.is_concyclic(p2, p3, p4) == True
True
```

Define a point not on that circle:

```
>>> p = Point(1, 1)
```

```
>>> p.is_concyclic(p1, p2, p3)
False
```

**property is\_nonzero**

True if any coordinate is nonzero, False if every coordinate is zero, and None if it cannot be determined.

**is\_scalar\_multiple(p)**

Returns whether each coordinate of *self* is a scalar multiple of the corresponding coordinate in point *p*.

**property is\_zero**

True if every coordinate is zero, False if any coordinate is not zero, and None if it cannot be determined.

**property length**

Treating a Point as a Line, this returns 0 for the length of a Point.

## Examples

```
>>> from sympy import Point
>>> p = Point(0, 1)
>>> p.length
0
```

### **midpoint**(*p*)

The midpoint between self and point *p*.

#### **Parameters**

**p** : Point

#### **Returns**

**midpoint** : Point

## Examples

```
>>> from sympy import Point
>>> p1, p2 = Point(1, 1), Point(13, 5)
>>> p1.midpoint(p2)
Point2D(7, 3)
```

### **See also:**

[\*sympy.geometry.line.Segment.midpoint\*](#) (page 2235)

### **property origin**

A point of all zeros of the same ambient dimension as the current point

### **property orthogonal\_direction**

Returns a non-zero point that is orthogonal to the line containing *self* and the origin.

## Examples

```
>>> from sympy import Line, Point
>>> a = Point(1, 2, 3)
>>> a.orthogonal_direction
Point3D(-2, 1, 0)
>>> b = _
>>> Line(b, b.origin).is_perpendicular(Line(a, a.origin))
True
```

### **static project**(*a*, *b*)

Project the point *a* onto the line between the origin and point *b* along the normal direction.

#### **Parameters**

**a** : Point

**b** : Point

#### **Returns**

**p** : Point

## Examples

```
>>> from sympy import Line, Point
>>> a = Point(1, 2)
>>> b = Point(2, 5)
>>> z = a.origin
>>> p = Point.project(a, b)
>>> Line(p, a).is_perpendicular(Line(p, b))
True
>>> Point.is_collinear(z, p, b)
True
```

### See also:

[\*sympy.geometry.line.LinearEntity.projection\*](#) (page 2225)

### **taxicab\_distance(p)**

The Taxicab Distance from self to point p.

Returns the sum of the horizontal and vertical distances to point p.

#### Parameters

**p** : Point

#### Returns

**taxicab\_distance** : The sum of the horizontal and vertical distances to point p.

## Examples

```
>>> from sympy import Point
>>> p1, p2 = Point(1, 1), Point(4, 5)
>>> p1.taxicab_distance(p2)
7
```

### See also:

[\*sympy.geometry.point.Point.distance\*](#) (page 2204)

### **property unit**

Return the Point that is in the same direction as *self* and a distance of 1 from the origin

**class** sympy.geometry.point.**Point2D**(\*args, \_nocheck=False, \*\*kwargs)

A point in a 2-dimensional Euclidean space.

#### Parameters

**coords** : sequence of 2 coordinate values.

#### Raises

##### **TypeError**

When trying to add or subtract points with different dimensions.  
When trying to create a point with more than two dimensions. When *intersection* is called with object other than a Point.



## Examples

```
>>> from sympy import Point2D
>>> from sympy.abc import x
>>> Point2D(1, 2)
Point2D(1, 2)
>>> Point2D([1, 2])
Point2D(1, 2)
>>> Point2D(0, x)
Point2D(0, x)
```

Floats are automatically converted to Rational unless the evaluate flag is False:

```
>>> Point2D(0.5, 0.25)
Point2D(1/2, 1/4)
>>> Point2D(0.5, 0.25, evaluate=False)
Point2D(0.5, 0.25)
```

## See also:

[\*sympy.geometry.Line.Segment\*](#) (page 2233)

Connects two Points

## Attributes

|        |  |
|--------|--|
| x      |  |
| y      |  |
| length |  |

### property bounds

Return a tuple (xmin, ymin, xmax, ymax) representing the bounding rectangle for the geometric figure.

### property coordinates

Returns the two coordinates of the Point.

## Examples

```
>>> from sympy import Point2D
>>> p = Point2D(0, 1)
>>> p.coordinates
(0, 1)
```

### rotate(*angle*, *pt=None*)

Rotate angle radians counterclockwise about Point pt.

## Examples

```
>>> from sympy import Point2D, pi
>>> t = Point2D(1, 0)
>>> t.rotate(pi/2)
Point2D(0, 1)
>>> t.rotate(pi/2, (2, 0))
Point2D(2, -1)
```

### See also:

[\*translate\*](#) (page 2210), [\*scale\*](#) (page 2210)

**scale**(*x=1, y=1, pt=None*)

Scale the coordinates of the Point by multiplying by *x* and *y* after subtracting *pt* – default is (0, 0) – and then adding *pt* back again (i.e. *pt* is the point of reference for the scaling).

## Examples

```
>>> from sympy import Point2D
>>> t = Point2D(1, 1)
>>> t.scale(2)
Point2D(2, 1)
>>> t.scale(2, 2)
Point2D(2, 2)
```

### See also:

[\*rotate\*](#) (page 2209), [\*translate\*](#) (page 2210)

**transform**(*matrix*)

Return the point after applying the transformation described by the 3x3 Matrix, *matrix*.

### See also:

[\*sympy.geometry.point.Point2D.rotate\*](#) (page 2209), [\*sympy.geometry.point.Point2D.scale\*](#) (page 2210), [\*sympy.geometry.point.Point2D.translate\*](#) (page 2210)

**translate**(*x=0, y=0*)

Shift the Point by adding *x* and *y* to the coordinates of the Point.

## Examples

```
>>> from sympy import Point2D
>>> t = Point2D(0, 1)
>>> t.translate(2)
Point2D(2, 1)
>>> t.translate(2, 2)
Point2D(2, 3)
```

(continues on next page)

(continued from previous page)

```
>>> t + Point2D(2, 2)
Point2D(2, 3)
```

**See also:**

[\*sympy.geometry.point.Point2D.rotate\*](#) (page 2209), [\*scale\*](#) (page 2210)

**property x**

Returns the X coordinate of the Point.

**Examples**

```
>>> from sympy import Point2D
>>> p = Point2D(0, 1)
>>> p.x
0
```

**property y**

Returns the Y coordinate of the Point.

**Examples**

```
>>> from sympy import Point2D
>>> p = Point2D(0, 1)
>>> p.y
1
```

**class** sympy.geometry.point.**Point3D**(\*args, \_nocheck=False, \*\*kwargs)

A point in a 3-dimensional Euclidean space.

**Parameters**

**coords** : sequence of 3 coordinate values.

**Raises****TypeError**

When trying to add or subtract points with different dimensions.

When *intersection* is called with object other than a Point.

**Examples**

```
>>> from sympy import Point3D
>>> from sympy.abc import x
>>> Point3D(1, 2, 3)
Point3D(1, 2, 3)
>>> Point3D([1, 2, 3])
Point3D(1, 2, 3)
>>> Point3D(0, x, 3)
Point3D(0, x, 3)
```

Floats are automatically converted to Rational unless the evaluate flag is False:

```
>>> Point3D(0.5, 0.25, 2)
Point3D(1/2, 1/4, 2)
>>> Point3D(0.5, 0.25, 3, evaluate=False)
Point3D(0.5, 0.25, 3)
```

### Attributes

|        |  |
|--------|--|
| x      |  |
| y      |  |
| z      |  |
| length |  |

**static** `are_collinear(*points)`

Is a sequence of points collinear?

Test whether or not a set of points are collinear. Returns True if the set of points are collinear, or False otherwise.

#### Parameters

**points** : sequence of Point

#### Returns

**are\_collinear** : boolean

### Examples

```
>>> from sympy import Point3D
>>> from sympy.abc import x
>>> p1, p2 = Point3D(0, 0, 0), Point3D(1, 1, 1)
>>> p3, p4, p5 = Point3D(2, 2, 2), Point3D(x, x, x), Point3D(1, 2, 6)
>>> Point3D.are_collinear(p1, p2, p3, p4)
True
>>> Point3D.are_collinear(p1, p2, p3, p5)
False
```

#### See also:

[\*sympy.geometry.line.Line3D\*](#) (page 2243)

### property coordinates

Returns the three coordinates of the Point.

### Examples

```
>>> from sympy import Point3D
>>> p = Point3D(0, 1, 2)
>>> p.coordinates
(0, 1, 2)
```

#### **direction\_cosine**(*point*)

Gives the direction cosine between 2 points

**Parameters**

**p** : Point3D

**Returns**

list

### Examples

```
>>> from sympy import Point3D
>>> p1 = Point3D(1, 2, 3)
>>> p1.direction_cosine(Point3D(2, 3, 5))
[sqrt(6)/6, sqrt(6)/6, sqrt(6)/3]
```

#### **direction\_ratio**(*point*)

Gives the direction ratio between 2 points

**Parameters**

**p** : Point3D

**Returns**

list

### Examples

```
>>> from sympy import Point3D
>>> p1 = Point3D(1, 2, 3)
>>> p1.direction_ratio(Point3D(2, 3, 5))
[1, 1, 2]
```

#### **intersection**(*other*)

The intersection between this point and another GeometryEntity.

**Parameters**

**other** : GeometryEntity or sequence of coordinates

**Returns**

**intersection** : list of Points

## Notes

The return value will either be an empty list if there is no intersection, otherwise it will contain this point.

## Examples

```
>>> from sympy import Point3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(1, 1, 1), Point3D(0, 0, 0)
>>> p1.intersection(p2)
[]
>>> p1.intersection(p3)
[Point3D(0, 0, 0)]
```

**scale**(*x=1, y=1, z=1, pt=None*)

Scale the coordinates of the Point by multiplying by *x* and *y* after subtracting *pt* – default is (0, 0) – and then adding *pt* back again (i.e. *pt* is the point of reference for the scaling).

## Examples

```
>>> from sympy import Point3D
>>> t = Point3D(1, 1, 1)
>>> t.scale(2)
Point3D(2, 1, 1)
>>> t.scale(2, 2)
Point3D(2, 2, 1)
```

### See also:

[\*translate\*](#) (page 2214)

**transform**(*matrix*)

Return the point after applying the transformation described by the 4x4 Matrix, *matrix*.

### See also:

[\*sympy.geometry.point.Point3D.scale\*](#) (page 2214), [\*sympy.geometry.point.Point3D.translate\*](#) (page 2214)

**translate**(*x=0, y=0, z=0*)

Shift the Point by adding *x* and *y* to the coordinates of the Point.

## Examples

```
>>> from sympy import Point3D
>>> t = Point3D(0, 1, 1)
>>> t.translate(2)
Point3D(2, 1, 1)
>>> t.translate(2, 2)
Point3D(2, 3, 1)
>>> t + Point3D(2, 2, 2)
Point3D(2, 3, 3)
```

### See also:

[scale](#) (page 2214)

### property `x`

Returns the X coordinate of the Point.

## Examples

```
>>> from sympy import Point3D
>>> p = Point3D(0, 1, 3)
>>> p.x
0
```

### property `y`

Returns the Y coordinate of the Point.

## Examples

```
>>> from sympy import Point3D
>>> p = Point3D(0, 1, 2)
>>> p.y
1
```

### property `z`

Returns the Z coordinate of the Point.

## Examples

```
>>> from sympy import Point3D
>>> p = Point3D(0, 1, 1)
>>> p.z
1
```

## Lines

**class** `sympy.geometry.line.LinearEntity(p1, p2=None, **kwargs)`

A base class for all linear entities (Line, Ray and Segment) in n-dimensional Euclidean space.

### Notes

This is an abstract class and is not meant to be instantiated.

### See also:

[\*sympy.geometry.entity.GeometryEntity\*](#) (page 2194)

### Attributes

|                                |  |
|--------------------------------|--|
| <code>ambient_dimension</code> |  |
| <code>direction</code>         |  |
| <code>length</code>            |  |
| <code>p1</code>                |  |
| <code>p2</code>                |  |
| <code>points</code>            |  |

### property `ambient_dimension`

A property method that returns the dimension of LinearEntity object.

#### Parameters

**p1** : LinearEntity

#### Returns

**dimension** : integer

### Examples

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(1, 1)
>>> l1 = Line(p1, p2)
>>> l1.ambient_dimension
2
```

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0, 0), Point(1, 1, 1)
>>> l1 = Line(p1, p2)
>>> l1.ambient_dimension
3
```

### `angle_between(l2)`

Return the non-reflex angle formed by rays emanating from the origin with directions the same as the direction vectors of the linear entities.



**Parameters****l1** : LinearEntity**l2** : LinearEntity**Returns****angle** : angle in radians**Notes**

From the dot product of vectors  $v_1$  and  $v_2$  it is known that:

$$\text{dot}(v_1, v_2) = |v_1| * |v_2| * \cos(A)$$

where  $A$  is the angle formed between the two vectors. We can get the directional vectors of the two lines and readily find the angle between the two using the above formula.

**Examples**

```
>>> from sympy import Line
>>> e = Line((0, 0), (1, 0))
>>> ne = Line((0, 0), (1, 1))
>>> sw = Line((1, 1), (0, 0))
>>> ne.angle_between(e)
pi/4
>>> sw.angle_between(e)
3*pi/4
```

To obtain the non-obtuse angle at the intersection of lines, use the `smallest_angle_between` method:

```
>>> sw.smallest_angle_between(e)
pi/4
```

```
>>> from sympy import Point3D, Line3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(1, 1, 1), Point3D(-1, 2, 0)
>>> l1, l2 = Line3D(p1, p2), Line3D(p2, p3)
>>> l1.angle_between(l2)
acos(-sqrt(2)/3)
>>> l1.smallest_angle_between(l2)
acos(sqrt(2)/3)
```

**See also:**

[\*is\\_perpendicular\*](#) (page 2221), [\*Ray2D.closing\\_angle\*](#) (page 2240)

**arbitrary\_point**(*parameter*='t')

A parameterized point on the Line.

**Parameters****parameter** : str, optional

The name of the parameter which will be used for the parametric point. The default value is 't'. When this parameter is 0, the first

point used to define the line will be returned, and when it is 1 the second point will be returned.

**Returns**

**point** : Point

**Raises**

**ValueError**

When parameter already appears in the Line's definition.

**Examples**

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(1, 0), Point(5, 3)
>>> l1 = Line(p1, p2)
>>> l1.arbitrary_point()
Point2D(4*t + 1, 3*t)
>>> from sympy import Point3D, Line3D
>>> p1, p2 = Point3D(1, 0, 0), Point3D(5, 3, 1)
>>> l1 = Line3D(p1, p2)
>>> l1.arbitrary_point()
Point3D(4*t + 1, 3*t, t)
```

**See also:**

[\*sympy.geometry.point.Point\*](#) (page 2202)

**static are\_concurrent(\*lines)**

Is a sequence of linear entities concurrent?

Two or more linear entities are concurrent if they all intersect at a single point.

**Parameters**

**lines** : a sequence of linear entities.

**Returns**

**True** : if the set of linear entities intersect in one point

**False** : otherwise.

**Examples**

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(3, 5)
>>> p3, p4 = Point(-2, -2), Point(0, 2)
>>> l1, l2, l3 = Line(p1, p2), Line(p1, p3), Line(p1, p4)
>>> Line.are_concurrent(l1, l2, l3)
True
>>> l4 = Line(p2, p3)
>>> Line.are_concurrent(l2, l3, l4)
False
>>> from sympy import Point3D, Line3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(3, 5, 2)
>>> p3, p4 = Point3D(-2, -2, -2), Point3D(0, 2, 1)
```

(continues on next page)

(continued from previous page)

```

>>> l1, l2, l3 = Line3D(p1, p2), Line3D(p1, p3), Line3D(p1, p4)
>>> Line3D.are_concurrent(l1, l2, l3)
True
>>> l4 = Line3D(p2, p3)
>>> Line3D.are_concurrent(l2, l3, l4)
False

```

**See also:**

[\*sympy.geometry.util.intersection\*](#) (page 2197)

**bisectors(*other*)**

Returns the perpendicular lines which pass through the intersections of self and other that are in the same plane.

**Parameters**

**line** : Line3D

**Returns**

list: two Line instances

**Examples**

```

>>> from sympy import Point3D, Line3D
>>> r1 = Line3D(Point3D(0, 0, 0), Point3D(1, 0, 0))
>>> r2 = Line3D(Point3D(0, 0, 0), Point3D(0, 1, 0))
>>> r1.bisectors(r2)
[Line3D(Point3D(0, 0, 0), Point3D(1, 1, 0)), Line3D(Point3D(0, 0, 0),
↪ Point3D(1, -1, 0))]

```

**contains(*other*)**

Subclasses should implement this method and should return True if other is on the boundaries of self; False if not on the boundaries of self; None if a determination cannot be made.

**property direction**

The direction vector of the LinearEntity.

**Returns**

**p** : a Point; the ray from the origin to this point is the direction of *self*

**Examples**

```

>>> from sympy import Line
>>> a, b = (1, 1), (1, 3)
>>> Line(a, b).direction
Point2D(0, 2)
>>> Line(b, a).direction
Point2D(0, -2)

```

This can be reported so the distance from the origin is 1:

```
>>> Line(b, a).direction.unit
Point2D(0, -1)
```

**See also:**

[\*sympy.geometry.point.Point.unit\*](#) (page 2208)

**intersection**(*other*)

The intersection with another geometrical entity.

**Parameters**

**o** : Point or LinearEntity

**Returns**

**intersection** : list of geometrical entities

**Examples**

```
>>> from sympy import Point, Line, Segment
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(7, 7)
>>> l1 = Line(p1, p2)
>>> l1.intersection(p3)
[Point2D(7, 7)]
>>> p4, p5 = Point(5, 0), Point(0, 3)
>>> l2 = Line(p4, p5)
>>> l1.intersection(l2)
[Point2D(15/8, 15/8)]
>>> p6, p7 = Point(0, 5), Point(2, 6)
>>> s1 = Segment(p6, p7)
>>> l1.intersection(s1)
[]
>>> from sympy import Point3D, Line3D, Segment3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(1, 1, 1), Point3D(7, 7, 7)
>>> l1 = Line3D(p1, p2)
>>> l1.intersection(p3)
[Point3D(7, 7, 7)]
>>> l1 = Line3D(Point3D(4, 19, 12), Point3D(5, 25, 17))
>>> l2 = Line3D(Point3D(-3, -15, -19), direction_ratio=[2, 8, 8])
>>> l1.intersection(l2)
[Point3D(1, 1, -3)]
>>> p6, p7 = Point3D(0, 5, 2), Point3D(2, 6, 3)
>>> s1 = Segment3D(p6, p7)
>>> l1.intersection(s1)
[]
```

**See also:**

[\*sympy.geometry.point.Point\*](#) (page 2202)

**is\_parallel**(*l2*)

Are two linear entities parallel?

**Parameters**

**l1** : LinearEntity

**l2** : LinearEntity

**Returns****True** : if  $l1$  and  $l2$  are parallel,**False** : otherwise.**Examples**

```

>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(1, 1)
>>> p3, p4 = Point(3, 4), Point(6, 7)
>>> l1, l2 = Line(p1, p2), Line(p3, p4)
>>> Line.is_parallel(l1, l2)
True
>>> p5 = Point(6, 6)
>>> l3 = Line(p3, p5)
>>> Line.is_parallel(l1, l3)
False
>>> from sympy import Point3D, Line3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(3, 4, 5)
>>> p3, p4 = Point3D(2, 1, 1), Point3D(8, 9, 11)
>>> l1, l2 = Line3D(p1, p2), Line3D(p3, p4)
>>> Line3D.is_parallel(l1, l2)
True
>>> p5 = Point3D(6, 6, 6)
>>> l3 = Line3D(p3, p5)
>>> Line3D.is_parallel(l1, l3)
False

```

**See also:***coefficients* (page 2239)**is\_perpendicular( $l2$ )**

Are two linear entities perpendicular?

**Parameters** **$l1$**  : LinearEntity **$l2$**  : LinearEntity**Returns****True** : if  $l1$  and  $l2$  are perpendicular,**False** : otherwise.**Examples**

```

>>> from sympy import Point, Line
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(-1, 1)
>>> l1, l2 = Line(p1, p2), Line(p1, p3)
>>> l1.is_perpendicular(l2)
True
>>> p4 = Point(5, 3)
>>> l3 = Line(p1, p4)

```

(continues on next page)

(continued from previous page)

```
>>> l1.is_perpendicular(l3)
False
>>> from sympy import Point3D, Line3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(1, 1, 1), Point3D(-1, 2, 0)
>>> l1, l2 = Line3D(p1, p2), Line3D(p2, p3)
>>> l1.is_perpendicular(l2)
False
>>> p4 = Point3D(5, 3, 7)
>>> l3 = Line3D(p1, p4)
>>> l1.is_perpendicular(l3)
False
```

**See also:**[\*coefficients\*](#) (page 2239)**is\_similar(*other*)**

Return True if self and other are contained in the same line.

**Examples**

```
>>> from sympy import Point, Line
>>> p1, p2, p3 = Point(0, 1), Point(3, 4), Point(2, 3)
>>> l1 = Line(p1, p2)
>>> l2 = Line(p1, p3)
>>> l1.is_similar(l2)
True
```

**property length**

The length of the line.

**Examples**

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(3, 5)
>>> l1 = Line(p1, p2)
>>> l1.length
00
```

**property p1**

The first defining point of a linear entity.

## Examples

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> l = Line(p1, p2)
>>> l.p1
Point2D(0, 0)
```

### See also:

[\*sympy.geometry.point.Point\*](#) (page 2202)

### property p2

The second defining point of a linear entity.

## Examples

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> l = Line(p1, p2)
>>> l.p2
Point2D(5, 3)
```

### See also:

[\*sympy.geometry.point.Point\*](#) (page 2202)

### parallel\_line(p)

Create a new Line parallel to this linear entity which passes through the point  $p$ .

#### Parameters

**p** : Point

#### Returns

**line** : Line

## Examples

```
>>> from sympy import Point, Line
>>> p1, p2, p3 = Point(0, 0), Point(2, 3), Point(-2, 2)
>>> l1 = Line(p1, p2)
>>> l2 = l1.parallel_line(p3)
>>> p3 in l2
True
>>> l1.is_parallel(l2)
True
>>> from sympy import Point3D, Line3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(2, 3, 4), Point3D(-2, 2, 0)
>>> l1 = Line3D(p1, p2)
>>> l2 = l1.parallel_line(p3)
>>> p3 in l2
True
```

(continues on next page)

(continued from previous page)

```
>>> l1.is_parallel(l2)
True
```

**See also:**[\*is\\_parallel\*](#) (page 2220)**perpendicular\_line(*p*)**

Create a new Line perpendicular to this linear entity which passes through the point *p*.

**Parameters****p** : Point**Returns****line** : Line**Examples**

```
>>> from sympy import Point3D, Line3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(2, 3, 4), Point3D(-2, 2, 0)
>>> L = Line3D(p1, p2)
>>> P = L.perpendicular_line(p3); P
Line3D(Point3D(-2, 2, 0), Point3D(4/29, 6/29, 8/29))
>>> L.is_perpendicular(P)
True
```

In 3D the, the first point used to define the line is the point through which the perpendicular was required to pass; the second point is (arbitrarily) contained in the given line:

```
>>> P.p2 in L
True
```

**See also:**[\*sympy.geometry.line.LinearEntity.is\\_perpendicular\\_perpendicular\\_segment\*](#) (page 2221),  
(page 2224)**perpendicular\_segment(*p*)**

Create a perpendicular line segment from *p* to this line.

The endpoints of the segment are *p* and the closest point in the line containing self. (If self is not a line, the point might not be in self.)

**Parameters****p** : Point**Returns****segment** : Segment



## Notes

Returns  $p$  itself if  $p$  is on this linear entity.

## Examples

```

>>> from sympy import Point, Line
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(0, 2)
>>> l1 = Line(p1, p2)
>>> s1 = l1.perpendicular_segment(p3)
>>> l1.is_perpendicular(s1)
True
>>> p3 in s1
True
>>> l1.perpendicular_segment(Point(4, 0))
Segment2D(Point2D(4, 0), Point2D(2, 2))
>>> from sympy import Point3D, Line3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(1, 1, 1), Point3D(0, 2, 0)
>>> l1 = Line3D(p1, p2)
>>> s1 = l1.perpendicular_segment(p3)
>>> l1.is_perpendicular(s1)
True
>>> p3 in s1
True
>>> l1.perpendicular_segment(Point3D(4, 0, 0))
Segment3D(Point3D(4, 0, 0), Point3D(4/3, 4/3, 4/3))

```

### See also:

[\*perpendicular\\_line\*](#) (page 2224)

### property points

The two points used to define this linear entity.

#### Returns

**points** : tuple of Points

## Examples

```

>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(5, 11)
>>> l1 = Line(p1, p2)
>>> l1.points
(Point2D(0, 0), Point2D(5, 11))

```

### See also:

[\*sympy.geometry.point.Point\*](#) (page 2202)

### projection(*other*)

Project a point, line, ray, or segment onto this linear entity.

#### Parameters

**other** : Point or LinearEntity (Line, Ray, Segment)

**Returns****projection** : Point or LinearEntity (Line, Ray, Segment)The return type matches the type of the parameter `other`.**Raises****GeometryError**

When method is unable to perform projection.

**Notes**

A projection involves taking the two points that define the linear entity and projecting those points onto a Line and then reforming the linear entity using these projections. A point  $P$  is projected onto a line  $L$  by finding the point on  $L$  that is closest to  $P$ . This point is the intersection of  $L$  and the line perpendicular to  $L$  that passes through  $P$ .

**Examples**

```
>>> from sympy import Point, Line, Segment, Rational
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(Rational(1, 2), 0)
>>> l1 = Line(p1, p2)
>>> l1.projection(p3)
Point2D(1/4, 1/4)
>>> p4, p5 = Point(10, 0), Point(12, 1)
>>> s1 = Segment(p4, p5)
>>> l1.projection(s1)
Segment2D(Point2D(5, 5), Point2D(13/2, 13/2))
>>> p1, p2, p3 = Point(0, 0, 1), Point(1, 1, 2), Point(2, 0, 1)
>>> l1 = Line(p1, p2)
>>> l1.projection(p3)
Point3D(2/3, 2/3, 5/3)
>>> p4, p5 = Point(10, 0, 1), Point(12, 1, 3)
>>> s1 = Segment(p4, p5)
>>> l1.projection(s1)
Segment3D(Point3D(10/3, 10/3, 13/3), Point3D(5, 5, 6))
```

**See also:**[sympy.geometry.point.Point](#) (page 2202), [perpendicular\\_line](#) (page 2224)**random\_point**(*seed=None*)

A random point on a LinearEntity.

**Returns****point** : Point

## Examples

```
>>> from sympy import Point, Line, Ray, Segment
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> line = Line(p1, p2)
>>> r = line.random_point(seed=42) # seed value is optional
>>> r.n(3)
Point2D(-0.72, -0.432)
>>> r in line
True
>>> Ray(p1, p2).random_point(seed=42).n(3)
Point2D(0.72, 0.432)
>>> Segment(p1, p2).random_point(seed=42).n(3)
Point2D(3.2, 1.92)
```

### See also:

[\*sympy.geometry.point.Point\*](#) (page 2202)

### `smallest_angle_between(l2)`

Return the smallest angle formed at the intersection of the lines containing the linear entities.

#### Parameters

**l1** : LinearEntity

**l2** : LinearEntity

#### Returns

**angle** : angle in radians

## Examples

```
>>> from sympy import Point, Line
>>> p1, p2, p3 = Point(0, 0), Point(0, 4), Point(2, -2)
>>> l1, l2 = Line(p1, p2), Line(p1, p3)
>>> l1.smallest_angle_between(l2)
pi/4
```

### See also:

[\*angle\\_between\*](#) (page 2216), [\*Ray2D.closing\\_angle\*](#) (page 2240)

**class** `sympy.geometry.line.Line(*args, **kwargs)`

An infinite line in space.

A 2D line is declared with two distinct points, point and slope, or an equation. A 3D line may be defined with a point and a direction ratio.

#### Parameters

**p1** : Point

**p2** : Point

**slope** : SymPy expression

**direction\_ratio** : list

**equation** : equation of a line

## Notes

*Line* will automatically subclass to *Line2D* or *Line3D* based on the dimension of *p1*. The *slope* argument is only relevant for *Line2D* and the *direction\_ratio* argument is only relevant for *Line3D*.

The order of the points will define the direction of the line which is used when calculating the angle between lines.

## Examples

```
>>> from sympy import Line, Segment, Point, Eq
>>> from sympy.abc import x, y, a, b
```

```
>>> L = Line(Point(2,3), Point(3,5))
>>> L
Line2D(Point2D(2, 3), Point2D(3, 5))
>>> L.points
(Point2D(2, 3), Point2D(3, 5))
>>> L.equation()
-2*x + y + 1
>>> L.coefficients
(-2, 1, 1)
```

Instantiate with keyword slope:

```
>>> Line(Point(0, 0), slope=0)
Line2D(Point2D(0, 0), Point2D(1, 0))
```

Instantiate with another linear object

```
>>> s = Segment((0, 0), (0, 1))
>>> Line(s).equation()
x
```

The line corresponding to an equation in the form  $ax + by + c = 0$ , can be entered:

```
>>> Line(3*x + y + 18)
Line2D(Point2D(0, -18), Point2D(1, -21))
```

If *x* or *y* has a different name, then they can be specified, too, as a string (to match the name) or symbol:

```
>>> Line(Eq(3*a + b, -18), x='a', y=b)
Line2D(Point2D(0, -18), Point2D(1, -21))
```

## See also:

[sympy.geometry.point.Point](#) (page 2202), [sympy.geometry.line.Line2D](#) (page 2238), [sympy.geometry.line.Line3D](#) (page 2243)

**contains(*other*)**

Return True if *other* is on this Line, or False otherwise.

**Examples**

```
>>> from sympy import Line, Point
>>> p1, p2 = Point(0, 1), Point(3, 4)
>>> l = Line(p1, p2)
>>> l.contains(p1)
True
>>> l.contains((0, 1))
True
>>> l.contains((0, 0))
False
>>> a = (0, 0, 0)
>>> b = (1, 1, 1)
>>> c = (2, 2, 2)
>>> l1 = Line(a, b)
>>> l2 = Line(b, a)
>>> l1 == l2
False
>>> l1 in l2
True
```

**distance(*other*)**

Finds the shortest distance between a line and a point.

**Raises**

**NotImplementedError** is raised if ``other`` is not a Point

**Examples**

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(1, 1)
>>> s = Line(p1, p2)
>>> s.distance(Point(-1, 1))
sqrt(2)
>>> s.distance((-1, 2))
3*sqrt(2)/2
>>> p1, p2 = Point(0, 0, 0), Point(1, 1, 1)
>>> s = Line(p1, p2)
>>> s.distance(Point(-1, 1, 1))
2*sqrt(6)/3
>>> s.distance((-1, 1, 1))
2*sqrt(6)/3
```

**equals(*other*)**

Returns True if self and other are the same mathematical entities

**plot\_interval(*parameter*='t')**

The plot interval for the default geometric plot of line. Gives values that will produce

a line that is +/- 5 units long (where a unit is the distance between the two points that define the line).

**Parameters**

**parameter** : str, optional

Default value is 't'.

**Returns**

**plot\_interval** : list (plot interval)

[parameter, lower\_bound, upper\_bound]

**Examples**

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> l1 = Line(p1, p2)
>>> l1.plot_interval()
[t, -5, 5]
```

**class** sympy.geometry.line.Ray(*p1*, *p2*=None, **\*\*kwargs**)

A Ray is a semi-line in the space with a source point and a direction.

**Parameters**

**p1** : Point

The source of the Ray

**p2** : Point or radian value

This point determines the direction in which the Ray propagates. If given as an angle it is interpreted in radians with the positive direction being ccw.

**Notes**

*Ray* will automatically subclass to *Ray2D* or *Ray3D* based on the dimension of *p1*.

**Examples**

```
>>> from sympy import Ray, Point, pi
>>> r = Ray(Point(2, 3), Point(3, 5))
>>> r
Ray2D(Point2D(2, 3), Point2D(3, 5))
>>> r.points
(Point2D(2, 3), Point2D(3, 5))
>>> r.source
Point2D(2, 3)
>>> r.xdirection
00
>>> r.ydirection
00
```

(continues on next page)

(continued from previous page)

```
>>> r.slope
2
>>> Ray(Point(0, 0), angle=pi/4).slope
1
```

**See also:**

[\*sympy.geometry.line.Ray2D\*](#) (page 2240), [\*sympy.geometry.line.Ray3D\*](#) (page 2244), [\*sympy.geometry.point.Point\*](#) (page 2202), [\*sympy.geometry.line.Line\*](#) (page 2227)

**Attributes**

|        |  |
|--------|--|
| source |  |
|--------|--|

**contains(*other*)**

Is other GeometryEntity contained in this Ray?

**Examples**

```
>>> from sympy import Ray, Point, Segment
>>> p1, p2 = Point(0, 0), Point(4, 4)
>>> r = Ray(p1, p2)
>>> r.contains(p1)
True
>>> r.contains((1, 1))
True
>>> r.contains((1, 3))
False
>>> s = Segment((1, 1), (2, 2))
>>> r.contains(s)
True
>>> s = Segment((1, 2), (2, 5))
>>> r.contains(s)
False
>>> r1 = Ray((2, 2), (3, 3))
>>> r.contains(r1)
True
>>> r1 = Ray((2, 2), (3, 5))
>>> r.contains(r1)
False
```

**distance(*other*)**

Finds the shortest distance between the ray and a point.

**Raises**

**NotImplementedError** is raised if ``other`` is not a **Point**

### Examples

```
>>> from sympy import Point, Ray
>>> p1, p2 = Point(0, 0), Point(1, 1)
>>> s = Ray(p1, p2)
>>> s.distance(Point(-1, -1))
sqrt(2)
>>> s.distance((-1, 2))
3*sqrt(2)/2
>>> p1, p2 = Point(0, 0, 0), Point(1, 1, 2)
>>> s = Ray(p1, p2)
>>> s
Ray3D(Point3D(0, 0, 0), Point3D(1, 1, 2))
>>> s.distance(Point(-1, -1, 2))
4*sqrt(3)/3
>>> s.distance((-1, -1, 2))
4*sqrt(3)/3
```

#### **equals**(*other*)

Returns True if self and other are the same mathematical entities

#### **plot\_interval**(*parameter*='t')

The plot interval for the default geometric plot of the Ray. Gives values that will produce a ray that is 10 units long (where a unit is the distance between the two points that define the ray).

##### **Parameters**

**parameter** : str, optional

Default value is 't'.

##### **Returns**

**plot\_interval** : list

[parameter, lower\_bound, upper\_bound]

### Examples

```
>>> from sympy import Ray, pi
>>> r = Ray((0, 0), angle=pi/4)
>>> r.plot_interval()
[t, 0, 10]
```

#### **property source**

The point from which the ray emanates.



## Examples

```
>>> from sympy import Point, Ray
>>> p1, p2 = Point(0, 0), Point(4, 1)
>>> r1 = Ray(p1, p2)
>>> r1.source
Point2D(0, 0)
>>> p1, p2 = Point(0, 0, 0), Point(4, 1, 5)
>>> r1 = Ray(p2, p1)
>>> r1.source
Point3D(4, 1, 5)
```

### See also:

[\*sympy.geometry.point.Point\*](#) (page 2202)

**class** sympy.geometry.line.Segment(p1, p2, \*\*kwargs)

A line segment in space.

### Parameters

**p1** : Point

**p2** : Point

## Notes

If 2D or 3D points are used to define *Segment*, it will be automatically subclassed to *Segment2D* or *Segment3D*.

## Examples

```
>>> from sympy import Point, Segment
>>> Segment((1, 0), (1, 1)) # tuples are interpreted as pts
Segment2D(Point2D(1, 0), Point2D(1, 1))
>>> s = Segment(Point(4, 3), Point(1, 1))
>>> s.points
(Point2D(4, 3), Point2D(1, 1))
>>> s.slope
2/3
>>> s.length
sqrt(13)
>>> s.midpoint
Point2D(5/2, 2)
>>> Segment((1, 0, 0), (1, 1, 1)) # tuples are interpreted as pts
Segment3D(Point3D(1, 0, 0), Point3D(1, 1, 1))
>>> s = Segment(Point(4, 3, 9), Point(1, 1, 7)); s
Segment3D(Point3D(4, 3, 9), Point3D(1, 1, 7))
>>> s.points
(Point3D(4, 3, 9), Point3D(1, 1, 7))
>>> s.length
sqrt(17)
>>> s.midpoint
Point3D(5/2, 2, 8)
```

**See also:**

[`sympy.geometry.line.Segment2D`](#) (page 2242), [`sympy.geometry.line.Segment3D`](#) (page 2247), [`sympy.geometry.point.Point`](#) (page 2202), [`sympy.geometry.line.Line`](#) (page 2227)

**Attributes**

|          |                              |
|----------|------------------------------|
| length   | (number or SymPy expression) |
| midpoint | (Point)                      |

**contains(*other*)**

Is the other GeometryEntity contained within this Segment?

**Examples**

```
>>> from sympy import Point, Segment
>>> p1, p2 = Point(0, 1), Point(3, 4)
>>> s = Segment(p1, p2)
>>> s2 = Segment(p2, p1)
>>> s.contains(s2)
True
>>> from sympy import Point3D, Segment3D
>>> p1, p2 = Point3D(0, 1, 1), Point3D(3, 4, 5)
>>> s = Segment3D(p1, p2)
>>> s2 = Segment3D(p2, p1)
>>> s.contains(s2)
True
>>> s.contains((p1 + p2)/2)
True
```

**distance(*other*)**

Finds the shortest distance between a line segment and a point.

**Raises**

**NotImplementedError** is raised if ``other`` is not a `Point`

**Examples**

```
>>> from sympy import Point, Segment
>>> p1, p2 = Point(0, 1), Point(3, 4)
>>> s = Segment(p1, p2)
>>> s.distance(Point(10, 15))
sqrt(170)
>>> s.distance((0, 12))
sqrt(73)
>>> from sympy import Point3D, Segment3D
>>> p1, p2 = Point3D(0, 0, 3), Point3D(1, 1, 4)
>>> s = Segment3D(p1, p2)
```

(continues on next page)

(continued from previous page)

```
>>> s.distance(Point3D(10, 15, 12))
sqrt(341)
>>> s.distance((10, 15, 12))
sqrt(341)
```

**equals**(*other*)

Returns True if self and other are the same mathematical entities

**property length**

The length of the line segment.

**Examples**

```
>>> from sympy import Point, Segment
>>> p1, p2 = Point(0, 0), Point(4, 3)
>>> s1 = Segment(p1, p2)
>>> s1.length
5
>>> from sympy import Point3D, Segment3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(4, 3, 3)
>>> s1 = Segment3D(p1, p2)
>>> s1.length
sqrt(34)
```

**See also:**

[\*sympy.geometry.point.Point.distance\*](#) (page 2204)

**property midpoint**

The midpoint of the line segment.

**Examples**

```
>>> from sympy import Point, Segment
>>> p1, p2 = Point(0, 0), Point(4, 3)
>>> s1 = Segment(p1, p2)
>>> s1.midpoint
Point2D(2, 3/2)
>>> from sympy import Point3D, Segment3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(4, 3, 3)
>>> s1 = Segment3D(p1, p2)
>>> s1.midpoint
Point3D(2, 3/2, 3/2)
```

**See also:**

[\*sympy.geometry.point.Point.midpoint\*](#) (page 2207)

**perpendicular\_bisector**(*p=None*)

The perpendicular bisector of this segment.

If no point is specified or the point specified is not on the bisector then the bisector is returned as a Line. Otherwise a Segment is returned that joins the point specified and the intersection of the bisector and the segment.

**Parameters**

**p** : Point

**Returns**

**bisector** : Line or Segment

**Examples**

```
>>> from sympy import Point, Segment
>>> p1, p2, p3 = Point(0, 0), Point(6, 6), Point(5, 1)
>>> s1 = Segment(p1, p2)
>>> s1.perpendicular_bisector()
Line2D(Point2D(3, 3), Point2D(-3, 9))
```

```
>>> s1.perpendicular_bisector(p3)
Segment2D(Point2D(5, 1), Point2D(3, 3))
```

**See also:**

[\*LinearEntity.perpendicular\\_segment\*](#) (page 2224)

**plot\_interval**(parameter='t')

The plot interval for the default geometric plot of the Segment gives values that will produce the full segment in a plot.

**Parameters**

**parameter** : str, optional

Default value is 't'.

**Returns**

**plot\_interval** : list

[parameter, lower\_bound, upper\_bound]

**Examples**

```
>>> from sympy import Point, Segment
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> s1 = Segment(p1, p2)
>>> s1.plot_interval()
[t, 0, 1]
```

**class** sympy.geometry.line.**LinearEntity2D**(p1, p2=None, \*\*kwargs)

A base class for all linear entities (line, ray and segment) in a 2-dimensional Euclidean space.

## Notes

This is an abstract class and is not meant to be instantiated.

### See also:

[\*sympy.geometry.entity.GeometryEntity\*](#) (page 2194)

## Attributes

|              |  |
|--------------|--|
| p1           |  |
| p2           |  |
| coefficients |  |
| slope        |  |
| points       |  |

### property bounds

Return a tuple (xmin, ymin, xmax, ymax) representing the bounding rectangle for the geometric figure.

### perpendicular\_line(*p*)

Create a new Line perpendicular to this linear entity which passes through the point *p*.

#### Parameters

**p** : Point

#### Returns

**line** : Line

## Examples

```
>>> from sympy import Point, Line
>>> p1, p2, p3 = Point(0, 0), Point(2, 3), Point(-2, 2)
>>> L = Line(p1, p2)
>>> P = L.perpendicular_line(p3); P
Line2D(Point2D(-2, 2), Point2D(-5, 4))
>>> L.is_perpendicular(P)
True
```

In 2D, the first point of the perpendicular line is the point through which was required to pass; the second point is arbitrarily chosen. To get a line that explicitly uses a point in the line, create a line from the perpendicular segment from the line to the point:

```
>>> Line(L.perpendicular_segment(p3))
Line2D(Point2D(-2, 2), Point2D(4/13, 6/13))
```

### See also:

[\*sympy.geometry.line.LinearEntity.is\\_perpendicular\*](#) (page 2221),  
[\*perpendicular\\_segment\*](#) (page 2224)

**property slope**

The slope of this linear entity, or infinity if vertical.

**Returns**

**slope** : number or SymPy expression

**Examples**

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(3, 5)
>>> l1 = Line(p1, p2)
>>> l1.slope
5/3
```

```
>>> p3 = Point(0, 4)
>>> l2 = Line(p1, p3)
>>> l2.slope
oo
```

**See also:**

[\*coefficients\*](#) (page 2239)

**class** sympy.geometry.line.Line2D(*p1*, *pt=None*, *slope=None*, *\*\*kwargs*)

An infinite line in space 2D.

A line is declared with two distinct points or a point and slope as defined using keyword *slope*.

**Parameters**

**p1** : Point

**pt** : Point

**slope** : SymPy expression

**Examples**

```
>>> from sympy import Line, Segment, Point
>>> L = Line(Point(2,3), Point(3,5))
>>> L
Line2D(Point2D(2, 3), Point2D(3, 5))
>>> L.points
(Point2D(2, 3), Point2D(3, 5))
>>> L.equation()
-2*x + y + 1
>>> L.coefficients
(-2, 1, 1)
```

Instantiate with keyword slope:

```
>>> Line(Point(0, 0), slope=0)
Line2D(Point2D(0, 0), Point2D(1, 0))
```

Instantiate with another linear object

```
>>> s = Segment((0, 0), (0, 1))
>>> Line(s).equation()
x
```

**See also:**

[\*sympy.geometry.point.Point\*](#) (page 2202)

**property coefficients**

The coefficients  $(a, b, c)$  for  $ax + by + c = 0$ .

### Examples

```
>>> from sympy import Point, Line
>>> from sympy.abc import x, y
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> l = Line(p1, p2)
>>> l.coefficients
(-3, 5, 0)
```

```
>>> p3 = Point(x, y)
>>> l2 = Line(p1, p3)
>>> l2.coefficients
(-y, x, 0)
```

**See also:**

[\*sympy.geometry.line.Line2D.equation\*](#) (page 2239)

**equation**( $x='x', y='y'$ )

The equation of the line:  $ax + by + c$ .

**Parameters**

**x** : str, optional

The name to use for the x-axis, default value is 'x'.

**y** : str, optional

The name to use for the y-axis, default value is 'y'.

**Returns**

**equation** : SymPy expression

### Examples

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(1, 0), Point(5, 3)
>>> l1 = Line(p1, p2)
>>> l1.equation()
-3*x + 4*y + 3
```

**See also:**

[\*sympy.geometry.line.Line2D.coefficients\*](#) (page 2239)

**class** `sympy.geometry.line.Ray2D(p1, pt=None, angle=None, **kwargs)`

A Ray is a semi-line in the space with a source point and a direction.

**Parameters**

**p1** : Point

The source of the Ray

**p2** : Point or radian value

This point determines the direction in which the Ray propagates. If given as an angle it is interpreted in radians with the positive direction being ccw.

**Examples**

```
>>> from sympy import Point, pi, Ray
>>> r = Ray(Point(2, 3), Point(3, 5))
>>> r
Ray2D(Point2D(2, 3), Point2D(3, 5))
>>> r.points
(Point2D(2, 3), Point2D(3, 5))
>>> r.source
Point2D(2, 3)
>>> r.xdirection
00
>>> r.ydirection
00
>>> r.slope
2
>>> Ray(Point(0, 0), angle=pi/4).slope
1
```

**See also:**

[\*sympy.geometry.point.Point\*](#) (page 2202), [\*Line\*](#) (page 2227)

**Attributes**

|            |  |
|------------|--|
| source     |  |
| xdirection |  |
| ydirection |  |

**closing\_angle(*r2*)**

Return the angle by which *r2* must be rotated so it faces the same direction as *r1*.

**Parameters**

**r1** : Ray2D

**r2** : Ray2D



**Returns****angle** : angle in radians (ccw angle is positive)**Examples**

```
>>> from sympy import Ray, pi
>>> r1 = Ray((0, 0), (1, 0))
>>> r2 = r1.rotate(-pi/2)
>>> angle = r1.closing_angle(r2); angle
pi/2
>>> r2.rotate(angle).direction.unit == r1.direction.unit
True
>>> r2.closing_angle(r1)
-pi/2
```

**See also:**[\*LinearEntity.angle\\_between\*](#) (page 2216)**property xdirection**

The x direction of the ray.

Positive infinity if the ray points in the positive x direction, negative infinity if the ray points in the negative x direction, or 0 if the ray is vertical.

**Examples**

```
>>> from sympy import Point, Ray
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(0, -1)
>>> r1, r2 = Ray(p1, p2), Ray(p1, p3)
>>> r1.xdirection
oo
>>> r2.xdirection
0
```

**See also:**[\*ydirection\*](#) (page 2241)**property ydirection**

The y direction of the ray.

Positive infinity if the ray points in the positive y direction, negative infinity if the ray points in the negative y direction, or 0 if the ray is horizontal.

### Examples

```
>>> from sympy import Point, Ray
>>> p1, p2, p3 = Point(0, 0), Point(-1, -1), Point(-1, 0)
>>> r1, r2 = Ray(p1, p2), Ray(p1, p3)
>>> r1.ydirection
-oo
>>> r2.ydirection
0
```

#### See also:

[\*xdirection\*](#) (page 2241)

**class** sympy.geometry.line.Segment2D(*p1*, *p2*, **\*\*kwargs**)

A line segment in 2D space.

#### Parameters

**p1** : Point

**p2** : Point

### Examples

```
>>> from sympy import Point, Segment
>>> Segment((1, 0), (1, 1)) # tuples are interpreted as pts
Segment2D(Point2D(1, 0), Point2D(1, 1))
>>> s = Segment(Point(4, 3), Point(1, 1)); s
Segment2D(Point2D(4, 3), Point2D(1, 1))
>>> s.points
(Point2D(4, 3), Point2D(1, 1))
>>> s.slope
2/3
>>> s.length
sqrt(13)
>>> s.midpoint
Point2D(5/2, 2)
```

#### See also:

[\*sympy.geometry.point.Point\*](#) (page 2202), [\*Line\*](#) (page 2227)

### Attributes

|          |                              |
|----------|------------------------------|
| length   | (number or SymPy expression) |
| midpoint | (Point)                      |

**class** sympy.geometry.line.LinearEntity3D(*p1*, *p2*, **\*\*kwargs**)

An base class for all linear entities (line, ray and segment) in a 3-dimensional Euclidean space.

## Notes

This is a base class and is not meant to be instantiated.

## Attributes

|                  |  |
|------------------|--|
| p1               |  |
| p2               |  |
| direction_ratio  |  |
| direction_cosine |  |
| points           |  |

### property direction\_cosine

The normalized direction ratio of a given line in 3D.

## Examples

```
>>> from sympy import Point3D, Line3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(5, 3, 1)
>>> l = Line3D(p1, p2)
>>> l.direction_cosine
[sqrt(35)/7, 3*sqrt(35)/35, sqrt(35)/35]
>>> sum(i**2 for i in _)
1
```

### See also:

[\*sympy.geometry.line.Line3D.equation\*](#) (page 2244)

### property direction\_ratio

The direction ratio of a given line in 3D.

## Examples

```
>>> from sympy import Point3D, Line3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(5, 3, 1)
>>> l = Line3D(p1, p2)
>>> l.direction_ratio
[5, 3, 1]
```

### See also:

[\*sympy.geometry.line.Line3D.equation\*](#) (page 2244)

**class** sympy.geometry.line.**Line3D**(p1, pt=None, direction\_ratio=(), \*\*kwargs)

An infinite 3D line in space.

A line is declared with two distinct points or a point and `direction_ratio` as defined using keyword *direction\_ratio*.

**Parameters****p1** : Point3D**pt** : Point3D**direction\_ratio** : list**Examples**

```
>>> from sympy import Line3D, Point3D
>>> L = Line3D(Point3D(2, 3, 4), Point3D(3, 5, 1))
>>> L
Line3D(Point3D(2, 3, 4), Point3D(3, 5, 1))
>>> L.points
(Point3D(2, 3, 4), Point3D(3, 5, 1))
```

**See also:**

[\*sympy.geometry.point.Point3D\*](#) (page 2211), [\*sympy.geometry.line.Line\*](#) (page 2227), [\*sympy.geometry.line.Line2D\*](#) (page 2238)

**equation**(*x*='x', *y*='y', *z*='z', *k*=None)

Return the equations that define the line in 3D.

**Parameters****x** : str, optional

The name to use for the x-axis, default value is 'x'.

**y** : str, optional

The name to use for the y-axis, default value is 'y'.

**z** : str, optional

The name to use for the z-axis, default value is 'z'.

**k** : str, optional

Deprecated since version 1.2: The k flag is deprecated. It does nothing.

**Returns****equation** : Tuple of simultaneous equations**Examples**

```
>>> from sympy import Point3D, Line3D, solve
>>> from sympy.abc import x, y, z
>>> p1, p2 = Point3D(1, 0, 0), Point3D(5, 3, 0)
>>> l1 = Line3D(p1, p2)
>>> eq = l1.equation(x, y, z); eq
(-3*x + 4*y + 3, z)
>>> solve(eq.subs(z, 0), (x, y, z))
{x: 4*y/3 + 1}
```

**class** sympy.geometry.line.Ray3D(*p1*, *pt=None*, *direction\_ratio=()*, *\*\*kwargs*)

A Ray is a semi-line in the space with a source point and a direction.

#### Parameters

**p1** : Point3D

The source of the Ray

**p2** : Point or a direction vector

**direction\_ratio**: Determines the direction in which the Ray propagates.

#### Examples

```
>>> from sympy import Point3D, Ray3D
>>> r = Ray3D(Point3D(2, 3, 4), Point3D(3, 5, 0))
>>> r
Ray3D(Point3D(2, 3, 4), Point3D(3, 5, 0))
>>> r.points
(Point3D(2, 3, 4), Point3D(3, 5, 0))
>>> r.source
Point3D(2, 3, 4)
>>> r.xdirection
00
>>> r.ydirection
00
>>> r.direction_ratio
[1, 2, -4]
```

#### See also:

[sympy.geometry.point.Point3D](#) (page 2211), [Line3D](#) (page 2243)

#### Attributes

|            |  |
|------------|--|
| source     |  |
| xdirection |  |
| ydirection |  |
| zdirection |  |

#### property xdirection

The x direction of the ray.

Positive infinity if the ray points in the positive x direction, negative infinity if the ray points in the negative x direction, or 0 if the ray is vertical.

## Examples

```
>>> from sympy import Point3D, Ray3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(1, 1, 1), Point3D(0, -1, 0)
>>> r1, r2 = Ray3D(p1, p2), Ray3D(p1, p3)
>>> r1.xdirection
oo
>>> r2.xdirection
0
```

### See also:

[\*ydirection\*](#) (page 2246)

### property **ydirection**

The y direction of the ray.

Positive infinity if the ray points in the positive y direction, negative infinity if the ray points in the negative y direction, or 0 if the ray is horizontal.

## Examples

```
>>> from sympy import Point3D, Ray3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(-1, -1, -1), Point3D(-1, 0,
→ 0)
>>> r1, r2 = Ray3D(p1, p2), Ray3D(p1, p3)
>>> r1.ydirection
-oo
>>> r2.ydirection
0
```

### See also:

[\*xdirection\*](#) (page 2245)

### property **zdirection**

The z direction of the ray.

Positive infinity if the ray points in the positive z direction, negative infinity if the ray points in the negative z direction, or 0 if the ray is horizontal.

## Examples

```
>>> from sympy import Point3D, Ray3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(-1, -1, -1), Point3D(-1, 0,
→ 0)
>>> r1, r2 = Ray3D(p1, p2), Ray3D(p1, p3)
>>> r1.ydirection
-oo
>>> r2.ydirection
0
>>> r2.zdirection
0
```

**See also:**[xdirection](#) (page 2245)**class** sympy.geometry.line.Segment3D(*p1*, *p2*, **\*\*kwargs**)

A line segment in a 3D space.

**Parameters****p1** : Point3D**p2** : Point3D**Examples**

```

>>> from sympy import Point3D, Segment3D
>>> Segment3D((1, 0, 0), (1, 1, 1)) # tuples are interpreted as pts
Segment3D(Point3D(1, 0, 0), Point3D(1, 1, 1))
>>> s = Segment3D(Point3D(4, 3, 9), Point3D(1, 1, 7)); s
Segment3D(Point3D(4, 3, 9), Point3D(1, 1, 7))
>>> s.points
(Point3D(4, 3, 9), Point3D(1, 1, 7))
>>> s.length
sqrt(17)
>>> s.midpoint
Point3D(5/2, 2, 8)

```

**See also:**[sympy.geometry.point.Point3D](#) (page 2211), [Line3D](#) (page 2243)**Attributes**

|          |                              |
|----------|------------------------------|
| length   | (number or SymPy expression) |
| midpoint | (Point3D)                    |

**Curves****class** sympy.geometry.curve.Curve(*function*, *limits*)

A curve in space.

A curve is defined by parametric functions for the coordinates, a parameter and the lower and upper bounds for the parameter value.

**Parameters****function** : list of functions**limits** : 3-tuple

Function parameter and lower and upper bounds.

**Raises****ValueError**

When *functions* are specified incorrectly. When *limits* are specified incorrectly.

### Examples

```
>>> from sympy import Curve, sin, cos, interpolate
>>> from sympy.abc import t, a
>>> C = Curve((sin(t), cos(t)), (t, 0, 2))
>>> C.functions
(sin(t), cos(t))
>>> C.limits
(t, 0, 2)
>>> C.parameter
t
>>> C = Curve((t, interpolate([1, 4, 9, 16], t)), (t, 0, 1)); C
Curve((t, t**2), (t, 0, 1))
>>> C.subs(t, 4)
Point2D(4, 16)
>>> C.arbitrary_point(a)
Point2D(a, a**2)
```

### See also:

[sympy.core.function.Function](#) (page 1050), [sympy.polys.polyfuncs.interpolate](#) (page 2426)

### Attributes

|           |  |
|-----------|--|
| functions |  |
| parameter |  |
| limits    |  |

### property `ambient_dimension`

The dimension of the curve.

#### Returns

int :

the dimension of curve.

### Examples

```
>>> from sympy.abc import t
>>> from sympy import Curve
>>> C = Curve((t, t**2), (t, 0, 2))
>>> C.ambient_dimension
2
```



**arbitrary\_point**(*parameter*='t')

A parameterized point on the curve.

**Parameters**

**parameter** : str or Symbol, optional

Default value is 't'. The Curve's parameter is selected with None or self.parameter otherwise the provided symbol is used.

**Returns**

Point :

Returns a point in parametric form.

**Raises**

**ValueError**

When *parameter* already appears in the functions.

### Examples

```
>>> from sympy import Curve, Symbol
>>> from sympy.abc import s
>>> C = Curve([2*s, s**2], (s, 0, 2))
>>> C.arbitrary_point()
Point2D(2*t, t**2)
>>> C.arbitrary_point(C.parameter)
Point2D(2*s, s**2)
>>> C.arbitrary_point(None)
Point2D(2*s, s**2)
>>> C.arbitrary_point(Symbol('a'))
Point2D(2*a, a**2)
```

**See also:**

[sympy.geometry.point.Point](#) (page 2202)

**property free\_symbols**

Return a set of symbols other than the bound symbols used to parametrically define the Curve.

**Returns**

set :

Set of all non-parameterized symbols.

### Examples

```
>>> from sympy.abc import t, a
>>> from sympy import Curve
>>> Curve((t, t**2), (t, 0, 2)).free_symbols
set()
>>> Curve((t, t**2), (t, a, 2)).free_symbols
{a}
```

**property functions**

The functions specifying the curve.

**Returns**

functions :

list of parameterized coordinate functions.

**Examples**

```
>>> from sympy.abc import t
>>> from sympy import Curve
>>> C = Curve((t, t**2), (t, 0, 2))
>>> C.functions
(t, t**2)
```

**See also:**

[\*parameter\*](#) (page 2250)

**property length**

The curve length.

**Examples**

```
>>> from sympy import Curve
>>> from sympy.abc import t
>>> Curve((t, t), (t, 0, 1)).length
sqrt(2)
```

**property limits**

The limits for the curve.

**Returns**

**limits** : tuple

Contains parameter and lower and upper limits.

**Examples**

```
>>> from sympy.abc import t
>>> from sympy import Curve
>>> C = Curve([t, t**3], (t, -2, 2))
>>> C.limits
(t, -2, 2)
```

**See also:**

[\*plot\\_interval\*](#) (page 2251)

**property parameter**

The curve function variable.

**Returns**

Symbol :

returns a bound symbol.

**Examples**

```
>>> from sympy.abc import t
>>> from sympy import Curve
>>> C = Curve([t, t**2], (t, 0, 2))
>>> C.parameter
t
```

**See also:**

[\*functions\*](#) (page 2249)

**plot\_interval(*parameter*='t')**

The plot interval for the default geometric plot of the curve.

**Parameters**

**parameter** : str or Symbol, optional

Default value is 't'; otherwise the provided symbol is used.

**Returns**

List :

**the plot interval as below:**

[parameter, lower\_bound, upper\_bound]

**Examples**

```
>>> from sympy import Curve, sin
>>> from sympy.abc import x, s
>>> Curve((x, sin(x)), (x, 1, 2)).plot_interval()
[t, 1, 2]
>>> Curve((x, sin(x)), (x, 1, 2)).plot_interval(s)
[s, 1, 2]
```

**See also:**

[\*limits\*](#) (page 2250)

Returns limits of the parameter interval

**rotate(*angle*=0, *pt*=None)**

This function is used to rotate a curve along given point *pt* at given angle(in radian).

**Parameters**

**angle** :

the angle at which the curve will be rotated(in radian) in counter-clockwise direction. default value of angle is 0.

**pt** : Point

the point along which the curve will be rotated. If no point given, the curve will be rotated around origin.

**Returns**

Curve :

returns a curve rotated at given angle along given point.

**Examples**

```
>>> from sympy import Curve, pi
>>> from sympy.abc import x
>>> Curve((x, x), (x, 0, 1)).rotate(pi/2)
Curve((-x, x), (x, 0, 1))
```

**scale**(*x=1, y=1, pt=None*)

Override GeometryEntity.scale since Curve is not made up of Points.

**Returns**

Curve :

returns scaled curve.

**Examples**

```
>>> from sympy import Curve
>>> from sympy.abc import x
>>> Curve((x, x), (x, 0, 1)).scale(2)
Curve((2*x, x), (x, 0, 1))
```

**translate**(*x=0, y=0*)

Translate the Curve by (x, y).

**Returns**

Curve :

returns a translated curve.

**Examples**

```
>>> from sympy import Curve
>>> from sympy.abc import x
>>> Curve((x, x), (x, 0, 1)).translate(1, 2)
Curve((x + 1, x + 2), (x, 0, 1))
```

## Ellipses

**class** sympy.geometry.ellipse.**Ellipse**(*center=None, hradius=None, vradius=None, eccentricity=None, \*\*kwargs*)

An elliptical GeometryEntity.

### Parameters

**center** : Point, optional

Default value is Point(0, 0)

**hradius** : number or SymPy expression, optional

**vradius** : number or SymPy expression, optional

**eccentricity** : number or SymPy expression, optional

Two of *hradius*, *vradius* and *eccentricity* must be supplied to create an Ellipse. The third is derived from the two supplied.

### Raises

#### GeometryError

When *hradius*, *vradius* and *eccentricity* are incorrectly supplied as parameters.

#### TypeError

When *center* is not a Point.

## Notes

Constructed from a center and two radii, the first being the horizontal radius (along the x-axis) and the second being the vertical radius (along the y-axis).

When symbolic value for *hradius* and *vradius* are used, any calculation that refers to the foci or the major or minor axis will assume that the ellipse has its major radius on the x-axis. If this is not true then a manual rotation is necessary.

## Examples

```
>>> from sympy import Ellipse, Point, Rational
>>> e1 = Ellipse(Point(0, 0), 5, 1)
>>> e1.hradius, e1.vradius
(5, 1)
>>> e2 = Ellipse(Point(3, 1), hradius=3, eccentricity=Rational(4, 5))
>>> e2
Ellipse(Point2D(3, 1), 3, 9/5)
```

### See also:

[Circle](#) (page 2269)

## Attributes

|                |  |
|----------------|--|
| center         |  |
| hradius        |  |
| vradius        |  |
| area           |  |
| circumference  |  |
| eccentricity   |  |
| periapsis      |  |
| apoapsis       |  |
| focus_distance |  |
| foci           |  |

### property apoapsis

The apoapsis of the ellipse.

The greatest distance between the focus and the contour.

#### Returns

**apoapsis** : number

## Examples

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.apoapsis
2*sqrt(2) + 3
```

### See also:

#### [\*periapsis\*](#) (page 2263)

Returns shortest distance between foci and contour

### **arbitrary\_point**(*parameter*='t')

A parameterized point on the ellipse.

#### Parameters

**parameter** : str, optional

Default value is 't'.

#### Returns

**arbitrary\_point** : Point

#### Raises

**ValueError**

When *parameter* already appears in the functions.

## Examples

```
>>> from sympy import Point, Ellipse
>>> e1 = Ellipse(Point(0, 0), 3, 2)
>>> e1.arbitrary_point()
Point2D(3*cos(t), 2*sin(t))
```

### See also:

[\*sympy.geometry.point.Point\*](#) (page 2202)

### property area

The area of the ellipse.

#### Returns

**area** : number

## Examples

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.area
3*pi
```

### auxiliary\_circle()

Returns a Circle whose diameter is the major axis of the ellipse.

## Examples

```
>>> from sympy import Ellipse, Point, symbols
>>> c = Point(1, 2)
>>> Ellipse(c, 8, 7).auxiliary_circle()
Circle(Point2D(1, 2), 8)
>>> a, b = symbols('a b')
>>> Ellipse(c, a, b).auxiliary_circle()
Circle(Point2D(1, 2), Max(a, b))
```

### property bounds

Return a tuple (xmin, ymin, xmax, ymax) representing the bounding rectangle for the geometric figure.

### property center

The center of the ellipse.

#### Returns

**center** : number

### Examples

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.center
Point2D(0, 0)
```

#### See also:

[\*sympy.geometry.point.Point\*](#) (page 2202)

#### property circumference

The circumference of the ellipse.

### Examples

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.circumference
12*elliptic_e(8/9)
```

#### director\_circle()

Returns a Circle consisting of all points where two perpendicular tangent lines to the ellipse cross each other.

##### Returns

Circle

A director circle returned as a geometric object.

### Examples

```
>>> from sympy import Ellipse, Point, symbols
>>> c = Point(3,8)
>>> Ellipse(c, 7, 9).director_circle()
Circle(Point2D(3, 8), sqrt(130))
>>> a, b = symbols('a b')
>>> Ellipse(c, a, b).director_circle()
Circle(Point2D(3, 8), sqrt(a**2 + b**2))
```



## References

[R503]

### property `eccentricity`

The eccentricity of the ellipse.

#### Returns

**`eccentricity`** : number

## Examples

```
>>> from sympy import Point, Ellipse, sqrt
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, sqrt(2))
>>> e1.eccentricity
sqrt(7)/3
```

### `encloses_point(p)`

Return True if `p` is enclosed by (is inside of) self.

#### Parameters

**`p`** : Point

#### Returns

**`encloses_point`** : True, False or None

## Notes

Being on the border of self is considered False.

## Examples

```
>>> from sympy import Ellipse, S
>>> from sympy.abc import t
>>> e = Ellipse((0, 0), 3, 2)
>>> e.encloses_point((0, 0))
True
>>> e.encloses_point(e.arbitrary_point(t).subs(t, S.Half))
False
>>> e.encloses_point((4, 0))
False
```

### See also:

[`sympy.geometry.point.Point`](#) (page 2202)

### `equation(x='x', y='y', _slope=None)`

Returns the equation of an ellipse aligned with the `x` and `y` axes; when `slope` is given, the equation returned corresponds to an ellipse with a major axis having that slope.

#### Parameters

**`x`** : str, optional

Label for the x-axis. Default value is 'x'.

**y** : str, optional

Label for the y-axis. Default value is 'y'.

**\_slope** : Expr, optional

The slope of the major axis. Ignored when 'None'.

#### Returns

**equation** : SymPy expression

### Examples

```
>>> from sympy import Point, Ellipse, pi
>>> from sympy.abc import x, y
>>> e1 = Ellipse(Point(1, 0), 3, 2)
>>> eq1 = e1.equation(x, y); eq1
y**2/4 + (x/3 - 1/3)**2 - 1
>>> eq2 = e1.equation(x, y, _slope=1); eq2
(-x + y + 1)**2/8 + (x + y - 1)**2/18 - 1
```

A point on e1 satisfies eq1. Let's use one on the x-axis:

```
>>> p1 = e1.center + Point(e1.major, 0)
>>> assert eq1.subs(x, p1.x).subs(y, p1.y) == 0
```

When rotated the same as the rotated ellipse, about the center point of the ellipse, it will satisfy the rotated ellipse's equation, too:

```
>>> r1 = p1.rotate(pi/4, e1.center)
>>> assert eq2.subs(x, r1.x).subs(y, r1.y) == 0
```

#### See also:

#### [\*arbitrary\\_point\*](#) (page 2254)

Returns parameterized point on ellipse

### References

[R504], [R505]

**evolute**(x='x', y='y')

The equation of evolute of the ellipse.

#### Parameters

**x** : str, optional

Label for the x-axis. Default value is 'x'.

**y** : str, optional

Label for the y-axis. Default value is 'y'.

#### Returns

**equation** : SymPy expression

## Examples

```
>>> from sympy import Point, Ellipse
>>> e1 = Ellipse(Point(1, 0), 3, 2)
>>> e1.evolute()
2**(2/3)*y**(2/3) + (3*x - 3)**(2/3) - 5**(2/3)
```

### property foci

The foci of the ellipse.

#### Raises

##### ValueError

When the major and minor axis cannot be determined.

## Notes

The foci can only be calculated if the major/minor axes are known.

## Examples

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.foci
(Point2D(-2*sqrt(2), 0), Point2D(2*sqrt(2), 0))
```

### See also:

[\*sympy.geometry.point.Point\*](#) (page 2202)

#### ***focus\_distance*** (page 2259)

Returns the distance between focus and center

### property focus\_distance

The focal distance of the ellipse.

The distance between the center and one focus.

#### Returns

**focus\_distance** : number

## Examples

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.focus_distance
2*sqrt(2)
```

### See also:

[\*foci\*](#) (page 2259)

**property hradius**

The horizontal radius of the ellipse.

**Returns**

**hradius** : number

**Examples**

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.hradius
3
```

**See also:**

[\*vradius\*](#) (page 2269), [\*major\*](#) (page 2261), [\*minor\*](#) (page 2262)

**intersection(*o*)**

The intersection of this ellipse and another geometrical entity *o*.

**Parameters**

**o** : GeometryEntity

**Returns**

**intersection** : list of GeometryEntity objects

**Notes**

Currently supports intersections with Point, Line, Segment, Ray, Circle and Ellipse types.

**Examples**

```
>>> from sympy import Ellipse, Point, Line
>>> e = Ellipse(Point(0, 0), 5, 7)
>>> e.intersection(Point(0, 0))
[]
>>> e.intersection(Point(5, 0))
[Point2D(5, 0)]
>>> e.intersection(Line(Point(0,0), Point(0, 1)))
[Point2D(0, -7), Point2D(0, 7)]
>>> e.intersection(Line(Point(5,0), Point(5, 1)))
[Point2D(5, 0)]
>>> e.intersection(Line(Point(6,0), Point(6, 1)))
[]
>>> e = Ellipse(Point(-1, 0), 4, 3)
>>> e.intersection(Ellipse(Point(1, 0), 4, 3))
[Point2D(0, -3*sqrt(15)/4), Point2D(0, 3*sqrt(15)/4)]
>>> e.intersection(Ellipse(Point(5, 0), 4, 3))
[Point2D(2, -3*sqrt(7)/4), Point2D(2, 3*sqrt(7)/4)]
```

(continues on next page)

(continued from previous page)

```

>>> e.intersection(Ellipse(Point(100500, 0), 4, 3))
[]
>>> e.intersection(Ellipse(Point(0, 0), 3, 4))
[Point2D(3, 0), Point2D(-363/175, -48*sqrt(111)/175), Point2D(-363/
↪ 175, 48*sqrt(111)/175)]
>>> e.intersection(Ellipse(Point(-1, 0), 3, 4))
[Point2D(-17/5, -12/5), Point2D(-17/5, 12/5), Point2D(7/5, -12/5),
↪ Point2D(7/5, 12/5)]

```

**See also:**

[\*sympy.geometry.entity.GeometryEntity\*](#) (page 2194)

**is\_tangent(*o*)**

Is *o* tangent to the ellipse?

**Parameters**

***o*** : GeometryEntity

An Ellipse, LinearEntity or Polygon

**Returns**

is\_tangent: boolean

True if *o* is tangent to the ellipse, False otherwise.

**Raises**

**NotImplementedError**

When the wrong type of argument is supplied.

**Examples**

```

>>> from sympy import Point, Ellipse, Line
>>> p0, p1, p2 = Point(0, 0), Point(3, 0), Point(3, 3)
>>> e1 = Ellipse(p0, 3, 2)
>>> l1 = Line(p1, p2)
>>> e1.is_tangent(l1)
True

```

**See also:**

[\*tangent\\_lines\*](#) (page 2268)

**property major**

Longer axis of the ellipse (if it can be determined) else hradius.

**Returns**

**major** : number or expression

### Examples

```
>>> from sympy import Point, Ellipse, Symbol
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.major
3
```

```
>>> a = Symbol('a')
>>> b = Symbol('b')
>>> Ellipse(p1, a, b).major
a
>>> Ellipse(p1, b, a).major
b
```

```
>>> m = Symbol('m')
>>> M = m + 1
>>> Ellipse(p1, m, M).major
m + 1
```

#### See also:

[\*hradius\*](#) (page 2259), [\*vradius\*](#) (page 2269), [\*minor\*](#) (page 2262)

#### property `minor`

Shorter axis of the ellipse (if it can be determined) else `vradius`.

#### Returns

**minor** : number or expression

### Examples

```
>>> from sympy import Point, Ellipse, Symbol
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.minor
1
```

```
>>> a = Symbol('a')
>>> b = Symbol('b')
>>> Ellipse(p1, a, b).minor
b
>>> Ellipse(p1, b, a).minor
a
```

```
>>> m = Symbol('m')
>>> M = m + 1
>>> Ellipse(p1, m, M).minor
m
```

#### See also:

[\*hradius\*](#) (page 2259), [\*vradius\*](#) (page 2269), [\*major\*](#) (page 2261)

**normal\_lines**(*p*, *prec*=None)

Normal lines between *p* and the ellipse.

**Parameters**

**p** : Point

**Returns**

**normal\_lines** : list with 1, 2 or 4 Lines

**Examples**

```
>>> from sympy import Point, Ellipse
>>> e = Ellipse((0, 0), 2, 3)
>>> c = e.center
>>> e.normal_lines(c + Point(1, 0))
[Line2D(Point2D(0, 0), Point2D(1, 0))]
>>> e.normal_lines(c)
[Line2D(Point2D(0, 0), Point2D(0, 1)), Line2D(Point2D(0, 0),
↪Point2D(1, 0))]
```

Off-axis points require the solution of a quartic equation. This often leads to very large expressions that may be of little practical use. An approximate solution of *prec* digits can be obtained by passing in the desired value:

```
>>> e.normal_lines((3, 3), prec=2)
[Line2D(Point2D(-0.81, -2.7), Point2D(0.19, -1.2)),
Line2D(Point2D(1.5, -2.0), Point2D(2.5, -2.7))]
```

Whereas the above solution has an operation count of 12, the exact solution has an operation count of 2020.

**property periapsis**

The periapsis of the ellipse.

The shortest distance between the focus and the contour.

**Returns**

**periapsis** : number

**Examples**

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.periapsis
3 - 2*sqrt(2)
```

**See also:**

**[apoapsis](#) (page 2254)**

Returns greatest distance between focus and contour

**plot\_interval**(*parameter*='t')

The plot interval for the default geometric plot of the Ellipse.

**Parameters**

**parameter** : str, optional

Default value is 't'.

**Returns**

**plot\_interval** : list

[parameter, lower\_bound, upper\_bound]

### Examples

```
>>> from sympy import Point, Ellipse
>>> e1 = Ellipse(Point(0, 0), 3, 2)
>>> e1.plot_interval()
[t, -pi, pi]
```

**polar\_second\_moment\_of\_area**()

Returns the polar second moment of area of an Ellipse

It is a constituent of the second moment of area, linked through the perpendicular axis theorem. While the planar second moment of area describes an object's resistance to deflection (bending) when subjected to a force applied to a plane parallel to the central axis, the polar second moment of area describes an object's resistance to deflection when subjected to a moment applied in a plane perpendicular to the object's central axis (i.e. parallel to the cross-section)

### Examples

```
>>> from sympy import symbols, Circle, Ellipse
>>> c = Circle((5, 5), 4)
>>> c.polar_second_moment_of_area()
128*pi
>>> a, b = symbols('a, b')
>>> e = Ellipse((0, 0), a, b)
>>> e.polar_second_moment_of_area()
pi*a**3*b/4 + pi*a*b**3/4
```

### References

[R506]

**random\_point**(*seed*=None)

A random point on the ellipse.

**Returns**

**point** : Point



## Examples

```
>>> from sympy import Point, Ellipse
>>> e1 = Ellipse(Point(0, 0), 3, 2)
>>> e1.random_point() # gives some random point
Point2D(...)
>>> p1 = e1.random_point(seed=0); p1.n(2)
Point2D(2.1, 1.4)
```

## Notes

When creating a random point, one may simply replace the parameter with a random number. When doing so, however, the random number should be made a Rational or else the point may not test as being in the ellipse:

```
>>> from sympy.abc import t
>>> from sympy import Rational
>>> arb = e1.arbitrary_point(t); arb
Point2D(3*cos(t), 2*sin(t))
>>> arb.subs(t, .1) in e1
False
>>> arb.subs(t, Rational(.1)) in e1
True
>>> arb.subs(t, Rational('.1')) in e1
True
```

## See also:

[sympy.geometry.point.Point](#) (page 2202)

### **`arbitrary_point`** (page 2254)

Returns parameterized point on ellipse

### **`reflect(line)`**

Override GeometryEntity.reflect since the radius is not a GeometryEntity.

## Examples

```
>>> from sympy import Circle, Line
>>> Circle((0, 1), 1).reflect(Line((0, 0), (1, 1)))
Circle(Point2D(1, 0), -1)
>>> from sympy import Ellipse, Line, Point
>>> Ellipse(Point(3, 4), 1, 3).reflect(Line(Point(0, -4), Point(5, 0)))
Traceback (most recent call last):
...
NotImplementedError:
General Ellipse is not supported but the equation of the reflected
Ellipse is given by the zeros of: f(x, y) = (9*x/41 + 40*y/41 +
37/41)**2 + (40*x/123 - 3*y/41 - 364/123)**2 - 1
```

## Notes

Until the general ellipse (with no axis parallel to the x-axis) is supported a `NotImplementedError` is raised and the equation whose zeros define the rotated ellipse is given.

**rotate**(*angle=0*, *pt=None*)

Rotate *angle* radians counterclockwise about Point *pt*.

Note: since the general ellipse is not supported, only rotations that are integer multiples of  $\pi/2$  are allowed.

## Examples

```
>>> from sympy import Ellipse, pi
>>> Ellipse((1, 0), 2, 1).rotate(pi/2)
Ellipse(Point2D(0, 1), 1, 2)
>>> Ellipse((1, 0), 2, 1).rotate(pi)
Ellipse(Point2D(-1, 0), 2, 1)
```

**scale**(*x=1*, *y=1*, *pt=None*)

Override `GeometryEntity.scale` since it is the major and minor axes which must be scaled and they are not `GeometryEntities`.

## Examples

```
>>> from sympy import Ellipse
>>> Ellipse((0, 0), 2, 1).scale(2, 4)
Circle(Point2D(0, 0), 4)
>>> Ellipse((0, 0), 2, 1).scale(2)
Ellipse(Point2D(0, 0), 4, 1)
```

**second\_moment\_of\_area**(*point=None*)

Returns the second moment and product moment area of an ellipse.

### Parameters

**point** : Point, two-tuple of sympifiable objects, or `None` (default=`None`)

*point* is the point about which second moment of area is to be found. If “*point=None*” it will be calculated about the axis passing through the centroid of the ellipse.

### Returns

**I<sub>xx</sub>**, **I<sub>yy</sub>**, **I<sub>xy</sub>** : number or SymPy expression

*I<sub>xx</sub>*, *I<sub>yy</sub>* are second moment of area of an ellipse. *I<sub>xy</sub>* is product moment of area of an ellipse.

## Examples

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.second_moment_of_area()
(3*pi/4, 27*pi/4, 0)
```

## References

[R507]

**section\_modulus**(*point=None*)

Returns a tuple with the section modulus of an ellipse

Section modulus is a geometric property of an ellipse defined as the ratio of second moment of area to the distance of the extreme end of the ellipse from the centroidal axis.

### Parameters

**point** : Point, two-tuple of sympifyable objects, or None(default=None)

point is the point at which section modulus is to be found. If "point=None" section modulus will be calculated for the point farthest from the centroidal axis of the ellipse.

### Returns

S\_x, S\_y: numbers or SymPy expressions

S\_x is the section modulus with respect to the x-axis S\_y is the section modulus with respect to the y-axis A negative sign indicates that the section modulus is determined for a point below the centroidal axis.

## Examples

```
>>> from sympy import Symbol, Ellipse, Circle, Point2D
>>> d = Symbol('d', positive=True)
>>> c = Circle((0, 0), d/2)
>>> c.section_modulus()
(pi*d**3/32, pi*d**3/32)
>>> e = Ellipse(Point2D(0, 0), 2, 4)
>>> e.section_modulus()
(8*pi, 4*pi)
>>> e.section_modulus((2, 2))
(16*pi, 4*pi)
```

## References

[R508]

### property `semilatus_rectum`

Calculates the semi-latus rectum of the Ellipse.

Semi-latus rectum is defined as one half of the chord through a focus parallel to the conic section directrix of a conic section.

#### Returns

**`semilatus_rectum`** : number

## Examples

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.semilatus_rectum
1/3
```

#### See also:

#### [\*apoapsis\*](#) (page 2254)

Returns greatest distance between focus and contour

#### [\*periapsis\*](#) (page 2263)

The shortest distance between the focus and the contour

## References

[R509], [R510]

### `tangent_lines(p)`

Tangent lines between  $p$  and the ellipse.

If  $p$  is on the ellipse, returns the tangent line through point  $p$ . Otherwise, returns the tangent line(s) from  $p$  to the ellipse, or None if no tangent line is possible (e.g.,  $p$  inside ellipse).

#### Parameters

**`p`** : Point

#### Returns

**`tangent_lines`** : list with 1 or 2 Lines

#### Raises

**`NotImplementedError`**

Can only find tangent lines for a point,  $p$ , on the ellipse.

## Examples

```
>>> from sympy import Point, Ellipse
>>> e1 = Ellipse(Point(0, 0), 3, 2)
>>> e1.tangent_lines(Point(3, 0))
[Line2D(Point2D(3, 0), Point2D(3, -12))]
```

### See also:

[\*sympy.geometry.point.Point\*](#) (page 2202), [\*sympy.geometry.line.Line\*](#) (page 2227)

### property `vradius`

The vertical radius of the ellipse.

#### Returns

**vradius** : number

## Examples

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.vradius
1
```

### See also:

[\*hradius\*](#) (page 2259), [\*major\*](#) (page 2261), [\*minor\*](#) (page 2262)

**class** `sympy.geometry.ellipse.Circle(*args, **kwargs)`

A circle in space.

Constructed simply from a center and a radius, from three non-collinear points, or the equation of a circle.

#### Parameters

**center** : Point

**radius** : number or SymPy expression

**points** : sequence of three Points

**equation** : equation of a circle

#### Raises

**GeometryError**

When the given equation is not that of a circle. When trying to construct circle from incorrect parameters.

## Examples

```
>>> from sympy import Point, Circle, Eq
>>> from sympy.abc import x, y, a, b
```

A circle constructed from a center and radius:

```
>>> c1 = Circle(Point(0, 0), 5)
>>> c1.hradius, c1.vradius, c1.radius
(5, 5, 5)
```

A circle constructed from three points:

```
>>> c2 = Circle(Point(0, 0), Point(1, 1), Point(1, 0))
>>> c2.hradius, c2.vradius, c2.radius, c2.center
(sqrt(2)/2, sqrt(2)/2, sqrt(2)/2, Point2D(1/2, 1/2))
```

A circle can be constructed from an equation in the form  $a*x**2 + b*y**2 + g*x + h*y + c = 0$ , too:

```
>>> Circle(x**2 + y**2 - 25)
Circle(Point2D(0, 0), 5)
```

If the variables corresponding to  $x$  and  $y$  are named something else, their name or symbol can be supplied:

```
>>> Circle(Eq(a**2 + b**2, 25), x='a', y=b)
Circle(Point2D(0, 0), 5)
```

### See also:

[Ellipse](#) (page 2253), [sympy.geometry.point.Point](#) (page 2202)

## Attributes

|                                                            |  |
|------------------------------------------------------------|--|
| radius (synonymous with hradius, vradius, major and minor) |  |
| circumference                                              |  |
| equation                                                   |  |

### property circumference

The circumference of the circle.

#### Returns

**circumference** : number or SymPy expression

## Examples

```
>>> from sympy import Point, Circle
>>> c1 = Circle(Point(3, 4), 6)
>>> c1.circumference
12*pi
```

**equation**(*x*='x', *y*='y')

The equation of the circle.

### Parameters

**x** : str or Symbol, optional

Default value is 'x'.

**y** : str or Symbol, optional

Default value is 'y'.

### Returns

**equation** : SymPy expression

## Examples

```
>>> from sympy import Point, Circle
>>> c1 = Circle(Point(0, 0), 5)
>>> c1.equation()
x**2 + y**2 - 25
```

**intersection**(*o*)

The intersection of this circle with another geometrical entity.

### Parameters

**o** : GeometryEntity

### Returns

**intersection** : list of GeometryEntities

## Examples

```
>>> from sympy import Point, Circle, Line, Ray
>>> p1, p2, p3 = Point(0, 0), Point(5, 5), Point(6, 0)
>>> p4 = Point(5, 0)
>>> c1 = Circle(p1, 5)
>>> c1.intersection(p2)
[]
>>> c1.intersection(p4)
[Point2D(5, 0)]
>>> c1.intersection(Ray(p1, p2))
[Point2D(5*sqrt(2)/2, 5*sqrt(2)/2)]
>>> c1.intersection(Line(p2, p3))
[]
```

**property radius**

The radius of the circle.

**Returns**

**radius** : number or SymPy expression

**Examples**

```
>>> from sympy import Point, Circle
>>> c1 = Circle(Point(3, 4), 6)
>>> c1.radius
6
```

**See also:**

[\*Ellipse.major\*](#) (page 2261), [\*Ellipse.minor\*](#) (page 2262), [\*Ellipse.hradius\*](#) (page 2259), [\*Ellipse.vradius\*](#) (page 2269)

**reflect(*line*)**

Override GeometryEntity.reflect since the radius is not a GeometryEntity.

**Examples**

```
>>> from sympy import Circle, Line
>>> Circle((0, 1), 1).reflect(Line((0, 0), (1, 1)))
Circle(Point2D(1, 0), -1)
```

**scale(*x=1, y=1, pt=None*)**

Override GeometryEntity.scale since the radius is not a GeometryEntity.

**Examples**

```
>>> from sympy import Circle
>>> Circle((0, 0), 1).scale(2, 2)
Circle(Point2D(0, 0), 2)
>>> Circle((0, 0), 1).scale(2, 4)
Ellipse(Point2D(0, 0), 2, 4)
```

**property vradius**

This Ellipse property is an alias for the Circle's radius.

Whereas hradius, major and minor can use Ellipse's conventions, the vradius does not exist for a circle. It is always a positive value in order that the Circle, like Polygons, will have an area that can be positive or negative as determined by the sign of the hradius.



## Examples

```
>>> from sympy import Point, Circle
>>> c1 = Circle(Point(3, 4), 6)
>>> c1.vradius
6
```

## Polygons

**class** sympy.geometry.polygon.**Polygon**(\*args, n=0, \*\*kwargs)

A two-dimensional polygon.

A simple polygon in space. Can be constructed from a sequence of points or from a center, radius, number of sides and rotation angle.

### Parameters

**vertices** : sequence of Points

### Optional parameters

=====

**n** : If > 0, an n-sided RegularPolygon is created. See below.

Default value is 0.

### Raises

#### GeometryError

If all parameters are not Points.

## Notes

Polygons are treated as closed paths rather than 2D areas so some calculations can be be negative or positive (e.g., area) based on the orientation of the points.

Any consecutive identical points are reduced to a single point and any points collinear and between two points will be removed unless they are needed to define an explicit intersection (see examples).

A Triangle, Segment or Point will be returned when there are 3 or fewer points provided.

## Examples

```
>>> from sympy import Polygon, pi
>>> p1, p2, p3, p4, p5 = [(0, 0), (1, 0), (5, 1), (0, 1), (3, 0)]
>>> Polygon(p1, p2, p3, p4)
Polygon(Point2D(0, 0), Point2D(1, 0), Point2D(5, 1), Point2D(0, 1))
>>> Polygon(p1, p2)
Segment2D(Point2D(0, 0), Point2D(1, 0))
>>> Polygon(p1, p2, p5)
Segment2D(Point2D(0, 0), Point2D(3, 0))
```

The area of a polygon is calculated as positive when vertices are traversed in a ccw direction. When the sides of a polygon cross the area will have positive and negative contributions. The following defines a Z shape where the bottom right connects back to the top left.

```
>>> Polygon((0, 2), (2, 2), (0, 0), (2, 0)).area
0
```

When the keyword *n* is used to define the number of sides of the Polygon then a RegularPolygon is created and the other arguments are interpreted as center, radius and rotation. The unrotated RegularPolygon will always have a vertex at Point(*r*, 0) where *r* is the radius of the circle that circumscribes the RegularPolygon. Its method *spin* can be used to increment that angle.

```
>>> p = Polygon((0,0), 1, n=3)
>>> p
RegularPolygon(Point2D(0, 0), 1, 3, 0)
>>> p.vertices[0]
Point2D(1, 0)
>>> p.args[0]
Point2D(0, 0)
>>> p.spin(pi/2)
>>> p.vertices[0]
Point2D(0, 1)
```

**See also:**

[sympy.geometry.point.Point](#) (page 2202), [sympy.geometry.line.Segment](#) (page 2233), [Triangle](#) (page 2293)

**Attributes**

|           |  |
|-----------|--|
| area      |  |
| angles    |  |
| perimeter |  |
| vertices  |  |
| centroid  |  |
| sides     |  |

**property angles**

The internal angle at each vertex.

**Returns**

**angles** : dict

A dictionary where each key is a vertex and each value is the internal angle at that vertex. The vertices are represented as Points.

## Examples

```
>>> from sympy import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.angles[p1]
pi/2
>>> poly.angles[p2]
acos(-4*sqrt(17)/17)
```

### See also:

[\*sympy.geometry.point.Point\*](#) (page 2202), [\*sympy.geometry.line.LinearEntity.angle\\_between\*](#) (page 2216)

### **arbitrary\_point**(parameter='t')

A parameterized point on the polygon.

The parameter, varying from 0 to 1, assigns points to the position on the perimeter that is that fraction of the total perimeter. So the point evaluated at  $t=1/2$  would return the point from the first vertex that is 1/2 way around the polygon.

#### Parameters

**parameter** : str, optional

Default value is 't'.

#### Returns

**arbitrary\_point** : Point

#### Raises

**ValueError**

When *parameter* already appears in the Polygon's definition.

## Examples

```
>>> from sympy import Polygon, Symbol
>>> t = Symbol('t', real=True)
>>> tri = Polygon((0, 0), (1, 0), (1, 1))
>>> p = tri.arbitrary_point('t')
>>> perimeter = tri.perimeter
>>> s1, s2 = [s.length for s in tri.sides[:2]]
>>> p.subs(t, (s1 + s2/2)/perimeter)
Point2D(1, 1/2)
```

### See also:

[\*sympy.geometry.point.Point\*](#) (page 2202)

### **property area**

The area of the polygon.

## Notes

The area calculation can be positive or negative based on the orientation of the points. If any side of the polygon crosses any other side, there will be areas having opposite signs.

## Examples

```
>>> from sympy import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.area
3
```

In the Z shaped polygon (with the lower right connecting back to the upper left) the areas cancel out:

```
>>> Z = Polygon((0, 1), (1, 1), (0, 0), (1, 0))
>>> Z.area
0
```

In the M shaped polygon, areas do not cancel because no side crosses any other (though there is a point of contact).

```
>>> M = Polygon((0, 0), (0, 1), (2, 0), (3, 1), (3, 0))
>>> M.area
-3/2
```

### See also:

[\*sympy.geometry.ellipse.Ellipse.area\*](#) (page 2255)

### **bisectors**(*prec=None*)

Returns angle bisectors of a polygon. If *prec* is given then approximate the point defining the ray to that precision.

The distance between the points defining the bisector ray is 1.

## Examples

```
>>> from sympy import Polygon, Point
>>> p = Polygon(Point(0, 0), Point(2, 0), Point(1, 1), Point(0, 3))
>>> p.bisectors(2)
{Point2D(0, 0): Ray2D(Point2D(0, 0), Point2D(0.71, 0.71)),
 Point2D(0, 3): Ray2D(Point2D(0, 3), Point2D(0.23, 2.0)),
 Point2D(1, 1): Ray2D(Point2D(1, 1), Point2D(0.19, 0.42)),
 Point2D(2, 0): Ray2D(Point2D(2, 0), Point2D(1.1, 0.38))}
```

### **property bounds**

Return a tuple (xmin, ymin, xmax, ymax) representing the bounding rectangle for the geometric figure.

**property centroid**

The centroid of the polygon.

**Returns**

**centroid** : Point

**Examples**

```
>>> from sympy import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.centroid
Point2D(31/18, 11/18)
```

**See also:**

[\*sympy.geometry.point.Point\*](#) (page 2202), [\*sympy.geometry.util.centroid\*](#) (page 2200)

**cut\_section(*line*)**

Returns a tuple of two polygon segments that lie above and below the intersecting line respectively.

**Parameters**

**line**: Line object of geometry module

line which cuts the Polygon. The part of the Polygon that lies above and below this line is returned.

**Returns**

upper\_polygon, lower\_polygon: Polygon objects or None

upper\_polygon is the polygon that lies above the given line.  
lower\_polygon is the polygon that lies below the given line. upper\_polygon and lower\_polygon are None when no polygon exists above the line or below the line.

**Raises**

**ValueError: When the line does not intersect the polygon**

**Examples**

```
>>> from sympy import Polygon, Line
>>> a, b = 20, 10
>>> p1, p2, p3, p4 = [(0, b), (0, 0), (a, 0), (a, b)]
>>> rectangle = Polygon(p1, p2, p3, p4)
>>> t = rectangle.cut_section(Line((0, 5), slope=0))
>>> t
(Polygon(Point2D(0, 10), Point2D(0, 5), Point2D(20, 5), Point2D(20, ↵
↵10)),
Polygon(Point2D(0, 5), Point2D(0, 0), Point2D(20, 0), Point2D(20, 5)))
>>> upper_segment, lower_segment = t
>>> upper_segment.area
100
```

(continues on next page)

(continued from previous page)

```
>>> upper_segment.centroid
Point2D(10, 15/2)
>>> lower_segment.centroid
Point2D(10, 5/2)
```

## References

[R511]

### **distance(*o*)**

Returns the shortest distance between self and *o*.

If *o* is a point, then self does not need to be convex. If *o* is another polygon self and *o* must be convex.

## Examples

```
>>> from sympy import Point, Polygon, RegularPolygon
>>> p1, p2 = map(Point, [(0, 0), (7, 5)])
>>> poly = Polygon(*RegularPolygon(p1, 1, 3).vertices)
>>> poly.distance(p2)
sqrt(61)
```

### **encloses\_point(*p*)**

Return True if *p* is enclosed by (is inside of) self.

#### **Parameters**

**p** : Point

#### **Returns**

**encloses\_point** : True, False or None

## Notes

Being on the border of self is considered False.

## Examples

```
>>> from sympy import Polygon, Point
>>> p = Polygon((0, 0), (4, 0), (4, 4))
>>> p.encloses_point(Point(2, 1))
True
>>> p.encloses_point(Point(2, 2))
False
>>> p.encloses_point(Point(5, 5))
False
```

**See also:**

*sympy.geometry.point.Point* (page 2202), *sympy.geometry.ellipse.Ellipse.encloses\_point* (page 2257)

**References**

[R512]

**first\_moment\_of\_area**(*point=None*)

Returns the first moment of area of a two-dimensional polygon with respect to a certain point of interest.

First moment of area is a measure of the distribution of the area of a polygon in relation to an axis. The first moment of area of the entire polygon about its own centroid is always zero. Therefore, here it is calculated for an area, above or below a certain point of interest, that makes up a smaller portion of the polygon. This area is bounded by the point of interest and the extreme end (top or bottom) of the polygon. The first moment for this area is then determined about the centroidal axis of the initial polygon.

**Parameters**

**point:** **Point, two-tuple of sympifyable objects, or None (default=None)**

point is the point above or below which the area of interest lies If point=None then the centroid acts as the point of interest.

**Returns**

Q\_x, Q\_y: number or SymPy expressions

Q\_x is the first moment of area about the x-axis Q\_y is the first moment of area about the y-axis A negative sign indicates that the section modulus is determined for a section below (or left of) the centroidal axis

**Examples**

```
>>> from sympy import Point, Polygon
>>> a, b = 50, 10
>>> p1, p2, p3, p4 = [(0, b), (0, 0), (a, 0), (a, b)]
>>> p = Polygon(p1, p2, p3, p4)
>>> p.first_moment_of_area()
(625, 3125)
>>> p.first_moment_of_area(point=Point(30, 7))
(525, 3000)
```

## References

[R513], [R514]

### `intersection(o)`

The intersection of polygon and geometry entity.

The intersection may be empty and can contain individual Points and complete Line Segments.

#### Parameters

**other:** `GeometryEntity`

#### Returns

**intersection** : list

The list of Segments and Points

## Examples

```
>>> from sympy import Point, Polygon, Line
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly1 = Polygon(p1, p2, p3, p4)
>>> p5, p6, p7 = map(Point, [(3, 2), (1, -1), (0, 2)])
>>> poly2 = Polygon(p5, p6, p7)
>>> poly1.intersection(poly2)
[Point2D(1/3, 1), Point2D(2/3, 0), Point2D(9/5, 1/5), Point2D(7/3, 1)]
>>> poly1.intersection(Line(p1, p2))
[Segment2D(Point2D(0, 0), Point2D(1, 0))]
>>> poly1.intersection(p1)
[Point2D(0, 0)]
```

#### See also:

[\*sympy.geometry.point.Point\*](#) (page 2202), [\*sympy.geometry.line.Segment\*](#) (page 2233)

### `is_convex()`

Is the polygon convex?

A polygon is convex if all its interior angles are less than 180 degrees and there are no intersections between sides.

#### Returns

**is\_convex** : boolean

True if this polygon is convex, False otherwise.



## Examples

```
>>> from sympy import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.is_convex()
True
```

### See also:

[\*sympy.geometry.util.convex\\_hull\*](#) (page 2198)

### property `perimeter`

The perimeter of the polygon.

#### Returns

**perimeter** : number or Basic instance

## Examples

```
>>> from sympy import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.perimeter
sqrt(17) + 7
```

### See also:

[\*sympy.geometry.line.Segment.length\*](#) (page 2235)

### `plot_interval(parameter='t')`

The plot interval for the default geometric plot of the polygon.

#### Parameters

**parameter** : str, optional

Default value is 't'.

#### Returns

**plot\_interval** : list (plot interval)

[parameter, lower\_bound, upper\_bound]

## Examples

```
>>> from sympy import Polygon
>>> p = Polygon((0, 0), (1, 0), (1, 1))
>>> p.plot_interval()
[t, 0, 1]
```

### `polar_second_moment_of_area()`

Returns the polar modulus of a two-dimensional polygon

It is a constituent of the second moment of area, linked through the perpendicular axis theorem. While the planar second moment of area describes an object's resistance to deflection (bending) when subjected to a force applied to a plane parallel to the central axis, the polar second moment of area describes an object's resistance to deflection when subjected to a moment applied in a plane perpendicular to the object's central axis (i.e. parallel to the cross-section)

### Examples

```
>>> from sympy import Polygon, symbols
>>> a, b = symbols('a, b')
>>> rectangle = Polygon((0, 0), (a, 0), (a, b), (0, b))
>>> rectangle.polar_second_moment_of_area()
a**3*b/12 + a*b**3/12
```

### References

[R515]

**second\_moment\_of\_area**(*point=None*)

Returns the second moment and product moment of area of a two dimensional polygon.

#### Parameters

**point** : Point, two-tuple of sympifyable objects, or None(default=None)

point is the point about which second moment of area is to be found.  
If "point=None" it will be calculated about the axis passing through the centroid of the polygon.

#### Returns

**I<sub>xx</sub>, I<sub>yy</sub>, I<sub>xy</sub>** : number or SymPy expression

I<sub>xx</sub>, I<sub>yy</sub> are second moment of area of a two dimensional polygon.

I<sub>xy</sub> is product moment of area of a two dimensional polygon.

### Examples

```
>>> from sympy import Polygon, symbols
>>> a, b = symbols('a, b')
>>> p1, p2, p3, p4, p5 = [(0, 0), (a, 0), (a, b), (0, b), (a/3, b/3)]
>>> rectangle = Polygon(p1, p2, p3, p4)
>>> rectangle.second_moment_of_area()
(a*b**3/12, a**3*b/12, 0)
>>> rectangle.second_moment_of_area(p5)
(a*b**3/9, a**3*b/9, a**2*b**2/36)
```

## References

[R516]

**section\_modulus**(*point=None*)

Returns a tuple with the section modulus of a two-dimensional polygon.

Section modulus is a geometric property of a polygon defined as the ratio of second moment of area to the distance of the extreme end of the polygon from the centroidal axis.

### Parameters

**point** : Point, two-tuple of sympifyable objects, or None(default=None)

point is the point at which section modulus is to be found. If “point=None” it will be calculated for the point farthest from the centroidal axis of the polygon.

### Returns

S\_x, S\_y: numbers or SymPy expressions

S\_x is the section modulus with respect to the x-axis S\_y is the section modulus with respect to the y-axis A negative sign indicates that the section modulus is determined for a point below the centroidal axis

## Examples

```
>>> from sympy import symbols, Polygon, Point
>>> a, b = symbols('a, b', positive=True)
>>> rectangle = Polygon((0, 0), (a, 0), (a, b), (0, b))
>>> rectangle.section_modulus()
(a*b**2/6, a**2*b/6)
>>> rectangle.section_modulus(Point(a/4, b/4))
(-a*b**2/3, -a**2*b/3)
```

## References

[R517]

**property sides**

The directed line segments that form the sides of the polygon.

### Returns

**sides** : list of sides

Each side is a directed Segment.

## Examples

```
>>> from sympy import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.sides
[Segment2D(Point2D(0, 0), Point2D(1, 0)),
Segment2D(Point2D(1, 0), Point2D(5, 1)),
Segment2D(Point2D(5, 1), Point2D(0, 1)), Segment2D(Point2D(0, 1),
↪Point2D(0, 0))]
```

### See also:

[\*sympy.geometry.point.Point\*](#) (page 2202), [\*sympy.geometry.line.Segment\*](#) (page 2233)

### property vertices

The vertices of the polygon.

#### Returns

**vertices** : list of Points

## Notes

When iterating over the vertices, it is more efficient to index self rather than to request the vertices and index them. Only use the vertices when you want to process all of them at once. This is even more important with RegularPolygons that calculate each vertex.

## Examples

```
>>> from sympy import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.vertices
[Point2D(0, 0), Point2D(1, 0), Point2D(5, 1), Point2D(0, 1)]
>>> poly.vertices[0]
Point2D(0, 0)
```

### See also:

[\*sympy.geometry.point.Point\*](#) (page 2202)

**class** sympy.geometry.polygon.**RegularPolygon**(*c, r, n, rot=0, \*\*kwargs*)

A regular polygon.

Such a polygon has all internal angles equal and all sides the same length.

#### Parameters

**center** : Point

**radius** : number or Basic instance

The distance from the center to a vertex

**n** : int

The number of sides

### Raises

#### GeometryError

If the *center* is not a Point, or the *radius* is not a number or Basic instance, or the number of sides, *n*, is less than three.

### Notes

A RegularPolygon can be instantiated with Polygon with the kwarg *n*.

Regular polygons are instantiated with a center, radius, number of sides and a rotation angle. Whereas the arguments of a Polygon are vertices, the vertices of the RegularPolygon must be obtained with the vertices method.

### Examples

```
>>> from sympy import RegularPolygon, Point
>>> r = RegularPolygon(Point(0, 0), 5, 3)
>>> r
RegularPolygon(Point2D(0, 0), 5, 3, 0)
>>> r.vertices[0]
Point2D(5, 0)
```

### See also:

[\*sympy.geometry.point.Point\*](#) (page 2202), [\*Polygon\*](#) (page 2273)

### Attributes

|                |  |
|----------------|--|
| vertices       |  |
| center         |  |
| radius         |  |
| rotation       |  |
| apothem        |  |
| interior_angle |  |
| exterior_angle |  |
| circumcircle   |  |
| incircle       |  |
| angles         |  |

### property angles

Returns a dictionary with keys, the vertices of the Polygon, and values, the interior angle at each vertex.

## Examples

```
>>> from sympy import RegularPolygon, Point
>>> r = RegularPolygon(Point(0, 0), 5, 3)
>>> r.angles
{Point2D(-5/2, -5*sqrt(3)/2): pi/3,
 Point2D(-5/2, 5*sqrt(3)/2): pi/3,
 Point2D(5, 0): pi/3}
```

### property apothem

The inradius of the RegularPolygon.

The apothem/inradius is the radius of the inscribed circle.

#### Returns

**apothem** : number or instance of Basic

## Examples

```
>>> from sympy import Symbol
>>> from sympy import RegularPolygon, Point
>>> radius = Symbol('r')
>>> rp = RegularPolygon(Point(0, 0), radius, 4)
>>> rp.apothem
sqrt(2)*r/2
```

#### See also:

[\*sympy.geometry.line.Segment.length\*](#) (page 2235), [\*sympy.geometry.ellipse.Circle.radius\*](#) (page 2271)

### property area

Returns the area.

## Examples

```
>>> from sympy import RegularPolygon
>>> square = RegularPolygon((0, 0), 1, 4)
>>> square.area
2
>>> _ == square.length**2
True
```

### property args

Returns the center point, the radius, the number of sides, and the orientation angle.

## Examples

```
>>> from sympy import RegularPolygon, Point
>>> r = RegularPolygon(Point(0, 0), 5, 3)
>>> r.args
(Point2D(0, 0), 5, 3, 0)
```

### property center

The center of the RegularPolygon

This is also the center of the circumscribing circle.

#### Returns

**center** : Point

## Examples

```
>>> from sympy import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 5, 4)
>>> rp.center
Point2D(0, 0)
```

#### See also:

[\*sympy.geometry.point.Point\*](#) (page 2202), [\*sympy.geometry.ellipse.Ellipse.center\*](#) (page 2255)

### property centroid

The center of the RegularPolygon

This is also the center of the circumscribing circle.

#### Returns

**center** : Point

## Examples

```
>>> from sympy import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 5, 4)
>>> rp.center
Point2D(0, 0)
```

#### See also:

[\*sympy.geometry.point.Point\*](#) (page 2202), [\*sympy.geometry.ellipse.Ellipse.center\*](#) (page 2255)

### property circumcenter

Alias for center.

### Examples

```
>>> from sympy import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 5, 4)
>>> rp.circumcenter
Point2D(0, 0)
```

#### property `circumcircle`

The circumcircle of the RegularPolygon.

##### Returns

**circumcircle** : Circle

### Examples

```
>>> from sympy import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 4, 8)
>>> rp.circumcircle
Circle(Point2D(0, 0), 4)
```

#### See also:

[\*circumcenter\*](#) (page 2287), [\*sympy.geometry.ellipse.Circle\*](#) (page 2269)

#### property `circumradius`

Alias for radius.

### Examples

```
>>> from sympy import Symbol
>>> from sympy import RegularPolygon, Point
>>> radius = Symbol('r')
>>> rp = RegularPolygon(Point(0, 0), radius, 4)
>>> rp.circumradius
r
```

#### `encloses_point(p)`

Return True if p is enclosed by (is inside of) self.

##### Parameters

**p** : Point

##### Returns

**encloses\_point** : True, False or None



## Notes

Being on the border of self is considered False.

The general Polygon.encloses\_point method is called only if a point is not within or beyond the incircle or circumcircle, respectively.

## Examples

```
>>> from sympy import RegularPolygon, S, Point, Symbol
>>> p = RegularPolygon((0, 0), 3, 4)
>>> p.encloses_point(Point(0, 0))
True
>>> r, R = p.inradius, p.circumradius
>>> p.encloses_point(Point((r + R)/2, 0))
True
>>> p.encloses_point(Point(R/2, R/2 + (R - r)/10))
False
>>> t = Symbol('t', real=True)
>>> p.encloses_point(p.arbitrary_point().subs(t, S.Half))
False
>>> p.encloses_point(Point(5, 5))
False
```

### See also:

[\*sympy.geometry.ellipse.Ellipse.encloses\\_point\*](#) (page 2257)

### property exterior\_angle

Measure of the exterior angles.

#### Returns

**exterior\_angle** : number

## Examples

```
>>> from sympy import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 4, 8)
>>> rp.exterior_angle
pi/4
```

### See also:

[\*sympy.geometry.line.LinearEntity.angle\\_between\*](#) (page 2216)

### property incircle

The incircle of the RegularPolygon.

#### Returns

**incircle** : Circle

### Examples

```
>>> from sympy import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 4, 7)
>>> rp.incircle
Circle(Point2D(0, 0), 4*cos(pi/7))
```

#### See also:

[\*inradius\*](#) (page 2290), [\*sympy.geometry.ellipse.Circle\*](#) (page 2269)

#### property **inradius**

Alias for apothem.

### Examples

```
>>> from sympy import Symbol
>>> from sympy import RegularPolygon, Point
>>> radius = Symbol('r')
>>> rp = RegularPolygon(Point(0, 0), radius, 4)
>>> rp.inradius
sqrt(2)*r/2
```

#### property **interior\_angle**

Measure of the interior angles.

##### Returns

**interior\_angle** : number

### Examples

```
>>> from sympy import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 4, 8)
>>> rp.interior_angle
3*pi/4
```

#### See also:

[\*sympy.geometry.line.LinearEntity.angle\\_between\*](#) (page 2216)

#### property **length**

Returns the length of the sides.

The half-length of the side and the apothem form two legs of a right triangle whose hypotenuse is the radius of the regular polygon.

## Examples

```
>>> from sympy import RegularPolygon
>>> from sympy import sqrt
>>> s = square_in_unit_circle = RegularPolygon((0, 0), 1, 4)
>>> s.length
sqrt(2)
>>> sqrt((_/2)**2 + s.apothem**2) == s.radius
True
```

### property radius

Radius of the RegularPolygon

This is also the radius of the circumscribing circle.

#### Returns

**radius** : number or instance of Basic

## Examples

```
>>> from sympy import Symbol
>>> from sympy import RegularPolygon, Point
>>> radius = Symbol('r')
>>> rp = RegularPolygon(Point(0, 0), radius, 4)
>>> rp.radius
r
```

### See also:

[\*sympy.geometry.line.Segment.length\*](#) (page 2235), [\*sympy.geometry.ellipse.Circle.radius\*](#) (page 2271)

### reflect(*line*)

Override GeometryEntity.reflect since this is not made of only points.

## Examples

```
>>> from sympy import RegularPolygon, Line
```

```
>>> RegularPolygon((0, 0), 1, 4).reflect(Line((0, 1), slope=-2))
RegularPolygon(Point2D(4/5, 2/5), -1, 4, atan(4/3))
```

### rotate(*angle*, *pt=None*)

Override GeometryEntity.rotate to first rotate the RegularPolygon about its center.

```
>>> from sympy import Point, RegularPolygon, pi
>>> t = RegularPolygon(Point(1, 0), 1, 3)
>>> t.vertices[0] # vertex on x-axis
Point2D(2, 0)
>>> t.rotate(pi/2).vertices[0] # vertex on y axis now
Point2D(0, 2)
```

**See also:**

[rotation](#) (page 2292)

**[spin](#) (page 2292)**

Rotates a RegularPolygon in place

**property rotation**

CCW angle by which the RegularPolygon is rotated

**Returns**

**rotation** : number or instance of Basic

**Examples**

```
>>> from sympy import pi
>>> from sympy.abc import a
>>> from sympy import RegularPolygon, Point
>>> RegularPolygon(Point(0, 0), 3, 4, pi/4).rotation
pi/4
```

Numerical rotation angles are made canonical:

```
>>> RegularPolygon(Point(0, 0), 3, 4, a).rotation
a
>>> RegularPolygon(Point(0, 0), 3, 4, pi).rotation
0
```

**scale(x=1, y=1, pt=None)**

Override GeometryEntity.scale since it is the radius that must be scaled (if x == y) or else a new Polygon must be returned.

```
>>> from sympy import RegularPolygon
```

Symmetric scaling returns a RegularPolygon:

```
>>> RegularPolygon((0, 0), 1, 4).scale(2, 2)
RegularPolygon(Point2D(0, 0), 2, 4, 0)
```

Asymmetric scaling returns a kite as a Polygon:

```
>>> RegularPolygon((0, 0), 1, 4).scale(2, 1)
Polygon(Point2D(2, 0), Point2D(0, 1), Point2D(-2, 0), Point2D(0, -1))
```

**spin(angle)**

Increment *in place* the virtual Polygon's rotation by ccw angle.

See also: rotate method which moves the center.

```
>>> from sympy import Polygon, Point, pi
>>> r = Polygon(Point(0,0), 1, n=3)
>>> r.vertices[0]
Point2D(1, 0)
>>> r.spin(pi/6)
```

(continues on next page)

(continued from previous page)

```
>>> r.vertices[0]
Point2D(sqrt(3)/2, 1/2)
```

**See also:**[rotation](#) (page 2292)**rotate** (page 2291)

Creates a copy of the RegularPolygon rotated about a Point

**property vertices**

The vertices of the RegularPolygon.

**Returns****vertices** : list

Each vertex is a Point.

**Examples**

```
>>> from sympy import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 5, 4)
>>> rp.vertices
[Point2D(5, 0), Point2D(0, 5), Point2D(-5, 0), Point2D(0, -5)]
```

**See also:**[sympy.geometry.point.Point](#) (page 2202)**class** sympy.geometry.polygon.**Triangle**(\*args, \*\*kwargs)

A polygon with three vertices and three sides.

**Parameters****points** : sequence of Points**keyword:** **asa**, **sas**, or **sss** to specify sides/angles of the triangle**Raises****GeometryError**

If the number of vertices is not equal to three, or one of the vertices is not a Point, or a valid keyword is not given.

**Examples**

```
>>> from sympy import Triangle, Point
>>> Triangle(Point(0, 0), Point(4, 0), Point(4, 3))
Triangle(Point2D(0, 0), Point2D(4, 0), Point2D(4, 3))
```

Keywords sss, sas, or asa can be used to give the desired side lengths (in order) and interior angles (in degrees) that define the triangle:

```
>>> Triangle(sss=(3, 4, 5))
Triangle(Point2D(0, 0), Point2D(3, 0), Point2D(3, 4))
>>> Triangle(asa=(30, 1, 30))
Triangle(Point2D(0, 0), Point2D(1, 0), Point2D(1/2, sqrt(3)/6))
>>> Triangle(sas=(1, 45, 2))
Triangle(Point2D(0, 0), Point2D(2, 0), Point2D(sqrt(2)/2, sqrt(2)/2))
```

**See also:**

[\*sympy.geometry.point.Point\*](#) (page 2202), [\*Polygon\*](#) (page 2273)

**Attributes**

|                   |  |
|-------------------|--|
| vertices          |  |
| altitudes         |  |
| orthocenter       |  |
| circumcenter      |  |
| circumradius      |  |
| circumcircle      |  |
| inradius          |  |
| incircle          |  |
| exradii           |  |
| medians           |  |
| medial            |  |
| nine_point_circle |  |

**property altitudes**

The altitudes of the triangle.

An altitude of a triangle is a segment through a vertex, perpendicular to the opposite side, with length being the height of the vertex measured from the line containing the side.

**Returns**

**altitudes** : dict

The dictionary consists of keys which are vertices and values which are Segments.

**Examples**

```
>>> from sympy import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.altitudes[p1]
Segment2D(Point2D(0, 0), Point2D(1/2, 1/2))
```

**See also:**

[\*sympy.geometry.point.Point\*](#) (page 2202), [\*sympy.geometry.line.Segment.length\*](#) (page 2235)

**bisectors()**

The angle bisectors of the triangle.

An angle bisector of a triangle is a straight line through a vertex which cuts the corresponding angle in half.

**Returns**

**bisectors** : dict

Each key is a vertex (Point) and each value is the corresponding bisector (Segment).

**Examples**

```
>>> from sympy import Point, Triangle, Segment
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> from sympy import sqrt
>>> t.bisectors()[p2] == Segment(Point(1, 0), Point(0, sqrt(2) - 1))
True
```

**See also:**

[\*sympy.geometry.point.Point\*](#) (page 2202), [\*sympy.geometry.line.Segment\*](#) (page 2233)

**property circumcenter**

The circumcenter of the triangle

The circumcenter is the center of the circumcircle.

**Returns**

**circumcenter** : Point

**Examples**

```
>>> from sympy import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.circumcenter
Point2D(1/2, 1/2)
```

**See also:**

[\*sympy.geometry.point.Point\*](#) (page 2202)

**property circumcircle**

The circle which passes through the three vertices of the triangle.

**Returns**

**circumcircle** : Circle

## Examples

```
>>> from sympy import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.circumcircle
Circle(Point2D(1/2, 1/2), sqrt(2)/2)
```

### See also:

[\*sympy.geometry.ellipse.Circle\*](#) (page 2269)

### property **circumradius**

The radius of the circumcircle of the triangle.

#### Returns

**circumradius** : number of Basic instance

## Examples

```
>>> from sympy import Symbol
>>> from sympy import Point, Triangle
>>> a = Symbol('a')
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, a)
>>> t = Triangle(p1, p2, p3)
>>> t.circumradius
sqrt(a**2/4 + 1/4)
```

### See also:

[\*sympy.geometry.ellipse.Circle.radius\*](#) (page 2271)

### property **eulerline**

The Euler line of the triangle.

The line which passes through circumcenter, centroid and orthocenter.

#### Returns

**eulerline** : Line (or Point for equilateral triangles in which case all centers coincide)

## Examples

```
>>> from sympy import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.eulerline
Line2D(Point2D(0, 0), Point2D(1/2, 1/2))
```

### property **excenters**

Excenters of the triangle.

An excenter is the center of a circle that is tangent to a side of the triangle and the extensions of the other two sides.



**Returns****excenters** : dict**Examples**

The excenters are keyed to the side of the triangle to which their corresponding excircle is tangent: The center is keyed, e.g. the excenter of a circle touching side 0 is:

```
>>> from sympy import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(6, 0), Point(0, 2)
>>> t = Triangle(p1, p2, p3)
>>> t.excenters[t.sides[0]]
Point2D(12*sqrt(10), 2/3 + sqrt(10)/3)
```

**See also:**

[\*sympy.geometry.polygon.Triangle.exradii\*](#) (page 2297)

**References**

[R518]

**property `exradii`**

The radius of excircles of a triangle.

An excircle of the triangle is a circle lying outside the triangle, tangent to one of its sides and tangent to the extensions of the other two.

**Returns****exradii** : dict**Examples**

The exradius touches the side of the triangle to which it is keyed, e.g. the exradius touching side 2 is:

```
>>> from sympy import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(6, 0), Point(0, 2)
>>> t = Triangle(p1, p2, p3)
>>> t.exradii[t.sides[2]]
-2 + sqrt(10)
```

**See also:**

[\*sympy.geometry.polygon.Triangle.inradius\*](#) (page 2298)

## References

[R519], [R520]

### property `incenter`

The center of the incircle.

The incircle is the circle which lies inside the triangle and touches all three sides.

#### Returns

**incenter** : Point

## Examples

```
>>> from sympy import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.incenter
Point2D(1 - sqrt(2)/2, 1 - sqrt(2)/2)
```

#### See also:

[\*incircle\*](#) (page 2298), [\*sympy.geometry.point.Point\*](#) (page 2202)

### property `incircle`

The incircle of the triangle.

The incircle is the circle which lies inside the triangle and touches all three sides.

#### Returns

**incircle** : Circle

## Examples

```
>>> from sympy import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(2, 0), Point(0, 2)
>>> t = Triangle(p1, p2, p3)
>>> t.incircle
Circle(Point2D(2 - sqrt(2), 2 - sqrt(2)), 2 - sqrt(2))
```

#### See also:

[\*sympy.geometry.ellipse.Circle\*](#) (page 2269)

### property `inradius`

The radius of the incircle.

#### Returns

**inradius** : number of Basic instance

## Examples

```
>>> from sympy import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(4, 0), Point(0, 3)
>>> t = Triangle(p1, p2, p3)
>>> t.inradius
1
```

### See also:

[\*incircle\*](#) (page 2298), [\*sympy.geometry.ellipse.Circle.radius\*](#) (page 2271)

## **is\_equilateral()**

Are all the sides the same length?

### Returns

**is\_equilateral** : boolean

## Examples

```
>>> from sympy import Triangle, Point
>>> t1 = Triangle(Point(0, 0), Point(4, 0), Point(4, 3))
>>> t1.is_equilateral()
False
```

```
>>> from sympy import sqrt
>>> t2 = Triangle(Point(0, 0), Point(10, 0), Point(5, 5*sqrt(3)))
>>> t2.is_equilateral()
True
```

### See also:

[\*sympy.geometry.entity.GeometryEntity.is\\_similar\*](#) (page 2195),  
[\*RegularPolygon\*](#) (page 2284), [\*is\\_isosceles\*](#) (page 2299), [\*is\\_right\*](#) (page 2299),  
[\*is\\_scalene\*](#) (page 2300)

## **is\_isosceles()**

Are two or more of the sides the same length?

### Returns

**is\_isosceles** : boolean

## Examples

```
>>> from sympy import Triangle, Point
>>> t1 = Triangle(Point(0, 0), Point(4, 0), Point(2, 4))
>>> t1.is_isosceles()
True
```

### See also:

[\*is\\_equilateral\*](#) (page 2299), [\*is\\_right\*](#) (page 2299), [\*is\\_scalene\*](#) (page 2300)

**is\_right()**

Is the triangle right-angled.

**Returns**

**is\_right** : boolean

**Examples**

```
>>> from sympy import Triangle, Point
>>> t1 = Triangle(Point(0, 0), Point(4, 0), Point(4, 3))
>>> t1.is_right()
True
```

**See also:**

[\*sympy.geometry.line.LinearEntity.is\\_perpendicular\*](#) (page 2221),  
[\*is\\_equilateral\*](#) (page 2299), [\*is\\_isosceles\*](#) (page 2299), [\*is\\_scalene\*](#) (page 2300)

**is\_scalene()**

Are all the sides of the triangle of different lengths?

**Returns**

**is\_scalene** : boolean

**Examples**

```
>>> from sympy import Triangle, Point
>>> t1 = Triangle(Point(0, 0), Point(4, 0), Point(1, 4))
>>> t1.is_scalene()
True
```

**See also:**

[\*is\\_equilateral\*](#) (page 2299), [\*is\\_isosceles\*](#) (page 2299), [\*is\\_right\*](#) (page 2299)

**is\_similar(t2)**

Is another triangle similar to this one.

Two triangles are similar if one can be uniformly scaled to the other.

**Parameters**

**other**: Triangle

**Returns**

**is\_similar** : boolean

## Examples

```
>>> from sympy import Triangle, Point
>>> t1 = Triangle(Point(0, 0), Point(4, 0), Point(4, 3))
>>> t2 = Triangle(Point(0, 0), Point(-4, 0), Point(-4, -3))
>>> t1.is_similar(t2)
True
```

```
>>> t2 = Triangle(Point(0, 0), Point(-4, 0), Point(-4, -4))
>>> t1.is_similar(t2)
False
```

### See also:

[\*sympy.geometry.entity.GeometryEntity.is\\_similar\*](#) (page 2195)

### property `medial`

The medial triangle of the triangle.

The triangle which is formed from the midpoints of the three sides.

#### Returns

**medial** : Triangle

## Examples

```
>>> from sympy import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.medial
Triangle(Point2D(1/2, 0), Point2D(1/2, 1/2), Point2D(0, 1/2))
```

### See also:

[\*sympy.geometry.line.Segment.midpoint\*](#) (page 2235)

### property `medians`

The medians of the triangle.

A median of a triangle is a straight line through a vertex and the midpoint of the opposite side, and divides the triangle into two equal areas.

#### Returns

**medians** : dict

Each key is a vertex (Point) and each value is the median (Segment) at that point.

## Examples

```
>>> from sympy import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.medians[p1]
Segment2D(Point2D(0, 0), Point2D(1/2, 1/2))
```

### See also:

[\*sympy.geometry.point.Point.midpoint\*](#) (page 2207), [\*sympy.geometry.line.Segment.midpoint\*](#) (page 2235)

### property `nine_point_circle`

The nine-point circle of the triangle.

Nine-point circle is the circumcircle of the medial triangle, which passes through the feet of altitudes and the middle points of segments connecting the vertices and the orthocenter.

### Returns

**`nine_point_circle`** : Circle

## Examples

```
>>> from sympy import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.nine_point_circle
Circle(Point2D(1/4, 1/4), sqrt(2)/4)
```

### See also:

[\*sympy.geometry.line.Segment.midpoint\*](#) (page 2235), [\*sympy.geometry.polygon.Triangle.medial\*](#) (page 2301), [\*sympy.geometry.polygon.Triangle.orthocenter\*](#) (page 2302)

### property `orthocenter`

The orthocenter of the triangle.

The orthocenter is the intersection of the altitudes of a triangle. It may lie inside, outside or on the triangle.

### Returns

**`orthocenter`** : Point

## Examples

```
>>> from sympy import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.orthocenter
Point2D(0, 0)
```

### See also:

[\*sympy.geometry.point.Point\*](#) (page 2202)

### property vertices

The triangle's vertices

#### Returns

**vertices** : tuple

Each element in the tuple is a Point

## Examples

```
>>> from sympy import Triangle, Point
>>> t = Triangle(Point(0, 0), Point(4, 0), Point(4, 3))
>>> t.vertices
(Point2D(0, 0), Point2D(4, 0), Point2D(4, 3))
```

### See also:

[\*sympy.geometry.point.Point\*](#) (page 2202)

## Plane

**class** `sympy.geometry.plane.Plane(p1, a=None, b=None, **kwargs)`

A plane is a flat, two-dimensional surface. A plane is the two-dimensional analogue of a point (zero-dimensions), a line (one-dimension) and a solid (three-dimensions). A plane can generally be constructed by two types of inputs. They are three non-collinear points and a point and the plane's normal vector.

## Examples

```
>>> from sympy import Plane, Point3D
>>> Plane(Point3D(1, 1, 1), Point3D(2, 3, 4), Point3D(2, 2, 2))
Plane(Point3D(1, 1, 1), (-1, 2, -1))
>>> Plane((1, 1, 1), (2, 3, 4), (2, 2, 2))
Plane(Point3D(1, 1, 1), (-1, 2, -1))
>>> Plane(Point3D(1, 1, 1), normal_vector=(1,4,7))
Plane(Point3D(1, 1, 1), (1, 4, 7))
```

## Attributes

|               |  |
|---------------|--|
| p1            |  |
| normal_vector |  |

### `angle_between(o)`

Angle between the plane and other geometric entity.

#### Parameters

**LinearEntity3D, Plane.**

#### Returns

**angle** : angle in radians

## Notes

This method accepts only 3D entities as it's parameter, but if you want to calculate the angle between a 2D entity and a plane you should first convert to a 3D entity by projecting onto a desired plane and then proceed to calculate the angle.

## Examples

```
>>> from sympy import Point3D, Line3D, Plane
>>> a = Plane(Point3D(1, 2, 2), normal_vector=(1, 2, 3))
>>> b = Line3D(Point3D(1, 3, 4), Point3D(2, 2, 2))
>>> a.angle_between(b)
-asin(sqrt(21)/6)
```

### `arbitrary_point(u=None, v=None)`

Returns an arbitrary point on the Plane. If given two parameters, the point ranges over the entire plane. If given 1 or no parameters, returns a point with one parameter which, when varying from 0 to  $2\pi$ , moves the point in a circle of radius 1 about p1 of the Plane.

#### Returns

Point3D

## Examples

```
>>> from sympy import Plane, Ray
>>> from sympy.abc import u, v, t, r
>>> p = Plane((1, 1, 1), normal_vector=(1, 0, 0))
>>> p.arbitrary_point(u, v)
Point3D(1, u + 1, v + 1)
>>> p.arbitrary_point(t)
Point3D(1, cos(t) + 1, sin(t) + 1)
```

While arbitrary values of  $u$  and  $v$  can move the point anywhere in the plane, the single-parameter point can be used to construct a ray whose arbitrary point can be located at angle  $t$  and radius  $r$  from  $p.p1$ :



```
>>> Ray(p.p1, _).arbitrary_point(r)
Point3D(1, r*cos(t) + 1, r*sin(t) + 1)
```

**static are\_concurrent(\*planes)**

Is a sequence of Planes concurrent?

Two or more Planes are concurrent if their intersections are a common line.

**Parameters**

**planes:** list

**Returns**

Boolean

### Examples

```
>>> from sympy import Plane, Point3D
>>> a = Plane(Point3D(5, 0, 0), normal_vector=(1, -1, 1))
>>> b = Plane(Point3D(0, -2, 0), normal_vector=(3, 1, 1))
>>> c = Plane(Point3D(0, -1, 0), normal_vector=(5, -1, 9))
>>> Plane.are_concurrent(a, b)
True
>>> Plane.are_concurrent(a, b, c)
False
```

**distance(o)**

Distance between the plane and another geometric entity.

**Parameters**

**Point3D, LinearEntity3D, Plane.**

**Returns**

distance

### Notes

This method accepts only 3D entities as it's parameter, but if you want to calculate the distance between a 2D entity and a plane you should first convert to a 3D entity by projecting onto a desired plane and then proceed to calculate the distance.

### Examples

```
>>> from sympy import Point3D, Line3D, Plane
>>> a = Plane(Point3D(1, 1, 1), normal_vector=(1, 1, 1))
>>> b = Point3D(1, 2, 3)
>>> a.distance(b)
sqrt(3)
>>> c = Line3D(Point3D(2, 3, 1), Point3D(1, 2, 2))
>>> a.distance(c)
0
```

**equals(o)**

Returns True if self and o are the same mathematical entities.

**Examples**

```
>>> from sympy import Plane, Point3D
>>> a = Plane(Point3D(1, 2, 3), normal_vector=(1, 1, 1))
>>> b = Plane(Point3D(1, 2, 3), normal_vector=(2, 2, 2))
>>> c = Plane(Point3D(1, 2, 3), normal_vector=(-1, 4, 6))
>>> a.equals(a)
True
>>> a.equals(b)
True
>>> a.equals(c)
False
```

**equation(x=None, y=None, z=None)**

The equation of the Plane.

**Examples**

```
>>> from sympy import Point3D, Plane
>>> a = Plane(Point3D(1, 1, 2), Point3D(2, 4, 7), Point3D(3, 5, 1))
>>> a.equation()
-23*x + 11*y - 2*z + 16
>>> a = Plane(Point3D(1, 4, 2), normal_vector=(6, 6, 6))
>>> a.equation()
6*x + 6*y + 6*z - 42
```

**intersection(o)**

The intersection with other geometrical entity.

**Parameters**

**Point, Point3D, LinearEntity, LinearEntity3D, Plane**

**Returns**

List

**Examples**

```
>>> from sympy import Point3D, Line3D, Plane
>>> a = Plane(Point3D(1, 2, 3), normal_vector=(1, 1, 1))
>>> b = Point3D(1, 2, 3)
>>> a.intersection(b)
[Point3D(1, 2, 3)]
>>> c = Line3D(Point3D(1, 4, 7), Point3D(2, 2, 2))
>>> a.intersection(c)
[Point3D(2, 2, 2)]
>>> d = Plane(Point3D(6, 0, 0), normal_vector=(2, -5, 3))
```

(continues on next page)

(continued from previous page)

```
>>> e = Plane(Point3D(2, 0, 0), normal_vector=(3, 4, -3))
>>> d.intersection(e)
[Line3D(Point3D(78/23, -24/23, 0), Point3D(147/23, 321/23, 23))]
```

**is\_coplanar(*o*)**

Returns True if *o* is coplanar with self, else False.

**Examples**

```
>>> from sympy import Plane
>>> o = (0, 0, 0)
>>> p = Plane(o, (1, 1, 1))
>>> p2 = Plane(o, (2, 2, 2))
>>> p == p2
False
>>> p.is_coplanar(p2)
True
```

**is\_parallel(*l*)**

Is the given geometric entity parallel to the plane?

**Parameters**

**LinearEntity3D or Plane**

**Returns**

Boolean

**Examples**

```
>>> from sympy import Plane, Point3D
>>> a = Plane(Point3D(1,4,6), normal_vector=(2, 4, 6))
>>> b = Plane(Point3D(3,1,3), normal_vector=(4, 8, 12))
>>> a.is_parallel(b)
True
```

**is\_perpendicular(*l*)**

is the given geometric entity perpendicular to the given plane?

**Parameters**

**LinearEntity3D or Plane**

**Returns**

Boolean

### Examples

```
>>> from sympy import Plane, Point3D
>>> a = Plane(Point3D(1,4,6), normal_vector=(2, 4, 6))
>>> b = Plane(Point3D(2, 2, 2), normal_vector=(-1, 2, -1))
>>> a.is_perpendicular(b)
True
```

#### property `normal_vector`

Normal vector of the given plane.

### Examples

```
>>> from sympy import Point3D, Plane
>>> a = Plane(Point3D(1, 1, 1), Point3D(2, 3, 4), Point3D(2, 2, 2))
>>> a.normal_vector
(-1, 2, -1)
>>> a = Plane(Point3D(1, 1, 1), normal_vector=(1, 4, 7))
>>> a.normal_vector
(1, 4, 7)
```

#### property `p1`

The only defining point of the plane. Others can be obtained from the `arbitrary_point` method.

### Examples

```
>>> from sympy import Point3D, Plane
>>> a = Plane(Point3D(1, 1, 1), Point3D(2, 3, 4), Point3D(2, 2, 2))
>>> a.p1
Point3D(1, 1, 1)
```

#### See also:

[\*sympy.geometry.point.Point3D\*](#) (page 2211)

#### `parallel_plane(pt)`

Plane parallel to the given plane and passing through the point `pt`.

##### Parameters

**pt:** `Point3D`

##### Returns

`Plane`

## Examples

```
>>> from sympy import Plane, Point3D
>>> a = Plane(Point3D(1, 4, 6), normal_vector=(2, 4, 6))
>>> a.parallel_plane(Point3D(2, 3, 5))
Plane(Point3D(2, 3, 5), (2, 4, 6))
```

**parameter\_value**(*other*, *u*, *v*=None)

Return the parameter(s) corresponding to the given point.

## Examples

```
>>> from sympy import pi, Plane
>>> from sympy.abc import t, u, v
>>> p = Plane((2, 0, 0), (0, 0, 1), (0, 1, 0))
```

By default, the parameter value returned defines a point that is a distance of 1 from the Plane's `p1` value and in line with the given point:

```
>>> on_circle = p.arbitrary_point(t).subs(t, pi/4)
>>> on_circle.distance(p.p1)
1
>>> p.parameter_value(on_circle, t)
{t: pi/4}
```

Moving the point twice as far from `p1` does not change the parameter value:

```
>>> off_circle = p.p1 + (on_circle - p.p1)*2
>>> off_circle.distance(p.p1)
2
>>> p.parameter_value(off_circle, t)
{t: pi/4}
```

If the 2-value parameter is desired, supply the two parameter symbols and a replacement dictionary will be returned:

```
>>> p.parameter_value(on_circle, u, v)
{u: sqrt(10)/10, v: sqrt(10)/30}
>>> p.parameter_value(off_circle, u, v)
{u: sqrt(10)/5, v: sqrt(10)/15}
```

**perpendicular\_line**(*pt*)

A line perpendicular to the given plane.

### Parameters

**pt:** **Point3D**

### Returns

Line3D

### Examples

```
>>> from sympy import Plane, Point3D
>>> a = Plane(Point3D(1,4,6), normal_vector=(2, 4, 6))
>>> a.perpendicular_line(Point3D(9, 8, 7))
Line3D(Point3D(9, 8, 7), Point3D(11, 12, 13))
```

#### **perpendicular\_plane(\*pts)**

Return a perpendicular passing through the given points. If the direction ratio between the points is the same as the Plane's normal vector then, to select from the infinite number of possible planes, a third point will be chosen on the z-axis (or the y-axis if the normal vector is already parallel to the z-axis). If less than two points are given they will be supplied as follows: if no point is given then pt1 will be self.p1; if a second point is not given it will be a point through pt1 on a line parallel to the z-axis (if the normal is not already the z-axis, otherwise on the line parallel to the y-axis).

##### **Parameters**

**pts:** 0, 1 or 2 Point3D

##### **Returns**

Plane

### Examples

```
>>> from sympy import Plane, Point3D
>>> a, b = Point3D(0, 0, 0), Point3D(0, 1, 0)
>>> Z = (0, 0, 1)
>>> p = Plane(a, normal_vector=Z)
>>> p.perpendicular_plane(a, b)
Plane(Point3D(0, 0, 0), (1, 0, 0))
```

#### **projection(pt)**

Project the given point onto the plane along the plane normal.

##### **Parameters**

**Point or Point3D**

##### **Returns**

Point3D

### Examples

```
>>> from sympy import Plane, Point3D
>>> A = Plane(Point3D(1, 1, 2), normal_vector=(1, 1, 1))
```

The projection is along the normal vector direction, not the z axis, so (1, 1) does not project to (1, 1, 2) on the plane A:

```
>>> b = Point3D(1, 1)
>>> A.projection(b)
Point3D(5/3, 5/3, 2/3)
```

(continues on next page)

(continued from previous page)

```
>>> _ in A
True
```

But the point (1, 1, 2) projects to (1, 1) on the XY-plane:

```
>>> XY = Plane((0, 0, 0), (0, 0, 1))
>>> XY.projection((1, 1, 2))
Point3D(1, 1, 0)
```

### **projection\_line(*line*)**

Project the given line onto the plane through the normal plane containing the line.

#### **Parameters**

**LinearEntity or LinearEntity3D**

#### **Returns**

Point3D, Line3D, Ray3D or Segment3D

### **Notes**

For the interaction between 2D and 3D lines(segments, rays), you should convert the line to 3D by using this method. For example for finding the intersection between a 2D and a 3D line, convert the 2D line to a 3D line by projecting it on a required plane and then proceed to find the intersection between those lines.

### **Examples**

```
>>> from sympy import Plane, Line, Line3D, Point3D
>>> a = Plane(Point3D(1, 1, 1), normal_vector=(1, 1, 1))
>>> b = Line(Point3D(1, 1), Point3D(2, 2))
>>> a.projection_line(b)
Line3D(Point3D(4/3, 4/3, 1/3), Point3D(5/3, 5/3, -1/3))
>>> c = Line3D(Point3D(1, 1, 1), Point3D(2, 2, 2))
>>> a.projection_line(c)
Point3D(1, 1, 1)
```

### **random\_point(*seed=None*)**

Returns a random point on the Plane.

#### **Returns**

Point3D

## Examples

```
>>> from sympy import Plane
>>> p = Plane((1, 0, 0), normal_vector=(0, 1, 0))
>>> r = p.random_point(seed=42) # seed value is optional
>>> r.n(3)
Point3D(2.29, 0, -1.35)
```

The random point can be moved to lie on the circle of radius 1 centered on p1:

```
>>> c = p.p1 + (r - p.p1).unit
>>> c.distance(p.p1).equals(1)
True
```

## Holonomic

The *holonomic* (page 2312) module is intended to deal with holonomic functions along with various operations on them like addition, multiplication, composition, integration and differentiation. The module also implements various kinds of conversions such as converting holonomic functions to a different form and the other way around.

## Contents

### About Holonomic Functions

This text aims to explain holonomic functions. We assume you have a basic idea of Differential equations and Abstract algebra.

### Definition

Holonomic function is a very general type of special function that includes lots of simple known functions as its special cases. In fact the more known hypergeometric function and Meijer G-function are also a special case of it.

A function is called holonomic if it's a solution to an ordinary differential equation having polynomial coefficients only. Since the general solution of a differential equation consists of a family of functions rather than a single function, holonomic functions are usually defined by a set of initial conditions along with the differential equation.

Let  $K$  be a field of characteristic 0. For example,  $K$  can be  $\mathbb{Q}\mathbb{Q}$  or  $\mathbb{R}\mathbb{R}$ . A function  $f(x)$  will be holonomic if there exists polynomials  $p_0, p_1, p_2, \dots, p_r \in K[x]$  such that

$$p_0 \cdot f(x) + p_1 \cdot f^{(1)}(x) + p_2 \cdot f^{(2)}(x) + \dots + p_r \cdot f^{(r)}(x) = 0$$

This differential equation can also be written as  $L \cdot f(x) = 0$  where

$$L = p_0 + p_1 \cdot D + p_2 \cdot D^2 + \dots + p_r \cdot D^r$$

Here  $D$  is the Differential Operator and  $L$  is called the annihilator of the function.



A unique holonomic function can be defined from the annihilator and a set of initial conditions. For instance:

$$\begin{aligned} f(x) &= \exp(x) : L = D - 1, f(0) = 1 \\ f(x) &= \sin(x) : L = D^2 + 1, f(0) = 0, f'(0) = 1 \end{aligned}$$

Other fundamental functions such as  $\cos(x)$ ,  $\log(x)$ ,  $\text{bessel}$  functions etc. are also holonomic. The family of holonomic functions is closed under addition, multiplication, integration, composition. This means if two functions are given are holonomic, then the function resulting on applying these operation on them will also be holonomic.

## References

[https://en.wikipedia.org/wiki/Holonomic\\_function](https://en.wikipedia.org/wiki/Holonomic_function)

## Representation of holonomic functions in SymPy

Class *DifferentialOperator* (page 2315) is used to represent the annihilator but we create differential operators easily using the function *DifferentialOperators()* (page 2316). Class *HolonomicFunction* (page 2314) represents a holonomic function.

Let's explain this with an example:

Take  $\sin(x)$  for instance, the differential equation satisfied by it is  $y^{(2)}(x) + y(x) = 0$ . By definition we conclude it is a holonomic function. The general solution of this ODE is  $C_1 \cdot \sin(x) + C_2 \cdot \cos(x)$  but to get  $\sin(x)$  we need to provide initial conditions i.e.  $y(0) = 0, y^{(1)}(0) = 1$ .

To represent the same in this module one needs to provide the differential equation in the form of annihilator. Basically a differential operator is an operator on functions that differentiates them. So  $D^n \cdot y(x) = y^{(n)}(x)$  where  $y^{(n)}(x)$  denotes  $n$  times differentiation of  $y(x)$  with respect to  $x$ .

So the differential equation can also be written as  $D^2 \cdot y(x) + y(x) = 0$  or  $(D^2 + 1) \cdot y(x) = 0$ . The part left of  $y(x)$  is the annihilator i.e.  $D^2 + 1$ .

So this is how one will represent  $\sin(x)$  as a Holonomic Function:

```
>>> from sympy.holonomic import DifferentialOperators, HolonomicFunction
>>> from sympy.abc import x
>>> from sympy import ZZ
>>> R, D = DifferentialOperators(ZZ.old_poly_ring(x), 'D')
>>> HolonomicFunction(D**2 + 1, x, 0, [0, 1])
HolonomicFunction((1) + (1)*D**2, x, 0, [0, 1])
```

The polynomial coefficients will be members of the ring  $\mathbb{Z}[x]$  in the example. The  $D$  operator returned by the function *DifferentialOperators()* (page 2316) can be used to create annihilators just like SymPy expressions. We currently use the older implementations of rings in SymPy for priority mechanism.

## HolonomicFunction

**class** sympy.holonomic.holonomic.HolonomicFunction(annihilator, x, x0=0, y0=None)

A Holonomic Function is a solution to a linear homogeneous ordinary differential equation with polynomial coefficients. This differential equation can also be represented by an annihilator i.e. a Differential Operator  $L$  such that  $L.f = 0$ . For uniqueness of these functions, initial conditions can also be provided along with the annihilator.

### Explanation

Holonomic functions have closure properties and thus forms a ring. Given two Holonomic Functions  $f$  and  $g$ , their sum, product, integral and derivative is also a Holonomic Function.

For ordinary points initial condition should be a vector of values of the derivatives i.e.  $[y(x_0), y'(x_0), y''(x_0), \dots]$ .

For regular singular points initial conditions can also be provided in this format:  $s_0 : [C_0, C_1, \dots], s_1 : [C_0^1, C_1^1, \dots], \dots$  where  $s_0, s_1, \dots$  are the roots of indicial equation and vectors  $[C_0, C_1, \dots], [C_0^0, C_1^0, \dots], \dots$  are the corresponding initial terms of the associated power series. See Examples below.

### Examples

```
>>> from sympy.holonomic.holonomic import HolonomicFunction, DifferentialOperators
>>> from sympy import QQ
>>> from sympy import symbols, S
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(QQ.old_poly_ring(x), 'Dx')
```

```
>>> p = HolonomicFunction(Dx - 1, x, 0, [1]) # e^x
>>> q = HolonomicFunction(Dx**2 + 1, x, 0, [0, 1]) # sin(x)
```

```
>>> p + q # annihilator of e^x + sin(x)
HolonomicFunction((-1) + (1)*Dx + (-1)*Dx**2 + (1)*Dx**3, x, 0, [1, 2, 1])
```

```
>>> p * q # annihilator of e^x * sin(x)
HolonomicFunction((2) + (-2)*Dx + (1)*Dx**2, x, 0, [0, 1])
```

An example of initial conditions for regular singular points, the indicial equation has only one root  $1/2$ .

```
>>> HolonomicFunction(-S(1)/2 + x*Dx, x, 0, {S(1)/2: [1]})
HolonomicFunction((-1/2) + (x)*Dx, x, 0, {1/2: [1]})
```

```
>>> HolonomicFunction(-S(1)/2 + x*Dx, x, 0, {S(1)/2: [1]}).to_expr()
sqrt(x)
```

To plot a Holonomic Function, one can use `.evalf()` for numerical computation. Here's an example on  $\sin(x) ** 2 / x$  using numpy and matplotlib.

```
>>> import sympy.holonomic
>>> from sympy import var, sin
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> var("x")
>>> r = np.linspace(1, 5, 100)
>>> y = sympy.holonomic.expr_to_holonomic(sin(x)**2/x, x0=1).evalf(r)
>>> plt.plot(r, y, label="holonomic function")
>>> plt.show()
```

## DifferentialOperator

**class** `sympy.holonomic.holonomic.DifferentialOperator(list_of_poly, parent)`

Differential Operators are elements of Weyl Algebra. The Operators are defined by a list of polynomials in the base ring and the parent ring of the Operator i.e. the algebra it belongs to.

### Explanation

Takes a list of polynomials for each power of Dx and the parent ring which must be an instance of `DifferentialOperatorAlgebra`.

A Differential Operator can be created easily using the operator Dx. See examples below.

### Examples

```
>>> from sympy.holonomic.holonomic import DifferentialOperator, _
↳ DifferentialOperators
>>> from sympy import ZZ
>>> from sympy import symbols
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(ZZ.old_poly_ring(x), 'Dx')
```

```
>>> DifferentialOperator([0, 1, x**2], R)
(1)*Dx + (x**2)*Dx**2
```

```
>>> (x*Dx*x + 1 - Dx**2)**2
(2*x**2 + 2*x + 1) + (4*x**3 + 2*x**2 - 4)*Dx + (x**4 - 6*x - 2)*Dx**2 + _
↳ (-2*x**2)*Dx**3 + (1)*Dx**4
```

**See also:**

[`DifferentialOperatorAlgebra`](#) (page 2316)

**is\_singular(x0)**

Checks if the differential equation is singular at x0.

## DifferentialOperators

`sympy.holonomic.holonomic.DifferentialOperators(base, generator)`

This function is used to create annihilators using Dx.

### Parameters

#### base:

Base polynomial ring for the algebra. The base polynomial ring is the ring of polynomials in  $x$  that will appear as coefficients in the operators.

#### generator:

Generator of the algebra which can be either a noncommutative Symbol or a string. e.g. "Dx" or "D".

## Explanation

Returns an Algebra of Differential Operators also called Weyl Algebra and the operator for differentiation i.e. the Dx operator.

## Examples

```
>>> from sympy import ZZ
>>> from sympy.abc import x
>>> from sympy.holonomic.holonomic import DifferentialOperators
>>> R, Dx = DifferentialOperators(ZZ.old_poly_ring(x), 'Dx')
>>> R
Univariate Differential Operator Algebra in intermediate Dx over the
↳base ring ZZ[x]
>>> Dx*x
(1) + (x)*Dx
```

## DifferentialOperatorAlgebra

`class sympy.holonomic.holonomic.DifferentialOperatorAlgebra(base, generator)`

An Ore Algebra is a set of noncommutative polynomials in the intermediate Dx and coefficients in a base polynomial ring  $A$ . It follows the commutation rule:

$$Dxa = \sigma(a)Dx + \delta(a)$$

for  $a \in A$ .

Where  $\sigma : A \Rightarrow A$  is an endomorphism and  $\delta : A \rightarrow A$  is a skew-derivation i.e.  $\delta(ab) = \delta(a)b + \sigma(a)\delta(b)$ .

If one takes the sigma as identity map and delta as the standard derivation then it becomes the algebra of Differential Operators also called a Weyl Algebra i.e. an algebra whose elements are Differential Operators.

This class represents a Weyl Algebra and serves as the parent ring for Differential Operators.

## Examples

```
>>> from sympy import ZZ
>>> from sympy import symbols
>>> from sympy.holonomic.holonomic import DifferentialOperators
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(ZZ.old_poly_ring(x), 'Dx')
>>> R
Univariate Differential Operator Algebra in intermediate Dx over the
↳base ring
ZZ[x]
```

### See also:

[\*DifferentialOperator\*](#) (page 2315)

## Operations on holonomic functions

### Addition and Multiplication

Two holonomic functions can be added or multiplied with the result also a holonomic functions.

```
>>> from sympy.holonomic.holonomic import HolonomicFunction,
↳DifferentialOperators
>>> from sympy.polys.domains import QQ
>>> from sympy import symbols
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(QQ.old_poly_ring(x), 'Dx')
```

$p$  and  $q$  here are holonomic representation of  $e^x$  and  $\sin(x)$  respectively.

```
>>> p = HolonomicFunction(Dx - 1, x, 0, [1])
>>> q = HolonomicFunction(Dx**2 + 1, x, 0, [0, 1])
```

Holonomic representation of  $e^x + \sin(x)$

```
>>> p + q
HolonomicFunction((-1) + (1)*Dx + (-1)*Dx**2 + (1)*Dx**3, x, 0, [1,
↳2, 1])
```

Holonomic representation of  $e^x \cdot \sin(x)$

```
>>> p * q
HolonomicFunction((2) + (-2)*Dx + (1)*Dx**2, x, 0, [0, 1])
```

## Integration and Differentiation

`HolonomicFunction.integrate(limits, initcond=False)`

Integrates the given holonomic function.

### Examples

```
>>> from sympy.holonomic.holonomic import HolonomicFunction, \
↳ DifferentialOperators
>>> from sympy import QQ
>>> from sympy import symbols
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(QQ.old_poly_ring(x), 'Dx')
>>> HolonomicFunction(Dx - 1, x, 0, [1]).integrate((x, 0, x)) # e^x - 1
HolonomicFunction((-1)*Dx + (1)*Dx**2, x, 0, [0, 1])
>>> HolonomicFunction(Dx**2 + 1, x, 0, [1, 0]).integrate((x, 0, x))
HolonomicFunction((1)*Dx + (1)*Dx**3, x, 0, [0, 1, 0])
```

`HolonomicFunction.diff(*args, **kwargs)`

Differentiation of the given Holonomic function.

### Examples

```
>>> from sympy.holonomic.holonomic import HolonomicFunction, \
↳ DifferentialOperators
>>> from sympy import ZZ
>>> from sympy import symbols
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(ZZ.old_poly_ring(x), 'Dx')
>>> HolonomicFunction(Dx**2 + 1, x, 0, [0, 1]).diff().to_expr()
cos(x)
>>> HolonomicFunction(Dx - 2, x, 0, [1]).diff().to_expr()
2*exp(2*x)
```

**See also:**

[\*integrate\*](#) (page 2318)

## Composition with polynomials

`HolonomicFunction.composition(expr, *args, **kwargs)`

Returns function after composition of a holonomic function with an algebraic function. The method cannot compute initial conditions for the result by itself, so they can be also be provided.

## Examples

```
>>> from sympy.holonomic.holonomic import HolonomicFunction, ␣
↪ DifferentialOperators
>>> from sympy import QQ
>>> from sympy import symbols
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(QQ.old_poly_ring(x), 'Dx')
>>> HolonomicFunction(Dx - 1, x).composition(x**2, 0, [1]) # e^(x**2)
HolonomicFunction((-2*x) + (1)*Dx, x, 0, [1])
>>> HolonomicFunction(Dx**2 + 1, x).composition(x**2 - 1, 1, [1, 0])
HolonomicFunction((4*x**3) + (-1)*Dx + (x)*Dx**2, x, 1, [1, 0])
```

### See also:

[\*from\\_hyper\*](#) (page 2323)

## Convert to holonomic sequence

`HolonomicFunction.to_sequence(lb=True)`

Finds recurrence relation for the coefficients in the series expansion of the function about  $x_0$ , where  $x_0$  is the point at which the initial condition is stored.

### Explanation

If the point  $x_0$  is ordinary, solution of the form  $[(R, n_0)]$  is returned. Where  $R$  is the recurrence relation and  $n_0$  is the smallest  $n$  for which the recurrence holds true.

If the point  $x_0$  is regular singular, a list of solutions in the format  $(R, p, n_0)$  is returned, i.e.  $[(R, p, n_0), \dots]$ . Each tuple in this vector represents a recurrence relation  $R$  associated with a root of the indicial equation  $p$ . Conditions of a different format can also be provided in this case, see the docstring of `HolonomicFunction` class.

If it's not possible to numerically compute a initial condition, it is returned as a symbol  $C_j$ , denoting the coefficient of  $(x - x_0)^j$  in the power series about  $x_0$ .

## Examples

```
>>> from sympy.holonomic.holonomic import HolonomicFunction, ␣
↪ DifferentialOperators
>>> from sympy import QQ
>>> from sympy import symbols, S
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(QQ.old_poly_ring(x), 'Dx')
>>> HolonomicFunction(Dx - 1, x, 0, [1]).to_sequence()
[(HolonomicSequence((-1) + (n + 1)Sn, n), u(0) = 1, 0)]
>>> HolonomicFunction((1 + x)*Dx**2 + Dx, x, 0, [0, 1]).to_sequence()
[(HolonomicSequence((n**2) + (n**2 + n)Sn, n), u(0) = 0, u(1) = 1, u(2) ␣
↪ = -1/2, 2)]
>>> HolonomicFunction(-S(1)/2 + x*Dx, x, 0, {S(1)/2: [1]}).to_sequence()
[(HolonomicSequence((n), n), u(0) = 1, 1/2, 1)]
```

**See also:**

[\*HolonomicFunction.series\*](#) (page 2320)

## References

[R523], [R524]

## Series expansion

`HolonomicFunction.series(n=6, coefficient=False, order=True, _recur=None)`

Finds the power series expansion of given holonomic function about  $x_0$ .

## Explanation

A list of series might be returned if  $x_0$  is a regular point with multiple roots of the indicial equation.

## Examples

```
>>> from sympy.holonomic.holonomic import HolonomicFunction, \
↳ DifferentialOperators
>>> from sympy import QQ
>>> from sympy import symbols
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(QQ.old_poly_ring(x), 'Dx')
>>> HolonomicFunction(Dx - 1, x, 0, [1]).series() # e^x
1 + x + x**2/2 + x**3/6 + x**4/24 + x**5/120 + O(x**6)
>>> HolonomicFunction(Dx**2 + 1, x, 0, [0, 1]).series(n=8) # sin(x)
x - x**3/6 + x**5/120 - x**7/5040 + O(x**8)
```

**See also:**

[\*HolonomicFunction.to\\_sequence\*](#) (page 2319)

## Numerical evaluation

`HolonomicFunction.evalf(points, method='RK4', h=0.05, derivatives=False)`

Finds numerical value of a holonomic function using numerical methods. (RK4 by default). A set of points (real or complex) must be provided which will be the path for the numerical integration.



## Explanation

The path should be given as a list  $[x_1, x_2, \dots, x_n]$ . The numerical values will be computed at each point in this order  $x_1 \rightarrow x_2 \rightarrow x_3 \cdots \rightarrow x_n$ .

Returns values of the function at  $x_1, x_2, \dots, x_n$  in a list.

## Examples

```
>>> from sympy.holonomic.holonomic import HolonomicFunction, \
↳ DifferentialOperators
>>> from sympy import QQ
>>> from sympy import symbols
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(QQ.old_poly_ring(x), 'Dx')
```

A straight line on the real axis from (0 to 1)

```
>>> r = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]
```

Runge-Kutta 4th order on  $e^x$  from 0.1 to 1. Exact solution at 1 is 2.71828182845905

```
>>> HolonomicFunction(Dx - 1, x, 0, [1]).evalf(r)
[1.10517083333333, 1.22140257085069, 1.34985849706254, 1.49182424008069,
1.64872063859684, 1.82211796209193, 2.01375162659678, 2.22553956329232,
2.45960141378007, 2.71827974413517]
```

Euler's method for the same

```
>>> HolonomicFunction(Dx - 1, x, 0, [1]).evalf(r, method='Euler')
[1.1, 1.21, 1.331, 1.4641, 1.61051, 1.771561, 1.9487171, 2.1435881,
2.357947691, 2.5937424601]
```

One can also observe that the value obtained using Runge-Kutta 4th order is much more accurate than Euler's method.

## Convert to a linear combination of hypergeometric functions

`HolonomicFunction.to_hyper(as_list=False, _recur=None)`

Returns a hypergeometric function (or linear combination of them) representing the given holonomic function.

## Explanation

Returns an answer of the form:  $a_1 \cdot x^{b_1} \cdot \text{hyper}() + a_2 \cdot x^{b_2} \cdot \text{hyper}() \dots$

This is very useful as one can now use `hyperexpand` to find the symbolic expressions/functions.

## Examples

```
>>> from sympy.holonomic.holonomic import HolonomicFunction,
↳ DifferentialOperators
>>> from sympy import ZZ
>>> from sympy import symbols
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(ZZ.old_poly_ring(x), 'Dx')
>>> # sin(x)
>>> HolonomicFunction(Dx**2 + 1, x, 0, [0, 1]).to_hyper()
x*hyper(), (3/2,), -x**2/4)
>>> # exp(x)
>>> HolonomicFunction(Dx - 1, x, 0, [1]).to_hyper()
hyper(), (), x)
```

### See also:

[`from\_hyper`](#) (page 2323), [`from\_meijerg`](#) (page 2323)

## Convert to a linear combination of Meijer G-functions

`HolonomicFunction.to_meijerg()`

Returns a linear combination of Meijer G-functions.

## Examples

```
>>> from sympy.holonomic import expr_to_holonomic
>>> from sympy import sin, cos, hyperexpand, log, symbols
>>> x = symbols('x')
>>> hyperexpand(expr_to_holonomic(cos(x) + sin(x)).to_meijerg())
sin(x) + cos(x)
>>> hyperexpand(expr_to_holonomic(log(x)).to_meijerg()).simplify()
log(x)
```

### See also:

[`to\_hyper`](#) (page 2321)

## Convert to expressions

`HolonomicFunction.to_expr()`

Converts a Holonomic Function back to elementary functions.

### Examples

```
>>> from sympy.holonomic.holonomic import HolonomicFunction, _
↳ DifferentialOperators
>>> from sympy import ZZ
>>> from sympy import symbols, S
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(ZZ.old_poly_ring(x), 'Dx')
>>> HolonomicFunction(x**2*Dx**2 + x*Dx + (x**2 - 1), x, 0, [0, S(1)/2]).
↳ to_expr()
besselj(1, x)
>>> HolonomicFunction((1 + x)*Dx**3 + Dx**2, x, 0, [1, 1, 1]).to_expr()
x*log(x + 1) + log(x + 1) + 1
```

## Converting other representations to holonomic

### Converting hypergeometric functions

`sympy.holonomic.holonomic.from_hyper(func, x0=0, evalf=False)`

Converts a hypergeometric function to holonomic. `func` is the Hypergeometric Function and `x0` is the point at which initial conditions are required.

### Examples

```
>>> from sympy.holonomic.holonomic import from_hyper
>>> from sympy import symbols, hyper, S
>>> x = symbols('x')
>>> from_hyper(hyper([], [S(3)/2], x**2/4))
HolonomicFunction((-x) + (2)*Dx + (x)*Dx**2, x, 1, [sinh(1), -sinh(1) + _
↳ cosh(1)])
```

## Converting Meijer G-functions

`sympy.holonomic.holonomic.from_meijerg(func, x0=0, evalf=False, initcond=True, domain=QQ)`

Converts a Meijer G-function to Holonomic. `func` is the G-Function and `x0` is the point at which initial conditions are required.

## Examples

```
>>> from sympy.holonomic.holonomic import from_meijerg
>>> from sympy import symbols, meijerg, S
>>> x = symbols('x')
>>> from_meijerg(meijerg([], []), ([S(1)/2], [0]), x**2/4))
HolonomicFunction((1) + (1)*Dx**2, x, 0, [0, 1/sqrt(pi)])
```

## Converting symbolic expressions

`sympy.holonomic.holonomic.expr_to_holonomic`(*func*, *x=None*, *x0=0*, *y0=None*,  
*lenics=None*, *domain=None*,  
*initcond=True*)

Converts a function or an expression to a holonomic function.

### Parameters

#### **func:**

The expression to be converted.

#### **x:**

variable for the function.

#### **x0:**

point at which initial condition must be computed.

#### **y0:**

One can optionally provide initial condition if the method is not able to do it automatically.

#### **lenics:**

Number of terms in the initial condition. By default it is equal to the order of the annihilator.

#### **domain:**

Ground domain for the polynomials in *x* appearing as coefficients in the annihilator.

#### **initcond:**

Set it false if you do not want the initial conditions to be computed.

## Examples

```
>>> from sympy.holonomic.holonomic import expr_to_holonomic
>>> from sympy import sin, exp, symbols
>>> x = symbols('x')
>>> expr_to_holonomic(sin(x))
HolonomicFunction((1) + (1)*Dx**2, x, 0, [0, 1])
>>> expr_to_holonomic(exp(x))
HolonomicFunction((-1) + (1)*Dx, x, 0, [1])
```

**See also:**

`sympy.integrals.meijerint._rewritel` (page 572), `_convert_poly_rat_alg` (page 2326), `_create_table` (page 2326)

**Uses and Current limitations****Integration**

One can perform integrations using holonomic functions by following these steps:

1. Convert the integrand to a holonomic function.
2. Now integrate the holonomic representation of the function.
3. Convert the integral back to expressions.

**Examples**

```
>>> from sympy.abc import x, a
>>> from sympy import sin
>>> from sympy.holonomic import expr_to_holonomic
>>> expr_to_holonomic(1/(x**2+a), x).integrate(x).to_expr()
atan(x/sqrt(a))/sqrt(a)
>>> expr_to_holonomic(sin(x)/x).integrate(x).to_expr()
Si(x)
```

As you can see in the first example we converted the function to holonomic, integrated the result and then converted back to symbolic expression.

**Limitations**

1. Converting to expressions is not always possible. The holonomic function should have a hypergeometric series at  $x_0$ .
2. Implementation of converting to holonomic sequence currently doesn't support Frobenius method when the solutions need to have log terms. This happens when at least one pair of the roots of the indicial equation differ by an integer and frobenius method yields linearly dependent series solutions. Since we use this while converting to expressions, sometimes `to_expr()` (page 2323) fails.
3. There doesn't seem to be a way for computing indefinite integrals, so `integrate()` (page 2318) basically computes  $\int_{x_0}^x f(x)dx$  if no limits are given, where  $x_0$  is the point at which initial conditions for the integrand are stored. Sometimes this gives an additional constant in the result. For instance:

```
>>> expr_to_holonomic(sin(x)).integrate(x).to_expr()
1 - cos(x)
>>> sin(x).integrate(x)
-cos(x)
```

The indefinite integral of  $\sin(x)$  is  $-\cos(x)$ . But the output is  $-\cos(x) + 1$  which is  $\int_0^x \sin(x)dx$ . Although both are considered correct but  $-\cos(x)$  is simpler.

## Internal API

`sympy.holonomic.holonomic._create_table(table, domain=QQ)`

Creates the look-up table. For a similar implementation see `meijerint._create_lookup_table`.

`sympy.holonomic.holonomic._convert_poly_rat_alg(func, x, x0=0, y0=None, lenics=None, domain=QQ, initcond=True)`

Converts polynomials, rationals and algebraic functions to holonomic.

## Lie Algebra

**class** `sympy.liealgebras.root_system.RootSystem(cartantype)`

Represent the root system of a simple Lie algebra

Every simple Lie algebra has a unique root system. To find the root system, we first consider the Cartan subalgebra of  $\mathfrak{g}$ , which is the maximal abelian subalgebra, and consider the adjoint action of  $\mathfrak{g}$  on this subalgebra. There is a root system associated with this action. Now, a root system over a vector space  $V$  is a set of finite vectors  $\Phi$  (called roots), which satisfy:

1. The roots span  $V$
2. The only scalar multiples of  $x$  in  $\Phi$  are  $x$  and  $-x$
3. For every  $x$  in  $\Phi$ , the set  $\Phi$  is closed under reflection through the hyperplane perpendicular to  $x$ .
4. If  $x$  and  $y$  are roots in  $\Phi$ , then the projection of  $y$  onto the line through  $x$  is a half-integral multiple of  $x$ .

Now, there is a subset of  $\Phi$ , which we will call  $\Delta$ , such that:

1.  $\Delta$  is a basis of  $V$
2. Each root  $x$  in  $\Phi$  can be written  $x = \sum k_y y$  for  $y$  in  $\Delta$

The elements of  $\Delta$  are called the simple roots. Therefore, we see that the simple roots span the root space of a given simple Lie algebra.

## References

[R555], [R556]

**add\_as\_roots**(*root1*, *root2*)

Add two roots together if and only if their sum is also a root

It takes as input two vectors which should be roots. It then computes their sum and checks if it is in the list of all possible roots. If it is, it returns the sum. Otherwise it returns a string saying that the sum is not a root.

## Examples

```
>>> from sympy.liealgebras.root_system import RootSystem
>>> c = RootSystem("A3")
>>> c.add_as_roots([1, 0, -1, 0], [0, 0, 1, -1])
[1, 0, 0, -1]
>>> c.add_as_roots([1, -1, 0, 0], [0, 0, -1, 1])
'The sum of these two roots is not a root'
```

### `add_simple_roots(root1, root2)`

Add two simple roots together

The function takes as input two integers, `root1` and `root2`. It then uses these integers as keys in the dictionary of simple roots, and gets the corresponding simple roots, and then adds them together.

## Examples

```
>>> from sympy.liealgebras.root_system import RootSystem
>>> c = RootSystem("A3")
>>> newroot = c.add_simple_roots(1, 2)
>>> newroot
[1, 0, -1, 0]
```

### `all_roots()`

Generate all the roots of a given root system

The result is a dictionary where the keys are integer numbers. It generates the roots by getting the dictionary of all positive roots from the bases classes, and then taking each root, and multiplying it by -1 and adding it to the dictionary. In this way all the negative roots are generated.

### `cartan_matrix()`

Cartan matrix of Lie algebra associated with this root system

## Examples

```
>>> from sympy.liealgebras.root_system import RootSystem
>>> c = RootSystem("A3")
>>> c.cartan_matrix()
Matrix([
 [2, -1, 0],
 [-1, 2, -1],
 [0, -1, 2]])
```

### `dynkin_diagram()`

Dynkin diagram of the Lie algebra associated with this root system

### Examples

```
>>> from sympy.liealgebras.root_system import RootSystem
>>> c = RootSystem("A3")
>>> print(c.dynkin_diagram())
0---0---0
1 2 3
```

#### **root\_space()**

Return the span of the simple roots

The root space is the vector space spanned by the simple roots, i.e. it is a vector space with a distinguished basis, the simple roots. This method returns a string that represents the root space as the span of the simple roots,  $\alpha[1], \dots, \alpha[n]$ .

### Examples

```
>>> from sympy.liealgebras.root_system import RootSystem
>>> c = RootSystem("A3")
>>> c.root_space()
'alpha[1] + alpha[2] + alpha[3]'
```

#### **simple\_roots()**

Generate the simple roots of the Lie algebra

The rank of the Lie algebra determines the number of simple roots that it has. This method obtains the rank of the Lie algebra, and then uses the `simple_root` method from the Lie algebra classes to generate all the simple roots.

### Examples

```
>>> from sympy.liealgebras.root_system import RootSystem
>>> c = RootSystem("A3")
>>> roots = c.simple_roots()
>>> roots
{1: [1, -1, 0, 0], 2: [0, 1, -1, 0], 3: [0, 0, 1, -1]}
```

#### **class sympy.liealgebras.type\_a.TypeA(n)**

This class contains the information about the A series of simple Lie algebras. ===

##### **basic\_root(i, j)**

This is a method just to generate roots with a 1 in the  $i$ th position and a -1 in the  $j$ th position.

##### **basis()**

Returns the number of independent generators of  $A_n$

##### **cartan\_matrix()**

Returns the Cartan matrix for  $A_n$ . The Cartan matrix for a Lie algebra is generated by assigning an ordering to the simple roots,  $(\alpha[1], \dots, \alpha[l])$ . Then the  $ij$ th entry of the Cartan matrix is  $(\langle \alpha[i], \alpha[j] \rangle)$ .



### Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType('A4')
>>> c.cartan_matrix()
Matrix([
[2, -1, 0, 0],
[-1, 2, -1, 0],
[0, -1, 2, -1],
[0, 0, -1, 2]])
```

#### **dimension()**

Dimension of the vector space  $V$  underlying the Lie algebra

### Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A4")
>>> c.dimension()
5
```

#### **highest\_root()**

Returns the highest weight root for  $A_n$

#### **lie\_algebra()**

Returns the Lie algebra associated with  $A_n$

#### **positive\_roots()**

This method generates all the positive roots of  $A_n$ . This is half of all of the roots of  $A_n$ ; by multiplying all the positive roots by -1 we get the negative roots.

### Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A3")
>>> c.positive_roots()
{1: [1, -1, 0, 0], 2: [1, 0, -1, 0], 3: [1, 0, 0, -1], 4: [0, 1, -1, 0],
 5: [0, 1, 0, -1], 6: [0, 0, 1, -1]}
```

#### **roots()**

Returns the total number of roots for  $A_n$

#### **simple\_root(i)**

Every lie algebra has a unique root system. Given a root system  $Q$ , there is a subset of the roots such that an element of  $Q$  is called a simple root if it cannot be written as the sum of two elements in  $Q$ . If we let  $D$  denote the set of simple roots, then it is clear that every element of  $Q$  can be written as a linear combination of elements of  $D$  with all coefficients non-negative.

In  $A_n$  the  $i$ th simple root is the root which has a 1 in the  $i$ th position, a -1 in the  $(i+1)$ th position, and zeroes elsewhere.

This method returns the  $i$ th simple root for the A series.

### Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A4")
>>> c.simple_root(1)
[1, -1, 0, 0, 0]
```

**class** sympy.liealgebras.type\_b.TypeB( $n$ )

**basic\_root**( $i, j$ )

This is a method just to generate roots with a 1 in the  $i$ th position and a -1 in the  $j$ th position.

**basis**()

Returns the number of independent generators of  $B_n$

**cartan\_matrix**()

Returns the Cartan matrix for  $B_n$ . The Cartan matrix for a Lie algebra is generated by assigning an ordering to the simple roots,  $(\alpha[1], \dots, \alpha[l])$ . Then the  $ij$ th entry of the Cartan matrix is  $\langle \alpha[i], \alpha[j] \rangle$ .

### Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType('B4')
>>> c.cartan_matrix()
Matrix([
[2, -1, 0, 0],
[-1, 2, -1, 0],
[0, -1, 2, -2],
[0, 0, -1, 2]])
```

**dimension**()

Dimension of the vector space  $V$  underlying the Lie algebra

### Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("B3")
>>> c.dimension()
3
```

**lie\_algebra**()

Returns the Lie algebra associated with  $B_n$

**positive\_roots**()

This method generates all the positive roots of  $A_n$ . This is half of all of the roots of  $B_n$ ; by multiplying all the positive roots by -1 we get the negative roots.

## Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A3")
>>> c.positive_roots()
{1: [1, -1, 0, 0], 2: [1, 0, -1, 0], 3: [1, 0, 0, -1], 4: [0, 1, -1, 0],
 5: [0, 1, 0, -1], 6: [0, 0, 1, -1]}
```

### roots()

Returns the total number of roots for B<sub>n</sub>

### simple\_root(i)

Every lie algebra has a unique root system. Given a root system Q, there is a subset of the roots such that an element of Q is called a simple root if it cannot be written as the sum of two elements in Q. If we let D denote the set of simple roots, then it is clear that every element of Q can be written as a linear combination of elements of D with all coefficients non-negative.

In B<sub>n</sub> the first n-1 simple roots are the same as the roots in A<sub>(n-1)</sub> (a 1 in the i<sup>th</sup> position, a -1 in the (i+1)<sup>th</sup> position, and zeroes elsewhere). The n-th simple root is the root with a 1 in the n<sup>th</sup> position and zeroes elsewhere.

This method returns the i<sup>th</sup> simple root for the B series.

## Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("B3")
>>> c.simple_root(2)
[0, 1, -1]
```

**class** sympy.liealgebras.type\_c.TypeC(n)

### basic\_root(i, j)

Generate roots with 1 in i<sup>th</sup> position and a -1 in j<sup>th</sup> position

### basis()

Returns the number of independent generators of C<sub>n</sub>

### cartan\_matrix()

The Cartan matrix for C<sub>n</sub>

The Cartan matrix matrix for a Lie algebra is generated by assigning an ordering to the simple roots, (alpha[1], ..., alpha[l]). Then the ij<sup>th</sup> entry of the Cartan matrix is (α[i], α[j]).

### Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType('C4')
>>> c.cartan_matrix()
Matrix([
[2, -1, 0, 0],
[-1, 2, -1, 0],
[0, -1, 2, -1],
[0, 0, -2, 2]])
```

#### **dimension()**

Dimension of the vector space  $V$  underlying the Lie algebra

### Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("C3")
>>> c.dimension()
3
```

#### **lie\_algebra()**

Returns the Lie algebra associated with  $C_n$

#### **positive\_roots()**

Generates all the positive roots of  $A_n$

This is half of all of the roots of  $C_n$ ; by multiplying all the positive roots by -1 we get the negative roots.

### Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A3")
>>> c.positive_roots()
{1: [1, -1, 0, 0], 2: [1, 0, -1, 0], 3: [1, 0, 0, -1], 4: [0, 1, -1, 0],
 5: [0, 1, 0, -1], 6: [0, 0, 1, -1]}
```

#### **roots()**

Returns the total number of roots for  $C_n$

#### **simple\_root(i)**

The  $i$ th simple root for the  $C$  series

Every lie algebra has a unique root system. Given a root system  $Q$ , there is a subset of the roots such that an element of  $Q$  is called a simple root if it cannot be written as the sum of two elements in  $Q$ . If we let  $D$  denote the set of simple roots, then it is clear that every element of  $Q$  can be written as a linear combination of elements of  $D$  with all coefficients non-negative.

In  $C_n$ , the first  $n-1$  simple roots are the same as the roots in  $A_{n-1}$  (a 1 in the  $i$ th position, a -1 in the  $(i+1)$ th position, and zeroes elsewhere). The  $n$ th simple root is the root in which there is a 2 in the  $n$ th position and zeroes elsewhere.

### Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("C3")
>>> c.simple_root(2)
[0, 1, -1]
```

**class** sympy.liealgebras.type\_d.TypeD( $n$ )

**basic\_root**( $i, j$ )

This is a method just to generate roots with a 1 in the  $i$ th position and a -1 in the  $j$ th position.

**basis**()

Returns the number of independent generators of  $D_n$

**cartan\_matrix**()

Returns the Cartan matrix for  $D_n$ . The Cartan matrix for a Lie algebra is generated by assigning an ordering to the simple roots,  $(\alpha[1], \dots, \alpha[l])$ . Then the  $ij$ th entry of the Cartan matrix is  $(\langle \alpha[i], \alpha[j] \rangle)$ .

### Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType('D4')
>>> c.cartan_matrix()
Matrix([
 [2, -1, 0, 0],
 [-1, 2, -1, -1],
 [0, -1, 2, 0],
 [0, -1, 0, 2]])
```

**dimension**()

Dimension of the vector space  $V$  underlying the Lie algebra

### Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("D4")
>>> c.dimension()
4
```

**lie\_algebra**()

Returns the Lie algebra associated with  $D_n$

**positive\_roots()**

This method generates all the positive roots of  $A_n$ . This is half of all of the roots of  $D_n$  by multiplying all the positive roots by -1 we get the negative roots.

**Examples**

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A3")
>>> c.positive_roots()
{1: [1, -1, 0, 0], 2: [1, 0, -1, 0], 3: [1, 0, 0, -1], 4: [0, 1, -1, 0],
 5: [0, 1, 0, -1], 6: [0, 0, 1, -1]}
```

**roots()**

Returns the total number of roots for  $D_n$

**simple\_root(i)**

Every lie algebra has a unique root system. Given a root system  $Q$ , there is a subset of the roots such that an element of  $Q$  is called a simple root if it cannot be written as the sum of two elements in  $Q$ . If we let  $D$  denote the set of simple roots, then it is clear that every element of  $Q$  can be written as a linear combination of elements of  $D$  with all coefficients non-negative.

In  $D_n$ , the first  $n-1$  simple roots are the same as the roots in  $A_{(n-1)}$  (a 1 in the  $i$ th position, a -1 in the  $(i+1)$ th position, and zeroes elsewhere). The  $n$ th simple root is the root in which there is 1 in the  $n$ th and  $(n-1)$ th positions, and zeroes elsewhere.

This method returns the  $i$ th simple root for the  $D$  series.

**Examples**

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("D4")
>>> c.simple_root(2)
[0, 1, -1, 0]
```

**class sympy.liealgebras.type\_e.TypeE(n)****basic\_root(i, j)**

This is a method just to generate roots with a -1 in the  $i$ th position and a 1 in the  $j$ th position.

**basis()**

Returns the number of independent generators of  $E_n$

**cartan\_matrix()**

Returns the Cartan matrix for  $G_2$ . The Cartan matrix for a Lie algebra is generated by assigning an ordering to the simple roots,  $(\alpha[1], \dots, \alpha[l])$ . Then the  $ij$ th entry of the Cartan matrix is  $\langle \alpha[i], \alpha[j] \rangle$ .

### Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType('A4')
>>> c.cartan_matrix()
Matrix([
[2, -1, 0, 0],
[-1, 2, -1, 0],
[0, -1, 2, -1],
[0, 0, -1, 2]])
```

#### **dimension()**

Dimension of the vector space  $V$  underlying the Lie algebra

### Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("E6")
>>> c.dimension()
8
```

#### **positive\_roots()**

This method generates all the positive roots of  $A_n$ . This is half of all of the roots of  $E_n$ ; by multiplying all the positive roots by  $-1$  we get the negative roots.

### Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A3")
>>> c.positive_roots()
{1: [1, -1, 0, 0], 2: [1, 0, -1, 0], 3: [1, 0, 0, -1], 4: [0, 1, -1, 0],
 5: [0, 1, 0, -1], 6: [0, 0, 1, -1]}
```

#### **roots()**

Returns the total number of roots of  $E_n$

#### **simple\_root(i)**

Every lie algebra has a unique root system. Given a root system  $Q$ , there is a subset of the roots such that an element of  $Q$  is called a simple root if it cannot be written as the sum of two elements in  $Q$ . If we let  $D$  denote the set of simple roots, then it is clear that every element of  $Q$  can be written as a linear combination of elements of  $D$  with all coefficients non-negative.

This method returns the  $i$ th simple root for  $E_n$ .

### Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("E6")
>>> c.simple_root(2)
[1, 1, 0, 0, 0, 0, 0]
```

**class** sympy.liealgebras.type\_f.TypeF(*n*)

**basic\_root**(*i, j*)

Generate roots with 1 in *i*th position and -1 in *j*th position

**basis**()

Returns the number of independent generators of F<sub>4</sub>

**cartan\_matrix**()

The Cartan matrix for F<sub>4</sub>

The Cartan matrix matrix for a Lie algebra is generated by assigning an ordering to the simple roots, ( $\alpha[1]$ , ...,  $\alpha[l]$ ). Then the *ij*th entry of the Cartan matrix is  $\langle \alpha[i], \alpha[j] \rangle$ .

### Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType('A4')
>>> c.cartan_matrix()
Matrix([
[2, -1, 0, 0],
[-1, 2, -1, 0],
[0, -1, 2, -1],
[0, 0, -1, 2]])
```

**dimension**()

Dimension of the vector space V underlying the Lie algebra

### Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("F4")
>>> c.dimension()
4
```

**positive\_roots**()

Generate all the positive roots of A<sub>n</sub>

This is half of all of the roots of F<sub>4</sub>; by multiplying all the positive roots by -1 we get the negative roots.



## Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A3")
>>> c.positive_roots()
{1: [1, -1, 0, 0], 2: [1, 0, -1, 0], 3: [1, 0, 0, -1], 4: [0, 1, -1, 0],
 5: [0, 1, 0, -1], 6: [0, 0, 1, -1]}
```

### roots()

Returns the total number of roots for  $F_4$

### simple\_root(i)

The  $i$ th simple root of  $F_4$

Every lie algebra has a unique root system. Given a root system  $Q$ , there is a subset of the roots such that an element of  $Q$  is called a simple root if it cannot be written as the sum of two elements in  $Q$ . If we let  $D$  denote the set of simple roots, then it is clear that every element of  $Q$  can be written as a linear combination of elements of  $D$  with all coefficients non-negative.

## Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("F4")
>>> c.simple_root(3)
[0, 0, 0, 1]
```

### class sympy.liealgebras.type\_g.TypeG(n)

#### basis()

Returns the number of independent generators of  $G_2$

#### cartan\_matrix()

The Cartan matrix for  $G_2$

The Cartan matrix matrix for a Lie algebra is generated by assigning an ordering to the simple roots, ( $\alpha[1]$ , ...,  $\alpha[l]$ ). Then the  $ij$ th entry of the Cartan matrix is  $\langle \alpha[i], \alpha[j] \rangle$ .

## Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("G2")
>>> c.cartan_matrix()
Matrix([
 [2, -1],
 [-3, 2]])
```

### dimension()

Dimension of the vector space  $V$  underlying the Lie algebra

### Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("G2")
>>> c.dimension()
3
```

#### **positive\_roots()**

Generate all the positive roots of  $A_n$

This is half of all of the roots of  $A_n$ ; by multiplying all the positive roots by -1 we get the negative roots.

### Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A3")
>>> c.positive_roots()
{1: [1, -1, 0, 0], 2: [1, 0, -1, 0], 3: [1, 0, 0, -1], 4: [0, 1, -1, 0],
 5: [0, 1, 0, -1], 6: [0, 0, 1, -1]}
```

#### **roots()**

Returns the total number of roots of  $G_2$

#### **simple\_root(i)**

The  $i$ th simple root of  $G_2$

Every lie algebra has a unique root system. Given a root system  $Q$ , there is a subset of the roots such that an element of  $Q$  is called a simple root if it cannot be written as the sum of two elements in  $Q$ . If we let  $D$  denote the set of simple roots, then it is clear that every element of  $Q$  can be written as a linear combination of elements of  $D$  with all coefficients non-negative.

### Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("G2")
>>> c.simple_root(1)
[0, 1, -1]
```

**class** sympy.liealgebras.weyl\_group.WeylGroup(*cartantype*)

For each semisimple Lie group, we have a Weyl group. It is a subgroup of the isometry group of the root system. Specifically, it's the subgroup that is generated by reflections through the hyperplanes orthogonal to the roots. Therefore, Weyl groups are reflection groups, and so a Weyl group is a finite Coxeter group.

#### **coxeter\_diagram()**

This method returns the Coxeter diagram corresponding to a Weyl group. The Coxeter diagram can be obtained from a Lie algebra's Dynkin diagram by deleting all arrows; the Coxeter diagram is the undirected graph. The vertices of the Coxeter diagram represent the generating reflections of the Weyl group,  $s_i$ . An edge is drawn

between  $s_i$  and  $s_j$  if the order  $m(i, j)$  of  $s_i s_j$  is greater than two. If there is one edge, the order  $m(i, j)$  is 3. If there are two edges, the order  $m(i, j)$  is 4, and if there are three edges, the order  $m(i, j)$  is 6.

### Examples

```
>>> from sympy.liealgebras.weyl_group import WeylGroup
>>> c = WeylGroup("B3")
>>> print(c.coxeter_diagram())
0--0==0
1 2 3
```

#### `delete_doubles(reflections)`

This is a helper method for determining the order of an element in the Weyl group of  $G_2$ . It takes a Weyl element and if repeated simple reflections in it, it deletes them.

#### `element_order(weylelt)`

This method returns the order of a given Weyl group element, which should be specified by the user in the form of products of the generating reflections, i.e. of the form  $r_1 r_2$  etc.

For types A-F, this method current works by taking the matrix form of the specified element, and then finding what power of the matrix is the identity. It then returns this power.

### Examples

```
>>> from sympy.liealgebras.weyl_group import WeylGroup
>>> b = WeylGroup("B4")
>>> b.element_order('r1*r4*r2')
4
```

#### `generators()`

This method creates the generating reflections of the Weyl group for a given Lie algebra. For a Lie algebra of rank  $n$ , there are  $n$  different generating reflections. This function returns them as a list.

### Examples

```
>>> from sympy.liealgebras.weyl_group import WeylGroup
>>> c = WeylGroup("F4")
>>> c.generators()
['r1', 'r2', 'r3', 'r4']
```

#### `group_name()`

This method returns some general information about the Weyl group for a given Lie algebra. It returns the name of the group and the elements it acts on, if relevant.

#### `group_order()`

This method returns the order of the Weyl group. For types A, B, C, D, and E the order depends on the rank of the Lie algebra. For types F and G, the order is fixed.

### Examples

```
>>> from sympy.liealgebras.weyl_group import WeylGroup
>>> c = WeylGroup("D4")
>>> c.group_order()
192.0
```

#### **matrix\_form(*weylelt*)**

This method takes input from the user in the form of products of the generating reflections, and returns the matrix corresponding to the element of the Weyl group. Since each element of the Weyl group is a reflection of some type, there is a corresponding matrix representation. This method uses the standard representation for all the generating reflections.

### Examples

```
>>> from sympy.liealgebras.weyl_group import WeylGroup
>>> f = WeylGroup("F4")
>>> f.matrix_form('r2*r3')
Matrix([
[1, 0, 0, 0],
[0, 1, 0, 0],
[0, 0, 0, -1],
[0, 0, 1, 0]])
```

#### **class sympy.liealgebras.cartan\_type.CartanType\_generator**

Constructor for actually creating things

#### **class sympy.liealgebras.cartan\_type.Standard\_Cartan(*series, n*)**

Concrete base class for Cartan types such as A4, etc

#### **rank()**

Returns the rank of the Lie algebra

#### **series()**

Returns the type of the Lie algebra

#### **sympy.liealgebras.dynkin\_diagram.DynkinDiagram(*t*)**

Display the Dynkin diagram of a given Lie algebra

Works by generating the CartanType for the input, *t*, and then returning the Dynkin diagram method from the individual classes.

### Examples

```
>>> from sympy.liealgebras.dynkin_diagram import DynkinDiagram
>>> print(DynkinDiagram("A3"))
0---0---0
1 2 3
```

```
>>> print(DynkinDiagram("B4"))
0---0---0==>0
1 2 3 4
```

`sympy.liealgebras.cartan_matrix.CartanMatrix(ct)`

Access the Cartan matrix of a specific Lie algebra

### Examples

```
>>> from sympy.liealgebras.cartan_matrix import CartanMatrix
>>> CartanMatrix("A2")
Matrix([
[2, -1],
[-1, 2]])
```

```
>>> CartanMatrix(['C', 3])
Matrix([
[2, -1, 0],
[-1, 2, -1],
[0, -2, 2]])
```

This method works by returning the Cartan matrix which corresponds to Cartan type `t`.

## Polynomial Manipulation

Computations with polynomials are at the core of computer algebra and having a fast and robust polynomials manipulation module is a key for building a powerful symbolic manipulation system. SymPy has a dedicated module `sympy.polys` (page 2360) for computing in polynomial algebras over various coefficient domains.

There is a vast number of methods implemented, ranging from simple tools like polynomial division, to advanced concepts including Gröbner bases and multivariate factorization over algebraic number domains.

## Contents

### Basic functionality of the module

#### Introduction

This tutorial tries to give an overview of the functionality concerning polynomials within SymPy. All code examples assume:

```
>>> from sympy import *
>>> x, y, z = symbols('x,y,z')
>>> init_printing(use_unicode=False, wrap_line=False)
```

#### Basic concepts

##### Polynomials

Given a family  $(x_i)$  of symbols, or other suitable objects, including numbers, expressions derived from them by repeated addition, subtraction and multiplication are called *polynomial expressions in the generators*  $x_i$ .

By the distributive law it is possible to perform multiplications before additions and subtractions. The products of generators thus obtained are called *monomials*. They are usually written in the form  $x_1^{\nu_1} x_2^{\nu_2} \dots x_n^{\nu_n}$  where the exponents  $\nu_i$  are nonnegative integers. It is often convenient to write this briefly as  $x^\nu$  where  $x = (x_1, x_2, \dots, x_n)$  denotes the family of generators and  $\nu = (\nu_1, \nu_2, \dots, \nu_n)$  is the family of exponents.

When all monomials having the same exponents are combined, the polynomial expression becomes a sum of products  $c_\nu x^\nu$ , called the *terms* of the polynomial, where the *coefficients*  $c_\nu$  are integers. If some of the  $x_i$  are manifest numbers, they are incorporated in the coefficients and not regarded as generators. Such coefficients are typically rational, real or complex numbers. Some symbolic numbers, e.g., `pi`, can be either coefficients or generators.

A polynomial expression that is a sum of terms with different monomials is uniquely determined by its family of coefficients  $(c_\nu)$ . Such an expression is customarily called a *polynomial*, though, more properly, that name does stand for the coefficient family once the generators are given. SymPy implements polynomials by default as dictionaries with monomials as keys and coefficients as values. Another implementation consists of nested lists of coefficients.

The set of all polynomials with integer coefficients in the generators  $x_i$  is a *ring*, i.e., the sums, differences and products of its elements are again polynomials in the same generators. This ring is denoted  $\mathbb{Z}[x_1, x_2, \dots, x_n]$ , or  $\mathbb{Z}[(x_i)]$ , and called the *ring of polynomials in the  $x_i$  with integer coefficients*.

More generally, the coefficients of a polynomial can be elements of any commutative ring  $A$ , and the corresponding polynomial ring is then denoted  $A[x_1, x_2, \dots, x_n]$ . The ring  $A$  can also be a polynomial ring. In SymPy, the coefficient ring is called the *domain* of the polynomial ring, and it can be given as a keyword parameter. By default, it is determined by the coefficients of the polynomial arguments.

Polynomial expressions can be transformed into polynomials by the method `sympy.core.expr.Expr.as_poly` (page 956):

```
>>> e = (x + y)*(y - 2*z)
>>> e.as_poly()
Poly(x*y - 2*x*z + y**2 - 2*y*z, x, y, z, domain='ZZ')
```

If a polynomial expression contains numbers that are not integers, they are regarded as coefficients and the coefficient ring is extended accordingly. In particular, division by integers leads to rational coefficients:

```
>>> e = (3*x/2 + y)*(z - 1)
>>> e.as_poly()
Poly(3/2*x*z - 3/2*x + y*z - y, x, y, z, domain='QQ')
```

Symbolic numbers are considered generators unless they are explicitly excluded, in which case they are adjoined to the coefficient ring:

```
>>> e = (x + 2*pi)*y
>>> e.as_poly()
Poly(x*y + 2*y*pi, x, y, pi, domain='ZZ')
>>> e.as_poly(x, y)
Poly(x*y + 2*pi*y, x, y, domain='ZZ[pi]')
```

Alternatively, the coefficient domain can be specified by means of a keyword argument:

```
>>> e = (x + 2*pi)*y
>>> e.as_poly(domain=ZZ[pi])
Poly(x*y + 2*pi*y, x, y, domain='ZZ[pi]')
```

Note that the ring  $\mathbb{Z}[\pi][x, y]$  of polynomials in  $x$  and  $y$  with coefficients in  $\mathbb{Z}[\pi]$  is mathematically equivalent to  $\mathbb{Z}[\pi, x, y]$ , only their implementations differ.

If an expression contains functions of the generators, other than their positive integer powers, these are interpreted as new generators:

```
>>> e = x*sin(y) - y
>>> e.as_poly()
Poly(x*(sin(y)) - y, x, y, sin(y), domain='ZZ')
```

Since  $y$  and  $\sin(y)$  are algebraically independent they can both appear as generators in a polynomial. However, *polynomial expressions must not contain negative powers of generators*:

```
>>> e = x - 1/x
>>> e.as_poly()
Poly(x - (1/x), x, 1/x, domain='ZZ')
```

It is important to realize that the generators  $x$  and  $1/x = x^{-1}$  are treated as algebraically independent variables. In particular, their product is not equal to 1. Hence *generators in denominators should be avoided even if they raise no error in the current implementation*. This behavior is undesirable and may change in the future. Similar problems emerge with rational powers of generators. So, for example,  $x$  and  $\sqrt{x} = x^{1/2}$  are not recognized as algebraically dependent.

If there are algebraic numbers in an expression, it is possible to adjoin them to the coefficient ring by setting the keyword extension:

```
>>> e = x + sqrt(2)
>>> e.as_poly()
Poly(x + (sqrt(2)), x, sqrt(2), domain='ZZ')
>>> e.as_poly(extension=True)
Poly(x + sqrt(2), x, domain='QQ<sqrt(2)>')
```

With the default setting `extension=False`, both  $x$  and  $\sqrt{2}$  are incorrectly considered algebraically independent variables. With coefficients in the extension field  $\mathbb{Q}(\sqrt{2})$  the square root is treated properly as an algebraic number. Setting `extension=True` whenever algebraic numbers are involved is definitely recommended even though it is not forced in the current implementation.

## Divisibility

The fourth rational operation, division, or inverted multiplication, is not generally possible in rings. If  $a$  and  $b$  are two elements of a ring  $A$ , then there may exist a third element  $q$  in  $A$  such that  $a = bq$ . In fact, there may exist several such elements.

If also  $a = bq'$  for some  $q'$  in  $A$ , then  $b(q - q') = 0$ . Hence either  $b$  or  $q - q'$  is zero, or they are both *zero divisors*, nonzero elements whose product is zero.

## Integral domains

Commutative rings with no zero divisors are called *integral domains*. Most of the commonly encountered rings, the ring of integers, fields, and polynomial rings over integral domains are integral domains.

Assume now that  $A$  is an integral domain, and consider the set  $P$  of its nonzero elements, which is closed under multiplication. If  $a$  and  $b$  are in  $P$ , and there exists an element  $q$  in  $P$  such that  $a = bq$ , then  $q$  is unique and called the *quotient*,  $a/b$ , of  $a$  by  $b$ . Moreover, it is said that

- $a$  is *divisible* by  $b$ ,
- $b$  is a *divisor* of  $a$ ,
- $a$  is a *multiple* of  $b$ ,
- $b$  is a *factor* of  $a$ .

An element  $a$  of  $P$  is a divisor of 1 if and only if it is *invertible* in  $A$ , with the inverse  $a^{-1} = 1/a$ . Such elements are called *units*. The units of the ring of integers are 1 and  $-1$ . The invertible elements in a polynomial ring over a field are the nonzero constant polynomials.

If two elements of  $P$ ,  $a$  and  $b$ , are divisible by each other, then the quotient  $a/b$  is invertible with inverse  $b/a$ , or equivalently,  $b = ua$  where  $u$  is a unit. Such elements are said to be *associated* with, or *associates* of, each other. The associates of an integer  $n$  are  $n$  and  $-n$ . In a polynomial ring over a field the associates of a polynomial are its constant multiples.

Each element of  $P$  is divisible by its associates and the units. An element is *irreducible* if it has no other divisors and is not a unit. The irreducible elements in the ring of integers are the prime numbers  $p$  and their opposites  $-p$ . In a field, every nonzero element is invertible and there are no irreducible elements.



## Factorial domains

In the ring of integers, each nonzero element can be represented as a product of irreducible elements and optionally a unit  $\pm 1$ . Moreover, any two such products have the same number of irreducible factors which are associated with each other in a suitable order. Integral domains having this property are called *factorial*, or *unique factorization domains*. In addition to the ring of integers, all polynomial rings over a field are factorial, and so are more generally polynomial rings over any factorial domain. Fields are trivially factorial since there are only units. The irreducible elements of a factorial domain are usually called *primes*.

A family of integers has only a finite number of common divisors and the greatest of them is divisible by all of them. More generally, given a family of nonzero elements  $(a_i)$  in an integral domain, a common divisor  $d$  of the elements is called a *greatest common divisor*, abbreviated *gcd*, of the family if it is a multiple of all common divisors. A greatest common divisor, if it exists, is not unique in general; all of its associates have the same property. It is denoted by  $d = \gcd(a_1, \dots, a_n)$  if there is no danger of confusion. A *least common multiple*, or *lcm*, of a family  $(a_i)$  is defined analogously as a common multiple  $m$  that divides all common multiples. It is denoted by  $m = \text{lcm}(a_1, \dots, a_n)$ .

In a factorial domain, greatest common divisors always exists. They can be found, at least in principle, by factoring each element of a family into a product of prime powers and an optional unit, and, for each prime, taking the least power that appears in the factorizations. The product of these prime powers is then a greatest common divisor. A least common multiple can be obtained from the same factorizations as the product of the greatest powers for each prime.

## Euclidean domains

A practical algorithm for computing a greatest common divisor can be implemented in *Euclidean domains*. They are integral domains that can be endowed with a function  $w$  assigning a nonnegative integer to each nonzero element of the domain and having the following property:

if  $a$  and  $b$  are nonzero, there are  $q$  and  $r$  that satisfy the *division identity*

$$a = qb + r$$

such that either  $r = 0$  or  $w(r) < w(b)$ .

The ring of integers and all univariate polynomial rings over fields are Euclidean domains with  $w(a) = |a|$  resp.  $w(a) = \deg(a)$ .

The division identity for integers is implemented in Python as the built-in function `divmod` that can also be applied to SymPy Integers:

```
>>> divmod(Integer(53), Integer(7))
(7, 4)
```

For polynomials the division identity is given in SymPy by the function `div()` (page 2363):

```
>>> f = 5*x**2 + 10*x + 3
>>> g = 2*x + 2

>>> q, r = div(f, g, domain='QQ')
>>> q
```

(continues on next page)

(continued from previous page)

```

5*x 5
--- + -
 2 2
>>> r
-2
>>> (q*g + r).expand()
 2
5*x + 10*x + 3

```

The division identity can be used to determine the divisibility of elements in a Euclidean domain. If  $r = 0$  in the division identity, then  $a$  is divisible by  $b$ . Conversely, if  $a = cb$  for some element  $c$ , then  $(c - q)b = r$ . It follows that  $c = q$  and  $r = 0$  if  $w$  has the additional property:

if  $a$  and  $b$  are nonzero, then  $w(ab) \geq w(b)$ .

This is satisfied by the functions given above. (And it is always possible to redefine  $w(a)$  by taking the minimum of the values  $w(xa)$  for  $x \neq 0$ .)

The principal application of the division identity is the efficient computation of a greatest common divisor by means of the [Euclidean algorithm](#). It applies to two elements of a Euclidean domain. A gcd of several elements can be obtained by iteration.

The function for computing the greatest common divisor of integers in SymPy is currently [igcd\(\)](#) (page 994):

```

>>> igcd(2, 4)
2
>>> igcd(5, 10, 15)
5

```

For univariate polynomials over a field the function has its common name [gcd\(\)](#) (page 2368), and the returned polynomial is monic:

```

>>> f = 4*x**2 - 1
>>> g = 8*x**3 + 1
>>> gcd(f, g, domain=QQ)
x + 1/2

```

## Divisibility of polynomials

The ring  $A = \mathbb{Z}[x]$  of univariate polynomials over the ring of integers is not Euclidean but it is still factorial. To see this, consider the divisibility in  $A$ .

Let  $f$  and  $g$  be two nonzero polynomials in  $A$ . If  $f$  is divisible by  $g$  in  $A$ , then it is also divisible in the ring  $B = \mathbb{Q}[x]$  of polynomials with rational coefficients. Since  $B$  is Euclidean, this can be determined by means of the division identity.

Assume, conversely, that  $f = gh$  for some polynomial  $h$  in  $B$ . Then  $f$  is divisible by  $g$  in  $A$  if and only if the coefficients of  $h$  are integers. To find out when this is true it is necessary to consider the divisibility of the coefficients.

For a polynomial  $f$  in  $A$ , let  $c$  be the greatest common divisor of its coefficients. Then  $f$  is divisible by the constant polynomial  $c$  in  $A$ , and the quotient  $f/c = p$  is a polynomial whose coefficients are integers that have no common divisor apart from the units. Such polynomials are called *primitive*. A polynomial with rational coefficients can also be written as  $f = cp$ ,

where  $c$  is a rational number and  $p$  is a primitive polynomial. The constant  $c$  is called the *content* of  $f$ , and  $p$  is its *primitive part*. These components can be found by the method `sympy.core.expr.Expr.as_content_primitive` (page 951):

```
>>> f = 6*x**2 - 3*x + 9
>>> c, p = f.as_content_primitive()
>>> c, p
2
(3, 2*x - x + 3)
>>> f = x**2/3 - x/2 + 1
>>> c, p = f.as_content_primitive()
>>> c, p
2
(1/6, 2*x - 3*x + 6)
```

Let  $f, f'$  be polynomials with contents  $c, c'$  and primitive parts  $p, p'$ . Then  $ff' = (cc')(pp')$  where the product  $pp'$  is primitive by [Gauss's lemma](#). It follows that

the content of a product of polynomials is the product of their contents and the primitive part of the product is the product of the primitive parts.

Returning to the divisibility in the ring  $\mathbb{Z}[x]$ , assume that  $f$  and  $g$  are two polynomials with integer coefficients such that the division identity in  $\mathbb{Q}[x]$  yields the equality  $f = gh$  for some polynomial  $h$  with rational coefficients. Then the content of  $f$  is equal to the content of  $g$  multiplied by the content of  $h$ . As  $h$  has integer coefficients if and only if its content is an integer, we get the following criterion:

$f$  is divisible by  $g$  in the ring  $\mathbb{Z}[x]$  if and only if

- i.  $f$  is divisible by  $g$  in  $\mathbb{Q}[x]$ , and
- ii. the content of  $f$  is divisible by the content of  $g$  in  $\mathbb{Z}$ .

If  $f = cp$  is irreducible in  $\mathbb{Z}[x]$ , then either  $c$  or  $p$  must be a unit. If  $p$  is not a unit, it must be irreducible also in  $\mathbb{Q}[x]$ . For if it is a product of two polynomials, it is also the product of their primitive parts, and one of them must be a unit. Hence there are two kinds of irreducible elements in  $\mathbb{Z}[x]$ :

- i. prime numbers of  $\mathbb{Z}$ , and
- ii. primitive polynomials that are irreducible in  $\mathbb{Q}[x]$ .

It follows that each polynomial in  $\mathbb{Z}[x]$  is a product of irreducible elements. It suffices to factor its content and primitive part separately. These products are essentially unique; hence  $\mathbb{Z}[x]$  is also factorial.

Another important consequence is that a greatest common divisor of two polynomials in  $\mathbb{Z}[x]$  can be found efficiently by applying the Euclidean algorithm separately to their contents and primitive parts in the Euclidean domains  $\mathbb{Z}$  and  $\mathbb{Q}[x]$ . This is also implemented in SymPy:

```
>>> f = 4*x**2 - 1
>>> g = 8*x**3 + 1
>>> gcd(f, g)
2*x + 1
>>> gcd(6*f, 3*g)
6*x + 3
```

## Basic functionality

These functions provide different algorithms dealing with polynomials in the form of SymPy expression, like symbols, sums etc.

### Division

The function `div()` (page 2363) provides division of polynomials with remainder. That is, for polynomials  $f$  and  $g$ , it computes  $q$  and  $r$ , such that  $f = g \cdot q + r$  and  $\deg(r) < \deg(g)$ . For polynomials in one variables with coefficients in a field, say, the rational numbers,  $q$  and  $r$  are uniquely defined this way:

```
>>> f = 5*x**2 + 10*x + 3
>>> g = 2*x + 2

>>> q, r = div(f, g, domain='QQ')
>>> q
5*x 5
--- + -
 2 2
>>> r
-2
>>> (q*g + r).expand()
 2
5*x + 10*x + 3
```

As you can see,  $q$  has a non-integer coefficient. If you want to do division only in the ring of polynomials with integer coefficients, you can specify an additional parameter:

```
>>> q, r = div(f, g, domain='ZZ')
>>> q
0
>>> r
 2
5*x + 10*x + 3
```

But be warned, that this ring is no longer Euclidean and that the degree of the remainder doesn't need to be smaller than that of  $f$ . Since 2 doesn't divide 5,  $2x$  doesn't divide  $5x^2$ , even if the degree is smaller. But:

```
>>> g = 5*x + 1

>>> q, r = div(f, g, domain='ZZ')
>>> q
x
>>> r
9*x + 3
>>> (q*g + r).expand()
 2
5*x + 10*x + 3
```

This also works for polynomials with multiple variables:

```

>>> f = x*y + y*z
>>> g = 3*x + 3*z

>>> q, r = div(f, g, domain='QQ')
>>> q
y
-
3
>>> r
0

```

In the last examples, all of the three variables  $x$ ,  $y$  and  $z$  are assumed to be variables of the polynomials. But if you have some unrelated constant as coefficient, you can specify the variables explicitly:

```

>>> a, b, c = symbols('a,b,c')
>>> f = a*x**2 + b*x + c
>>> g = 3*x + 2
>>> q, r = div(f, g, domain='QQ')
>>> q
a*x 2*a b
--- - --- + -
3 9 3

>>> r
4*a 2*b
--- - --- + c
9 3

```

## GCD and LCM

With division, there is also the computation of the greatest common divisor and the least common multiple.

When the polynomials have integer coefficients, the contents' gcd is also considered:

```

>>> f = (12*x + 12)*x
>>> g = 16*x**2
>>> gcd(f, g)
4*x

```

But if the polynomials have rational coefficients, then the returned polynomial is monic:

```

>>> f = 3*x**2/2
>>> g = 9*x/4
>>> gcd(f, g)
x

```

It also works with multiple variables. In this case, the variables are ordered alphabetically, by default, which has influence on the leading coefficient:

```
>>> f = x*y/2 + y**2
>>> g = 3*x + 6*y

>>> gcd(f, g)
x + 2*y
```

The lcm is connected with the gcd and one can be computed using the other:

```
>>> f = x*y**2 + x**2*y
>>> g = x**2*y**2
>>> gcd(f, g)
x*y
>>> lcm(f, g)
 3 2 2 3
x *y + x *y
>>> (f*g).expand()
 4 3 3 4
x *y + x *y
>>> (gcd(f, g, x, y)*lcm(f, g, x, y)).expand()
 4 3 3 4
x *y + x *y
```

## Square-free factorization

The square-free factorization of a univariate polynomial is the product of all factors (not necessarily irreducible) of degree 1, 2 etc.:

```
>>> f = 2*x**2 + 5*x**3 + 4*x**4 + x**5

>>> sqf_list(f)
(1, [(x + 2, 1), (x2 + x, 2)])

>>> sqf(f)
 2
 / 2 \
(x + 2)*\x + x/
```

## Factorization

This function provides factorization of univariate and multivariate polynomials with rational coefficients:

```
>>> factor(x**4/2 + 5*x**3/12 - x**2/3)
 2
x *(2*x - 1)*(3*x + 4)

 12

>>> factor(x**2 + 4*x*y + 4*y**2)
```

(continues on next page)

(continued from previous page)

$$(x + 2*y)^2$$

## Groebner bases

Buchberger's algorithm is implemented, supporting various monomial orders:

```
>>> groebner([x**2 + 1, y**4*x + x**3], x, y, order='lex')
 / [2 4] \
GroebnerBasis\ [x + 1, y - 1], x, y, domain=ZZ, order=lex/

>>> groebner([x**2 + 1, y**4*x + x**3, x*y*z**3], x, y, z, order='grevlex')
 / [4 3 2] \
GroebnerBasis\ [y - 1, z , x + 1], x, y, z, domain=ZZ, order=grevlex/
```

## Solving Equations

We have (incomplete) methods to find the complex or even symbolic roots of polynomials and to solve some systems of polynomial equations:

```
>>> from sympy import roots, solve_poly_system

>>> solve(x**3 + 2*x + 3, x)

 1 \ / 11 *I 1 \ / 11 *I
[-1, - - - ----, - - - + ----]
 2 2 2 2

>>> p = Symbol('p')
>>> q = Symbol('q')

>>> solve(x**2 + p*x + q, x)

 / 2 \ / 2
 p \ / p - 4*q p \ / p - 4*q
[- - - ----, - - - + ----]
 2 2 2 2

>>> solve_poly_system([y - x, x - 5], x, y)
[(5, 5)]

>>> solve_poly_system([y**2 - x**3 + 1, y*x], x, y)

 1 \ / 3 *I 1 \ / 3 *I
[(0, -I), (0, I), (1, 0), (- - - ----, 0), (- - - + ----, 0)]
 2 2 2 2
```

## Examples from Wester's Article

### Introduction

In this tutorial we present examples from Wester's article concerning comparison and critique of mathematical abilities of several computer algebra systems (see [Wester1999]). All the examples are related to polynomial and algebraic computations and SymPy specific remarks were added to all of them.

### Examples

All examples in this tutorial are computable, so one can just copy and paste them into a Python shell and do something useful with them. All computations were done using the following setup:

```
>>> from sympy import *

>>> init_printing(use_unicode=True, wrap_line=False)

>>> var('x,y,z,s,c,n')
(x, y, z, s, c, n)
```

### Simple univariate polynomial factorization

To obtain a factorization of a polynomial use *factor()* (page 2373) function. By default *factor()* (page 2373) returns the result in unevaluated form, so the content of the input polynomial is left unexpanded, as in the following example:

```
>>> factor(6*x - 10)
2*(3*x - 5)
```

To achieve the same effect in a more systematic way use *primitive()* (page 2369) function, which returns the content and the primitive part of the input polynomial:

```
>>> primitive(6*x - 10)
(2, 3*x - 5)
```

---

**Note:** The content and the primitive part can be computed only over a ring. To simplify coefficients of a polynomial over a field use *monic()* (page 2369).

---



## Univariate GCD, resultant and factorization

Consider univariate polynomials  $f$ ,  $g$  and  $h$  over integers:

```
>>> f = 64*x**34 - 21*x**47 - 126*x**8 - 46*x**5 - 16*x**60 - 81
>>> g = 72*x**60 - 25*x**25 - 19*x**23 - 22*x**39 - 83*x**52 + 54*x**10 + 81
>>> h = 34*x**19 - 25*x**16 + 70*x**7 + 20*x**3 - 91*x - 86
```

We can compute the greatest common divisor (GCD) of two polynomials using `gcd()` (page 2368) function:

```
>>> gcd(f, g)
1
```

We see that  $f$  and  $g$  have no common factors. However,  $f*h$  and  $g*h$  have an obvious factor  $h$ :

```
>>> gcd(expand(f*h), expand(g*h)) - h
0
```

The same can be verified using the resultant of univariate polynomials:

```
>>> resultant(expand(f*h), expand(g*h))
0
```

Factorization of large univariate polynomials (of degree 120 in this case) over integers is also possible:

```
>>> factor(expand(f*g))
(60 47 34 8 5 \ (60 52 39 25
 25 23 10 \
- (16*x + 21*x - 64*x + 126*x + 46*x + 81) (72*x - 83*x - 22*x -
 25*x - 19*x + 54*x + 81)
```

## Multivariate GCD and factorization

What can be done in univariate case, can be also done for multivariate polynomials. Consider the following polynomials  $f$ ,  $g$  and  $h$  in  $\mathbb{Z}[x, y, z]$ :

```
>>> f = 24*x*y**19*z**8 - 47*x**17*y**5*z**8 + 6*x**15*y**9*z**2 - 3*x**22 + 5
>>> g = 34*x**5*y**8*z**13 + 20*x**7*y**7*z**7 + 12*x**9*y**16*z**4 +
 80*y**14*z
>>> h = 11*x**12*y**7*z**13 - 23*x**2*y**8*z**10 + 47*x**17*y**5*z**8
```

As previously, we can verify that  $f$  and  $g$  have no common factors:

```
>>> gcd(f, g)
1
```

However,  $f*h$  and  $g*h$  have an obvious factor  $h$ :

```
>>> gcd(expand(f*h), expand(g*h)) - h
0
```

Multivariate factorization of large polynomials is also possible:

```
>>> factor(expand(f*g))
 7 (9 9 3 7 6 5 12 7) (22 17 5 8
→ 15 9 2 19 8)
-2·y ·z·(6·x ·y ·z + 10·x ·z + 17·x ·y·z + 40·y)·(3·x + 47·x ·y ·z -
→6·x ·y ·z - 24·x·y ·z - 5)
```

## Support for symbols in exponents

Polynomial manipulation functions provided by `sympy.polys` (page 2360) are mostly used with integer exponents. However, it's perfectly valid to compute with symbolic exponents, e.g.:

```
>>> gcd(2*x**(n + 4) - x**(n + 2), 4*x**(n + 1) + 3*x**n)
n
x
```

## Testing if polynomials have common zeros

To test if two polynomials have a root in common we can use `resultant()` (page 2366) function. The theory says that the resultant of two polynomials vanishes if there is a common zero of those polynomials. For example:

```
>>> resultant(3*x**4 + 3*x**3 + x**2 - x - 2, x**3 - 3*x**2 + x + 5)
0
```

We can visualize this fact by factoring the polynomials:

```
>>> factor(3*x**4 + 3*x**3 + x**2 - x - 2)
 (3
(x + 1)·(3·x + x - 2)

>>> factor(x**3 - 3*x**2 + x + 5)
 (2
(x + 1)·(x - 4·x + 5)
```

In both cases we obtained the factor  $x + 1$  which tells us that the common root is  $x = -1$ .

## Normalizing simple rational functions

To remove common factors from the numerator and the denominator of a rational function the elegant way, use `cancel()` (page 2376) function. For example:

```
>>> cancel((x**2 - 4)/(x**2 + 4*x + 4))
x - 2

x + 2
```

## Expanding expressions and factoring back

One can work easily with expressions in both expanded and factored forms. Consider a polynomial  $f$  in expanded form. We differentiate it and factor the result back:

```
>>> f = expand((x + 1)**20)
>>> g = diff(f, x)
>>> factor(g)
 19
20·(x + 1)
```

The same can be achieved in factored form:

```
>>> diff((x + 1)**20, x)
 19
20·(x + 1)
```

## Factoring in terms of cyclotomic polynomials

SymPy can very efficiently decompose polynomials of the form  $x^n \pm 1$  in terms of cyclotomic polynomials:

```
>>> factor(x**15 - 1)
 2 4 3 2 8 7 5 4 3
(x - 1)·(x + x + 1)·(x + x + x + x + 1)·(x - x + x - x + x - x + 1)
```

The original Wester's example was  $x^{100} - 1$ , but was truncated for readability purpose. Note that this is not a big struggle for `factor()` (page 2373) to decompose polynomials of degree 1000 or greater.

## Univariate factoring over Gaussian numbers

Consider a univariate polynomial  $f$  with integer coefficients:

```
>>> f = 4*x**4 + 8*x**3 + 77*x**2 + 18*x + 153
```

We want to obtain a factorization of  $f$  over Gaussian numbers. To do this we use `factor()` (page 2373) as previously, but this time we set `gaussian` keyword to `True`:

```
>>> factor(f, gaussian=True)
 3·i 3·i
4·(x - —)·(x + —)·(x + 1 - 4·i)·(x + 1 + 4·i)
 2 2
```

As the result we got a splitting factorization of  $f$  with monic factors (this is a general rule when computing in a field with SymPy). The `gaussian` keyword is useful for improving code readability, however the same result can be computed using more general syntax:

```
>>> factor(f, extension=I)
4· $\left(x - \frac{3\cdot i}{2}\right) \cdot \left(x + \frac{3\cdot i}{2}\right) \cdot (x + 1 - 4\cdot i) \cdot (x + 1 + 4\cdot i)$
```

## Computing with automatic field extensions

Consider two univariate polynomials  $f$  and  $g$ :

```
>>> f = x**3 + (sqrt(2) - 2)*x**2 - (2*sqrt(2) + 3)*x - 3*sqrt(2)
>>> g = x**2 - 2
```

We would like to reduce degrees of the numerator and the denominator of a rational function  $f/g$ . To do this we employ `cancel()` (page 2376) function:

```
>>> cancel(f/g)

$$\frac{x^3 - 2x^2 + \sqrt{2}x^2 - 3x - 2\sqrt{2}x - 3\sqrt{2}}{x^2 - 2}$$

```

Unfortunately nothing interesting happened. This is because by default SymPy treats  $\sqrt{2}$  as a generator, obtaining a bivariate polynomial for the numerator. To make `cancel()` (page 2376) recognize algebraic properties of  $\sqrt{2}$ , one needs to use extension keyword:

```
>>> cancel(f/g, extension=True)

$$\frac{x^2 - 2x - 3}{x - \sqrt{2}}$$

```

Setting `extension=True` tells `cancel()` (page 2376) to find minimal algebraic number domain for the coefficients of  $f/g$ . The automatically inferred domain is  $\mathbb{Q}(\sqrt{2})$ . If one doesn't want to rely on automatic inference, the same result can be obtained by setting the extension keyword with an explicit algebraic number:

```
>>> cancel(f/g, extension=sqrt(2))

$$\frac{x^2 - 2x - 3}{x - \sqrt{2}}$$

```

## Univariate factoring over various domains

Consider a univariate polynomial  $f$  with integer coefficients:

```
>>> f = x**4 - 3*x**2 + 1
```

With `sympy.polys` (page 2360) we can obtain factorizations of  $f$  over different domains, which includes:

- rationals:

```
>>> factor(f)
(2) (2)
(x - x - 1)·(x + x - 1)
```

- finite fields:

```
>>> factor(f, modulus=5)
 2 2
(x - 2) · (x + 2)
```

- algebraic numbers:

```
>>> alg = AlgebraicNumber((sqrt(5) - 1)/2, alias='alpha')
>>> factor(f, extension=alg)
(x - α)·(x + α)·(x - 1 - α)·(x + α + 1)
```

## Factoring polynomials into linear factors

Currently SymPy can factor polynomials into irreducibles over various domains, which can result in a splitting factorization (into linear factors). However, there is currently no systematic way to infer a splitting field (algebraic number field) automatically. In future the following syntax will be implemented:

```
>>> factor(x**3 + x**2 - 7, split=True)
Traceback (most recent call last):
...
NotImplementedError: 'split' option is not implemented yet
```

Note this is different from `extension=True`, because the later only tells how expression parsing should be done, not what should be the domain of computation. One can simulate the `split` keyword for several classes of polynomials using `solve()` (page 836) function.

## Advanced factoring over finite fields

Consider a univariate polynomial  $f$  with integer coefficients:

```
>>> f = x**11 + x + 1
```

We can factor  $f$  over a large finite field  $F_{65537}$ :

```
>>> factor(f, modulus=65537)
(2) (9 8 6 5 3 2)
(x + x + 1) (x - x + x - x + x - x + 1)
```

and expand the resulting factorization back:

```
>>> expand(_)
11
x + x + 1
```

obtaining polynomial  $f$ . This was done using symmetric polynomial representation over finite fields. The same thing can be done using non-symmetric representation:

```
>>> factor(f, modulus=65537, symmetric=False)
(2) (9 8 6 5 3 2)
(x + x + 1) (x + 65536·x + x + 65536·x + x + 65536·x + 1)
```

As with symmetric representation we can expand the factorization to get the input polynomial back. This time, however, we need to truncate coefficients of the expanded polynomial modulo 65537:

```
>>> trunc(expand(_), 65537)
11
x + x + 1
```

## Working with expressions as polynomials

Consider a multivariate polynomial  $f$  in  $\mathbb{Z}[x, y, z]$ :

```
>>> f = expand((x - 2*y**2 + 3*z**3)**20)
```

We want to compute factorization of  $f$ . To do this we use `factor` as usually, however we note that the polynomial in consideration is already in expanded form, so we can tell the factorization routine to skip expanding  $f$ :

```
>>> factor(f, expand=False)
20
(2 3)
(x - 2·y + 3·z)
```

The default in `sympy.polys` (page 2360) is to expand all expressions given as arguments to polynomial manipulation functions and `Poly` (page 2378) class. If we know that expanding is unnecessary, then by setting `expand=False` we can save quite a lot of time for complicated inputs. This can be really important when computing with expressions like:

```
>>> g = expand((sin(x) - 2*cos(y)**2 + 3*tan(z)**3)**20)

>>> factor(g, expand=False)
(
 2 3
 (-sin(x) + 2*cos(y) - 3*tan(z))
)20
```

## Computing reduced Gröbner bases

To compute a reduced Gröbner basis for a set of polynomials use the `groebner()` (page 2377) function. The function accepts various monomial orderings, e.g.: `lex`, `grlex` and `grevlex`, or a user defined one, via `order` keyword. The `lex` ordering is the most interesting because it has elimination property, which means that if the system of polynomial equations to `groebner()` (page 2377) is zero-dimensional (has finite number of solutions) the last element of the basis is a univariate polynomial. Consider the following example:

```
>>> f = expand((1 - c**2)**5 * (1 - s**2)**5 * (c**2 + s**2)**10)

>>> groebner([f, c**2 + s**2 - 1])
 2 2 20 18 16 14 12 10
GroebnerBasis([c + s - 1, c - 5*c + 10*c - 10*c + 5*c - c], s,
 c, domain=ZZ, order=lex)
```

The result is an ordinary Python list, so we can easily apply a function to all its elements, for example we can factor those elements:

```
>>> list(map(factor, _))
 2 2 10 5 5
[c + s - 1, c * (c - 1) * (c + 1)]
```

From the above we can easily find all solutions of the system of polynomial equations. Or we can use `solve()` (page 836) to achieve this in a more systematic way:

```
>>> solve([f, s**2 + c**2 - 1], c, s)
[(-1, 0), (0, -1), (0, 1), (1, 0)]
```

## Multivariate factoring over algebraic numbers

Computing with multivariate polynomials over various domains is as simple as in univariate case. For example consider the following factorization over  $\mathbb{Q}(\sqrt{-3})$ :

```
>>> factor(x**3 + y**3, extension=sqrt(-3))
(x + y) * (x + y * (1/2 - sqrt(3)*I/2)) * (x + y * (1/2 + sqrt(3)*I/2))
```

---

**Note:** Currently multivariate polynomials over finite fields aren't supported.

---

## Partial fraction decomposition

Consider a univariate rational function  $f$  with integer coefficients:

```
>>> f = (x**2 + 2*x + 3)/(x**3 + 4*x**2 + 5*x + 2)
```

To decompose  $f$  into partial fractions use [`apart\(\)`](#) (page 2443) function:

```
>>> apart(f)
 3 2 2
----- - ---- + ----
x + 2 x + 1 (x + 1)2
```

To return from partial fractions to the rational function use a composition of [`together\(\)`](#) (page 2442) and [`cancel\(\)`](#) (page 2376):

```
>>> cancel(together(_))
 2
 x + 2·x + 3

 3 2
 x + 4·x + 5·x + 2
```

## Literature

### Polynomials Manipulation Module Reference

Polynomial manipulation algorithms and algebraic objects.

See [Polynomial Manipulation](#) (page 2341) for an index of documentation for the polys module and [Basic functionality of the module](#) (page 2342) for an introductory explanation.

### Basic polynomial manipulation functions

`sympy.polys.polytools.poly(expr, *gens, **args)`

Efficiently transform an expression into a polynomial.

#### Examples

```
>>> from sympy import poly
>>> from sympy.abc import x
```

```
>>> poly(x*(x**2 + x - 1)**2)
Poly(x**5 + 2*x**4 - x**3 - 2*x**2 + x, x, domain='ZZ')
```

`sympy.polys.polytools.poly_from_expr(expr, *gens, **args)`

Construct a polynomial from an expression.



`sympy.polys.polytools.parallel_poly_from_expr(exprs, *gens, **args)`

Construct polynomials from expressions.

`sympy.polys.polytools.degree(f, gen=0)`

Return the degree of  $f$  in the given variable.

The degree of 0 is negative infinity.

### Examples

```
>>> from sympy import degree
>>> from sympy.abc import x, y
```

```
>>> degree(x**2 + y*x + 1, gen=x)
2
>>> degree(x**2 + y*x + 1, gen=y)
1
>>> degree(0, x)
-oo
```

See also:

[`sympy.polys.polytools.Poly.total\_degree`](#) (page 2421), [`degree\_list`](#) (page 2361)

`sympy.polys.polytools.degree_list(f, *gens, **args)`

Return a list of degrees of  $f$  in all variables.

### Examples

```
>>> from sympy import degree_list
>>> from sympy.abc import x, y
```

```
>>> degree_list(x**2 + y*x + 1)
(2, 1)
```

`sympy.polys.polytools.LC(f, *gens, **args)`

Return the leading coefficient of  $f$ .

### Examples

```
>>> from sympy import LC
>>> from sympy.abc import x, y
```

```
>>> LC(4*x**2 + 2*x*y**2 + x*y + 3*y)
4
```

`sympy.polys.polytools.LM(f, *gens, **args)`

Return the leading monomial of  $f$ .

### Examples

```
>>> from sympy import LM
>>> from sympy.abc import x, y
```

```
>>> LM(4*x**2 + 2*x*y**2 + x*y + 3*y)
x**2
```

`sympy.polys.polytools.LT(f, *gens, **args)`  
Return the leading term of *f*.

### Examples

```
>>> from sympy import LT
>>> from sympy.abc import x, y
```

```
>>> LT(4*x**2 + 2*x*y**2 + x*y + 3*y)
4*x**2
```

`sympy.polys.polytools.pdiv(f, g, *gens, **args)`  
Compute polynomial pseudo-division of *f* and *g*.

### Examples

```
>>> from sympy import pdiv
>>> from sympy.abc import x
```

```
>>> pdiv(x**2 + 1, 2*x - 4)
(2*x + 4, 20)
```

`sympy.polys.polytools.prem(f, g, *gens, **args)`  
Compute polynomial pseudo-remainder of *f* and *g*.

### Examples

```
>>> from sympy import prem
>>> from sympy.abc import x
```

```
>>> prem(x**2 + 1, 2*x - 4)
20
```

`sympy.polys.polytools.pquo(f, g, *gens, **args)`  
Compute polynomial pseudo-quotient of *f* and *g*.

## Examples

```
>>> from sympy import pquo
>>> from sympy.abc import x
```

```
>>> pquo(x**2 + 1, 2*x - 4)
2*x + 4
>>> pquo(x**2 - 1, 2*x - 1)
2*x + 1
```

`sympy.polys.polytools.pquo(f, g, *gens, **args)`

Compute polynomial exact pseudo-quotient of *f* and *g*.

## Examples

```
>>> from sympy import pexquo
>>> from sympy.abc import x
```

```
>>> pexquo(x**2 - 1, 2*x - 2)
2*x + 2
```

```
>>> pexquo(x**2 + 1, 2*x - 4)
Traceback (most recent call last):
...
ExactQuotientFailed: 2*x - 4 does not divide x**2 + 1
```

`sympy.polys.polytools.div(f, g, *gens, **args)`

Compute polynomial division of *f* and *g*.

## Examples

```
>>> from sympy import div, ZZ, QQ
>>> from sympy.abc import x
```

```
>>> div(x**2 + 1, 2*x - 4, domain=ZZ)
(0, x**2 + 1)
>>> div(x**2 + 1, 2*x - 4, domain=QQ)
(x/2 + 1, 5)
```

`sympy.polys.polytools.rem(f, g, *gens, **args)`

Compute polynomial remainder of *f* and *g*.

### Examples

```
>>> from sympy import rem, ZZ, QQ
>>> from sympy.abc import x
```

```
>>> rem(x**2 + 1, 2*x - 4, domain=ZZ)
x**2 + 1
>>> rem(x**2 + 1, 2*x - 4, domain=QQ)
5
```

`sympy.polys.polytools.quo(f, g, *gens, **args)`  
Compute polynomial quotient of *f* and *g*.

### Examples

```
>>> from sympy import quo
>>> from sympy.abc import x
```

```
>>> quo(x**2 + 1, 2*x - 4)
x/2 + 1
>>> quo(x**2 - 1, x - 1)
x + 1
```

`sympy.polys.polytools.exquo(f, g, *gens, **args)`  
Compute polynomial exact quotient of *f* and *g*.

### Examples

```
>>> from sympy import exquo
>>> from sympy.abc import x
```

```
>>> exquo(x**2 - 1, x - 1)
x + 1
```

```
>>> exquo(x**2 + 1, 2*x - 4)
Traceback (most recent call last):
...
ExactQuotientFailed: 2*x - 4 does not divide x**2 + 1
```

`sympy.polys.polytools.half_gcdex(f, g, *gens, **args)`  
Half extended Euclidean algorithm of *f* and *g*.  
Returns (*s*, *h*) such that  $h = \gcd(f, g)$  and  $s*f = h \pmod{g}$ .

## Examples

```
>>> from sympy import half_gcdex
>>> from sympy.abc import x
```

```
>>> half_gcdex(x**4 - 2*x**3 - 6*x**2 + 12*x + 15, x**3 + x**2 - 4*x - 4)
(3/5 - x/5, x + 1)
```

`sympy.polys.polytools.gcdex(f, g, *gens, **args)`

Extended Euclidean algorithm of  $f$  and  $g$ .

Returns  $(s, t, h)$  such that  $h = \gcd(f, g)$  and  $s*f + t*g = h$ .

## Examples

```
>>> from sympy import gcdex
>>> from sympy.abc import x
```

```
>>> gcdex(x**4 - 2*x**3 - 6*x**2 + 12*x + 15, x**3 + x**2 - 4*x - 4)
(3/5 - x/5, x**2/5 - 6*x/5 + 2, x + 1)
```

`sympy.polys.polytools.invert(f, g, *gens, **args)`

Invert  $f$  modulo  $g$  when possible.

## Examples

```
>>> from sympy import invert, S, mod_inverse
>>> from sympy.abc import x
```

```
>>> invert(x**2 - 1, 2*x - 1)
-4/3
```

```
>>> invert(x**2 - 1, x - 1)
Traceback (most recent call last):
...
NotInvertible: zero divisor
```

For more efficient inversion of Rationals, use the `mod_inverse` (page 1004) function:

```
>>> mod_inverse(3, 5)
2
>>> (S(2)/5).invert(S(7)/3)
5/2
```

## See also:

[`sympy.core.numbers.mod\_inverse`](#) (page 1004)

`sympy.polys.polytools.subresultants(f, g, *gens, **args)`

Compute subresultant PRS of  $f$  and  $g$ .

### Examples

```
>>> from sympy import subresultants
>>> from sympy.abc import x
```

```
>>> subresultants(x**2 + 1, x**2 - 1)
[x**2 + 1, x**2 - 1, -2]
```

`sympy.polys.polytools.resultant(f, g, *gens, includePRS=False, **args)`  
Compute resultant of f and g.

### Examples

```
>>> from sympy import resultant
>>> from sympy.abc import x
```

```
>>> resultant(x**2 + 1, x**2 - 1)
4
```

`sympy.polys.polytools.discriminant(f, *gens, **args)`  
Compute discriminant of f.

### Examples

```
>>> from sympy import discriminant
>>> from sympy.abc import x
```

```
>>> discriminant(x**2 + 2*x + 3)
-8
```

`sympy.polys.polytools.terms_gcd(f, *gens, **args)`  
Remove GCD of terms from f.

If the deep flag is True, then the arguments of f will have terms\_gcd applied to them.

If a fraction is factored out of f and f is an Add, then an unevaluated Mul will be returned so that automatic simplification does not redistribute it. The hint `clear`, when set to False, can be used to prevent such factoring when all coefficients are not fractions.

### Examples

```
>>> from sympy import terms_gcd, cos
>>> from sympy.abc import x, y
>>> terms_gcd(x**6*y**2 + x**3*y, x, y)
x**3*y*(x**3*y + 1)
```

The default action of polys routines is to expand the expression given to them. `terms_gcd` follows this behavior:

```
>>> terms_gcd((3+3*x)*(x+x*y))
3*x*(x*y + x + y + 1)
```

If this is not desired then the hint `expand` can be set to `False`. In this case the expression will be treated as though it were comprised of one or more terms:

```
>>> terms_gcd((3+3*x)*(x+x*y), expand=False)
(3*x + 3)*(x*y + x)
```

In order to traverse factors of a `Mul` or the arguments of other functions, the `deep` hint can be used:

```
>>> terms_gcd((3 + 3*x)*(x + x*y), expand=False, deep=True)
3*x*(x + 1)*(y + 1)
>>> terms_gcd(cos(x + x*y), deep=True)
cos(x*(y + 1))
```

Rationals are factored out by default:

```
>>> terms_gcd(x + y/2)
(2*x + y)/2
```

Only the `y`-term had a coefficient that was a fraction; if one does not want to factor out the  $1/2$  in cases like this, the flag `clear` can be set to `False`:

```
>>> terms_gcd(x + y/2, clear=False)
x + y/2
>>> terms_gcd(x*y/2 + y**2, clear=False)
y*(x/2 + y)
```

The `clear` flag is ignored if all coefficients are fractions:

```
>>> terms_gcd(x/3 + y/2, clear=False)
(2*x + 3*y)/6
```

#### See also:

[`sympy.core.exprtools.gcd\_terms`](#) (page 1071), [`sympy.core.exprtools.factor\_terms`](#) (page 1072)

`sympy.polys.polytools.cofactors(f, g, *gens, **args)`

Compute GCD and cofactors of `f` and `g`.

Returns polynomials `(h, cff, cfg)` such that  $h = \gcd(f, g)$ , and  $cff = \text{quo}(f, h)$  and  $cfg = \text{quo}(g, h)$  are, so called, cofactors of `f` and `g`.

### Examples

```
>>> from sympy import cofactors
>>> from sympy.abc import x
```

```
>>> cofactors(x**2 - 1, x**2 - 3*x + 2)
(x - 1, x + 1, x - 2)
```

`sympy.polys.polytools.gcd(f, g=None, *gens, **args)  
Compute GCD of f and g.`

### Examples

```
>>> from sympy import gcd
>>> from sympy.abc import x
```

```
>>> gcd(x**2 - 1, x**2 - 3*x + 2)
x - 1
```

`sympy.polys.polytools.gcd_list(seq, *gens, **args)  
Compute GCD of a list of polynomials.`

### Examples

```
>>> from sympy import gcd_list
>>> from sympy.abc import x
```

```
>>> gcd_list([x**3 - 1, x**2 - 1, x**2 - 3*x + 2])
x - 1
```

`sympy.polys.polytools.lcm(f, g=None, *gens, **args)  
Compute LCM of f and g.`

### Examples

```
>>> from sympy import lcm
>>> from sympy.abc import x
```

```
>>> lcm(x**2 - 1, x**2 - 3*x + 2)
x**3 - 2*x**2 - x + 2
```

`sympy.polys.polytools.lcm_list(seq, *gens, **args)  
Compute LCM of a list of polynomials.`



### Examples

```
>>> from sympy import lcm_list
>>> from sympy.abc import x
```

```
>>> lcm_list([x**3 - 1, x**2 - 1, x**2 - 3*x + 2])
x**5 - x**4 - 2*x**3 - x**2 + x + 2
```

`sympy.polys.polytools.trunc(f, p, *gens, **args)`  
Reduce  $f$  modulo a constant  $p$ .

### Examples

```
>>> from sympy import trunc
>>> from sympy.abc import x
```

```
>>> trunc(2*x**3 + 3*x**2 + 5*x + 7, 3)
-x**3 - x + 1
```

`sympy.polys.polytools.monic(f, *gens, **args)`  
Divide all coefficients of  $f$  by  $\text{LC}(f)$ .

### Examples

```
>>> from sympy import monic
>>> from sympy.abc import x
```

```
>>> monic(3*x**2 + 4*x + 2)
x**2 + 4*x/3 + 2/3
```

`sympy.polys.polytools.content(f, *gens, **args)`  
Compute GCD of coefficients of  $f$ .

### Examples

```
>>> from sympy import content
>>> from sympy.abc import x
```

```
>>> content(6*x**2 + 8*x + 12)
2
```

`sympy.polys.polytools.primitive(f, *gens, **args)`  
Compute content and the primitive form of  $f$ .

### Examples

```
>>> from sympy.polys.polytools import primitive
>>> from sympy.abc import x
```

```
>>> primitive(6*x**2 + 8*x + 12)
(2, 3*x**2 + 4*x + 6)
```

```
>>> eq = (2 + 2*x)*x + 2
```

Expansion is performed by default:

```
>>> primitive(eq)
(2, x**2 + x + 1)
```

Set `expand` to `False` to shut this off. Note that the extraction will not be recursive; use the `as_content_primitive` method for recursive, non-destructive Rational extraction.

```
>>> primitive(eq, expand=False)
(1, x*(2*x + 2) + 2)
```

```
>>> eq.as_content_primitive()
(2, x*(x + 1) + 1)
```

`sympy.polys.polytools.compose(f, g, *gens, **args)`  
Compute functional composition  $f(g)$ .

### Examples

```
>>> from sympy import compose
>>> from sympy.abc import x
```

```
>>> compose(x**2 + x, x - 1)
x**2 - x
```

`sympy.polys.polytools.decompose(f, *gens, **args)`  
Compute functional decomposition of  $f$ .

### Examples

```
>>> from sympy import decompose
>>> from sympy.abc import x
```

```
>>> decompose(x**4 + 2*x**3 - x - 1)
[x**2 - x - 1, x**2 + x]
```

`sympy.polys.polytools.sturm(f, *gens, **args)`  
Compute Sturm sequence of  $f$ .

## Examples

```
>>> from sympy import sturm
>>> from sympy.abc import x
```

```
>>> sturm(x**3 - 2*x**2 + x - 3)
[x**3 - 2*x**2 + x - 3, 3*x**2 - 4*x + 1, 2*x/9 + 25/9, -2079/4]
```

`sympy.polys.polytools.gff_list(f, *gens, **args)`

Compute a list of greatest factorial factors of  $f$ .

Note that the input to `ff()` and `rf()` should be Poly instances to use the definitions here.

## Examples

```
>>> from sympy import gff_list, ff, Poly
>>> from sympy.abc import x
```

```
>>> f = Poly(x**5 + 2*x**4 - x**3 - 2*x**2, x)
```

```
>>> gff_list(f)
[(Poly(x, x, domain='ZZ'), 1), (Poly(x + 2, x, domain='ZZ'), 4)]
```

```
>>> (ff(Poly(x), 1)*ff(Poly(x + 2), 4)) == f
True
```

```
>>> f = Poly(x**12 + 6*x**11 - 11*x**10 - 56*x**9 + 220*x**8 + 208*x**7 -
→ 1401*x**6 + 1090*x**5 + 2715*x**4 - 6720*x**3 - 1092*x**2 +
→ 5040*x, x)
```

```
>>> gff_list(f)
[(Poly(x**3 + 7, x, domain='ZZ'), 2), (Poly(x**2 + 5*x, x, domain='ZZ'),
→ 3)]
```

```
>>> ff(Poly(x**3 + 7, x), 2)*ff(Poly(x**2 + 5*x, x), 3) == f
True
```

`sympy.polys.polytools.gff(f, *gens, **args)`

Compute greatest factorial factorization of  $f$ .

`sympy.polys.polytools.sqf_norm(f, *gens, **args)`

Compute square-free norm of  $f$ .

Returns  $s, f, r$ , such that  $g(x) = f(x - sa)$  and  $r(x) = \text{Norm}(g(x))$  is a square-free polynomial over  $K$ , where  $a$  is the algebraic extension of the ground domain.

### Examples

```
>>> from sympy import sqf_norm, sqrt
>>> from sympy.abc import x
```

```
>>> sqf_norm(x**2 + 1, extension=[sqrt(3)])
(1, x**2 - 2*sqrt(3)*x + 4, x**4 - 4*x**2 + 16)
```

`sympy.polys.polytools.sqf_part(f, *gens, **args)`  
Compute square-free part of  $f$ .

### Examples

```
>>> from sympy import sqf_part
>>> from sympy.abc import x
```

```
>>> sqf_part(x**3 - 3*x - 2)
x**2 - x - 2
```

`sympy.polys.polytools.sqf_list(f, *gens, **args)`  
Compute a list of square-free factors of  $f$ .

### Examples

```
>>> from sympy import sqf_list
>>> from sympy.abc import x
```

```
>>> sqf_list(2*x**5 + 16*x**4 + 50*x**3 + 76*x**2 + 56*x + 16)
(2, [(x + 1, 2), (x + 2, 3)])
```

`sympy.polys.polytools.sqf(f, *gens, **args)`  
Compute square-free factorization of  $f$ .

### Examples

```
>>> from sympy import sqf
>>> from sympy.abc import x
```

```
>>> sqf(2*x**5 + 16*x**4 + 50*x**3 + 76*x**2 + 56*x + 16)
2*(x + 1)**2*(x + 2)**3
```

`sympy.polys.polytools.factor_list(f, *gens, **args)`  
Compute a list of irreducible factors of  $f$ .

## Examples

```
>>> from sympy import factor_list
>>> from sympy.abc import x, y
```

```
>>> factor_list(2*x**5 + 2*x**4*y + 4*x**3 + 4*x**2*y + 2*x + 2*y)
(2, [(x + y, 1), (x**2 + 1, 2)])
```

`sympy.polys.polytools.factor(f, *gens, deep=False, **args)`

Compute the factorization of expression, *f*, into irreducibles. (To factor an integer into primes, use `factorint`.)

There two modes implemented: symbolic and formal. If *f* is not an instance of *Poly* (page 2378) and generators are not specified, then the former mode is used. Otherwise, the formal mode is used.

In symbolic mode, *factor()* (page 2373) will traverse the expression tree and factor its components without any prior expansion, unless an instance of *Add* (page 1013) is encountered (in this case formal factorization is used). This way *factor()* (page 2373) can handle large or symbolic exponents.

By default, the factorization is computed over the rationals. To factor over other domain, e.g. an algebraic or finite field, use appropriate options: `extension`, `modulus` or `domain`.

## Examples

```
>>> from sympy import factor, sqrt, exp
>>> from sympy.abc import x, y
```

```
>>> factor(2*x**5 + 2*x**4*y + 4*x**3 + 4*x**2*y + 2*x + 2*y)
2*(x + y)*(x**2 + 1)**2
```

```
>>> factor(x**2 + 1)
x**2 + 1
>>> factor(x**2 + 1, modulus=2)
(x + 1)**2
>>> factor(x**2 + 1, gaussian=True)
(x - I)*(x + I)
```

```
>>> factor(x**2 - 2, extension=sqrt(2))
(x - sqrt(2))*(x + sqrt(2))
```

```
>>> factor((x**2 - 1)/(x**2 + 4*x + 4))
(x - 1)*(x + 1)/(x + 2)**2
>>> factor((x**2 + 4*x + 4)**1000000*(x**2 + 1))
(x + 2)**2000000*(x**2 + 1)
```

By default, `factor` deals with an expression as a whole:

```
>>> eq = 2*(x**2 + 2*x + 1)
>>> factor(eq)
2*(x**2 + 2*x + 1)
```

If the `deep` flag is `True` then subexpressions will be factored:

```
>>> factor(eq, deep=True)
2**((x + 1)**2)
```

If the `fraction` flag is `False` then rational expressions will not be combined. By default it is `True`.

```
>>> factor(5*x + 3*exp(2 - 7*x), deep=True)
(5*x*exp(7*x) + 3*exp(2))*exp(-7*x)
>>> factor(5*x + 3*exp(2 - 7*x), deep=True, fraction=False)
5*x + 3*exp(2)*exp(-7*x)
```

**See also:**

[`sympy.ntheory.factor\_.factorint`](#) (page 1493)

`sympy.polys.polytools.intervals`(*F*, *all*=*False*, *eps*=*None*, *inf*=*None*, *sup*=*None*, *strict*=*False*, *fast*=*False*, *sqf*=*False*)

Compute isolating intervals for roots of *f*.

### Examples

```
>>> from sympy import intervals
>>> from sympy.abc import x
```

```
>>> intervals(x**2 - 3)
[((-2, -1), 1), ((1, 2), 1)]
>>> intervals(x**2 - 3, eps=1e-2)
[((-26/15, -19/11), 1), ((19/11, 26/15), 1)]
```

`sympy.polys.polytools.refine_root`(*f*, *s*, *t*, *eps*=*None*, *steps*=*None*, *fast*=*False*, *check\_sqf*=*False*)

Refine an isolating interval of a root to the given precision.

### Examples

```
>>> from sympy import refine_root
>>> from sympy.abc import x
```

```
>>> refine_root(x**2 - 3, 1, 2, eps=1e-2)
(19/11, 26/15)
```

`sympy.polys.polytools.count_roots`(*f*, *inf*=*None*, *sup*=*None*)

Return the number of roots of *f* in [*inf*, *sup*] interval.

If one of *inf* or *sup* is complex, it will return the number of roots in the complex rectangle with corners at *inf* and *sup*.

### Examples

```
>>> from sympy import count_roots, I
>>> from sympy.abc import x
```

```
>>> count_roots(x**4 - 4, -3, 3)
2
>>> count_roots(x**4 - 4, 0, 1 + 3*I)
1
```

`sympy.polys.polytools.real_roots(f, multiple=True)`

Return a list of real roots with multiplicities of  $f$ .

### Examples

```
>>> from sympy import real_roots
>>> from sympy.abc import x
```

```
>>> real_roots(2*x**3 - 7*x**2 + 4*x + 4)
[-1/2, 2, 2]
```

`sympy.polys.polytools.nroots(f, n=15, maxsteps=50, cleanup=True)`

Compute numerical approximations of roots of  $f$ .

### Examples

```
>>> from sympy import nroots
>>> from sympy.abc import x
```

```
>>> nroots(x**2 - 3, n=15)
[-1.73205080756888, 1.73205080756888]
>>> nroots(x**2 - 3, n=30)
[-1.73205080756887729352744634151, 1.73205080756887729352744634151]
```

`sympy.polys.polytools.ground_roots(f, *gens, **args)`

Compute roots of  $f$  by factorization in the ground domain.

### Examples

```
>>> from sympy import ground_roots
>>> from sympy.abc import x
```

```
>>> ground_roots(x**6 - 4*x**4 + 4*x**3 - x**2)
{0: 2, 1: 2}
```

`sympy.polys.polytools.nth_power_roots_poly(f, n, *gens, **args)`

Construct a polynomial with  $n$ -th powers of roots of  $f$ .

### Examples

```
>>> from sympy import nth_power_roots_poly, factor, roots
>>> from sympy.abc import x
```

```
>>> f = x**4 - x**2 + 1
>>> g = factor(nth_power_roots_poly(f, 2))
```

```
>>> g
(x**2 - x + 1)**2
```

```
>>> R_f = [(r**2).expand() for r in roots(f)]
>>> R_g = roots(g).keys()
```

```
>>> set(R_f) == set(R_g)
True
```

`sympy.polys.polytools.cancel(f, *gens, _signsimp=True, **args)`  
Cancel common factors in a rational function `f`.

### Examples

```
>>> from sympy import cancel, sqrt, Symbol, together
>>> from sympy.abc import x
>>> A = Symbol('A', commutative=False)
```

```
>>> cancel((2*x**2 - 2)/(x**2 - 2*x + 1))
(2*x + 2)/(x - 1)
>>> cancel((sqrt(3) + sqrt(15)*A)/(sqrt(2) + sqrt(10)*A))
sqrt(6)/2
```

Note: due to automatic distribution of Rationals, a sum divided by an integer will appear as a sum. To recover a rational form use *together* on the result:

```
>>> cancel(x/2 + 1)
x/2 + 1
>>> together(_)
(x + 2)/2
```

`sympy.polys.polytools.reduced(f, G, *gens, **args)`

Reduces a polynomial `f` modulo a set of polynomials `G`.

Given a polynomial `f` and a set of polynomials `G = (g_1, ..., g_n)`, computes a set of quotients `q = (q_1, ..., q_n)` and the remainder `r` such that  $f = q_1 g_1 + \dots + q_n g_n + r$ , where `r` vanishes or `r` is a completely reduced polynomial with respect to `G`.



## Examples

```
>>> from sympy import reduced
>>> from sympy.abc import x, y
```

```
>>> reduced(2*x**4 + y**2 - x**2 + y**3, [x**3 - x, y**3 - y])
([2*x, 1], x**2 + y**2 + y)
```

`sympy.polys.polytools.groebner(F, *gens, **args)`

Computes the reduced Groebner basis for a set of polynomials.

Use the `order` argument to set the monomial ordering that will be used to compute the basis. Allowed orders are `lex`, `grlex` and `grevlex`. If no order is specified, it defaults to `lex`.

For more information on Groebner bases, see the references and the docstring of [`solve\_poly\_system\(\)`](#) (page 856).

## Examples

Example taken from [1].

```
>>> from sympy import groebner
>>> from sympy.abc import x, y
```

```
>>> F = [x*y - 2*y, 2*y**2 - x**2]
```

```
>>> groebner(F, x, y, order='lex')
GroebnerBasis([x**2 - 2*y**2, x*y - 2*y, y**3 - 2*y], x, y,
 domain='ZZ', order='lex')
>>> groebner(F, x, y, order='grlex')
GroebnerBasis([y**3 - 2*y, x**2 - 2*y**2, x*y - 2*y], x, y,
 domain='ZZ', order='grlex')
>>> groebner(F, x, y, order='grevlex')
GroebnerBasis([y**3 - 2*y, x**2 - 2*y**2, x*y - 2*y], x, y,
 domain='ZZ', order='grevlex')
```

By default, an improved implementation of the Buchberger algorithm is used. Optionally, an implementation of the F5B algorithm can be used. The algorithm can be set using the method flag or with the [`sympy.polys.polyconfig.setup\(\)`](#) (page 2643) function.

```
>>> F = [x**2 - x - 1, (2*x - 1) * y - (x**10 - (1 - x)**10)]
```

```
>>> groebner(F, x, y, method='buchberger')
GroebnerBasis([x**2 - x - 1, y - 55], x, y, domain='ZZ', order='lex')
>>> groebner(F, x, y, method='f5b')
GroebnerBasis([x**2 - x - 1, y - 55], x, y, domain='ZZ', order='lex')
```

## References

1. [Buchberger01]
2. [Cox97]

[Buchberger01], [Cox97]

`sympy.polys.polytools.is_zero_dimensional(F, *gens, **args)`

Checks if the ideal generated by a Groebner basis is zero-dimensional.

The algorithm checks if the set of monomials not divisible by the leading monomial of any element of  $F$  is bounded.

## References

David A. Cox, John B. Little, Donal O'Shea. Ideals, Varieties and Algorithms, 3rd edition, p. 230

**class** `sympy.polys.polytools.Poly(rep, *gens, **args)`

Generic class for representing and operating on polynomial expressions.

See [Polynomial Manipulation](#) (page 2341) for general documentation.

Poly is a subclass of Basic rather than Expr but instances can be converted to Expr with the `as_expr()` (page 2383) method.

Deprecated since version 1.6: Combining Poly with non-Poly objects in binary operations is deprecated. Explicitly convert both objects to either Poly or Expr first. See [Mixing Poly and non-polynomial expressions in binary operations](#) (page 175).

## Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

Create a univariate polynomial:

```
>>> Poly(x*(x**2 + x - 1)**2)
Poly(x**5 + 2*x**4 - x**3 - 2*x**2 + x, x, domain='ZZ')
```

Create a univariate polynomial with specific domain:

```
>>> from sympy import sqrt
>>> Poly(x**2 + 2*x + sqrt(3), domain='R')
Poly(1.0*x**2 + 2.0*x + 1.73205080756888, x, domain='RR')
```

Create a multivariate polynomial:

```
>>> Poly(y*x**2 + x*y + 1)
Poly(x**2*y + x*y + 1, x, y, domain='ZZ')
```

Create a univariate polynomial, where  $y$  is a constant:

```
>>> Poly(y*x**2 + x*y + 1, x)
Poly(y*x**2 + y*x + 1, x, domain='ZZ[y]')
```

You can evaluate the above polynomial as a function of y:

```
>>> Poly(y*x**2 + x*y + 1, x).eval(2)
6*y + 1
```

#### See also:

[\*sympy.core.expr.Expr\*](#) (page 947)

**EC**(*order=None*)

Returns the last non-zero coefficient of f.

#### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**3 + 2*x**2 + 3*x, x).EC()
3
```

**EM**(*order=None*)

Returns the last non-zero monomial of f.

#### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(4*x**2 + 2*x*y**2 + x*y + 3*y, x, y).EM()
x**0*y**1
```

**ET**(*order=None*)

Returns the last non-zero term of f.

#### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(4*x**2 + 2*x*y**2 + x*y + 3*y, x, y).ET()
(x**0*y**1, 3)
```

**LC**(*order=None*)

Returns the leading coefficient of f.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(4*x**3 + 2*x**2 + 3*x, x).LC()
4
```

**LM**(*order=None*)

Returns the leading monomial of *f*.

The Leading monomial signifies the monomial having the highest power of the principal generator in the expression *f*.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(4*x**2 + 2*x*y**2 + x*y + 3*y, x, y).LM()
x**2*y**0
```

**LT**(*order=None*)

Returns the leading term of *f*.

The Leading term signifies the term having the highest power of the principal generator in the expression *f* along with its coefficient.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(4*x**2 + 2*x*y**2 + x*y + 3*y, x, y).LT()
(x**2*y**0, 4)
```

**TC**()

Returns the trailing coefficient of *f*.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**3 + 2*x**2 + 3*x, x).TC()
0
```

**abs**()

Make all coefficients in *f* positive.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 1, x).abs()
Poly(x**2 + 1, x, domain='ZZ')
```

#### **add(*g*)**

Add two polynomials *f* and *g*.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).add(Poly(x - 2, x))
Poly(x**2 + x - 1, x, domain='ZZ')
```

```
>>> Poly(x**2 + 1, x) + Poly(x - 2, x)
Poly(x**2 + x - 1, x, domain='ZZ')
```

#### **add\_ground(*coeff*)**

Add an element of the ground domain to *f*.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x + 1).add_ground(2)
Poly(x + 3, x, domain='ZZ')
```

#### **all\_coeffs()**

Returns all coefficients from a univariate polynomial *f*.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**3 + 2*x - 1, x).all_coeffs()
[1, 0, 2, -1]
```

#### **all\_monoms()**

Returns all monomials from a univariate polynomial *f*.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**3 + 2*x - 1, x).all_monoms()
[(3,), (2,), (1,), (0,)]
```

#### See also:

[`all\_terms`](#) (page 2382)

**`all_roots`**(*multiple=True, radicals=True*)

Return a list of real and complex roots with multiplicities.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(2*x**3 - 7*x**2 + 4*x + 4).all_roots()
[-1/2, 2, 2]
>>> Poly(x**3 + x + 1).all_roots()
[CRootOf(x**3 + x + 1, 0),
 CRootOf(x**3 + x + 1, 1),
 CRootOf(x**3 + x + 1, 2)]
```

**`all_terms`**()

Returns all terms from a univariate polynomial *f*.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**3 + 2*x - 1, x).all_terms()
[((3,), 1), ((2,), 0), ((1,), 2), ((0,), -1)]
```

**`as_dict`**(*native=False, zero=False*)

Switch to a dict representation.

## Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + 2*x*y**2 - y, x, y).as_dict()
{(0, 1): -1, (1, 2): 2, (2, 0): 1}
```

### **as\_expr(\*gens)**

Convert a Poly instance to an Expr instance.

## Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> f = Poly(x**2 + 2*x*y**2 - y, x, y)
```

```
>>> f.as_expr()
x**2 + 2*x*y**2 - y
>>> f.as_expr({x: 5})
10*y**2 - y + 25
>>> f.as_expr(5, 6)
379
```

### **as\_list(native=False)**

Switch to a list representation.

### **as\_poly(\*gens, \*\*args)**

Converts self to a polynomial or returns None.

```
>>> from sympy import sin
>>> from sympy.abc import x, y
```

```
>>> print((x**2 + x*y).as_poly())
Poly(x**2 + x*y, x, y, domain='ZZ')
```

```
>>> print((x**2 + x*y).as_poly(x, y))
Poly(x**2 + x*y, x, y, domain='ZZ')
```

```
>>> print((x**2 + sin(y)).as_poly(x, y))
None
```

### **cancel(g, include=False)**

Cancel common factors in a rational function f/g.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(2*x**2 - 2, x).cancel(Poly(x**2 - 2*x + 1, x))
(1, Poly(2*x + 2, x, domain='ZZ'), Poly(x - 1, x, domain='ZZ'))
```

```
>>> Poly(2*x**2 - 2, x).cancel(Poly(x**2 - 2*x + 1, x), include=True)
(Poly(2*x + 2, x, domain='ZZ'), Poly(x - 1, x, domain='ZZ'))
```

**clear\_denoms**(*convert=False*)

Clear denominators, but keep the ground domain.

### Examples

```
>>> from sympy import Poly, S, QQ
>>> from sympy.abc import x
```

```
>>> f = Poly(x/2 + S(1)/3, x, domain=QQ)
```

```
>>> f.clear_denoms()
(6, Poly(3*x + 2, x, domain='QQ'))
>>> f.clear_denoms(convert=True)
(6, Poly(3*x + 2, x, domain='ZZ'))
```

**coeff\_monomial**(*monom*)

Returns the coefficient of monom in f if there, else None.

### Examples

```
>>> from sympy import Poly, exp
>>> from sympy.abc import x, y
```

```
>>> p = Poly(24*x*y*exp(8) + 23*x, x, y)
```

```
>>> p.coeff_monomial(x)
23
>>> p.coeff_monomial(y)
0
>>> p.coeff_monomial(x*y)
24*exp(8)
```

Note that `Expr.coeff()` behaves differently, collecting terms if possible; the `Poly` must be converted to an `Expr` to use that method, however:



```
>>> p.as_expr().coeff(x)
24*y*exp(8) + 23
>>> p.as_expr().coeff(y)
24*x*exp(8)
>>> p.as_expr().coeff(x*y)
24*exp(8)
```

**See also:**

[\*nth\*](#) (page 2408)

more efficient query using exponents of the monomial's generators

**coeffs**(*order=None*)

Returns all non-zero coefficients from *f* in lex order.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**3 + 2*x + 3, x).coeffs()
[1, 2, 3]
```

**See also:**

[\*all\\_coeffs\*](#) (page 2381), [\*coeff\\_monomial\*](#) (page 2384), [\*nth\*](#) (page 2408)

**cofactors**(*g*)

Returns the GCD of *f* and *g* and their cofactors.

Returns polynomials (*h*, *cff*, *cfg*) such that  $h = \gcd(f, g)$ , and  $cff = \text{quo}(f, h)$  and  $cfg = \text{quo}(g, h)$  are, so called, cofactors of *f* and *g*.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 1, x).cofactors(Poly(x**2 - 3*x + 2, x))
(Poly(x - 1, x, domain='ZZ'),
 Poly(x + 1, x, domain='ZZ'),
 Poly(x - 2, x, domain='ZZ'))
```

**compose**(*g*)

Computes the functional composition of *f* and *g*.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + x, x).compose(Poly(x - 1, x))
Poly(x**2 - x, x, domain='ZZ')
```

#### **content()**

Returns the GCD of polynomial coefficients.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(6*x**2 + 8*x + 12, x).content()
2
```

#### **count\_roots(*inf=None, sup=None*)**

Return the number of roots of  $f$  in  $[inf, sup]$  interval.

### Examples

```
>>> from sympy import Poly, I
>>> from sympy.abc import x
```

```
>>> Poly(x**4 - 4, x).count_roots(-3, 3)
2
>>> Poly(x**4 - 4, x).count_roots(0, 1 + 3*I)
1
```

#### **decompose()**

Computes a functional decomposition of  $f$ .

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**4 + 2*x**3 - x - 1, x, domain='ZZ').decompose()
[Poly(x**2 - x - 1, x, domain='ZZ'), Poly(x**2 + x, x, domain='ZZ')]
```

#### **deflate()**

Reduce degree of  $f$  by mapping  $x_i^{**m}$  to  $y_i$ .

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**6*y**2 + x**3 + 1, x, y).deflate()
((3, 2), Poly(x**2*y + x + 1, x, y, domain='ZZ'))
```

#### **degree(*gen=0*)**

Returns degree of  $f$  in  $x_j$ .

The degree of 0 is negative infinity.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + y*x + 1, x, y).degree()
2
>>> Poly(x**2 + y*x + y, x, y).degree(y)
1
>>> Poly(0, x).degree()
-oo
```

#### **degree\_list()**

Returns a list of degrees of  $f$ .

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + y*x + 1, x, y).degree_list()
(2, 1)
```

#### **diff(\**specs*, \*\**kwargs*)**

Computes partial derivative of  $f$ .

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + 2*x + 1, x).diff()
Poly(2*x + 2, x, domain='ZZ')
```

```
>>> Poly(x*y**2 + x, x, y).diff((0, 0), (1, 1))
Poly(2*x*y, x, y, domain='ZZ')
```

**discriminant()**

Computes the discriminant of  $f$ .

**Examples**

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 2*x + 3, x).discriminant()
-8
```

**dispersion( $g=None$ )**

Compute the *dispersion* of polynomials.

For two polynomials  $f(x)$  and  $g(x)$  with  $\deg f > 0$  and  $\deg g > 0$  the dispersion  $\text{dis}(f, g)$  is defined as:

$$\begin{aligned}\text{dis}(f, g) &:= \max\{J(f, g) \cup \{0\}\} \\ &= \max\{a \in \mathbb{N} \mid \gcd(f(x), g(x+a)) \neq 1\} \cup \{0\}\end{aligned}$$

and for a single polynomial  $\text{dis}(f) := \text{dis}(f, f)$ .

**Examples**

```
>>> from sympy import poly
>>> from sympy.polys.dispersion import dispersion, dispersionset
>>> from sympy.abc import x
```

Dispersion set and dispersion of a simple polynomial:

```
>>> fp = poly((x - 3)*(x + 3), x)
>>> sorted(dispersionset(fp))
[0, 6]
>>> dispersion(fp)
6
```

Note that the definition of the dispersion is not symmetric:

```
>>> fp = poly(x**4 - 3*x**2 + 1, x)
>>> gp = fp.shift(-3)
>>> sorted(dispersionset(fp, gp))
[2, 3, 4]
>>> dispersion(fp, gp)
4
>>> sorted(dispersionset(gp, fp))
[]
>>> dispersion(gp, fp)
-oo
```

Computing the dispersion also works over field extensions:

```
>>> from sympy import sqrt
>>> fp = poly(x**2 + sqrt(5)*x - 1, x, domain='QQ<sqrt(5)>')
>>> gp = poly(x**2 + (2 + sqrt(5))*x + sqrt(5), x, domain='QQ<sqrt(5)>')
>>> sorted(dispersionset(fp, gp))
[2]
>>> sorted(dispersionset(gp, fp))
[1, 4]
```

We can even perform the computations for polynomials having symbolic coefficients:

```
>>> from sympy.abc import a
>>> fp = poly(4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2 + 2*a)*x, x)
>>> sorted(dispersionset(fp))
[0, 1]
```

**See also:**

[`dispersionset`](#) (page 2389)

## References

1. [ManWright94]
2. [Koepf98]
3. [Abramov71]
4. [Man93]

[ManWright94], [Koepf98], [Abramov71], [Man93]

**`dispersionset(g=None)`**

Compute the *dispersion set* of two polynomials.

For two polynomials  $f(x)$  and  $g(x)$  with  $\deg f > 0$  and  $\deg g > 0$  the dispersion set  $J(f, g)$  is defined as:

$$\begin{aligned} J(f, g) &:= \{a \in \mathbb{N}_0 \mid \gcd(f(x), g(x+a)) \neq 1\} \\ &= \{a \in \mathbb{N}_0 \mid \deg \gcd(f(x), g(x+a)) \geq 1\} \end{aligned}$$

For a single polynomial one defines  $J(f) := J(f, f)$ .

## Examples

```
>>> from sympy import poly
>>> from sympy.polys.dispersion import dispersion, dispersionset
>>> from sympy.abc import x
```

Dispersion set and dispersion of a simple polynomial:

```
>>> fp = poly((x - 3)*(x + 3), x)
>>> sorted(dispersionset(fp))
[0, 6]
>>> dispersion(fp)
6
```

Note that the definition of the dispersion is not symmetric:

```
>>> fp = poly(x**4 - 3*x**2 + 1, x)
>>> gp = fp.shift(-3)
>>> sorted(dispersionset(fp, gp))
[2, 3, 4]
>>> dispersion(fp, gp)
4
>>> sorted(dispersionset(gp, fp))
[]
>>> dispersion(gp, fp)
-oo
```

Computing the dispersion also works over field extensions:

```
>>> from sympy import sqrt
>>> fp = poly(x**2 + sqrt(5)*x - 1, x, domain='QQ<sqrt(5)>')
>>> gp = poly(x**2 + (2 + sqrt(5))*x + sqrt(5), x, domain='QQ<sqrt(5)>')
>>> sorted(dispersionset(fp, gp))
[2]
>>> sorted(dispersionset(gp, fp))
[1, 4]
```

We can even perform the computations for polynomials having symbolic coefficients:

```
>>> from sympy.abc import a
>>> fp = poly(4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2 + 2*a)*x, x)
>>> sorted(dispersionset(fp))
[0, 1]
```

**See also:**

[\*dispersion\*](#) (page 2388)

## References

1. [ManWright94]
2. [Koepf98]
3. [Abramov71]
4. [Man93]

[ManWright94], [Koepf98], [Abramov71], [Man93]

**div**(*g*, *auto=True*)

Polynomial division with remainder of *f* by *g*.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).div(Poly(2*x - 4, x))
(Poly(1/2*x + 1, x, domain='QQ'), Poly(5, x, domain='QQ'))
```

```
>>> Poly(x**2 + 1, x).div(Poly(2*x - 4, x), auto=False)
(Poly(0, x, domain='ZZ'), Poly(x**2 + 1, x, domain='ZZ'))
```

**property domain**

Get the ground domain of a *Poly* (page 2378)

**Returns**

*Domain* (page 2504):

Ground domain of the *Poly* (page 2378).

### Examples

```
>>> from sympy import Poly, Symbol
>>> x = Symbol('x')
>>> p = Poly(x**2 + x)
>>> p
Poly(x**2 + x, x, domain='ZZ')
>>> p.domain
ZZ
```

**eject**(\**gens*)

Eject selected generators into the ground domain.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> f = Poly(x**2*y + x*y**3 + x*y + 1, x, y)
```

```
>>> f.eject(x)
Poly(x*y**3 + (x**2 + x)*y + 1, y, domain='ZZ[x]')
>>> f.eject(y)
Poly(y*x**2 + (y**3 + y)*x + 1, x, domain='ZZ[y]')
```

**eval**(*x*, *a=None*, *auto=True*)

Evaluate *f* at *a* in the given variable.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y, z
```

```
>>> Poly(x**2 + 2*x + 3, x).eval(2)
11
```

```
>>> Poly(2*x*y + 3*x + y + 2, x, y).eval(x, 2)
Poly(5*y + 8, y, domain='ZZ')
```

```
>>> f = Poly(2*x*y + 3*x + y + 2*z, x, y, z)
```

```
>>> f.eval({x: 2})
Poly(5*y + 2*z + 6, y, z, domain='ZZ')
>>> f.eval({x: 2, y: 5})
Poly(2*z + 31, z, domain='ZZ')
>>> f.eval({x: 2, y: 5, z: 7})
45
```

```
>>> f.eval((2, 5))
Poly(2*z + 31, z, domain='ZZ')
>>> f(2, 5)
Poly(2*z + 31, z, domain='ZZ')
```

#### **exclude()**

Remove unnecessary generators from *f*.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import a, b, c, d, x
```

```
>>> Poly(a + x, a, b, c, d, x).exclude()
Poly(a + x, a, x, domain='ZZ')
```

#### **exquo(*g*, *auto*=True)**

Computes polynomial exact quotient of *f* by *g*.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 1, x).exquo(Poly(x - 1, x))
Poly(x + 1, x, domain='ZZ')
```



```
>>> Poly(x**2 + 1, x).exquo(Poly(2*x - 4, x))
Traceback (most recent call last):
...
ExactQuotientFailed: 2*x - 4 does not divide x**2 + 1
```

**exquo\_ground(*coeff*)**

Exact quotient of  $f$  by a an element of the ground domain.

**Examples**

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(2*x + 4).exquo_ground(2)
Poly(x + 2, x, domain='ZZ')
```

```
>>> Poly(2*x + 3).exquo_ground(2)
Traceback (most recent call last):
...
ExactQuotientFailed: 2 does not divide 3 in ZZ
```

**factor\_list()**

Returns a list of irreducible factors of  $f$ .

**Examples**

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> f = 2*x**5 + 2*x**4*y + 4*x**3 + 4*x**2*y + 2*x + 2*y
```

```
>>> Poly(f).factor_list()
(2, [(Poly(x + y, x, y, domain='ZZ'), 1),
 (Poly(x**2 + 1, x, y, domain='ZZ'), 2)])
```

**factor\_list\_include()**

Returns a list of irreducible factors of  $f$ .

**Examples**

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> f = 2*x**5 + 2*x**4*y + 4*x**3 + 4*x**2*y + 2*x + 2*y
```

```
>>> Poly(f).factor_list_include()
[(Poly(2*x + 2*y, x, y, domain='ZZ'), 1),
 (Poly(x**2 + 1, x, y, domain='ZZ'), 2)]
```

**property free\_symbols**

Free symbols of a polynomial expression.

**Examples**

```
>>> from sympy import Poly
>>> from sympy.abc import x, y, z
```

```
>>> Poly(x**2 + 1).free_symbols
{x}
>>> Poly(x**2 + y).free_symbols
{x, y}
>>> Poly(x**2 + y, x).free_symbols
{x, y}
>>> Poly(x**2 + y, x, z).free_symbols
{x, y}
```

**property free\_symbols\_in\_domain**

Free symbols of the domain of self.

**Examples**

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + 1).free_symbols_in_domain
set()
>>> Poly(x**2 + y).free_symbols_in_domain
set()
>>> Poly(x**2 + y, x).free_symbols_in_domain
{y}
```

**classmethod from\_dict(*rep*, \**gens*, \*\**args*)**

Construct a polynomial from a dict.

**classmethod from\_expr(*rep*, \**gens*, \*\**args*)**

Construct a polynomial from an expression.

**classmethod from\_list(*rep*, \**gens*, \*\**args*)**

Construct a polynomial from a list.

**classmethod from\_poly(*rep*, \**gens*, \*\**args*)**

Construct a polynomial from a polynomial.

**gcd(*g*)**

Returns the polynomial GCD of *f* and *g*.

## Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 1, x).gcd(Poly(x**2 - 3*x + 2, x))
Poly(x - 1, x, domain='ZZ')
```

### **gcdex**(*g*, *auto*=True)

Extended Euclidean algorithm of *f* and *g*.

Returns (*s*, *t*, *h*) such that  $h = \gcd(f, g)$  and  $s*f + t*g = h$ .

## Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> f = x**4 - 2*x**3 - 6*x**2 + 12*x + 15
>>> g = x**3 + x**2 - 4*x - 4
```

```
>>> Poly(f).gcdex(Poly(g))
(Poly(-1/5*x + 3/5, x, domain='QQ'),
 Poly(1/5*x**2 - 6/5*x + 2, x, domain='QQ'),
 Poly(x + 1, x, domain='QQ'))
```

### **property gen**

Return the principal generator.

## Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).gen
x
```

### **get\_domain()**

Get the ground domain of *f*.

### **get\_modulus()**

Get the modulus of *f*.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, modulus=2).get_modulus()
2
```

### `gff_list()`

Computes greatest factorial factorization of  $f$ .

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> f = x**5 + 2*x**4 - x**3 - 2*x**2
```

```
>>> Poly(f).gff_list()
[(Poly(x, x, domain='ZZ'), 1), (Poly(x + 2, x, domain='ZZ'), 4)]
```

### `ground_roots()`

Compute roots of  $f$  by factorization in the ground domain.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**6 - 4*x**4 + 4*x**3 - x**2).ground_roots()
{0: 2, 1: 2}
```

### `half_gcdex(g, auto=True)`

Half extended Euclidean algorithm of  $f$  and  $g$ .

Returns  $(s, h)$  such that  $h = \gcd(f, g)$  and  $s*f = h \pmod{g}$ .

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> f = x**4 - 2*x**3 - 6*x**2 + 12*x + 15
>>> g = x**3 + x**2 - 4*x - 4
```

```
>>> Poly(f).half_gcdex(Poly(g))
(Poly(-1/5*x + 3/5, x, domain='QQ'), Poly(x + 1, x, domain='QQ'))
```

**has\_only\_gens(\*gens)**

Return True if `Poly(f, *gens)` retains ground domain.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y, z
```

```
>>> Poly(x*y + 1, x, y, z).has_only_gens(x, y)
True
>>> Poly(x*y + z, x, y, z).has_only_gens(x, y)
False
```

**homogeneous\_order()**

Returns the homogeneous order of `f`.

A homogeneous polynomial is a polynomial whose all monomials with non-zero coefficients have the same total degree. This degree is the homogeneous order of `f`. If you only want to check if a polynomial is homogeneous, then use `Poly.is_homogeneous()` (page 2400).

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> f = Poly(x**5 + 2*x**3*y**2 + 9*x*y**4)
>>> f.homogeneous_order()
5
```

**homogenize(s)**

Returns the homogeneous polynomial of `f`.

A homogeneous polynomial is a polynomial whose all monomials with non-zero coefficients have the same total degree. If you only want to check if a polynomial is homogeneous, then use `Poly.is_homogeneous()` (page 2400). If you want not only to check if a polynomial is homogeneous but also compute its homogeneous order, then use `Poly.homogeneous_order()` (page 2397).

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y, z
```

```
>>> f = Poly(x**5 + 2*x**2*y**2 + 9*x*y**3)
>>> f.homogenize(z)
Poly(x**5 + 2*x**2*y**2*z + 9*x*y**3*z, x, y, z, domain='ZZ')
```

**inject(front=False)**

Inject ground domain generators into `f`.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> f = Poly(x**2*y + x*y**3 + x*y + 1, x)
```

```
>>> f.inject()
Poly(x**2*y + x*y**3 + x*y + 1, x, y, domain='ZZ')
>>> f.inject(front=True)
Poly(y**3*x + y*x**2 + y*x + 1, y, x, domain='ZZ')
```

**integrate**(*\*specs, \*\*args*)

Computes indefinite integral of *f*.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + 2*x + 1, x).integrate()
Poly(1/3*x**3 + x**2 + x, x, domain='QQ')
```

```
>>> Poly(x*y**2 + x, x, y).integrate((0, 1), (1, 0))
Poly(1/2*x**2*y**2 + 1/2*x**2, x, y, domain='QQ')
```

**intervals**(*all=False, eps=None, inf=None, sup=None, fast=False, sqf=False*)

Compute isolating intervals for roots of *f*.

For real roots the Vincent-Akritas-Strzebonski (VAS) continued fractions method is used.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 3, x).intervals()
[((-2, -1), 1), ((1, 2), 1)]
>>> Poly(x**2 - 3, x).intervals(eps=1e-2)
[((-26/15, -19/11), 1), ((19/11, 26/15), 1)]
```

## References

**invert**(*g*, *auto=True*)

Invert *f* modulo *g* when possible.

## Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 1, x).invert(Poly(2*x - 1, x))
Poly(-4/3, x, domain='QQ')
```

```
>>> Poly(x**2 - 1, x).invert(Poly(x - 1, x))
Traceback (most recent call last):
...
NotInvertible: zero divisor
```

**property is\_cyclotomic**

Returns True if *f* is a cyclotomic polynomial.

## Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> f = x**16 + x**14 - x**10 + x**8 - x**6 + x**2 + 1
```

```
>>> Poly(f).is_cyclotomic
False
```

```
>>> g = x**16 + x**14 - x**10 - x**8 - x**6 + x**2 + 1
```

```
>>> Poly(g).is_cyclotomic
True
```

**property is\_ground**

Returns True if *f* is an element of the ground domain.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x, x).is_ground
False
>>> Poly(2, x).is_ground
True
>>> Poly(y, x).is_ground
True
```

#### property `is_homogeneous`

Returns True if  $f$  is a homogeneous polynomial.

A homogeneous polynomial is a polynomial whose all monomials with non-zero coefficients have the same total degree. If you want not only to check if a polynomial is homogeneous but also compute its homogeneous order, then use [\*Poly.homogeneous\\_order\(\)\*](#) (page 2397).

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + x*y, x, y).is_homogeneous
True
>>> Poly(x**3 + x*y, x, y).is_homogeneous
False
```

#### property `is_irreducible`

Returns True if  $f$  has no factors over its domain.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + x + 1, x, modulus=2).is_irreducible
True
>>> Poly(x**2 + 1, x, modulus=2).is_irreducible
False
```

#### property `is_linear`

Returns True if  $f$  is linear in all its variables.



### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x + y + 2, x, y).is_linear
True
>>> Poly(x*y + 2, x, y).is_linear
False
```

#### property `is_monic`

Returns True if the leading coefficient of  $f$  is one.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x + 2, x).is_monic
True
>>> Poly(2*x + 2, x).is_monic
False
```

#### property `is_monomial`

Returns True if  $f$  is zero or has only one term.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(3*x**2, x).is_monomial
True
>>> Poly(3*x**2 + 1, x).is_monomial
False
```

#### property `is_multivariate`

Returns True if  $f$  is a multivariate polynomial.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + x + 1, x).is_multivariate
False
>>> Poly(x*y**2 + x*y + 1, x, y).is_multivariate
```

(continues on next page)

(continued from previous page)

```
True
>>> Poly(x*y**2 + x*y + 1, x).is_multivariate
False
>>> Poly(x**2 + x + 1, x, y).is_multivariate
True
```

**property is\_one**

Returns True if  $f$  is a unit polynomial.

**Examples**

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(0, x).is_one
False
>>> Poly(1, x).is_one
True
```

**property is\_primitive**

Returns True if GCD of the coefficients of  $f$  is one.

**Examples**

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(2*x**2 + 6*x + 12, x).is_primitive
False
>>> Poly(x**2 + 3*x + 6, x).is_primitive
True
```

**property is\_quadratic**

Returns True if  $f$  is quadratic in all its variables.

**Examples**

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x*y + 2, x, y).is_quadratic
True
>>> Poly(x*y**2 + 2, x, y).is_quadratic
False
```

**property is\_sqf**

Returns True if  $f$  is a square-free polynomial.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 2*x + 1, x).is_sqf
False
>>> Poly(x**2 - 1, x).is_sqf
True
```

### property `is_univariate`

Returns True if  $f$  is a univariate polynomial.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + x + 1, x).is_univariate
True
>>> Poly(x*y**2 + x*y + 1, x, y).is_univariate
False
>>> Poly(x*y**2 + x*y + 1, x).is_univariate
True
>>> Poly(x**2 + x + 1, x, y).is_univariate
False
```

### property `is_zero`

Returns True if  $f$  is a zero polynomial.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(0, x).is_zero
True
>>> Poly(1, x).is_zero
False
```

### `l1_norm()`

Returns l1 norm of  $f$ .

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(-x**2 + 2*x - 3, x).ll_norm()
6
```

### `lcm(g)`

Returns polynomial LCM of f and g.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 1, x).lcm(Poly(x**2 - 3*x + 2, x))
Poly(x**3 - 2*x**2 - x + 2, x, domain='ZZ')
```

### `length()`

Returns the number of non-zero terms in f.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 2*x - 1).length()
3
```

### `lift()`

Convert algebraic coefficients to rationals.

### Examples

```
>>> from sympy import Poly, I
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + I*x + 1, x, extension=I).lift()
Poly(x**4 + 3*x**2 + 1, x, domain='QQ')
```

### `ltrim(gen)`

Remove dummy generators from f that are to the left of specified gen in the generators as ordered. When gen is an integer, it refers to the generator located at that position within the tuple of generators of f.

## Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y, z
```

```
>>> Poly(y**2 + y*z**2, x, y, z).ltrim(y)
Poly(y**2 + y*z**2, y, z, domain='ZZ')
>>> Poly(z, x, y, z).ltrim(-1)
Poly(z, z, domain='ZZ')
```

**match**(\*args, \*\*kwargs)

Match expression from Poly. See Basic.match()

**max\_norm**()

Returns maximum norm of f.

## Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(-x**2 + 2*x - 3, x).max_norm()
3
```

**monic**(auto=True)

Divides all coefficients by LC(f).

## Examples

```
>>> from sympy import Poly, ZZ
>>> from sympy.abc import x
```

```
>>> Poly(3*x**2 + 6*x + 9, x, domain=ZZ).monic()
Poly(x**2 + 2*x + 3, x, domain='QQ')
```

```
>>> Poly(3*x**2 + 4*x + 2, x, domain=ZZ).monic()
Poly(x**2 + 4/3*x + 2/3, x, domain='QQ')
```

**monoms**(order=None)

Returns all non-zero monomials from f in lex order.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + 2*x*y**2 + x*y + 3*y, x, y).monoms()
[(2, 0), (1, 2), (1, 1), (0, 1)]
```

#### See also:

[\*all\\_monoms\*](#) (page 2381)

#### `mul(g)`

Multiply two polynomials *f* and *g*.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).mul(Poly(x - 2, x))
Poly(x**3 - 2*x**2 + x - 2, x, domain='ZZ')
```

```
>>> Poly(x**2 + 1, x)*Poly(x - 2, x)
Poly(x**3 - 2*x**2 + x - 2, x, domain='ZZ')
```

#### `mul_ground(coeff)`

Multiply *f* by an element of the ground domain.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x + 1).mul_ground(2)
Poly(2*x + 2, x, domain='ZZ')
```

#### `neg()`

Negate all coefficients in *f*.

## Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 1, x).neg()
Poly(-x**2 + 1, x, domain='ZZ')
```

```
>>> -Poly(x**2 - 1, x)
Poly(-x**2 + 1, x, domain='ZZ')
```

**classmethod** `new(rep, *gens)`

Construct `Poly` (page 2378) instance from raw representation.

**norm()**

Computes the product,  $\text{Norm}(f)$ , of the conjugates of a polynomial  $f$  defined over a number field  $K$ .

## Examples

```
>>> from sympy import Poly, sqrt
>>> from sympy.abc import x
```

```
>>> a, b = sqrt(2), sqrt(3)
```

A polynomial over a quadratic extension. Two conjugates  $x - a$  and  $x + a$ .

```
>>> f = Poly(x - a, x, extension=a)
>>> f.norm()
Poly(x**2 - 2, x, domain='QQ')
```

A polynomial over a quartic extension. Four conjugates  $x - a$ ,  $x - a$ ,  $x + a$  and  $x + a$ .

```
>>> f = Poly(x - a, x, extension=(a, b))
>>> f.norm()
Poly(x**4 - 4*x**2 + 4, x, domain='QQ')
```

**roots**(*n=15, maxsteps=50, cleanup=True*)

Compute numerical approximations of roots of  $f$ .

### Parameters

**n ... the number of digits to calculate**

**maxsteps ... the maximum number of iterations to do**

**If the accuracy ``n`` cannot be reached in ``maxsteps``, it will raise an**

**exception. You need to rerun with higher maxsteps.**

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 3).nroots(n=15)
[-1.73205080756888, 1.73205080756888]
>>> Poly(x**2 - 3).nroots(n=30)
[-1.73205080756887729352744634151, 1.73205080756887729352744634151]
```

#### **nth(\*N)**

Returns the n-th coefficient of  $f$  where  $N$  are the exponents of the generators in the term of interest.

### Examples

```
>>> from sympy import Poly, sqrt
>>> from sympy.abc import x, y
```

```
>>> Poly(x**3 + 2*x**2 + 3*x, x).nth(2)
2
>>> Poly(x**3 + 2*x*y**2 + y**2, x, y).nth(1, 2)
2
>>> Poly(4*sqrt(x)*y)
Poly(4*y*(sqrt(x)), y, sqrt(x), domain='ZZ')
>>> _.nth(1, 1)
4
```

#### **See also:**

[\*coeff\\_monomial\*](#) (page 2384)

#### **nth\_power\_roots\_poly(n)**

Construct a polynomial with n-th powers of roots of  $f$ .

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> f = Poly(x**4 - x**2 + 1)
```

```
>>> f.nth_power_roots_poly(2)
Poly(x**4 - 2*x**3 + 3*x**2 - 2*x + 1, x, domain='ZZ')
>>> f.nth_power_roots_poly(3)
Poly(x**4 + 2*x**2 + 1, x, domain='ZZ')
>>> f.nth_power_roots_poly(4)
Poly(x**4 + 2*x**3 + 3*x**2 + 2*x + 1, x, domain='ZZ')
>>> f.nth_power_roots_poly(12)
Poly(x**4 - 4*x**3 + 6*x**2 - 4*x + 1, x, domain='ZZ')
```



**property one**

Return one polynomial with self's properties.

**pdiv(*g*)**

Polynomial pseudo-division of *f* by *g*.

**Examples**

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).pdiv(Poly(2*x - 4, x))
(Poly(2*x + 4, x, domain='ZZ'), Poly(20, x, domain='ZZ'))
```

**per(*rep*, *gens*=None, *remove*=None)**

Create a Poly out of the given representation.

**Examples**

```
>>> from sympy import Poly, ZZ
>>> from sympy.abc import x, y
```

```
>>> from sympy.polys.polyclasses import DMP
```

```
>>> a = Poly(x**2 + 1)
```

```
>>> a.per(DMP([ZZ(1), ZZ(1)], ZZ), gens=[y])
Poly(y + 1, y, domain='ZZ')
```

**pexquo(*g*)**

Polynomial exact pseudo-quotient of *f* by *g*.

**Examples**

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 1, x).pexquo(Poly(2*x - 2, x))
Poly(2*x + 2, x, domain='ZZ')
```

```
>>> Poly(x**2 + 1, x).pexquo(Poly(2*x - 4, x))
Traceback (most recent call last):
...
ExactQuotientFailed: 2*x - 4 does not divide x**2 + 1
```

**pow(*n*)**

Raise *f* to a non-negative power *n*.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x - 2, x).pow(3)
Poly(x**3 - 6*x**2 + 12*x - 8, x, domain='ZZ')
```

```
>>> Poly(x - 2, x)**3
Poly(x**3 - 6*x**2 + 12*x - 8, x, domain='ZZ')
```

#### `pquo(g)`

Polynomial pseudo-quotient of  $f$  by  $g$ .

See the Caveat note in the function `prem(f, g)`.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).pquo(Poly(2*x - 4, x))
Poly(2*x + 4, x, domain='ZZ')
```

```
>>> Poly(x**2 - 1, x).pquo(Poly(2*x - 2, x))
Poly(2*x + 2, x, domain='ZZ')
```

#### `prem(g)`

Polynomial pseudo-remainder of  $f$  by  $g$ .

**Caveat: The function `prem(f, g, x)` can be safely used to compute**  
in  $\mathbb{Z}[x]$  `_only_subresultant` polynomial remainder sequences (prs's).

To safely compute Euclidean and Sturmian prs's in  $\mathbb{Z}[x]$  employ anyone of the corresponding functions found in the module `sympy.polys.subresultants_qq_zz`. The functions in the module with suffix `_pg` compute prs's in  $\mathbb{Z}[x]$  employing `rem(f, g, x)`, whereas the functions with suffix `_amv` compute prs's in  $\mathbb{Z}[x]$  employing `rem_z(f, g, x)`.

The function `rem_z(f, g, x)` differs from `prem(f, g, x)` in that to compute the remainder polynomials in  $\mathbb{Z}[x]$  it premultiplies the dividend times the absolute value of the leading coefficient of the divisor raised to the power `degree(f, x) - degree(g, x) + 1`.

## Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).prem(Poly(2*x - 4, x))
Poly(20, x, domain='ZZ')
```

### **primitive()**

Returns the content and a primitive form of  $f$ .

## Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(2*x**2 + 8*x + 12, x).primitive()
(2, Poly(x**2 + 4*x + 6, x, domain='ZZ'))
```

### **quo( $g$ , $auto=True$ )**

Computes polynomial quotient of  $f$  by  $g$ .

## Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).quo(Poly(2*x - 4, x))
Poly(1/2*x + 1, x, domain='QQ')
```

```
>>> Poly(x**2 - 1, x).quo(Poly(x - 1, x))
Poly(x + 1, x, domain='ZZ')
```

### **quo\_ground( $coeff$ )**

Quotient of  $f$  by a an element of the ground domain.

## Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(2*x + 4).quo_ground(2)
Poly(x + 2, x, domain='ZZ')
```

```
>>> Poly(2*x + 3).quo_ground(2)
Poly(x + 1, x, domain='ZZ')
```

**rat\_clear\_denoms**(*g*)Clear denominators in a rational function  $f/g$ .**Examples**

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> f = Poly(x**2/y + 1, x)
>>> g = Poly(x**3 + y, x)
```

```
>>> p, q = f.rat_clear_denoms(g)
```

```
>>> p
Poly(x**2 + y, x, domain='ZZ[y]')
>>> q
Poly(y*x**3 + y**2, x, domain='ZZ[y]')
```

**real\_roots**(*multiple=True*, *radicals=True*)

Return a list of real roots with multiplicities.

**Examples**

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(2*x**3 - 7*x**2 + 4*x + 4).real_roots()
[-1/2, 2, 2]
>>> Poly(x**3 + x + 1).real_roots()
[CRootOf(x**3 + x + 1, 0)]
```

**refine\_root**(*s*, *t*, *eps=None*, *steps=None*, *fast=False*, *check\_sqf=False*)

Refine an isolating interval of a root to the given precision.

**Examples**

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 3, x).refine_root(1, 2, eps=1e-2)
(19/11, 26/15)
```

**rem**(*g*, *auto=True*)Computes the polynomial remainder of  $f$  by  $g$ .

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).rem(Poly(2*x - 4, x))
Poly(5, x, domain='ZZ')
```

```
>>> Poly(x**2 + 1, x).rem(Poly(2*x - 4, x), auto=False)
Poly(x**2 + 1, x, domain='ZZ')
```

**reorder**(\*gens, \*\*args)

Efficiently apply new order of generators.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + x*y**2, x, y).reorder(y, x)
Poly(y**2*x + x**2, y, x, domain='ZZ')
```

**replace**(x, y=None, \*\*\_ignore)

Replace x with y in generators list.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + 1, x).replace(x, y)
Poly(y**2 + 1, y, domain='ZZ')
```

**resultant**(g, includePRS=False)

Computes the resultant of f and g via PRS.

If includePRS=True, it includes the subresultant PRS in the result. Because the PRS is used to calculate the resultant, this is more efficient than calling *subresultants()* (page 2365) separately.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> f = Poly(x**2 + 1, x)
```

```
>>> f.resultant(Poly(x**2 - 1, x))
4
>>> f.resultant(Poly(x**2 - 1, x), includePRS=True)
(4, [Poly(x**2 + 1, x, domain='ZZ'), Poly(x**2 - 1, x, domain='ZZ'),
 Poly(-2, x, domain='ZZ')])
```

**retract**(*field=None*)

Recalculate the ground domain of a polynomial.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> f = Poly(x**2 + 1, x, domain='QQ[y]')
>>> f
Poly(x**2 + 1, x, domain='QQ[y]')
```

```
>>> f.retract()
Poly(x**2 + 1, x, domain='ZZ')
>>> f.retract(field=True)
Poly(x**2 + 1, x, domain='QQ')
```

**revert**(*n*)

Compute  $f^{**}(-1) \bmod x^{**n}$ .

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(1, x).revert(2)
Poly(1, x, domain='ZZ')
```

```
>>> Poly(1 + x, x).revert(1)
Poly(1, x, domain='ZZ')
```

```
>>> Poly(x**2 - 2, x).revert(2)
Traceback (most recent call last):
...
NotReversible: only units are reversible in a ring
```

```
>>> Poly(1/x, x).revert(1)
Traceback (most recent call last):
...
PolynomialError: 1/x contains an element of the generators set
```

**root**(*index*, *radicals=True*)

Get an indexed root of a polynomial.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> f = Poly(2*x**3 - 7*x**2 + 4*x + 4)
```

```
>>> f.root(0)
-1/2
>>> f.root(1)
2
>>> f.root(2)
2
>>> f.root(3)
Traceback (most recent call last):
...
IndexError: root index out of [-3, 2] range, got 3
```

```
>>> Poly(x**5 + x + 1).root(0)
CRootOf(x**3 - x**2 + 1, 0)
```

**same\_root**(*a*, *b*)

Decide whether two roots of this polynomial are equal.

#### Raises

##### DomainError

If the domain of the polynomial is not [ZZ](#) (page 2525), [QQ](#) (page 2529), [RR](#) (page 2545), or [CC](#) (page 2546).

##### MultivariatePolynomialError

If the polynomial is not univariate.

##### PolynomialError

If the polynomial is of degree < 2.

### Examples

```
>>> from sympy import Poly, cyclotomic_poly, exp, I, pi
>>> f = Poly(cyclotomic_poly(5))
>>> r0 = exp(2*I*pi/5)
>>> indices = [i for i, r in enumerate(f.all_roots()) if f.same_
↳ root(r, r0)]
>>> print(indices)
[3]
```

**set\_domain(*domain*)**

Set the ground domain of *f*.

**set\_modulus(*modulus*)**

Set the modulus of *f*.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(5*x**2 + 2*x - 1, x).set_modulus(2)
Poly(x**2 + 1, x, modulus=2)
```

**shift(*a*)**

Efficiently compute Taylor shift  $f(x + a)$ .

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 2*x + 1, x).shift(2)
Poly(x**2 + 2*x + 1, x, domain='ZZ')
```

**slice(*x*, *m*, *n*=None)**

Take a continuous subsequence of terms of *f*.

**sqf\_list(*all*=False)**

Returns a list of square-free factors of *f*.



## Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> f = 2*x**5 + 16*x**4 + 50*x**3 + 76*x**2 + 56*x + 16
```

```
>>> Poly(f).sqf_list()
(2, [(Poly(x + 1, x, domain='ZZ'), 2),
 (Poly(x + 2, x, domain='ZZ'), 3)])
```

```
>>> Poly(f).sqf_list(all=True)
(2, [(Poly(1, x, domain='ZZ'), 1),
 (Poly(x + 1, x, domain='ZZ'), 2),
 (Poly(x + 2, x, domain='ZZ'), 3)])
```

**sqf\_list\_include**(*all=False*)

Returns a list of square-free factors of *f*.

## Examples

```
>>> from sympy import Poly, expand
>>> from sympy.abc import x
```

```
>>> f = expand(2*(x + 1)**3*x**4)
>>> f
2*x**7 + 6*x**6 + 6*x**5 + 2*x**4
```

```
>>> Poly(f).sqf_list_include()
[(Poly(2, x, domain='ZZ'), 1),
 (Poly(x + 1, x, domain='ZZ'), 3),
 (Poly(x, x, domain='ZZ'), 4)]
```

```
>>> Poly(f).sqf_list_include(all=True)
[(Poly(2, x, domain='ZZ'), 1),
 (Poly(1, x, domain='ZZ'), 2),
 (Poly(x + 1, x, domain='ZZ'), 3),
 (Poly(x, x, domain='ZZ'), 4)]
```

**sqf\_norm**()

Computes square-free norm of *f*.

Returns *s*, *f*, *r*, such that  $g(x) = f(x - sa)$  and  $r(x) = \text{Norm}(g(x))$  is a square-free polynomial over *K*, where *a* is the algebraic extension of the ground domain.

### Examples

```
>>> from sympy import Poly, sqrt
>>> from sympy.abc import x
```

```
>>> s, f, r = Poly(x**2 + 1, x, extension=[sqrt(3)]).sqf_norm()
```

```
>>> s
1
>>> f
Poly(x**2 - 2*sqrt(3)*x + 4, x, domain='QQ<sqrt(3)>')
>>> r
Poly(x**4 - 4*x**2 + 16, x, domain='QQ')
```

#### **sqf\_part()**

Computes square-free part of  $f$ .

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**3 - 3*x - 2, x).sqf_part()
Poly(x**2 - x - 2, x, domain='ZZ')
```

#### **sqr()**

Square a polynomial  $f$ .

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x - 2, x).sqr()
Poly(x**2 - 4*x + 4, x, domain='ZZ')
```

```
>>> Poly(x - 2, x)**2
Poly(x**2 - 4*x + 4, x, domain='ZZ')
```

#### **sturm(auto=True)**

Computes the Sturm sequence of  $f$ .

## Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**3 - 2*x**2 + x - 3, x).sturm()
[Poly(x**3 - 2*x**2 + x - 3, x, domain='QQ'),
 Poly(3*x**2 - 4*x + 1, x, domain='QQ'),
 Poly(2/9*x + 25/9, x, domain='QQ'),
 Poly(-2079/4, x, domain='QQ')]
```

### **sub(*g*)**

Subtract two polynomials *f* and *g*.

## Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).sub(Poly(x - 2, x))
Poly(x**2 - x + 3, x, domain='ZZ')
```

```
>>> Poly(x**2 + 1, x) - Poly(x - 2, x)
Poly(x**2 - x + 3, x, domain='ZZ')
```

### **sub\_ground(*coeff*)**

Subtract an element of the ground domain from *f*.

## Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x + 1).sub_ground(2)
Poly(x - 1, x, domain='ZZ')
```

### **subresultants(*g*)**

Computes the subresultant PRS of *f* and *g*.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).subresultants(Poly(x**2 - 1, x))
[Poly(x**2 + 1, x, domain='ZZ'),
 Poly(x**2 - 1, x, domain='ZZ'),
 Poly(-2, x, domain='ZZ')]
```

**terms**(*order=None*)

Returns all non-zero terms from *f* in lex order.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + 2*x*y**2 + x*y + 3*y, x, y).terms()
[(2, 0), (1, 1), ((1, 2), 2), ((1, 1), 1), ((0, 1), 3)]
```

**See also:**

[\*all\\_terms\*](#) (page 2382)

**terms\_gcd**()

Remove GCD of terms from the polynomial *f*.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**6*y**2 + x**3*y, x, y).terms_gcd()
((3, 1), Poly(x**3*y + 1, x, y, domain='ZZ'))
```

**termwise**(*func, \*gens, \*\*args*)

Apply a function to all terms of *f*.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> def func(k, coeff):
... k = k[0]
... return coeff//10**(2-k)
```

```
>>> Poly(x**2 + 20*x + 400).termwise(func)
Poly(x**2 + 2*x + 4, x, domain='ZZ')
```

**to\_exact()**

Make the ground domain exact.

**Examples**

```
>>> from sympy import Poly, RR
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1.0, x, domain=RR).to_exact()
Poly(x**2 + 1, x, domain='QQ')
```

**to\_field()**

Make the ground domain a field.

**Examples**

```
>>> from sympy import Poly, ZZ
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x, domain=ZZ).to_field()
Poly(x**2 + 1, x, domain='QQ')
```

**to\_ring()**

Make the ground domain a ring.

**Examples**

```
>>> from sympy import Poly, QQ
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, domain=QQ).to_ring()
Poly(x**2 + 1, x, domain='ZZ')
```

**total\_degree()**

Returns the total degree of  $f$ .

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + y*x + 1, x, y).total_degree()
2
>>> Poly(x + y**5, x, y).total_degree()
5
```

### **transform**(*p*, *q*)

Efficiently evaluate the functional transformation  $q^{**n} * f(p/q)$ .

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 2*x + 1, x).transform(Poly(x + 1, x), Poly(x - 1, x))
Poly(4, x, domain='ZZ')
```

### **trunc**(*p*)

Reduce *f* modulo a constant *p*.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(2*x**3 + 3*x**2 + 5*x + 7, x).trunc(3)
Poly(-x**3 - x + 1, x, domain='ZZ')
```

### **unify**(*g*)

Make *f* and *g* belong to the same domain.

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> f, g = Poly(x/2 + 1, x), Poly(2*x + 1)
```

```
>>> f
Poly(1/2*x + 1, x, domain='QQ')
>>> g
Poly(2*x + 1, x, domain='ZZ')
```

```
>>> F, G = f.unify(g)
```

```
>>> F
Poly(1/2*x + 1, x, domain='QQ')
>>> G
Poly(2*x + 1, x, domain='QQ')
```

#### property unit

Return unit polynomial with `self`'s properties.

#### property zero

Return zero polynomial with `self`'s properties.

**class** `sympy.polys.polytools.PurePoly(rep, *gens, **args)`

Class for representing pure polynomials.

#### property free\_symbols

Free symbols of a polynomial.

### Examples

```
>>> from sympy import PurePoly
>>> from sympy.abc import x, y
```

```
>>> PurePoly(x**2 + 1).free_symbols
set()
>>> PurePoly(x**2 + y).free_symbols
set()
>>> PurePoly(x**2 + y, x).free_symbols
{y}
```

**class** `sympy.polys.polytools.GroebnerBasis(F, *gens, **args)`

Represents a reduced Groebner basis.

#### contains(*poly*)

Check if `poly` belongs the ideal generated by `self`.

### Examples

```
>>> from sympy import groebner
>>> from sympy.abc import x, y
```

```
>>> f = 2*x**3 + y**3 + 3*y
>>> G = groebner([x**2 + y**2 - 1, x*y - 2])
```

```
>>> G.contains(f)
True
>>> G.contains(f + 1)
False
```

**fglm(*order*)**

Convert a Groebner basis from one ordering to another.

The FGLM algorithm converts reduced Groebner bases of zero-dimensional ideals from one ordering to another. This method is often used when it is infeasible to compute a Groebner basis with respect to a particular ordering directly.

**Examples**

```
>>> from sympy.abc import x, y
>>> from sympy import groebner
```

```
>>> F = [x**2 - 3*y - x + 1, y**2 - 2*x + y - 1]
>>> G = groebner(F, x, y, order='grlex')
```

```
>>> list(G.fglm('lex'))
[2*x - y**2 - y + 1, y**4 + 2*y**3 - 3*y**2 - 16*y + 7]
>>> list(groebner(F, x, y, order='lex'))
[2*x - y**2 - y + 1, y**4 + 2*y**3 - 3*y**2 - 16*y + 7]
```

**References**

[R719]

**property is\_zero\_dimensional**

Checks if the ideal generated by a Groebner basis is zero-dimensional.

The algorithm checks if the set of monomials not divisible by the leading monomial of any element of  $F$  is bounded.

**References**

David A. Cox, John B. Little, Donal O'Shea. Ideals, Varieties and Algorithms, 3rd edition, p. 230

**reduce(*expr*, *auto*=True)**

Reduces a polynomial modulo a Groebner basis.

Given a polynomial  $f$  and a set of polynomials  $G = (g_1, \dots, g_n)$ , computes a set of quotients  $q = (q_1, \dots, q_n)$  and the remainder  $r$  such that  $f = q_1*f_1 + \dots + q_n*f_n + r$ , where  $r$  vanishes or  $r$  is a completely reduced polynomial with respect to  $G$ .



## Examples

```
>>> from sympy import groebner, expand
>>> from sympy.abc import x, y
```

```
>>> f = 2*x**4 - x**2 + y**3 + y**2
>>> G = groebner([x**3 - x, y**3 - y])
```

```
>>> G.reduce(f)
([2*x, 1], x**2 + y**2 + y)
>>> Q, r = _
```

```
>>> expand(sum(q*g for q, g in zip(Q, G)) + r)
2*x**4 - x**2 + y**3 + y**2
>>> _ == f
True
```

## Extra polynomial manipulation functions

`sympy.polys.polyfuncs.symmetrize(F, *gens, **args)`

Rewrite a polynomial in terms of elementary symmetric polynomials.

A symmetric polynomial is a multivariate polynomial that remains invariant under any variable permutation, i.e., if  $f = f(x_1, x_2, \dots, x_n)$ , then  $f = f(x_{i_1}, x_{i_2}, \dots, x_{i_n})$ , where  $(i_1, i_2, \dots, i_n)$  is a permutation of  $(1, 2, \dots, n)$  (an element of the group  $S_n$ ).

Returns a tuple of symmetric polynomials  $(f_1, f_2, \dots, f_n)$  such that  $f = f_1 + f_2 + \dots + f_n$ .

## Examples

```
>>> from sympy.polys.polyfuncs import symmetrize
>>> from sympy.abc import x, y
```

```
>>> symmetrize(x**2 + y**2)
(-2*x*y + (x + y)**2, 0)
```

```
>>> symmetrize(x**2 + y**2, formal=True)
(s1**2 - 2*s2, 0, [(s1, x + y), (s2, x*y)])
```

```
>>> symmetrize(x**2 - y**2)
(-2*x*y + (x + y)**2, -2*y**2)
```

```
>>> symmetrize(x**2 - y**2, formal=True)
(s1**2 - 2*s2, -2*y**2, [(s1, x + y), (s2, x*y)])
```

`sympy.polys.polyfuncs.horner(f, *gens, **args)`

Rewrite a polynomial in Horner form.

Among other applications, evaluation of a polynomial at a point is optimal when it is applied using the Horner scheme ([1]).

### Examples

```
>>> from sympy.polys.polyfuncs import horner
>>> from sympy.abc import x, y, a, b, c, d, e
```

```
>>> horner(9*x**4 + 8*x**3 + 7*x**2 + 6*x + 5)
x*(x*(x*(9*x + 8) + 7) + 6) + 5
```

```
>>> horner(a*x**4 + b*x**3 + c*x**2 + d*x + e)
e + x*(d + x*(c + x*(a*x + b)))
```

```
>>> f = 4*x**2*y**2 + 2*x**2*y + 2*x*y**2 + x*y
```

```
>>> horner(f, wrt=x)
x*(x*y*(4*y + 2) + y*(2*y + 1))
```

```
>>> horner(f, wrt=y)
y*(x*y*(4*x + 2) + x*(2*x + 1))
```

### References

[1] - [https://en.wikipedia.org/wiki/Horner\\_scheme](https://en.wikipedia.org/wiki/Horner_scheme)

`sympy.polys.polyfuncs.interpolate(data, x)`

Construct an interpolating polynomial for the data points evaluated at point `x` (which can be symbolic or numeric).

### Examples

```
>>> from sympy.polys.polyfuncs import interpolate
>>> from sympy.abc import a, b, x
```

A list is interpreted as though it were paired with a range starting from 1:

```
>>> interpolate([1, 4, 9, 16], x)
x**2
```

This can be made explicit by giving a list of coordinates:

```
>>> interpolate([(1, 1), (2, 4), (3, 9)], x)
x**2
```

The `(x, y)` coordinates can also be given as keys and values of a dictionary (and the points need not be equispaced):

```
>>> interpolate([(-1, 2), (1, 2), (2, 5)], x)
x**2 + 1
>>> interpolate({-1: 2, 1: 2, 2: 5}, x)
x**2 + 1
```

If the interpolation is going to be used only once then the value of interest can be passed instead of passing a symbol:

```
>>> interpolate([1, 4, 9], 5)
25
```

Symbolic coordinates are also supported:

```
>>> [(i, interpolate((a, b), i)) for i in range(1, 4)]
[(1, a), (2, b), (3, -a + 2*b)]
```

`sympy.polys.polyfuncs.viete(f, roots=None, *gens, **args)`

Generate Viète's formulas for  $f$ .

### Examples

```
>>> from sympy.polys.polyfuncs import viete
>>> from sympy import symbols
```

```
>>> x, a, b, c, r1, r2 = symbols('x,a:c,r1:3')
```

```
>>> viete(a*x**2 + b*x + c, [r1, r2], x)
[(r1 + r2, -b/a), (r1*r2, c/a)]
```

## Domain constructors

`sympy.polys.constructor.construct_domain(obj, **args)`

Construct a minimal domain for a list of expressions.

### Parameters

**obj: list or dict**

The expressions to build a domain for.

**\*\*args: keyword arguments**

Options that affect the choice of domain.

### Returns

(K, elements): Domain and list of domain elements

The domain  $K$  that can represent the expressions and the list or dict of domain elements representing the same expressions as elements of  $K$ .

## Explanation

Given a list of normal SymPy expressions (of type *Expr* (page 947)) `construct_domain` will find a minimal *Domain* (page 2504) that can represent those expressions. The expressions will be converted to elements of the domain and both the domain and the domain elements are returned.

## Examples

Given a list of *Integer* (page 987) `construct_domain` will return the domain *ZZ* (page 2525) and a list of integers as elements of *ZZ* (page 2525).

```
>>> from sympy import construct_domain, S
>>> expressions = [S(2), S(3), S(4)]
>>> K, elements = construct_domain(expressions)
>>> K
ZZ
>>> elements
[2, 3, 4]
>>> type(elements[0])
<class 'int'>
>>> type(expressions[0])
<class 'sympy.core.numbers.Integer'>
```

If there are any *Rational* (page 985) then *QQ* (page 2529) is returned instead.

```
>>> construct_domain([S(1)/2, S(3)/4])
(QQ, [1/2, 3/4])
```

If there are symbols then a polynomial ring *K[x]* (page 2547) is returned.

```
>>> from sympy import symbols
>>> x, y = symbols('x, y')
>>> construct_domain([2*x + 1, S(3)/4])
(QQ[x], [2*x + 1, 3/4])
>>> construct_domain([2*x + 1, y])
(ZZ[x,y], [2*x + 1, y])
```

If any symbols appear with negative powers then a rational function field *K(x)* (page 2548) will be returned.

```
>>> construct_domain([y/x, x/(1 - y)])
(ZZ(x,y), [y/x, -x/(y - 1)])
```

Irrational algebraic numbers will result in the *EX* (page 2549) domain by default. The keyword argument `extension=True` leads to the construction of an algebraic number field *QQ<a>* (page 2539).

```
>>> from sympy import sqrt
>>> construct_domain([sqrt(2)])
(EX, [EX(sqrt(2))])
>>> construct_domain([sqrt(2)], extension=True)
(QQ<sqrt(2)>, [ANP([1, 0], [1, 0, -2], QQ)])
```

**See also:**

[Domain](#) (page 2504), [Expr](#) (page 947)

## Monomials encoded as tuples

**class** `sympy.polys.monomials.Monomial(monom, gens=None)`

Class representing a monomial, i.e. a product of powers.

**as\_expr**(*\*gens*)

Convert a monomial instance to a SymPy expression.

**gcd**(*other*)

Greatest common divisor of monomials.

**lcm**(*other*)

Least common multiple of monomials.

**sympy.polys.monomials.itermonomials**(*variables*, *max\_degrees*, *min\_degrees*=None)

*max\_degrees* and *min\_degrees* are either both integers or both lists. Unless otherwise specified, *min\_degrees* is either 0 or [0, ..., 0].

A generator of all monomials *monom* is returned, such that either  $\text{min\_degree} \leq \text{total\_degree}(\text{monom}) \leq \text{max\_degree}$ , or  $\text{min\_degrees}[i] \leq \text{degree\_list}(\text{monom})[i] \leq \text{max\_degrees}[i]$ , for all *i*.

### Case I. *max\_degrees* And *min\_degrees* Are Both Integers

Given a set of variables *V* and a *min\_degree* *N* and a *max\_degree* *M* generate a set of monomials of degree less than or equal to *N* and greater than or equal to *M*. The total number of monomials in commutative variables is huge and is given by the following formula if *M* = 0:

$$\frac{(\#V + N)!}{\#V!N!}$$

For example if we would like to generate a dense polynomial of a total degree *N* = 50 and *M* = 0, which is the worst case, in 5 variables, assuming that exponents and all of coefficients are 32-bit long and stored in an array we would need almost 80 GiB of memory! Fortunately most polynomials, that we will encounter, are sparse.

Consider monomials in commutative variables *x* and *y* and non-commutative variables *a* and *b*:

```
>>> from sympy import symbols
>>> from sympy.polys.monomials import itermonomials
>>> from sympy.polys.orderings import monomial_key
>>> from sympy.abc import x, y

>>> sorted(itermonomials([x, y], 2), key=monomial_key('grlex', [y, x]))
[1, x, y, x**2, x*y, y**2]
```

(continues on next page)

(continued from previous page)

```
>>> sorted(itermonomials([x, y], 3), key=monomial_key('grlex', [y, x]))
[1, x, y, x**2, x*y, y**2, x**3, x**2*y, x*y**2, y**3]

>>> a, b = symbols('a, b', commutative=False)
>>> set(itermonomials([a, b, x], 2))
{1, a, a**2, b, b**2, x, x**2, a*b, b*a, x*a, x*b}

>>> sorted(itermonomials([x, y], 2, 1), key=monomial_key('grlex', [y,
↪x]))
[x, y, x**2, x*y, y**2]
```

### Case II. max\_degrees And min\_degrees Are Both Lists

If `max_degrees = [d1, ..., dn]` and `min_degrees = [e1, ..., en]`, the number of monomials generated is:

$$(d_1 - e_1 + 1)(d_2 - e_2 + 1) \cdots (d_n - e_n + 1)$$

Let us generate all monomials `monom` in variables `x` and `y` such that `[1, 2][i] <= degree_list(monom)[i] <= [2, 4][i], i = 0, 1`

```
>>> from sympy import symbols
>>> from sympy.polys.monomials import itermonomials
>>> from sympy.polys.orderings import monomial_key
>>> from sympy.abc import x, y

>>> sorted(itermonomials([x, y], [2, 4], [1, 2]), reverse=True,
↪key=monomial_key('lex', [x, y]))
[x**2*y**4, x**2*y**3, x**2*y**2, x*y**4, x*y**3, x*y**2]
```

`sympy.polys.monomials.monomial_count(V, N)`

Computes the number of monomials.

The number of monomials is given by the following formula:

$$\frac{(\#V + N)!}{\#V!N!}$$

where `N` is a total degree and `V` is a set of variables.

### Examples

```
>>> from sympy.polys.monomials import itermonomials, monomial_count
>>> from sympy.polys.orderings import monomial_key
>>> from sympy.abc import x, y
```

```
>>> monomial_count(2, 2)
6
```

```
>>> M = list(itermonomials([x, y], 2))

>>> sorted(M, key=monomial_key('grlex', [y, x]))
[1, x, y, x**2, x*y, y**2]
>>> len(M)
6
```

## Orderings of monomials

**class** sympy.polys.orderings.**MonomialOrder**

Base class for monomial orderings.

**class** sympy.polys.orderings.**LexOrder**

Lexicographic order of monomials.

**class** sympy.polys.orderings.**GradedLexOrder**

Graded lexicographic order of monomials.

**class** sympy.polys.orderings.**ReversedGradedLexOrder**

Reversed graded lexicographic order of monomials.

## Formal manipulation of roots of polynomials

sympy.polys.rootoftools.**rootof**(*f*, *x*, *index=None*, *radicals=True*, *expand=True*)

An indexed root of a univariate polynomial.

Returns either a [ComplexRootOf](#) (page 2431) object or an explicit expression involving radicals.

### Parameters

**f** : Expr

Univariate polynomial.

**x** : Symbol, optional

Generator for *f*.

**index** : int or Integer

**radicals** : bool

Return a radical expression if possible.

**expand** : bool

Expand *f*.

**class** sympy.polys.rootoftools.**RootOf**(*f*, *x*, *index=None*, *radicals=True*, *expand=True*)

Represents a root of a univariate polynomial.

Base class for roots of different kinds of polynomials. Only complex roots are currently supported.

```
class sympy.polys.rootoftools.ComplexRootOf(f, x, index=None, radicals=False,
 expand=True)
```

Represents an indexed complex root of a polynomial.

Roots of a univariate polynomial separated into disjoint real or complex intervals and indexed in a fixed order:

- real roots come first and are sorted in increasing order;
- complex roots come next and are sorted primarily by increasing real part, secondarily by increasing imaginary part.

Currently only rational coefficients are allowed. Can be imported as `CRootOf`. To avoid confusion, the generator must be a `Symbol`.

### Examples

```
>>> from sympy import CRootOf, rootof
>>> from sympy.abc import x
```

`CRootOf` is a way to reference a particular root of a polynomial. If there is a rational root, it will be returned:

```
>>> CRootOf.clear_cache() # for doctest reproducibility
>>> CRootOf(x**2 - 4, 0)
-2
```

Whether roots involving radicals are returned or not depends on whether the `radicals` flag is true (which is set to `True` with `rootof`):

```
>>> CRootOf(x**2 - 3, 0)
CRootOf(x**2 - 3, 0)
>>> CRootOf(x**2 - 3, 0, radicals=True)
-sqrt(3)
>>> rootof(x**2 - 3, 0)
-sqrt(3)
```

The following cannot be expressed in terms of radicals:

```
>>> r = rootof(4*x**5 + 16*x**3 + 12*x**2 + 7, 0); r
CRootOf(4*x**5 + 16*x**3 + 12*x**2 + 7, 0)
```

The root bounds can be seen, however, and they are used by the evaluation methods to get numerical approximations for the root.

```
>>> interval = r._get_interval(); interval
(-1, 0)
>>> r.evalf(2)
-0.98
```

The `evalf` method refines the width of the root bounds until it guarantees that any decimal approximation within those bounds will satisfy the desired precision. It then stores the refined interval so subsequent requests at or below the requested precision will not have to recompute the root bounds and will return very quickly.

Before evaluation above, the interval was



```
>>> interval
(-1, 0)
```

After evaluation it is now

```
>>> r._get_interval()
(-165/169, -206/211)
```

To reset all intervals for a given polynomial, the `_reset()` (page 2435) method can be called from any `CRootOf` instance of the polynomial:

```
>>> r._reset()
>>> r._get_interval()
(-1, 0)
```

The `eval_approx()` (page 2435) method will also find the root to a given precision but the interval is not modified unless the search for the root fails to converge within the root bounds. And the secant method is used to find the root. (The `evalf` method uses bisection and will always update the interval.)

```
>>> r.eval_approx(2)
-0.98
```

The interval needed to be slightly updated to find that root:

```
>>> r._get_interval()
(-1, -1/2)
```

The `evalf_rational` will compute a rational approximation of the root to the desired accuracy or precision.

```
>>> r.eval_rational(n=2)
-69629/71318
```

```
>>> t = CRootOf(x**3 + 10*x + 1, 1)
>>> t.eval_rational(1e-1)
15/256 - 805*I/256
>>> t.eval_rational(1e-1, 1e-4)
3275/65536 - 414645*I/131072
>>> t.eval_rational(1e-4, 1e-4)
6545/131072 - 414645*I/131072
>>> t.eval_rational(n=2)
104755/2097152 - 6634255*I/2097152
```

## Notes

Although a PurePoly can be constructed from a non-symbol generator RootOf instances of non-symbols are disallowed to avoid confusion over what root is being represented.

```
>>> from sympy import exp, PurePoly
>>> PurePoly(x) == PurePoly(exp(x))
True
>>> CRootOf(x - 1, 0)
1
>>> CRootOf(exp(x) - 1, 0) # would correspond to x == 0
Traceback (most recent call last):
...
sympy.polys.polyerrors.PolynomialError: generator must be a Symbol
```

### See also:

[`eval\_approx`](#) (page 2435), [`eval\_rational`](#) (page 2436)

**classmethod** `_all_roots`(*poly*, *use\_cache=True*)

Get real and complex roots of a composite polynomial.

**classmethod** `_complexes_index`(*complexes*, *index*)

Map initial complex root index to an index in a factor where the root belongs.

**classmethod** `_complexes_sorted`(*complexes*)

Make complex isolating intervals disjoint and sort roots.

**classmethod** `_count_roots`(*roots*)

Count the number of real or complex roots with multiplicities.

**method** `_ensure_complexes_init`()

Ensure that our poly has entries in the complexes cache.

**method** `_ensure_reals_init`()

Ensure that our poly has entries in the reals cache.

**method** `_eval_evalf`(*prec*, *\*\*kwargs*)

Evaluate this complex root to the given precision.

**method** `_eval_is_imaginary`()

Return True if the root is imaginary.

**method** `_eval_is_real`()

Return True if the root is real.

**classmethod** `_get_complexes`(*factors*, *use\_cache=True*)

Compute complex root isolating intervals for a list of factors.

**classmethod** `_get_complexes_sqf`(*currentfactor*, *use\_cache=True*)

Get complex root isolating intervals for a square-free factor.

**method** `_get_interval`()

Internal function for retrieving isolation interval from cache.

**classmethod** `_get_reals`(*factors*, *use\_cache=True*)

Compute real root isolating intervals for a list of factors.

**classmethod** `_get_reals_sqf(currentfactor, use_cache=True)`

Get real root isolating intervals for a square-free factor.

**classmethod** `_get_roots(method, poly, radicals)`

Return postprocessed roots of specified kind.

**classmethod** `_indexed_root(poly, index, lazy=False)`

Get a root of a composite polynomial by index.

**classmethod** `_new(poly, index)`

Construct new CRootOf object from raw data.

**classmethod** `_postprocess_root(root, radicals)`

Return the root if it is trivial or a CRootOf object.

**classmethod** `_preprocess_roots(poly)`

Take heroic measures to make poly compatible with CRootOf.

**classmethod** `_real_roots(poly)`

Get real roots of a composite polynomial.

**classmethod** `_reals_index(reals, index)`

Map initial real root index to an index in a factor where the root belongs.

**classmethod** `_reals_sorted(reals)`

Make real isolating intervals disjoint and sort roots.

**classmethod** `_refine_complexes(complexes)`

return complexes such that no bounding rectangles of non-conjugate roots would intersect. In addition, assure that neither  $a$  nor  $b$  is 0 to guarantee that non-real roots are distinct from real roots in terms of the y-bounds.

**method** `_reset()`

Reset all intervals

**classmethod** `_roots_trivial(poly, radicals)`

Compute roots in linear, quadratic and binomial cases.

**method** `_set_interval(interval)`

Internal function for updating isolation interval in cache.

**classmethod** `all_roots(poly, radicals=True)`

Get real and complex roots of a polynomial.

**classmethod** `clear_cache()`

Reset cache for reals and complexes.

The intervals used to approximate a root instance are updated as needed. When a request is made to see the intervals, the most current values are shown. `clear_cache` will reset all CRootOf instances back to their original state.

**See also:**

[`\_reset`](#) (page 2435)

**method** `eval_approx(n)`

Evaluate this complex root to the given precision.

This uses secant method and root bounds are used to both generate an initial guess and to check that the root returned is valid. If ever the method converges outside the root bounds, the bounds will be made smaller and updated.

**eval\_rational**(*dx=None, dy=None, n=15*)

Return a Rational approximation of *self* that has real and imaginary component approximations that are within *dx* and *dy* of the true values, respectively. Alternatively, *n* digits of precision can be specified.

The interval is refined with bisection and is sure to converge. The root bounds are updated when the refinement is complete so recalculation at the same or lesser precision will not have to repeat the refinement and should be much faster.

The following example first obtains Rational approximation to 1e-8 accuracy for all roots of the 4-th order Legendre polynomial. Since the roots are all less than 1, this will ensure the decimal representation of the approximation will be correct (including rounding) to 6 digits:

```
>>> from sympy import legendre_poly, Symbol
>>> x = Symbol("x")
>>> p = legendre_poly(4, x, polys=True)
>>> r = p.real_roots()[-1]
>>> r.eval_rational(10**-8).n(6)
0.861136
```

It is not necessary to a two-step calculation, however: the decimal representation can be computed directly:

```
>>> r.evalf(17)
0.86113631159405258
```

**classmethod** **real\_roots**(*poly, radicals=True*)

Get real roots of a polynomial.

**class** `sympy.polys.rootoftools.RootSum`(*expr, func=None, x=None, auto=True, quadratic=False*)

Represents a sum of all roots of a univariate polynomial.

**classmethod** **new**(*poly, func, auto=True*)

Construct new RootSum instance.

## Symbolic root-finding algorithms

`sympy.polys.polyroots.roots`(*f, \*gens, auto=True, cubics=True, trig=False, quartics=True, quintics=False, multiple=False, filter=None, predicate=None, strict=False, \*\*flags*)

Computes symbolic roots of a univariate polynomial.

Given a univariate polynomial *f* with symbolic coefficients (or a list of the polynomial's coefficients), returns a dictionary with its roots and their multiplicities.

Only roots expressible via radicals will be returned. To get a complete set of roots use RootOf class or numerical methods instead. By default cubic and quartic formulas are used in the algorithm. To disable them because of unreadable output set *cubics=False* or *quartics=False* respectively. If cubic roots are real but are expressed in terms of complex numbers (casus irreducibilis [1]) the *trig* flag can be set to True to have the solutions returned in terms of cosine and inverse cosine functions.

To get roots from a specific domain set the `filter` flag with one of the following specifiers: Z, Q, R, I, C. By default all roots are returned (this is equivalent to setting `filter='C'`).

By default a dictionary is returned giving a compact result in case of multiple roots. However to get a list containing all those roots set the `multiple` flag to `True`; the list will have identical roots appearing next to each other in the result. (For a given `Poly`, the `all_roots` method will give the roots in sorted numerical order.)

If the `strict` flag is `True`, `UnsolvableFactorError` will be raised if the roots found are known to be incomplete (because some roots are not expressible in radicals).

## Examples

```
>>> from sympy import Poly, roots, degree
>>> from sympy.abc import x, y
```

```
>>> roots(x**2 - 1, x)
{-1: 1, 1: 1}
```

```
>>> p = Poly(x**2-1, x)
>>> roots(p)
{-1: 1, 1: 1}
```

```
>>> p = Poly(x**2-y, x, y)
```

```
>>> roots(Poly(p, x))
{-sqrt(y): 1, sqrt(y): 1}
```

```
>>> roots(x**2 - y, x)
{-sqrt(y): 1, sqrt(y): 1}
```

```
>>> roots([1, 0, -1])
{-1: 1, 1: 1}
```

`roots` will only return roots expressible in radicals. If the given polynomial has some or all of its roots inexpressible in radicals, the result of `roots` will be incomplete or empty respectively.

Example where result is incomplete:

```
>>> roots((x-1)*(x**5-x+1), x)
{1: 1}
```

In this case, the polynomial has an unsolvable quintic factor whose roots cannot be expressed by radicals. The polynomial has a rational root (due to the factor  $(x - 1)$ ), which is returned since `roots` always finds all rational roots.

Example where result is empty:

```
>>> roots(x**7-3*x**2+1, x)
{}
```

Here, the polynomial has no roots expressible in radicals, so `roots` returns an empty dictionary.

The result produced by `roots` is complete if and only if the sum of the multiplicity of each root is equal to the degree of the polynomial. If `strict=True`, `UnsolvableFactorError` will be raised if the result is incomplete.

The result can be checked for completeness as follows:

```
>>> f = x**3-2*x**2+1
>>> sum(roots(f, x).values()) == degree(f, x)
True
>>> f = (x-1)*(x**5-x+1)
>>> sum(roots(f, x).values()) == degree(f, x)
False
```

## References

[R720]

## Special polynomials

`sympy.polys.specialpolys.swinnerton_dyer_poly(n, x=None, polys=False)`

Generates *n*-th Swinnerton-Dyer polynomial in *x*.

### Parameters

**n** : int

*n* decides the order of polynomial

**x** : optional

**polys** : bool, optional

`polys=True` returns an expression, otherwise (default) returns an expression.

`sympy.polys.specialpolys.interpolating_poly(n, x, X='x', Y='y')`

Construct Lagrange interpolating polynomial for *n* data points. If a sequence of values are given for *X* and *Y* then the first *n* values will be used.

`sympy.polys.specialpolys.cyclotomic_poly(n, x=None, polys=False)`

Generates cyclotomic polynomial of order *n* in *x*.

### Parameters

**n** : int

*n* decides the order of polynomial

**x** : optional

**polys** : bool, optional

`polys=True` returns an expression, otherwise (default) returns an expression.

`sympy.polys.specialpolys.symmetric_poly(n, *gens, **args)`

Generates symmetric polynomial of order  $n$ .

Returns a Poly object when `polys=True`, otherwise (default) returns an expression.

`sympy.polys.specialpolys.random_poly(x, n, inf, sup, domain=ZZ, polys=False)`

Generates a polynomial of degree  $n$  with coefficients in  $[\text{inf}, \text{sup}]$ .

#### Parameters

**x**

$x$  is the independent term of polynomial

**n** : int

$n$  decides the order of polynomial

**inf**

Lower limit of range in which coefficients lie

**sup**

Upper limit of range in which coefficients lie

**domain** : optional

Decides what ring the coefficients are supposed to belong. Default is set to Integers.

**polys** : bool, optional

`polys=True` returns an expression, otherwise (default) returns an expression.

### Orthogonal polynomials

`sympy.polys.orthopolys.chebyshevt_poly(n, x=None, polys=False)`

Generates Chebyshev polynomial of the first kind of degree  $n$  in  $x$ .

#### Parameters

**n** : int

$n$  decides the degree of polynomial

**x** : optional

**polys** : bool, optional

`polys=True` returns an expression, otherwise (default) returns an expression.

`sympy.polys.orthopolys.chebyshevu_poly(n, x=None, polys=False)`

Generates Chebyshev polynomial of the second kind of degree  $n$  in  $x$ .

#### Parameters

**n** : int

$n$  decides the degree of polynomial

**x** : optional

**polys** : bool, optional

`polys=True` returns an expression, otherwise (default) returns an expression.

`sympy.polys.orthopolys.gegenbauer_poly(n, a, x=None, polys=False)`

Generates Gegenbauer polynomial of degree  $n$  in  $x$ .

**Parameters**

**n** : int

$n$  decides the degree of polynomial

**x** : optional

**a**

Decides minimal domain for the list of coefficients.

**polys** : bool, optional

`polys=True` returns an expression, otherwise (default) returns an expression.

`sympy.polys.orthopolys.hermite_poly(n, x=None, polys=False)`

Generates Hermite polynomial of degree  $n$  in  $x$ .

**Parameters**

**n** : int

$n$  decides the degree of polynomial

**x** : optional

**polys** : bool, optional

`polys=True` returns an expression, otherwise (default) returns an expression.

`sympy.polys.orthopolys.jacobi_poly(n, a, b, x=None, polys=False)`

Generates Jacobi polynomial of degree  $n$  in  $x$ .

**Parameters**

**n** : int

$n$  decides the degree of polynomial

**a**

Lower limit of minimal domain for the list of coefficients.

**b**

Upper limit of minimal domain for the list of coefficients.

**x** : optional

**polys** : bool, optional

`polys=True` returns an expression, otherwise (default) returns an expression.

`sympy.polys.orthopolys.legendre_poly(n, x=None, polys=False)`

Generates Legendre polynomial of degree  $n$  in  $x$ .

**Parameters**

**n** : int



$n$  decides the degree of polynomial

**x** : optional

**polys** : bool, optional

polys=True returns an expression, otherwise (default) returns an expression.

`sympy.polys.orthopolys.laguerre_poly(n, x=None, alpha=None, polys=False)`

Generates Laguerre polynomial of degree  $n$  in  $x$ .

#### Parameters

**n** : int

$n$  decides the degree of polynomial

**x** : optional

**alpha**

Decides minimal domain for the list of coefficients.

**polys** : bool, optional

polys=True returns an expression, otherwise (default) returns an expression.

`sympy.polys.orthopolys.spherical_bessel_fn(n, x=None, polys=False)`

Coefficients for the spherical Bessel functions.

Those are only needed in the `jn()` function.

The coefficients are calculated from:

$$fn(0, z) = 1/z \quad fn(1, z) = 1/z^2 \quad fn(n-1, z) + fn(n+1, z) == (2*n+1)/z * fn(n, z)$$

#### Parameters

**n** : int

$n$  decides the degree of polynomial

**x** : optional

**polys** : bool, optional

polys=True returns an expression, otherwise (default) returns an expression.

#### Examples

```
>>> from sympy.polys.orthopolys import spherical_bessel_fn as fn
>>> from sympy import Symbol
>>> z = Symbol("z")
>>> fn(1, z)
z**(-2)
>>> fn(2, z)
-1/z + 3/z**3
>>> fn(3, z)
-6/z**2 + 15/z**4
>>> fn(4, z)
1/z - 45/z**3 + 105/z**5
```

## Manipulation of rational functions

`sympy.polys.rationaltools.together(expr, deep=False, fraction=True)`

Denest and combine rational expressions using symbolic methods.

This function takes an expression or a container of expressions and puts it (them) together by denesting and combining rational subexpressions. No heroic measures are taken to minimize degree of the resulting numerator and denominator. To obtain completely reduced expression use [cancel\(\)](#) (page 2376). However, [together\(\)](#) (page 2442) can preserve as much as possible of the structure of the input expression in the output (no expansion is performed).

A wide variety of objects can be put together including lists, tuples, sets, relational objects, integrals and others. It is also possible to transform interior of function applications, by setting `deep` flag to `True`.

By definition, [together\(\)](#) (page 2442) is a complement to [apart\(\)](#) (page 2443), so `apart(together(expr))` should return `expr` unchanged. Note however, that [together\(\)](#) (page 2442) uses only symbolic methods, so it might be necessary to use [cancel\(\)](#) (page 2376) to perform algebraic simplification and minimize degree of the numerator and denominator.

### Examples

```
>>> from sympy import together, exp
>>> from sympy.abc import x, y, z
```

```
>>> together(1/x + 1/y)
(x + y)/(x*y)
>>> together(1/x + 1/y + 1/z)
(x*y + x*z + y*z)/(x*y*z)
```

```
>>> together(1/(x*y) + 1/y**2)
(x + y)/(x*y**2)
```

```
>>> together(1/(1 + 1/x) + 1/(1 + 1/y))
(x*(y + 1) + y*(x + 1))/((x + 1)*(y + 1))
```

```
>>> together(exp(1/x + 1/y))
exp(1/y + 1/x)
>>> together(exp(1/x + 1/y), deep=True)
exp((x + y)/(x*y))
```

```
>>> together(1/exp(x) + 1/(x*exp(x)))
(x + 1)*exp(-x)/x
```

```
>>> together(1/exp(2*x) + 1/(x*exp(3*x)))
(x*exp(x) + 1)*exp(-3*x)/x
```

## Partial fraction decomposition

`sympy.polys.partfrac.apart(f, x=None, full=False, **options)`

Compute partial fraction decomposition of a rational function.

Given a rational function `f`, computes the partial fraction decomposition of `f`. Two algorithms are available: One is based on the undertermined coefficients method, the other is Bronstein's full partial fraction decomposition algorithm.

The undetermined coefficients method (selected by `full=False`) uses polynomial factorization (and therefore accepts the same options as `factor`) for the denominator. Per default it works over the rational numbers, therefore decomposition of denominators with non-rational roots (e.g. irrational, complex roots) is not supported by default (see options of `factor`).

Bronstein's algorithm can be selected by using `full=True` and allows a decomposition of denominators with non-rational roots. A human-readable result can be obtained via `doit()` (see examples below).

### Examples

```
>>> from sympy.polys.partfrac import apart
>>> from sympy.abc import x, y
```

By default, using the undetermined coefficients method:

```
>>> apart(y/(x + 2)/(x + 1), x)
-y/(x + 2) + y/(x + 1)
```

The undetermined coefficients method does not provide a result when the denominators roots are not rational:

```
>>> apart(y/(x**2 + x + 1), x)
y/(x**2 + x + 1)
```

You can choose Bronstein's algorithm by setting `full=True`:

```
>>> apart(y/(x**2 + x + 1), x, full=True)
RootSum(_w**2 + _w + 1, Lambda(_a, (-2*_a*y/3 - y/3)/(-_a + x)))
```

Calling `doit()` yields a human-readable result:

```
>>> apart(y/(x**2 + x + 1), x, full=True).doit()
(-y/3 - 2*y*(-1/2 - sqrt(3)*I/2)/3)/(x + 1/2 + sqrt(3)*I/2) + (-y/3 -
2*y*(-1/2 + sqrt(3)*I/2)/3)/(x + 1/2 - sqrt(3)*I/2)
```

### See also:

[`apart\_list`](#) (page 2443), [`assemble\_partfrac\_list`](#) (page 2445)

`sympy.polys.partfrac.apart_list(f, x=None, dummies=None, **options)`

Compute partial fraction decomposition of a rational function and return the result in structured form.

Given a rational function `f` compute the partial fraction decomposition of `f`. Only Bronstein's full partial fraction decomposition algorithm is supported by this method. The

return value is highly structured and perfectly suited for further algorithmic treatment rather than being human-readable. The function returns a tuple holding three elements:

- The first item is the common coefficient, free of the variable  $x$  used for decomposition. (It is an element of the base field  $K$ .)
- The second item is the polynomial part of the decomposition. This can be the zero polynomial. (It is an element of  $K[x]$ .)
- The third part itself is a list of quadruples. Each quadruple has the following elements in this order:
  - The (not necessarily irreducible) polynomial  $D$  whose roots  $w_i$  appear in the linear denominator of a bunch of related fraction terms. (This item can also be a list of explicit roots. However, at the moment `apart_list` never returns a result this way, but the related `assemble_partfrac_list` function accepts this format as input.)
  - The numerator of the fraction, written as a function of the root  $w$
  - The linear denominator of the fraction *excluding its power exponent*, written as a function of the root  $w$ .
  - The power to which the denominator has to be raised.

One can always rebuild a plain expression by using the function `assemble_partfrac_list`.

## Examples

A first example:

```
>>> from sympy.polys.partfrac import apart_list, assemble_partfrac_list
>>> from sympy.abc import x, t
```

```
>>> f = (2*x**3 - 2*x) / (x**2 - 2*x + 1)
>>> pfd = apart_list(f)
>>> pfd
(1,
 Poly(2*x + 4, x, domain='ZZ'),
 [(Poly(_w - 1, _w, domain='ZZ'), Lambda(_a, 4), Lambda(_a, -_a + x), 1)])
```

```
>>> assemble_partfrac_list(pfd)
2*x + 4 + 4/(x - 1)
```

Second example:

```
>>> f = (-2*x - 2*x**2) / (3*x**2 - 6*x)
>>> pfd = apart_list(f)
>>> pfd
(-1,
 Poly(2/3, x, domain='QQ'),
 [(Poly(_w - 2, _w, domain='ZZ'), Lambda(_a, 2), Lambda(_a, -_a + x), 1)])
```

```
>>> assemble_partfrac_list(pfd)
-2/3 - 2/(x - 2)
```

Another example, showing symbolic parameters:

```
>>> pfd = apart_list(t/(x**2 + x + t), x)
>>> pfd
(1,
 Poly(0, x, domain='ZZ[t]'),
 [(Poly(_w**2 + _w + t, _w, domain='ZZ[t]'),
 Lambda(_a, -2*_a*t/(4*t - 1) - t/(4*t - 1)),
 Lambda(_a, -_a + x),
 1)])
```

```
>>> assemble_partfrac_list(pfd)
RootSum(_w**2 + _w + t, Lambda(_a, (-2*_a*t/(4*t - 1) - t/(4*t - 1))/(-_
→ a + x)))
```

This example is taken from Bronstein's original paper:

```
>>> f = 36 / (x**5 - 2*x**4 - 2*x**3 + 4*x**2 + x - 2)
>>> pfd = apart_list(f)
>>> pfd
(1,
 Poly(0, x, domain='ZZ'),
 [(Poly(_w - 2, _w, domain='ZZ'), Lambda(_a, 4), Lambda(_a, -_a + x), 1),
 (Poly(_w**2 - 1, _w, domain='ZZ'), Lambda(_a, -3*_a - 6), Lambda(_a, -_a
→ + x), 2),
 (Poly(_w + 1, _w, domain='ZZ'), Lambda(_a, -4), Lambda(_a, -_a + x), 1)])
```

```
>>> assemble_partfrac_list(pfd)
-4/(x + 1) - 3/(x + 1)**2 - 9/(x - 1)**2 + 4/(x - 2)
```

**See also:**

[apart](#) (page 2443), [assemble\\_partfrac\\_list](#) (page 2445)

## References

[R721]

`sympy.polys.partfrac.assemble_partfrac_list`(*partial\_list*)

Reassemble a full partial fraction decomposition from a structured result obtained by the function `apart_list`.

## Examples

This example is taken from Bronstein's original paper:

```
>>> from sympy.polys.partfrac import apart_list, assemble_partfrac_list
>>> from sympy.abc import x
```

```
>>> f = 36 / (x**5 - 2*x**4 - 2*x**3 + 4*x**2 + x - 2)
>>> pfd = apart_list(f)
```

(continues on next page)

(continued from previous page)

```
>>> pfd
(1,
 Poly(0, x, domain='ZZ'),
 [(Poly(_w - 2, _w, domain='ZZ'), Lambda(_a, 4), Lambda(_a, -_a + x), 1),
 (Poly(_w**2 - 1, _w, domain='ZZ'), Lambda(_a, -3*_a - 6), Lambda(_a, -_a +
 ↪+ x), 2),
 (Poly(_w + 1, _w, domain='ZZ'), Lambda(_a, -4), Lambda(_a, -_a + x), 1)])
```

```
>>> assemble_partfrac_list(pfd)
-4/(x + 1) - 3/(x + 1)**2 - 9/(x - 1)**2 + 4/(x - 2)
```

If we happen to know some roots we can provide them easily inside the structure:

```
>>> pfd = apart_list(2/(x**2-2))
>>> pfd
(1,
 Poly(0, x, domain='ZZ'),
 [(Poly(_w**2 - 2, _w, domain='ZZ'),
 Lambda(_a, _a/2),
 Lambda(_a, -_a + x),
 1)])
```

```
>>> pfda = assemble_partfrac_list(pfd)
>>> pfda
RootSum(_w**2 - 2, Lambda(_a, _a/(-_a + x)))/2
```

```
>>> pfda.doit()
-sqrt(2)/(2*(x + sqrt(2))) + sqrt(2)/(2*(x - sqrt(2)))
```

```
>>> from sympy import Dummy, Poly, Lambda, sqrt
>>> a = Dummy("a")
>>> pfd = (1, Poly(0, x, domain='ZZ'), [[(sqrt(2), -sqrt(2)), Lambda(a, a/
 ↪2), Lambda(a, -a + x), 1)])
```

```
>>> assemble_partfrac_list(pfd)
-sqrt(2)/(2*(x + sqrt(2))) + sqrt(2)/(2*(x - sqrt(2)))
```

**See also:**

[apart](#) (page 2443), [apart\\_list](#) (page 2443)

## Dispersion of Polynomials

`sympy.polys.dispersion.dispersionset(p, q=None, *gens, **args)`

Compute the *dispersion set* of two polynomials.

For two polynomials  $f(x)$  and  $g(x)$  with  $\deg f > 0$  and  $\deg g > 0$  the dispersion set  $J(f, g)$  is defined as:

$$\begin{aligned} J(f, g) &:= \{a \in \mathbb{N}_0 \mid \gcd(f(x), g(x+a)) \neq 1\} \\ &= \{a \in \mathbb{N}_0 \mid \deg \gcd(f(x), g(x+a)) \geq 1\} \end{aligned}$$

For a single polynomial one defines  $J(f) := J(f, f)$ .

## Examples

```
>>> from sympy import poly
>>> from sympy.polys.dispersion import dispersion, dispersionset
>>> from sympy.abc import x
```

Dispersion set and dispersion of a simple polynomial:

```
>>> fp = poly((x - 3)*(x + 3), x)
>>> sorted(dispersionset(fp))
[0, 6]
>>> dispersion(fp)
6
```

Note that the definition of the dispersion is not symmetric:

```
>>> fp = poly(x**4 - 3*x**2 + 1, x)
>>> gp = fp.shift(-3)
>>> sorted(dispersionset(fp, gp))
[2, 3, 4]
>>> dispersion(fp, gp)
4
>>> sorted(dispersionset(gp, fp))
[]
>>> dispersion(gp, fp)
-oo
```

Computing the dispersion also works over field extensions:

```
>>> from sympy import sqrt
>>> fp = poly(x**2 + sqrt(5)*x - 1, x, domain='QQ<sqrt(5)>')
>>> gp = poly(x**2 + (2 + sqrt(5))*x + sqrt(5), x, domain='QQ<sqrt(5)>')
>>> sorted(dispersionset(fp, gp))
[2]
>>> sorted(dispersionset(gp, fp))
[1, 4]
```

We can even perform the computations for polynomials having symbolic coefficients:

```
>>> from sympy.abc import a
>>> fp = poly(4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2 +
→ 2*a)*x, x)
>>> sorted(dispersionset(fp))
[0, 1]
```

**See also:**

[\*dispersion\*](#) (page 2448)

## References

[R722], [R723], [R724], [R725]

`sympy.polys.dispersion.dispersion(p, q=None, *gens, **args)`

Compute the *dispersion* of polynomials.

For two polynomials  $f(x)$  and  $g(x)$  with  $\deg f > 0$  and  $\deg g > 0$  the dispersion  $\text{dis}(f, g)$  is defined as:

$$\begin{aligned}\text{dis}(f, g) &:= \max\{J(f, g) \cup \{0\}\} \\ &= \max\{\{a \in \mathbb{N} \mid \gcd(f(x), g(x+a)) \neq 1\} \cup \{0\}\}\end{aligned}$$

and for a single polynomial  $\text{dis}(f) := \text{dis}(f, f)$ . Note that we make the definition  $\max\{\} := -\infty$ .

## Examples

```
>>> from sympy import poly
>>> from sympy.polys.dispersion import dispersion, dispersionset
>>> from sympy.abc import x
```

Dispersion set and dispersion of a simple polynomial:

```
>>> fp = poly((x - 3)*(x + 3), x)
>>> sorted(dispersionset(fp))
[0, 6]
>>> dispersion(fp)
6
```

Note that the definition of the dispersion is not symmetric:

```
>>> fp = poly(x**4 - 3*x**2 + 1, x)
>>> gp = fp.shift(-3)
>>> sorted(dispersionset(fp, gp))
[2, 3, 4]
>>> dispersion(fp, gp)
4
>>> sorted(dispersionset(gp, fp))
[]
>>> dispersion(gp, fp)
-oo
```

The maximum of an empty set is defined to be  $-\infty$  as seen in this example.

Computing the dispersion also works over field extensions:

```
>>> from sympy import sqrt
>>> fp = poly(x**2 + sqrt(5)*x - 1, x, domain='QQ<sqrt(5)>')
>>> gp = poly(x**2 + (2 + sqrt(5))*x + sqrt(5), x, domain='QQ<sqrt(5)>')
>>> sorted(dispersionset(fp, gp))
[2]
>>> sorted(dispersionset(gp, fp))
[1, 4]
```



We can even perform the computations for polynomials having symbolic coefficients:

```
>>> from sympy.abc import a
>>> fp = poly(4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2 +
↪ 2*a)*x, x)
>>> sorted(dispersionset(fp))
[0, 1]
```

**See also:**

[\*dispersionset\*](#) (page 2446)

## References

[R726], [R727], [R728], [R729]

## AGCA - Algebraic Geometry and Commutative Algebra Module

### Introduction

Algebraic geometry is a mixture of the ideas of two Mediterranean cultures. It is the superposition of the Arab science of the lightening calculation of the solutions of equations over the Greek art of position and shape. This tapestry was originally woven on European soil and is still being refined under the influence of international fashion. Algebraic geometry studies the delicate balance between the geometrically plausible and the algebraically possible. Whenever one side of this mathematical teeter-totter outweighs the other, one immediately loses interest and runs off in search of a more exciting amusement.

—George R. Kempf (1944 - 2002)

Algebraic Geometry refers to the study of geometric problems via algebraic methods (and sometimes vice versa). While this is a rather old topic, algebraic geometry as understood today is very much a 20th century development. Building on ideas of e.g. Riemann and Dedekind, it was realized that there is an intimate connection between properties of the set of solutions of a system of polynomial equations (called an algebraic variety) and the behavior of the set of polynomial functions on that variety (called the coordinate ring).

As in many geometric disciplines, we can distinguish between local and global questions (and methods). Local investigations in algebraic geometry are essentially equivalent to the study of certain rings, their ideals and modules. This latter topic is also called commutative algebra. It is the basic local toolset of algebraic geometers, in much the same way that differential analysis is the local toolset of differential geometers.

A good conceptual introduction to commutative algebra is [Atiyah69]. An introduction more geared towards computations, and the work most of the algorithms in this module are based on, is [Greuel2008].

This module aims to eventually allow expression and solution of both local and global geometric problems, both in the classical case over a field and in the more modern arithmetic cases. So far, however, there is no geometric functionality at all. Currently the module only provides tools for computational commutative algebra over fields.

All code examples assume:

```
>>> from sympy import *
>>> x, y, z = symbols('x,y,z')
>>> init_printing(use_unicode=True, wrap_line=False)
```

## Reference

In this section we document the usage of the AGCA module. For convenience of the reader, some definitions and examples/explanations are interspersed.

### Base Rings

Almost all computations in commutative algebra are relative to a “base ring”. (For example, when asking questions about an ideal, the base ring is the ring the ideal is a subset of.) In principle all polys “domains” can be used as base rings. However, useful functionality is only implemented for polynomial rings over fields, and various localizations and quotients thereof.

As demonstrated in the examples below, the most convenient method to create objects you are interested in is to build them up from the ground field, and then use the various methods to create new objects from old. For example, in order to create the local ring of the nodal cubic  $y^2 = x^3$  at the origin, over  $\mathbb{Q}$ , you do:

```
>>> lr = QQ.old_poly_ring(x, y, order="ilex") / [y**2 - x**3]
>>> lr
Q[x, y, order=ilex]

$$\left\langle \begin{array}{cc} 3 & 2 \\ -x & +y \end{array} \right\rangle$$

```

Note how the python list notation can be used as a short cut to express ideals. You can use the `convert` method to return ordinary sympy objects into objects understood by the AGCA module (although in many cases this will be done automatically - for example the list was automatically turned into an ideal, and in the process the symbols  $x$  and  $y$  were automatically converted into other representations). For example:

```
>>> X, Y = lr.convert(x), lr.convert(y) ; X

$$x + \left\langle \begin{array}{cc} 3 & 2 \\ -x & +y \end{array} \right\rangle$$

>>> x**3 == y**2
False
>>> X**3 == Y**2
True
```

When no localisation is needed, a more mathematical notation can be used. For example, let us create the coordinate ring of three-dimensional affine space  $\mathbb{A}^3$ :

```
>>> ar = QQ.old_poly_ring(x, y, z); ar
Q[x, y, z]
```

For more details, refer to the following class documentation. Note that the base rings, being domains, are the main point of overlap between the AGCA module and the rest of the polys module. All domains are documented in detail in the polys reference, so we show here only an abridged version, with the methods most pertinent to the AGCA module.

**class** sympy.polys.domains.ring.**Ring**

Represents a ring domain.

**free\_module**(rank)

Generate a free module of rank rank over self.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(2)
QQ[x]**2
```

**ideal**(\*gens)

Generate an ideal of self.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).ideal(x**2)
<x**2>
```

**quotient\_ring**(e)

Form a quotient ring of self.

Here e can be an ideal or an iterable.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).quotient_ring(QQ.old_poly_ring(x).ideal(x**2))
QQ[x]/<x**2>
>>> QQ.old_poly_ring(x).quotient_ring([x**2])
QQ[x]/<x**2>
```

The division operator has been overloaded for this:

```
>>> QQ.old_poly_ring(x)/[x**2]
QQ[x]/<x**2>
```

sympy.polys.domains.polynomialring.**PolynomialRing**(domain\_or\_ring, symbols=None, order=None)

A class for representing multivariate polynomial rings.

**class** sympy.polys.domains.quotientring.**QuotientRing**(ring, ideal)

Class representing (commutative) quotient rings.

You should not usually instantiate this by hand, instead use the constructor from the base ring in the construction.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> I = QQ.old_poly_ring(x).ideal(x**3 + 1)
>>> QQ.old_poly_ring(x).quotient_ring(I)
QQ[x]/<x**3 + 1>
```

Shorter versions are possible:

```
>>> QQ.old_poly_ring(x)/I
QQ[x]/<x**3 + 1>
```

```
>>> QQ.old_poly_ring(x)/[x**3 + 1]
QQ[x]/<x**3 + 1>
```

Attributes:

- ring - the base ring
- base\_ideal - the ideal used to form the quotient

## Modules, Ideals and their Elementary Properties

Let  $A$  be a ring. An  $A$ -module is a set  $M$ , together with two binary operations  $+: M \times M \rightarrow M$  and  $\times: R \times M \rightarrow M$  called addition and scalar multiplication. These are required to satisfy certain axioms, which can be found in e.g. [Atiyah69]. In this way modules are a direct generalisation of both vector spaces ( $A$  being a field) and abelian groups ( $A = \mathbb{Z}$ ). A *submodule* of the  $A$ -module  $M$  is a subset  $N \subset M$ , such that the binary operations restrict to  $N$ , and  $N$  becomes an  $A$ -module with these operations.

The ring  $A$  itself has a natural  $A$ -module structure where addition and multiplication in the module coincide with addition and multiplication in the ring. This  $A$ -module is also written as  $A$ . An  $A$ -submodule of  $A$  is called an *ideal* of  $A$ . Ideals come up very naturally in algebraic geometry. More general modules can be seen as a technically convenient “elbow room” beyond talking only about ideals.

If  $M, N$  are  $A$ -modules, then there is a natural (componentwise)  $A$ -module structure on  $M \times N$ . Similarly there are  $A$ -module structures on cartesian products of more components. (For the categorically inclined: the cartesian product of finitely many  $A$ -modules, with this  $A$ -module structure, is the finite biproduct in the category of all  $A$ -modules. With infinitely many components, it is the direct product (but the infinite direct sum has to be constructed differently).) As usual, repeated product of the  $A$ -module  $M$  is denoted  $M, M^2, M^3 \dots$ , or  $M^I$  for arbitrary index sets  $I$ .

An  $A$ -module  $M$  is called *free* if it is isomorphic to the  $A$ -module  $A^I$  for some (not necessarily finite) index set  $I$  (refer to the next section for a definition of isomorphism). The cardinality of  $I$  is called the *rank* of  $M$ ; one may prove this is well-defined. In general, the AGCA module only works with free modules of finite rank, and other closely related modules. The easiest way to create modules is to use member methods of the objects they are made up from. For example, let us create a free module of rank 4 over the coordinate ring of  $\mathbb{A}^2$  we created above, together with a submodule:

```
>>> F = ar.free_module(4) ; F
 4
QQ[x, y, z]

>>> S = F.submodule([1, x, x**2, x**3], [0, 1, 0, y]) ; S
/[2 3 \
 \[1, x, x , x], [0, 1, 0, y]/
```

Note how python lists can be used as a short-cut notation for module elements (vectors). As usual, the `convert` method can be used to convert sympy/python objects into the internal

AGCA representation (see detailed reference below).

Here is the detailed documentation of the classes for modules, free modules, and submodules:

**class** `sympy.polys.agca.modules.Module(ring)`

Abstract base class for modules.

Do not instantiate - use ring explicit constructors instead:

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> QQ.old_poly_ring(x).free_module(2)
QQ[x]**2
```

Attributes:

- `dtype` - type of elements
- `ring` - containing ring

Non-implemented methods:

- `submodule`
- `quotient_module`
- `is_zero`
- `is_submodule`
- `multiply_ideal`

The method `convert` likely needs to be changed in subclasses.

**contains**(*elem*)

Return True if *elem* is an element of this module.

**convert**(*elem*, *M=None*)

Convert *elem* into internal representation of this module.

If *M* is not None, it should be a module containing it.

**identity\_hom**()

Return the identity homomorphism on *self*.

**is\_submodule**(*other*)

Returns True if *other* is a submodule of *self*.

**is\_zero**()

Returns True if *self* is a zero module.

**multiply\_ideal**(*other*)

Multiply *self* by the ideal *other*.

**quotient\_module**(*other*)

Generate a quotient module.

**submodule**(\**gens*)

Generate a submodule.

**subset**(*other*)

Returns True if *other* is a subset of *self*.

### Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.subset([(1, x), (x, 2)])
True
>>> F.subset([(1/x, x), (x, 2)])
False
```

**class** sympy.polys.agca.modules.**FreeModule**(*ring, rank*)

Abstract base class for free modules.

Additional attributes:

- rank - rank of the free module

Non-implemented methods:

- submodule

**basis**()

Return a set of basis elements.

### Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(3).basis()
([1, 0, 0], [0, 1, 0], [0, 0, 1])
```

**convert**(*elem, M=None*)

Convert *elem* into the internal representation.

This method is called implicitly whenever computations involve elements not in the internal representation.

### Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.convert([1, 0])
[1, 0]
```

**dtype**

alias of [FreeModuleElement](#) (page 2456)

**identity\_hom**()

Return the identity homomorphism on self.

## Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(2).identity_hom()
Matrix([
[1, 0], : QQ[x]**2 -> QQ[x]**2
[0, 1]])
```

### **is\_submodule(*other*)**

Returns True if *other* is a submodule of *self*.

## Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> M = F.submodule([2, x])
>>> F.is_submodule(F)
True
>>> F.is_submodule(M)
True
>>> M.is_submodule(F)
False
```

### **is\_zero()**

Returns True if *self* is a zero module.

(If, as this implementation assumes, the coefficient ring is not the zero ring, then this is equivalent to the rank being zero.)

## Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(0).is_zero()
True
>>> QQ.old_poly_ring(x).free_module(1).is_zero()
False
```

### **multiply\_ideal(*other*)**

Multiply *self* by the ideal *other*.

### Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> I = QQ.old_poly_ring(x).ideal(x)
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.multiply_ideal(I)
<[x, 0], [0, x]>
```

**quotient\_module**(*submodule*)

Return a quotient module.

### Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> M = QQ.old_poly_ring(x).free_module(2)
>>> M.quotient_module(M.submodule([1, x], [x, 2]))
QQ[x]**2/<[1, x], [x, 2]>
```

Or more concisely, using the overloaded division operator:

```
>>> QQ.old_poly_ring(x).free_module(2) / [[1, x], [x, 2]]
QQ[x]**2/<[1, x], [x, 2]>
```

**class** sympy.polys.agca.modules.**FreeModuleElement**(*module, data*)

Element of a free module. Data stored as a tuple.

**class** sympy.polys.agca.modules.**SubModule**(*gens, container*)

Base class for submodules.

Attributes:

- container - containing module
- gens - generators (subset of containing module)
- rank - rank of containing module

Non-implemented methods:

- `_contains`
- `_syzygies`
- `_in_terms_of_generators`
- `_intersect`
- `_module_quotient`

Methods that likely need change in subclasses:

- `reduce_element`

**convert**(*elem, M=None*)

Convert *elem* into the internal representation.

Mostly called implicitly.



### Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> M = QQ.old_poly_ring(x).free_module(2).submodule([1, x])
>>> M.convert([2, 2*x])
[2, 2*x]
```

#### **identity\_hom()**

Return the identity homomorphism on self.

### Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(2).submodule([x, x]).identity_
 ↪ hom()
Matrix([
[1, 0], : <[x, x]> -> <[x, x]>
[0, 1]])
```

#### **in\_terms\_of\_generators(e)**

Express element e of self in terms of the generators.

### Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> M = F.submodule([1, 0], [1, 1])
>>> M.in_terms_of_generators([x, x**2])
[-x**2 + x, x**2]
```

#### **inclusion\_hom()**

Return a homomorphism representing the inclusion map of self.

That is, the natural map from self to self.container.

### Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(2).submodule([x, x]).inclusion_
 ↪ hom()
Matrix([
[1, 0], : <[x, x]> -> QQ[x]**2
[0, 1]])
```

#### **intersect(other, \*\*options)**

Returns the intersection of self with submodule other.

### Examples

```
>>> from sympy.abc import x, y
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x, y).free_module(2)
>>> F.submodule([x, x]).intersect(F.submodule([y, y]))
<[x*y, x*y]>
```

Some implementation allow further options to be passed. Currently, to only one implemented is `relations=True`, in which case the function will return a triple `(res, rela, relb)`, where `res` is the intersection module, and `rela` and `relb` are lists of coefficient vectors, expressing the generators of `res` in terms of the generators of `self` (`rela`) and `other` (`relb`).

```
>>> F.submodule([x, x]).intersect(F.submodule([y, y]), relations=True)
(<[x*y, x*y]>, [(y,)], [(x,)])
```

The above result says: the intersection module is generated by the single element  $(-xy, -xy) = -y(x, x) = -x(y, y)$ , where  $(x, x)$  and  $(y, y)$  respectively are the unique generators of the two modules being intersected.

#### `is_full_module()`

Return True if `self` is the entire free module.

### Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.submodule([x, 1]).is_full_module()
False
>>> F.submodule([1, 1], [1, 2]).is_full_module()
True
```

#### `is_submodule(other)`

Returns True if `other` is a submodule of `self`.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> M = F.submodule([2, x])
>>> N = M.submodule([2*x, x**2])
>>> M.is_submodule(M)
True
>>> M.is_submodule(N)
True
>>> N.is_submodule(M)
False
```

#### `is_zero()`

Return True if `self` is a zero module.

## Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.submodule([x, 1]).is_zero()
False
>>> F.submodule([0, 0]).is_zero()
True
```

**module\_quotient**(*other*, *\*\*options*)

Returns the module quotient of *self* by submodule *other*.

That is, if *self* is the module  $M$  and *other* is  $N$ , then return the ideal  $\{f \in R \mid fN \subset M\}$ .

## Examples

```
>>> from sympy import QQ
>>> from sympy.abc import x, y
>>> F = QQ.old_poly_ring(x, y).free_module(2)
>>> S = F.submodule([x*y, x*y])
>>> T = F.submodule([x, x])
>>> S.module_quotient(T)
<y>
```

Some implementations allow further options to be passed. Currently, the only one implemented is *relations=True*, which may only be passed if *other* is principal. In this case the function will return a pair (*res*, *rel*) where *res* is the ideal, and *rel* is a list of coefficient vectors, expressing the generators of the ideal, multiplied by the generator of *other* in terms of generators of *self*.

```
>>> S.module_quotient(T, relations=True)
(<y>, [[1]])
```

This means that the quotient ideal is generated by the single element  $y$ , and that  $y(x, x) = 1(xy, xy)$ ,  $(x, x)$  and  $(xy, xy)$  being the generators of  $T$  and  $S$ , respectively.

**multiply\_ideal**(*I*)

Multiply *self* by the ideal *I*.

## Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> I = QQ.old_poly_ring(x).ideal(x**2)
>>> M = QQ.old_poly_ring(x).free_module(2).submodule([1, 1])
>>> I*M
<[x**2, x**2]>
```

**quotient\_module**(*other*, *\*\*opts*)

Return a quotient module.

This is the same as taking a submodule of a quotient of the containing module.

### Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> S1 = F.submodule([x, 1])
>>> S2 = F.submodule([x**2, x])
>>> S1.quotient_module(S2)
<[x, 1] + <[x**2, x]>>
```

Or more coincisely, using the overloaded division operator:

```
>>> F.submodule([x, 1]) / [(x**2, x)]
<[x, 1] + <[x**2, x]>>
```

**reduce\_element**(*x*)

Reduce the element *x* of our ring modulo the ideal *self*.

Here “reduce” has no specific meaning, it could return a unique normal form, simplify the expression a bit, or just do nothing.

**submodule**(*\*gens*)

Generate a submodule.

### Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> M = QQ.old_poly_ring(x).free_module(2).submodule([x, 1])
>>> M.submodule([x**2, x])
<[x**2, x]>
```

**syzygy\_module**(*\*\*opts*)

Compute the syzygy module of the generators of *self*.

Suppose  $M$  is generated by  $f_1, \dots, f_n$  over the ring  $R$ . Consider the homomorphism  $\phi : R^n \rightarrow M$ , given by sending  $(r_1, \dots, r_n) \rightarrow r_1 f_1 + \dots + r_n f_n$ . The syzygy module is defined to be the kernel of  $\phi$ .

## Examples

The syzygy module is zero iff the generators generate freely a free submodule:

```
>>> from sympy.abc import x, y
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(2).submodule([1, 0], [1, 1]).
↳syzygy_module().is_zero()
True
```

A slightly more interesting example:

```
>>> M = QQ.old_poly_ring(x, y).free_module(2).submodule([x, 2*x], [y,
↳2*y])
>>> S = QQ.old_poly_ring(x, y).free_module(2).submodule([y, -x])
>>> M.syzygy_module() == S
True
```

### **union**(*other*)

Returns the module generated by the union of self and other.

## Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(1)
>>> M = F.submodule([x**2 + x]) # <x(x+1)>
>>> N = F.submodule([x**2 - 1]) # <(x-1)(x+1)>
>>> M.union(N) == F.submodule([x+1])
True
```

Ideals are created very similarly to modules. For example, let's verify that the nodal cubic is indeed singular at the origin:

```
>>> I = lr.ideal(x, y)
>>> I == lr.ideal(x)
False

>>> I == lr.ideal(y)
False
```

We are using here the fact that a curve is non-singular at a point if and only if the maximal ideal of the local ring is principal, and that in this case at least one of  $x$  and  $y$  must be generators.

This is the detailed documentation of the class `ideal`. Please note that most of the methods regarding properties of ideals (primality etc.) are not yet implemented.

**class** sympy.polys.agca.ideals.**Ideal**(*ring*)

Abstract base class for ideals.

Do not instantiate - use explicit constructors in the ring class instead:

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> QQ.old_poly_ring(x).ideal(x+1)
<x + 1>
```

Attributes

- ring - the ring this ideal belongs to

Non-implemented methods:

- `_contains_elem`
- `_contains_ideal`
- `_quotient`
- `_intersect`
- `_union`
- `_product`
- `is_whole_ring`
- `is_zero`
- `is_prime`, `is_maximal`, `is_primary`, `is_radical`
- `is_principal`
- `height`, `depth`
- `radical`

Methods that likely should be overridden in subclasses:

- `reduce_element`

**contains**(*elem*)

Return True if *elem* is an element of this ideal.

## Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).ideal(x+1, x-1).contains(3)
True
>>> QQ.old_poly_ring(x).ideal(x**2, x**3).contains(x)
False
```

**depth**()

Compute the depth of self.

**height**()

Compute the height of self.

**intersect**(*J*)

Compute the intersection of self with ideal *J*.

## Examples

```
>>> from sympy.abc import x, y
>>> from sympy import QQ
>>> R = QQ.old_poly_ring(x, y)
>>> R.ideal(x).intersect(R.ideal(y))
<x*y>
```

**is\_maximal()**

Return True if `self` is a maximal ideal.

**is\_primary()**

Return True if `self` is a primary ideal.

**is\_prime()**

Return True if `self` is a prime ideal.

**is\_principal()**

Return True if `self` is a principal ideal.

**is\_radical()**

Return True if `self` is a radical ideal.

**is\_whole\_ring()**

Return True if `self` is the whole ring.

**is\_zero()**

Return True if `self` is the zero ideal.

**product(J)**

Compute the ideal product of `self` and `J`.

That is, compute the ideal generated by products  $xy$ , for  $x$  an element of `self` and  $y \in J$ .

## Examples

```
>>> from sympy.abc import x, y
>>> from sympy import QQ
>>> QQ.old_poly_ring(x, y).ideal(x).product(QQ.old_poly_ring(x, y).
↪ ideal(y))
<x*y>
```

**quotient(J, \*\*opts)**

Compute the ideal quotient of `self` by `J`.

That is, if `self` is the ideal  $I$ , compute the set  $I : J = \{x \in R \mid xJ \subset I\}$ .

### Examples

```
>>> from sympy.abc import x, y
>>> from sympy import QQ
>>> R = QQ.old_poly_ring(x, y)
>>> R.ideal(x*y).quotient(R.ideal(x))
<y>
```

#### **radical()**

Compute the radical of self.

#### **reduce\_element(x)**

Reduce the element  $x$  of our ring modulo the ideal self.

Here “reduce” has no specific meaning: it could return a unique normal form, simplify the expression a bit, or just do nothing.

#### **saturate(J)**

Compute the ideal saturation of self by  $J$ .

That is, if self is the ideal  $I$ , compute the set  $I : J^\infty = \{x \in R \mid xJ^n \subset I \text{ for some } n\}$ .

#### **subset(other)**

Returns True if other is a subset of self.

Here other may be an ideal.

### Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> I = QQ.old_poly_ring(x).ideal(x+1)
>>> I.subset([x**2 - 1, x**2 + 2*x + 1])
True
>>> I.subset([x**2 + 1, x + 1])
False
>>> I.subset(QQ.old_poly_ring(x).ideal(x**2 - 1))
True
```

#### **union(J)**

Compute the ideal generated by the union of self and  $J$ .

### Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).ideal(x**2 - 1).union(QQ.old_poly_ring(x).
↪ ideal((x+1)**2)) == QQ.old_poly_ring(x).ideal(x+1)
True
```

If  $M$  is an  $A$ -module and  $N$  is an  $A$ -submodule, we can define two elements  $x$  and  $y$  of  $M$  to be equivalent if  $x - y \in N$ . The set of equivalence classes is written  $M/N$ , and has a natural  $A$ -module structure. This is called the quotient module of  $M$  by  $N$ . If  $K$  is a submodule of  $M$



containing  $N$ , then  $K/N$  is in a natural way a submodule of  $M/N$ . Such a module is called a subquotient. Here is the documentation of quotient and subquotient modules:

**class** `sympy.polys.agca.modules.QuotientModule`(*ring, base, submodule*)

Class for quotient modules.

Do not instantiate this directly. For subquotients, see the SubQuotientModule class.

Attributes:

- `base` - the base module we are a quotient of
- `killed_module` - the submodule used to form the quotient
- `rank` of the base

**convert**(*elem, M=None*)

Convert `elem` into the internal representation.

This method is called implicitly whenever computations involve elements not in the internal representation.

### Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2) / [(1, 2), (1, x)]
>>> F.convert([1, 0])
[1, 0] + <[1, 2], [1, x]>
```

**dtype**

alias of [QuotientModuleElement](#) (page 2467)

**identity\_hom()**

Return the identity homomorphism on self.

### Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> M = QQ.old_poly_ring(x).free_module(2) / [(1, 2), (1, x)]
>>> M.identity_hom()
Matrix([
[1, 0], : QQ[x]**2/<[1, 2], [1, x]> -> QQ[x]**2/<[1, 2], [1, x]>
[0, 1]])
```

**is\_submodule**(*other*)

Return True if `other` is a submodule of self.

### Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> Q = QQ.old_poly_ring(x).free_module(2) / [(x, x)]
>>> S = Q.submodule([1, 0])
>>> Q.is_submodule(S)
True
>>> S.is_submodule(Q)
False
```

#### **is\_zero()**

Return True if `self` is a zero module.

This happens if and only if the base module is the same as the submodule being killed.

### Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> (F/[(1, 0)]).is_zero()
False
>>> (F/[(1, 0), (0, 1)]).is_zero()
True
```

#### **quotient\_hom()**

Return the quotient homomorphism to `self`.

That is, return a homomorphism representing the natural map from `self.base` to `self`.

### Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> M = QQ.old_poly_ring(x).free_module(2) / [(1, 2), (1, x)]
>>> M.quotient_hom()
Matrix([
[1, 0], : QQ[x]**2 -> QQ[x]**2/<[1, 2], [1, x]>
[0, 1]])
```

#### **submodule(\*gens, \*\*opts)**

Generate a submodule.

This is the same as taking a quotient of a submodule of the base module.

## Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> Q = QQ.old_poly_ring(x).free_module(2) / [(x, x)]
>>> Q.submodule([x, 0])
<[x, 0] + <[x, x]>>
```

**class** sympy.polys.agca.modules.**QuotientModuleElement**(*module, data*)

Element of a quotient module.

**eq**(*d1, d2*)

Equality comparison.

**class** sympy.polys.agca.modules.**SubQuotientModule**(*gens, container, \*\*opts*)

Submodule of a quotient module.

Equivalently, quotient module of a submodule.

Do not instantiate this, instead use the submodule or quotient\_module constructing methods:

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> S = F.submodule([1, 0], [1, x])
>>> Q = F/[(1, 0)]
>>> S/[(1, 0)] == Q.submodule([5, x])
True
```

Attributes:

- **base** - base module we are quotient of
- **killed\_module** - submodule used to form the quotient

**is\_full\_module**()

Return True if `self` is the entire free module.

## Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.submodule([x, 1]).is_full_module()
False
>>> F.submodule([1, 1], [1, 2]).is_full_module()
True
```

**quotient\_hom**()

Return the quotient homomorphism to self.

That is, return the natural map from `self.base` to self.

## Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> M = (QQ.old_poly_ring(x).free_module(2) / [(1, x)]).submodule([1,
↳ 0])
>>> M.quotient_hom()
Matrix([
[1, 0], : <[1, 0], [1, x]> -> <[1, 0] + <[1, x]>, [1, x] + <[1, x]>>
[0, 1]])
```

## Module Homomorphisms and Syzygies

Let  $M$  and  $N$  be  $A$ -modules. A mapping  $f : M \rightarrow N$  satisfying various obvious properties (see [Atiyah69]) is called an  $A$ -module homomorphism. In this case  $M$  is called the *domain* and  $N$  the *codomain*. The set  $\{x \in M \mid f(x) = 0\}$  is called the *kernel*  $\ker(f)$ , whereas the set  $\{f(x) \mid x \in M\}$  is called the *image*  $\operatorname{im}(f)$ . The kernel is a submodule of  $M$ , the image is a submodule of  $N$ . The homomorphism  $f$  is injective if and only if  $\ker(f) = 0$  and surjective if and only if  $\operatorname{im}(f) = N$ . A bijective homomorphism is called an *isomorphism*. Equivalently,  $\ker(f) = 0$  and  $\operatorname{im}(f) = N$ . (A related notion, which currently has no special name in the AGCA module, is that of the *cokernel*,  $\operatorname{coker}(f) = N/\operatorname{im}(f)$ .)

Suppose now  $M$  is an  $A$ -module.  $M$  is called *finitely generated* if there exists a surjective homomorphism  $A^n \rightarrow M$  for some  $n$ . If such a morphism  $f$  is chosen, the images of the standard basis of  $A^n$  are called the *generators* of  $M$ . The module  $\ker(f)$  is called *syzygy module* with respect to the generators. A module is called *finitely presented* if it is finitely generated with a finitely generated syzygy module. The class of finitely presented modules is essentially the largest class we can hope to be able to meaningfully compute in.

It is an important theorem that, for all the rings we are considering, all submodules of finitely generated modules are finitely generated, and hence finitely generated and finitely presented modules are the same.

The notion of syzygies, while it may first seem rather abstract, is actually very computational. This is because there exist (fairly easy) algorithms for computing them, and more general questions (kernels, intersections, ...) are often reduced to syzygy computation.

Let us say a few words about the definition of homomorphisms in the AGCA module. Suppose first that  $f : M \rightarrow N$  is an arbitrary morphism of  $A$ -modules. Then if  $K$  is a submodule of  $M$ ,  $f$  naturally defines a new homomorphism  $g : K \rightarrow N$  (via  $g(x) = f(x)$ ), called the *restriction* of  $f$  to  $K$ . If now  $K$  contained in the kernel of  $f$ , then moreover  $f$  defines in a natural homomorphism  $g : M/K \rightarrow N$  (same formula as above!), and we say that  $f$  *descends* to  $M/K$ . Similarly, if  $L$  is a submodule of  $N$ , there is a natural homomorphism  $g : M \rightarrow N/L$ , we say that  $g$  *factors* through  $f$ . Finally, if now  $L$  contains the image of  $f$ , then there is a natural homomorphism  $g : M \rightarrow L$  (defined, again, by the same formula), and we say  $g$  is obtained from  $f$  by restriction of codomain. Observe also that each of these four operations is reversible, in the sense that given  $g$ , one can always (non-uniquely) find  $f$  such that  $g$  is obtained from  $f$  in the above way.

Note that all modules implemented in AGCA are obtained from free modules by taking a succession of submodules and quotients. Hence, in order to explain how to define a homomorphism between arbitrary modules, in light of the above, we need only explain how to define homomorphisms of free modules. But, essentially by the definition of free module, a homomorphism from a free module  $A^n$  to any module  $M$  is precisely the same as giving  $n$  elements of  $M$  (the images of the standard basis), and giving an element of a free module

$A^m$  is precisely the same as giving  $m$  elements of  $A$ . Hence a homomorphism of free modules  $A^n \rightarrow A^m$  can be specified via a matrix, entirely analogously to the case of vector spaces.

The functions `restrict_domain` etc. of the class `Homomorphism` can be used to carry out the operations described above, and homomorphisms of free modules can in principle be instantiated by hand. Since these operations are so common, there is a convenience function `homomorphism` to define a homomorphism between arbitrary modules via the method outlined above. It is essentially the only way homomorphisms need ever be created by the user.

`sympy.polys.agca.homomorphisms.homomorphism(domain, codomain, matrix)`

Create a homomorphism object.

This function tries to build a homomorphism from `domain` to `codomain` via the matrix `matrix`.

## Examples

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> R = QQ.old_poly_ring(x)
>>> T = R.free_module(2)
```

If `domain` is a free module generated by  $e_1, \dots, e_n$ , then `matrix` should be an  $n$ -element iterable  $(b_1, \dots, b_n)$  where the  $b_i$  are elements of `codomain`. The constructed homomorphism is the unique homomorphism sending  $e_i$  to  $b_i$ .

```
>>> F = R.free_module(2)
>>> h = homomorphism(F, T, [[1, x], [x**2, 0]])
>>> h
Matrix([
[1, x**2], : QQ[x]**2 -> QQ[x]**2
[x, 0]])
>>> h([1, 0])
[1, x]
>>> h([0, 1])
[x**2, 0]
>>> h([1, 1])
[x**2 + 1, x]
```

If `domain` is a submodule of a free module, then `matrix` determines a homomorphism from the containing free module to `codomain`, and the homomorphism returned is obtained by restriction to `domain`.

```
>>> S = F.submodule([1, 0], [0, x])
>>> homomorphism(S, T, [[1, x], [x**2, 0]])
Matrix([
[1, x**2], : <[1, 0], [0, x]> -> QQ[x]**2
[x, 0]])
```

If `domain` is a (sub)quotient  $N/K$ , then `matrix` determines a homomorphism from  $N$  to `codomain`. If the kernel contains  $K$ , this homomorphism descends to `domain` and is returned; otherwise an exception is raised.

```
>>> homomorphism(S/[(1, 0)], T, [0, [x**2, 0]])
Matrix([
[0, x**2], : <[1, 0] + <[1, 0]>, [0, x] + <[1, 0]>, [1, 0] + <[1, 0]>> ->
 ↳ QQ[x]**2
[0, 0]])
>>> homomorphism(S/[(0, x)], T, [0, [x**2, 0]])
Traceback (most recent call last):
...
ValueError: kernel <[1, 0], [0, 0]> must contain sm, got <[0,x]>
```

Finally, here is the detailed reference of the actual homomorphism class:

**class** sympy.polys.agca.homomorphisms.**ModuleHomomorphism**(*domain, codomain*)

Abstract base class for module homomorphisms. Do not instantiate.

Instead, use the homomorphism function:

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> homomorphism(F, F, [[1, 0], [0, 1]])
Matrix([
[1, 0], : QQ[x]**2 -> QQ[x]**2
[0, 1]])
```

Attributes:

- ring - the ring over which we are considering modules
- domain - the domain module
- codomain - the codomain module
- `_ker` - cached kernel
- `_img` - cached image

Non-implemented methods:

- `_kernel`
- `_image`
- `_restrict_domain`
- `_restrict_codomain`
- `_quotient_domain`
- `_quotient_codomain`
- `_apply`
- `_mul_scalar`
- `_compose`
- `_add`

**image()**

Compute the image of `self`.

That is, if `self` is the homomorphism  $\phi : M \rightarrow N$ , then compute  $im(\phi) = \{\phi(x) | x \in M\}$ . This is a submodule of  $N$ .

**Examples**

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> homomorphism(F, F, [[1, 0], [x, 0]]).image() == F.submodule([1,
↪ 0])
True
```

**is\_injective()**

Return True if `self` is injective.

That is, check if the elements of the domain are mapped to the same codomain element.

**Examples**

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h.is_injective()
False
>>> h.quotient_domain(h.kernel()).is_injective()
True
```

**is\_isomorphism()**

Return True if `self` is an isomorphism.

That is, check if every element of the codomain has precisely one preimage. Equivalently, `self` is both injective and surjective.

### Examples

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h = h.restrict_codomain(h.image())
>>> h.is_isomorphism()
False
>>> h.quotient_domain(h.kernel()).is_isomorphism()
True
```

#### `is_surjective()`

Return True if self is surjective.

That is, check if every element of the codomain has at least one preimage.

### Examples

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h.is_surjective()
False
>>> h.restrict_codomain(h.image()).is_surjective()
True
```

#### `is_zero()`

Return True if self is a zero morphism.

That is, check if every element of the domain is mapped to zero under self.

### Examples

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h.is_zero()
False
>>> h.restrict_domain(F.submodule()).is_zero()
True
```

(continues on next page)



(continued from previous page)

```
>>> h.quotient_codomain(h.image()).is_zero()
True
```

**kernel()**

Compute the kernel of self.

That is, if self is the homomorphism  $\phi : M \rightarrow N$ , then compute  $\ker(\phi) = \{x \in M | \phi(x) = 0\}$ . This is a submodule of  $M$ .

**Examples**

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> homomorphism(F, F, [[1, 0], [x, 0]]).kernel()
<[x, -1]>
```

**quotient\_codomain(sm)**

Return self with codomain replaced by codomain/sm.

Here sm must be a submodule of self.codomain.

**Examples**

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h
Matrix([
[1, x], : QQ[x]**2 -> QQ[x]**2
[0, 0]])
>>> h.quotient_codomain(F.submodule([1, 1]))
Matrix([
[1, x], : QQ[x]**2 -> QQ[x]**2/<[1, 1]>
[0, 0]])
```

This is the same as composing with the quotient map on the left:

```
>>> (F/([1, 1])).quotient_hom() * h
Matrix([
[1, x], : QQ[x]**2 -> QQ[x]**2/<[1, 1]>
[0, 0]])
```

**quotient\_domain(sm)**

Return self with domain replaced by domain/sm.

Here sm must be a submodule of self.kernel().

**Examples**

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h
Matrix([
[1, x], : QQ[x]**2 -> QQ[x]**2
[0, 0]])
>>> h.quotient_domain(F.submodule([-x, 1]))
Matrix([
[1, x], : QQ[x]**2/<[-x, 1]> -> QQ[x]**2
[0, 0]])
```

**restrict\_codomain(sm)**

Return self, with codomain restricted to to sm.

Here sm has to be a submodule of self.codomain containing the image.

**Examples**

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h
Matrix([
[1, x], : QQ[x]**2 -> QQ[x]**2
[0, 0]])
>>> h.restrict_codomain(F.submodule([1, 0]))
Matrix([
[1, x], : QQ[x]**2 -> <[1, 0]>
[0, 0]])
```

**restrict\_domain(sm)**

Return self, with the domain restricted to sm.

Here sm has to be a submodule of self.domain.

## Examples

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h
Matrix([
[1, x], : QQ[x]**2 -> QQ[x]**2
[0, 0]])
>>> h.restrict_domain(F.submodule([1, 0]))
Matrix([
[1, x], : <[1, 0]> -> QQ[x]**2
[0, 0]])
```

This is the same as just composing on the right with the submodule inclusion:

```
>>> h * F.submodule([1, 0]).inclusion_hom()
Matrix([
[1, x], : <[1, 0]> -> QQ[x]**2
[0, 0]])
```

## Finite Extensions

Let  $A$  be a (commutative) ring and  $B$  an extension ring of  $A$ . An element  $t$  of  $B$  is a generator of  $B$  (over  $A$ ) if all elements of  $B$  can be represented as polynomials in  $t$  with coefficients in  $A$ . The representation is unique if and only if  $t$  satisfies no non-trivial polynomial relation, in which case  $B$  can be identified with a (univariate) polynomial ring over  $A$ .

The polynomials having  $t$  as a root form a non-zero ideal in general. The most important case in practice is that of an ideal generated by a single monic polynomial. If  $t$  satisfies such a polynomial relation, then its highest power  $t^n$  can be written as linear combination of lower powers. It follows, inductively, that all higher powers of  $t$  also have such a representation. Hence the lower powers  $t^i$  ( $i = 0, \dots, n-1$ ) form a basis of  $B$ , which is then called a finite extension of  $A$ , or, more precisely, a monogenic finite extension as it is generated by a single element  $t$ .

**class** sympy.polys.agca.extensions.**MonogenicFiniteExtension**(*mod*)

Finite extension generated by an integral element.

The generator is defined by a monic univariate polynomial derived from the argument *mod*.

A shorter alias is `FiniteExtension`.

## Examples

Quadratic integer ring  $\mathbb{Z}[\sqrt{2}]$ :

```
>>> from sympy import Symbol, Poly
>>> from sympy.polys.agca.extensions import FiniteExtension
>>> x = Symbol('x')
>>> R = FiniteExtension(Poly(x**2 - 2)); R
ZZ[x]/(x**2 - 2)
>>> R.rank
2
>>> R(1 + x)*(3 - 2*x)
x - 1
```

Finite field  $GF(5^3)$  defined by the primitive polynomial  $x^3 + x^2 + 2$  (over  $\mathbb{Z}_5$ ).

```
>>> F = FiniteExtension(Poly(x**3 + x**2 + 2, modulus=5)); F
GF(5)[x]/(x**3 + x**2 + 2)
>>> F.basis
(1, x, x**2)
>>> F(x + 3)/(x**2 + 2)
-2*x**2 + x + 2
```

Function field of an elliptic curve:

```
>>> t = Symbol('t')
>>> FiniteExtension(Poly(t**2 - x**3 - x + 1, t, field=True))
ZZ(x)[t]/(t**2 - x**3 - x + 1)
```

### dtype

alias of [ExtensionElement](#) (page 2476)

**class** sympy.polys.agca.extensions.**ExtensionElement**(rep, ext)

Element of a finite extension.

A class of univariate polynomials modulo the modulus of the extension ext. It is represented by the unique polynomial rep of lowest degree. Both rep and the representation mod of modulus are of class DMP.

### inverse()

Multiplicative inverse.

### Raises

#### **NotInvertible**

If the element is a zero divisor.

## Introducing the Domains of the poly module

This page introduces the idea of the “domains” that are used in SymPy’s `sympy.polys` (page 2360) module. The emphasis is on introducing how to use the domains directly and on understanding how they are used internally as part of the `Poly` (page 2378) class. This is a relatively advanced topic so for a more introductory understanding of the `Poly` (page 2378) class and the `sympy.polys` (page 2360) module it is recommended to read *Basic functionality of the module* (page 2342) instead. The reference documentation for the domain classes is in *Reference docs for the Poly Domains* (page 2504). Internal functions that make use of the domains are documented in *Internals of the Polynomial Manipulation Module* (page 2569).

## What are the domains?

For most users the domains are only really noticeable in the printed output of a `Poly` (page 2378):

```
>>> from sympy import Symbol, Poly
>>> x = Symbol('x')
>>> Poly(x**2 + x)
Poly(x**2 + x, x, domain='ZZ')
>>> Poly(x**2 + x/2)
Poly(x**2 + 1/2*x, x, domain='QQ')
```

We see here that one `Poly` (page 2378) has domain `ZZ` (page 2525) representing the integers and the other has domain `QQ` (page 2529) representing the rationals. These indicate the “domain” from which the coefficients of the polynomial are drawn.

From a high-level the domains represent formal concepts such as the set of integers  $\mathbb{Z}$  or rationals  $\mathbb{Q}$ . The word “domain” here is a reference to the mathematical concept of an *integral domain*.

Internally the domains correspond to different computational implementations and representations of the expressions that the polynomials correspond to. The `Poly` (page 2378) object itself has an internal representation as a list of coefficients and a `domain` (page 2391) attribute representing the implementation of those coefficients:

```
>>> p = Poly(x**2 + x/2)
>>> p
Poly(x**2 + 1/2*x, x, domain='QQ')
>>> p.domain
QQ
>>> p.rep
DMP([1, 1/2, 0], QQ, None)
>>> p.rep.rep
[1, 1/2, 0]
>>> type(p.rep.rep[0])
<class 'sympy.external.pythonmpq.PythonMPQ'>
```

Here the domain is `QQ` (page 2529) which represents the implementation of the rational numbers in the domain system. The `Poly` (page 2378) instance itself has a `Poly.domain` (page 2391) attribute `QQ` (page 2529) and then a list of `PythonMPQ` (page 2532) coefficients where `PythonMPQ` (page 2532) is the class that implements the elements of the `QQ` (page 2529) domain. The list of coefficients `[1, 1/2, 0]` gives a standardised low-level representation of the polynomial expression  $(1)x^2 + (1/2)x + (0)$ .

This page looks at the different domains that are defined in SymPy, how they are implemented and how they can be used. It introduces how to use the domains and domain elements directly and explains how they are used internally as part of *Poly* (page 2378) objects. This information is more relevant for development in SymPy than it is for users of the *sympy.polys* (page 2360) module.

## Representing expressions symbolically

There are many different ways that a mathematical expression can be represented symbolically. The purpose of the polynomial domains is to provide suitable implementations for different classes of expressions. This section considers the basic approaches to the symbolic representation of mathematical expressions: “tree”, “dense polynomial” and “sparse polynomial”.

### Tree representation

The most general representation of symbolic expressions is as a *tree* and this is the representation used for most ordinary SymPy expressions which are instances of *Expr* (page 947) (a subclass of *Basic* (page 927)). We can see this representation using the *srepr()* (page 2178) function:

```
>>> from sympy import Symbol, srepr
>>> x = Symbol('x')
>>> e = 1 + 1/(2 + x**2)
>>> e
1 + 1/(x**2 + 2)
>>> print(srepr(e))
Add(Integer(1), Pow(Add(Pow(Symbol('x'), Integer(2)), Integer(2)), Integer(-
↪1)))
```

Here the expression *e* is represented as an *Add* (page 1013) node which has two children 1 and  $1/(x^2 + 2)$ . The child 1 is represented as an *Integer* (page 987) and the other child is represented as a *Pow* (page 1005) with base  $x^2 + 2$  and exponent 1. Then  $x^2 + 2$  is represented as an *Add* (page 1013) with children  $x^2$  and 2 and so on. In this way the expression is represented as a tree where the internal nodes are operations like *Add* (page 1013), *Mul* (page 1009), *Pow* (page 1005) and so on and the leaf nodes are atomic expression types like *Integer* (page 987) and *Symbol* (page 976). See *Advanced Expression Manipulation* (page 61) for more about this representation.

The tree representation is core to the architecture of *Expr* (page 947) in SymPy. It is a highly flexible representation that can represent a very wide range of possible expressions. It can also represent equivalent expressions in different ways e.g.:

```
>>> e = x*(x + 1)
>>> e
x*(x + 1)
>>> e.expand()
x**2 + x
```

These two expression although equivalent have different tree representations:

```
>>> print(srepr(e))
Mul(Symbol('x'), Add(Symbol('x'), Integer(1)))
>>> print(srepr(e.expand()))
Add(Pow(Symbol('x'), Integer(2)), Symbol('x'))
```

Being able to represent the same expression in different ways is both a strength and a weakness. It is useful to be able to convert an expression in to different forms for different tasks but having non-unique representations makes it hard to tell when two expressions are equivalent which is in fact very important for many computational algorithms. The most important task is being able to tell when an expression is equal to zero which is undecidable in general ([Richardson's theorem](#)) but is decidable in many important special cases.

## DUP representation

Restricting the set of allowed expressions to special cases allows for much more efficient symbolic representations. As we already saw [Poly](#) (page 2378) can represent a polynomial as a list of coefficients. This means that an expression like  $x^4 + x + 1$  could be represented simply as `[1, 0, 0, 1, 1]`. This list of coefficients representation of a polynomial expression is known as the “dense univariate polynomial” (DUP) representation. Working within that representation algorithms for multiplication, addition and crucially zero-testing can be much more efficient than with the corresponding tree representations. We can see this representation from a [Poly](#) (page 2378) instance by looking at its `rep.rep` attribute:

```
>>> p = Poly(x**4 + x + 1)
>>> p.rep.rep
[1, 0, 0, 1, 1]
```

In the DUP representation it is not possible to represent the same expression in different ways. There is no distinction between  $x(x + 1)$  and  $x^2 + x$  because both are just `[1, 1, 0]`. This means that comparing two expressions is easy: they are equal if and only if all of their coefficients are equal. Zero-testing is particularly easy: the polynomial is zero if and only if all coefficients are zero (of course we need to have easy zero-testing for the coefficients themselves).

We can make functions that operate on the DUP representation much more efficiently than functions that operate on the tree representation. Many operations with standard sympy expressions are in fact computed by converting to a polynomial representation and then performing the calculation. An example is the [factor\(\)](#) (page 2373) function:

```
>>> from sympy import factor
>>> e = 2*x**3 + 10*x**2 + 16*x + 8
>>> e
2*x**3 + 10*x**2 + 16*x + 8
>>> factor(e)
2*(x + 1)*(x + 2)**2
```

Internally [factor\(\)](#) (page 2373) will convert the expression from the tree representation into the DUP representation and then use the function `dup_factor_list`:

```
>>> from sympy import ZZ
>>> from sympy.polys.factortools import dup_factor_list
>>> p = [ZZ(2), ZZ(10), ZZ(16), ZZ(8)]
```

(continues on next page)

(continued from previous page)

```
>>> p
[2, 10, 16, 8]
>>> dup_factor_list(p, ZZ)
(2, [[1, 1], 1), ([1, 2], 2)])
```

There are many more examples of functions with `dup_*` names for operating on the DUP representation that are documented in *Internals of the Polynomial Manipulation Module* (page 2569). There are also functions with the `dmp_*` prefix for operating on multivariate polynomials.

## DMP representation

A multivariate polynomial (a polynomial in multiple variables) can be represented as a polynomial with coefficients that are themselves polynomials. For example  $x^{**2}y + x^{**2} + x*y + y + 1$  can be represented as polynomial in  $x$  where the coefficients are themselves polynomials in  $y$  i.e.:  $(y + 1)*x^{**2} + (y)*x + (y+1)$ . Since we can represent a polynomial with a list of coefficients a multivariate polynomial can be represented with a list of lists of coefficients:

```
>>> from sympy import symbols
>>> x, y = symbols('x, y')
>>> p = Poly(x**2*y + x**2 + x*y + y + 1)
>>> p
Poly(x**2*y + x**2 + x*y + y + 1, x, y, domain='ZZ')
>>> p.rep.rep
[[1, 1], [1, 0], [1, 1]]
```

This list of lists of (lists of...) coefficients representation is known as the “dense multivariate polynomial” (DMP) representation.

## Sparse polynomial representation

Instead of lists we can use a dict mapping nonzero monomial terms to their coefficients. This is known as the “sparse polynomial” representation. We can see what this would look like using the `as_dict()` (page 2382) method:

```
>>> Poly(7*x**20 + 8*x + 9).rep.rep
[7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 8, 9]
>>> Poly(7*x**20 + 8*x + 9).as_dict()
{(0,): 9, (1,): 8, (20,): 7}
```

The keys of this dict are the exponents of the powers of  $x$  and the values are the coefficients so e.g.  $7*x^{**20}$  becomes  $(20,): 7$  in the dict. The key is a tuple so that in the multivariate case something like  $4*x^{**2}*y^{**3}$  can be represented as  $(2, 3): 4$ . The sparse representation can be more efficient as it avoids the need to store and manipulate the zero coefficients. With a large number of generators (variables) the dense representation becomes particularly inefficient and it is better to use the sparse representation:

```
>>> from sympy import prod
>>> gens = symbols('x:10')
>>> gens
```

(continues on next page)





(continued from previous page)

```
>>> ZZ.is_Field
False
```

Most domains representing non-field rings allow floor and modulo division (remainder) with Python's floor division `//` and modulo division `%` operators. For example with `ZZ` (page 2525):

```
>>> z1 // z1
1
>>> z1 % z1
0
```

The `QQ` (page 2529) domain represents the field of rational numbers and does allow division:

```
>>> from sympy import QQ
>>> q1 = QQ(1, 2)
>>> q1
1/2
>>> q2 = QQ(2, 3)
>>> q2
2/3
>>> q1 / q2
3/4
>>> type(q1)
<class 'sympy.external.pythonmpq.PythonMPQ'>
```

In general code that is expected to work with elements of an arbitrary domain should not use the division operators `/`, `//` and `%`. Only the operators `+`, `-`, `*` and `**` (with nonnegative integer exponent) should be assumed to work with arbitrary domain elements. All other operations should be accessed as functions from the `Domain` (page 2504) object:

```
>>> ZZ.quo(ZZ(5), ZZ(3)) # 5 // 3
1
>>> ZZ.rem(ZZ(5), ZZ(3)) # 5 % 3
2
>>> ZZ.div(ZZ(5), ZZ(3)) # divmod(5, 3)
(1, 2)
>>> QQ.div(QQ(5), QQ(3))
(5/3, 0)
```

The `exquo()` (page 2511) function is used to compute an exact quotient. This is the analogue of `a / b` but where the division is expected to be exact (with no remainder) or an error will be raised:

```
>>> QQ.exquo(QQ(5), QQ(3))
5/3
>>> ZZ.exquo(ZZ(4), ZZ(2))
2
>>> ZZ.exquo(ZZ(5), ZZ(3))
Traceback (most recent call last):
...
ExactQuotientFailed: 3 does not divide 5 in ZZ
```

The exact methods and attributes of the domain elements are not guaranteed in general beyond the basic arithmetic operations. It should not be presumed that e.g. `ZZ` (page 2525) will

always be of type `int`. If `gmpy` or `gmpy2` is installed then the `mpz` or `mpq` types are used instead for `ZZ` (page 2525) and `QQ` (page 2529):

```
>>> from sympy import ZZ, QQ
>>> ZZ(2)
mpz(2)
>>> QQ(2, 3)
mpq(2, 3)
```

The `mpz` type is faster than Python's standard `int` type for operations with large integers although for smaller integers the difference is not so significant. The `mpq` type representing rational numbers is implemented in C rather than Python and is many times faster than the pure Python implementation of `QQ` (page 2529) that is used when `gmpy` is not installed.

In general the Python type used for the elements of a domain can be checked from the `dtype` (page 2510) attribute of the domain. When `gmpy` is installed the `dtype` for `ZZ` (page 2525) is `mpz` which is not an actual type and can not be used with `isinstance`. For this reason the `of_type()` (page 2516) method can be used to check if an object is an element of `dtype` (page 2510):

```
>>> z = ZZ(2)
>>> type(z)
<class 'int'>
>>> ZZ.dtype
<class 'int'>
>>> ZZ.of_type(z)
True
```

## Domain elements vs sympy expressions

Note that domain elements are not of the same type as ordinary sympy expressions which are subclasses of `Expr` (page 947) such as `Integer` (page 987). Ordinary sympy expressions are created with the `sympify()` (page 918) function.:

```
>>> from sympy import sympify
>>> z1_sympy = sympify(2) # Normal sympy object
>>> z1_sympy
2
>>> type(z1_sympy)
<class 'sympy.core.numbers.Integer'>
>>> from sympy import Expr
>>> isinstance(z1_sympy, Expr)
True
```

It is important when working with the domains not to mix sympy expressions with domain elements even though it will sometimes work in simple cases. Each domain object has the methods `to_sympy()` (page 2517) and `from_sympy()` (page 2513) for converting back and forth between sympy expressions and domain elements:

```
>>> z_sympy = sympify(2)
>>> z_zz = ZZ.from_sympy(z_sympy)
>>> z_zz
2
```

(continues on next page)

(continued from previous page)

```
>>> type(z_sympy)
<class 'sympy.core.numbers.Integer'>
>>> type(z_zz)
<class 'int'>
>>> ZZ.to_sympy(z_zz)
2
>>> type(ZZ.to_sympy(z_zz))
<class 'sympy.core.numbers.Integer'>
```

Any particular domain will only be able to represent some sympy expressions so conversion will fail if the expression can not be represented in the domain:

```
>>> from sympy import sqrt
>>> e = sqrt(2)
>>> e
sqrt(2)
>>> ZZ.from_sympy(e)
Traceback (most recent call last):
...
CoercionFailed: expected an integer, got sqrt(2)
```

We have already seen that in some cases we can use the domain object itself as a constructor e.g. `QQ(2)`. This will generally work provided the arguments given are valid for the *dtype* (page 2510) of the domain. Although it is convenient to use this in interactive sessions and in demonstrations it is generally better to use the *from\_sympy()* (page 2513) method for constructing domain elements from sympy expressions (or from objects that can be sympified to sympy expressions).

It is important not to mix domain elements with other Python types such as `int`, `float`, as well as standard sympy *Expr* (page 947) expressions. When working in a domain, care should be taken as some Python operations will do this implicitly. for example the `sum` function will use the regular `int` value of zero so that `sum([a, b])` is effectively evaluated as  $(0 + a) + b$  where `0` is of type `int`.

Every domain is at least a ring if not a field and as such is guaranteed to have two elements in particular corresponding to 1 and 0. The domain object provides domain elements for these as the attributes *one* (page 2516) and *zero* (page 2519). These are useful for something like Python's `sum` function which allows to provide an alternative object as the "zero":

```
>>> ZZ.one
1
>>> ZZ.zero
0
>>> sum([ZZ(1), ZZ(2)]) # don't do this (even it sometimes works)
3
>>> sum([ZZ(1), ZZ(2)], ZZ.zero) # provide the zero from the domain
3
```

A standard pattern then for performing calculations in a domain is:

1. Start with sympy *Expr* (page 947) instances representing expressions.
2. Choose an appropriate domain that can represent the expressions.
3. Convert all expressions to domain elements using *from\_sympy()* (page 2513).

4. Perform the calculation with the domain elements.
5. Convert back to `Expr` (page 947) with `to_sympy()` (page 2517).

Here is an implementation of the `sum` function that illustrates these steps and sums some integers but performs the calculation using the domain elements rather than standard sympy expressions:

```
def sum_domain(expressions_sympy):
 """Sum sympy expressions but performing calculations in domain ZZ"""

 # Convert to domain
 expressions_dom = [ZZ.from_sympy(e) for e in expressions_sympy]

 # Perform calculations in the domain
 result_dom = ZZ.zero
 for e_dom in expressions_dom:
 result_dom += e_dom

 # Convert the result back to Expr
 result_sympy = ZZ.to_sympy(result_dom)
 return result_sympy
```

## Gaussian integers and Gaussian rationals

The two example domains that we have seen so far are `ZZ` (page 2525) and `QQ` (page 2529) representing the integers and the rationals respectively. There are other simple domains such as `ZZ_I` (page 2534) and `QQ_I` (page 2536) representing the **Gaussian integers** and **Gaussian rationals**. The Gaussian integers are numbers of the form  $a\sqrt{-1}+b$  where  $a$  and  $b$  are integers. The Gaussian rationals are defined similarly except that  $a$  and  $b$  can be rationals. We can use the Gaussian domains like:

```
>>> from sympy import ZZ_I, QQ_I, I
>>> z = ZZ_I.from_sympy(1 + 2*I)
>>> z
(1 + 2*I)
>>> z**2
(-3 + 4*I)
```

Note the contrast with the way this calculation works in the tree representation where `expand()` (page 1053) is needed to get the reduced form:

```
>>> from sympy import expand, I
>>> z = 1 + 2*I
>>> z**2
(1 + 2*I)**2
>>> expand(z**2)
-3 + 4*I
```

The `ZZ_I` (page 2534) and `QQ_I` (page 2536) domains are implemented by the classes `GaussianIntegerRing` (page 2534) and `GaussianRationalField` (page 2536) and their elements by `GaussianInteger` (page 2536) and `GaussianRational` (page 2539) respectively. The internal representation for an element of `ZZ_I` (page 2534) or `QQ_I` (page 2536) is simply as a pair  $(a, b)$  of elements of `ZZ` (page 2525) or `QQ` (page 2529) respectively. The domain

[ZZ\\_I](#) (page 2534) is a ring with similar properties to [ZZ](#) (page 2525) whereas [QQ\\_I](#) (page 2536) is a field much like [QQ](#) (page 2529):

```
>>> ZZ.is_Field
False
>>> QQ.is_Field
True
>>> ZZ_I.is_Field
False
>>> QQ_I.is_Field
True
```

Since [QQ\\_I](#) (page 2536) is a field division by nonzero elements is always possible whereas in [ZZ\\_I](#) (page 2534) we have the important concept of the greatest common divisor (GCD):

```
>>> e1 = QQ_I.from_sympy(1+I)
>>> e2 = QQ_I.from_sympy(2-I/2)
>>> e1/e2
(6/17 + 10/17*I)
>>> ZZ_I.gcd(ZZ_I(5), ZZ_I.from_sympy(1+2*I))
(1 + 2*I)
```

## Finite fields

So far we have seen the domains [ZZ](#) (page 2525), [QQ](#) (page 2529), [ZZ\\_I](#) (page 2534), and [QQ\\_I](#) (page 2536). There are also domains representing the [Finite fields](#) although the implementation of these is incomplete. A finite field [GF\(p\)](#) (page 2522) of *prime* order can be constructed with FF or GF. A domain for the finite field of prime order  $p$  can be constructed with [GF\(p\)](#) (page 2522):

```
>>> from sympy import GF
>>> K = GF(5)
>>> two = K(2)
>>> two
2 mod 5
>>> two ** 2
4 mod 5
>>> two ** 3
3 mod 5
```

There is also FF as an alias for GF (standing for “finite field” and “Galois field” respectively). These are equivalent and both `FF(n)` and `GF(n)` will create a domain which is an instance of [FiniteField](#) (page 2522). The associated domain elements will be instances of [PythonFiniteField](#) (page 2525) or [GMPYFiniteField](#) (page 2525) depending on whether or not gmpy is installed.

Finite fields of order  $p^n$  where  $n \neq 1$  are not implemented. It is possible to use e.g. `GF(6)` or `GF(9)` but the resulting domain is *not* a field. It is just the integers modulo 6 or 9 and therefore has zero divisors and non-invertible elements:

```
>>> K = GF(6)
>>> K(3) * K(2)
0 mod 6
```

It would be good to have a proper implementation of prime-power order finite fields but this is not yet available in SymPy (contributions welcome!).

## Real and complex fields

The fields `RR` (page 2545) and `CC` (page 2546) are intended mathematically to correspond to the reals and the complex numbers,  $\mathbb{R}$  and  $\mathbb{C}$  respectively. The implementation of these uses floating point arithmetic. In practice this means that these are the domains that are used to represent expressions containing floats. Elements of `RR` (page 2545) are instances of the class `RealElement` (page 2546) and have an `mpf` tuple which is used to represent a float in `mpmath`. Elements of `CC` (page 2546) are instances of `ComplexElement` (page 2546) and have an `mpc` tuple which is a pair of `mpf` tuples representing the real and imaginary parts. See the [mpmath docs](#) for more about how floating point numbers are represented:

```
>>> from sympy import RR, CC
>>> xr = RR(3)
>>> xr
3.0
>>> xr._mpf_
(0, 3, 0, 2)
>>> zc = CC(3+1j)
>>> zc
(3.0 + 1.0j)
>>> zc._mpc_
((0, 3, 0, 2), (0, 1, 0, 1))
```

The use of approximate floating point arithmetic in these domains comes with all of the usual pitfalls. Many algorithms in the *sympy.polys* (page 2360) module are fundamentally designed for exact arithmetic making the use of these domains potentially problematic:

```
>>> RR('0.1') + RR('0.2') == RR('0.3')
False
```

Since these are implemented using `mpmath` which is a multiprecision library it is possible to create different domains with different working precisions. The default domains `RR` (page 2545) and `CC` (page 2546) use 53 binary digits of precision much like standard `double precision` floating point which corresponds to approximately 15 decimal digits:

```
>>> from sympy.polys.domains.realfield import RealField
>>> RR.precision
53
>>> RR.dps
15
>>> RR(1) / RR(3)
0.333333333333333
>>> RR100 = RealField(100)
>>> RR100.precision
100
>>> RR100.dps
29
>>> RR100(1) / RR100(3)
0.33333333333333333333333333333333333333333333333333
```

There is however a bug in the implementation of this so that actually a global precision setting is used by all [RealElement](#) (page 2546). This means that just creating `RR100` above has altered the global precision and we will need to restore it in the doctest here:

```
>>> RR(1) / RR(3) # wrong result!
0.33333333333333333333333333333333
>>> dummy = RealField(53) # hack to restore precision
>>> RR(1) / RR(3) # restored
0.3333333333333333
```

(Obviously that should be fixed!)

## Algebraic number fields

An [algebraic extension](#) of the rationals  $\mathbb{Q}$  is known as an [algebraic number field](#) and these are implemented in sympy as `QQ<a>` (page 2539). The natural syntax for these would be something like `QQ(sqrt(2))` however `QQ()` is already overloaded as the constructor for elements of `QQ` (page 2529). These domains are instead created using the `algebraic_field()` (page 2508) method e.g. `QQ.algebraic_field(sqrt(2))`. The resulting domain will be an instance of [AlgebraicField](#) (page 2539) with elements that are instances of [ANP](#) (page 2568).

The printing support for these is less developed but we can use `to_sympy()` (page 2517) to take advantage of the corresponding [Expr](#) (page 947) printing support:

```
>>> K = QQ.algebraic_field(sqrt(2))
>>> K
QQ<sqrt(2)>
>>> b = K.one + K.from_sympy(sqrt(2))
>>> b
ANP([1, 1], [1, 0, -2], QQ)
>>> K.to_sympy(b)
1 + sqrt(2)
>>> b ** 2
ANP([2, 3], [1, 0, -2], QQ)
>>> K.to_sympy(b**2)
2*sqrt(2) + 3
```

The raw printed display immediately shows the internal representation of the elements as [ANP](#) (page 2568) instances. The field  $\mathbb{Q}(\sqrt{2})$  consists of numbers of the form  $a\sqrt{2} + b$  where  $a$  and  $b$  are rational numbers. Consequently every number in this field can be represented as a pair  $(a, b)$  of elements of `QQ` (page 2529). The domain element stores these two in a list and also stores a list representation of the *minimal polynomial* for the extension element  $\sqrt{2}$ . There is a sympy function `minpoly()` (page 2711) that can compute the minimal polynomial of any algebraic expression over the rationals:

```
>>> from sympy import minpoly, Symbol
>>> x = Symbol('x')
>>> minpoly(sqrt(2), x)
x**2 - 2
```

In the dense polynomial representation as a list of coefficients this polynomial is represented as `[1, 0, -2]` as seen in the [ANP](#) (page 2568) display for the elements of `QQ<sqrt(2)>` above.

It is also possible to create an algebraic number field with multiple generators such as  $\mathbb{Q}(\sqrt{2}, \sqrt{3})$ :



```

>>> K = QQ.algebraic_field(sqrt(2), sqrt(3))
>>> K
QQ<sqrt(2) + sqrt(3)>
>>> sqrt2 = K.from_sympy(sqrt(2))
>>> sqrt3 = K.from_sympy(sqrt(3))
>>> p = (K.one + sqrt2) * (K.one + sqrt3)
>>> p
ANP([1/2, 1, -3/2], [1, 0, -10, 0, 1], QQ)
>>> K.to_sympy(p)
1 + sqrt(2) + sqrt(3) + sqrt(6)
>>> K.to_sympy(p**2)
4*sqrt(6) + 6*sqrt(3) + 8*sqrt(2) + 12

```

Here the algebraic extension  $\mathbb{Q}(\sqrt{2}, \sqrt{3})$  is converted to the (isomorphic)  $\mathbb{Q}(\sqrt{2} + \sqrt{3})$  with a single generator  $\sqrt{2} + \sqrt{3}$ . It is always possible to find a single generator like this due to the [primitive element theorem](#). There is a sympy function [primitive\\_element\(\)](#) (page 2712) that can compute the minimal polynomial for a primitive element of an extension:

```

>>> from sympy import primitive_element, minpoly
>>> e = primitive_element([sqrt(2), sqrt(3)], x)
>>> e[0]
x**4 - 10*x**2 + 1
>>> e[0].subs(x, sqrt(2) + sqrt(3)).expand()
0

```

The minimal polynomial  $x^4 - 10x^2 + 1$  has the dense list representation  $[1, 0, -10, 0, 1]$  as seen in the [ANP](#) (page 2568) output above. What the primitive element theorem means is that all algebraic number fields can be represented as an extension of the rationals by a single generator with some minimal polynomial. Calculations over the algebraic number field only need to take advantage of the minimal polynomial and that makes it possible to compute all arithmetic operations and also to carry out higher level operations like factorisation of polynomials.

## Polynomial ring domains

There are also domains implemented to represent a polynomial ring like  $K[x]$  (page 2547) which is the domain of polynomials in the generator  $x$  with coefficients over another domain  $K$ :

```

>>> from sympy import ZZ, symbols
>>> x = symbols('x')
>>> K = ZZ[x]
>>> K
ZZ[x]
>>> x_dom = K(x)
>>> x_dom + K.one
x + 1

```

All the operations discussed before will work with elements of a polynomial ring:

```

>>> p = x_dom + K.one
>>> p

```

(continues on next page)

(continued from previous page)

```

x + 1
>>> p + p
2*x + 2
>>> p - p
0
>>> p * p
x**2 + 2*x + 1
>>> p ** 3
x**3 + 3*x**2 + 3*x + 1
>>> K.exquo(x_dom**2 - K.one, x_dom - K.one)
x + 1

```

The internal representation of elements of  $K[x]$  is different from the way that ordinary sympy (*Expr* (page 947)) expressions are represented. The *Expr* (page 947) representation of any expression is as a tree e.g.:

```

>>> from sympy import srepr
>>> K = ZZ[x]
>>> p_expr = x**2 + 2*x + 1
>>> p_expr
x**2 + 2*x + 1
>>> srepr(p_expr)
"Add(Pow(Symbol('x'), Integer(2)), Mul(Integer(2), Symbol('x')), Integer(1))"

```

Here the expression is a tree where the top node is an *Add* (page 1013) and its children nodes are *Pow* (page 1005) etc. This tree representation makes it possible to represent equivalent expressions in different ways e.g.:

```

>>> x = symbols('x')
>>> p_expr = x*(x + 1) + x
>>> p_expr
x*(x + 1) + x
>>> p_expr.expand()
x**2 + 2*x

```

By contrast the domain  $\mathbb{Z}\mathbb{Z}[x]$  represents only polynomials and does so by simply storing the non-zero coefficients of the expanded polynomial (the “sparse” polynomial representation). In particular elements of  $\mathbb{Z}\mathbb{Z}[x]$  are represented as a Python dict. Their type is *PolyElement* (page 2554) which is a subclass of dict. Converting to a normal dict shows the internal representation:

```

>>> x = symbols('x')
>>> K = ZZ[x]
>>> x_dom = K(x)
>>> p_dom = K(3)*x_dom**2 + K(2)*x_dom + K(7)
>>> p_dom
3*x**2 + 2*x + 7
>>> dict(p_dom)
{(0,): 7, (1,): 2, (2,): 3}

```

This internal form makes it impossible to represent unexpanded multiplications so any multiplication of elements of  $\mathbb{Z}\mathbb{Z}[x]$  will always be expanded:

```

>>> x = symbols('x')
>>> K = ZZ[x]
>>> x_dom = K(x)
>>> p_expr = x * (x + 1) + x
>>> p_expr
x*(x + 1) + x
>>> p_dom = x_dom * (x_dom + K.one) + x_dom
>>> p_dom
x**2 + 2*x

```

These same considerations apply to powers:

```

>>> (x + 1) ** 2
(x + 1)**2
>>> (x_dom + K.one) ** 2
x**2 + 2*x + 1

```

We can also construct multivariate polynomial rings:

```

>>> x, y = symbols('x, y')
>>> K = ZZ[x,y]
>>> xk = K(x)
>>> yk = K(y)
>>> xk**2*yk + xk + yk
x**2*y + x + y

```

It is also possible to construct nested polynomial rings (although it is less efficient). The ring  $K[x][y]$  is formally equivalent to  $K[x,y]$  although their implementations in sympy are different:

```

>>> K = ZZ[x][y]
>>> p = K(x**2 + x*y + y**2)
>>> p
y**2 + x*y + x**2
>>> dict(p)
{(0,): x**2, (1,): x, (2,): 1}

```

Here the coefficients like  $x**2$  are instances of [PolyElement](#) (page 2554) as well so this is a dict where the values are also dicts. The full representation is more like:

```

>>> {k: dict(v) for k, v in p.items()}
{(0,): {(2,): 1}, (1,): {(1,): 1}, (2,): {(0,): 1}}

```

The multivariate ring domain  $ZZ[x,y]$  has a more efficient representation as a single flattened dict:

```

>>> K = ZZ[x,y]
>>> p = K(x**2 + x*y + y**2)
>>> p
x**2 + x*y + y**2
>>> dict(p)
{(0, 2): 1, (1, 1): 1, (2, 0): 1}

```

The difference in efficiency between these representations grows as the number of generators increases i.e.  $ZZ[x,y,z,t,\dots]$  vs  $ZZ[x][y][z][t]\dots$

## Old (dense) polynomial rings

In the last section we saw that the domain representation of a polynomial ring like  $K[x]$  (page 2547) uses a sparse representation of a polynomial as a dict mapping monomial exponents to coefficients. There is also an older version of  $K[x]$  (page 2547) that uses the dense *DMP representation* (page 2480). We can create these two versions of  $K[x]$  (page 2547) using `poly_ring()` (page 2516) and `old_poly_ring()` (page 2516) where the syntax  $K[x]$  is equivalent to `K.poly_ring(x)`:

```
>>> K1 = ZZ.poly_ring(x)
>>> K2 = ZZ.old_poly_ring(x)
>>> K1
ZZ[x]
>>> K2
ZZ[x]
>>> K1 == ZZ[x]
True
>>> K2 == ZZ[x]
False
>>> p1 = K1.from_sympy(x**2 + 1)
>>> p2 = K2.from_sympy(x**2 + 1)
>>> p1
x**2 + 1
>>> p2
x**2 + 1
>>> type(K1)
<class 'sympy.polys.domains.polynomialring.PolynomialRing'>
>>> type(p1)
<class 'sympy.polys.rings.PolyElement'>
>>> type(K2)
<class 'sympy.polys.domains.old_polynomialring.GlobalPolynomialRing'>
>>> type(p2)
<class 'sympy.polys.polyclasses.DMP'>
```

The internal representation of the old polynomial ring domain is the *DMP* (page 2561) representation as a list of (lists of) coefficients:

```
>>> repr(p2)
'DMP([1, 0, 1], ZZ, ZZ[x])'
```

The most notable use of the *DMP* (page 2561) representation of polynomials is as the internal representation used by *Poly* (page 2378) (this is discussed later in this page of the docs).

## PolyRing vs PolynomialRing

You might just want to perform calculations in some particular polynomial ring without being concerned with implementing something that works for arbitrary domains. In that case you can construct the ring more directly with the `ring()` (page 2551) function:

```
>>> from sympy import ring
>>> K, xr, yr = ring([x, y], ZZ)
>>> K
```

(continues on next page)

(continued from previous page)

```

Polynomial ring in x, y over ZZ with lex order
>>> xr**2 - yr**2
x**2 - y**2
>>> (xr**2 - yr**2) // (xr - yr)
x + y

```

The object `K` here represents the ring and is an instance of *PolyRing* (page 2553) but is not a **polys domain** (it is not an instance of a subclass of *Domain* (page 2504) so it can not be used with *Poly* (page 2378)). In this way the implementation of polynomial rings that is used in the domain system can be used independently of the domain system.

The purpose of the domain system is to provide a unified interface for working with and converting between different representations of expressions. To make the *PolyRing* (page 2553) implementation usable in that context the *PolynomialRing* (page 2547) class is a wrapper around the *PolyRing* (page 2553) class that provides the interface expected in the domain system. That makes this implementation of polynomial rings usable as part of the broader codebase that is designed to work with expressions from different domains. The domain for polynomial rings is a distinct object from the ring returned by *ring()* (page 2551) although both have the same elements:

```

>>> K, xr, yr = ring([x, y], ZZ)
>>> K
Polynomial ring in x, y over ZZ with lex order
>>> K2 = ZZ[x,y]
>>> K2
ZZ[x,y]
>>> K2.ring
Polynomial ring in x, y over ZZ with lex order
>>> K2.ring == K
True
>>> K(x+y)
x + y
>>> K2(x+y)
x + y
>>> type(K(x+y))
<class 'sympy.polys.rings.PolyElement'>
>>> type(K2(x+y))
<class 'sympy.polys.rings.PolyElement'>
>>> K(x+y) == K2(x+y)
True

```

## Rational function fields

Some domains are classified as fields and others are not. The principal difference between a field and a non-field domain is that in a field it is always possible to divide any element by any nonzero element. It is usually possible to convert any domain to a field that contains that domain with the *get\_field()* (page 2514) method:

```

>>> from sympy import ZZ, QQ, symbols
>>> x, y = symbols('x, y')
>>> ZZ.is_Field

```

(continues on next page)

(continued from previous page)

```
False
>>> QQ.is_Field
True
>>> QQ[x]
QQ[x]
>>> QQ[x].is_Field
False
>>> QQ[x].get_field()
QQ(x)
>>> QQ[x].get_field().is_Field
True
>>> QQ.frac_field(x)
QQ(x)
```

This introduces a new kind of domain  $K(x)$  (page 2548) representing a rational function field in the generator  $x$  over another domain  $K$ . It is not possible to construct the domain  $QQ(x)$  with the `()` syntax so the easiest ways to create it are using the domain methods `frac_field()` (page 2512) (`QQ.frac_field(x)`) or `get_field()` (page 2514) (`QQ[x].get_field()`). The `frac_field()` (page 2512) method is the more direct approach.

The rational function field  $K(x)$  (page 2548) is an instance of `RationalField` (page 2530). This domain represents functions of the form  $p(x)/q(x)$  for polynomials  $p$  and  $q$ . The domain elements are represented as a pair of polynomials in  $K[x]$  (page 2547):

```
>>> K = QQ.frac_field(x)
>>> xk = K(x)
>>> f = xk / (K.one + xk**2)
>>> f
x/(x**2 + 1)
>>> f.numer
x
>>> f.denom
x**2 + 1
>>> QQ[x].of_type(f.numer)
True
>>> QQ[x].of_type(f.denom)
True
```

Cancellation between the numerator and denominator is automatic in this field:

```
>>> p1 = xk**2 - 1
>>> p2 = xk - 1
>>> p1
x**2 - 1
>>> p2
x - 1
>>> p1 / p2
x + 1
```

Computing this cancellation can be slow which makes rational function fields potentially slower than polynomial rings or algebraic fields.

Just like in the case of polynomial rings there is both a new (sparse) and old (dense) version of fraction fields:

```

>>> K1 = QQ.frac_field(x)
>>> K2 = QQ.old_frac_field(x)
>>> K1
QQ(x)
>>> K2
QQ(x)
>>> type(K1)
<class 'sympy.polys.domains.fractionfield.FractionField'>
>>> type(K2)
<class 'sympy.polys.domains.old_fractionfield.FractionField'>

```

Also just like in the case of polynomials rings the implementation of rational function fields can be used independently of the domain system:

```

>>> from sympy import field
>>> K, xf, yf = field([x, y], ZZ)
>>> xf / (1 - yf)
-x/(y - 1)

```

Here K is an instance of *FracField* (page 2561) rather than *RationalField* (page 2530) as it would be for the domain  $\mathbb{Z}\langle x, y \rangle$ .

## Expression domain

The final domain to consider is the “expression domain” which is known as *EX* (page 2549). Expressions that can not be represented using the other domains can be always represented using the expression domain. An element of *EX* (page 2549) is actually just a wrapper around a *Expr* (page 947) instance:

```

>>> from sympy import EX
>>> p = EX.from_sympy(1 + x)
>>> p
EX(x + 1)
>>> type(p)
<class 'sympy.polys.domains.expressiondomain.ExpressionDomain.Expression'>
>>> p.ex
x + 1
>>> type(p.ex)
<class 'sympy.core.add.Add'>

```

For other domains the domain representation of expressions is usually more efficient than the tree representation used by *Expr* (page 947). In *EX* (page 2549) the internal representation is *Expr* (page 947) so it is clearly not more efficient. The purpose of the *EX* (page 2549) domain is to be able to wrap up arbitrary expressions in an interface that is consistent with the other domains. The *EX* (page 2549) domain is used as a fallback when an appropriate domain can not be found. Although this does not offer any particular efficiency it does allow the algorithms that are implemented to work over arbitrary domains to be usable when working with expressions that do not have an appropriate domain representation.

## Choosing a domain

In the workflow described above the idea is to start with some sympy expressions, choose a domain and convert all the expressions into that domain in order to perform some calculation. The obvious question that arises is how to choose an appropriate domain to represent some sympy expressions. For this there is a function `construct_domain()` (page 2427) which takes a list of expressions and will choose a domain and convert all of the expressions to that domain:

```
>>> from sympy import construct_domain, Integer
>>> elements_sympy = [Integer(3), Integer(2)] # elements as Expr instances
>>> elements_sympy
[3, 2]
>>> K, elements_K = construct_domain(elements_sympy)
>>> K
ZZ
>>> elements_K
[3, 2]
>>> type(elements_sympy[0])
<class 'sympy.core.numbers.Integer'>
>>> type(elements_K[0])
<class 'int'>
```

In this example we see that the two integers 3 and 2 can be represented in the domain `ZZ` (page 2525). The expressions have been converted to elements of that domain which in this case means the `int` type rather than instances of `Expr` (page 947). It is not necessary to explicitly create `Expr` (page 947) instances when the inputs can be sympified so e.g. `construct_domain([3, 2])` would give the same output as above.

Given more complicated inputs `construct_domain()` (page 2427) will choose more complicated domains:

```
>>> from sympy import Rational, symbols
>>> x, y = symbols('x, y')
>>> construct_domain([Rational(1, 2), Integer(3)])[0]
QQ
>>> construct_domain([2*x, 3])[0]
ZZ[x]
>>> construct_domain([x/2, 3])[0]
QQ[x]
>>> construct_domain([2/x, 3])[0]
ZZ(x)
>>> construct_domain([x, y])[0]
ZZ[x,y]
```

If any noninteger rational numbers are found in the inputs then the ground domain will be `QQ` (page 2529) rather than `ZZ` (page 2525). If any symbol is found in the inputs then a `PolynomialRing` (page 2547) will be created. A multivariate polynomial ring such as `QQ[x, y]` can also be created if there are multiple symbols in the inputs. If any symbols appear in the denominators then a `RationalField` (page 2530) like `QQ(x)` will be created instead.

Some of the domains above are fields and others are (non-field) rings. In some contexts it is necessary to have a field domain so that division is possible and for this `construct_domain()` (page 2427) has an option `field=True` which will force the construction of a field domain even if the expressions can all be represented in a non-field ring:



```
>>> construct_domain([1, 2], field=True)[0]
QQ
>>> construct_domain([2*x, 3], field=True)[0]
ZZ(x)
>>> construct_domain([x/2, 3], field=True)[0]
ZZ(x)
>>> construct_domain([2/x, 3], field=True)[0]
ZZ(x)
>>> construct_domain([x, y], field=True)[0]
ZZ(x,y)
```

By default `construct_domain()` (page 2427) will not construct an algebraic extension field and will instead use the `EX` (page 2549) domain (`ExpressionDomain` (page 2549)). The keyword argument `extension=True` can be used to construct an `AlgebraicField` (page 2539) if the inputs are irrational but algebraic:

```
>>> from sympy import sqrt
>>> construct_domain([sqrt(2)])[0]
EX
>>> construct_domain([sqrt(2)], extension=True)[0]
QQ<sqrt(2)>
>>> construct_domain([sqrt(2), sqrt(3)], extension=True)[0]
QQ<sqrt(2) + sqrt(3)>
```

When there are algebraically independent transcendentals in the inputs a `PolynomialRing` (page 2547) or `RationalField` (page 2530) will be constructed treating those transcendentals as generators:

```
>>> from sympy import sin, cos
>>> construct_domain([sin(x), y])[0]
ZZ[y,sin(x)]
```

However if there is a possibility that the inputs are not algebraically independent then the domain will be `EX` (page 2549):

```
>>> construct_domain([sin(x), cos(x)])[0]
EX
```

Here  $\sin(x)$  and  $\cos(x)$  are not algebraically independent since  $\sin(x)^2 + \cos(x)^2 = 1$ .

## Converting elements between different domains

It is often useful to combine calculations performed over different domains. However just as it is important to avoid mixing domain elements with normal sympy expressions and other Python types it is also important to avoid mixing elements from different domains. The `convert_from()` (page 2508) method is used to convert elements from one domain into elements of another domain:

```
>>> num_zz = ZZ(3)
>>> ZZ.of_type(num_zz)
True
```

(continues on next page)

(continued from previous page)

```
>>> num_qq = QQ.convert_from(num_zz, ZZ)
>>> ZZ.of_type(num_qq)
False
>>> QQ.of_type(num_qq)
True
```

The `convert()` (page 2508) method can be called without specifying the source domain as the second argument e.g.:

```
>>> QQ.convert(ZZ(2))
2
```

This works because `convert()` (page 2508) can check the type of `ZZ(2)` and can try to work out what domain (`ZZ` (page 2525)) it is an element of. Certain domains like `ZZ` (page 2525) and `QQ` (page 2529) are treated as special cases to make this work. Elements of more complicated domains are instances of subclasses of `DomainElement` (page 2519) which has a `parent()` (page 2519) method that can identify the domain that the element belongs to. For example in the polynomial ring `ZZ[x]` we have:

```
>>> from sympy import ZZ, Symbol
>>> x = Symbol('x')
>>> K = ZZ[x]
>>> K
ZZ[x]
>>> p = K(x) + K.one
>>> p
x + 1
>>> type(p)
<class 'sympy.polys.rings.PolyElement'>
>>> p.parent()
ZZ[x]
>>> p.parent() == K
True
```

It is more efficient though to call `convert_from()` (page 2508) with the source domain specified as the second argument:

```
>>> QQ.convert_from(ZZ(2), ZZ)
2
```

## Unifying domains

When we want to combine elements from two different domains and perform mixed calculations with them we need to

1. Choose a new domain that can represent all elements of both.
2. Convert all elements to the new domain.
3. Perform the calculation in the new domain.

The key question arising from point 1. is how to choose a domain that can represent the elements of both domains. For this there is the `unify()` (page 2519) method:

```

>>> x1, K1 = ZZ(2), ZZ
>>> y2, K2 = QQ(3, 2), QQ
>>> K1
ZZ
>>> K2
QQ
>>> K3 = K1.unify(K2)
>>> K3
QQ
>>> x3 = K3.convert_from(x1, K1)
>>> y3 = K3.convert_from(y2, K2)
>>> x3 + y3
7/2

```

The `unify()` (page 2519) method will find a domain that encompasses both domains so in this example `ZZ.unify(QQ)` gives `QQ` (page 2529) because every element of `ZZ` (page 2525) can be represented as an element of `QQ` (page 2529). This means that all inputs (`x1` and `y2`) can be converted to the elements of the common domain `K3` (as `x3` and `y3`). Once in the common domain we can safely use arithmetic operations like `+`. In this example one domain is a superset of the other and we see that `K1.unify(K2) == K2` so it is not actually necessary to convert `y2`. In general though `K1.unify(K2)` can give a new domain that is not equal to either `K1` or `K2`.

The `unify()` (page 2519) method understands how to combine different polynomial ring domains and how to unify the base domain:

```

>>> ZZ[x].unify(ZZ[y])
ZZ[x,y]
>>> ZZ[x,y].unify(ZZ[y])
ZZ[x,y]
>>> ZZ[x].unify(QQ)
QQ[x]

```

It is also possible to unify algebraic fields and rational function fields as well:

```

>>> K1 = QQ.algebraic_field(sqrt(2))[x]
>>> K2 = QQ.algebraic_field(sqrt(3))[y]
>>> K1
QQ<sqrt(2)>[x]
>>> K2
QQ<sqrt(3)>[y]
>>> K1.unify(K2)
QQ<sqrt(2) + sqrt(3)>[x,y]
>>> QQ.frac_field(x).unify(ZZ[y])
ZZ(x,y)

```

## Internals of a Poly

We are now in a position to understand how the *Poly* (page 2378) class works internally. This is the public interface of *Poly* (page 2378):

```
>>> from sympy import Poly, symbols, ZZ
>>> x, y, z, t = symbols('x, y, z, t')
>>> p = Poly(x**2 + 1, x, domain=ZZ)
>>> p
Poly(x**2 + 1, x, domain='ZZ')
>>> p.gens
(x,)
>>> p.domain
ZZ
>>> p.all_coeffs()
[1, 0, 1]
>>> p.as_expr()
x**2 + 1
```

This is the internal implementation of *Poly* (page 2378):

```
>>> d = p.rep # internal representation of Poly
>>> d
DMP([1, 0, 1], ZZ, None)
>>> d.rep # internal representation of DMP
[1, 0, 1]
>>> type(d.rep)
<class 'list'>
>>> type(d.rep[0])
<class 'int'>
>>> d.dom
ZZ
```

The internal representation of a *Poly* (page 2378) instance is an instance of *DMP* (page 2561) which is the class used for domain elements in the old polynomial ring domain *old\_poly\_ring()* (page 2516). This represents the polynomial as a list of coefficients which are themselves elements of a domain and keeps a reference to their domain (*ZZ* (page 2525) in this example).

## Choosing a domain for a Poly

If the domain is not specified for the *Poly* (page 2378) constructor then it is inferred using *construct\_domain()* (page 2427). Arguments like *field=True* are passed along to *construct\_domain()* (page 2427):

```
>>> from sympy import sqrt
>>> Poly(x**2 + 1, x)
Poly(x**2 + 1, x, domain='ZZ')
>>> Poly(x**2 + 1, x, field=True)
Poly(x**2 + 1, x, domain='QQ')
>>> Poly(x**2/2 + 1, x)
Poly(1/2*x**2 + 1, x, domain='QQ')
```

(continues on next page)

(continued from previous page)

```
>>> Poly(x**2 + sqrt(2), x)
Poly(x**2 + sqrt(2), x, domain='EX')
>>> Poly(x**2 + sqrt(2), x, extension=True)
Poly(x**2 + sqrt(2), x, domain='QQ<sqrt(2)>')
```

It is also possible to use the extension argument to specify generators of an extension even if no extension is required to represent the coefficients although this does not work when using `construct_domain()` (page 2427) directly. A list of extension elements will be passed to `primitive_element()` (page 2712) to create an appropriate *AlgebraicField* (page 2539) domain:

```
>>> from sympy import construct_domain
>>> Poly(x**2 + 1, x)
Poly(x**2 + 1, x, domain='ZZ')
>>> Poly(x**2 + 1, x, extension=sqrt(2))
Poly(x**2 + 1, x, domain='QQ<sqrt(2)>')
>>> Poly(x**2 + 1, x, extension=[sqrt(2), sqrt(3)])
Poly(x**2 + 1, x, domain='QQ<sqrt(2) + sqrt(3)>')
>>> construct_domain([1, 0, 1], extension=sqrt(2))[0]
ZZ
```

(Perhaps `construct_domain()` (page 2427) should do the same as *Poly* (page 2378) here...)

## Choosing generators

If there are symbols other than the generators then a polynomial ring or rational function field domain will be created. The domain used for the coefficients in this case is the sparse (“new”) polynomial ring:

```
>>> p = Poly(x**2*y + z, x)
>>> p
Poly(y*x**2 + z, x, domain='ZZ[y,z]')
>>> p.gens
(x,)
>>> p.domain
ZZ[y,z]
>>> p.domain == ZZ[y,z]
True
>>> p.domain == ZZ.poly_ring(y, z)
True
>>> p.domain == ZZ.old_poly_ring(y, z)
False
>>> p.rep.rep
[y, 0, z]
>>> p.rep.rep[0]
y
>>> type(p.rep.rep[0])
<class 'sympy.polys.rings.PolyElement'>
>>> dict(p.rep.rep[0])
{(1, 0): 1}
```

What we have here is a strange hybrid of dense and sparse implementations. The *Poly*

(page 2378) instance considers itself to be an univariate polynomial in the generator  $x$  but with coefficients from the domain  $\mathbb{Z}[y, z]$ . The internal representation of the `Poly` (page 2378) is a list of coefficients in the “dense univariate polynomial” (DUP) format. However each coefficient is implemented as a sparse polynomial in  $y$  and  $z$ .

If we make  $x$ ,  $y$  and  $z$  all be generators for the `Poly` (page 2378) then we get a fully dense DMP list of lists of lists representation:

```
>>> p = Poly(x**2*y + z, x, y, z)
>>> p
Poly(x**2*y + z, x, y, z, domain='ZZ')
>>> p.rep
DMP([[[1], []], [[]], [[1, 0]]], ZZ, None)
>>> p.rep.rep
[[[1], []], [[]], [[1, 0]]]
>>> p.rep.rep[0][0][0]
1
>>> type(p.rep.rep[0][0][0])
<class 'int'>
```

On the other hand we can make a `Poly` (page 2378) with a fully sparse representation by choosing a generator that is not in the expression at all:

```
>>> p = Poly(x**2*y + z, t)
>>> p
Poly(x**2*y + z, t, domain='ZZ[x,y,z]')
>>> p.rep
DMP([x**2*y + z], ZZ[x,y,z], None)
>>> p.rep.rep[0]
x**2*y + z
>>> type(p.rep.rep[0])
<class 'sympy.polys.rings.PolyElement'>
>>> dict(p.rep.rep[0])
{(0, 0, 1): 1, (2, 1, 0): 1}
```

If no generators are provided to the `Poly` (page 2378) constructor then it will attempt to choose generators so that the expression is polynomial in those. In the common case that the expression is a polynomial expression in some symbols then those symbols will be taken as generators. However other non-symbol expressions can also be taken as generators:

```
>>> Poly(x**2*y + z)
Poly(x**2*y + z, x, y, z, domain='ZZ')
>>> from sympy import pi, exp
>>> Poly(exp(x) + exp(2*x) + 1)
Poly((exp(x))**2 + (exp(x)) + 1, exp(x), domain='ZZ')
>>> Poly(pi*x)
Poly(x*pi, x, pi, domain='ZZ')
>>> Poly(pi*x, x)
Poly(pi*x, x, domain='ZZ[pi]')
```

## Algebraically dependent generators

Taking  $\exp(x)$  or  $\pi$  as generators for a *Poly* (page 2378) or for its polynomial ring domain is mathematically valid because these objects are transcendental and so the ring extension containing them is isomorphic to a polynomial ring. Since  $x$  and  $\exp(x)$  are algebraically independent it is also valid to use both as generators for the same *Poly* (page 2378). However some other combinations of generators are invalid such as  $x$  and  $\sqrt{x}$  or  $\sin(x)$  and  $\cos(x)$ . These examples are invalid because the generators are not algebraically independent (e.g.  $\sqrt{x}^2 = x$  and  $\sin(x)^2 + \cos(x)^2 = 1$ ). The implementation is not able to detect these algebraic relationships though:

```
>>> from sympy import sin, cos, sqrt
>>> Poly(x*exp(x)) # fine
Poly(x*(exp(x)), x, exp(x), domain='ZZ')
>>> Poly(sin(x)+cos(x)) # not fine
Poly((cos(x)) + (sin(x)), cos(x), sin(x), domain='ZZ')
>>> Poly(x + sqrt(x)) # not fine
Poly(x + (sqrt(x)), x, sqrt(x), domain='ZZ')
```

Calculations with a *Poly* (page 2378) such as this are unreliable because zero-testing will not work properly in this implementation:

```
>>> p1 = Poly(x, x, sqrt(x))
>>> p2 = Poly(sqrt(x), x, sqrt(x))
>>> p1
Poly(x, x, sqrt(x), domain='ZZ')
>>> p2
Poly((sqrt(x)), x, sqrt(x), domain='ZZ')
>>> p3 = p1 - p2**2
>>> p3
Poly(x - (sqrt(x))**2, x, sqrt(x), domain='ZZ')
>>> p3.as_expr()
0
```

This aspect of *Poly* (page 2378) could be improved by:

1. Expanding the domain system with new domains that can represent more classes of algebraic extension.
2. Improving the detection of algebraic dependencies in *construct\_domain()* (page 2427).
3. Improving the automatic selection of generators.

Examples of the above are that it would be useful to have a domain that can represent more general algebraic extensions (*AlgebraicField* (page 2539) is only for extensions of *QQ* (page 2529)). Improving the detection of algebraic dependencies is harder but at least common cases like  $\sin(x)$  and  $\cos(x)$  could be handled. When choosing generators it should be possible to recognise that  $\sqrt{x}$  can be the only generator for  $x + \sqrt{x}$ :

```
>>> Poly(x + sqrt(x)) # this could be improved!
Poly(x + (sqrt(x)), x, sqrt(x), domain='ZZ')
>>> Poly(x + sqrt(x), sqrt(x)) # this could be improved!
Poly((sqrt(x)) + x, sqrt(x), domain='ZZ[x]')
```

## Reference docs for the Poly Domains

This page lists the reference documentation for the domains in the polys module. For a general introduction to the polys module it is recommended to read [Basic functionality of the module](#) (page 2342) instead. For an introductory explanation of the what the domain system is and how it is used it is recommended to read [Introducing the Domains of the poly module](#) (page 2477). This page lists the reference docs for the [Domain](#) (page 2504) class and its subclasses (the specific domains such as [ZZ](#)) as well as the classes that represent the domain elements.

## Domains

Here we document the various implemented ground domains (see [Introducing the Domains of the poly module](#) (page 2477) for more of an explanation). There are three types of [Domain](#) (page 2504) subclass: abstract domains, concrete domains, and “implementation domains”. Abstract domains cannot be (usefully) instantiated at all, and just collect together functionality shared by many other domains. Concrete domains are those meant to be instantiated and used in the polynomial manipulation algorithms. In some cases, there are various possible ways to implement the data type the domain provides. For example, depending on what libraries are available on the system, the integers are implemented either using the python built-in integers, or using gmpy. Note that various aliases are created automatically depending on the libraries available. As such e.g. [ZZ](#) always refers to the most efficient implementation of the integer ring available.

## Abstract Domains

**class** `sympy.polys.domains.domain.Domain`

Superclass for all domains in the polys domains system.

See [Introducing the Domains of the poly module](#) (page 2477) for an introductory explanation of the domains system.

The [Domain](#) (page 2504) class is an abstract base class for all of the concrete domain types. There are many different [Domain](#) (page 2504) subclasses each of which has an associated dtype which is a class representing the elements of the domain. The coefficients of a [Poly](#) (page 2378) are elements of a domain which must be a subclass of [Domain](#) (page 2504).

## Examples

The most common example domains are the integers [ZZ](#) (page 2525) and the rationals [QQ](#) (page 2529).

```
>>> from sympy import Poly, symbols, Domain
>>> x, y = symbols('x, y')
>>> p = Poly(x**2 + y)
>>> p
Poly(x**2 + y, x, y, domain='ZZ')
>>> p.domain
ZZ
```

(continues on next page)



(continued from previous page)

```
>>> isinstance(p.domain, Domain)
True
>>> Poly(x**2 + y/2)
Poly(x**2 + 1/2*y, x, y, domain='QQ')
```

The domains can be used directly in which case the domain object e.g. ([ZZ](#) (page 2525) or [QQ](#) (page 2529)) can be used as a constructor for elements of dtype.

```
>>> from sympy import ZZ, QQ
>>> ZZ(2)
2
>>> ZZ.dtype
<class 'int'>
>>> type(ZZ(2))
<class 'int'>
>>> QQ(1, 2)
1/2
>>> type(QQ(1, 2))
<class 'sympy.polys.domains.pythonrational.PythonRational'>
```

The corresponding domain elements can be used with the arithmetic operations  $+$ ,  $-$ ,  $*$ ,  $**$  and depending on the domain some combination of  $/$ ,  $//$ ,  $\%$  might be usable. For example in [ZZ](#) (page 2525) both  $//$  (floor division) and  $\%$  (modulo division) can be used but  $/$  (true division) cannot. Since [QQ](#) (page 2529) is a *Field* (page 2520) its elements can be used with  $/$  but  $//$  and  $\%$  should not be used. Some domains have a [gcd\(\)](#) (page 2514) method.

```
>>> ZZ(2) + ZZ(3)
5
>>> ZZ(5) // ZZ(2)
2
>>> ZZ(5) % ZZ(2)
1
>>> QQ(1, 2) / QQ(2, 3)
3/4
>>> ZZ.gcd(ZZ(4), ZZ(2))
2
>>> QQ.gcd(QQ(2,7), QQ(5,3))
1/21
>>> ZZ.is_Field
False
>>> QQ.is_Field
True
```

There are also many other domains including:

1. [GF\(p\)](#) (page 2522) for finite fields of prime order.
2. [RR](#) (page 2545) for real (floating point) numbers.
3. [CC](#) (page 2546) for complex (floating point) numbers.
4. [QQ<a>](#) (page 2539) for algebraic number fields.
5. [K\[x\]](#) (page 2547) for polynomial rings.

6.  $K(x)$  (page 2548) for rational function fields.

7.  $EX$  (page 2549) for arbitrary expressions.

Each domain is represented by a domain object and also an implementation class (dtype) for the elements of the domain. For example the  $K[x]$  (page 2547) domains are represented by a domain object which is an instance of *PolynomialRing* (page 2547) and the elements are always instances of *PolyElement* (page 2554). The implementation class represents particular types of mathematical expressions in a way that is more efficient than a normal SymPy expression which is of type *Expr* (page 947). The domain methods *from\_sympy()* (page 2513) and *to\_sympy()* (page 2517) are used to convert from *Expr* (page 947) to a domain element and vice versa.

```
>>> from sympy import Symbol, ZZ, Expr
>>> x = Symbol('x')
>>> K = ZZ[x] # polynomial ring domain
>>> K
ZZ[x]
>>> type(K) # class of the domain
<class 'sympy.polys.domains.polynomialring.PolynomialRing'>
>>> K.dtype # class of the elements
<class 'sympy.polys.rings.PolyElement'>
>>> p_expr = x**2 + 1 # Expr
>>> p_expr
x**2 + 1
>>> type(p_expr)
<class 'sympy.core.add.Add'>
>>> isinstance(p_expr, Expr)
True
>>> p_domain = K.from_sympy(p_expr)
>>> p_domain # domain element
x**2 + 1
>>> type(p_domain)
<class 'sympy.polys.rings.PolyElement'>
>>> K.to_sympy(p_domain) == p_expr
True
```

The *convert\_from()* (page 2508) method is used to convert domain elements from one domain to another.

```
>>> from sympy import ZZ, QQ
>>> ez = ZZ(2)
>>> eq = QQ.convert_from(ez, ZZ)
>>> type(ez)
<class 'int'>
>>> type(eq)
<class 'sympy.polys.domains.pythonrational.PythonRational'>
```

Elements from different domains should not be mixed in arithmetic or other operations: they should be converted to a common domain first. The domain method *unify()* (page 2519) is used to find a domain that can represent all the elements of two given domains.

```
>>> from sympy import ZZ, QQ, symbols
>>> x, y = symbols('x, y')
```

(continues on next page)

(continued from previous page)

```
>>> ZZ.unify(QQ)
QQ
>>> ZZ[x].unify(QQ)
QQ[x]
>>> ZZ[x].unify(QQ[y])
QQ[x,y]
```

If a domain is a [Ring](#) (page 2521) then it might have an associated [Field](#) (page 2520) and vice versa. The [get\\_field\(\)](#) (page 2514) and [get\\_ring\(\)](#) (page 2514) methods will find or create the associated domain.

```
>>> from sympy import ZZ, QQ, Symbol
>>> x = Symbol('x')
>>> ZZ.has_assoc_Field
True
>>> ZZ.get_field()
QQ
>>> QQ.has_assoc_Ring
True
>>> QQ.get_ring()
ZZ
>>> K = QQ[x]
>>> K
QQ[x]
>>> K.get_field()
QQ(x)
```

#### See also:

##### [DomainElement](#) (page 2519)

abstract base class for domain elements

##### [construct\\_domain](#) (page 2427)

construct a minimal domain for some expressions

##### [abs\(a\)](#)

Absolute value of a, implies `__abs__`.

##### [add\(a, b\)](#)

Sum of a and b, implies `__add__`.

##### [alg\\_field\\_from\\_poly\(poly, alias=None, root\\_index=-1\)](#)

Convenience method to construct an algebraic extension on a root of a polynomial, chosen by root index.

##### Parameters

**poly** : [Poly](#) (page 2378)

The polynomial whose root generates the extension.

**alias** : str, optional (default=None)

Symbol name for the generator of the extension. E.g. "alpha" or "theta".

**root\_index** : int, optional (default=-1)

Specifies which root of the polynomial is desired. The ordering is as defined by the [ComplexRootOf](#) (page 2431) class. The default of -1 selects the most natural choice in the common cases of quadratic and cyclotomic fields (the square root on the positive real or imaginary axis, resp.  $e^{2\pi i/n}$ ).

### Examples

```
>>> from sympy import QQ, Poly
>>> from sympy.abc import x
>>> f = Poly(x**2 - 2)
>>> K = QQ.alg_field_from_poly(f)
>>> K.ext.minpoly == f
True
>>> g = Poly(8*x**3 - 6*x - 1)
>>> L = QQ.alg_field_from_poly(g, "alpha")
>>> L.ext.minpoly == g
True
>>> L.to_sympy(L([1, 1, 1]))
alpha**2 + alpha + 1
```

**algebraic\_field**(\*extension, alias=None)

Returns an algebraic field, i.e.  $K(\alpha, \dots)$ .

**almosteq**(a, b, tolerance=None)

Check if a and b are almost equal.

**characteristic**()

Return the characteristic of this domain.

**cofactors**(a, b)

Returns GCD and cofactors of a and b.

**convert**(element, base=None)

Convert element to self.dtype.

**convert\_from**(element, base)

Convert element to self.dtype given the base domain.

**cyclotomic\_field**(n, ss=False, alias='zeta', gen=None, root\_index=-1)

Convenience method to construct a cyclotomic field.

#### Parameters

**n** : int

Construct the nth cyclotomic field.

**ss** : boolean, optional (default=False)

If True, append n as a subscript on the alias string.

**alias** : str, optional (default="zeta")

Symbol name for the generator.

**gen** : [Symbol](#) (page 976), optional (default=None)

Desired variable for the cyclotomic polynomial that defines the field.  
If None, a dummy variable will be used.

**root\_index** : int, optional (default=-1)

Specifies which root of the polynomial is desired. The ordering is as defined by the [ComplexRootOf](#) (page 2431) class. The default of -1 selects the root  $e^{2\pi i/n}$ .

## Examples

```
>>> from sympy import QQ, latex
>>> K = QQ.cyclotomic_field(5)
>>> K.to_sympy(K([-1, 1]))
1 - zeta
>>> L = QQ.cyclotomic_field(7, True)
>>> a = L.to_sympy(L([-1, 1]))
>>> print(a)
1 - zeta7
>>> print(latex(a))
1 - \zeta_7
```

**denom**(*a*)

Returns denominator of *a*.

**div**(*a*, *b*)

Quotient and remainder for *a* and *b*. Analogue of `divmod(a, b)`

### Parameters

**a**: domain element

The dividend

**b**: domain element

The divisor

### Returns

(*q*, *r*): tuple of domain elements

The quotient and remainder

### Raises

**ZeroDivisionError**: when the divisor is zero.

## Explanation

This is essentially the same as `divmod(a, b)` except that is more consistent when working over some [Field](#) (page 2520) domains such as [QQ](#) (page 2529). When working over an arbitrary [Domain](#) (page 2504) the [div\(\)](#) (page 2509) method should be used instead of `divmod`.

The key invariant is that if *q*, *r* = `K.div(a, b)` then `a == b*q + r`.

The result of `K.div(a, b)` is the same as the tuple `(K.quo(a, b), K.rem(a, b))` except that if both quotient and remainder are needed then it is more efficient to use [div\(\)](#) (page 2509).

## Examples

We can use `K.div` instead of `divmod` for floor division and remainder.

```
>>> from sympy import ZZ, QQ
>>> ZZ.div(ZZ(5), ZZ(2))
(2, 1)
```

If `K` is a [Field](#) (page 2520) then the division is always exact with a remainder of [zero](#) (page 2519).

```
>>> QQ.div(QQ(5), QQ(2))
(5/2, 0)
```

## Notes

If `gmpy` is installed then the `gmpy.mpq` type will be used as the [dtype](#) (page 2510) for [QQ](#) (page 2529). The `gmpy.mpq` type defines `divmod` in a way that is undesirable so [div\(\)](#) (page 2509) should be used instead of `divmod`.

```
>>> a = QQ(1)
>>> b = QQ(3, 2)
>>> a
mpq(1,1)
>>> b
mpq(3,2)
>>> divmod(a, b)
(mpz(0), mpq(1,1))
>>> QQ.div(a, b)
(mpq(2,3), mpq(0,1))
```

Using `//` or `%` with [QQ](#) (page 2529) will lead to incorrect results so [div\(\)](#) (page 2509) should be used instead.

**See also:**

[quo](#) (page 2516)

Analogue of `a // b`

[rem](#) (page 2517)

Analogue of `a % b`

[exquo](#) (page 2511)

Analogue of `a / b`

**drop(\*symbols)**

Drop generators from this domain.

**dtype: Optional[Type] = None**

The type (class) of the elements of this [Domain](#) (page 2504):

```
>>> from sympy import ZZ, QQ, Symbol
>>> ZZ.dtype
<class 'int'>
```

(continues on next page)

(continued from previous page)

```

>>> z = ZZ(2)
>>> z
2
>>> type(z)
<class 'int'>
>>> type(z) == ZZ.dtype
True

```

Every domain has an associated **dtype** (“datatype”) which is the class of the associated domain elements.

**See also:**

[of\\_type](#) (page 2516)

**evalf**(*a*, *prec=None*, *\*\*options*)

Returns numerical approximation of *a*.

**exquo**(*a*, *b*)

Exact quotient of *a* and *b*. Analogue of  $a / b$ .

**Parameters**

**a: domain element**

The dividend

**b: domain element**

The divisor

**Returns**

q: domain element

The exact quotient

**Raises**

**ExactQuotientFailed:** if exact division is not possible.

**ZeroDivisionError:** when the divisor is zero.

**Explanation**

This is essentially the same as  $a / b$  except that an error will be raised if the division is inexact (if there is any remainder) and the result will always be a domain element. When working in a [Domain](#) (page 2504) that is not a [Field](#) (page 2520) (e.g. [ZZ](#) (page 2525) or [K\[x\]](#) (page 2547)) `exquo` should be used instead of `/`.

The key invariant is that if  $q = K.\text{exquo}(a, b)$  (and `exquo` does not raise an exception) then  $a == b*q$ .

## Examples

We can use `K.exquo` instead of `/` for exact division.

```
>>> from sympy import ZZ
>>> ZZ.exquo(ZZ(4), ZZ(2))
2
>>> ZZ.exquo(ZZ(5), ZZ(2))
Traceback (most recent call last):
...
ExactQuotientFailed: 2 does not divide 5 in ZZ
```

Over a *Field* (page 2520) such as *QQ* (page 2529), division (with nonzero divisor) is always exact so in that case `/` can be used instead of `exquo()` (page 2511).

```
>>> from sympy import QQ
>>> QQ.exquo(QQ(5), QQ(2))
5/2
>>> QQ(5) / QQ(2)
5/2
```

## Notes

Since the default *dtype* (page 2510) for *ZZ* (page 2525) is `int` (or `mpz`) division as `a / b` should not be used as it would give a float.

```
>>> ZZ(4) / ZZ(2)
2.0
>>> ZZ(5) / ZZ(2)
2.5
```

Using `/` with *ZZ* (page 2525) will lead to incorrect results so `exquo()` (page 2511) should be used instead.

**See also:**

***quo* (page 2516)**

Analogue of `a // b`

***rem* (page 2517)**

Analogue of `a % b`

***div* (page 2509)**

Analogue of `divmod(a, b)`

**`frac_field(*symbols, order=LexOrder())`**

Returns a fraction field, i.e.  $K(X)$ .

**`from_AlgebraicField(a, K0)`**

Convert an algebraic number to dtype.

**`from_ComplexField(a, K0)`**

Convert a complex element to dtype.



**from\_ExpressionDomain**(*a*, *K0*)  
Convert a EX object to dtype.

**from\_ExpressionRawDomain**(*a*, *K0*)  
Convert a EX object to dtype.

**from\_FF**(*a*, *K0*)  
Convert ModularInteger(int) to dtype.

**from\_FF\_gmpy**(*a*, *K0*)  
Convert ModularInteger(mpz) to dtype.

**from\_FF\_python**(*a*, *K0*)  
Convert ModularInteger(int) to dtype.

**from\_FractionField**(*a*, *K0*)  
Convert a rational function to dtype.

**from\_GlobalPolynomialRing**(*a*, *K0*)  
Convert a polynomial to dtype.

**from\_MonogenicFiniteExtension**(*a*, *K0*)  
Convert an ExtensionElement to dtype.

**from\_PolynomialRing**(*a*, *K0*)  
Convert a polynomial to dtype.

**from\_QQ\_gmpy**(*a*, *K0*)  
Convert a GMPY mpq object to dtype.

**from\_QQ\_python**(*a*, *K0*)  
Convert a Python Fraction object to dtype.

**from\_RealField**(*a*, *K0*)  
Convert a real element object to dtype.

**from\_ZZ\_gmpy**(*a*, *K0*)  
Convert a GMPY mpz object to dtype.

**from\_ZZ\_python**(*a*, *K0*)  
Convert a Python int object to dtype.

**from\_sympy**(*a*)  
Convert a SymPy expression to an element of this domain.

**Parameters****expr:** ExprA normal SymPy expression of type [Expr](#) (page 947).**Returns**

a: domain element

An element of this [Domain](#) (page 2504).

**Explanation**

See [to\\_sympy\(\)](#) (page 2517) for explanation and examples.

**See also:**

[to\\_sympy](#) (page 2517), [convert\\_from](#) (page 2508)

**gcd**(*a*, *b*)

Returns GCD of *a* and *b*.

**gcdex**(*a*, *b*)

Extended GCD of *a* and *b*.

**get\_exact**()

Returns an exact domain associated with *self*.

**get\_field**()

Returns a field associated with *self*.

**get\_ring**()

Returns a ring associated with *self*.

**half\_gcdex**(*a*, *b*)

Half extended GCD of *a* and *b*.

**has\_assoc\_Field** = **False**

Boolean flag indicating if the domain has an associated [Field](#) (page 2520).

```
>>> from sympy import ZZ
>>> ZZ.has_assoc_Field
True
>>> ZZ.get_field()
QQ
```

**See also:**

[is\\_Field](#) (page 2514), [get\\_field](#) (page 2514)

**has\_assoc\_Ring** = **False**

Boolean flag indicating if the domain has an associated [Ring](#) (page 2521).

```
>>> from sympy import QQ
>>> QQ.has_assoc_Ring
True
>>> QQ.get_ring()
ZZ
```

**See also:**

[is\\_Field](#) (page 2514), [get\\_ring](#) (page 2514)

**inject**(\**symbols*)

Inject generators into this domain.

**invert**(*a*, *b*)

Returns inversion of *a* mod *b*, implies something.

**is\_Field = False**

Boolean flag indicating if the domain is a *Field* (page 2520).

```
>>> from sympy import ZZ, QQ
>>> ZZ.is_Field
False
>>> QQ.is_Field
True
```

**See also:**

*is\_PID* (page 2515), *is\_Ring* (page 2515), *get\_field* (page 2514), *has\_assoc\_Field* (page 2514)

**is\_PID = False**

Boolean flag indicating if the domain is a *principal ideal domain*.

```
>>> from sympy import ZZ
>>> ZZ.has_assoc_Field
True
>>> ZZ.get_field()
QQ
```

**See also:**

*is\_Field* (page 2514), *get\_field* (page 2514)

**is\_Ring = False**

Boolean flag indicating if the domain is a *Ring* (page 2521).

```
>>> from sympy import ZZ
>>> ZZ.is_Ring
True
```

Basically every *Domain* (page 2504) represents a ring so this flag is not that useful.

**See also:**

*is\_PID* (page 2515), *is\_Field* (page 2514), *get\_ring* (page 2514), *has\_assoc\_Ring* (page 2514)

**is\_negative(a)**

Returns True if a is negative.

**is\_nonnegative(a)**

Returns True if a is non-negative.

**is\_nonpositive(a)**

Returns True if a is non-positive.

**is\_one(a)**

Returns True if a is one.

**is\_positive(a)**

Returns True if a is positive.

**is\_zero(a)**

Returns True if a is zero.

**lcm**(*a*, *b*)

Returns LCM of *a* and *b*.

**log**(*a*, *b*)

Returns *b*-base logarithm of *a*.

**map**(*seq*)

Reversively apply *self* to all elements of *seq*.

**mul**(*a*, *b*)

Product of *a* and *b*, implies `__mul__`.

**n**(*a*, *prec*=None, *\*\*options*)

Returns numerical approximation of *a*.

**neg**(*a*)

Returns *a* negated, implies `__neg__`.

**numer**(*a*)

Returns numerator of *a*.

**of\_type**(*element*)

Check if *a* is of type *dtype*.

**old\_frac\_field**(*\*symbols*, *\*\*kwargs*)

Returns a fraction field, i.e.  $K(X)$ .

**old\_poly\_ring**(*\*symbols*, *\*\*kwargs*)

Returns a polynomial ring, i.e.  $K[X]$ .

**one**: `Optional[Any] = None`

The one element of the [Domain](#) (page 2504):

```
>>> from sympy import QQ
>>> QQ.one
1
>>> QQ.of_type(QQ.one)
True
```

**See also:**

[of\\_type](#) (page 2516), [zero](#) (page 2519)

**poly\_ring**(*\*symbols*, *order*=`LexOrder()`)

Returns a polynomial ring, i.e.  $K[X]$ .

**pos**(*a*)

Returns *a* positive, implies `__pos__`.

**pow**(*a*, *b*)

Raise *a* to power *b*, implies `__pow__`.

**quo**(*a*, *b*)

Quotient of *a* and *b*. Analogue of *a* // *b*.

*K*.quo(*a*, *b*) is equivalent to *K*.div(*a*, *b*)[0]. See [div\(\)](#) (page 2509) for more explanation.

**See also:**

**rem (page 2517)**Analogue of  $a \% b$ **div (page 2509)**Analogue of `divmod(a, b)`**exquo (page 2511)**Analogue of  $a / b$ **rem(*a*, *b*)**Modulo division of *a* and *b*. Analogue of  $a \% b$ .

$K.\text{rem}(a, b)$  is equivalent to  $K.\text{div}(a, b)[1]$ . See [div\(\)](#) (page 2509) for more explanation.

**See also:****quo (page 2516)**Analogue of  $a // b$ **div (page 2509)**Analogue of `divmod(a, b)`**exquo (page 2511)**Analogue of  $a / b$ **revert(*a*)**Returns  $a^{**}(-1)$  if possible.**sqrt(*a*)**Returns square root of *a*.**sub(*a*, *b*)**Difference of *a* and *b*, implies `__sub__`.**to\_sympy(*a*)**Convert domain element *a* to a SymPy expression (`Expr`).**Parameters****a: domain element**An element of this [Domain](#) (page 2504).**Returns**expr: `Expr`A normal SymPy expression of type [Expr](#) (page 947).**Explanation**

Convert a [Domain](#) (page 2504) element *a* to [Expr](#) (page 947). Most public SymPy functions work with objects of type [Expr](#) (page 947). The elements of a [Domain](#) (page 2504) have a different internal representation. It is not possible to mix domain elements with [Expr](#) (page 947) so each domain has [to\\_sympy\(\)](#) (page 2517) and [from\\_sympy\(\)](#) (page 2513) methods to convert its domain elements to and from [Expr](#) (page 947).

## Examples

Construct an element of the [QQ](#) (page 2529) domain and then convert it to [Expr](#) (page 947).

```
>>> from sympy import QQ, Expr
>>> q_domain = QQ(2)
>>> q_domain
2
>>> q_expr = QQ.to_sympy(q_domain)
>>> q_expr
2
```

Although the printed forms look similar these objects are not of the same type.

```
>>> isinstance(q_domain, Expr)
False
>>> isinstance(q_expr, Expr)
True
```

Construct an element of [K\[x\]](#) (page 2547) and convert to [Expr](#) (page 947).

```
>>> from sympy import Symbol
>>> x = Symbol('x')
>>> K = QQ[x]
>>> x_domain = K.gens[0] # generator x as a domain element
>>> p_domain = x_domain**2/3 + 1
>>> p_domain
1/3*x**2 + 1
>>> p_expr = K.to_sympy(p_domain)
>>> p_expr
x**2/3 + 1
```

The [from\\_sympy\(\)](#) (page 2513) method is used for the opposite conversion from a normal SymPy expression to a domain element.

```
>>> p_domain == p_expr
False
>>> K.from_sympy(p_expr) == p_domain
True
>>> K.to_sympy(p_domain) == p_expr
True
>>> K.from_sympy(K.to_sympy(p_domain)) == p_domain
True
>>> K.to_sympy(K.from_sympy(p_expr)) == p_expr
True
```

The [from\\_sympy\(\)](#) (page 2513) method makes it easier to construct domain elements interactively.

```
>>> from sympy import Symbol
>>> x = Symbol('x')
>>> K = QQ[x]
```

(continues on next page)

(continued from previous page)

```
>>> K.from_sympy(x**2/3 + 1)
1/3*x**2 + 1
```

**See also:**

[from\\_sympy](#) (page 2513), [convert\\_from](#) (page 2508)

**property tp**

Alias for [dtype](#) (page 2510)

**unify(K1, symbols=None)**

Construct a minimal domain that contains elements of K0 and K1.

Known domains (from smallest to largest):

- GF(p)
- ZZ
- QQ
- RR(prec, tol)
- CC(prec, tol)
- ALG(a, b, c)
- K[x, y, z]
- K(x, y, z)
- EX

**zero: Optional[Any] = None**

The zero element of the [Domain](#) (page 2504):

```
>>> from sympy import QQ
>>> QQ.zero
0
>>> QQ.of_type(QQ.zero)
True
```

**See also:**

[of\\_type](#) (page 2516), [one](#) (page 2516)

**class sympy.polys.domains.domainelement.DomainElement**

Represents an element of a domain.

Mix in this trait into a class whose instances should be recognized as elements of a domain. Method `parent()` gives that domain.

**parent()**

Get the domain associated with `self`

## Examples

```
>>> from sympy import ZZ, symbols
>>> x, y = symbols('x, y')
>>> K = ZZ[x,y]
>>> p = K(x)**2 + K(y)**2
>>> p
x**2 + y**2
>>> p.parent()
ZZ[x,y]
```

## Notes

This is used by [convert\(\)](#) (page 2508) to identify the domain associated with a domain element.

**class** sympy.polys.domains.field.**Field**

Represents a field domain.

**div**(a, b)

Division of a and b, implies `__truediv__`.

**exquo**(a, b)

Exact quotient of a and b, implies `__truediv__`.

**gcd**(a, b)

Returns GCD of a and b.

This definition of GCD over fields allows to clear denominators in *primitive()*.

## Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy import S, gcd, primitive
>>> from sympy.abc import x
```

```
>>> QQ.gcd(QQ(2, 3), QQ(4, 9))
2/9
>>> gcd(S(2)/3, S(4)/9)
2/9
>>> primitive(2*x/3 + S(4)/9)
(2/9, 3*x + 2)
```

**get\_field()**

Returns a field associated with self.

**get\_ring()**

Returns a ring associated with self.

**is\_unit**(a)

Return true if a is a invertible



**lcm**(*a*, *b*)

Returns LCM of *a* and *b*.

```
>>> from sympy.polys.domains import QQ
>>> from sympy import S, lcm
```

```
>>> QQ.lcm(QQ(2, 3), QQ(4, 9))
4/3
>>> lcm(S(2)/3, S(4)/9)
4/3
```

**quo**(*a*, *b*)

Quotient of *a* and *b*, implies `__truediv__`.

**rem**(*a*, *b*)

Remainder of *a* and *b*, implies nothing.

**revert**(*a*)

Returns  $a^{**}(-1)$  if possible.

**class** `sympy.polys.domains.ring.Ring`

Represents a ring domain.

**denom**(*a*)

Returns denominator of *a*.

**div**(*a*, *b*)

Division of *a* and *b*, implies `__divmod__`.

**exquo**(*a*, *b*)

Exact quotient of *a* and *b*, implies `__floordiv__`.

**free\_module**(*rank*)

Generate a free module of rank *rank* over self.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(2)
QQ[x]**2
```

**get\_ring**()

Returns a ring associated with self.

**ideal**(\**gens*)

Generate an ideal of self.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).ideal(x**2)
<x**2>
```

**invert**(*a*, *b*)

Returns inversion of *a* mod *b*.

**numer**(*a*)

Returns numerator of *a*.

**quo**(*a*, *b*)

Quotient of *a* and *b*, implies `__floordiv__`.

**quotient\_ring**(*e*)

Form a quotient ring of *self*.

Here *e* can be an ideal or an iterable.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).quotient_ring(QQ.old_poly_ring(x).ideal(x**2))
QQ[x]/<x**2>
>>> QQ.old_poly_ring(x).quotient_ring([x**2])
QQ[x]/<x**2>
```

The division operator has been overloaded for this:

```
>>> QQ.old_poly_ring(x)/[x**2]
QQ[x]/<x**2>
```

**rem**(*a*, *b*)

Remainder of *a* and *b*, implies `__mod__`.

**revert**(*a*)

Returns *a*\*\*(-1) if possible.

**class** sympy.polys.domains.simplesdomain.**SimpleDomain**

Base class for simple domains, e.g. ZZ, QQ.

**inject**(\**gens*)

Inject generators into this domain.

**class** sympy.polys.domains.compositedomain.**CompositeDomain**

Base class for composite domains, e.g. ZZ[x], ZZ(X).

**drop**(\**symbols*)

Drop generators from this domain.

**inject**(\**symbols*)

Inject generators into this domain.

## GF(*p*)

**class** sympy.polys.domains.**FiniteField**(*mod*, *symmetric*=True)

Finite field of prime order [GF\(\*p\*\)](#) (page 2522)

A [GF\(\*p\*\)](#) (page 2522) domain represents a [finite field](#)  $\mathbb{F}_p$  of prime order as [Domain](#) (page 2504) in the domain system (see [Introducing the Domains of the poly module](#) (page 2477)).

A [Poly](#) (page 2378) created from an expression with integer coefficients will have the domain [ZZ](#) (page 2525). However, if the `modulus=p` option is given then the domain will be a finite field instead.

```
>>> from sympy import Poly, Symbol
>>> x = Symbol('x')
>>> p = Poly(x**2 + 1)
>>> p
Poly(x**2 + 1, x, domain='ZZ')
>>> p.domain
ZZ
>>> p2 = Poly(x**2 + 1, modulus=2)
>>> p2
Poly(x**2 + 1, x, modulus=2)
>>> p2.domain
GF(2)
```

It is possible to factorise a polynomial over  $GF(p)$  (page 2522) using the modulus argument to `factor()` (page 2373) or by specifying the domain explicitly. The domain can also be given as a string.

```
>>> from sympy import factor, GF
>>> factor(x**2 + 1)
x**2 + 1
>>> factor(x**2 + 1, modulus=2)
(x + 1)**2
>>> factor(x**2 + 1, domain=GF(2))
(x + 1)**2
>>> factor(x**2 + 1, domain='GF(2)')
(x + 1)**2
```

It is also possible to use  $GF(p)$  (page 2522) with the `cancel()` (page 2376) and `gcd()` (page 2368) functions.

```
>>> from sympy import cancel, gcd
>>> cancel((x**2 + 1)/(x + 1))
(x**2 + 1)/(x + 1)
>>> cancel((x**2 + 1)/(x + 1), domain=GF(2))
x + 1
>>> gcd(x**2 + 1, x + 1)
1
>>> gcd(x**2 + 1, x + 1, domain=GF(2))
x + 1
```

When using the domain directly  $GF(p)$  (page 2522) can be used as a constructor to create instances which then support the operations `+`, `-`, `*`, `**`, `/`

```
>>> from sympy import GF
>>> K = GF(5)
>>> K
GF(5)
>>> x = K(3)
>>> y = K(2)
>>> x
3 mod 5
>>> y
2 mod 5
```

(continues on next page)

(continued from previous page)

```
>>> x * y
1 mod 5
>>> x / y
4 mod 5
```

## Notes

It is also possible to create a  $GF(p)$  (page 2522) domain of **non-prime** order but the resulting ring is **not** a field: it is just the ring of the integers modulo  $n$ .

```
>>> K = GF(9)
>>> z = K(3)
>>> z
3 mod 9
>>> z**2
0 mod 9
```

It would be good to have a proper implementation of prime power fields ( $GF(p^n)$ ) but these are not yet implemented in SymPy.

### `characteristic()`

Return the characteristic of this domain.

### `from_FF(a, K0=None)`

Convert `ModularInteger(int)` to dtype.

### `from_FF_gmpy(a, K0=None)`

Convert `ModularInteger(mpz)` to dtype.

### `from_FF_python(a, K0=None)`

Convert `ModularInteger(int)` to dtype.

### `from_QQ(a, K0=None)`

Convert Python's `Fraction` to dtype.

### `from_QQ_gmpy(a, K0=None)`

Convert GMPY's `mpq` to dtype.

### `from_QQ_python(a, K0=None)`

Convert Python's `Fraction` to dtype.

### `from_RealField(a, K0)`

Convert `mpmath's mpf` to dtype.

### `from_ZZ(a, K0=None)`

Convert Python's `int` to dtype.

### `from_ZZ_gmpy(a, K0=None)`

Convert GMPY's `mpz` to dtype.

### `from_ZZ_python(a, K0=None)`

Convert Python's `int` to dtype.

**from\_sympy(a)**

Convert SymPy's Integer to SymPy's Integer.

**get\_field()**

Returns a field associated with self.

**to\_sympy(a)**

Convert a to a SymPy object.

**class sympy.polys.domains.PythonFiniteField(mod, symmetric=True)**

Finite field based on Python's integers.

**class sympy.polys.domains.GMPYFiniteField(mod, symmetric=True)**

Finite field based on GMPY integers.

## ZZ

The [ZZ](#) (page 2525) domain represents the [integers  \$\mathbb{Z}\$](#)  as a [Domain](#) (page 2504) in the domain system (see [Introducing the Domains of the poly module](#) (page 2477)).

By default a [Poly](#) (page 2378) created from an expression with integer coefficients will have the domain [ZZ](#) (page 2525):

```
>>> from sympy import Poly, Symbol
>>> x = Symbol('x')
>>> p = Poly(x**2 + 1)
>>> p
Poly(x**2 + 1, x, domain='ZZ')
>>> p.domain
ZZ
```

The corresponding [field of fractions](#) is the domain of the rationals [QQ](#) (page 2529). Conversely [ZZ](#) (page 2525) is the [ring of integers](#) of [QQ](#) (page 2529):

```
>>> from sympy import ZZ, QQ
>>> ZZ.get_field()
QQ
>>> QQ.get_ring()
ZZ
```

When using the domain directly [ZZ](#) (page 2525) can be used as a constructor to create instances which then support the operations `+`, `-`, `*`, `**`, `//`, `%` (true division / should not be used with [ZZ](#) (page 2525) - see the [exquo\(\)](#) (page 2511) domain method):

```
>>> x = ZZ(5)
>>> y = ZZ(2)
>>> x // y # floor division
2
>>> x % y # modulo division (remainder)
1
```

The [gcd\(\)](#) (page 2514) method can be used to compute the [gcd](#) of any two elements:

```
>>> ZZ.gcd(ZZ(10), ZZ(2))
2
```

There are two implementations of [ZZ](#) (page 2525) in SymPy. If `gmpy` or `gmpy2` is installed then [ZZ](#) (page 2525) will be implemented by [GMPYIntegerRing](#) (page 2528) and the elements will be instances of the `gmpy.mpz` type. Otherwise if `gmpy` and `gmpy2` are not installed then [ZZ](#) (page 2525) will be implemented by [PythonIntegerRing](#) (page 2528) which uses Python's standard builtin `int` type. With larger integers `gmpy` can be more efficient so it is preferred when available.

**class** `sympy.polys.domains.IntegerRing`

The domain `ZZ` representing the integers  $\mathbb{Z}$ .

The [IntegerRing](#) (page 2526) class represents the ring of integers as a [Domain](#) (page 2504) in the domain system. [IntegerRing](#) (page 2526) is a super class of [PythonIntegerRing](#) (page 2528) and [GMPYIntegerRing](#) (page 2528) one of which will be the implementation for [ZZ](#) (page 2525) depending on whether or not `gmpy` or `gmpy2` is installed.

**See also:**

[Domain](#) (page 2504)

**algebraic\_field(\*extension, alias=None)**

Returns an algebraic field, i.e.  $\mathbb{Q}(\alpha, \dots)$ .

**Parameters**

**\*extension** : One or more [Expr](#) (page 947).

Generators of the extension. These should be expressions that are algebraic over  $\mathbb{Q}$ .

**alias** : str, [Symbol](#) (page 976), None, optional (default=None)

If provided, this will be used as the alias symbol for the primitive element of the returned [AlgebraicField](#) (page 2539).

**Returns**

[AlgebraicField](#) (page 2539)

A [Domain](#) (page 2504) representing the algebraic field extension.

**Examples**

```
>>> from sympy import ZZ, sqrt
>>> ZZ.algebraic_field(sqrt(2))
QQ<sqrt(2)>
```

**factorial(a)**

Compute factorial of `a`.

**from\_AlgebraicField(a, K0)**

Convert a [ANP](#) (page 2568) object to [ZZ](#) (page 2525).

See [convert\(\)](#) (page 2508).

**from\_FF(a, K0)**

Convert `ModularInteger(int)` to GMPY's `mpz`.

**from\_FF\_gmpy(a, K0)**

Convert `ModularInteger(mpz)` to GMPY's `mpz`.

**from\_FF\_python**(*a*, *K0*)

Convert ModularInteger(int) to GMPY's mpz.

**from\_QQ**(*a*, *K0*)

Convert Python's Fraction to GMPY's mpz.

**from\_QQ\_gmpy**(*a*, *K0*)

Convert GMPY mpq to GMPY's mpz.

**from\_QQ\_python**(*a*, *K0*)

Convert Python's Fraction to GMPY's mpz.

**from\_RealField**(*a*, *K0*)

Convert mpmath's mpf to GMPY's mpz.

**from\_ZZ**(*a*, *K0*)

Convert Python's int to GMPY's mpz.

**from\_ZZ\_gmpy**(*a*, *K0*)

Convert GMPY's mpz to GMPY's mpz.

**from\_ZZ\_python**(*a*, *K0*)

Convert Python's int to GMPY's mpz.

**from\_sympy**(*a*)

Convert SymPy's Integer to dtype.

**gcd**(*a*, *b*)

Compute GCD of *a* and *b*.

**gcdex**(*a*, *b*)

Compute extended GCD of *a* and *b*.

**get\_field**()

Return the associated field of fractions [QQ](#) (page 2529)

#### Returns

[QQ](#) (page 2529):

The associated field of fractions [QQ](#) (page 2529), a [Domain](#) (page 2504) representing the rational numbers  $\mathbb{Q}$ .

#### Examples

```
>>> from sympy import ZZ
>>> ZZ.get_field()
QQ
```

**lcm**(*a*, *b*)

Compute LCM of *a* and *b*.

**log**(*a*, *b*)

logarithm of *a* to the base *b*

#### Parameters

**a:** number

**b:** number

**Returns**

*lfloor*  $\log(a, b)$   
*rfloor*:

Floor of the logarithm of  $a$  to the base  $b$

**Examples**

```
>>> from sympy import ZZ
>>> ZZ.log(ZZ(8), ZZ(2))
3
>>> ZZ.log(ZZ(9), ZZ(2))
3
```

**Notes**

This function uses `math.log` which is based on float so it will fail for large integer arguments.

**`sqrt(a)`**

Compute square root of  $a$ .

**`to_sympy(a)`**

Convert  $a$  to a SymPy object.

**`class sympy.polys.domains.PythonIntegerRing`**

Integer ring based on Python's `int` type.

This will be used as `ZZ` (page 2525) if `gmpy` and `gmpy2` are not installed. Elements are instances of the standard Python `int` type.

**`class sympy.polys.domains.GMPYIntegerRing`**

Integer ring based on GMPY's `mpz` type.

This will be the implementation of `ZZ` (page 2525) if `gmpy` or `gmpy2` is installed. Elements will be of type `gmpy.mpz`.

**`factorial(a)`**

Compute factorial of  $a$ .

**`from_FF_gmpy(a, K0)`**

Convert `ModularInteger(mpz)` to GMPY's `mpz`.

**`from_FF_python(a, K0)`**

Convert `ModularInteger(int)` to GMPY's `mpz`.

**`from_QQ(a, K0)`**

Convert Python's `Fraction` to GMPY's `mpz`.

**`from_QQ_gmpy(a, K0)`**

Convert GMPY `mpq` to GMPY's `mpz`.

**`from_QQ_python(a, K0)`**

Convert Python's `Fraction` to GMPY's `mpz`.



**from\_RealField**(*a*, *K0*)  
 Convert mpmath's mpf to GMPY's mpz.

**from\_ZZ\_gmpy**(*a*, *K0*)  
 Convert GMPY's mpz to GMPY's mpz.

**from\_ZZ\_python**(*a*, *K0*)  
 Convert Python's int to GMPY's mpz.

**from\_sympy**(*a*)  
 Convert SymPy's Integer to dtype.

**gcd**(*a*, *b*)  
 Compute GCD of *a* and *b*.

**gcdex**(*a*, *b*)  
 Compute extended GCD of *a* and *b*.

**lcm**(*a*, *b*)  
 Compute LCM of *a* and *b*.

**sqrt**(*a*)  
 Compute square root of *a*.

**to\_sympy**(*a*)  
 Convert *a* to a SymPy object.

## QQ

The [QQ](#) (page 2529) domain represents the [rationals](#)  $\mathbb{Q}$  as a [Domain](#) (page 2504) in the domain system (see [Introducing the Domains of the poly module](#) (page 2477)).

By default a [Poly](#) (page 2378) created from an expression with rational coefficients will have the domain [QQ](#) (page 2529):

```
>>> from sympy import Poly, Symbol
>>> x = Symbol('x')
>>> p = Poly(x**2 + x/2)
>>> p
Poly(x**2 + 1/2*x, x, domain='QQ')
>>> p.domain
QQ
```

The corresponding [ring of integers](#) is the [Domain](#) (page 2504) of the integers [ZZ](#) (page 2525). Conversely [QQ](#) (page 2529) is the [field of fractions](#) of [ZZ](#) (page 2525):

```
>>> from sympy import ZZ, QQ
>>> QQ.get_ring()
ZZ
>>> ZZ.get_field()
QQ
```

When using the domain directly [QQ](#) (page 2529) can be used as a constructor to create instances which then support the operations `+`, `-`, `*`, `**`, `/` (true division `/` is always possible for nonzero divisors in [QQ](#) (page 2529)):

```
>>> x = QQ(5)
>>> y = QQ(2)
>>> x / y # true division
5/2
```

There are two implementations of [QQ](#) (page 2529) in SymPy. If `gmpy` or `gmpy2` is installed then [QQ](#) (page 2529) will be implemented by [GMPYRationalField](#) (page 2531) and the elements will be instances of the `gmpy.mpq` type. Otherwise if `gmpy` and `gmpy2` are not installed then [QQ](#) (page 2529) will be implemented by [PythonRationalField](#) (page 2531) which is a pure Python class as part of `sympy`. The `gmpy` implementation is preferred because it is significantly faster.

**class** `sympy.polys.domains.RationalField`

Abstract base class for the domain [QQ](#) (page 2529).

The [RationalField](#) (page 2530) class represents the field of rational numbers  $\mathbb{Q}$  as a [Domain](#) (page 2504) in the domain system. [RationalField](#) (page 2530) is a superclass of [PythonRationalField](#) (page 2531) and [GMPYRationalField](#) (page 2531) one of which will be the implementation for [QQ](#) (page 2529) depending on whether either of `gmpy` or `gmpy2` is installed or not.

**See also:**

[Domain](#) (page 2504)

**algebraic\_field**(\*extension, alias=None)

Returns an algebraic field, i.e.  $\mathbb{Q}(\alpha, \dots)$ .

**Parameters**

**\*extension** : One or more [Expr](#) (page 947)

Generators of the extension. These should be expressions that are algebraic over  $\mathbb{Q}$ .

**alias** : str, [Symbol](#) (page 976), None, optional (default=None)

If provided, this will be used as the alias symbol for the primitive element of the returned [AlgebraicField](#) (page 2539).

**Returns**

[AlgebraicField](#) (page 2539)

A [Domain](#) (page 2504) representing the algebraic field extension.

### Examples

```
>>> from sympy import QQ, sqrt
>>> QQ.algebraic_field(sqrt(2))
QQ<sqrt(2)>
```

**denom**(a)

Returns denominator of a.

**div**(a, b)

Division of a and b, implies `__truediv__`.

**exquo**(*a*, *b*)

Exact quotient of *a* and *b*, implies `__truediv__`.

**from\_AlgebraicField**(*a*, *K0*)

Convert a [ANP](#) (page 2568) object to [QQ](#) (page 2529).

See [convert\(\)](#) (page 2508)

**from\_GaussianRationalField**(*a*, *K0*)

Convert a GaussianElement object to dtype.

**from\_QQ**(*a*, *K0*)

Convert a Python Fraction object to dtype.

**from\_QQ\_gmpy**(*a*, *K0*)

Convert a GMPY mpq object to dtype.

**from\_QQ\_python**(*a*, *K0*)

Convert a Python Fraction object to dtype.

**from\_RealField**(*a*, *K0*)

Convert a mpmath mpf object to dtype.

**from\_ZZ**(*a*, *K0*)

Convert a Python int object to dtype.

**from\_ZZ\_gmpy**(*a*, *K0*)

Convert a GMPY mpz object to dtype.

**from\_ZZ\_python**(*a*, *K0*)

Convert a Python int object to dtype.

**from\_sympy**(*a*)

Convert SymPy's Integer to dtype.

**get\_ring**()

Returns ring associated with `self`.

**numer**(*a*)

Returns numerator of *a*.

**quo**(*a*, *b*)

Quotient of *a* and *b*, implies `__truediv__`.

**rem**(*a*, *b*)

Remainder of *a* and *b*, implies nothing.

**to\_sympy**(*a*)

Convert *a* to a SymPy object.

**class** `sympy.polys.domains.PythonRationalField`

Rational field based on [MPQ](#) (page 2533).

This will be used as [QQ](#) (page 2529) if gmpy and gmpy2 are not installed. Elements are instances of [MPQ](#) (page 2533).

**class** sympy.polys.domains.**GMPYRationalField**

Rational field based on GMPY's mpq type.

This will be the implementation of [QQ](#) (page 2529) if gmpy or gmpy2 is installed. Elements will be of type `gmpy.mpq`.

**denom**(*a*)

Returns denominator of *a*.

**div**(*a*, *b*)

Division of *a* and *b*, implies `__truediv__`.

**exquo**(*a*, *b*)

Exact quotient of *a* and *b*, implies `__truediv__`.

**factorial**(*a*)

Returns factorial of *a*.

**from\_GaussianRationalField**(*a*, *K0*)

Convert a GaussianElement object to dtype.

**from\_QQ\_gmpy**(*a*, *K0*)

Convert a GMPY mpq object to dtype.

**from\_QQ\_python**(*a*, *K0*)

Convert a Python Fraction object to dtype.

**from\_RealField**(*a*, *K0*)

Convert a mpmath mpf object to dtype.

**from\_ZZ\_gmpy**(*a*, *K0*)

Convert a GMPY mpz object to dtype.

**from\_ZZ\_python**(*a*, *K0*)

Convert a Python int object to dtype.

**from\_sympy**(*a*)

Convert SymPy's Integer to dtype.

**get\_ring**()

Returns ring associated with `self`.

**numer**(*a*)

Returns numerator of *a*.

**quo**(*a*, *b*)

Quotient of *a* and *b*, implies `__truediv__`.

**rem**(*a*, *b*)

Remainder of *a* and *b*, implies nothing.

**to\_sympy**(*a*)

Convert *a* to a SymPy object.

**class** sympy.external.pythonmpq.**PythonMPQ**(*numerator*, *denominator*=None)

Rational number implementation that is intended to be compatible with gmpy2's mpq.

Also slightly faster than `fractions.Fraction`.

PythonMPQ should be treated as immutable although no effort is made to prevent mutation (since that might slow down calculations).

## MPQ

The MPQ type is either [PythonMPQ](#) (page 2532) or otherwise the mpq type from gmpy2.

## Gaussian domains

The Gaussian domains [ZZ\\_I](#) (page 2534) and [QQ\\_I](#) (page 2536) share common superclasses [GaussianElement](#) (page 2533) for the domain elements and [GaussianDomain](#) (page 2533) for the domains themselves.

**class** sympy.polys.domains.gaussiandomains.**GaussianDomain**

Base class for Gaussian domains.

**from\_AlgebraicField**(*a*, *K0*)

Convert an element from ZZ<I> or QQ<I> to self.dtype.

**from\_QQ**(*a*, *K0*)

Convert a GMPY mpq to self.dtype.

**from\_QQ\_gmpy**(*a*, *K0*)

Convert a GMPY mpq to self.dtype.

**from\_QQ\_python**(*a*, *K0*)

Convert a QQ\_python element to self.dtype.

**from\_ZZ**(*a*, *K0*)

Convert a ZZ\_python element to self.dtype.

**from\_ZZ\_gmpy**(*a*, *K0*)

Convert a GMPY mpz to self.dtype.

**from\_ZZ\_python**(*a*, *K0*)

Convert a ZZ\_python element to self.dtype.

**from\_sympy**(*a*)

Convert a SymPy object to self.dtype.

**inject**(\*gens)

Inject generators into this domain.

**is\_negative**(*element*)

Returns False for any GaussianElement.

**is\_nonnegative**(*element*)

Returns False for any GaussianElement.

**is\_nonpositive**(*element*)

Returns False for any GaussianElement.

**is\_positive**(*element*)

Returns False for any GaussianElement.

**to\_sympy**(*a*)

Convert a to a SymPy object.

**class** sympy.polys.domains.gaussiandomains.**GaussianElement**(x, y=0)

Base class for elements of Gaussian type domains.

**classmethod** new(x, y)

Create a new GaussianElement of the same domain.

**parent**()

The domain that this is an element of (ZZ\_I or QQ\_I)

**quadrant**()

Return quadrant index 0-3.

0 is included in quadrant 0.

## ZZ\_I

**class** sympy.polys.domains.gaussiandomains.**GaussianIntegerRing**

Ring of Gaussian integers ZZ\_I

The [ZZ\\_I](#) (page 2534) domain represents the [Gaussian integers](#)  $\mathbb{Z}[i]$  as a [Domain](#) (page 2504) in the domain system (see [Introducing the Domains of the poly module](#) (page 2477)).

By default a [Poly](#) (page 2378) created from an expression with coefficients that are combinations of integers and  $i$  ( $\sqrt{-1}$ ) will have the domain [ZZ\\_I](#) (page 2534).

```
>>> from sympy import Poly, Symbol, I
>>> x = Symbol('x')
>>> p = Poly(x**2 + I)
>>> p
Poly(x**2 + I, x, domain='ZZ_I')
>>> p.domain
ZZ_I
```

The [ZZ\\_I](#) (page 2534) domain can be used to factorise polynomials that are reducible over the Gaussian integers.

```
>>> from sympy import factor
>>> factor(x**2 + 1)
x**2 + 1
>>> factor(x**2 + 1, domain='ZZ_I')
(x - I)*(x + I)
```

The corresponding [field of fractions](#) is the domain of the Gaussian rationals [QQ\\_I](#) (page 2536). Conversely [ZZ\\_I](#) (page 2534) is the [ring of integers](#) of [QQ\\_I](#) (page 2536).

```
>>> from sympy import ZZ_I, QQ_I
>>> ZZ_I.get_field()
QQ_I
>>> QQ_I.get_ring()
ZZ_I
```

When using the domain directly [ZZ\\_I](#) (page 2534) can be used as a constructor.

```
>>> ZZ_I(3, 4)
(3 + 4*I)
>>> ZZ_I(5)
(5 + 0*I)
```

The domain elements of `ZZ_I` (page 2534) are instances of `GaussianInteger` (page 2536) which support the rings operations `+`, `-`, `*`, `**`.

```
>>> z1 = ZZ_I(5, 1)
>>> z2 = ZZ_I(2, 3)
>>> z1
(5 + 1*I)
>>> z2
(2 + 3*I)
>>> z1 + z2
(7 + 4*I)
>>> z1 * z2
(7 + 17*I)
>>> z1 ** 2
(24 + 10*I)
```

Both floor (`//`) and modulo (`%`) division work with `GaussianInteger` (page 2536) (see the `div()` (page 2509) method).

```
>>> z3, z4 = ZZ_I(5), ZZ_I(1, 3)
>>> z3 // z4 # floor division
(1 + -1*I)
>>> z3 % z4 # modulo division (remainder)
(1 + -2*I)
>>> (z3//z4)*z4 + z3%z4 == z3
True
```

True division (`/`) in `ZZ_I` (page 2534) gives an element of `QQ_I` (page 2536). The `exquo()` (page 2511) method can be used to divide in `ZZ_I` (page 2534) when exact division is possible.

```
>>> z1 / z2
(1 + -1*I)
>>> ZZ_I.exquo(z1, z2)
(1 + -1*I)
>>> z3 / z4
(1/2 + -3/2*I)
>>> ZZ_I.exquo(z3, z4)
Traceback (most recent call last):
...
ExactQuotientFailed: (1 + 3*I) does not divide (5 + 0*I) in ZZ_I
```

The `gcd()` (page 2514) method can be used to compute the `gcd` of any two elements.

```
>>> ZZ_I.gcd(ZZ_I(10), ZZ_I(2))
(2 + 0*I)
>>> ZZ_I.gcd(ZZ_I(5), ZZ_I(2, 1))
(2 + 1*I)
```

**dtype**

alias of [GaussianInteger](#) (page 2536)

**from\_GaussianIntegerRing(a, K0)**

Convert a ZZ\_I element to ZZ\_I.

**from\_GaussianRationalField(a, K0)**

Convert a QQ\_I element to ZZ\_I.

**gcd(a, b)**

Greatest common divisor of a and b over ZZ\_I.

**get\_field()**

Returns a field associated with self.

**get\_ring()**

Returns a ring associated with self.

**lcm(a, b)**

Least common multiple of a and b over ZZ\_I.

**normalize(d, \*args)**

Return first quadrant element associated with d.

Also multiply the other arguments by the same power of i.

**class** sympy.polys.domains.gaussiandomains.**GaussianInteger**(x, y=0)

Gaussian integer: domain element for [ZZ\\_I](#) (page 2534)

```
>>> from sympy import ZZ_I
>>> z = ZZ_I(2, 3)
>>> z
(2 + 3*I)
>>> type(z)
<class 'sympy.polys.domains.gaussiandomains.GaussianInteger'>
```

## QQ\_I

**class** sympy.polys.domains.gaussiandomains.**GaussianRationalField**

Field of Gaussian rationals QQ\_I

The [QQ\\_I](#) (page 2536) domain represents the [Gaussian rationals](#)  $\mathbb{Q}(i)$  as a [Domain](#) (page 2504) in the domain system (see [Introducing the Domains of the poly module](#) (page 2477)).

By default a [Poly](#) (page 2378) created from an expression with coefficients that are combinations of rationals and  $I$  ( $\sqrt{-1}$ ) will have the domain [QQ\\_I](#) (page 2536).

```
>>> from sympy import Poly, Symbol, I
>>> x = Symbol('x')
>>> p = Poly(x**2 + I/2)
>>> p
Poly(x**2 + I/2, x, domain='QQ_I')
>>> p.domain
QQ_I
```



The polys option `gaussian=True` can be used to specify that the domain should be `QQ_I` (page 2536) even if the coefficients do not contain `I` or are all integers.

```
>>> Poly(x**2)
Poly(x**2, x, domain='ZZ')
>>> Poly(x**2 + I)
Poly(x**2 + I, x, domain='ZZ_I')
>>> Poly(x**2/2)
Poly(1/2*x**2, x, domain='QQ')
>>> Poly(x**2, gaussian=True)
Poly(x**2, x, domain='QQ_I')
>>> Poly(x**2 + I, gaussian=True)
Poly(x**2 + I, x, domain='QQ_I')
>>> Poly(x**2/2, gaussian=True)
Poly(1/2*x**2, x, domain='QQ_I')
```

The `QQ_I` (page 2536) domain can be used to factorise polynomials that are reducible over the Gaussian rationals.

```
>>> from sympy import factor, QQ_I
>>> factor(x**2/4 + 1)
(x**2 + 4)/4
>>> factor(x**2/4 + 1, domain='QQ_I')
(x - 2*I)*(x + 2*I)/4
>>> factor(x**2/4 + 1, domain=QQ_I)
(x - 2*I)*(x + 2*I)/4
```

It is also possible to specify the `QQ_I` (page 2536) domain explicitly with polys functions like `apart()` (page 2443).

```
>>> from sympy import apart
>>> apart(1/(1 + x**2))
1/(x**2 + 1)
>>> apart(1/(1 + x**2), domain=QQ_I)
I/(2*(x + I)) - I/(2*(x - I))
```

The corresponding `ring of integers` is the domain of the Gaussian integers `ZZ_I` (page 2534). Conversely `QQ_I` (page 2536) is the `field of fractions` of `ZZ_I` (page 2534).

```
>>> from sympy import ZZ_I, QQ_I, QQ
>>> ZZ_I.get_field()
QQ_I
>>> QQ_I.get_ring()
ZZ_I
```

When using the domain directly `QQ_I` (page 2536) can be used as a constructor.

```
>>> QQ_I(3, 4)
(3 + 4*I)
>>> QQ_I(5)
(5 + 0*I)
>>> QQ_I(QQ(2, 3), QQ(4, 5))
(2/3 + 4/5*I)
```

The domain elements of `QQ_I` (page 2536) are instances of `GaussianRational`

(page 2539) which support the field operations  $+$ ,  $-$ ,  $*$ ,  $**$ ,  $/$ .

```
>>> z1 = QQ_I(5, 1)
>>> z2 = QQ_I(2, QQ(1, 2))
>>> z1
(5 + 1*I)
>>> z2
(2 + 1/2*I)
>>> z1 + z2
(7 + 3/2*I)
>>> z1 * z2
(19/2 + 9/2*I)
>>> z2 ** 2
(15/4 + 2*I)
```

True division ( $/$ ) in `QQ_I` (page 2536) gives an element of `QQ_I` (page 2536) and is always exact.

```
>>> z1 / z2
(42/17 + -2/17*I)
>>> QQ_I.exquo(z1, z2)
(42/17 + -2/17*I)
>>> z1 == (z1/z2)*z2
True
```

Both floor ( $//$ ) and modulo ( $\%$ ) division can be used with `GaussianRational` (page 2539) (see `div()` (page 2509)) but division is always exact so there is no remainder.

```
>>> z1 // z2
(42/17 + -2/17*I)
>>> z1 % z2
(0 + 0*I)
>>> QQ_I.div(z1, z2)
((42/17 + -2/17*I), (0 + 0*I))
>>> (z1//z2)*z2 + z1%z2 == z1
True
```

### **as\_AlgebraicField()**

Get equivalent domain as an `AlgebraicField`.

### **denom(a)**

Get the denominator of  $a$ .

### **dtype**

alias of `GaussianRational` (page 2539)

### **from\_GaussianIntegerRing(a, K0)**

Convert a `ZZ_I` element to `QQ_I`.

### **from\_GaussianRationalField(a, K0)**

Convert a `QQ_I` element to `QQ_I`.

### **get\_field()**

Returns a field associated with `self`.

**get\_ring()**

Returns a ring associated with self.

**numer(a)**

Get the numerator of a.

**class** sympy.polys.domains.gaussiandomains.**GaussianRational**(x, y=0)

Gaussian rational: domain element for  $QQ_I$  (page 2536)

```
>>> from sympy import QQ_I, QQ
>>> z = QQ_I(QQ(2, 3), QQ(4, 5))
>>> z
(2/3 + 4/5*I)
>>> type(z)
<class 'sympy.polys.domains.gaussiandomains.GaussianRational'>
```

## $QQ\langle a \rangle$

**class** sympy.polys.domains.**AlgebraicField**(dom, \*ext, alias=None)

Algebraic number field  $QQ\langle a \rangle$  (page 2539)

A  $QQ\langle a \rangle$  (page 2539) domain represents an algebraic number field  $\mathbb{Q}(a)$  as a *Domain* (page 2504) in the domain system (see *Introducing the Domains of the poly module* (page 2477)).

A *Poly* (page 2378) created from an expression involving algebraic numbers will treat the algebraic numbers as generators if the generators argument is not specified.

```
>>> from sympy import Poly, Symbol, sqrt
>>> x = Symbol('x')
>>> Poly(x**2 + sqrt(2))
Poly(x**2 + (sqrt(2)), x, sqrt(2), domain='ZZ')
```

That is a multivariate polynomial with  $\sqrt{2}$  treated as one of the generators (variables). If the generators are explicitly specified then  $\sqrt{2}$  will be considered to be a coefficient but by default the *EX* (page 2549) domain is used. To make a *Poly* (page 2378) with a  $QQ\langle a \rangle$  (page 2539) domain the argument `extension=True` can be given.

```
>>> Poly(x**2 + sqrt(2), x)
Poly(x**2 + sqrt(2), x, domain='EX')
>>> Poly(x**2 + sqrt(2), x, extension=True)
Poly(x**2 + sqrt(2), x, domain='QQ<sqrt(2)>')
```

A generator of the algebraic field extension can also be specified explicitly which is particularly useful if the coefficients are all rational but an extension field is needed (e.g. to factor the polynomial).

```
>>> Poly(x**2 + 1)
Poly(x**2 + 1, x, domain='ZZ')
>>> Poly(x**2 + 1, extension=sqrt(2))
Poly(x**2 + 1, x, domain='QQ<sqrt(2)>')
```

It is possible to factorise a polynomial over a  $QQ\langle a \rangle$  (page 2539) domain using the extension argument to *factor()* (page 2373) or by specifying the domain explicitly.

```

>>> from sympy import factor, QQ
>>> factor(x**2 - 2)
x**2 - 2
>>> factor(x**2 - 2, extension=sqrt(2))
(x - sqrt(2))*(x + sqrt(2))
>>> factor(x**2 - 2, domain='QQ<sqrt(2)>')
(x - sqrt(2))*(x + sqrt(2))
>>> factor(x**2 - 2, domain=QQ.algebraic_field(sqrt(2)))
(x - sqrt(2))*(x + sqrt(2))

```

The `extension=True` argument can be used but will only create an extension that contains the coefficients which is usually not enough to factorise the polynomial.

```

>>> p = x**3 + sqrt(2)*x**2 - 2*x - 2*sqrt(2)
>>> factor(p) # treats sqrt(2) as a symbol
(x + sqrt(2))*(x**2 - 2)
>>> factor(p, extension=True)
(x - sqrt(2))*(x + sqrt(2))**2
>>> factor(x**2 - 2, extension=True) # all rational coefficients
x**2 - 2

```

It is also possible to use `QQ<a>` (page 2539) with the `cancel()` (page 2376) and `gcd()` (page 2368) functions.

```

>>> from sympy import cancel, gcd
>>> cancel((x**2 - 2)/(x - sqrt(2)))
(x**2 - 2)/(x - sqrt(2))
>>> cancel((x**2 - 2)/(x - sqrt(2)), extension=sqrt(2))
x + sqrt(2)
>>> gcd(x**2 - 2, x - sqrt(2))
1
>>> gcd(x**2 - 2, x - sqrt(2), extension=sqrt(2))
x - sqrt(2)

```

When using the domain directly `QQ<a>` (page 2539) can be used as a constructor to create instances which then support the operations `+`, `-`, `*`, `**`, `/`. The `algebraic_field()` (page 2508) method is used to construct a particular `QQ<a>` (page 2539) domain. The `from_sympy()` (page 2513) method can be used to create domain elements from normal SymPy expressions.

```

>>> K = QQ.algebraic_field(sqrt(2))
>>> K
QQ<sqrt(2)>
>>> xk = K.from_sympy(3 + 4*sqrt(2))
>>> xk
ANP([4, 3], [1, 0, -2], QQ)

```

Elements of `QQ<a>` (page 2539) are instances of `ANP` (page 2568) which have limited printing support. The raw display shows the internal representation of the element as the list `[4, 3]` representing the coefficients of 1 and `sqrt(2)` for this element in the form  $a * \sqrt{2} + b * 1$  where  $a$  and  $b$  are elements of `QQ` (page 2529). The minimal polynomial for the generator  $(x^2 - 2)$  is also shown in the *DUP representation* (page 2479) as the list `[1, 0, -2]`. We can use `to_sympy()` (page 2517) to get a better printed form for the elements and to see the results of operations.

```

>>> xk = K.from_sympy(3 + 4*sqrt(2))
>>> yk = K.from_sympy(2 + 3*sqrt(2))
>>> xk * yk
ANP([17, 30], [1, 0, -2], QQ)
>>> K.to_sympy(xk * yk)
17*sqrt(2) + 30
>>> K.to_sympy(xk + yk)
5 + 7*sqrt(2)
>>> K.to_sympy(xk ** 2)
24*sqrt(2) + 41
>>> K.to_sympy(xk / yk)
sqrt(2)/14 + 9/7

```

Any expression representing an algebraic number can be used to generate a *QQ<a>* (page 2539) domain provided its *minimal polynomial* can be computed. The function *minpoly()* (page 2711) function is used for this.

```

>>> from sympy import exp, I, pi, minpoly
>>> g = exp(2*I*pi/3)
>>> g
exp(2*I*pi/3)
>>> g.is_algebraic
True
>>> minpoly(g, x)
x**2 + x + 1
>>> factor(x**3 - 1, extension=g)
(x - 1)*(x - exp(2*I*pi/3))*(x + 1 + exp(2*I*pi/3))

```

It is also possible to make an algebraic field from multiple extension elements.

```

>>> K = QQ.algebraic_field(sqrt(2), sqrt(3))
>>> K
QQ<sqrt(2) + sqrt(3)>
>>> p = x**4 - 5*x**2 + 6
>>> factor(p)
(x**2 - 3)*(x**2 - 2)
>>> factor(p, domain=K)
(x - sqrt(2))*(x + sqrt(2))*(x - sqrt(3))*(x + sqrt(3))
>>> factor(p, extension=[sqrt(2), sqrt(3)])
(x - sqrt(2))*(x + sqrt(2))*(x - sqrt(3))*(x + sqrt(3))

```

Multiple extension elements are always combined together to make a single *primitive element*. In the case of  $[\sqrt{2}, \sqrt{3}]$  the primitive element chosen is  $\sqrt{2} + \sqrt{3}$  which is why the domain displays as *QQ<sqrt(2) + sqrt(3)>*. The minimal polynomial for the primitive element is computed using the *primitive\_element()* (page 2712) function.

```

>>> from sympy import primitive_element
>>> primitive_element([sqrt(2), sqrt(3)], x)
(x**4 - 10*x**2 + 1, [1, 1])
>>> minpoly(sqrt(2) + sqrt(3), x)
x**4 - 10*x**2 + 1

```

The extension elements that generate the domain can be accessed from the do-

main using the `ext` (page 2543) and `orig_ext` (page 2545) attributes as instances of `AlgebraicNumber` (page 988). The minimal polynomial for the primitive element as a `DMP` (page 2561) instance is available as `mod` (page 2545).

```
>>> K = QQ.algebraic_field(sqrt(2), sqrt(3))
>>> K
QQ<sqrt(2) + sqrt(3)>
>>> K.ext
sqrt(2) + sqrt(3)
>>> K.orig_ext
(sqrt(2), sqrt(3))
>>> K.mod
DMP([1, 0, -10, 0, 1], QQ, None)
```

The `discriminant` of the field can be obtained from the `discriminant()` (page 2543) method, and an `integral basis` from the `integral_basis()` (page 2544) method. The latter returns a list of `ANP` (page 2568) instances by default, but can be made to return instances of `Expr` (page 947) or `AlgebraicNumber` (page 988) by passing a `fmt` argument. The maximal order, or ring of integers, of the field can also be obtained from the `maximal_order()` (page 2545) method, as a `Submodule` (page 2726).

```
>>> zeta5 = exp(2*I*pi/5)
>>> K = QQ.algebraic_field(zeta5)
>>> K
QQ<exp(2*I*pi/5)>
>>> K.discriminant()
125
>>> K = QQ.algebraic_field(sqrt(5))
>>> K
QQ<sqrt(5)>
>>> K.integral_basis(fmt='sympy')
[1, 1/2 + sqrt(5)/2]
>>> K.maximal_order()
Submodule[[2, 0], [1, 1]]/2
```

The factorization of a rational prime into prime ideals of the field is computed by the `primes_above()` (page 2545) method, which returns a list of `PrimeIdeal` (page 2706) instances.

```
>>> zeta7 = exp(2*I*pi/7)
>>> K = QQ.algebraic_field(zeta7)
>>> K
QQ<exp(2*I*pi/7)>
>>> K.primes_above(11)
[(11, _x**3 + 5*_x**2 + 4*_x - 1), (11, _x**3 - 4*_x**2 - 5*_x - 1)]
```

## Notes

It is not currently possible to generate an algebraic extension over any domain other than `QQ` (page 2529). Ideally it would be possible to generate extensions like `QQ(x)(sqrt(x**2 - 2))`. This is equivalent to the quotient ring `QQ(x)[y]/(y**2 - x**2 + 2)` and there are two implementations of this kind of quotient ring/extension in the `QuotientRing` (page 2551) and `MonogenicFiniteExtension` (page 2475) classes. Each of those implementations needs some work to make them fully usable though.

**algebraic\_field(\*extension, alias=None)**

Returns an algebraic field, i.e.  $\mathbb{Q}(\alpha, \dots)$ .

**denom(a)**

Returns denominator of `a`.

**discriminant()**

Get the discriminant of the field.

**dtype**

alias of `ANP` (page 2568)

**ext**

Primitive element used for the extension.

```
>>> from sympy import QQ, sqrt
>>> K = QQ.algebraic_field(sqrt(2), sqrt(3))
>>> K.ext
sqrt(2) + sqrt(3)
```

**from\_AlgebraicField(a, K0)**

Convert AlgebraicField element 'a' to another AlgebraicField

**from\_GaussianIntegerRing(a, K0)**

Convert a GaussianInteger element 'a' to dtype.

**from\_GaussianRationalField(a, K0)**

Convert a GaussianRational element 'a' to dtype.

**from\_QQ(a, K0)**

Convert a Python Fraction object to dtype.

**from\_QQ\_gmpy(a, K0)**

Convert a GMPY mpq object to dtype.

**from\_QQ\_python(a, K0)**

Convert a Python Fraction object to dtype.

**from\_RealField(a, K0)**

Convert a mpmath mpf object to dtype.

**from\_ZZ(a, K0)**

Convert a Python int object to dtype.

**from\_ZZ\_gmpy(a, K0)**

Convert a GMPY mpz object to dtype.

**from\_ZZ\_python**(*a*, *K0*)

Convert a Python int object to dtype.

**from\_sympy**(*a*)

Convert SymPy's expression to dtype.

**get\_ring**()

Returns a ring associated with self.

**integral\_basis**(*fmt=None*)

Get an integral basis for the field.

**Parameters**

**fmt** : str, None, optional (default=None)

If None, return a list of [ANP](#) (page 2568) instances. If "sympy", convert each element of the list to an [Expr](#) (page 947), using self.to\_sympy(). If "alg", convert each element of the list to an [AlgebraicNumber](#) (page 988), using self.to\_alg\_num().

**Examples**

```
>>> from sympy import QQ, AlgebraicNumber, sqrt
>>> alpha = AlgebraicNumber(sqrt(5), alias='alpha')
>>> k = QQ.algebraic_field(alpha)
>>> B0 = k.integral_basis()
>>> B1 = k.integral_basis(fmt='sympy')
>>> B2 = k.integral_basis(fmt='alg')
>>> print(B0[1])
ANP([mpq(1,2), mpq(1,2)], [mpq(1,1), mpq(0,1), mpq(-5,1)], QQ)
>>> print(B1[1])
1/2 + alpha/2
>>> print(B2[1])
alpha/2 + 1/2
```

In the last two cases we get legible expressions, which print somewhat differently because of the different types involved:

```
>>> print(type(B1[1]))
<class 'sympy.core.add.Add'>
>>> print(type(B2[1]))
<class 'sympy.core.numbers.AlgebraicNumber'>
```

**See also:**

[to\\_sympy](#) (page 2545), [to\\_alg\\_num](#) (page 2545), [maximal\\_order](#) (page 2545)

**is\_negative**(*a*)

Returns True if *a* is negative.

**is\_nonnegative**(*a*)

Returns True if *a* is non-negative.

**is\_nonpositive**(*a*)

Returns True if *a* is non-positive.



**is\_positive(*a*)**

Returns True if *a* is positive.

**maximal\_order()**

Compute the maximal order, or ring of integers, of the field.

**Returns**

*Submodule* (page 2726).

**See also:**

*integral\_basis* (page 2544)

**mod**

Minimal polynomial for the primitive element of the extension.

```
>>> from sympy import QQ, sqrt
>>> K = QQ.algebraic_field(sqrt(2))
>>> K.mod
DMP([1, 0, -2], QQ, None)
```

**numer(*a*)**

Returns numerator of *a*.

**orig\_ext**

Original elements given to generate the extension.

```
>>> from sympy import QQ, sqrt
>>> K = QQ.algebraic_field(sqrt(2), sqrt(3))
>>> K.orig_ext
(sqrt(2), sqrt(3))
```

**primes\_above(*p*)**

Compute the prime ideals lying above a given rational prime *p*.

**to\_alg\_num(*a*)**

Convert *a* of dtype to an *AlgebraicNumber* (page 988).

**to\_sympy(*a*)**

Convert *a* of dtype to a SymPy object.

## RR

**class** sympy.polys.domains.**RealField**(*prec=53, dps=None, tol=None*)

Real numbers up to the given precision.

**almosteq(*a, b, tolerance=None*)**

Check if *a* and *b* are almost equal.

**from\_sympy(*expr*)**

Convert SymPy's number to dtype.

**gcd(*a, b*)**

Returns GCD of *a* and *b*.

**get\_exact()**

Returns an exact domain associated with self.

**get\_ring()**

Returns a ring associated with self.

**lcm(a, b)**

Returns LCM of a and b.

**to\_rational(element, limit=True)**

Convert a real number to rational number.

**to\_sympy(element)**

Convert element to SymPy number.

**class** sympy.polys.domains.mpelements.**RealElement**(val=(0, 0, 0, 0), \*\*kwargs)

An element of a real domain.

## CC

**class** sympy.polys.domains.**ComplexField**(prec=53, dps=None, tol=None)

Complex numbers up to the given precision.

**almosteq(a, b, tolerance=None)**

Check if a and b are almost equal.

**from\_sympy(expr)**

Convert SymPy's number to dtype.

**gcd(a, b)**

Returns GCD of a and b.

**get\_exact()**

Returns an exact domain associated with self.

**get\_ring()**

Returns a ring associated with self.

**is\_negative(element)**

Returns False for any ComplexElement.

**is\_nonnegative(element)**

Returns False for any ComplexElement.

**is\_nonpositive(element)**

Returns False for any ComplexElement.

**is\_positive(element)**

Returns False for any ComplexElement.

**lcm(a, b)**

Returns LCM of a and b.

**to\_sympy(element)**

Convert element to SymPy number.

**class** sympy.polys.domains.mpelements.**ComplexElement**(real=0, imag=0)

An element of a complex domain.

**K[x]**

**class** sympy.polys.domains.**PolynomialRing**(*domain\_or\_ring*, *symbols=None*,  
*order=None*)

A class for representing multivariate polynomial rings.

**factorial**(*a*)

Returns factorial of *a*.

**from\_AlgebraicField**(*a*, *K0*)

Convert an algebraic number to *dtype*.

**from\_ComplexField**(*a*, *K0*)

Convert a mpmath *mpf* object to *dtype*.

**from\_FractionField**(*a*, *K0*)

Convert a rational function to *dtype*.

**from\_GaussianIntegerRing**(*a*, *K0*)

Convert a *GaussianInteger* object to *dtype*.

**from\_GaussianRationalField**(*a*, *K0*)

Convert a *GaussianRational* object to *dtype*.

**from\_GlobalPolynomialRing**(*a*, *K0*)

Convert from old poly ring to *dtype*.

**from\_PolynomialRing**(*a*, *K0*)

Convert a polynomial to *dtype*.

**from\_QQ**(*a*, *K0*)

Convert a Python *Fraction* object to *dtype*.

**from\_QQ\_gmpy**(*a*, *K0*)

Convert a GMPY *mpq* object to *dtype*.

**from\_QQ\_python**(*a*, *K0*)

Convert a Python *Fraction* object to *dtype*.

**from\_RealField**(*a*, *K0*)

Convert a mpmath *mpf* object to *dtype*.

**from\_ZZ**(*a*, *K0*)

Convert a Python *int* object to *dtype*.

**from\_ZZ\_gmpy**(*a*, *K0*)

Convert a GMPY *mpz* object to *dtype*.

**from\_ZZ\_python**(*a*, *K0*)

Convert a Python *int* object to *dtype*.

**from\_sympy**(*a*)

Convert SymPy's expression to *dtype*.

**gcd**(*a*, *b*)

Returns GCD of *a* and *b*.

**gcdex**( $a, b$ )  
Extended GCD of  $a$  and  $b$ .

## get field()

**is\_negative( $a$ )**

**is nonnegative**( $a$ )

**is\_nonpositive( $a$ )**

**is positive( $a$ )**

**is unit( $a$ )**

 $\mathbf{lcm}(a, b)$ 

to `sympy(a)`

Convert  $a$  to a SymPy object.

 $K(x)$ 

```
class sympy.polys.domains.FractionField(domain_or_field, symbols=None,
 order=None)
```

$$\text{denom}(a)$$
**factorial**( $a$ )

```
from AlgebraicField(a, K0)
```

```
from ComplexField(a, K0)
```

`from FractionField(a, K0)`

```
from GaussianIntegerRing(a, K0)
```

```
from GaussianRationalField(a, K0)
```

```
from PolynomialRing(a, K0)
```

## Convert a polynomial to dtype.

**from\_QQ**(*a*, *K0*)  
Convert a Python Fraction object to dtype.

**from\_QQ\_gmpy**(*a*, *K0*)  
Convert a GMPY mpq object to dtype.

**from\_QQ\_python**(*a*, *K0*)  
Convert a Python Fraction object to dtype.

**from\_RealField**(*a*, *K0*)  
Convert a mpmath mpf object to dtype.

**from\_ZZ**(*a*, *K0*)  
Convert a Python int object to dtype.

**from\_ZZ\_gmpy**(*a*, *K0*)  
Convert a GMPY mpz object to dtype.

**from\_ZZ\_python**(*a*, *K0*)  
Convert a Python int object to dtype.

**from\_sympy**(*a*)  
Convert SymPy's expression to dtype.

**get\_ring**()  
Returns a field associated with self.

**is\_negative**(*a*)  
Returns True if  $LC(a)$  is negative.

**is\_nonnegative**(*a*)  
Returns True if  $LC(a)$  is non-negative.

**is\_nonpositive**(*a*)  
Returns True if  $LC(a)$  is non-positive.

**is\_positive**(*a*)  
Returns True if  $LC(a)$  is positive.

**numer**(*a*)  
Returns numerator of *a*.

**to\_sympy**(*a*)  
Convert *a* to a SymPy object.

## EX

**class** sympy.polys.domains.**ExpressionDomain**  
A class for arbitrary expressions.

**class** **Expression**(*ex*)  
An arbitrary expression.

**denom**(*a*)  
Returns denominator of *a*.

**dtype**

alias of [Expression](#) (page 2551)

**from\_ExpressionDomain(*a*, *K0*)**

Convert a EX object to dtype.

**from\_FractionField(*a*, *K0*)**

Convert a DMF object to dtype.

**from\_GaussianIntegerRing(*a*, *K0*)**

Convert a GaussianRational object to dtype.

**from\_GaussianRationalField(*a*, *K0*)**

Convert a GaussianRational object to dtype.

**from\_PolynomialRing(*a*, *K0*)**

Convert a DMP object to dtype.

**from\_QQ(*a*, *K0*)**

Convert a Python Fraction object to dtype.

**from\_QQ\_gmpy(*a*, *K0*)**

Convert a GMPY mpq object to dtype.

**from\_QQ\_python(*a*, *K0*)**

Convert a Python Fraction object to dtype.

**from\_RealField(*a*, *K0*)**

Convert a mpmath mpf object to dtype.

**from\_ZZ(*a*, *K0*)**

Convert a Python int object to dtype.

**from\_ZZ\_gmpy(*a*, *K0*)**

Convert a GMPY mpz object to dtype.

**from\_ZZ\_python(*a*, *K0*)**

Convert a Python int object to dtype.

**from\_sympy(*a*)**

Convert SymPy's expression to dtype.

**get\_field()**

Returns a field associated with self.

**get\_ring()**

Returns a ring associated with self.

**is\_negative(*a*)**

Returns True if *a* is negative.

**is\_nonnegative(*a*)**

Returns True if *a* is non-negative.

**is\_nonpositive(*a*)**

Returns True if *a* is non-positive.

**is\_positive(a)**

Returns True if a is positive.

**numer(a)**

Returns numerator of a.

**to\_sympy(a)**

Convert a to a SymPy object.

**class** ExpressionDomain.**Expression(ex)**

An arbitrary expression.

## Quotient ring

**class** sympy.polys.domains.quotientring.**QuotientRing**(ring, ideal)

Class representing (commutative) quotient rings.

You should not usually instantiate this by hand, instead use the constructor from the base ring in the construction.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> I = QQ.old_poly_ring(x).ideal(x**3 + 1)
>>> QQ.old_poly_ring(x).quotient_ring(I)
QQ[x]/<x**3 + 1>
```

Shorter versions are possible:

```
>>> QQ.old_poly_ring(x)/I
QQ[x]/<x**3 + 1>
```

```
>>> QQ.old_poly_ring(x)/[x**3 + 1]
QQ[x]/<x**3 + 1>
```

Attributes:

- ring - the base ring
- base\_ideal - the ideal used to form the quotient

## Sparse polynomials

Sparse polynomials are represented as dictionaries.

**sympy.polys.rings.ring**(symbols, domain, order=LexOrder())

Construct a polynomial ring returning (ring, x<sub>1</sub>, ..., x<sub>n</sub>).

### Parameters

**symbols** : str

Symbol/Expr or sequence of str, Symbol/Expr (non-empty)

**domain** : *Domain* (page 2504) or coercible

**order** : *MonomialOrder* (page 2431) or coercible, optional, defaults to lex

## Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.orderings import lex
```

```
>>> R, x, y, z = ring("x,y,z", ZZ, lex)
>>> R
Polynomial ring in x, y, z over ZZ with lex order
>>> x + y + z
x + y + z
>>> type(_)
<class 'sympy.polys.rings.PolyElement'>
```

`sympy.polys.rings.xring(symbols, domain, order=LexOrder())`

Construct a polynomial ring returning (ring, (x<sub>1</sub>, ..., x<sub>n</sub>)).

### Parameters

**symbols** : str

Symbol/Expr or sequence of str, Symbol/Expr (non-empty)

**domain** : *Domain* (page 2504) or coercible

**order** : *MonomialOrder* (page 2431) or coercible, optional, defaults to lex

## Examples

```
>>> from sympy.polys.rings import xring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.orderings import lex
```

```
>>> R, (x, y, z) = xring("x,y,z", ZZ, lex)
>>> R
Polynomial ring in x, y, z over ZZ with lex order
>>> x + y + z
x + y + z
>>> type(_)
<class 'sympy.polys.rings.PolyElement'>
```

`sympy.polys.rings.vring(symbols, domain, order=LexOrder())`

Construct a polynomial ring and inject x<sub>1</sub>, ..., x<sub>n</sub> into the global namespace.

### Parameters

**symbols** : str

Symbol/Expr or sequence of str, Symbol/Expr (non-empty)

**domain** : *Domain* (page 2504) or coercible

**order** : *MonomialOrder* (page 2431) or coercible, optional, defaults to lex



## Examples

```
>>> from sympy.polys.rings import vring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.orderings import lex
```

```
>>> vring("x,y,z", ZZ, lex)
Polynomial ring in x, y, z over ZZ with lex order
>>> x + y + z # noqa:
x + y + z
>>> type(_)
<class 'sympy.polys.rings.PolyElement'>
```

`sympy.polys.rings.sring(exprs, *symbols, **options)`

Construct a ring deriving generators and domain from options and input expressions.

### Parameters

**exprs** : *Expr* (page 947) or sequence of *Expr* (page 947) (sympifiable)

**symbols** : sequence of *Symbol* (page 976)/*Expr* (page 947)

**options** : keyword arguments understood by *Options* (page 2642)

## Examples

```
>>> from sympy import sring, symbols
```

```
>>> x, y, z = symbols("x,y,z")
>>> R, f = sring(x + 2*y + 3*z)
>>> R
Polynomial ring in x, y, z over ZZ with lex order
>>> f
x + 2*y + 3*z
>>> type(_)
<class 'sympy.polys.rings.PolyElement'>
```

**class** `sympy.polys.rings.PolyRing(symbols, domain, order=LexOrder())`

Multivariate distributed polynomial ring.

**add(\**objs*)**

Add a sequence of polynomials or containers of polynomials.

## Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
```

```
>>> R, x = ring("x", ZZ)
>>> R.add([x**2 + 2*i + 3 for i in range(4)])
4*x**2 + 24
```

(continues on next page)

(continued from previous page)

```
>>> _.factor_list()
(4, [(x**2 + 6, 1)])
```

**add\_gens**(*symbols*)Add the elements of *symbols* as generators to self**compose**(*other*)Add the generators of *other* to self**drop**(\**gens*)

Remove specified generators from this ring.

**drop\_to\_ground**(\**gens*)

Remove specified generators from the ring and inject them into its domain.

**index**(*gen*)Compute index of *gen* in self.gens.**monomial\_basis**(*i*)Return the *i*th-basis element.**mul**(\**objs*)

Multiply a sequence of polynomials or containers of polynomials.

## Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
```

```
>>> R, x = ring("x", ZZ)
>>> R.mul([x**2 + 2*i + 3 for i in range(4)])
x**8 + 24*x**6 + 206*x**4 + 744*x**2 + 945
>>> _.factor_list()
(1, [(x**2 + 3, 1), (x**2 + 5, 1), (x**2 + 7, 1), (x**2 + 9, 1)])
```

**class** sympy.polys.rings.PolyElement

Element of multivariate distributed polynomial ring.

**almosteq**(*p2*, *tolerance=None*)

Approximate equality test for polynomials.

**cancel**(*g*)Cancel common factors in a rational function *f/g*.

## Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x,y", ZZ)
```

```
>>> (2*x**2 - 2).cancel(x**2 - 2*x + 1)
(2*x + 2, x - 1)
```

### **coeff**(*element*)

Returns the coefficient that stands next to the given monomial.

#### Parameters

**element** : PolyElement (with `is_monomial = True`) or 1

## Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
```

```
>>> _, x, y, z = ring("x,y,z", ZZ)
>>> f = 3*x**2*y - x*y*z + 7*z**3 + 23
```

```
>>> f.coeff(x**2*y)
3
>>> f.coeff(x*y)
0
>>> f.coeff(1)
23
```

### **coffs**(*order=None*)

Ordered list of polynomial coefficients.

#### Parameters

**order** : *MonomialOrder* (page 2431) or coercible, optional

## Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.orderings import lex, grlex
```

```
>>> _, x, y = ring("x, y", ZZ, lex)
>>> f = x*y**7 + 2*x**2*y**3
```

```
>>> f.coffs()
[2, 1]
>>> f.coffs(grlex)
[1, 2]
```

**const()**

Returns the constant coefficient.

**content()**

Returns GCD of polynomial's coefficients.

**copy()**

Return a copy of polynomial self.

Polynomials are mutable; if one is interested in preserving a polynomial, and one plans to use inplace operations, one can copy the polynomial. This method makes a shallow copy.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.rings import ring
```

```
>>> R, x, y = ring('x, y', ZZ)
>>> p = (x + y)**2
>>> p1 = p.copy()
>>> p2 = p
>>> p[R.zero_monom] = 3
>>> p
x**2 + 2*x*y + y**2 + 3
>>> p1
x**2 + 2*x*y + y**2
>>> p2
x**2 + 2*x*y + y**2 + 3
```

**degree(x=None)**

The leading degree in x or the main variable.

Note that the degree of 0 is negative infinity (the SymPy object -oo).

**degrees()**

A tuple containing leading degrees in all variables.

Note that the degree of 0 is negative infinity (the SymPy object -oo)

**diff(x)**

Computes partial derivative in x.

### Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
```

```
>>> _, x, y = ring("x,y", ZZ)
>>> p = x + x**2*y**3
>>> p.diff(x)
2*x*y**3 + 1
```

**div(fv)**

Division algorithm, see [CLO] p64.

**fv array of polynomials**

return qv, r such that  $\text{self} = \sum(\text{fv}[i] * \text{qv}[i]) + r$

All polynomials are required not to be Laurent polynomials.

**Examples**

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
```

```
>>> _, x, y = ring('x, y', ZZ)
>>> f = x**3
>>> f0 = x - y**2
>>> f1 = x - y
>>> qv, r = f.div((f0, f1))
>>> qv[0]
x**2 + x*y**2 + y**4
>>> qv[1]
0
>>> r
y**6
```

**imul\_num(c)**

multiply inplace the polynomial p by an element in the coefficient ring, provided p is not one of the generators; else multiply not inplace

**Examples**

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
```

```
>>> _, x, y = ring('x, y', ZZ)
>>> p = x + y**2
>>> p1 = p.imul_num(3)
>>> p1
3*x + 3*y**2
>>> p1 is p
True
>>> p = x
>>> p1 = p.imul_num(3)
>>> p1
3*x
>>> p1 is p
False
```

**itercoeffs()**

Iterator over coefficients of a polynomial.

**itermonoms()**

Iterator over monomials of a polynomial.

**iterterms()**

Iterator over terms of a polynomial.

**leading\_expv()**

Leading monomial tuple according to the monomial ordering.

### Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ

>>> _, x, y, z = ring('x, y, z', ZZ)
>>> p = x**4 + x**3*y + x**2*z**2 + z**7
>>> p.leading_expv()
(4, 0, 0)
```

**leading\_monom()**

Leading monomial as a polynomial element.

### Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ

>>> _, x, y = ring('x, y', ZZ)
>>> (3*x*y + y**2).leading_monom()
x*y
```

**leading\_term()**

Leading term as a polynomial element.

### Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ

>>> _, x, y = ring('x, y', ZZ)
>>> (3*x*y + y**2).leading_term()
3*x*y
```

**listcoeffs()**

Unordered list of polynomial coefficients.

**listmonoms()**

Unordered list of polynomial monomials.

**listterms()**

Unordered list of polynomial terms.

**monic()**

Divides all coefficients by the leading coefficient.

**monoms**(*order=None*)

Ordered list of polynomial monomials.

**Parameters****order** : *MonomialOrder* (page 2431) or coercible, optional**Examples**

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.orderings import lex, grlex
```

```
>>> _, x, y = ring("x, y", ZZ, lex)
>>> f = x*y**7 + 2*x**2*y**3
```

```
>>> f.monoms()
[(2, 3), (1, 7)]
>>> f.monoms(grlex)
[(1, 7), (2, 3)]
```

**primitive()**

Returns content and a primitive polynomial.

**square()**

square of a polynomial

**Examples**

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
```

```
>>> _, x, y = ring('x, y', ZZ)
>>> p = x + y**2
>>> p.square()
x**2 + 2*x*y**2 + y**4
```

**strip\_zero()**

Eliminate monomials with zero coefficient.

**tail\_degree**(*x=None*)The tail degree in *x* or the main variable.

Note that the degree of 0 is negative infinity (the SymPy object -oo)

**tail\_degrees()**

A tuple containing tail degrees in all variables.

Note that the degree of 0 is negative infinity (the SymPy object -oo)

**terms(*order=None*)**

Ordered list of polynomial terms.

**Parameters**

**order** : [MonomialOrder](#) (page 2431) or coercible, optional

**Examples**

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.orderings import lex, grlex
```

```
>>> _, x, y = ring("x, y", ZZ, lex)
>>> f = x*y**7 + 2*x**2*y**3
```

```
>>> f.terms()
[((2, 3), 2), ((1, 7), 1)]
>>> f.terms(grlex)
[((1, 7), 1), ((2, 3), 2)]
```

## Sparse rational functions

Sparse polynomials are represented as dictionaries.

`sympy.polys.fields.field(symbols, domain, order=LexOrder())`

Construct new rational function field returning (field, x1, ..., xn).

`sympy.polys.fields.xfield(symbols, domain, order=LexOrder())`

Construct new rational function field returning (field, (x1, ..., xn)).

`sympy.polys.fields.vfield(symbols, domain, order=LexOrder())`

Construct new rational function field and inject generators into global namespace.

`sympy.polys.fields.sfield(exprs, *symbols, **options)`

Construct a field deriving generators and domain from options and input expressions.

**Parameters**

**exprs** : py:class: *Expr* or sequence of [Expr](#) (page 947) (sympifiable)

**symbols** : sequence of [Symbol](#) (page 976)/[Expr](#) (page 947)

**options** : keyword arguments understood by [Options](#) (page 2642)



## Examples

```
>>> from sympy import exp, log, symbols, sfield
```

```
>>> x = symbols("x")
>>> K, f = sfield((x*log(x) + 4*x**2)*exp(1/x + log(x)/3)/x**2)
>>> K
Rational function field in x, exp(1/x), log(x), x**(1/3) over ZZ with
↳ lex order
>>> f
(4*x**2*(exp(1/x)) + x*(exp(1/x))*(log(x)))/((x**(1/3))**5)
```

**class** sympy.polys.fields.FracField(symbols, domain, order=LexOrder())  
Multivariate distributed rational function field.

**class** sympy.polys.fields.FracElement(numer, denom=None)  
Element of multivariate distributed rational function field.

**diff(x)**  
Computes partial derivative in x.

## Examples

```
>>> from sympy.polys.fields import field
>>> from sympy.polys.domains import ZZ
```

```
>>> _, x, y, z = field("x,y,z", ZZ)
>>> ((x**2 + y)/(z + 1)).diff(x)
2*x/(z + 1)
```

## Dense polynomials

**class** sympy.polys.polyclasses.DMP(rep, dom, lev=None, ring=None)  
Dense Multivariate Polynomials over  $K$ .

**LC()**  
Returns the leading coefficient of  $f$ .

**TC()**  
Returns the trailing coefficient of  $f$ .

**abs()**  
Make all coefficients in  $f$  positive.

**add(g)**  
Add two multivariate polynomials  $f$  and  $g$ .

**add\_ground(c)**  
Add an element of the ground domain to  $f$ .

**all\_coeffs()**

Returns all coefficients from  $f$ .

**all\_monoms()**

Returns all monomials from  $f$ .

**all\_terms()**

Returns all terms from a  $f$ .

**cancel( $g$ ,  $include=True$ )**

Cancel common factors in a rational function  $f/g$ .

**cauchy\_lower\_bound()**

Computes the Cauchy lower bound on the nonzero roots of  $f$ .

**cauchy\_upper\_bound()**

Computes the Cauchy upper bound on the roots of  $f$ .

**clear\_denoms()**

Clear denominators, but keep the ground domain.

**coeffs( $order=None$ )**

Returns all non-zero coefficients from  $f$  in lex order.

**cofactors( $g$ )**

Returns GCD of  $f$  and  $g$  and their cofactors.

**compose( $g$ )**

Computes functional composition of  $f$  and  $g$ .

**content()**

Returns GCD of polynomial coefficients.

**convert( $dom$ )**

Convert the ground domain of  $f$ .

**count\_complex\_roots( $inf=None$ ,  $sup=None$ )**

Return the number of complex roots of  $f$  in  $[inf, sup]$ .

**count\_real\_roots( $inf=None$ ,  $sup=None$ )**

Return the number of real roots of  $f$  in  $[inf, sup]$ .

**decompose()**

Computes functional decomposition of  $f$ .

**deflate()**

Reduce degree of  $f$  by mapping  $x_i^m$  to  $y_i$ .

**degree( $j=0$ )**

Returns the leading degree of  $f$  in  $x_j$ .

**degree\_list()**

Returns a list of degrees of  $f$ .

**diff( $m=1$ ,  $j=0$ )**

Computes the  $m$ -th order derivative of  $f$  in  $x_j$ .

**discriminant()**Computes discriminant of  $f$ .**div( $g$ )**Polynomial division with remainder of  $f$  and  $g$ .**eject( $dom$ ,  $front=False$ )**

Eject selected generators into the ground domain.

**eval( $a$ ,  $j=0$ )**Evaluates  $f$  at the given point  $a$  in  $x_j$ .**exclude()**Remove useless generators from  $f$ .Returns the removed generators and the new excluded  $f$ .

### Examples

```
>>> from sympy.polys.polyclasses import DMP
>>> from sympy.polys.domains import ZZ
```

```
>>> DMP([[[ZZ(1)]], [[ZZ(1)], [ZZ(2)]]], ZZ).exclude()
([2], DMP([[1], [1, 2]], ZZ, None))
```

**exquo( $g$ )**Computes polynomial exact quotient of  $f$  and  $g$ .**exquo\_ground( $c$ )**Exact quotient of  $f$  by a an element of the ground domain.**factor\_list()**Returns a list of irreducible factors of  $f$ .**factor\_list\_include()**Returns a list of irreducible factors of  $f$ .**classmethod from\_dict( $rep$ ,  $lev$ ,  $dom$ )**Construct and instance of  $cls$  from a dict representation.**classmethod from\_list( $rep$ ,  $lev$ ,  $dom$ )**Create an instance of  $cls$  given a list of native coefficients.**classmethod from\_sympy\_list( $rep$ ,  $lev$ ,  $dom$ )**Create an instance of  $cls$  given a list of SymPy coefficients.**gcd( $g$ )**Returns polynomial GCD of  $f$  and  $g$ .**gcdex( $g$ )**

Extended Euclidean algorithm, if univariate.

**gff\_list()**Computes greatest factorial factorization of  $f$ .

**half\_gcdex(*g*)**

Half extended Euclidean algorithm, if univariate.

**homogeneous\_order()**

Returns the homogeneous order of *f*.

**homogenize(*s*)**

Return homogeneous polynomial of *f*

**inject(*front=False*)**

Inject ground domain generators into *f*.

**integrate(*m=1, j=0*)**

Computes the *m*-th order indefinite integral of *f* in *x<sub>j</sub>*.

**intervals(*all=False, eps=None, inf=None, sup=None, fast=False, sqf=False*)**

Compute isolating intervals for roots of *f*.

**invert(*g*)**

Invert *f* modulo *g*, if possible.

**property is\_cyclotomic**

Returns True if *f* is a cyclotomic polynomial.

**property is\_ground**

Returns True if *f* is an element of the ground domain.

**property is\_homogeneous**

Returns True if *f* is a homogeneous polynomial.

**property is\_irreducible**

Returns True if *f* has no factors over its domain.

**property is\_linear**

Returns True if *f* is linear in all its variables.

**property is\_monic**

Returns True if the leading coefficient of *f* is one.

**property is\_monomial**

Returns True if *f* is zero or has only one term.

**property is\_one**

Returns True if *f* is a unit polynomial.

**property is\_primitive**

Returns True if the GCD of the coefficients of *f* is one.

**property is\_quadratic**

Returns True if *f* is quadratic in all its variables.

**property is\_sqf**

Returns True if *f* is a square-free polynomial.

**property is\_zero**

Returns True if *f* is a zero polynomial.

**l1\_norm()**

Returns l1 norm of *f*.

**l2\_norm\_squared()**  
Return squared l2 norm of  $f$ .

**lcm( $g$ )**  
Returns polynomial LCM of  $f$  and  $g$ .

**lift()**  
Convert algebraic coefficients to rationals.

**max\_norm()**  
Returns maximum norm of  $f$ .

**mignotte\_sep\_bound\_squared()**  
Computes the squared Mignotte bound on root separations of  $f$ .

**monic()**  
Divides all coefficients by  $LC(f)$ .

**monoms( $order=None$ )**  
Returns all non-zero monomials from  $f$  in lex order.

**mul( $g$ )**  
Multiply two multivariate polynomials  $f$  and  $g$ .

**mul\_ground( $c$ )**  
Multiply  $f$  by an element of the ground domain.

**neg()**  
Negate all coefficients in  $f$ .

**norm()**  
Computes  $\text{Norm}(f)$ .

**nth( $*N$ )**  
Returns the  $n$ -th coefficient of  $f$ .

**pdiv( $g$ )**  
Polynomial pseudo-division of  $f$  and  $g$ .

**per( $rep, dom=None, kill=False, ring=None$ )**  
Create a DMP out of the given representation.

**permute( $P$ )**  
Returns a polynomial in  $K[x_{P(1)}, \dots, x_{P(n)}]$ .

### Examples

```
>>> from sympy.polys.polyclasses import DMP
>>> from sympy.polys.domains import ZZ
```

```
>>> DMP([[[ZZ(2)], [ZZ(1), ZZ(0)]], [[]]], ZZ).permute([1, 0, 2])
DMP([[[2], []], [[1, 0], []]], ZZ, None)
```

```
>>> DMP([[[ZZ(2)], [ZZ(1), ZZ(0)]], [[]]], ZZ).permute([1, 2, 0])
DMP([[[1], []], [[2, 0], []]], ZZ, None)
```

**pexquo**(*g*)  
Polynomial exact pseudo-quotient of *f* and *g*.

**pow**(*n*)  
Raise *f* to a non-negative power *n*.

**pquo**(*g*)  
Polynomial pseudo-quotient of *f* and *g*.

**prem**(*g*)  
Polynomial pseudo-remainder of *f* and *g*.

**primitive**()  
Returns content and a primitive form of *f*.

**quo**(*g*)  
Computes polynomial quotient of *f* and *g*.

**quo\_ground**(*c*)  
Quotient of *f* by a an element of the ground domain.

**refine\_root**(*s*, *t*, *eps=None*, *steps=None*, *fast=False*)  
Refine an isolating interval to the given precision.  
*eps* should be a rational number.

**rem**(*g*)  
Computes polynomial remainder of *f* and *g*.

**resultant**(*g*, *includePRS=False*)  
Computes resultant of *f* and *g* via PRS.

**revert**(*n*)  
Compute  $f^{**}(-1) \bmod x^{**n}$ .

**shift**(*a*)  
Efficiently compute Taylor shift  $f(x + a)$ .

**slice**(*m*, *n*, *j=0*)  
Take a continuous subsequence of terms of *f*.

**sqf\_list**(*all=False*)  
Returns a list of square-free factors of *f*.

**sqf\_list\_include**(*all=False*)  
Returns a list of square-free factors of *f*.

**sqf\_norm**()  
Computes square-free norm of *f*.

**sqf\_part**()  
Computes square-free part of *f*.

**sqr**()  
Square a multivariate polynomial *f*.

**sturm**()  
Computes the Sturm sequence of *f*.

**sub(*g*)**

Subtract two multivariate polynomials *f* and *g*.

**sub\_ground(*c*)**

Subtract an element of the ground domain from *f*.

**subresultants(*g*)**

Computes subresultant PRS sequence of *f* and *g*.

**terms(*order=None*)**

Returns all non-zero terms from *f* in lex order.

**terms\_gcd()**

Remove GCD of terms from the polynomial *f*.

**to\_dict(*zero=False*)**

Convert *f* to a dict representation with native coefficients.

**to\_exact()**

Make the ground domain exact.

**to\_field()**

Make the ground domain a field.

**to\_list()**

Convert *f* to a list representation with native coefficients.

**to\_ring()**

Make the ground domain a ring.

**to\_sympy\_dict(*zero=False*)**

Convert *f* to a dict representation with SymPy coefficients.

**to\_sympy\_list()**

Convert *f* to a list representation with SymPy coefficients.

**to\_tuple()**

Convert *f* to a tuple representation with native coefficients.

This is needed for hashing.

**total\_degree()**

Returns the total degree of *f*.

**transform(*p, q*)**

Evaluate functional transformation  $q^{*n} * f(p/q)$ .

**trunc(*p*)**

Reduce *f* modulo a constant *p*.

**unify(*g*)**

Unify representations of two multivariate polynomials.

**class** sympy.polys.polyclasses.DMF(*rep, dom, lev=None, ring=None*)

Dense Multivariate Fractions over *K*.

**add(*g*)**

Add two multivariate fractions *f* and *g*.

**cancel()**

Remove common factors from `f.num` and `f.den`.

**denom()**

Returns the denominator of `f`.

**exquo(*g*)**

Computes quotient of fractions `f` and `g`.

**frac\_unify(*g*)**

Unify representations of two multivariate fractions.

**half\_per(*rep*, *kill*=False)**

Create a DMP out of the given representation.

**invert(*check*=True)**

Computes inverse of a fraction `f`.

**property is\_one**

Returns True if `f` is a unit fraction.

**property is\_zero**

Returns True if `f` is a zero fraction.

**mul(*g*)**

Multiply two multivariate fractions `f` and `g`.

**neg()**

Negate all coefficients in `f`.

**numer()**

Returns the numerator of `f`.

**per(*num*, *den*, *cancel*=True, *kill*=False, *ring*=None)**

Create a DMF out of the given representation.

**poly\_unify(*g*)**

Unify a multivariate fraction and a polynomial.

**pow(*n*)**

Raise `f` to a non-negative power `n`.

**quo(*g*)**

Computes quotient of fractions `f` and `g`.

**sub(*g*)**

Subtract two multivariate fractions `f` and `g`.

**class** `sympy.polys.polyclasses.ANP(rep, mod, dom)`

Dense Algebraic Number Polynomials over a field.

**LC()**

Returns the leading coefficient of `f`.

**TC()**

Returns the trailing coefficient of `f`.



**property is\_ground**

Returns True if  $f$  is an element of the ground domain.

**property is\_one**

Returns True if  $f$  is a unit algebraic number.

**property is\_zero**

Returns True if  $f$  is a zero algebraic number.

**pow( $n$ )**

Raise  $f$  to a non-negative power  $n$ .

**to\_dict()**

Convert  $f$  to a dict representation with native coefficients.

**to\_list()**

Convert  $f$  to a list representation with native coefficients.

**to\_sympy\_dict()**

Convert  $f$  to a dict representation with SymPy coefficients.

**to\_sympy\_list()**

Convert  $f$  to a list representation with SymPy coefficients.

**to\_tuple()**

Convert  $f$  to a tuple representation with native coefficients.

This is needed for hashing.

**unify( $g$ )**

Unify representations of two algebraic numbers.

## Internals of the Polynomial Manipulation Module

The implementation of the polynomials module is structured internally in “levels”. There are four levels, called L0, L1, L2 and L3. The levels three and four contain the user-facing functionality and were described in the previous section. This section focuses on levels zero and one.

Level zero provides core polynomial manipulation functionality with C-like, low-level interfaces. Level one wraps this low-level functionality into object oriented structures. These are *not* the classes seen by the user, but rather classes used internally throughout the polys module.

There is one additional complication in the implementation. This comes from the fact that all polynomial manipulations are relative to a *ground domain*. For example, when factoring a polynomial like  $x^{10} - 1$ , one has to decide what ring the coefficients are supposed to belong to, or less trivially, what coefficients are allowed to appear in the factorization. This choice of coefficients is called a ground domain. Typical choices include the integers  $\mathbb{Z}$ , the rational numbers  $\mathbb{Q}$  or various related rings and fields. But it is perfectly legitimate (although in this case uninteresting) to factorize over polynomial rings such as  $k[Y]$ , where  $k$  is some fixed field.

Thus the polynomial manipulation algorithms (both complicated ones like factoring, and simpler ones like addition or multiplication) have to rely on other code to manipulate the coefficients. In the polynomial manipulation module, such code is encapsulated in so-called “domains”. A domain is basically a factory object: it takes various representations of data, and converts them into objects with unified interface. Every object created by a domain has

to implement the arithmetic operations  $+$ ,  $-$  and  $\times$ . Other operations are accessed through the domain, e.g. as in `ZZ.quo(ZZ(4), ZZ(2))`.

Note that there is some amount of *circularity*: the polynomial ring domains use the level one classes, the level one classes use the level zero functions, and level zero functions use domains. It is possible, in principle, but not in the current implementation, to work in rings like  $k[X][Y]$ . This would create even more layers. For this reason, working in the isomorphic ring  $k[X, Y]$  is preferred.

## Level Zero

Level zero contains the bulk code of the polynomial manipulation module.

## Manipulation of dense, multivariate polynomials

These functions can be used to manipulate polynomials in  $K[X_0, \dots, X_u]$ . Functions for manipulating multivariate polynomials in the dense representation have the prefix `dmp_`. Functions which only apply to univariate polynomials (i.e.  $u = 0$ ) have the prefix `dup_`. The ground domain  $K$  has to be passed explicitly. For many multivariate polynomial manipulation functions also the level  $u$ , i.e. the number of generators minus one, has to be passed. (Note that, in many cases, `dup_` versions of functions are available, which may be slightly more efficient.)

### Basic manipulation:

`sympy.polys.densebasic.dmp_LC(f, K)`

Return leading coefficient of  $f$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import poly_LC
```

```
>>> poly_LC([], ZZ)
0
>>> poly_LC([ZZ(1), ZZ(2), ZZ(3)], ZZ)
1
```

`sympy.polys.densebasic.dmp_TC(f, K)`

Return trailing coefficient of  $f$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import poly_TC
```

```
>>> poly_TC([], ZZ)
0
>>> poly_TC([ZZ(1), ZZ(2), ZZ(3)], ZZ)
3
```

`sympy.polys.densebasic.dmp_ground_LC(f, u, K)`

Return the ground leading coefficient.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_ground_LC
```

```
>>> f = ZZ.map([[[1], [2, 3]]])
```

```
>>> dmp_ground_LC(f, 2, ZZ)
1
```

`sympy.polys.densebasic.dmp_ground_TC(f, u, K)`

Return the ground trailing coefficient.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_ground_TC
```

```
>>> f = ZZ.map([[[1], [2, 3]]])
```

```
>>> dmp_ground_TC(f, 2, ZZ)
3
```

`sympy.polys.densebasic.dmp_true_LT(f, u, K)`

Return the leading term  $c * x_1^{n_1} \dots x_k^{n_k}$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_true_LT
```

```
>>> f = ZZ.map([[4], [2, 0], [3, 0, 0]])
```

```
>>> dmp_true_LT(f, 1, ZZ)
((2, 0), 4)
```

`sympy.polys.densebasic.dmp_degree(f, u)`

Return the leading degree of  $f$  in  $x_0$  in  $K[X]$ .

Note that the degree of 0 is negative infinity (the SymPy object `-oo`).

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_degree
```

```
>>> dmp_degree([[[[]]], 2)
-oo
```

```
>>> f = ZZ.map([[2], [1, 2, 3]])
```

```
>>> dmp_degree(f, 1)
1
```

`sympy.polys.densebasic.dmp_degree_in(f, j, u)`  
Return the leading degree of  $f$  in  $x_j$  in  $K[X]$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_degree_in
```

```
>>> f = ZZ.map([[2], [1, 2, 3]])
```

```
>>> dmp_degree_in(f, 0, 1)
1
>>> dmp_degree_in(f, 1, 1)
2
```

`sympy.polys.densebasic.dmp_degree_list(f, u)`  
Return a list of degrees of  $f$  in  $K[X]$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_degree_list
```

```
>>> f = ZZ.map([[1], [1, 2, 3]])
```

```
>>> dmp_degree_list(f, 1)
(1, 2)
```

`sympy.polys.densebasic.dmp_strip(f, u)`  
Remove leading zeros from  $f$  in  $K[X]$ .

### Examples

```
>>> from sympy.polys.densebasic import dmp_strip
```

```
>>> dmp_strip([], [0, 1, 2], [1], 1)
[[0, 1, 2], [1]]
```

`sympy.polys.densebasic.dmp_validate(f, K=None)`

Return the number of levels in `f` and recursively strip it.

### Examples

```
>>> from sympy.polys.densebasic import dmp_validate
```

```
>>> dmp_validate([], [0, 1, 2], [1])
([[1, 2], [1]], 1)
```

```
>>> dmp_validate([[1], 1])
Traceback (most recent call last):
...
ValueError: invalid data structure for a multivariate polynomial
```

`sympy.polys.densebasic.dup_reverse(f)`

Compute  $x^n * f(1/x)$ , i.e.: reverse `f` in  $K[x]$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dup_reverse
```

```
>>> f = ZZ.map([1, 2, 3, 0])
```

```
>>> dup_reverse(f)
[3, 2, 1]
```

`sympy.polys.densebasic.dmp_copy(f, u)`

Create a new copy of a polynomial `f` in  $K[X]$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_copy
```

```
>>> f = ZZ.map([[1], [1, 2]])
```

```
>>> dmp_copy(f, 1)
[[1], [1, 2]]
```

`sympy.polys.densebasic.dmp_to_tuple(f, u)`

Convert  $f$  into a nested tuple of tuples.

This is needed for hashing. This is similar to `dmp_copy()`.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_to_tuple
```

```
>>> f = ZZ.map([[1], [1, 2]])
```

```
>>> dmp_to_tuple(f, 1)
((1,), (1, 2))
```

`sympy.polys.densebasic.dmp_normal(f, u, K)`

Normalize a multivariate polynomial in the given domain.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_normal
```

```
>>> dmp_normal([], [0, 1.5, 2], 1, ZZ)
[[1, 2]]
```

`sympy.polys.densebasic.dmp_convert(f, u, K0, K1)`

Convert the ground domain of  $f$  from  $K_0$  to  $K_1$ .

### Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_convert
```

```
>>> R, x = ring("x", ZZ)
```

```
>>> dmp_convert([[R(1)], [R(2)]], 1, R.to_domain(), ZZ)
[[1], [2]]
>>> dmp_convert([[ZZ(1)], [ZZ(2)]], 1, ZZ, R.to_domain())
[[1], [2]]
```

`sympy.polys.densebasic.dmp_from_sympy(f, u, K)`

Convert the ground domain of  $f$  from SymPy to  $K$ .

### Examples

```
>>> from sympy import S
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_from_sympy
```

```
>>> dmp_from_sympy([[S(1)], [S(2)]], 1, ZZ) == [[ZZ(1)], [ZZ(2)]]
True
```

`sympy.polys.densebasic.dmp_nth(f, n, u, K)`

Return the *n*-th coefficient of *f* in  $K[x]$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_nth
```

```
>>> f = ZZ.map([[1], [2], [3]])
```

```
>>> dmp_nth(f, 0, 1, ZZ)
[3]
>>> dmp_nth(f, 4, 1, ZZ)
[]
```

`sympy.polys.densebasic.dmp_ground_nth(f, N, u, K)`

Return the ground *n*-th coefficient of *f* in  $K[x]$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_ground_nth
```

```
>>> f = ZZ.map([[1], [2, 3]])
```

```
>>> dmp_ground_nth(f, (0, 1), 1, ZZ)
2
```

`sympy.polys.densebasic.dmp_zero_p(f, u)`

Return True if *f* is zero in  $K[X]$ .

### Examples

```
>>> from sympy.polys.densebasic import dmp_zero_p
```

```
>>> dmp_zero_p([[[[[]]]]], 4)
True
>>> dmp_zero_p([[[[1]]]]], 4)
False
```

`sympy.polys.densebasic.dmp_zero(u)`

Return a multivariate zero.

### Examples

```
>>> from sympy.polys.densebasic import dmp_zero
```

```
>>> dmp_zero(4)
[[[[]]]]
```

`sympy.polys.densebasic.dmp_one_p(f, u, K)`

Return True if f is one in  $K[X]$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_one_p
```

```
>>> dmp_one_p([[[ZZ(1)]]], 2, ZZ)
True
```

`sympy.polys.densebasic.dmp_one(u, K)`

Return a multivariate one over K.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_one
```

```
>>> dmp_one(2, ZZ)
[[[1]]]
```

`sympy.polys.densebasic.dmp_ground_p(f, c, u)`

Return True if f is constant in  $K[X]$ .



### Examples

```
>>> from sympy.polys.densebasic import dmp_ground_p
```

```
>>> dmp_ground_p([[[3]]], 3, 2)
True
>>> dmp_ground_p([[[4]]], None, 2)
True
```

`sympy.polys.densebasic.dmp_ground(c, u)`

Return a multivariate constant.

### Examples

```
>>> from sympy.polys.densebasic import dmp_ground
```

```
>>> dmp_ground(3, 5)
[[[[[3]]]]]
>>> dmp_ground(1, -1)
1
```

`sympy.polys.densebasic.dmp_zeros(n, u, K)`

Return a list of multivariate zeros.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_zeros
```

```
>>> dmp_zeros(3, 2, ZZ)
[[[]], [[]], [[]]]
>>> dmp_zeros(3, -1, ZZ)
[0, 0, 0]
```

`sympy.polys.densebasic.dmp_grounds(c, n, u)`

Return a list of multivariate constants.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_grounds
```

```
>>> dmp_grounds(ZZ(4), 3, 2)
[[[4]], [[4]], [[4]]]
>>> dmp_grounds(ZZ(4), 3, -1)
[4, 4, 4]
```

`sympy.polys.densebasic.dmp_negative_p(f, u, K)`

Return True if  $LC(f)$  is negative.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_negative_p
```

```
>>> dmp_negative_p([[ZZ(1)], [-ZZ(1)]], 1, ZZ)
False
>>> dmp_negative_p([[-ZZ(1)], [ZZ(1)]], 1, ZZ)
True
```

`sympy.polys.densebasic.dmp_positive_p(f, u, K)`

Return True if  $LC(f)$  is positive.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_positive_p
```

```
>>> dmp_positive_p([[ZZ(1)], [-ZZ(1)]], 1, ZZ)
True
>>> dmp_positive_p([[-ZZ(1)], [ZZ(1)]], 1, ZZ)
False
```

`sympy.polys.densebasic.dmp_from_dict(f, u, K)`

Create a  $K[X]$  polynomial from a dict.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_from_dict
```

```
>>> dmp_from_dict({(0, 0): ZZ(3), (0, 1): ZZ(2), (2, 1): ZZ(1)}, 1, ZZ)
[[1, 0], [], [2, 3]]
>>> dmp_from_dict({}, 0, ZZ)
[]
```

`sympy.polys.densebasic.dmp_to_dict(f, u, K=None, zero=False)`

Convert a  $K[X]$  polynomial to a dict ``.

## Examples

```
>>> from sympy.polys.densebasic import dmp_to_dict
```

```
>>> dmp_to_dict([[1, 0], [], [2, 3]], 1)
{(0, 0): 3, (0, 1): 2, (2, 1): 1}
>>> dmp_to_dict([], 0)
{}
```

`sympy.polys.densebasic.dmp_swap(f, i, j, u, K)  
 Transform  $K[\dots x_i \dots x_j \dots]$  to  $K[\dots x_j \dots x_i \dots]$ .`

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_swap
```

```
>>> f = ZZ.map([[[2], [1, 0]], []])
```

```
>>> dmp_swap(f, 0, 1, 2, ZZ)
[[[2], []], [[1, 0], []]]
>>> dmp_swap(f, 1, 2, 2, ZZ)
[[[1], [2, 0]], [[]]]
>>> dmp_swap(f, 0, 2, 2, ZZ)
[[[1, 0]], [[2, 0], []]]
```

`sympy.polys.densebasic.dmp_permute(f, P, u, K)  
 Return a polynomial in  $K[x_{\{P(1)\}}, \dots, x_{\{P(n)\}}]$ .`

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_permute
```

```
>>> f = ZZ.map([[[2], [1, 0]], []])
```

```
>>> dmp_permute(f, [1, 0, 2], 2, ZZ)
[[[2], []], [[1, 0], []]]
>>> dmp_permute(f, [1, 2, 0], 2, ZZ)
[[[1], []], [[2, 0], []]]
```

`sympy.polys.densebasic.dmp_nest(f, l, K)  
 Return a multivariate value nested l-levels.`

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_nest
```

```
>>> dmp_nest([[ZZ(1)]], 2, ZZ)
[[[1]]]
```

`sympy.polys.densebasic.dmp_raise(f, l, u, K)`  
Return a multivariate polynomial raised *l*-levels.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_raise
```

```
>>> f = ZZ.map([], [1, 2])
```

```
>>> dmp_raise(f, 2, 1, ZZ)
[[[]], [[1]], [[2]]]
```

`sympy.polys.densebasic.dmp_deflate(f, u, K)`  
Map  $x_i^{m_i}$  to  $y_i$  in a polynomial in  $K[X]$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_deflate
```

```
>>> f = ZZ.map([[1, 0, 0, 2], [], [3, 0, 0, 4]])
```

```
>>> dmp_deflate(f, 1, ZZ)
((2, 3), [[1, 2], [3, 4]])
```

`sympy.polys.densebasic.dmp_multi_deflate(polys, u, K)`  
Map  $x_i^{m_i}$  to  $y_i$  in a set of polynomials in  $K[X]$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_multi_deflate
```

```
>>> f = ZZ.map([[1, 0, 0, 2], [], [3, 0, 0, 4]])
>>> g = ZZ.map([[1, 0, 2], [], [3, 0, 4]])
```

```
>>> dmp_multi_deflate((f, g), 1, ZZ)
((2, 1), ([[1, 0, 0, 2], [3, 0, 0, 4]], [[1, 0, 2], [3, 0, 4]]))
```

`sympy.polys.densebasic.dmp_inflate(f, M, u, K)`

Map  $y_i$  to  $x_i^{k_i}$  in a polynomial in  $K[X]$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_inflate
```

```
>>> f = ZZ.map([[1, 2], [3, 4]])
```

```
>>> dmp_inflate(f, (2, 3), 1, ZZ)
[[1, 0, 0, 2], [], [3, 0, 0, 4]]
```

`sympy.polys.densebasic.dmp_exclude(f, u, K)`

Exclude useless levels from  $f$ .

Return the levels excluded, the new excluded  $f$ , and the new  $u$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_exclude
```

```
>>> f = ZZ.map([[[1]], [[1], [2]]])
```

```
>>> dmp_exclude(f, 2, ZZ)
([2], [[1], [1, 2]], 1)
```

`sympy.polys.densebasic.dmp_include(f, J, u, K)`

Include useless levels in  $f$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_include
```

```
>>> f = ZZ.map([[1], [1, 2]])
```

```
>>> dmp_include(f, [2], 1, ZZ)
[[[1]], [[1], [2]]]
```

`sympy.polys.densebasic.dmp_inject(f, u, K, front=False)`

Convert  $f$  from  $K[X][Y]$  to  $K[X, Y]$ .

### Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_inject
```

```
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> dmp_inject([R(1), x + 2], 0, R.to_domain())
([[[1]], [[1], [2]]], 2)
>>> dmp_inject([R(1), x + 2], 0, R.to_domain(), front=True)
([[[1]], [[1, 2]]], 2)
```

`sympy.polys.densebasic.dmp_eject(f, u, K, front=False)`  
Convert  $f$  from  $K[X, Y]$  to  $K[X][Y]$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_eject
```

```
>>> dmp_eject([[[1]], [[1], [2]]], 2, ZZ['x', 'y'])
[1, x + 2]
```

`sympy.polys.densebasic.dmp_terms_gcd(f, u, K)`  
Remove GCD of terms from  $f$  in  $K[X]$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_terms_gcd
```

```
>>> f = ZZ.map([[1, 0], [1, 0, 0], [], []])
```

```
>>> dmp_terms_gcd(f, 1, ZZ)
((2, 1), [[1], [1, 0]])
```

`sympy.polys.densebasic.dmp_list_terms(f, u, K, order=None)`  
List all non-zero terms from  $f$  in the given order.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_list_terms
```

```
>>> f = ZZ.map([[1, 1], [2, 3]])
```

```
>>> dmp_list_terms(f, 1, ZZ)
[((1, 1), 1), ((1, 0), 1), ((0, 1), 2), ((0, 0), 3)]
>>> dmp_list_terms(f, 1, ZZ, order='grevlex')
[((1, 1), 1), ((1, 0), 1), ((0, 1), 2), ((0, 0), 3)]
```

`sympy.polys.densebasic.dmp_apply_pairs(f, g, h, args, u, K)`  
 Apply *h* to pairs of coefficients of *f* and *g*.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_apply_pairs
```

```
>>> h = lambda x, y, z: 2*x + y - z
```

```
>>> dmp_apply_pairs([[1], [2, 3]], [[3], [2, 1]], h, (1,), 1, ZZ)
[[4], [5, 6]]
```

`sympy.polys.densebasic.dmp_slice(f, m, n, u, K)`  
 Take a continuous subsequence of terms of *f* in  $K[X]$ .

`sympy.polys.densebasic.dup_random(n, a, b, K)`  
 Return a polynomial of degree *n* with coefficients in  $[a, b]$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dup_random
```

```
>>> dup_random(3, -10, 10, ZZ)
[-2, -8, 9, -4]
```

### Arithmetic operations:

`sympy.polys.densearith.dmp_add_term(f, c, i, u, K)`  
 Add  $c(x_2 \dots x_u) x_0^{**i}$  to *f* in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_add_term(x*y + 1, 2, 2)
2*x**2 + x*y + 1
```

`sympy.polys.densearith.dmp_sub_term(f, c, i, u, K)`  
Subtract  $c(x_2..x_u)*x_0^{**i}$  from *f* in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_sub_term(2*x**2 + x*y + 1, 2, 2)
x*y + 1
```

`sympy.polys.densearith.dmp_mul_term(f, c, i, u, K)`  
Multiply *f* by  $c(x_2..x_u)*x_0^{**i}$  in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_mul_term(x**2*y + x, 3*y, 2)
3*x**4*y**2 + 3*x**3*y
```

`sympy.polys.densearith.dmp_add_ground(f, c, u, K)`  
Add an element of the ground domain to *f*.

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_add_ground(x**3 + 2*x**2 + 3*x + 4, ZZ(4))
x**3 + 2*x**2 + 3*x + 8
```

`sympy.polys.densearith.dmp_sub_ground(f, c, u, K)`  
Subtract an element of the ground domain from *f*.



### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_sub_ground(x**3 + 2*x**2 + 3*x + 4, ZZ(4))
x**3 + 2*x**2 + 3*x
```

`sympy.polys.densearith.dmp_mul_ground(f, c, u, K)`  
 Multiply *f* by a constant value in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_mul_ground(2*x + 2*y, ZZ(3))
6*x + 6*y
```

`sympy.polys.densearith.dmp_quo_ground(f, c, u, K)`  
 Quotient by a constant in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ, QQ
```

```
>>> R, x,y = ring("x,y", ZZ)
>>> R.dmp_quo_ground(2*x**2*y + 3*x, ZZ(2))
x**2*y + x
```

```
>>> R, x,y = ring("x,y", QQ)
>>> R.dmp_quo_ground(2*x**2*y + 3*x, QQ(2))
x**2*y + 3/2*x
```

`sympy.polys.densearith.dmp_exquo_ground(f, c, u, K)`  
 Exact quotient by a constant in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)
```

```
>>> R.dmp_exquo_ground(x**2*y + 2*x, QQ(2))
1/2*x**2*y + x
```

`sympy.polys.densearith.dup_lshift(f, n, K)`  
 Efficiently multiply *f* by  $x^{**n}$  in  $K[x]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_lshift(x**2 + 1, 2)
x**4 + x**2
```

`sympy.polys.densearith.dup_rshift(f, n, K)`  
Efficiently divide  $f$  by  $x^n$  in  $K[x]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_rshift(x**4 + x**2, 2)
x**2 + 1
>>> R.dup_rshift(x**4 + x**2 + 2, 2)
x**2 + 1
```

`sympy.polys.densearith.dmp_abs(f, u, K)`  
Make all coefficients positive in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_abs(x**2*y - x)
x**2*y + x
```

`sympy.polys.densearith.dmp_neg(f, u, K)`  
Negate a polynomial in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_neg(x**2*y - x)
-x**2*y + x
```

`sympy.polys.densearith.dmp_add(f, g, u, K)`  
Add dense polynomials in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_add(x**2 + y, x**2*y + x)
x**2*y + x**2 + x + y
```

`sympy.polys.densearith.dmp_sub(f, g, u, K)`  
Subtract dense polynomials in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_sub(x**2 + y, x**2*y + x)
-x**2*y + x**2 - x + y
```

`sympy.polys.densearith.dmp_add_mul(f, g, h, u, K)`  
Returns  $f + g*h$  where  $f, g, h$  are in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_add_mul(x**2 + y, x, x + 2)
2*x**2 + 2*x + y
```

`sympy.polys.densearith.dmp_sub_mul(f, g, h, u, K)`  
Returns  $f - g*h$  where  $f, g, h$  are in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_sub_mul(x**2 + y, x, x + 2)
-2*x + y
```

`sympy.polys.densearith.dmp_mul(f, g, u, K)`  
Multiply dense polynomials in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_mul(x*y + 1, x)
x**2*y + x
```

`sympy.polys.densearith.dmp_sqr(f, u, K)`  
Square dense polynomials in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_sqr(x**2 + x*y + y**2)
x**4 + 2*x**3*y + 3*x**2*y**2 + 2*x*y**3 + y**4
```

`sympy.polys.densearith.dmp_pow(f, n, u, K)`  
Raise  $f$  to the  $n$ -th power in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_pow(x*y + 1, 3)
x**3*y**3 + 3*x**2*y**2 + 3*x*y + 1
```

`sympy.polys.densearith.dmp_pdiv(f, g, u, K)`  
Polynomial pseudo-division in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_pdiv(x**2 + x*y, 2*x + 2)
(2*x + 2*y - 2, -4*y + 4)
```

`sympy.polys.densearith.dmp_prem(f, g, u, K)`  
Polynomial pseudo-remainder in  $K[X]$ .

## Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_prem(x**2 + x*y, 2*x + 2)
-4*y + 4
```

`sympy.polys.densearith.dmp_pquo(f, g, u, K)`  
Polynomial exact pseudo-quotient in  $K[X]$ .

## Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = x**2 + x*y
>>> g = 2*x + 2*y
>>> h = 2*x + 2
```

```
>>> R.dmp_pquo(f, g)
2*x
```

```
>>> R.dmp_pquo(f, h)
2*x + 2*y - 2
```

`sympy.polys.densearith.dmp_pexquo(f, g, u, K)`  
Polynomial pseudo-quotient in  $K[X]$ .

## Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = x**2 + x*y
>>> g = 2*x + 2*y
>>> h = 2*x + 2
```

```
>>> R.dmp_pexquo(f, g)
2*x
```

```
>>> R.dmp_pexquo(f, h)
Traceback (most recent call last):
...
ExactQuotientFailed: [[2], [2]] does not divide [[1], [1, 0], []]
```

`sympy.polys.densearith.dmp_rr_div(f, g, u, K)`  
Multivariate division with remainder over a ring.

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_rr_div(x**2 + x*y, 2*x + 2)
(0, x**2 + x*y)
```

`sympy.polys.densearith.dmp_ff_div(f, g, u, K)`  
Polynomial division with remainder over a field.

### Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)
```

```
>>> R.dmp_ff_div(x**2 + x*y, 2*x + 2)
(1/2*x + 1/2*y - 1/2, -y + 1)
```

`sympy.polys.densearith.dmp_div(f, g, u, K)`  
Polynomial division with remainder in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ, QQ
```

```
>>> R, x,y = ring("x,y", ZZ)
>>> R.dmp_div(x**2 + x*y, 2*x + 2)
(0, x**2 + x*y)
```

```
>>> R, x,y = ring("x,y", QQ)
>>> R.dmp_div(x**2 + x*y, 2*x + 2)
(1/2*x + 1/2*y - 1/2, -y + 1)
```

`sympy.polys.densearith.dmp_rem(f, g, u, K)`  
Returns polynomial remainder in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ, QQ
```

```
>>> R, x,y = ring("x,y", ZZ)
>>> R.dmp_rem(x**2 + x*y, 2*x + 2)
x**2 + x*y
```

```
>>> R, x,y = ring("x,y", QQ)
>>> R.dmp_rem(x**2 + x*y, 2*x + 2)
-y + 1
```

`sympy.polys.densearith.dmp_quo(f, g, u, K)`  
Returns exact polynomial quotient in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ, QQ
```

```
>>> R, x,y = ring("x,y", ZZ)
>>> R.dmp_quo(x**2 + x*y, 2*x + 2)
0
```

```
>>> R, x,y = ring("x,y", QQ)
>>> R.dmp_quo(x**2 + x*y, 2*x + 2)
1/2*x + 1/2*y - 1/2
```

`sympy.polys.densearith.dmp_exquo(f, g, u, K)`  
Returns polynomial quotient in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = x**2 + x*y
>>> g = x + y
>>> h = 2*x + 2
```

```
>>> R.dmp_exquo(f, g)
x
```

```
>>> R.dmp_exquo(f, h)
Traceback (most recent call last):
...
ExactQuotientFailed: [[2], [2]] does not divide [[1], [1, 0], []]
```

`sympy.polys.densearith.dmp_max_norm(f, u, K)`  
Returns maximum norm of a polynomial in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_max_norm(2*x*y - x - 3)
3
```

`sympy.polys.densearith.dmp_l1_norm(f, u, K)`  
Returns l1 norm of a polynomial in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_l1_norm(2*x*y - x - 3)
6
```

`sympy.polys.densearith.dmp_expand(polys, u, K)`  
Multiply together several polynomials in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_expand([x**2 + y**2, x + 1])
x**3 + x**2 + x*y**2 + y**2
```

### Further tools:

`sympy.polys.densetools.dmp_integrate(f, m, u, K)`  
Computes the indefinite integral of *f* in  $x_0$  in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)
```

```
>>> R.dmp_integrate(x + 2*y, 1)
1/2*x**2 + 2*x*y
>>> R.dmp_integrate(x + 2*y, 2)
1/6*x**3 + x**2*y
```

`sympy.polys.densetools.dmp_integrate_in(f, m, j, u, K)`  
Computes the indefinite integral of *f* in  $x_j$  in  $K[X]$ .



## Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)
```

```
>>> R.dmp_integrate_in(x + 2*y, 1, 0)
1/2*x**2 + 2*x*y
>>> R.dmp_integrate_in(x + 2*y, 1, 1)
x*y + y**2
```

`sympy.polys.densetools.dmp_diff(f, m, u, K)`  
 m-th order derivative in  $x_u$  of a polynomial in  $K[X]$ .

## Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = x*y**2 + 2*x*y + 3*x + 2*y**2 + 3*y + 1
```

```
>>> R.dmp_diff(f, 1)
y**2 + 2*y + 3
>>> R.dmp_diff(f, 2)
0
```

`sympy.polys.densetools.dmp_diff_in(f, m, j, u, K)`  
 m-th order derivative in  $x_j$  of a polynomial in  $K[X]$ .

## Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = x*y**2 + 2*x*y + 3*x + 2*y**2 + 3*y + 1
```

```
>>> R.dmp_diff_in(f, 1, 0)
y**2 + 2*y + 3
>>> R.dmp_diff_in(f, 1, 1)
2*x*y + 2*x + 4*y + 3
```

`sympy.polys.densetools.dmp_eval(f, a, u, K)`  
 Evaluate a polynomial at  $x_u = a$  in  $K[X]$  using the Horner scheme.

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_eval(2*x*y + 3*x + y + 2, 2)
5*y + 8
```

`sympy.polys.densetools.dmp_eval_in(f, a, j, u, K)`

Evaluate a polynomial at  $x_j = a$  in  $K[X]$  using the Horner scheme.

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = 2*x*y + 3*x + y + 2
```

```
>>> R.dmp_eval_in(f, 2, 0)
5*y + 8
>>> R.dmp_eval_in(f, 2, 1)
7*x + 4
```

`sympy.polys.densetools.dmp_eval_tail(f, A, u, K)`

Evaluate a polynomial at  $x_j = a_j, \dots$  in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = 2*x*y + 3*x + y + 2
```

```
>>> R.dmp_eval_tail(f, [2])
7*x + 4
>>> R.dmp_eval_tail(f, [2, 2])
18
```

`sympy.polys.densetools.dmp_diff_eval_in(f, m, a, j, u, K)`

Differentiate and evaluate a polynomial in  $x_j$  at  $a$  in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = x*y**2 + 2*x*y + 3*x + 2*y**2 + 3*y + 1
```

```
>>> R.dmp_diff_eval_in(f, 1, 2, 0)
y**2 + 2*y + 3
>>> R.dmp_diff_eval_in(f, 1, 2, 1)
6*x + 11
```

`sympy.polys.densetools.dmp_trunc(f, p, u, K)`  
Reduce a  $K[X]$  polynomial modulo a polynomial  $p$  in  $K[Y]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = 3*x**2*y + 8*x**2 + 5*x*y + 6*x + 2*y + 3
>>> g = (y - 1).drop(x)
```

```
>>> R.dmp_trunc(f, g)
11*x**2 + 11*x + 5
```

`sympy.polys.densetools.dmp_ground_trunc(f, p, u, K)`  
Reduce a  $K[X]$  polynomial modulo a constant  $p$  in  $K$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = 3*x**2*y + 8*x**2 + 5*x*y + 6*x + 2*y + 3
```

```
>>> R.dmp_ground_trunc(f, ZZ(3))
-x**2 - x*y - y
```

`sympy.polys.densetools.dup_monic(f, K)`  
Divide all coefficients by  $LC(f)$  in  $K[x]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ, QQ
```

```
>>> R, x = ring("x", ZZ)
>>> R.dup_monic(3*x**2 + 6*x + 9)
x**2 + 2*x + 3
```

```
>>> R, x = ring("x", QQ)
>>> R.dup_monic(3*x**2 + 4*x + 2)
x**2 + 4/3*x + 2/3
```

`sympy.polys.denseutils.dmp_ground_monic(f, u, K)`  
Divide all coefficients by  $\text{LC}(f)$  in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ, QQ
```

```
>>> R, x,y = ring("x,y", ZZ)
>>> f = 3*x**2*y + 6*x**2 + 3*x*y + 9*y + 3
```

```
>>> R.dmp_ground_monic(f)
x**2*y + 2*x**2 + x*y + 3*y + 1
```

```
>>> R, x,y = ring("x,y", QQ)
>>> f = 3*x**2*y + 8*x**2 + 5*x*y + 6*x + 2*y + 3
```

```
>>> R.dmp_ground_monic(f)
x**2*y + 8/3*x**2 + 5/3*x*y + 2*x + 2/3*y + 1
```

`sympy.polys.denseutils.dup_content(f, K)`  
Compute the GCD of coefficients of  $f$  in  $K[x]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ, QQ
```

```
>>> R, x = ring("x", ZZ)
>>> f = 6*x**2 + 8*x + 12
```

```
>>> R.dup_content(f)
2
```

```
>>> R, x = ring("x", QQ)
>>> f = 6*x**2 + 8*x + 12
```

```
>>> R.dup_content(f)
2
```

`sympy.polys.densetools.dmp_ground_content(f, u, K)`

Compute the GCD of coefficients of  $f$  in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ, QQ
```

```
>>> R, x,y = ring("x,y", ZZ)
>>> f = 2*x*y + 6*x + 4*y + 12
```

```
>>> R.dmp_ground_content(f)
2
```

```
>>> R, x,y = ring("x,y", QQ)
>>> f = 2*x*y + 6*x + 4*y + 12
```

```
>>> R.dmp_ground_content(f)
2
```

`sympy.polys.densetools.dup_primitive(f, K)`

Compute content and the primitive form of  $f$  in  $K[x]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ, QQ
```

```
>>> R, x = ring("x", ZZ)
>>> f = 6*x**2 + 8*x + 12
```

```
>>> R.dup_primitive(f)
(2, 3*x**2 + 4*x + 6)
```

```
>>> R, x = ring("x", QQ)
>>> f = 6*x**2 + 8*x + 12
```

```
>>> R.dup_primitive(f)
(2, 3*x**2 + 4*x + 6)
```

`sympy.polys.densetools.dmp_ground_primitive(f, u, K)`

Compute content and the primitive form of  $f$  in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ, QQ
```

```
>>> R, x,y = ring("x,y", ZZ)
>>> f = 2*x*y + 6*x + 4*y + 12
```

```
>>> R.dmp_ground_primitive(f)
(2, x*y + 3*x + 2*y + 6)
```

```
>>> R, x,y = ring("x,y", QQ)
>>> f = 2*x*y + 6*x + 4*y + 12
```

```
>>> R.dmp_ground_primitive(f)
(2, x*y + 3*x + 2*y + 6)
```

`sympy.polys.densetools.dup_extract(f, g, K)`

Extract common content from a pair of polynomials in  $K[x]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_extract(6*x**2 + 12*x + 18, 4*x**2 + 8*x + 12)
(2, 3*x**2 + 6*x + 9, 2*x**2 + 4*x + 6)
```

`sympy.polys.densetools.dmp_ground_extract(f, g, u, K)`

Extract common content from a pair of polynomials in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_ground_extract(6*x*y + 12*x + 18, 4*x*y + 8*x + 12)
(2, 3*x*y + 6*x + 9, 2*x*y + 4*x + 6)
```

`sympy.polys.densetools.dup_real_imag(f, K)`

Return bivariate polynomials  $f_1$  and  $f_2$ , such that  $f = f_1 + f_2*I$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x, y", ZZ)
```

```
>>> R.dup_real_imag(x**3 + x**2 + x + 1)
(x**3 + x**2 - 3*x*y**2 + x - y**2 + 1, 3*x**2*y + 2*x*y - y**3 + y)
```

`sympy.polys.densetools.dup_mirror(f, K)`  
Evaluate efficiently the composition  $f(-x)$  in  $K[x]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_mirror(x**3 + 2*x**2 - 4*x + 2)
-x**3 + 2*x**2 + 4*x + 2
```

`sympy.polys.densetools.dup_scale(f, a, K)`  
Evaluate efficiently composition  $f(ax)$  in  $K[x]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_scale(x**2 - 2*x + 1, ZZ(2))
4*x**2 - 4*x + 1
```

`sympy.polys.densetools.dup_shift(f, a, K)`  
Evaluate efficiently Taylor shift  $f(x + a)$  in  $K[x]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_shift(x**2 - 2*x + 1, ZZ(2))
x**2 + 2*x + 1
```

`sympy.polys.densetools.dup_transform(f, p, q, K)`  
Evaluate functional transformation  $q^n * f(p/q)$  in  $K[x]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_transform(x**2 - 2*x + 1, x**2 + 1, x - 1)
x**4 - 2*x**3 + 5*x**2 - 4*x + 4
```

`sympy.polys.densetools.dmp_compose(f, g, u, K)`  
Evaluate functional composition  $f(g)$  in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x,y", ZZ)
```

```
>>> R.dmp_compose(x*y + 2*x + y, y)
y**2 + 3*y
```

`sympy.polys.densetools.dup_decompose(f, K)`  
Computes functional decomposition of  $f$  in  $K[x]$ .

Given a univariate polynomial  $f$  with coefficients in a field of characteristic zero, returns list  $[f_1, f_2, \dots, f_n]$ , where:

```
f = f_1 o f_2 o ... f_n = f_1(f_2(... f_n))
```

and  $f_2, \dots, f_n$  are monic and homogeneous polynomials of at least second degree.

Unlike factorization, complete functional decompositions of polynomials are not unique, consider examples:

1.  $f \circ g = f(x + b) \circ (g - b)$
2.  $x^{**n} \circ x^{**m} = x^{**m} \circ x^{**n}$
3.  $T_n \circ T_m = T_m \circ T_n$

where  $T_n$  and  $T_m$  are Chebyshev polynomials.

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_decompose(x**4 - 2*x**3 + x**2)
[x**2, x**2 - x]
```



## References

[R695]

`sympy.polys.denseutils.dmp_lift(f, u, K)`

Convert algebraic coefficients to integers in  $K[X]$ .

## Examples

```
>>> from sympy.polys import ring, QQ
>>> from sympy import I
```

```
>>> K = QQ.algebraic_field(I)
>>> R, x = ring("x", K)
```

```
>>> f = x**2 + K([QQ(1), QQ(0)])*x + K([QQ(2), QQ(0)])
```

```
>>> R.dmp_lift(f)
x**8 + 2*x**6 + 9*x**4 - 8*x**2 + 16
```

`sympy.polys.denseutils.dup_sign_variations(f, K)`

Compute the number of sign variations of  $f$  in  $K[x]$ .

## Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_sign_variations(x**4 - x**2 - x + 1)
2
```

`sympy.polys.denseutils.dmp_clear_denoms(f, u, K0, K1=None, convert=False)`

Clear denominators, i.e. transform  $K_0$  to  $K_1$ .

## Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x, y = ring("x,y", QQ)
```

```
>>> f = QQ(1,2)*x + QQ(1,3)*y + 1
```

```
>>> R.dmp_clear_denoms(f, convert=False)
(6, 3*x + 2*y + 6)
>>> R.dmp_clear_denoms(f, convert=True)
(6, 3*x + 2*y + 6)
```

`sympy.polys.denseutils.dmp_revert(f, g, u, K)`

Compute  $f^{**}(-1) \bmod x^{**}n$  using Newton iteration.

## Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x, y = ring("x,y", QQ)
```

## Manipulation of dense, univariate polynomials with finite field coefficients

Functions in this module carry the prefix `gf_`, referring to the classical name “Galois Fields” for finite fields. Note that many polynomial factorization algorithms work by reduction to the finite field case, so having special implementations for this case is justified both by performance, and by the necessity of certain methods which do not even make sense over general fields.

`sympy.polys.galoistools.gf_crt(U, M, K=None)`

Chinese Remainder Theorem.

Given a set of integer residues  $u_0, \dots, u_n$  and a set of co-prime integer moduli  $m_0, \dots, m_n$ , returns an integer  $u$ , such that  $u = u_i \bmod m_i$  for  $i = 0, \dots, n$ .

## Examples

Consider a set of residues  $U = [49, 76, 65]$  and a set of moduli  $M = [99, 97, 95]$ . Then we have:

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_crt

>>> gf_crt([49, 76, 65], [99, 97, 95], ZZ)
639985
```

This is the correct result because:

```
>>> [639985 % m for m in [99, 97, 95]]
[49, 76, 65]
```

Note: this is a low-level routine with no error checking.

**See also:**

[`sympy.ntheory.modular.crt` \(page 1509\)](#)  
a higher level crt routine

[`sympy.ntheory.modular.solve\_congruence` \(page 1511\)](#)

`sympy.polys.galoistools.gf_crt1(M, K)`

First part of the Chinese Remainder Theorem.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_crt1
```

```
>>> gf_crt1([99, 97, 95], ZZ)
(912285, [9215, 9405, 9603], [62, 24, 12])
```

`sympy.polys.galoistools.gf_crt2(U, M, p, E, S, K)`  
 Second part of the Chinese Remainder Theorem.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_crt2
```

```
>>> U = [49, 76, 65]
>>> M = [99, 97, 95]
>>> p = 912285
>>> E = [9215, 9405, 9603]
>>> S = [62, 24, 12]
```

```
>>> gf_crt2(U, M, p, E, S, ZZ)
639985
```

`sympy.polys.galoistools.gf_int(a, p)`  
 Coerce  $a \bmod p$  to an integer in the range  $[-p/2, p/2]$ .

### Examples

```
>>> from sympy.polys.galoistools import gf_int
```

```
>>> gf_int(2, 7)
2
>>> gf_int(5, 7)
-2
```

`sympy.polys.galoistools.gf_degree(f)`  
 Return the leading degree of  $f$ .

### Examples

```
>>> from sympy.polys.galoistools import gf_degree
```

```
>>> gf_degree([1, 1, 2, 0])
3
>>> gf_degree([])
-1
```

`sympy.polys.galoistools.gf_LC(f, K)`

Return the leading coefficient of *f*.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_LC
```

```
>>> gf_LC([3, 0, 1], ZZ)
3
```

`sympy.polys.galoistools.gf_TC(f, K)`

Return the trailing coefficient of *f*.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_TC
```

```
>>> gf_TC([3, 0, 1], ZZ)
1
```

`sympy.polys.galoistools.gf_strip(f)`

Remove leading zeros from *f*.

### Examples

```
>>> from sympy.polys.galoistools import gf_strip
```

```
>>> gf_strip([0, 0, 0, 3, 0, 1])
[3, 0, 1]
```

`sympy.polys.galoistools.gf_trunc(f, p)`

Reduce all coefficients modulo *p*.

### Examples

```
>>> from sympy.polys.galoistools import gf_trunc
```

```
>>> gf_trunc([7, -2, 3], 5)
[2, 3, 3]
```

`sympy.polys.galoistools.gf_normal(f, p, K)`  
 Normalize all coefficients in *K*.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_normal
```

```
>>> gf_normal([5, 10, 21, -3], 5, ZZ)
[1, 2]
```

`sympy.polys.galoistools.gf_from_dict(f, p, K)`  
 Create a  $\text{GF}(p)[x]$  polynomial from a dict.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_from_dict
```

```
>>> gf_from_dict({10: ZZ(4), 4: ZZ(33), 0: ZZ(-1)}, 5, ZZ)
[4, 0, 0, 0, 0, 0, 3, 0, 0, 0, 4]
```

`sympy.polys.galoistools.gf_to_dict(f, p, symmetric=True)`  
 Convert a  $\text{GF}(p)[x]$  polynomial to a dict.

### Examples

```
>>> from sympy.polys.galoistools import gf_to_dict
```

```
>>> gf_to_dict([4, 0, 0, 0, 0, 0, 3, 0, 0, 0, 4], 5)
{0: -1, 4: -2, 10: -1}
>>> gf_to_dict([4, 0, 0, 0, 0, 0, 3, 0, 0, 0, 4], 5, symmetric=False)
{0: 4, 4: 3, 10: 4}
```

`sympy.polys.galoistools.gf_from_int_poly(f, p)`  
 Create a  $\text{GF}(p)[x]$  polynomial from  $\mathbb{Z}[x]$ .

### Examples

```
>>> from sympy.polys.galoistools import gf_from_int_poly
```

```
>>> gf_from_int_poly([7, -2, 3], 5)
[2, 3, 3]
```

`sympy.polys.galoistools.gf_to_int_poly(f, p, symmetric=True)`  
Convert a  $\text{GF}(p)[x]$  polynomial to  $\mathbb{Z}[x]$ .

### Examples

```
>>> from sympy.polys.galoistools import gf_to_int_poly
```

```
>>> gf_to_int_poly([2, 3, 3], 5)
[2, -2, -2]
>>> gf_to_int_poly([2, 3, 3], 5, symmetric=False)
[2, 3, 3]
```

`sympy.polys.galoistools.gf_neg(f, p, K)`  
Negate a polynomial in  $\text{GF}(p)[x]$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_neg
```

```
>>> gf_neg([3, 2, 1, 0], 5, ZZ)
[2, 3, 4, 0]
```

`sympy.polys.galoistools.gf_add_ground(f, a, p, K)`  
Compute  $f + a$  where  $f$  in  $\text{GF}(p)[x]$  and  $a$  in  $\text{GF}(p)$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_add_ground
```

```
>>> gf_add_ground([3, 2, 4], 2, 5, ZZ)
[3, 2, 1]
```

`sympy.polys.galoistools.gf_sub_ground(f, a, p, K)`  
Compute  $f - a$  where  $f$  in  $\text{GF}(p)[x]$  and  $a$  in  $\text{GF}(p)$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_sub_ground
```

```
>>> gf_sub_ground([3, 2, 4], 2, 5, ZZ)
[3, 2, 2]
```

`sympy.polys.galoistools.gf_mul_ground(f, a, p, K)`  
 Compute  $f * a$  where  $f$  in  $GF(p)[x]$  and  $a$  in  $GF(p)$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_mul_ground
```

```
>>> gf_mul_ground([3, 2, 4], 2, 5, ZZ)
[1, 4, 3]
```

`sympy.polys.galoistools.gf_quo_ground(f, a, p, K)`  
 Compute  $f/a$  where  $f$  in  $GF(p)[x]$  and  $a$  in  $GF(p)$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_quo_ground
```

```
>>> gf_quo_ground(ZZ.map([3, 2, 4]), ZZ(2), 5, ZZ)
[4, 1, 2]
```

`sympy.polys.galoistools.gf_add(f, g, p, K)`  
 Add polynomials in  $GF(p)[x]$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_add
```

```
>>> gf_add([3, 2, 4], [2, 2, 2], 5, ZZ)
[4, 1]
```

`sympy.polys.galoistools.gf_sub(f, g, p, K)`  
 Subtract polynomials in  $GF(p)[x]$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_sub
```

```
>>> gf_sub([3, 2, 4], [2, 2, 2], 5, ZZ)
[1, 0, 2]
```

`sympy.polys.galoistools.gf_mul(f, g, p, K)`  
Multiply polynomials in  $\text{GF}(p)[x]$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_mul
```

```
>>> gf_mul([3, 2, 4], [2, 2, 2], 5, ZZ)
[1, 0, 3, 2, 3]
```

`sympy.polys.galoistools.gf_sqr(f, p, K)`  
Square polynomials in  $\text{GF}(p)[x]$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_sqr
```

```
>>> gf_sqr([3, 2, 4], 5, ZZ)
[4, 2, 3, 1, 1]
```

`sympy.polys.galoistools.gf_add_mul(f, g, h, p, K)`  
Returns  $f + g \cdot h$  where  $f, g, h$  in  $\text{GF}(p)[x]$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_add_mul
>>> gf_add_mul([3, 2, 4], [2, 2, 2], [1, 4], 5, ZZ)
[2, 3, 2, 2]
```

`sympy.polys.galoistools.gf_sub_mul(f, g, h, p, K)`  
Compute  $f - g \cdot h$  where  $f, g, h$  in  $\text{GF}(p)[x]$ .



## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_sub_mul
```

```
>>> gf_sub_mul([3, 2, 4], [2, 2, 2], [1, 4], 5, ZZ)
[3, 3, 2, 1]
```

`sympy.polys.galoistools.gf_expand(F, p, K)`

Expand results of `factor()` (page 2373) in  $\text{GF}(p)[x]$ .

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_expand
```

```
>>> gf_expand([(3, 2, 4), 1], ([2, 2], 2), ([3, 1], 3), 5, ZZ)
[4, 3, 0, 3, 0, 1, 4, 1]
```

`sympy.polys.galoistools.gf_div(f, g, p, K)`

Division with remainder in  $\text{GF}(p)[x]$ .

Given univariate polynomials *f* and *g* with coefficients in a finite field with *p* elements, returns polynomials *q* and *r* (quotient and remainder) such that  $f = q \cdot g + r$ .

Consider polynomials  $x^3 + x + 1$  and  $x^2 + x$  in  $\text{GF}(2)$ :

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_div, gf_add_mul

>>> gf_div(ZZ.map([1, 0, 1, 1]), ZZ.map([1, 1, 0]), 2, ZZ)
([1, 1], [1])
```

As result we obtained quotient  $x + 1$  and remainder 1, thus:

```
>>> gf_add_mul(ZZ.map([1]), ZZ.map([1, 1]), ZZ.map([1, 1, 0]), 2, ZZ)
[1, 0, 1, 1]
```

## References

[R696], [R697]

`sympy.polys.galoistools.gf_rem(f, g, p, K)`

Compute polynomial remainder in  $\text{GF}(p)[x]$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_rem
```

```
>>> gf_rem(ZZ.map([1, 0, 1, 1]), ZZ.map([1, 1, 0]), 2, ZZ)
[1]
```

`sympy.polys.galoistools.gf_quo(f, g, p, K)`  
Compute exact quotient in  $\text{GF}(p)[x]$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_quo
```

```
>>> gf_quo(ZZ.map([1, 0, 1, 1]), ZZ.map([1, 1, 0]), 2, ZZ)
[1, 1]
>>> gf_quo(ZZ.map([1, 0, 3, 2, 3]), ZZ.map([2, 2, 2]), 5, ZZ)
[3, 2, 4]
```

`sympy.polys.galoistools.gf_exquo(f, g, p, K)`  
Compute polynomial quotient in  $\text{GF}(p)[x]$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_exquo
```

```
>>> gf_exquo(ZZ.map([1, 0, 3, 2, 3]), ZZ.map([2, 2, 2]), 5, ZZ)
[3, 2, 4]
```

```
>>> gf_exquo(ZZ.map([1, 0, 1, 1]), ZZ.map([1, 1, 0]), 2, ZZ)
Traceback (most recent call last):
...
ExactQuotientFailed: [1, 1, 0] does not divide [1, 0, 1, 1]
```

`sympy.polys.galoistools.gf_lshift(f, n, K)`  
Efficiently multiply *f* by  $x^{**n}$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_lshift
```

```
>>> gf_lshift([3, 2, 4], 4, ZZ)
[3, 2, 4, 0, 0, 0, 0]
```

`sympy.polys.galoistools.gf_rshift(f, n, K)`

Efficiently divide *f* by  $x^{**n}$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_rshift
```

```
>>> gf_rshift([1, 2, 3, 4, 0], 3, ZZ)
([1, 2], [3, 4, 0])
```

`sympy.polys.galoistools.gf_pow(f, n, p, K)`

Compute  $f^{**n}$  in  $GF(p)[x]$  using repeated squaring.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_pow
```

```
>>> gf_pow([3, 2, 4], 3, 5, ZZ)
[2, 4, 4, 2, 2, 1, 4]
```

`sympy.polys.galoistools.gf_pow_mod(f, n, g, p, K)`

Compute  $f^{**n}$  in  $GF(p)[x]/(g)$  using repeated squaring.

Given polynomials *f* and *g* in  $GF(p)[x]$  and a non-negative integer *n*, efficiently computes  $f^{**n} \pmod{g}$  i.e. the remainder of  $f^{**n}$  from division by *g*, using the repeated squaring algorithm.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_pow_mod
```

```
>>> gf_pow_mod(ZZ.map([3, 2, 4]), 3, ZZ.map([1, 1]), 5, ZZ)
[]
```

## References

[R698]

`sympy.polys.galoistools.gf_gcd(f, g, p, K)`

Euclidean Algorithm in  $\text{GF}(p)[x]$ .

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_gcd
```

```
>>> gf_gcd(ZZ.map([3, 2, 4]), ZZ.map([2, 2, 3]), 5, ZZ)
[1, 3]
```

`sympy.polys.galoistools.gf_lcm(f, g, p, K)`

Compute polynomial LCM in  $\text{GF}(p)[x]$ .

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_lcm
```

```
>>> gf_lcm(ZZ.map([3, 2, 4]), ZZ.map([2, 2, 3]), 5, ZZ)
[1, 2, 0, 4]
```

`sympy.polys.galoistools.gf_cofactors(f, g, p, K)`

Compute polynomial GCD and cofactors in  $\text{GF}(p)[x]$ .

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_cofactors
```

```
>>> gf_cofactors(ZZ.map([3, 2, 4]), ZZ.map([2, 2, 3]), 5, ZZ)
([1, 3], [3, 3], [2, 1])
```

`sympy.polys.galoistools.gf_gcdex(f, g, p, K)`

Extended Euclidean Algorithm in  $\text{GF}(p)[x]$ .

Given polynomials  $f$  and  $g$  in  $\text{GF}(p)[x]$ , computes polynomials  $s$ ,  $t$  and  $h$ , such that  $h = \text{gcd}(f, g)$  and  $s*f + t*g = h$ . The typical application of EEA is solving polynomial diophantine equations.

Consider polynomials  $f = (x + 7)(x + 1)$ ,  $g = (x + 7)(x^2 + 1)$  in  $\text{GF}(11)[x]$ . Application of Extended Euclidean Algorithm gives:

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_gcdex, gf_mul, gf_add

>>> s, t, g = gf_gcdex(ZZ.map([1, 8, 7]), ZZ.map([1, 7, 1, 7]), 11, ZZ)
>>> s, t, g
([5, 6], [6], [1, 7])
```

As result we obtained polynomials  $s = 5x + 6$  and  $t = 6$ , and additionally  $\gcd(f, g) = x + 7$ . This is correct because:

```
>>> S = gf_mul(s, ZZ.map([1, 8, 7]), 11, ZZ)
>>> T = gf_mul(t, ZZ.map([1, 7, 1, 7]), 11, ZZ)

>>> gf_add(S, T, 11, ZZ) == [1, 7]
True
```

## References

[R699]

`sympy.polys.galoistools.gf_monic( $f, p, K$ )`  
 Compute LC and a monic polynomial in  $\text{GF}(p)[x]$ .

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_monic

>>> gf_monic(ZZ.map([3, 2, 4]), 5, ZZ)
(3, [1, 4, 3])
```

`sympy.polys.galoistools.gf_diff( $f, p, K$ )`  
 Differentiate polynomial in  $\text{GF}(p)[x]$ .

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_diff

>>> gf_diff([3, 2, 4], 5, ZZ)
[1, 2]
```

`sympy.polys.galoistools.gf_eval( $f, a, p, K$ )`  
 Evaluate  $f(a)$  in  $\text{GF}(p)$  using Horner scheme.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_eval
```

```
>>> gf_eval([3, 2, 4], 2, 5, ZZ)
0
```

`sympy.polys.galoistools.gf_multi_eval(f, A, p, K)`  
Evaluate  $f(a)$  for  $a$  in  $[a_1, \dots, a_n]$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_multi_eval
```

```
>>> gf_multi_eval([3, 2, 4], [0, 1, 2, 3, 4], 5, ZZ)
[4, 4, 0, 2, 0]
```

`sympy.polys.galoistools.gf_compose(f, g, p, K)`  
Compute polynomial composition  $f(g)$  in  $\text{GF}(p)[x]$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_compose
```

```
>>> gf_compose([3, 2, 4], [2, 2, 2], 5, ZZ)
[2, 4, 0, 3, 0]
```

`sympy.polys.galoistools.gf_compose_mod(g, h, f, p, K)`  
Compute polynomial composition  $g(h)$  in  $\text{GF}(p)[x]/(f)$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_compose_mod
```

```
>>> gf_compose_mod(ZZ.map([3, 2, 4]), ZZ.map([2, 2, 2]), ZZ.map([4, 3]), 5, ZZ)
[4]
```

`sympy.polys.galoistools.gf_trace_map(a, b, c, n, f, p, K)`  
Compute polynomial trace map in  $\text{GF}(p)[x]/(f)$ .  
Given a polynomial  $f$  in  $\text{GF}(p)[x]$ , polynomials  $a, b, c$  in the quotient ring  $\text{GF}(p)[x]/(f)$  such that  $b = c \cdot t \pmod{f}$  for some positive power  $t$  of  $p$ , and a positive integer  $n$ , returns a mapping:

```
a -> a**t**n, a + a**t + a**t**2 + ... + a**t**n (mod f)
```

In factorization context,  $b = x^p \bmod f$  and  $c = x \bmod f$ . This way we can efficiently compute trace polynomials in equal degree factorization routine, much faster than with other methods, like iterated Frobenius algorithm, for large degrees.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_trace_map
```

```
>>> gf_trace_map([1, 2], [4, 4], [1, 1], 4, [3, 2, 4], 5, ZZ)
([1, 3], [1, 3])
```

### References

[R700]

`sympy.polys.galoistools.gf_random( $n, p, K$ )`  
Generate a random polynomial in  $\text{GF}(p)[x]$  of degree  $n$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_random
>>> gf_random(10, 5, ZZ)
[1, 2, 3, 2, 1, 1, 1, 2, 0, 4, 2]
```

`sympy.polys.galoistools.gf_irreducible( $n, p, K$ )`  
Generate random irreducible polynomial of degree  $n$  in  $\text{GF}(p)[x]$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_irreducible
>>> gf_irreducible(10, 5, ZZ)
[1, 4, 2, 2, 3, 2, 4, 1, 4, 0, 4]
```

`sympy.polys.galoistools.gf_irreducible_p( $f, p, K$ )`  
Test irreducibility of a polynomial  $f$  in  $\text{GF}(p)[x]$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_irreducible_p
```

```
>>> gf_irreducible_p(ZZ.map([1, 4, 2, 2, 3, 2, 4, 1, 4, 0, 4]), 5, ZZ)
True
>>> gf_irreducible_p(ZZ.map([3, 2, 4]), 5, ZZ)
False
```

`sympy.polys.galoistools.gf_sqf_p(f, p, K)`  
Return True if *f* is square-free in  $\text{GF}(p)[x]$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_sqf_p
```

```
>>> gf_sqf_p(ZZ.map([3, 2, 4]), 5, ZZ)
True
>>> gf_sqf_p(ZZ.map([2, 4, 4, 2, 2, 1, 4]), 5, ZZ)
False
```

`sympy.polys.galoistools.gf_sqf_part(f, p, K)`  
Return square-free part of a  $\text{GF}(p)[x]$  polynomial.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_sqf_part
```

```
>>> gf_sqf_part(ZZ.map([1, 1, 3, 0, 1, 0, 2, 2, 1]), 5, ZZ)
[1, 4, 3]
```

`sympy.polys.galoistools.gf_sqf_list(f, p, K, all=False)`  
Return the square-free decomposition of a  $\text{GF}(p)[x]$  polynomial.

Given a polynomial *f* in  $\text{GF}(p)[x]$ , returns the leading coefficient of *f* and a square-free decomposition  $f_1^{e_1} f_2^{e_2} \dots f_k^{e_k}$  such that all *f<sub>i</sub>* are monic polynomials and (*f<sub>i</sub>*, *f<sub>j</sub>*) for *i*  $\neq$  *j* are co-prime and *e<sub>1</sub>*  $\dots$  *e<sub>k</sub>* are given in increasing order. All trivial terms (i.e. *f<sub>i</sub>* = 1) are not included in the output.

Consider polynomial *f* =  $x^{11} + 1$  over  $\text{GF}(11)[x]$ :

```
>>> from sympy.polys.domains import ZZ

>>> from sympy.polys.galoistools import (
... gf_from_dict, gf_diff, gf_sqf_list, gf_pow,
...)
...
```

(continues on next page)



(continued from previous page)

```
>>> f = gf_from_dict({11: ZZ(1), 0: ZZ(1)}, 11, ZZ)
```

Note that  $f'(x) = 0$ :

```
>>> gf_diff(f, 11, ZZ)
[]
```

This phenomenon does not happen in characteristic zero. However we can still compute square-free decomposition of  $f$  using `gf_sqf()`:

```
>>> gf_sqf_list(f, 11, ZZ)
(1, [[1, 1], 11])
```

We obtained factorization  $f = (x + 1)^{11}$ . This is correct because:

```
>>> gf_pow([1, 1], 11, 11, ZZ) == f
True
```

## References

[R701]

`sympy.polys.galoistools.gf_Qmatrix( $f, p, K$ )`

Calculate Berlekamp's  $Q$  matrix.

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_Qmatrix
```

```
>>> gf_Qmatrix([3, 2, 4], 5, ZZ)
[[1, 0],
 [3, 4]]
```

```
>>> gf_Qmatrix([1, 0, 0, 0, 1], 5, ZZ)
[[1, 0, 0, 0, 0],
 [0, 4, 0, 0, 0],
 [0, 0, 1, 0, 0],
 [0, 0, 0, 0, 4]]
```

`sympy.polys.galoistools.gf_Qbasis( $Q, p, K$ )`

Compute a basis of the kernel of  $Q$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_Qmatrix, gf_Qbasis
```

```
>>> gf_Qbasis(gf_Qmatrix([1, 0, 0, 0, 1], 5, ZZ), 5, ZZ)
[[1, 0, 0, 0], [0, 0, 1, 0]]
```

```
>>> gf_Qbasis(gf_Qmatrix([3, 2, 4], 5, ZZ), 5, ZZ)
[[1, 0]]
```

`sympy.polys.galoistools.gf_berlekamp(f, p, K)`  
Factor a square-free *f* in  $\text{GF}(p)[x]$  for small *p*.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_berlekamp
```

```
>>> gf_berlekamp([1, 0, 0, 0, 1], 5, ZZ)
[[1, 0, 2], [1, 0, 3]]
```

`sympy.polys.galoistools.gf_zassenhaus(f, p, K)`  
Factor a square-free *f* in  $\text{GF}(p)[x]$  for medium *p*.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_zassenhaus
```

```
>>> gf_zassenhaus(ZZ.map([1, 4, 3]), 5, ZZ)
[[1, 1], [1, 3]]
```

`sympy.polys.galoistools.gf_shoup(f, p, K)`  
Factor a square-free *f* in  $\text{GF}(p)[x]$  for large *p*.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_shoup
```

```
>>> gf_shoup(ZZ.map([1, 4, 3]), 5, ZZ)
[[1, 1], [1, 3]]
```

`sympy.polys.galoistools.gf_factor_sqf(f, p, K, method=None)`  
Factor a square-free polynomial *f* in  $\text{GF}(p)[x]$ .

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_factor_sqf
```

```
>>> gf_factor_sqf(ZZ.map([3, 2, 4]), 5, ZZ)
(3, [[1, 1], [1, 3]])
```

`sympy.polys.galoistools.gf_factor(f, p, K)`

Factor (non square-free) polynomials in  $\text{GF}(p)[x]$ .

Given a possibly non square-free polynomial  $f$  in  $\text{GF}(p)[x]$ , returns its complete factorization into irreducibles:

```
f_1(x)**e_1 f_2(x)**e_2 ... f_d(x)**e_d
```

where each  $f_i$  is a monic polynomial and  $\text{gcd}(f_i, f_j) == 1$ , for  $i \neq j$ . The result is given as a tuple consisting of the leading coefficient of  $f$  and a list of factors of  $f$  with their multiplicities.

The algorithm proceeds by first computing square-free decomposition of  $f$  and then iteratively factoring each of square-free factors.

Consider a non square-free polynomial  $f = (7x + 1)(x + 2)^2$  in  $\text{GF}(11)[x]$ . We obtain its factorization into irreducibles as follows:

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_factor

>>> gf_factor(ZZ.map([5, 2, 7, 2]), 11, ZZ)
(5, [[1, 2], 1), ([1, 8], 2)])
```

We arrived with factorization  $f = 5(x + 2)(x + 8)^2$ . We did not recover the exact form of the input polynomial because we requested to get monic factors of  $f$  and its leading coefficient separately.

Square-free factors of  $f$  can be factored into irreducibles over  $\text{GF}(p)$  using three very different methods:

### Berlekamp

efficient for very small values of  $p$  (usually  $p < 25$ )

### Cantor-Zassenhaus

efficient on average input and with “typical”  $p$

### Shoup-Kaltofen-Gathen

efficient with very large inputs and modulus

If you want to use a specific factorization method, instead of the default one, set `GF_FACTOR_METHOD` with one of `berlekamp`, `zassenhaus` or `shoup` values.

## References

[R702]

`sympy.polys.galoistools.gf_value(f, a)`

Value of polynomial 'f' at 'a' in field R.

## Examples

```
>>> from sympy.polys.galoistools import gf_value
```

```
>>> gf_value([1, 7, 2, 4], 11)
2204
```

`sympy.polys.galoistools.gf_csolve(f, n)`

To solve  $f(x)$  congruent 0 mod(n).

n is divided into canonical factors and  $f(x) \text{ cong } 0 \text{ mod}(p**e)$  will be solved for each factor. Applying the Chinese Remainder Theorem to the results returns the final answers.

## Examples

Solve  $[1, 1, 7]$  congruent 0 mod(189):

```
>>> from sympy.polys.galoistools import gf_csolve
>>> gf_csolve([1, 1, 7], 189)
[13, 49, 76, 112, 139, 175]
```

## References

[R703]

## Manipulation of sparse, distributed polynomials and vectors

Dense representations quickly require infeasible amounts of storage and computation time if the number of variables increases. For this reason, there is code to manipulate polynomials in a *sparse* representation. The Ring object and elements are implemented by the classes *PolyRing* (page 2553) and *PolyElement* (page 2554).

In commutative algebra, one often studies not only polynomials, but also *modules* over polynomial rings. The polynomial manipulation module provides rudimentary low-level support for finitely generated free modules. This is mainly used for Groebner basis computations (see there), so manipulation functions are only provided to the extend needed. They carry the prefix `sdm_`. Note that in examples, the generators of the free module are called  $f_1, f_2, \dots$

`sympy.polys.distributedmodules.sdm_monomial_mul(M, X)`

Multiply tuple X representing a monomial of  $K[X]$  into the tuple M representing a monomial of  $F$ .

## Examples

Multiplying  $xy^3$  into  $xf_1$  yields  $x^2y^3f_1$ :

```
>>> from sympy.polys.distributedmodules import sdm_monomial_mul
>>> sdm_monomial_mul((1, 1, 0), (1, 3))
(1, 2, 3)
```

`sympy.polys.distributedmodules.sdm_monomial_deg(M)`

Return the total degree of M.

## Examples

For example, the total degree of  $x^2yf_5$  is 3:

```
>>> from sympy.polys.distributedmodules import sdm_monomial_deg
>>> sdm_monomial_deg((5, 2, 1))
3
```

`sympy.polys.distributedmodules.sdm_monomial_divides(A, B)`

Does there exist a (polynomial) monomial X such that  $XA = B$ ?

## Examples

Positive examples:

In the following examples, the monomial is given in terms of x, y and the generator(s),  $f_1, f_2$  etc. The tuple form of that monomial is used in the call to `sdm_monomial_divides`. Note: the generator appears last in the expression but first in the tuple and other factors appear in the same order that they appear in the monomial expression.

$A = f_1$  divides  $B = f_1$

```
>>> from sympy.polys.distributedmodules import sdm_monomial_divides
>>> sdm_monomial_divides((1, 0, 0), (1, 0, 0))
True
```

$A = f_1$  divides  $B = x^2yf_1$

```
>>> sdm_monomial_divides((1, 0, 0), (1, 2, 1))
True
```

$A = xyf_5$  divides  $B = x^2yf_5$

```
>>> sdm_monomial_divides((5, 1, 1), (5, 2, 1))
True
```

Negative examples:

$A = f_1$  does not divide  $B = f_2$

```
>>> sdm_monomial_divides((1, 0, 0), (2, 0, 0))
False
```

$A = xf_1$  does not divide  $B = f_1$

```
>>> sdm_monomial_divides((1, 1, 0), (1, 0, 0))
False
```

$A = xy^2f_5$  does not divide  $B = yf_5$

```
>>> sdm_monomial_divides((5, 1, 2), (5, 0, 1))
False
```

`sympy.polys.distributedmodules.sdm_LC(f, K)`

Returns the leading coefficient of  $f$ .

`sympy.polys.distributedmodules.sdm_to_dict(f)`

Make a dictionary from a distributed polynomial.

`sympy.polys.distributedmodules.sdm_from_dict(d, O)`

Create an sdm from a dictionary.

Here  $O$  is the monomial order to use.

### Examples

```
>>> from sympy.polys.distributedmodules import sdm_from_dict
>>> from sympy.polys import QQ, lex
>>> dic = {(1, 1, 0): QQ(1), (1, 0, 0): QQ(2), (0, 1, 0): QQ(0)}
>>> sdm_from_dict(dic, lex)
[((1, 1, 0), 1), ((1, 0, 0), 2)]
```

`sympy.polys.distributedmodules.sdm_add(f, g, O, K)`

Add two module elements  $f, g$ .

Addition is done over the ground field  $K$ , monomials are ordered according to  $O$ .

### Examples

All examples use lexicographic order.

$(xyf_1) + (f_2) = f_2 + xyf_1$

```
>>> from sympy.polys.distributedmodules import sdm_add
>>> from sympy.polys import lex, QQ
>>> sdm_add([(1, 1, 1), QQ(1)], [(2, 0, 0), QQ(1)], lex, QQ)
[(2, 0, 0), 1], [(1, 1, 1), 1]
```

$(xyf_1) + (-xyf_1) = 0$

```
>>> sdm_add([(1, 1, 1), QQ(1)], [(1, 1, 1), QQ(-1)], lex, QQ)
[]
```

$(f_1) + (2f_1) = 3f_1$

```
>>> sdm_add([(1, 0, 0), QQ(1)], [(1, 0, 0), QQ(2)], lex, QQ)
[(1, 0, 0), 3]
```

$$(yf_1) + (xf_1) = xf_1 + yf_1$$

```
>>> sdm_add([((1, 0, 1), QQ(1))], [((1, 1, 0), QQ(1))], lex, QQ)
[((1, 1, 0), 1), ((1, 0, 1), 1)]
```

`sympy.polys.distributedmodules.sdm_LM(f)`

Returns the leading monomial of  $f$ .

Only valid if  $f \neq 0$ .

### Examples

```
>>> from sympy.polys.distributedmodules import sdm_LM, sdm_from_dict
>>> from sympy.polys import QQ, lex
>>> dic = {(1, 2, 3): QQ(1), (4, 0, 0): QQ(1), (4, 0, 1): QQ(1)}
>>> sdm_LM(sdm_from_dict(dic, lex))
(4, 0, 1)
```

`sympy.polys.distributedmodules.sdm_LT(f)`

Returns the leading term of  $f$ .

Only valid if  $f \neq 0$ .

### Examples

```
>>> from sympy.polys.distributedmodules import sdm_LT, sdm_from_dict
>>> from sympy.polys import QQ, lex
>>> dic = {(1, 2, 3): QQ(1), (4, 0, 0): QQ(2), (4, 0, 1): QQ(3)}
>>> sdm_LT(sdm_from_dict(dic, lex))
((4, 0, 1), 3)
```

`sympy.polys.distributedmodules.sdm_mul_term(f, term, O, K)`

Multiply a distributed module element  $f$  by a (polynomial) term  $term$ .

Multiplication of coefficients is done over the ground field  $K$ , and monomials are ordered according to  $O$ .

### Examples

$$0f_1 = 0$$

```
>>> from sympy.polys.distributedmodules import sdm_mul_term
>>> from sympy.polys import lex, QQ
>>> sdm_mul_term([((1, 0, 0), QQ(1))], ((0, 0), QQ(0)), lex, QQ)
[]
```

$$x0 = 0$$

```
>>> sdm_mul_term([], ((1, 0), QQ(1)), lex, QQ)
[]
```

$$(x)(f_1) = xf_1$$

```
>>> sdm_mul_term([((1, 0, 0), QQ(1))], ((1, 0), QQ(1)), lex, QQ)
[((1, 1, 0), 1)]
```

$$(2xy)(3xf_1 + 4yf_2) = 8xy^2f_2 + 6x^2yf_1$$

```
>>> f = [((2, 0, 1), QQ(4)), ((1, 1, 0), QQ(3))]
>>> sdm_mul_term(f, ((1, 1), QQ(2)), lex, QQ)
[((2, 1, 2), 8), ((1, 2, 1), 6)]
```

`sympy.polys.distributedmodules.sdm_zero()`

Return the zero module element.

`sympy.polys.distributedmodules.sdm_deg(f)`

Degree of  $f$ .

This is the maximum of the degrees of all its monomials. Invalid if  $f$  is zero.

### Examples

```
>>> from sympy.polys.distributedmodules import sdm_deg
>>> sdm_deg([((1, 2, 3), 1), ((10, 0, 1), 1), ((2, 3, 4), 4)])
7
```

`sympy.polys.distributedmodules.sdm_from_vector(vec, O, K, **opts)`

Create an sdm from an iterable of expressions.

Coefficients are created in the ground field  $K$ , and terms are ordered according to monomial order  $O$ . Named arguments are passed on to the polys conversion code and can be used to specify for example generators.

### Examples

```
>>> from sympy.polys.distributedmodules import sdm_from_vector
>>> from sympy.abc import x, y, z
>>> from sympy.polys import QQ, lex
>>> sdm_from_vector([x**2+y**2, 2*z], lex, QQ)
[((1, 0, 0, 1), 2), ((0, 2, 0, 0), 1), ((0, 0, 2, 0), 1)]
```

`sympy.polys.distributedmodules.sdm_to_vector(f, gens, K, n=None)`

Convert sdm  $f$  into a list of polynomial expressions.

The generators for the polynomial ring are specified via  $gens$ . The rank of the module is guessed, or passed via  $n$ . The ground field is assumed to be  $K$ .



## Examples

```
>>> from sympy.polys.distributedmodules import sdm_to_vector
>>> from sympy.abc import x, y, z
>>> from sympy.polys import QQ
>>> f = [((1, 0, 0, 1), QQ(2)), ((0, 2, 0, 0), QQ(1)), ((0, 0, 2, 0),
→ QQ(1))]
>>> sdm_to_vector(f, [x, y, z], QQ)
[x**2 + y**2, 2*z]
```

## Polynomial factorization algorithms

Many variants of Euclid's algorithm:

### Classical remainder sequence

Let  $K$  be a field, and consider the ring  $K[X]$  of polynomials in a single indeterminate  $X$  with coefficients in  $K$ . Given two elements  $f$  and  $g$  of  $K[X]$  with  $g \neq 0$  there are unique polynomials  $q$  and  $r$  such that  $f = qg + r$  and  $\deg(r) < \deg(g)$  or  $r = 0$ . They are denoted by  $\text{quo}(f, g)$  (*quotient*) and  $\text{rem}(f, g)$  (*remainder*), so we have the *division identity*

$$f = \text{quo}(f, g)g + \text{rem}(f, g).$$

It follows that every ideal  $I$  of  $K[X]$  is a principal ideal, generated by any element  $\neq 0$  of minimum degree (assuming  $I$  non-zero). In fact, if  $g$  is such a polynomial and  $f$  is any element of  $I$ ,  $\text{rem}(f, g)$  belongs to  $I$  as a linear combination of  $f$  and  $g$ , hence must be zero; therefore  $f$  is a multiple of  $g$ .

Using this result it is possible to find a **greatest common divisor** (gcd) of any polynomials  $f, g, \dots$  in  $K[X]$ . If  $I$  is the ideal formed by all linear combinations of the given polynomials with coefficients in  $K[X]$ , and  $d$  is its generator, then every common divisor of the polynomials also divides  $d$ . On the other hand, the given polynomials are multiples of the generator  $d$ ; hence  $d$  is a gcd of the polynomials, denoted  $\text{gcd}(f, g, \dots)$ .

An algorithm for the gcd of two polynomials  $f$  and  $g$  in  $K[X]$  can now be obtained as follows. By the division identity,  $r = \text{rem}(f, g)$  is in the ideal generated by  $f$  and  $g$ , as well as  $f$  is in the ideal generated by  $g$  and  $r$ . Hence the ideals generated by the pairs  $(f, g)$  and  $(g, r)$  are the same. Set  $f_0 = f$ ,  $f_1 = g$ , and define recursively  $f_i = \text{rem}(f_{i-2}, f_{i-1})$  for  $i \geq 2$ . The recursion ends after a finite number of steps with  $f_{k+1} = 0$ , since the degrees of the polynomials are strictly decreasing. By the above remark, all the pairs  $(f_{i-1}, f_i)$  generate the same ideal. In particular, the ideal generated by  $f$  and  $g$  is generated by  $f_k$  alone as  $f_{k+1} = 0$ . Hence  $d = f_k$  is a gcd of  $f$  and  $g$ .

The sequence of polynomials  $f_0, f_1, \dots, f_k$  is called the *Euclidean polynomial remainder sequence* determined by  $(f, g)$  because of the analogy with the classical **Euclidean algorithm** for the gcd of natural numbers.

The algorithm may be extended to obtain an expression for  $d$  in terms of  $f$  and  $g$  by using the full division identities to write recursively each  $f_i$  as a linear combination of  $f$  and  $g$ . This leads to an equation

$$d = uf + vg \quad (u, v \in K[X])$$

analogous to **Bézout's identity** in the case of integers.

`sympy.polys.euclidtools.dmp_half_gcdex(f, g, u, K)`

Half extended Euclidean algorithm in  $F[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

`sympy.polys.euclidtools.dmp_gcdex(f, g, u, K)`

Extended Euclidean algorithm in  $F[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

`sympy.polys.euclidtools.dmp_invert(f, g, u, K)`

Compute multiplicative inverse of  $f$  modulo  $g$  in  $F[X]$ .

### Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x = ring("x", QQ)
```

`sympy.polys.euclidtools.dmp_euclidean_prs(f, g, u, K)`

Euclidean polynomial remainder sequence (PRS) in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

## Simplified remainder sequences

Assume, as is usual, that the coefficient field  $K$  is the field of fractions of an integral domain  $A$ . In this case the coefficients (numerators and denominators) of the polynomials in the Euclidean remainder sequence tend to grow very fast.

If  $A$  is a unique factorization domain, the coefficients may be reduced by cancelling common factors of numerators and denominators. Further reduction is possible noting that a gcd of polynomials in  $K[X]$  is not unique: it may be multiplied by any (non-zero) constant factor.

Any polynomial  $f$  in  $K[X]$  can be simplified by extracting the denominators and common factors of the numerators of its coefficients. This yields the representation  $f = cF$  where  $c \in K$  is the *content* of  $f$  and  $F$  is a *primitive* polynomial, i.e., a polynomial in  $A[X]$  with coprime coefficients.

It is possible to start the algorithm by replacing the given polynomials  $f$  and  $g$  with their primitive parts. This will only modify  $\text{rem}(f, g)$  by a constant factor. Replacing it with its primitive part and continuing recursively we obtain all the primitive parts of the polynomials in the Euclidean remainder sequence, including the primitive  $\text{gcd}(f, g)$ .

This sequence is the *primitive polynomial remainder sequence*. It is an example of *general polynomial remainder sequences* where the computed remainders are modified by constant multipliers (or divisors) in order to simplify the results.

`sympy.polys.euclidtools.dmp_primitive_prs(f, g, u, K)`  
 Primitive polynomial remainder sequence (PRS) in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x,y", ZZ)
```

### Subresultant sequence

The coefficients of the primitive polynomial sequence do not grow exceedingly, but the computation of the primitive parts requires extra processing effort. Besides, the method only works with fraction fields of unique factorization domains, excluding, for example, the general number fields.

Collins [Collins67] realized that the so-called *subresultant polynomials* of a pair of polynomials also form a generalized remainder sequence. The coefficients of these polynomials are expressible as determinants in the coefficients of the given polynomials. Hence (the logarithm of) their size only grows linearly. In addition, if the coefficients of the given polynomials are in the subdomain  $A$ , so are those of the subresultant polynomials. This means that the subresultant sequence is comparable to the primitive remainder sequence without relying on unique factorization in  $A$ .

To see how subresultants are associated with remainder sequences recall that all polynomials  $h$  in the sequence are linear combinations of the given polynomials  $f$  and  $g$

$$h = uf + vg$$

with polynomials  $u$  and  $v$  in  $K[X]$ . Moreover, as is seen from the extended Euclidean algorithm, the degrees of  $u$  and  $v$  are relatively low, with limited growth from step to step.

Let  $n = \deg(f)$ , and  $m = \deg(g)$ , and assume  $n \geq m$ . If  $\deg(h) = j < m$ , the coefficients of the powers  $X^k$  ( $k > j$ ) in the products  $uf$  and  $vg$  cancel each other. In particular, the products must have the same degree, say,  $l$ . Then  $\deg(u) = l - n$  and  $\deg(v) = l - m$  with a total of  $2l - n - m + 2$  coefficients to be determined.

On the other hand, the equality  $h = uf + vg$  implies that  $l - j$  linear combinations of the coefficients are zero, those associated with the powers  $X^i$  ( $j < i \leq l$ ), and one has a given non-zero value, namely the leading coefficient of  $h$ .

To satisfy these  $l - j + 1$  linear equations the total number of coefficients to be determined cannot be lower than  $l - j + 1$ , in general. This leads to the inequality  $l \geq n + m - j - 1$ . Taking  $l = n + m - j - 1$ , we obtain  $\deg(u) = m - j - 1$  and  $\deg(v) = n - j - 1$ .

In the case  $j = 0$  the matrix of the resulting system of linear equations is the [Sylvester matrix](#)  $S(f, g)$  associated to  $f$  and  $g$ , an  $(n + m) \times (n + m)$  matrix with coefficients of  $f$  and  $g$  as entries.

Its determinant is the **resultant**  $\text{res}(f, g)$  of the pair  $(f, g)$ . It is non-zero if and only if  $f$  and  $g$  are relatively prime.

For any  $j$  in the interval from 0 to  $m$  the matrix of the linear system is an  $(n+m-2j) \times (n+m-2j)$  submatrix of the Sylvester matrix. Its determinant  $s_j(f, g)$  is called the  $j$  th *scalar subresultant* of  $f$  and  $g$ .

If  $s_j(f, g)$  is not zero, the associated equation  $h = uf + vg$  has a unique solution where  $\deg(h) = j$  and the leading coefficient of  $h$  has any given value; the one with leading coefficient  $s_j(f, g)$  is the  $j$  th *subresultant polynomial* or, briefly, *subresultant* of the pair  $(f, g)$ , and denoted  $S_j(f, g)$ . This choice guarantees that the remaining coefficients are also certain subdeterminants of the Sylvester matrix. In particular, if  $f$  and  $g$  are in  $A[X]$ , so is  $S_j(f, g)$  as well. This construction of subresultants applies to any  $j$  between 0 and  $m$  regardless of the value of  $s_j(f, g)$ ; if it is zero, then  $\deg(S_j(f, g)) < j$ .

The properties of subresultants are as follows. Let  $n_0 = \deg(f)$ ,  $n_1 = \deg(g)$ ,  $n_2, \dots, n_k$  be the decreasing sequence of degrees of polynomials in a remainder sequence. Let  $0 \leq j \leq n_1$ ; then

- $s_j(f, g) \neq 0$  if and only if  $j = n_i$  for some  $i$ .
- $S_j(f, g) \neq 0$  if and only if  $j = n_i$  or  $j = n_i - 1$  for some  $i$ .

Normally,  $n_{i-1} - n_i = 1$  for  $1 < i \leq k$ . If  $n_{i-1} - n_i > 1$  for some  $i$  (the *abnormal* case), then  $S_{n_{i-1}-1}(f, g)$  and  $S_{n_i}(f, g)$  are constant multiples of each other. Hence either one could be included in the polynomial remainder sequence. The former is given by smaller determinants, so it is expected to have smaller coefficients.

Collins defined the *subresultant remainder sequence* by setting

$$f_i = S_{n_{i-1}-1}(f, g) \quad (2 \leq i \leq k).$$

In the normal case, these are the same as the  $S_{n_i}(f, g)$ . He also derived expressions for the constants  $\gamma_i$  in the remainder formulas

$$\gamma_i f_i = \text{rem}(f_{i-2}, f_{i-1})$$

in terms of the leading coefficients of  $f_1, \dots, f_{i-1}$ , working in the field  $K$ .

Brown and Traub [BrownTraub71] later developed a recursive procedure for computing the coefficients  $\gamma_i$ . Their algorithm deals with elements of the domain  $A$  exclusively (assuming  $f, g \in A[X]$ ). However, in the abnormal case there was a problem, a division in  $A$  which could only be conjectured to be exact.

This was subsequently justified by Brown [Brown78] who showed that the result of the division is, in fact, a scalar subresultant. More specifically, the constant appearing in the computation of  $f_i$  is  $s_{n_{i-2}}(f, g)$  (Theorem 3). The implication of this discovery is that the scalar subresultants are computed as by-products of the algorithm, all but  $s_{n_k}(f, g)$  which is not needed after finding  $f_{k+1} = 0$ . Completing the last step we obtain all non-zero scalar subresultants, including the last one which is the resultant if this does not vanish.

`sympy.polys.euclidtools.dmp_inner_subresultants(f, g, u, K)`

Subresultant PRS algorithm in  $K[X]$ .

## Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = 3*x**2*y - y**3 - 4
>>> g = x**2 + x*y**3 - 9
```

```
>>> a = 3*x*y**4 + y**3 - 27*y + 4
>>> b = -3*y**10 - 12*y**7 + y**6 - 54*y**4 + 8*y**3 + 729*y**2 - 216*y
↳+ 16
```

```
>>> prs = [f, g, a, b]
>>> sres = [[1], [1], [3, 0, 0, 0, 0], [-3, 0, 0, -12, 1, 0, -54, 8, 729,
↳ -216, 16]]
```

```
>>> R.dmp_inner_subresultants(f, g) == (prs, sres)
True
```

`sympy.polys.euclidtools.dmp_subresultants(f, g, u, K)`  
Computes subresultant PRS of two polynomials in  $K[X]$ .

## Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = 3*x**2*y - y**3 - 4
>>> g = x**2 + x*y**3 - 9
```

```
>>> a = 3*x*y**4 + y**3 - 27*y + 4
>>> b = -3*y**10 - 12*y**7 + y**6 - 54*y**4 + 8*y**3 + 729*y**2 - 216*y
↳+ 16
```

```
>>> R.dmp_subresultants(f, g) == [f, g, a, b]
True
```

`sympy.polys.euclidtools.dmp_prs_resultant(f, g, u, K)`  
Resultant algorithm in  $K[X]$  using subresultant PRS.

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = 3*x**2*y - y**3 - 4
>>> g = x**2 + x*y**3 - 9
```

```
>>> a = 3*x*y**4 + y**3 - 27*y + 4
>>> b = -3*y**10 - 12*y**7 + y**6 - 54*y**4 + 8*y**3 + 729*y**2 - 216*y
↪ + 16
```

```
>>> res, prs = R.dmp_prs_resultant(f, g)
```

```
>>> res == b # resultant has n-1 variables
False
>>> res == b.drop(x)
True
>>> prs == [f, g, a, b]
True
```

`sympy.polys.euclidtools.dmp_zz_modular_resultant(f, g, p, u, K)`  
Compute resultant of  $f$  and  $g$  modulo a prime  $p$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = x + y + 2
>>> g = 2*x*y + x + 3
```

```
>>> R.dmp_zz_modular_resultant(f, g, 5)
-2*y**2 + 1
```

`sympy.polys.euclidtools.dmp_zz_collins_resultant(f, g, u, K)`  
Collins's modular resultant algorithm in  $\mathbb{Z}[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = x + y + 2
>>> g = 2*x*y + x + 3
```

```
>>> R.dmp_zz_collins_resultant(f, g)
-2*y**2 - 5*y + 1
```

`sympy.polys.euclidtools.dmp_qq_collins_resultant(f, g, u, K0)`  
 Collins's modular resultant algorithm in  $Q[X]$ .

### Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)
```

```
>>> f = QQ(1,2)*x + y + QQ(2,3)
>>> g = 2*x*y + x + 3
```

```
>>> R.dmp_qq_collins_resultant(f, g)
-2*y**2 - 7/3*y + 5/6
```

`sympy.polys.euclidtools.dmp_resultant(f, g, u, K, includePRS=False)`  
 Computes resultant of two polynomials in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = 3*x**2*y - y**3 - 4
>>> g = x**2 + x*y**3 - 9
```

```
>>> R.dmp_resultant(f, g)
-3*y**10 - 12*y**7 + y**6 - 54*y**4 + 8*y**3 + 729*y**2 - 216*y + 16
```

`sympy.polys.euclidtools.dmp_discriminant(f, u, K)`  
 Computes discriminant of a polynomial in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y,z,t = ring("x,y,z,t", ZZ)
```

```
>>> R.dmp_discriminant(x**2*y + x*z + t)
-4*y*t + z**2
```

`sympy.polys.euclidtools.dmp_rr_prs_gcd(f, g, u, K)`  
 Computes polynomial GCD using subresultants over a ring.  
 Returns  $(h, cff, cfg)$  such that  $a = \gcd(f, g)$ ,  $cff = \text{quo}(f, h)$ , and  $cfg = \text{quo}(g, h)$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)
```

```
>>> f = x**2 + 2*x*y + y**2
>>> g = x**2 + x*y
```

```
>>> R.dmp_rr_prs_gcd(f, g)
(x + y, x + y, x)
```

`sympy.polys.euclidtools.dmp_ff_prs_gcd(f, g, u, K)`

Computes polynomial GCD using subresultants over a field.

Returns  $(h, cff, cfg)$  such that  $a = \gcd(f, g)$ ,  $cff = \text{quo}(f, h)$ , and  $cfg = \text{quo}(g, h)$ .

### Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y, = ring("x,y", QQ)
```

```
>>> f = QQ(1,2)*x**2 + x*y + QQ(1,2)*y**2
>>> g = x**2 + x*y
```

```
>>> R.dmp_ff_prs_gcd(f, g)
(x + y, 1/2*x + 1/2*y, x)
```

`sympy.polys.euclidtools.dmp_zz_heu_gcd(f, g, u, K)`

Heuristic polynomial GCD in  $Z[X]$ .

Given univariate polynomials  $f$  and  $g$  in  $Z[X]$ , returns their GCD and cofactors, i.e. polynomials  $h$ ,  $cff$  and  $cfg$  such that:

```
h = gcd(f, g), cff = quo(f, h) and cfg = quo(g, h)
```

The algorithm is purely heuristic which means it may fail to compute the GCD. This will be signaled by raising an exception. In this case you will need to switch to another GCD method.

The algorithm computes the polynomial GCD by evaluating polynomials  $f$  and  $g$  at certain points and computing (fast) integer GCD of those evaluations. The polynomial GCD is recovered from the integer image by interpolation. The evaluation process reduces  $f$  and  $g$  variable by variable into a large integer. The final step is to verify if the interpolated polynomial is the correct GCD. This gives cofactors of the input polynomials as a side effect.



## Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)
```

```
>>> f = x**2 + 2*x*y + y**2
>>> g = x**2 + x*y
```

```
>>> R.dmp_zz_heu_gcd(f, g)
(x + y, x + y, x)
```

## References

[R704]

`sympy.polys.euclidtools.dmp_qq_heu_gcd(f, g, u, K0)`

Heuristic polynomial GCD in  $Q[X]$ .

Returns (*h*, *cff*, *cfg*) such that  $a = \gcd(f, g)$ ,  $cff = \text{quo}(f, h)$ , and  $cfg = \text{quo}(g, h)$ .

## Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y, = ring("x,y", QQ)
```

```
>>> f = QQ(1,4)*x**2 + x*y + y**2
>>> g = QQ(1,2)*x**2 + x*y
```

```
>>> R.dmp_qq_heu_gcd(f, g)
(x + 2*y, 1/4*x + 1/2*y, 1/2*x)
```

`sympy.polys.euclidtools.dmp_inner_gcd(f, g, u, K)`

Computes polynomial GCD and cofactors of *f* and *g* in  $K[X]$ .

Returns (*h*, *cff*, *cfg*) such that  $a = \gcd(f, g)$ ,  $cff = \text{quo}(f, h)$ , and  $cfg = \text{quo}(g, h)$ .

## Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)
```

```
>>> f = x**2 + 2*x*y + y**2
>>> g = x**2 + x*y
```

```
>>> R.dmp_inner_gcd(f, g)
(x + y, x + y, x)
```

`sympy.polys.euclidtools.dmp_gcd(f, g, u, K)`  
Computes polynomial GCD of  $f$  and  $g$  in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)
```

```
>>> f = x**2 + 2*x*y + y**2
>>> g = x**2 + x*y
```

```
>>> R.dmp_gcd(f, g)
x + y
```

`sympy.polys.euclidtools.dmp_lcm(f, g, u, K)`  
Computes polynomial LCM of  $f$  and  $g$  in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)
```

```
>>> f = x**2 + 2*x*y + y**2
>>> g = x**2 + x*y
```

```
>>> R.dmp_lcm(f, g)
x**3 + 2*x**2*y + x*y**2
```

`sympy.polys.euclidtools.dmp_content(f, u, K)`  
Returns GCD of multivariate coefficients.

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)
```

```
>>> R.dmp_content(2*x*y + 6*x + 4*y + 12)
2*y + 6
```

`sympy.polys.euclidtools.dmp_primitive(f, u, K)`  
Returns multivariate content and a primitive polynomial.

## Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)
```

```
>>> R.dmp_primitive(2*x*y + 6*x + 4*y + 12)
(2*y + 6, x + 2)
```

`sympy.polys.euclidtools.dmp_cancel(f, g, u, K, include=True)`  
Cancel common factors in a rational function  $f/g$ .

## Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_cancel(2*x**2 - 2, x**2 - 2*x + 1)
(2*x + 2, x - 1)
```

Polynomial factorization in characteristic zero:

`sympy.polys.factortools.dmp_trial_division(f, factors, u, K)`  
Determine multiplicities of factors for a multivariate polynomial using trial division.

`sympy.polys.factortools.dmp_zz_mignotte_bound(f, u, K)`  
Mignotte bound for multivariate polynomials in  $K[X]$ .

`sympy.polys.factortools.dup_zz_hensel_step(m, f, g, h, s, t, K)`  
One step in Hensel lifting in  $Z[x]$ .  
Given positive integer  $m$  and  $Z[x]$  polynomials  $f, g, h, s$  and  $t$  such that:

```
f = g*h (mod m)
s*g + t*h = 1 (mod m)

lc(f) is not a zero divisor (mod m)
lc(h) = 1

deg(f) = deg(g) + deg(h)
deg(s) < deg(h)
deg(t) < deg(g)
```

returns polynomials  $G, H, S$  and  $T$ , such that:

```
f = G*H (mod m**2)
S*G + T*H = 1 (mod m**2)
```

## References

[R705]

`sympy.polys.factortools.dup_zz_hensel_lift(p, f, f_list, l, K)`

Multifactor Hensel lifting in  $Z[x]$ .

Given a prime  $p$ , polynomial  $f$  over  $Z[x]$  such that  $lc(f)$  is a unit modulo  $p$ , monic pair-wise coprime polynomials  $f_i$  over  $Z[x]$  satisfying:

$$f = lc(f) f_1 \dots f_r \pmod{p}$$

and a positive integer  $l$ , returns a list of monic polynomials  $F_1, F_2, \dots, F_r$  satisfying:

$$f = lc(f) F_1 \dots F_r \pmod{p^{*l}}$$

$$F_i = f_i \pmod{p}, i = 1..r$$

## References

[R706]

`sympy.polys.factortools.dup_zz_zassenhaus(f, K)`

Factor primitive square-free polynomials in  $Z[x]$ .

`sympy.polys.factortools.dup_zz_irreducible_p(f, K)`

Test irreducibility using Eisenstein's criterion.

`sympy.polys.factortools.dup_cyclotomic_p(f, K, irreducible=False)`

Efficiently test if  $f$  is a cyclotomic polynomial.

## Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> f = x**16 + x**14 - x**10 + x**8 - x**6 + x**2 + 1
>>> R.dup_cyclotomic_p(f)
False
```

```
>>> g = x**16 + x**14 - x**10 - x**8 - x**6 + x**2 + 1
>>> R.dup_cyclotomic_p(g)
True
```

## References

Bradford, Russell J., and James H. Davenport. "Effective tests for cyclotomic polynomials." In International Symposium on Symbolic and Algebraic Computation, pp. 244-251. Springer, Berlin, Heidelberg, 1988.

`sympy.polys.factortools.dup_zz_cyclotomic_poly(n, K)`

Efficiently generate  $n$ -th cyclotomic polynomial.

`sympy.polys.factortools.dup_zz_cyclotomic_factor(f, K)`

Efficiently factor polynomials  $x^{**n} - 1$  and  $x^{**n} + 1$  in  $Z[x]$ .

Given a univariate polynomial  $f$  in  $Z[x]$  returns a list of factors of  $f$ , provided that  $f$  is in the form  $x^{**n} - 1$  or  $x^{**n} + 1$  for  $n \geq 1$ . Otherwise returns None.

Factorization is performed using cyclotomic decomposition of  $f$ , which makes this method much faster than any other direct factorization approach (e.g. Zassenhaus's).

## References

[R707]

`sympy.polys.factortools.dup_zz_factor_sqf(f, K)`

Factor square-free (non-primitive) polynomials in  $Z[x]$ .

`sympy.polys.factortools.dup_zz_factor(f, K)`

Factor (non square-free) polynomials in  $Z[x]$ .

Given a univariate polynomial  $f$  in  $Z[x]$  computes its complete factorization  $f_1, \dots, f_n$  into irreducibles over integers:

```
f = content(f) f_1**k_1 ... f_n**k_n
```

The factorization is computed by reducing the input polynomial into a primitive square-free polynomial and factoring it using Zassenhaus algorithm. Trial division is used to recover the multiplicities of factors.

The result is returned as a tuple consisting of:

```
(content(f), [(f_1, k_1), ..., (f_n, k_n)])
```

## Examples

Consider the polynomial  $f = 2 * x^{**4} - 2$ :

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)

>>> R.dup_zz_factor(2*x**4 - 2)
(2, [(x - 1, 1), (x + 1, 1), (x**2 + 1, 1)])
```

In result we got the following factorization:

```
f = 2 (x - 1) (x + 1) (x**2 + 1)
```

Note that this is a complete factorization over integers, however over Gaussian integers we can factor the last term.

By default, polynomials  $x^{**n}-1$  and  $x^{**n}+1$  are factored using cyclotomic decomposition to speedup computations. To disable this behaviour set `cyclotomic=False`.

## References

[R708]

`sympy.polys.factortools.dmp_zz_wang_non_divisors(E, cs, ct, K)`

Wang/EEZ: Compute a set of valid divisors.

`sympy.polys.factortools.dmp_zz_wang_test_points(f, T, ct, A, u, K)`

Wang/EEZ: Test evaluation points for suitability.

`sympy.polys.factortools.dmp_zz_wang_lead_coeffs(f, T, cs, E, H, A, u, K)`

Wang/EEZ: Compute correct leading coefficients.

`sympy.polys.factortools.dmp_zz_diophantine(F, c, A, d, p, u, K)`

Wang/EEZ: Solve multivariate Diophantine equations.

`sympy.polys.factortools.dmp_zz_wang_hensel_lifting(f, H, LC, A, p, u, K)`

Wang/EEZ: Parallel Hensel lifting algorithm.

`sympy.polys.factortools.dmp_zz_wang(f, u, K, mod=None, seed=None)`

Factor primitive square-free polynomials in  $Z[X]$ .

Given a multivariate polynomial  $f$  in  $Z[x_1, \dots, x_n]$ , which is primitive and square-free in  $x_1$ , computes factorization of  $f$  into irreducibles over integers.

The procedure is based on Wang's Enhanced Extended Zassenhaus algorithm. The algorithm works by viewing  $f$  as a univariate polynomial in  $Z[x_2, \dots, x_n][x_1]$ , for which an evaluation mapping is computed:

$$x_2 \rightarrow a_2, \dots, x_n \rightarrow a_n$$

where  $a_i$ , for  $i = 2, \dots, n$ , are carefully chosen integers. The mapping is used to transform  $f$  into a univariate polynomial in  $Z[x_1]$ , which can be factored efficiently using Zassenhaus algorithm. The last step is to lift univariate factors to obtain true multivariate factors. For this purpose a parallel Hensel lifting procedure is used.

The parameter `seed` is passed to `_randint` and can be used to seed `randint` (when an integer) or (for testing purposes) can be a sequence of numbers.

## References

[R709], [R710]

`sympy.polys.factortools.dmp_zz_factor(f, u, K)`

Factor (non square-free) polynomials in  $Z[X]$ .

Given a multivariate polynomial  $f$  in  $Z[x]$  computes its complete factorization  $f_1, \dots, f_n$  into irreducibles over integers:

```
f = content(f) f_1**k_1 ... f_n**k_n
```

The factorization is computed by reducing the input polynomial into a primitive square-free polynomial and factoring it using Enhanced Extended Zassenhaus (EEZ) algorithm. Trial division is used to recover the multiplicities of factors.

The result is returned as a tuple consisting of:

```
(content(f), [(f_1, k_1), ..., (f_n, k_n)])
```

Consider polynomial  $f = 2 * (x ** 2 - y ** 2)$ :

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> R.dmp_zz_factor(2*x**2 - 2*y**2)
(2, [(x - y, 1), (x + y, 1)])
```

In result we got the following factorization:

```
f = 2 (x - y) (x + y)
```

## References

[R711]

`sympy.polys.factortools.dmp_ext_factor(f, u, K)`

Factor multivariate polynomials over algebraic number fields.

`sympy.polys.factortools.dup_gf_factor(f, K)`

Factor univariate polynomials over finite fields.

`sympy.polys.factortools.dmp_factor_list(f, u, K0)`

Factor multivariate polynomials into irreducibles in  $K[X]$ .

`sympy.polys.factortools.dmp_factor_list_include(f, u, K)`

Factor multivariate polynomials into irreducibles in  $K[X]$ .

`sympy.polys.factortools.dmp_irreducible_p(f, u, K)`

Returns True if a multivariate polynomial  $f$  has no factors over its domain.

## Groebner basis algorithms

Groebner bases can be used to answer many problems in computational commutative algebra. Their computation is rather complicated, and very performance-sensitive. We present here various low-level implementations of Groebner basis computation algorithms; please see the previous section of the manual for usage.

`sympy.polys.groebnertools.groebner(seq, ring, method=None)`

Computes Groebner basis for a set of polynomials in  $K[X]$ .

Wrapper around the (default) improved Buchberger and the other algorithms for computing Groebner bases. The choice of algorithm can be changed via method argument or `sympy.polys.polyconfig.setup()` (page 2643), where method can be either buchberger or f5b.

`sympy.polys.groebnertools.spoly(p1, p2, ring)`

Compute  $\text{LCM}(\text{LM}(p1), \text{LM}(p2))/\text{LM}(p1)*p1 - \text{LCM}(\text{LM}(p1), \text{LM}(p2))/\text{LM}(p2)*p2$  This is the S-poly provided p1 and p2 are monic

`sympy.polys.groebnertools.red_groebner(G, ring)`

Compute reduced Groebner basis, from BeckerWeispfenning93, p. 216

Selects a subset of generators, that already generate the ideal and computes a reduced Groebner basis for them.

`sympy.polys.groebnertools.is_groebner(G, ring)`

Check if G is a Groebner basis.

`sympy.polys.groebnertools.is_minimal(G, ring)`

Checks if G is a minimal Groebner basis.

`sympy.polys.groebnertools.is_reduced(G, ring)`

Checks if G is a reduced Groebner basis.

`sympy.polys.fglmtools.matrix_fgfm(F, ring, O_to)`

Converts the reduced Groebner basis F of a zero-dimensional ideal w.r.t.  $O_{\text{from}}$  to a reduced Groebner basis w.r.t.  $O_{\text{to}}$ .

## References

[R712]

Groebner basis algorithms for modules are also provided:

`sympy.polys.distributedmodules.sdm_spoly(f, g, O, K, phantom=None)`

Compute the generalized s-polynomial of f and g.

The ground field is assumed to be K, and monomials ordered according to O.

This is invalid if either of f or g is zero.

If the leading terms of f and g involve different basis elements of F, their s-poly is defined to be zero. Otherwise it is a certain linear combination of f and g in which the leading terms cancel. See [SCA, defn 2.3.6] for details.

If phantom is not None, it should be a pair of module elements on which to perform the same operation(s) as on f and g. The in this case both results are returned.



## Examples

```
>>> from sympy.polys.distributedmodules import sdm_spoly
>>> from sympy.polys import QQ, lex
>>> f = [((2, 1, 1), QQ(1)), ((1, 0, 1), QQ(1))]
>>> g = [((2, 3, 0), QQ(1))]
>>> h = [((1, 2, 3), QQ(1))]
>>> sdm_spoly(f, h, lex, QQ)
[]
>>> sdm_spoly(f, g, lex, QQ)
[((1, 2, 1), 1)]
```

`sympy.polys.distributedmodules.sdm_ecart(f)`

Compute the ecart of  $f$ .

This is defined to be the difference of the total degree of  $f$  and the total degree of the leading monomial of  $f$  [SCA, defn 2.3.7].

Invalid if  $f$  is zero.

## Examples

```
>>> from sympy.polys.distributedmodules import sdm_ecart
>>> sdm_ecart([((1, 2, 3), 1), ((1, 0, 1), 1)])
0
>>> sdm_ecart([((2, 2, 1), 1), ((1, 5, 1), 1)])
3
```

`sympy.polys.distributedmodules.sdm_nf_mora(f, G, O, K, phantom=None)`

Compute a weak normal form of  $f$  with respect to  $G$  and order 0.

The ground field is assumed to be  $K$ , and monomials ordered according to 0.

Weak normal forms are defined in [SCA, defn 2.3.3]. They are not unique. This function deterministically computes a weak normal form, depending on the order of  $G$ .

The most important property of a weak normal form is the following: if  $R$  is the ring associated with the monomial ordering (if the ordering is global, we just have  $R = K[x_1, \dots, x_n]$ , otherwise it is a certain localization thereof),  $I$  any ideal of  $R$  and  $G$  a standard basis for  $I$ , then for any  $f \in R$ , we have  $f \in I$  if and only if  $NF(f|G) = 0$ .

This is the generalized Mora algorithm for computing weak normal forms with respect to arbitrary monomial orders [SCA, algorithm 2.3.9].

If `phantom` is not `None`, it should be a pair of “phantom” arguments on which to perform the same computations as on  $f$ ,  $G$ , both results are then returned.

`sympy.polys.distributedmodules.sdm_groebner(G, NF, O, K, extended=False)`

Compute a minimal standard basis of  $G$  with respect to order 0.

The algorithm uses a normal form  $NF$ , for example `sdm_nf_mora`. The ground field is assumed to be  $K$ , and monomials ordered according to 0.

Let  $N$  denote the submodule generated by elements of  $G$ . A standard basis for  $N$  is a subset  $S$  of  $N$ , such that  $\text{in}(S) = \text{in}(N)$ , where for any subset  $X$  of  $F$ ,  $\text{in}(X)$  denotes the submodule generated by the initial forms of elements of  $X$ . [SCA, defn 2.3.2]

A standard basis is called minimal if no subset of it is a standard basis.

One may show that standard bases are always generating sets.

Minimal standard bases are not unique. This algorithm computes a deterministic result, depending on the particular order of  $G$ .

If `extended=True`, also compute the transition matrix from the initial generators to the groebner basis. That is, return a list of coefficient vectors, expressing the elements of the groebner basis in terms of the elements of  $G$ .

This functions implements the “sugar” strategy, see

Giovini et al: “One sugar cube, please” OR Selection strategies in Buchberger algorithm.

## Options

Options manager for *Poly* (page 2378) and public API functions.

**class** `sympy.polys.polyoptions.Options`(*gens*, *args*, *flags=None*, *strict=False*)

Options manager for polynomial manipulation module.

## Examples

```
>>> from sympy.polys.polyoptions import Options
>>> from sympy.polys.polyoptions import build_options
```

```
>>> from sympy.abc import x, y, z
```

```
>>> Options((x, y, z), {'domain': 'ZZ'})
{'auto': False, 'domain': ZZ, 'gens': (x, y, z)}
```

```
>>> build_options((x, y, z), {'domain': 'ZZ'})
{'auto': False, 'domain': ZZ, 'gens': (x, y, z)}
```

## Options

- Expand — boolean option
- Gens — option
- Wrt — option
- Sort — option
- Order — option
- Field — boolean option
- Greedy — boolean option
- Domain — option
- Split — boolean option
- Gaussian — boolean option
- Extension — option

- Modulus — option
- Symmetric — boolean option
- Strict — boolean option

**Flags**

- Auto — boolean flag
- Frac — boolean flag
- Formal — boolean flag
- Polys — boolean flag
- Include — boolean flag
- All — boolean flag
- Gen — flag
- Series — boolean flag

**clone**(*updates*={})

Clone self and update specified options.

`sympy.polys.polyoptions.build_options(gens, args=None)`

Construct options from keyword arguments or ... options.

**Configuration**

Configuration utilities for polynomial manipulation algorithms.

`sympy.polys.polyconfig.setup(key, value=None)`

Assign a value to (or reset) a configuration item.

**Exceptions**

These are exceptions defined by the polynomials module.

TODO sort and explain

**class** `sympy.polys.polyerrors.BasePolynomialError`

Base class for polynomial related exceptions.

**class** `sympy.polys.polyerrors.ExactQuotientFailed(f, g, dom=None)`

**class** `sympy.polys.polyerrors.OperationNotSupported(poly, func)`

**class** `sympy.polys.polyerrors.HeuristicGCDFailed`

**class** `sympy.polys.polyerrors.HomomorphismFailed`

**class** `sympy.polys.polyerrors.IsomorphismFailed`

**class** `sympy.polys.polyerrors.ExtraneousFactors`

**class** `sympy.polys.polyerrors.EvaluationFailed`

```
class sympy.polys.polyerrors.RefinementFailed
class sympy.polys.polyerrors.CoercionFailed
class sympy.polys.polyerrors.NotInvertible
class sympy.polys.polyerrors.NotReversible
class sympy.polys.polyerrors.NotAlgebraic
class sympy.polys.polyerrors.DomainError
class sympy.polys.polyerrors.PolynomialError
class sympy.polys.polyerrors.UnificationFailed
class sympy.polys.polyerrors.GeneratorsNeeded
class sympy.polys.polyerrors.ComputationFailed(func, nargs, exc)
class sympy.polys.polyerrors.GeneratorsError
class sympy.polys.polyerrors.UnivariatePolynomialError
class sympy.polys.polyerrors.MultivariatePolynomialError
class sympy.polys.polyerrors.PolificationFailed(opt, origs, exprs, seq=False)
class sympy.polys.polyerrors.OptionError
class sympy.polys.polyerrors.FlagError
```

## Reference

### Modular GCD

`sympy.polys.modulargcd.modgcd_univariate(f, g)`

Computes the GCD of two polynomials in  $\mathbb{Z}[x]$  using a modular algorithm.

The algorithm computes the GCD of two univariate integer polynomials  $f$  and  $g$  by computing the GCD in  $\mathbb{Z}_p[x]$  for suitable primes  $p$  and then reconstructing the coefficients with the Chinese Remainder Theorem. Trial division is only made for candidates which are very likely the desired GCD.

#### Parameters

**f** : PolyElement

univariate integer polynomial

**g** : PolyElement

univariate integer polynomial

#### Returns

**h** : PolyElement

GCD of the polynomials  $f$  and  $g$

**cff** : PolyElement

cofactor of  $f$ , i.e.  $\frac{f}{h}$   
**cfg** : PolyElement  
 cofactor of  $g$ , i.e.  $\frac{g}{h}$

## Examples

```
>>> from sympy.polys.modulargcd import modgcd_univariate
>>> from sympy.polys import ring, ZZ
```

```
>>> R, x = ring("x", ZZ)
```

```
>>> f = x**5 - 1
>>> g = x - 1
```

```
>>> h, cff, cfg = modgcd_univariate(f, g)
>>> h, cff, cfg
(x - 1, x**4 + x**3 + x**2 + x + 1, 1)
```

```
>>> cff * h == f
True
>>> cfg * h == g
True
```

```
>>> f = 6*x**2 - 6
>>> g = 2*x**2 + 4*x + 2
```

```
>>> h, cff, cfg = modgcd_univariate(f, g)
>>> h, cff, cfg
(2*x + 2, 3*x - 3, x + 1)
```

```
>>> cff * h == f
True
>>> cfg * h == g
True
```

## References

1. [Monagan00]

[Monagan00]

`sympy.polys.modulargcd.modgcd_bivariate( $f, g$ )`

Computes the GCD of two polynomials in  $\mathbb{Z}[x, y]$  using a modular algorithm.

The algorithm computes the GCD of two bivariate integer polynomials  $f$  and  $g$  by calculating the GCD in  $\mathbb{Z}_p[x, y]$  for suitable primes  $p$  and then reconstructing the coefficients with the Chinese Remainder Theorem. To compute the bivariate GCD over  $\mathbb{Z}_p$ , the polynomials  $f \bmod p$  and  $g \bmod p$  are evaluated at  $y = a$  for certain  $a \in \mathbb{Z}_p$  and then their univariate GCD in  $\mathbb{Z}_p[x]$  is computed. Interpolating those yields the bivariate GCD in

$\mathbb{Z}_p[x, y]$ . To verify the result in  $\mathbb{Z}[x, y]$ , trial division is done, but only for candidates which are very likely the desired GCD.

**Parameters**

**f** : PolyElement  
bivariate integer polynomial

**g** : PolyElement  
bivariate integer polynomial

**Returns**

**h** : PolyElement  
GCD of the polynomials  $f$  and  $g$

**cff** : PolyElement  
cofactor of  $f$ , i.e.  $\frac{f}{h}$

**cfg** : PolyElement  
cofactor of  $g$ , i.e.  $\frac{g}{h}$

**Examples**

```
>>> from sympy.polys.modulargcd import modgcd_bivariate
>>> from sympy.polys import ring, ZZ
```

```
>>> R, x, y = ring("x, y", ZZ)
```

```
>>> f = x**2 - y**2
>>> g = x**2 + 2*x*y + y**2
```

```
>>> h, cff, cfg = modgcd_bivariate(f, g)
>>> h, cff, cfg
(x + y, x - y, x + y)
```

```
>>> cff * h == f
True
>>> cfg * h == g
True
```

```
>>> f = x**2*y - x**2 - 4*y + 4
>>> g = x + 2
```

```
>>> h, cff, cfg = modgcd_bivariate(f, g)
>>> h, cff, cfg
(x + 2, x*y - x - 2*y + 2, 1)
```

```
>>> cff * h == f
True
>>> cfg * h == g
True
```

## References

1. [Monagan00]

[Monagan00]

`sympy.polys.modulargcd.modgcd_multivariate(f, g)`

Compute the GCD of two polynomials in  $\mathbb{Z}[x_0, \dots, x_{k-1}]$  using a modular algorithm.

The algorithm computes the GCD of two multivariate integer polynomials  $f$  and  $g$  by calculating the GCD in  $\mathbb{Z}_p[x_0, \dots, x_{k-1}]$  for suitable primes  $p$  and then reconstructing the coefficients with the Chinese Remainder Theorem. To compute the multivariate GCD over  $\mathbb{Z}_p$  the recursive subroutine `_modgcd_multivariate_p()` (page 2648) is used. To verify the result in  $\mathbb{Z}[x_0, \dots, x_{k-1}]$ , trial division is done, but only for candidates which are very likely the desired GCD.

### Parameters

**f** : PolyElement

multivariate integer polynomial

**g** : PolyElement

multivariate integer polynomial

### Returns

**h** : PolyElement

GCD of the polynomials  $f$  and  $g$

**cff** : PolyElement

cofactor of  $f$ , i.e.  $\frac{f}{h}$

**cfg** : PolyElement

cofactor of  $g$ , i.e.  $\frac{g}{h}$

## Examples

```
>>> from sympy.polys.modulargcd import modgcd_multivariate
>>> from sympy.polys import ring, ZZ
```

```
>>> R, x, y = ring("x, y", ZZ)
```

```
>>> f = x**2 - y**2
>>> g = x**2 + 2*x*y + y**2
```

```
>>> h, cff, cfg = modgcd_multivariate(f, g)
>>> h, cff, cfg
(x + y, x - y, x + y)
```

```
>>> cff * h == f
True
>>> cfg * h == g
True
```

```
>>> R, x, y, z = ring("x, y, z", ZZ)
```

```
>>> f = x*z**2 - y*z**2
>>> g = x**2*z + z
```

```
>>> h, cff, cfg = modgcd_multivariate(f, g)
>>> h, cff, cfg
(z, x*z - y*z, x**2 + 1)
```

```
>>> cff * h == f
True
>>> cfg * h == g
True
```

**See also:**

[`\_modgcd\_multivariate\_p`](#) (page 2648)

## References

1. [Monagan00]
2. [Brown71]

[Monagan00], [Brown71]

`sympy.polys.modulargcd._modgcd_multivariate_p(f, g, p, degbound, contbound)`

Compute the GCD of two polynomials in  $\mathbb{Z}_p[x_0, \dots, x_{k-1}]$ .

The algorithm reduces the problem step by step by evaluating the polynomials  $f$  and  $g$  at  $x_{k-1} = a$  for suitable  $a \in \mathbb{Z}_p$  and then calls itself recursively to compute the GCD in  $\mathbb{Z}_p[x_0, \dots, x_{k-2}]$ . If these recursive calls are successful for enough evaluation points, the GCD in  $k$  variables is interpolated, otherwise the algorithm returns `None`. Every time a GCD or a content is computed, their degrees are compared with the bounds. If a degree greater than the bound is encountered, then the current call returns `None` and a new evaluation point has to be chosen. If at some point the degree is smaller, the correspondent bound is updated and the algorithm fails.

### Parameters

**f** : PolyElement

multivariate integer polynomial with coefficients in  $\mathbb{Z}_p$

**g** : PolyElement

multivariate integer polynomial with coefficients in  $\mathbb{Z}_p$

**p** : Integer

prime number, modulus of  $f$  and  $g$

**degbound** : list of Integer objects

`degbound[i]` is an upper bound for the degree of the GCD of  $f$  and  $g$  in the variable  $x_i$

**contbound** : list of Integer objects



`contbound[i]` is an upper bound for the degree of the content of the GCD in  $\mathbb{Z}_p[x_i][x_0, \dots, x_{i-1}]$ , `contbound[0]` is not used can therefore be chosen arbitrarily.

### Returns

**h** : PolyElement

GCD of the polynomials  $f$  and  $g$  or None

### References

1. [Monagan00]
2. [Brown71]

[Monagan00], [Brown71]

`sympy.polys.modulargcd.func_field_modgcd(f, g)`

Compute the GCD of two polynomials  $f$  and  $g$  in  $\mathbb{Q}(\alpha)[x_0, \dots, x_{n-1}]$  using a modular algorithm.

The algorithm first computes the primitive associate  $\check{m}_\alpha(z)$  of the minimal polynomial  $m_\alpha$  in  $\mathbb{Z}[z]$  and the primitive associates of  $f$  and  $g$  in  $\mathbb{Z}[x_1, \dots, x_{n-1}][z]/(\check{m}_\alpha)[x_0]$ . Then it computes the GCD in  $\mathbb{Q}(x_1, \dots, x_{n-1})[z]/(m_\alpha(z))[x_0]$ . This is done by calculating the GCD in  $\mathbb{Z}_p(x_1, \dots, x_{n-1})[z]/(\check{m}_\alpha(z))[x_0]$  for suitable primes  $p$  and then reconstructing the coefficients with the Chinese Remainder Theorem and Rational Reconstruction. The GCD over  $\mathbb{Z}_p(x_1, \dots, x_{n-1})[z]/(\check{m}_\alpha(z))[x_0]$  is computed with a recursive subroutine, which evaluates the polynomials at  $x_{n-1} = a$  for suitable evaluation points  $a \in \mathbb{Z}_p$  and then calls itself recursively until the ground domain does no longer contain any parameters. For  $\mathbb{Z}_p[z]/(\check{m}_\alpha(z))[x_0]$  the Euclidean Algorithm is used. The results of those recursive calls are then interpolated and Rational Function Reconstruction is used to obtain the correct coefficients. The results, both in  $\mathbb{Q}(x_1, \dots, x_{n-1})[z]/(m_\alpha(z))[x_0]$  and  $\mathbb{Z}_p(x_1, \dots, x_{n-1})[z]/(\check{m}_\alpha(z))[x_0]$ , are verified by a fraction free trial division.

Apart from the above GCD computation some GCDs in  $\mathbb{Q}(\alpha)[x_1, \dots, x_{n-1}]$  have to be calculated, because treating the polynomials as univariate ones can result in a spurious content of the GCD. For this `func_field_modgcd` is called recursively.

### Parameters

**f, g** : PolyElement

polynomials in  $\mathbb{Q}(\alpha)[x_0, \dots, x_{n-1}]$

### Returns

**h** : PolyElement

monic GCD of the polynomials  $f$  and  $g$

**cff** : PolyElement

cofactor of  $f$ , i.e.  $\frac{f}{h}$

**cfg** : PolyElement

cofactor of  $g$ , i.e.  $\frac{g}{h}$

## Examples

```
>>> from sympy.polys.modulargcd import func_field_modgcd
>>> from sympy.polys import AlgebraicField, QQ, ring
>>> from sympy import sqrt
```

```
>>> A = AlgebraicField(QQ, sqrt(2))
>>> R, x = ring('x', A)
```

```
>>> f = x**2 - 2
>>> g = x + sqrt(2)
```

```
>>> h, cff, cfg = func_field_modgcd(f, g)
```

```
>>> h == x + sqrt(2)
True
>>> cff * h == f
True
>>> cfg * h == g
True
```

```
>>> R, x, y = ring('x, y', A)
```

```
>>> f = x**2 + 2*sqrt(2)*x*y + 2*y**2
>>> g = x + sqrt(2)*y
```

```
>>> h, cff, cfg = func_field_modgcd(f, g)
```

```
>>> h == x + sqrt(2)*y
True
>>> cff * h == f
True
>>> cfg * h == g
True
```

```
>>> f = x + sqrt(2)*y
>>> g = x + y
```

```
>>> h, cff, cfg = func_field_modgcd(f, g)
```

```
>>> h == R.one
True
>>> cff * h == f
True
>>> cfg * h == g
True
```

## References

1. [Hoeij04]

[Hoeij04]

## Undocumented

Many parts of the polys module are still undocumented, and even where there is documentation it is scarce. Please contribute!

## Series Manipulation using Polynomials

Any finite Taylor series, for all practical purposes is, in fact a polynomial. This module makes use of the efficient representation and operations of sparse polynomials for very fast multi-variate series manipulations. Typical speedups compared to SymPy's `series` method are in the range 20-100, with the gap widening as the series being handled gets larger.

All the functions expand any given series on some ring specified by the user. Thus, the coefficients of the calculated series depend on the ring being used. For example:

```
>>> from sympy.polys import ring, QQ, RR
>>> from sympy.polys.ring_series import rs_sin
>>> R, x, y = ring('x, y', QQ)
>>> rs_sin(x*y, x, 5)
-1/6*x**3*y**3 + x*y
```

QQ stands for the Rational domain. Here all coefficients are rationals. It is recommended to use QQ with ring series as it automatically chooses the fastest Rational type.

Similarly, if a Real domain is used:

```
>>> R, x, y = ring('x, y', RR)
>>> rs_sin(x*y, x, 5)
-0.166666666666667*x**3*y**3 + x*y
```

Though the definition of a polynomial limits the use of Polynomial module to Taylor series, we extend it to allow Laurent and even Puiseux series (with fractional exponents):

```
>>> from sympy.polys.ring_series import rs_cos, rs_tan
>>> R, x, y = ring('x, y', QQ)

>>> rs_cos(x + x*y, x, 3)/x**3
-1/2*x**(-1)*y**2 - x**(-1)*y - 1/2*x**(-1) + x**(-3)

>>> rs_tan(x**QQ(2, 5)*y**QQ(1, 2), x, 2)
1/3*x**(6/5)*y**(3/2) + x**(2/5)*y**(1/2)
```

By default, PolyElement did not allow non-natural numbers as exponents. It converted a fraction to an integer and raised an error on getting negative exponents. The goal of the ring series module is fast series expansion, and not to use the polys module. The reason we use it as our backend is simply because it implements a sparse representation and most of the

basic functions that we need. However, this default behaviour of polys was limiting for ring series.

Note that there is no such constraint (in having rational exponents) in the data-structure used by polys- dict. Sparse polynomials (PolyElement) use the Python dict to store a polynomial term by term, where a tuple of exponents is the key and the coefficient of that term is the value. There is no reason why we can't have rational values in the dict so as to support rational exponents.

So the approach we took was to modify sparse polys to allow non-natural exponents. And it turned out to be quite simple. We only had to delete the conversion to int of exponents in the `__pow__` method of PolyElement. So:

```
>>> x**QQ(3, 4)
x**(3/4)
```

and not 1 as was the case earlier.

Though this change violates the definition of a polynomial, it doesn't break anything yet. Ideally, we shouldn't modify polys in any way. But to have all the series capabilities we want, no other simple way was found. If need be, we can separate the modified part of polys from core polys. It would be great if any other elegant solution is found.

All series returned by the functions of this module are instances of the PolyElement class. To use them with other SymPy types, convert them to Expr:

```
>>> from sympy.polys.ring_series import rs_exp
>>> from sympy.abc import a, b, c
>>> series = rs_exp(x, x, 5)
>>> a + series.as_expr()
a + x**4/24 + x**3/6 + x**2/2 + x + 1
```

## rs\_series

Direct use of elementary ring series functions does give more control, but is limiting at the same time. Creating an appropriate ring for the desired series expansion and knowing which ring series function to call, are things not everyone might be familiar with.

*rs\_series* is a function that takes an arbitrary Expr and returns its expansion by calling the appropriate ring series functions. The returned series is a polynomial over the simplest (almost) possible ring that does the job. It recursively builds the ring as it parses the given expression, adding generators to the ring when it needs them. Some examples:

```
>>> from sympy.polys.ring_series import rs_series
>>> from sympy.functions.elementary.trigonometric import sin
>>> rs_series(sin(a + b), a, 5)
1/24*sin(b)*a**4 - 1/2*sin(b)*a**2 + sin(b) - 1/6*cos(b)*a**3 + cos(b)*a

>>> rs_series(sin(exp(a*b) + cos(a + c)), a, 2)
-sin(c)*cos(cos(c) + 1)*a + cos(cos(c) + 1)*a*b + sin(cos(c) + 1)

>>> rs_series(sin(a + b)*cos(a + c)*tan(a**2 + b), a, 2)
cos(b)*cos(c)*tan(b)*a - sin(b)*sin(c)*tan(b)*a + sin(b)*cos(c)*tan(b)
```

It can expand complicated multivariate expressions involving multiple functions and most importantly, it does so blazingly fast:

```
>>> %timeit ((sin(a) + cos(a))**10).series(a, 0, 5)
1 loops, best of 3: 1.33 s per loop

>>> %timeit rs_series((sin(a) + cos(a))**10, a, 5)
100 loops, best of 3: 4.13 ms per loop
```

*rs\_series* is over 300 times faster. Given an expression to expand, there is some fixed overhead to parse it. Thus, for larger orders, the speed improvement becomes more prominent:

```
>>> %timeit rs_series((sin(a) + cos(a))**10, a, 100)
10 loops, best of 3: 32.8 ms per loop
```

To figure out the right ring for a given expression, *rs\_series* uses the *sring* function, which in turn uses other functions of *polys*. As explained above, non-natural exponents are not allowed. But the restriction is on exponents and not generators. So, *polys* allows all sorts of symbolic terms as generators to make sure that the exponent is a natural number:

```
>>> from sympy.polys.rings import sring
>>> R, expr = sring(1/a**3 + a**QQ(3, 7)); R
Polynomial ring in 1/a, a**(1/7) over ZZ with lex order
```

In the above example,  $1/a$  and  $a^{1/7}$  will be treated as completely different atoms. For all practical purposes, we could let  $b = 1/a$  and  $c = a^{1/7}$  and do the manipulations. Effectively, expressions involving  $1/a$  and  $a^{1/7}$  (and their powers) will never simplify:

```
>>> expr*R(1/a)
(1/a)**4 + (1/a)*(a**(1/7))**3
```

This leads to similar issues with manipulating Laurent and Puiseux series as faced earlier. Fortunately, this time we have an elegant solution and are able to isolate the *series* and *polys* behaviour from one another. We introduce a boolean flag *series* in the list of allowed Options for polynomials (see [sympy.polys.polyoptions.Options](#) (page 2642)). Thus, when we want *sring* to allow rational exponents we supply a *series=True* flag to *sring*:

```
>>> rs_series(sin(a**QQ(1, 3)), a, 3)
-1/5040*a**(7/3) + 1/120*a**(5/3) - 1/6*a + a**(1/3)
```

## Contribute

*rs\_series* is not fully implemented yet. As of now, it supports only multivariate Taylor expansions of expressions involving *sin*, *cos*, *exp* and *tan*. Adding the remaining functions is not at all difficult and they will be gradually added. If you are interested in helping, read the comments in *ring\_series.py*. Currently, it does not support Puiseux series (though the elementary functions do). This is expected to be fixed soon.

You can also add more functions to *ring\_series.py*. Only elementary functions are supported currently. The long term goal is to replace SymPy's current *series* method with *rs\_series*.

## Manipulation of power series

Functions in this module carry the prefix `rs_`, standing for “ring series”. They manipulate finite power series in the sparse representation provided by `polys.ring.ring`.

### Elementary functions

`sympy.polys.ring_series.rs_log(p, x, prec)`

The Logarithm of  $p$  modulo  $O(x^{**prec})$ .

### Notes

Truncation of integral  $\int x^{p-1} dp/dx$  is used.

### Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_log
>>> R, x = ring('x', QQ)
>>> rs_log(1 + x, x, 8)
1/7*x**7 - 1/6*x**6 + 1/5*x**5 - 1/4*x**4 + 1/3*x**3 - 1/2*x**2 + x
>>> rs_log(x**QQ(3, 2) + 1, x, 5)
1/3*x**(9/2) - 1/2*x**3 + x**(3/2)
```

`sympy.polys.ring_series.rs_LambertW(p, x, prec)`

Calculate the series expansion of the principal branch of the Lambert W function.

### Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_LambertW
>>> R, x, y = ring('x, y', QQ)
>>> rs_LambertW(x + x*y, x, 3)
-x**2*y**2 - 2*x**2*y - x**2 + x*y + x
```

### See also:

[\*LambertW\*](#) (page 412)

`sympy.polys.ring_series.rs_exp(p, x, prec)`

Exponentiation of a series modulo  $O(x^{**prec})$

## Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_exp
>>> R, x = ring('x', QQ)
>>> rs_exp(x**2, x, 7)
1/6*x**6 + 1/2*x**4 + x**2 + 1
```

`sympy.polys.ring_series.rs_atan(p, x, prec)`

The arctangent of a series

Return the series expansion of the atan of p, about 0.

## Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_atan
>>> R, x, y = ring('x, y', QQ)
>>> rs_atan(x + x*y, x, 4)
-1/3*x**3*y**3 - x**3*y**2 - x**3*y - 1/3*x**3 + x*y + x
```

### See also:

[atan](#) (page 397)

`sympy.polys.ring_series.rs_asin(p, x, prec)`

Arcsine of a series

Return the series expansion of the asin of p, about 0.

## Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_asin
>>> R, x, y = ring('x, y', QQ)
>>> rs_asin(x, x, 8)
5/112*x**7 + 3/40*x**5 + 1/6*x**3 + x
```

### See also:

[asin](#) (page 395)

`sympy.polys.ring_series.rs_tan(p, x, prec)`

Tangent of a series.

Return the series expansion of the tan of p, about 0.

## Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_tan
>>> R, x, y = ring('x, y', QQ)
>>> rs_tan(x + x*y, x, 4)
1/3*x**3*y**3 + x**3*y**2 + x**3*y + 1/3*x**3 + x*y + x
```

### See also:

[`\_tan1`](#) (page 2656), [`tan`](#) (page 391)

`sympy.polys.ring_series._tan1(p, x, prec)`

Helper function of `rs_tan()` (page 2655).

Return the series expansion of tan of a univariate series using Newton's method. It takes advantage of the fact that series expansion of atan is easier than that of tan.

Consider  $f(x) = y - \arctan(x)$  Let  $r$  be a root of  $f(x)$  found using Newton's method. Then  $f(r) = 0$  Or  $y = \arctan(x)$  where  $x = \tan(y)$  as required.

`sympy.polys.ring_series.rs_cot(p, x, prec)`

Cotangent of a series

Return the series expansion of the cot of  $p$ , about 0.

## Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_cot
>>> R, x, y = ring('x, y', QQ)
>>> rs_cot(x, x, 6)
-2/945*x**5 - 1/45*x**3 - 1/3*x + x**(-1)
```

### See also:

[`cot`](#) (page 392)

`sympy.polys.ring_series.rs_sin(p, x, prec)`

Sine of a series

Return the series expansion of the sin of  $p$ , about 0.

## Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_sin
>>> R, x, y = ring('x, y', QQ)
>>> rs_sin(x + x*y, x, 4)
-1/6*x**3*y**3 - 1/2*x**3*y**2 - 1/2*x**3*y - 1/6*x**3 + x*y + x
```

(continues on next page)



(continued from previous page)

```
>>> rs_sin(x**QQ(3, 2) + x*y**QQ(7, 5), x, 4)
-1/2*x**(7/2)*y**(14/5) - 1/6*x**3*y**(21/5) + x**(3/2) + x*y**(7/5)
```

**See also:**[sin](#) (page 389)`sympy.polys.ring_series.rs_cos(p, x, prec)`

Cosine of a series

Return the series expansion of the cos of p, about 0.

**Examples**

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_cos
>>> R, x, y = ring('x, y', QQ)
>>> rs_cos(x + x*y, x, 4)
-1/2*x**2*y**2 - x**2*y - 1/2*x**2 + 1
>>> rs_cos(x + x*y, x, 4)/x**QQ(7, 5)
-1/2*x**(3/5)*y**2 - x**(3/5)*y - 1/2*x**(3/5) + x**(-7/5)
```

**See also:**[cos](#) (page 390)`sympy.polys.ring_series.rs_cos_sin(p, x, prec)`Return the tuple (`rs_cos(p, x, prec)`, `rs_sin(p, x, prec)`).Is faster than calling `rs_cos` and `rs_sin` separately`sympy.polys.ring_series.rs_atanh(p, x, prec)`

Hyperbolic arctangent of a series

Return the series expansion of the atanh of p, about 0.

**Examples**

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_atanh
>>> R, x, y = ring('x, y', QQ)
>>> rs_atanh(x + x*y, x, 4)
1/3*x**3*y**3 + x**3*y**2 + x**3*y + 1/3*x**3 + x*y + x
```

**See also:**[atanh](#) (page 406)`sympy.polys.ring_series.rs_sinh(p, x, prec)`

Hyperbolic sine of a series

Return the series expansion of the sinh of p, about 0.

## Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_sinh
>>> R, x, y = ring('x, y', QQ)
>>> rs_sinh(x + x*y, x, 4)
1/6*x**3*y**3 + 1/2*x**3*y**2 + 1/2*x**3*y + 1/6*x**3 + x*y + x
```

### See also:

[\*sinh\*](#) (page 402)

`sympy.polys.ring_series.rs_cosh(p, x, prec)`

Hyperbolic cosine of a series

Return the series expansion of the cosh of p, about 0.

## Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_cosh
>>> R, x, y = ring('x, y', QQ)
>>> rs_cosh(x + x*y, x, 4)
1/2*x**2*y**2 + x**2*y + 1/2*x**2 + 1
```

### See also:

[\*cosh\*](#) (page 403)

`sympy.polys.ring_series.rs_tanh(p, x, prec)`

Hyperbolic tangent of a series

Return the series expansion of the tanh of p, about 0.

## Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_tanh
>>> R, x, y = ring('x, y', QQ)
>>> rs_tanh(x + x*y, x, 4)
-1/3*x**3*y**3 - x**3*y**2 - x**3*y - 1/3*x**3 + x*y + x
```

### See also:

[\*tanh\*](#) (page 403)

`sympy.polys.ring_series.rs_hadamard_exp(p1, inverse=False)`

Return sum  $f_i/i!x^{**i}$  from sum  $f_ix^{**i}$ , where x is the first variable.

If `invers=True` return sum  $f_i*i!x^{**i}$

## Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_hadamard_exp
>>> R, x = ring('x', QQ)
>>> p = 1 + x + x**2 + x**3
>>> rs_hadamard_exp(p)
1/6*x**3 + 1/2*x**2 + x + 1
```

## Operations

`sympy.polys.ring_series.rs_mul(p1, p2, x, prec)`

Return the product of the given two series, modulo  $O(x^{**prec})$ .

`x` is the series variable or its position in the generators.

## Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_mul
>>> R, x = ring('x', QQ)
>>> p1 = x**2 + 2*x + 1
>>> p2 = x + 1
>>> rs_mul(p1, p2, x, 3)
3*x**2 + 3*x + 1
```

`sympy.polys.ring_series.rs_square(p1, x, prec)`

Square the series modulo  $O(x^{**prec})$

## Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_square
>>> R, x = ring('x', QQ)
>>> p = x**2 + 2*x + 1
>>> rs_square(p, x, 3)
6*x**2 + 4*x + 1
```

`sympy.polys.ring_series.rs_pow(p1, n, x, prec)`

Return  $p1^{**n}$  modulo  $O(x^{**prec})$

### Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_pow
>>> R, x = ring('x', QQ)
>>> p = x + 1
>>> rs_pow(p, 4, x, 3)
6*x**2 + 4*x + 1
```

`sympy.polys.ring_series.rs_series_inversion(p, x, prec)`  
Multivariate series inversion  $1/p$  modulo  $O(x^{**prec})$ .

### Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_series_inversion
>>> R, x, y = ring('x, y', QQ)
>>> rs_series_inversion(1 + x*y**2, x, 4)
-x**3*y**6 + x**2*y**4 - x*y**2 + 1
>>> rs_series_inversion(1 + x*y**2, y, 4)
-x*y**2 + 1
>>> rs_series_inversion(x + x**2, x, 4)
x**3 - x**2 + x - 1 + x**(-1)
```

`sympy.polys.ring_series.rs_series_reversion(p, x, n, y)`  
Reversion of a series.

$p$  is a series with  $O(x^{**n})$  of the form  $p = ax + f(x)$  where  $a$  is a number different from 0.

$$f(x) = \sum_{k=2}^{n-1} a_k x_k$$

#### Parameters

**a\_k** : Can depend polynomially on other variables, not indicated.

**x** : Variable with name x. **y** : Variable with name y.

#### Returns

Solve  $p = y$ , that is, given  $ax + f(x) - y = 0$ ,

find the solution  $x = r(y)$  up to  $O(y^n)$ .

### Algorithm

If  $r_i$  is the solution at order  $i$ , then:  $ar_i + f(r_i) - y = O(y^{i+1})$

and if  $r_{i+1}$  is the solution at order  $i + 1$ , then:  $ar_{i+1} + f(r_{i+1}) - y = O(y^{i+2})$

We have,  $r_{i+1} = r_i + e$ , such that,  $ae + f(r_i) = O(y^{i+2})$  or  $e = -f(r_i)/a$

So we use the recursion relation:  $r_{i+1} = r_i - f(r_i)/a$  with the boundary condition:  $r_1 = y$

## Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_series_reversion, rs_trunc
>>> R, x, y, a, b = ring('x, y, a, b', QQ)
>>> p = x - x**2 - 2*b*x**2 + 2*a*b*x**2
>>> p1 = rs_series_reversion(p, x, 3, y); p1
-2*y**2*a*b + 2*y**2*b + y**2 + y
>>> rs_trunc(p.compose(x, p1), y, 3)
y
```

`sympy.polys.ring_series.rs_nth_root(p, n, x, prec)`

Multivariate series expansion of the  $n$ th root of  $p$ .

### Parameters

**p** : Expr

The polynomial to computer the root of.

**n** : integer

The order of the root to be computed.

**x** : *PolyElement* (page 2554)

**prec** : integer

Order of the expanded series.

## Notes

The result of this function is dependent on the ring over which the polynomial has been defined. If the answer involves a root of a constant, make sure that the polynomial is over a real field. It cannot yet handle roots of symbols.

## Examples

```
>>> from sympy.polys.domains import QQ, RR
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_nth_root
>>> R, x, y = ring('x, y', QQ)
>>> rs_nth_root(1 + x + x*y, -3, x, 3)
2/9*x**2*y**2 + 4/9*x**2*y + 2/9*x**2 - 1/3*x*y - 1/3*x + 1
>>> R, x, y = ring('x, y', RR)
>>> rs_nth_root(3 + x + x*y, 3, x, 2)
0.160249952256379*x*y + 0.160249952256379*x + 1.44224957030741
```

`sympy.polys.ring_series.rs_trunc(p1, x, prec)`

Truncate the series in the  $x$  variable with precision  $prec$ , that is, modulo  $O(x^{**prec})$

## Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_trunc
>>> R, x = ring('x', QQ)
>>> p = x**10 + x**5 + x + 1
>>> rs_trunc(p, x, 12)
x**10 + x**5 + x + 1
>>> rs_trunc(p, x, 10)
x**5 + x + 1
```

`sympy.polys.ring_series.rs_subs(p, rules, x, prec)`

Substitution with truncation according to the mapping in `rules`.

Return a series with precision `prec` in the generator `x`

Note that substitutions are not done one after the other

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_subs
>>> R, x, y = ring('x, y', QQ)
>>> p = x**2 + y**2
>>> rs_subs(p, {x: x+ y, y: x+ 2*y}, x, 3)
2*x**2 + 6*x*y + 5*y**2
>>> (x + y)**2 + (x + 2*y)**2
2*x**2 + 6*x*y + 5*y**2
```

which differs from

```
>>> rs_subs(rs_subs(p, {x: x+ y}, x, 3), {y: x+ 2*y}, x, 3)
5*x**2 + 12*x*y + 8*y**2
```

### Parameters

**p** : *PolyElement* (page 2554) Input series.

**rules** : dict with substitution mappings.

**x** : *PolyElement* (page 2554) in which the series truncation is to be done.

**prec** : *Integer* (page 987) order of the series after truncation.

## Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_subs
>>> R, x, y = ring('x, y', QQ)
>>> rs_subs(x**2+y**2, {y: (x+y)**2}, x, 3)
6*x**2*y**2 + x**2 + 4*x*y**3 + y**4
```

`sympy.polys.ring_series.rs_diff(p, x)`

Return partial derivative of `p` with respect to `x`.

**Parameters**

**x** : *PolyElement* (page 2554) with respect to which p is differentiated.

**Examples**

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_diff
>>> R, x, y = ring('x, y', QQ)
>>> p = x + x**2*y**3
>>> rs_diff(p, x)
2*x*y**3 + 1
```

`sympy.polys.ring_series.rs_integrate(p, x)`

Integrate p with respect to x.

**Parameters**

**x** : *PolyElement* (page 2554) with respect to which p is integrated.

**Examples**

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_integrate
>>> R, x, y = ring('x, y', QQ)
>>> p = x + x**2*y**3
>>> rs_integrate(p, x)
1/3*x**3*y**3 + 1/2*x**2
```

`sympy.polys.ring_series.rs_newton(p, x, prec)`

Compute the truncated Newton sum of the polynomial p

**Examples**

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_newton
>>> R, x = ring('x', QQ)
>>> p = x**2 - 2
>>> rs_newton(p, x, 5)
8*x**4 + 4*x**2 + 2
```

`sympy.polys.ring_series.rs_compose_add(p1, p2)`

compute the composed sum  $\text{prod}(p2(x - \text{beta}))$  for beta root of p1)

## Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_compose_add
>>> R, x = ring('x', QQ)
>>> f = x**2 - 2
>>> g = x**2 - 3
>>> rs_compose_add(f, g)
x**4 - 10*x**2 + 1
```

## References

[R730]

## Utility functions

`sympy.polys.ring_series.rs_is_puiseux(p, x)`

Test if *p* is Puiseux series in *x*.

Raise an exception if it has a negative power in *x*.

## Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_is_puiseux
>>> R, x = ring('x', QQ)
>>> p = x**QQ(2,5) + x**QQ(2,3) + x
>>> rs_is_puiseux(p, x)
True
```

`sympy.polys.ring_series.rs_puiseux(f, p, x, prec)`

Return the puiseux series for  $f(p, x, prec)$ .

To be used when function *f* is implemented only for regular series.

## Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_puiseux, rs_exp
>>> R, x = ring('x', QQ)
>>> p = x**QQ(2,5) + x**QQ(2,3) + x
>>> rs_puiseux(rs_exp, p, x, 1)
1/2*x**(4/5) + x**(2/3) + x**(2/5) + 1
```

`sympy.polys.ring_series.rs_puiseux2(f, p, q, x, prec)`

Return the puiseux series for  $f(p, q, x, prec)$ .

To be used when function *f* is implemented only for regular series.



`sympy.polys.ring_series.rs_series_from_list(p, c, x, prec, concur=1)`

Return a series  $\sum c[n] * p^{**n}$  modulo  $O(x^{**prec})$ .

It reduces the number of multiplications by summing concurrently.

$ax = [1, p, p^{**2}, \dots, p^{*(J-1)}]$   $s = \sum(c[i] * ax[i] \text{ for } i \text{ in } \text{range}(r, (r+1) * J)) * p^{*((K-1) * J)}$   
with  $K \geq (n+1)/J$

## Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_series_from_list, rs_trunc
>>> R, x = ring('x', QQ)
>>> p = x**2 + x + 1
>>> c = [1, 2, 3]
>>> rs_series_from_list(p, c, x, 4)
6*x**3 + 11*x**2 + 8*x + 6
>>> rs_trunc(1 + 2*p + 3*p**2, x, 4)
6*x**3 + 11*x**2 + 8*x + 6
>>> pc = R.from_list(list(reversed(c)))
>>> rs_trunc(pc.compose(x, p), x, 4)
6*x**3 + 11*x**2 + 8*x + 6
```

`sympy.polys.ring_series.rs_fun(p, f, *args)`

Function of a multivariate series computed by substitution.

The case with `f` method name is used to compute `rs_tan` and `rs_nth_root` of a multivariate series:

`rs_fun(p, tan, iv, prec)`

`tan` series is first computed for a dummy variable `_x`, i.e., `rs_tan(_x, iv, prec)`. Then we substitute `_x` with `p` to get the desired series

## Parameters

**p** : *PolyElement* (page 2554) The multivariate series to be expanded.

**f** : *ring\_series* function to be applied on `p`.

**args[-2]** : *PolyElement* (page 2554) with respect to which, the series is to be expanded.

**args[-1]** : Required order of the expanded series.

## Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_fun, _tan1
>>> R, x, y = ring('x, y', QQ)
>>> p = x + x*y + x**2*y + x**3*y**2
>>> rs_fun(p, _tan1, x, 4)
1/3*x**3*y**3 + 2*x**3*y**2 + x**3*y + 1/3*x**3 + x**2*y + x*y + x
```

`sympy.polys.ring_series.mul_xin(p, i, n)`

Return  $p * x_i * n$ .

$x_i$  is the  $i$ th variable in  $p$ .

`sympy.polys.ring_series.pow_xin(p, i, n)`

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import pow_xin
>>> R, x, y = ring('x, y', QQ)
>>> p = x**QQ(2,5) + x + x**QQ(2,3)
>>> index = p.ring.gens.index(x)
>>> pow_xin(p, index, 15)
x**15 + x**10 + x**6
```

## Literature

The following is a non-comprehensive list of publications that were used as a theoretical foundation for implementing polynomials manipulation module.

## Poly solvers

This module provides functions for solving systems of linear equations that are used internally in sympy. Low-level linear systems solver.

### `solve_lin_sys`

`sympy.polys.solvers.solve_lin_sys(eqs, ring, _raw=True)`

Solve a system of linear equations from a PolynomialRing

#### Parameters

**eqs:** list[PolyElement]

The linear equations to be solved as elements of a PolynomialRing (assumed equal to zero).

**ring:** PolynomialRing

The polynomial ring from which eqs are drawn. The generators of this ring are the unknowns to be solved for and the domain of the ring is the domain of the coefficients of the system of equations.

**\_raw:** bool

If `_raw` is False, the keys and values in the returned dictionary will be of type Expr (and the unit of the field will be removed from the keys) otherwise the low-level polys types will be returned, e.g. PolyElement: PythonRational.

#### Returns

None if the system has no solution.

dict[Symbol, Expr] if `_raw=False`

dict[Symbol, DomainElement] if `_raw=True`.

### Explanation

Solves a system of linear equations given as PolyElement instances of a PolynomialRing. The basic arithmetic is carried out using instance of DomainElement which is more efficient than *Expr* (page 947) for the most common inputs.

While this is a public function it is intended primarily for internal use so its interface is not necessarily convenient. Users are suggested to use the *sympy.solvers.solve*.*linsolve()* (page 872) function (which uses this function internally) instead.

### Examples

```
>>> from sympy import symbols
>>> from sympy.polys.solvers import solve_lin_sys, sympy_eqs_to_ring
>>> x, y = symbols('x, y')
>>> eqs = [x - y, x + y - 2]
>>> eqs_ring, ring = sympy_eqs_to_ring(eqs, [x, y])
>>> solve_lin_sys(eqs_ring, ring)
{y: 1, x: 1}
```

Passing `_raw=False` returns the same result except that the keys are Expr rather than low-level poly types.

```
>>> solve_lin_sys(eqs_ring, ring, _raw=False)
{x: 1, y: 1}
```

### See also:

*sympy\_eqs\_to\_ring* (page 2668)

prepares the inputs to *solve\_lin\_sys*.

*linsolve* (page 872)

*linsolve* uses *solve\_lin\_sys* internally.

*sympy.solvers.solvers.solve* (page 836)

*solve* uses *solve\_lin\_sys* internally.

### eqs\_to\_matrix

*sympy.polys.solvers.eqs\_to\_matrix*(*eqs\_coeffs*, *eqs\_rhs*, *gens*, *domain*)

Get matrix from linear equations in dict format.

#### Parameters

**eqs\_coeffs:** list[dict[Symbol, DomainElement]]

The left hand sides of the equations as dicts mapping from symbols to coefficients where the coefficients are instances of DomainElement.

**eqs\_rhs:** list[DomainElements]

The right hand sides of the equations as instances of DomainElement.

**gens:** list[Symbol]

The unknowns in the system of equations.

**domain:** Domain

The domain for coefficients of both lhs and rhs.

**Returns**

The augmented matrix representation of the system as a DomainMatrix.

## Explanation

Get the matrix representation of a system of linear equations represented as dicts with low-level DomainElement coefficients. This is an *internal* function that is used by `solve_lin_sys`.

## Examples

```
>>> from sympy import symbols, ZZ
>>> from sympy.polys.solvers import eqs_to_matrix
>>> x, y = symbols('x, y')
>>> eqs_coeff = [{x:ZZ(1), y:ZZ(1)}, {x:ZZ(1), y:ZZ(-1)}]
>>> eqs_rhs = [ZZ(0), ZZ(-1)]
>>> eqs_to_matrix(eqs_coeff, eqs_rhs, [x, y], ZZ)
DomainMatrix([[1, 1, 0], [1, -1, 1]], (2, 3), ZZ)
```

**See also:**

[`solve\_lin\_sys`](#) (page 2666)

Uses `eqs_to_matrix()` (page 2667) internally

## `sympy_eqs_to_ring`

`sympy.polys.solvers.sympy_eqs_to_ring(eqs, symbols)`

Convert a system of equations from Expr to a PolyRing

**Parameters**

**eqs:** List of Expr

A list of equations as Expr instances

**symbols:** List of Symbol

A list of the symbols that are the unknowns in the system of equations.

**Returns**

Tuple[List[PolyElement], Ring]: The equations as PolyElement instances and the ring of polynomials within which each equation is represented.

## Explanation

High-level functions like `solve` expect `Expr` as inputs but can use `solve_lin_sys` internally. This function converts equations from `Expr` to the low-level poly types used by the `solve_lin_sys` function.

## Examples

```
>>> from sympy import symbols
>>> from sympy.polys.solvers import sympy_eqs_to_ring
>>> a, x, y = symbols('a, x, y')
>>> eqs = [x-y, x+a*y]
>>> eqs_ring, ring = sympy_eqs_to_ring(eqs, [x, y])
>>> eqs_ring
[x - y, x + a*y]
>>> type(eqs_ring[0])
<class 'sympy.polys.rings.PolyElement'>
>>> ring
ZZ(a)[x,y]
```

With the equations in this form they can be passed to `solve_lin_sys`:

```
>>> from sympy.polys.solvers import solve_lin_sys
>>> solve_lin_sys(eqs_ring, ring)
{y: 0, x: 0}
```

## `_solve_lin_sys`

`sympy.polys.solvers._solve_lin_sys(eqs_coeffs, eqs_rhs, ring)`

Solve a linear system from dict of PolynomialRing coefficients

## Explanation

This is an **internal** function used by `solve_lin_sys()` (page 2666) after the equations have been preprocessed. The role of this function is to split the system into connected components and pass those to `_solve_lin_sys_component()` (page 2670).

## Examples

Setup a system for  $x - y = 0$  and  $x + y = 2$  and solve:

```
>>> from sympy import symbols, sring
>>> from sympy.polys.solvers import _solve_lin_sys
>>> x, y = symbols('x, y')
>>> R, (xr, yr) = sring([x, y], [x, y])
>>> eqs = [{xr:R.one, yr:-R.one}, {xr:R.one, yr:R.one}]
>>> eqs_rhs = [R.zero, -2*R.one]
>>> _solve_lin_sys(eqs, eqs_rhs, R)
{y: 1, x: 1}
```

See also:

[`solve\_lin\_sys`](#) (page 2666)

This function is used internally by [`solve\_lin\_sys\(\)`](#) (page 2666).

### `_solve_lin_sys_component`

`sympy.polys.solvers._solve_lin_sys_component(eqs_coeffs, eqs_rhs, ring)`

Solve a linear system from dict of PolynomialRing coefficients

### Explanation

This is an **internal** function used by [`solve\_lin\_sys\(\)`](#) (page 2666) after the equations have been preprocessed. After [`\_solve\_lin\_sys\(\)`](#) (page 2669) splits the system into connected components this function is called for each component. The system of equations is solved using Gauss-Jordan elimination with division followed by back-substitution.

### Examples

Setup a system for  $x - y = 0$  and  $x + y = 2$  and solve:

```
>>> from sympy import symbols, sring
>>> from sympy.polys.solvers import _solve_lin_sys_component
>>> x, y = symbols('x, y')
>>> R, (xr, yr) = sring([x, y], [x, y])
>>> eqs = [{xr:R.one, yr:-R.one}, {xr:R.one, yr:R.one}]
>>> eqs_rhs = [R.zero, -2*R.one]
>>> _solve_lin_sys_component(eqs, eqs_rhs, R)
{y: 1, x: 1}
```

See also:

[`solve\_lin\_sys`](#) (page 2666)

This function is used internally by [`solve\_lin\_sys\(\)`](#) (page 2666).

## Introducing the domainmatrix of the poly module

This page introduces the idea behind domainmatrix which is used in SymPy's [`sympy.polys`](#) (page 2360) module. This is a relatively advanced topic so for a better understanding it is recommended to read about [`Domain`](#) (page 2504) and [`DDM`](#) (page 2690) along with [`sympy.matrices`](#) (page 1217) module.

## What is domainmatrix?

It is way of associating Matrix with *Domain* (page 2504).

A domainmatrix represents a matrix with elements that are in a particular Domain. Each domainmatrix internally wraps a DDM which is used for the lower-level operations. The idea is that the domainmatrix class provides the convenience routines for converting between Expr and the poly domains as well as unifying matrices with different domains.

**In general, we represent a matrix without concerning about the *Domain* (page 2504) as:**

```
>>> from sympy import Matrix
>>> from sympy.polys.matrices import DomainMatrix
>>> A = Matrix([
... [1, 2],
... [3, 4]])
>>> A
Matrix([
[1, 2],
[3, 4]])
```

## DomainMatrix Class Reference

```
class sympy.polys.matrices.domainmatrix.DomainMatrix(rows, shape, domain, *,
 fmt=None)
```

Associate Matrix with *Domain* (page 2504)

### Explanation

DomainMatrix uses *Domain* (page 2504) for its internal representation which makes it more faster for many common operations than current SymPy Matrix class, but this advantage makes it not entirely compatible with Matrix. DomainMatrix could be found analogous to numpy arrays with “dtype”. In the DomainMatrix, each matrix has a domain such as *ZZ* (page 2525) or *QQ<a>* (page 2539).

### Examples

Creating a DomainMatrix from the existing Matrix class:

```
>>> from sympy import Matrix
>>> from sympy.polys.matrices import DomainMatrix
>>> Matrix1 = Matrix([
... [1, 2],
... [3, 4]])
>>> A = DomainMatrix.from_Matrix(Matrix1)
>>> A
DomainMatrix({0: {0: 1, 1: 2}, 1: {0: 3, 1: 4}}, (2, 2), ZZ)
```

Directly forming a DomainMatrix:

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
... [ZZ(1), ZZ(2)],
... [ZZ(3), ZZ(4)]], (2, 2), ZZ)
>>> A
DomainMatrix([[1, 2], [3, 4]], (2, 2), ZZ)
```

**See also:**

[DDM](#) (page 2690), [SDM](#) (page 2692), [Domain](#) (page 2504), [Poly](#) (page 2378)

**add(B)**

Adds two DomainMatrix matrices of the same Domain

**Parameters**

**A, B: DomainMatrix**

matrices to add

**Returns**

DomainMatrix

DomainMatrix after Addition

**Raises**

**DMShapeError**

If the dimensions of the two DomainMatrix are not equal

**ValueError**

If the domain of the two DomainMatrix are not same

**Examples**

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
... [ZZ(1), ZZ(2)],
... [ZZ(3), ZZ(4)]], (2, 2), ZZ)
>>> B = DomainMatrix([
... [ZZ(4), ZZ(3)],
... [ZZ(2), ZZ(1)]], (2, 2), ZZ)
```

```
>>> A.add(B)
DomainMatrix([[5, 5], [5, 5]], (2, 2), ZZ)
```

**See also:**

[sub](#) (page 2686), [matmul](#) (page 2680)

**charpoly()**

Returns the coefficients of the characteristic polynomial of the DomainMatrix. These elements will be domain elements. The domain of the elements will be same as domain of the DomainMatrix.



**Returns**

list

coefficients of the characteristic polynomial

**Raises****DMNonSquareMatrixError**

If the DomainMatrix is not a not Square DomainMatrix

**Examples**

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
... [ZZ(1), ZZ(2)],
... [ZZ(3), ZZ(4)]], (2, 2), ZZ)
```

```
>>> A.charpoly()
[1, -5, -2]
```

**columnspace()**

Returns the columnspace for the DomainMatrix

**Returns**

DomainMatrix

The columns of this matrix form a basis for the columnspace.

**Examples**

```
>>> from sympy import QQ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
... [QQ(1), QQ(-1)],
... [QQ(2), QQ(-2)]], (2, 2), QQ)
>>> A.columnspace()
DomainMatrix([[1], [2]], (2, 1), QQ)
```

**convert\_to(K)**

Change the domain of DomainMatrix to desired domain or field

**Parameters****K** : Represents the desired domain or field.

Alternatively, None may be passed, in which case this method just returns a copy of this DomainMatrix.

**Returns**

DomainMatrix

DomainMatrix with the desired domain or field

### Examples

```
>>> from sympy import ZZ, ZZ_I
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
... [ZZ(1), ZZ(2)],
... [ZZ(3), ZZ(4)]], (2, 2), ZZ)
```

```
>>> A.convert_to(ZZ_I)
DomainMatrix([[1, 2], [3, 4]], (2, 2), ZZ_I)
```

#### **det()**

Returns the determinant of a Square DomainMatrix

##### **Returns**

S.Complexes

determinant of Square DomainMatrix

##### **Raises**

##### **ValueError**

If the domain of DomainMatrix not a Field

### Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
... [ZZ(1), ZZ(2)],
... [ZZ(3), ZZ(4)]], (2, 2), ZZ)
```

```
>>> A.det()
-2
```

#### **classmethod diag(*diagonal*, *domain*, *shape*=None)**

Return diagonal matrix with entries from *diagonal*.

### Examples

```
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy import ZZ
>>> DomainMatrix.diag([ZZ(5), ZZ(6)], ZZ)
DomainMatrix({0: {0: 5}, 1: {1: 6}}, (2, 2), ZZ)
```

#### **classmethod eye(*shape*, *domain*)**

Return identity matrix of size *n*

## Examples

```
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy import QQ
>>> DomainMatrix.eye(3, QQ)
DomainMatrix({0: {0: 1}, 1: {1: 1}, 2: {2: 1}}, (3, 3), QQ)
```

**classmethod** `from_Matrix(M, fmt='sparse', **kwargs)`

Convert Matrix to DomainMatrix

### Parameters

**M: Matrix**

### Returns

Returns DomainMatrix with identical elements as M

## Examples

```
>>> from sympy import Matrix
>>> from sympy.polys.matrices import DomainMatrix
>>> M = Matrix([
... [1.0, 3.4],
... [2.4, 1]])
>>> A = DomainMatrix.from_Matrix(M)
>>> A
DomainMatrix({0: {0: 1.0, 1: 3.4}, 1: {0: 2.4, 1: 1.0}}, (2, 2), RR)
```

We can keep internal representation as ddm using `fmt='dense'`

```
>>> from sympy import Matrix, QQ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix.from_Matrix(Matrix([[QQ(1, 2), QQ(3, 4)], [QQ(0, 1), QQ(0, 1)]]),
fmt='dense')
>>> A.rep
[[1/2, 3/4], [0, 0]]
```

### See also:

[Matrix](#) (page 1361)

**classmethod** `from_dict_sympy(nrows, ncols, elemsdict, **kwargs)`

### Parameters

**nrows: number of rows**

**ncols: number of cols**

**elemsdict: dict of dicts containing non-zero elements of the DomainMatrix**

### Returns

DomainMatrix containing elements of elemsdict

## Examples

```
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy.abc import x,y,z
>>> elemsdict = {0: {0:x}, 1:{1: y}, 2: {2: z}}
>>> A = DomainMatrix.from_dict_sympy(3, 3, elemsdict)
>>> A
DomainMatrix({0: {0: x}, 1: {1: y}, 2: {2: z}}, (3, 3), ZZ[x,y,z])
```

### See also:

[\*from\\_list\\_sympy\*](#) (page 2676)

**classmethod** `from_list(rows, domain)`

Convert a list of lists into a DomainMatrix

#### Parameters

**rows:** list of lists

Each element of the inner lists should be either the single arg, or tuple of args, that would be passed to the domain constructor in order to form an element of the domain. See examples.

#### Returns

DomainMatrix containing elements defined in rows

## Examples

```
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy import FF, QQ, ZZ
>>> A = DomainMatrix.from_list([[1, 0, 1], [0, 0, 1]], ZZ)
>>> A
DomainMatrix([[1, 0, 1], [0, 0, 1]], (2, 3), ZZ)
>>> B = DomainMatrix.from_list([[1, 0, 1], [0, 0, 1]], FF(7))
>>> B
DomainMatrix([[1 mod 7, 0 mod 7, 1 mod 7], [0 mod 7, 0 mod 7, 1 mod 7]], (2, 3), GF(7))
>>> C = DomainMatrix.from_list([(1, 2), (3, 1)], [(1, 4), (5, 1)]), QQ
>>> C
DomainMatrix([[1/2, 3], [1/4, 5]], (2, 2), QQ)
```

### See also:

[\*from\\_list\\_sympy\*](#) (page 2676)

**classmethod** `from_list_sympy(nrows, ncols, rows, **kwargs)`

Convert a list of lists of Expr into a DomainMatrix using `construct_domain`

#### Parameters

**nrows:** number of rows

**ncols:** number of columns

**rows:** list of lists

**Returns**

DomainMatrix containing elements of rows

**Examples**

```
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy.abc import x, y, z
>>> A = DomainMatrix.from_list_sympy(1, 3, [[x, y, z]])
>>> A
DomainMatrix([[x, y, z]], (1, 3), ZZ[x,y,z])
```

**See also:**

[sympy.polys.constructor.construct\\_domain](#) (page 2427), [from\\_dict\\_sympy](#) (page 2675)

**classmethod from\_rep(rep)**

Create a new DomainMatrix efficiently from DDM/SDM.

**Parameters**

**rep: SDM or DDM**

The internal sparse or dense representation of the matrix.

**Returns**

DomainMatrix

A *DomainMatrix* (page 2671) wrapping *rep*.

**Examples**

Create a *DomainMatrix* (page 2671) with an dense internal representation as *DDM* (page 2690):

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy.polys.matrices.ddm import DDM
>>> drep = DDM([[ZZ(1), ZZ(2)], [ZZ(3), ZZ(4)]], (2, 2), ZZ)
>>> dM = DomainMatrix.from_rep(drep)
>>> dM
DomainMatrix([[1, 2], [3, 4]], (2, 2), ZZ)
```

Create a *DomainMatrix* (page 2671) with a sparse internal representation as *SDM* (page 2692):

```
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy.polys.matrices.sdm import SDM
>>> from sympy import ZZ
>>> drep = SDM({0:{1:ZZ(1)},1:{0:ZZ(2)}} , (2, 2), ZZ)
>>> dM = DomainMatrix.from_rep(drep)
>>> dM
DomainMatrix({0: {1: 1}, 1: {0: 2}}, (2, 2), ZZ)
```

## Notes

This takes ownership of *rep* as its internal representation. If *rep* is being mutated elsewhere then a copy should be provided to *from\_rep*. Only minimal verification or checking is done on *rep* as this is supposed to be an efficient internal routine.

### **hstack(\*B)**

Horizontally stack the given matrices.

#### **Parameters**

##### **B: DomainMatrix**

Matrices to stack horizontally.

#### **Returns**

DomainMatrix

DomainMatrix by stacking horizontally.

## Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
```

```
>>> A = DomainMatrix([[ZZ(1), ZZ(2)], [ZZ(3), ZZ(4)]], (2, 2), ZZ)
>>> B = DomainMatrix([[ZZ(5), ZZ(6)], [ZZ(7), ZZ(8)]], (2, 2), ZZ)
>>> A.hstack(B)
DomainMatrix([[1, 2, 5, 6], [3, 4, 7, 8]], (2, 4), ZZ)
```

```
>>> C = DomainMatrix([[ZZ(9), ZZ(10)], [ZZ(11), ZZ(12)]], (2, 2), ZZ)
>>> A.hstack(B, C)
DomainMatrix([[1, 2, 5, 6, 9, 10], [3, 4, 7, 8, 11, 12]], (2, 6), ZZ)
```

### **See also:**

[\*unify\*](#) (page 2688)

### **inv()**

Finds the inverse of the DomainMatrix if exists

#### **Returns**

DomainMatrix

DomainMatrix after inverse

#### **Raises**

##### **ValueError**

If the domain of DomainMatrix not a Field

##### **DMNonSquareMatrixError**

If the DomainMatrix is not a not Square DomainMatrix

## Examples

```
>>> from sympy import QQ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
... [QQ(2), QQ(-1), QQ(0)],
... [QQ(-1), QQ(2), QQ(-1)],
... [QQ(0), QQ(0), QQ(2)]] , (3, 3), QQ)
>>> A.inv()
DomainMatrix([[2/3, 1/3, 1/6], [1/3, 2/3, 1/3], [0, 0, 1/2]], (3, 3),
↪ QQ)
```

### See also:

[neg](#) (page 2681)

### property is\_lower

Says whether this matrix is lower-triangular. True can be returned even if the matrix is not square.

### property is\_upper

Says whether this matrix is upper-triangular. True can be returned even if the matrix is not square.

### lu()

Returns Lower and Upper decomposition of the DomainMatrix

#### Returns

(L, U, exchange)

L, U are Lower and Upper decomposition of the DomainMatrix, exchange is the list of indices of rows exchanged in the decomposition.

#### Raises

##### ValueError

If the domain of DomainMatrix not a Field

## Examples

```
>>> from sympy import QQ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
... [QQ(1), QQ(-1)],
... [QQ(2), QQ(-2)]] , (2, 2), QQ)
>>> A.lu()
(DomainMatrix([[1, 0], [2, 1]], (2, 2), QQ), DomainMatrix([[1, -1],
↪ [0, 0]], (2, 2), QQ), [])
```

### See also:

[lu\\_solve](#) (page 2679)

### lu\_solve(rhs)

Solver for DomainMatrix x in the  $A*x = B$

**Parameters****rhs** : DomainMatrix B**Returns**

DomainMatrix

 $x$  in  $A*x = B$ **Raises****DMShapeError**

If the DomainMatrix A and rhs have different number of rows

**ValueError**

If the domain of DomainMatrix A not a Field

**Examples**

```
>>> from sympy import QQ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
... [QQ(1), QQ(2)],
... [QQ(3), QQ(4)]], (2, 2), QQ)
>>> B = DomainMatrix([
... [QQ(1), QQ(1)],
... [QQ(0), QQ(1)]], (2, 2), QQ)
```

```
>>> A.lu_solve(B)
DomainMatrix([[-2, -1], [3/2, 1]], (2, 2), QQ)
```

**See also:**[lu](#) (page 2679)**matmul(B)**

Performs matrix multiplication of two DomainMatrix matrices

**Parameters****A, B: DomainMatrix**

to multiply

**Returns**

DomainMatrix

DomainMatrix after multiplication



## Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
... [ZZ(1), ZZ(2)],
... [ZZ(3), ZZ(4)]], (2, 2), ZZ)
>>> B = DomainMatrix([
... [ZZ(1), ZZ(1)],
... [ZZ(0), ZZ(1)]], (2, 2), ZZ)
```

```
>>> A.matmul(B)
DomainMatrix([[1, 3], [3, 7]], (2, 2), ZZ)
```

### See also:

[mul](#) (page 2681), [pow](#) (page 2683), [add](#) (page 2672), [sub](#) (page 2686)

### `mul(b)`

Performs term by term multiplication for the second DomainMatrix w.r.t first DomainMatrix. Returns a DomainMatrix whose rows are list of DomainMatrix matrices created after term by term multiplication.

#### Parameters

**A, B: DomainMatrix**

matrices to multiply term-wise

#### Returns

DomainMatrix

DomainMatrix after term by term multiplication

## Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
... [ZZ(1), ZZ(2)],
... [ZZ(3), ZZ(4)]], (2, 2), ZZ)
>>> B = DomainMatrix([
... [ZZ(1), ZZ(1)],
... [ZZ(0), ZZ(1)]], (2, 2), ZZ)
```

```
>>> A.mul(B)
DomainMatrix([[DomainMatrix([[1, 1], [0, 1]], (2, 2), ZZ),
DomainMatrix([[2, 2], [0, 2]], (2, 2), ZZ)],
[DomainMatrix([[3, 3], [0, 3]], (2, 2), ZZ),
DomainMatrix([[4, 4], [0, 4]], (2, 2), ZZ)]], (2, 2), ZZ)
```

### See also:

[matmul](#) (page 2680)

**neg()**

Returns the negative of DomainMatrix

**Parameters****A** : Represents a DomainMatrix**Returns**

DomainMatrix

DomainMatrix after Negation

**Examples**

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
... [ZZ(1), ZZ(2)],
... [ZZ(3), ZZ(4)]], (2, 2), ZZ)
```

```
>>> A.neg()
DomainMatrix([[-1, -2], [-3, -4]], (2, 2), ZZ)
```

**nullspace()**

Returns the nullspace for the DomainMatrix

**Returns**

DomainMatrix

The rows of this matrix form a basis for the nullspace.

**Examples**

```
>>> from sympy import QQ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
... [QQ(1), QQ(-1)],
... [QQ(2), QQ(-2)]], (2, 2), QQ)
>>> A.nullspace()
DomainMatrix([[1, 1]], (1, 2), QQ)
```

**classmethod ones(*shape, domain*)**Returns a DomainMatrix of 1s, of size *shape*, belonging to the specified domain

## Examples

```
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy import QQ
>>> DomainMatrix.ones((2,3), QQ)
DomainMatrix([[1, 1, 1], [1, 1, 1]], (2, 3), QQ)
```

### **pow(*n*)**

Computes  $A^{**n}$

#### **Parameters**

**A** : DomainMatrix

**n** : exponent for A

#### **Returns**

DomainMatrix

DomainMatrix on computing  $A^{**n}$

#### **Raises**

**NotImplementedError**

if n is negative.

## Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
... [ZZ(1), ZZ(1)],
... [ZZ(0), ZZ(1)]], (2, 2), ZZ)
```

```
>>> A.pow(2)
DomainMatrix([[1, 2], [0, 1]], (2, 2), ZZ)
```

#### **See also:**

[\*matmul\*](#) (page 2680)

### **rowspace()**

Returns the rowspace for the DomainMatrix

#### **Returns**

DomainMatrix

The rows of this matrix form a basis for the rowspace.

### Examples

```
>>> from sympy import QQ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
... [QQ(1), QQ(-1)],
... [QQ(2), QQ(-2)]]], (2, 2), QQ)
>>> A.rowspace()
DomainMatrix([[1, -1]], (1, 2), QQ)
```

#### **rref()**

Returns reduced-row echelon form and list of pivots for the DomainMatrix

##### **Returns**

(DomainMatrix, list)

reduced-row echelon form and list of pivots for the DomainMatrix

##### **Raises**

##### **ValueError**

If the domain of DomainMatrix not a Field

### Examples

```
>>> from sympy import QQ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
... [QQ(2), QQ(-1), QQ(0)],
... [QQ(-1), QQ(2), QQ(-1)],
... [QQ(0), QQ(0), QQ(2)]]], (3, 3), QQ)
```

```
>>> rref_matrix, rref_pivots = A.rref()
>>> rref_matrix
DomainMatrix([[1, 0, 0], [0, 1, 0], [0, 0, 1]], (3, 3), QQ)
>>> rref_pivots
(0, 1, 2)
```

#### **See also:**

[convert\\_to](#) (page 2673), [lu](#) (page 2679)

#### **scc()**

Compute the strongly connected components of a DomainMatrix

##### **Returns**

List of lists of integers

Each list represents a strongly connected component.

## Explanation

A square matrix can be considered as the adjacency matrix for a directed graph where the row and column indices are the vertices. In this graph if there is an edge from vertex  $i$  to vertex  $j$  if  $M[i, j]$  is nonzero. This routine computes the strongly connected components of that graph which are subsets of the rows and columns that are connected by some nonzero element of the matrix. The strongly connected components are useful because many operations such as the determinant can be computed by working with the submatrices corresponding to each component.

## Examples

Find the strongly connected components of a matrix:

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> M = DomainMatrix([[ZZ(1), ZZ(0), ZZ(2)],
... [ZZ(0), ZZ(3), ZZ(0)],
... [ZZ(4), ZZ(6), ZZ(5)]]), (3, 3), ZZ)
>>> M.scc()
[[1], [0, 2]]
```

Compute the determinant from the components:

```
>>> MM = M.to_Matrix()
>>> MM
Matrix([
[1, 0, 2],
[0, 3, 0],
[4, 6, 5]])
>>> MM[[1], [1]]
Matrix([[3]])
>>> MM[[0, 2], [0, 2]]
Matrix([
[1, 2],
[4, 5]])
>>> MM.det()
-9
>>> MM[[1], [1]].det() * MM[[0, 2], [0, 2]].det()
-9
```

The components are given in reverse topological order and represent a permutation of the rows and columns that will bring the matrix into block lower-triangular form:

```
>>> MM[[1, 0, 2], [1, 0, 2]]
Matrix([
[3, 0, 0],
[0, 1, 2],
[6, 4, 5]])
```

## See also:

[\*sympy.matrices.matrices.MatrixBase.strongly\\_connected\\_components\*](#)  
(page 1314), [\*sympy.utilities.iterables.strongly\\_connected\\_components\*](#)

(page 2094)

**sub(*B*)**

Subtracts two DomainMatrix matrices of the same Domain

**Parameters**

**A, B: DomainMatrix**

matrices to subtract

**Returns**

DomainMatrix

DomainMatrix after Subtraction

**Raises**

**DMShapeError**

If the dimensions of the two DomainMatrix are not equal

**ValueError**

If the domain of the two DomainMatrix are not same

**Examples**

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
... [ZZ(1), ZZ(2)],
... [ZZ(3), ZZ(4)]], (2, 2), ZZ)
>>> B = DomainMatrix([
... [ZZ(4), ZZ(3)],
... [ZZ(2), ZZ(1)]], (2, 2), ZZ)
```

```
>>> A.sub(B)
DomainMatrix([[-3, -1], [1, 3]], (2, 2), ZZ)
```

**See also:**

[add](#) (page 2672), [matmul](#) (page 2680)

**to\_Matrix()**

Convert DomainMatrix to Matrix

**Returns**

Matrix

MutableDenseMatrix for the DomainMatrix

### Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
... [ZZ(1), ZZ(2)],
... [ZZ(3), ZZ(4)]], (2, 2), ZZ)
```

```
>>> A.to_Matrix()
Matrix([
 [1, 2],
 [3, 4]])
```

#### See also:

[`from\_Matrix`](#) (page 2675)

#### `to_dense()`

Return a dense DomainMatrix representation of *self*.

### Examples

```
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy import QQ
>>> A = DomainMatrix({0: {0: 1}, 1: {1: 2}}, (2, 2), QQ)
>>> A.rep
{0: {0: 1}, 1: {1: 2}}
>>> B = A.to_dense()
>>> B.rep
[[1, 0], [0, 2]]
```

#### `to_field()`

Returns a DomainMatrix with the appropriate field

##### Returns

DomainMatrix

DomainMatrix with the appropriate field

### Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
... [ZZ(1), ZZ(2)],
... [ZZ(3), ZZ(4)]], (2, 2), ZZ)
```

```
>>> A.to_field()
DomainMatrix([[1, 2], [3, 4]], (2, 2), QQ)
```

#### `to_sparse()`

Return a sparse DomainMatrix representation of *self*.

## Examples

```
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy import QQ
>>> A = DomainMatrix([[1, 0],[0, 2]], (2, 2), QQ)
>>> A.rep
[[1, 0], [0, 2]]
>>> B = A.to_sparse()
>>> B.rep
{0: {0: 1}, 1: {1: 2}}
```

### **transpose()**

Matrix transpose of self

### **unify(\*others, fmt=None)**

Unifies the domains and the format of self and other matrices.

#### **Parameters**

**others** : DomainMatrix

**fmt**: string 'dense', 'sparse' or 'None' (default)

The preferred format to convert to if self and other are not already in the same format. If *None* or not specified then no conversion is performed.

#### **Returns**

Tuple[DomainMatrix]

Matrices with unified domain and format

## Examples

Unify the domain of DomainMatrix that have different domains:

```
>>> from sympy import ZZ, QQ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([[ZZ(1), ZZ(2)]], (1, 2), ZZ)
>>> B = DomainMatrix([[QQ(1, 2), QQ(2)]]), (1, 2), QQ)
>>> Aq, Bq = A.unify(B)
>>> Aq
DomainMatrix([[1, 2]], (1, 2), QQ)
>>> Bq
DomainMatrix([[1/2, 2]], (1, 2), QQ)
```

Unify the format (dense or sparse):

```
>>> A = DomainMatrix([[ZZ(1), ZZ(2)]], (1, 2), ZZ)
>>> B = DomainMatrix({0:{0: ZZ(1)}}), (2, 2), ZZ)
>>> B.rep
{0: {0: 1}}
```

```
>>> A2, B2 = A.unify(B, fmt='dense')
>>> B2.rep
[[1, 0], [0, 0]]
```



**See also:**

[convert\\_to](#) (page 2673), [to\\_dense](#) (page 2687), [to\\_sparse](#) (page 2687)

**vstack(\*B)**

Vertically stack the given matrices.

**Parameters****B: DomainMatrix**

Matrices to stack vertically.

**Returns**

DomainMatrix

DomainMatrix by stacking vertically.

**Examples**

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
```

```
>>> A = DomainMatrix([[ZZ(1), ZZ(2)], [ZZ(3), ZZ(4)]], (2, 2), ZZ)
>>> B = DomainMatrix([[ZZ(5), ZZ(6)], [ZZ(7), ZZ(8)]], (2, 2), ZZ)
>>> A.vstack(B)
DomainMatrix([[1, 2], [3, 4], [5, 6], [7, 8]], (4, 2), ZZ)
```

```
>>> C = DomainMatrix([[ZZ(9), ZZ(10)], [ZZ(11), ZZ(12)]], (2, 2), ZZ)
>>> A.vstack(B, C)
DomainMatrix([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12]], (6, 2), ZZ)
```

**See also:**

[unify](#) (page 2688)

**classmethod zeros(shape, domain, \*, fmt='sparse')**

Returns a zero DomainMatrix of size shape, belonging to the specified domain

**Examples**

```
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy import QQ
>>> DomainMatrix.zeros((2, 3), QQ)
DomainMatrix({}, (2, 3), QQ)
```

## DDM Class Reference

**class** `sympy.polys.matrices.ddm.DDM(rowlist, shape, domain)`

Dense matrix based on polys domain elements

This is a list subclass and is a wrapper for a list of lists that supports basic matrix arithmetic  $+$ ,  $-$ ,  $*$ .

**add**(*b*)

$a + b$

**charpoly**()

Coefficients of characteristic polynomial of *a*

**det**()

Determinant of *a*

**hstack**(\**B*)

Horizontally stacks [DDM](#) (page 2690) matrices.

### Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices.sdm import DDM
```

```
>>> A = DDM([[ZZ(1), ZZ(2)], [ZZ(3), ZZ(4)]], (2, 2), ZZ)
>>> B = DDM([[ZZ(5), ZZ(6)], [ZZ(7), ZZ(8)]], (2, 2), ZZ)
>>> A.hstack(B)
[[1, 2, 5, 6], [3, 4, 7, 8]]
```

```
>>> C = DDM([[ZZ(9), ZZ(10)], [ZZ(11), ZZ(12)]], (2, 2), ZZ)
>>> A.hstack(B, C)
[[1, 2, 5, 6, 9, 10], [3, 4, 7, 8, 11, 12]]
```

**inv**()

Inverse of *a*

**is\_lower**()

Says whether this matrix is lower-triangular. True can be returned even if the matrix is not square.

**is\_upper**()

Says whether this matrix is upper-triangular. True can be returned even if the matrix is not square.

**is\_zero\_matrix**()

Says whether this matrix has all zero entries.

**lu**()

L, U decomposition of *a*

**lu\_solve**(*b*)

*x* where  $a*x = b$

**matmul(*b*)***a* @ *b* (matrix product)**neg()**-*a***rref()**Reduced-row echelon form of *a* and list of pivots**scc()**Strongly connected components of a square matrix *a*.

### Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices.sdm import DDM
>>> A = DDM([[ZZ(1), ZZ(0)], [ZZ(0), ZZ(1)]], (2, 2), ZZ)
>>> A.scc()
[[0], [1]]
```

#### See also:

[\*sympy.polys.matrices.domainmatrix.DomainMatrix.scc\*](#) (page 2684)**sub(*b*)***a* - *b***vstack(\**B*)**Vertically stacks [\*DDM\*](#) (page 2690) matrices.

### Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices.sdm import DDM
```

```
>>> A = DDM([[ZZ(1), ZZ(2)], [ZZ(3), ZZ(4)]], (2, 2), ZZ)
>>> B = DDM([[ZZ(5), ZZ(6)], [ZZ(7), ZZ(8)]], (2, 2), ZZ)
>>> A.vstack(B)
[[1, 2], [3, 4], [5, 6], [7, 8]]
```

```
>>> C = DDM([[ZZ(9), ZZ(10)], [ZZ(11), ZZ(12)]], (2, 2), ZZ)
>>> A.vstack(B, C)
[[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12]]
```

## SDM Class Reference

**class** sympy.polys.matrices.sdm.**SDM**(elemsdict, shape, domain)

Sparse matrix based on polys domain elements

This is a dict subclass and is a wrapper for a dict of dicts that supports basic matrix arithmetic  $+$ ,  $-$ ,  $*$ .

In order to create a new [SDM](#) (page 2692), a dict of dicts mapping non-zero elements to their corresponding row and column in the matrix is needed.

We also need to specify the shape and [Domain](#) (page 2504) of our [SDM](#) (page 2692) object.

We declare a 2x2 [SDM](#) (page 2692) matrix belonging to QQ domain as shown below. The 2x2 Matrix in the example is

$$A = \begin{bmatrix} 0 & \frac{1}{2} \\ 0 & 0 \end{bmatrix}$$

```
>>> from sympy.polys.matrices.sdm import SDM
>>> from sympy import QQ
>>> elemsdict = {0:{1:QQ(1, 2)}}
>>> A = SDM(elemsdict, (2, 2), QQ)
>>> A
{0: {1: 1/2}}
```

We can manipulate [SDM](#) (page 2692) the same way as a Matrix class

```
>>> from sympy import ZZ
>>> A = SDM({0:{1: ZZ(2)}}, 1:{0:ZZ(1)}}, (2, 2), ZZ)
>>> B = SDM({0:{0: ZZ(3)}}, 1:{1:ZZ(4)}}, (2, 2), ZZ)
>>> A + B
{0: {0: 3, 1: 2}, 1: {0: 1, 1: 4}}
```

Multiplication

```
>>> A*B
{0: {1: 8}, 1: {0: 3}}
>>> A*ZZ(2)
{0: {1: 4}, 1: {0: 2}}
```

**add(B)**

Adds two [SDM](#) (page 2692) matrices

### Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices.sdm import SDM
>>> A = SDM({0:{1: ZZ(2)}}, 1:{0:ZZ(1)}}, (2, 2), ZZ)
>>> B = SDM({0:{0: ZZ(3)}}, 1:{1:ZZ(4)}}, (2, 2), ZZ)
>>> A.add(B)
{0: {0: 3, 1: 2}, 1: {0: 1, 1: 4}}
```

**charpoly()**

Returns the coefficients of the characteristic polynomial of the *SDM* (page 2692) matrix. These elements will be domain elements. The domain of the elements will be same as domain of the *SDM* (page 2692).

**Examples**

```
>>> from sympy import QQ, Symbol
>>> from sympy.polys.matrices.sdm import SDM
>>> from sympy.polys import Poly
>>> A = SDM({0:{0:QQ(1), 1:QQ(2)}, 1:{0:QQ(3), 1:QQ(4)}}), (2, 2), QQ)
>>> A.charpoly()
[1, -5, -2]
```

We can create a polynomial using the coefficients using *Poly* (page 2378)

```
>>> x = Symbol('x')
>>> p = Poly(A.charpoly(), x, domain=A.domain)
>>> p
Poly(x**2 - 5*x - 2, x, domain='QQ')
```

**convert\_to(K)**

Converts the *Domain* (page 2504) of a *SDM* (page 2692) matrix to K

**Examples**

```
>>> from sympy import ZZ, QQ
>>> from sympy.polys.matrices.sdm import SDM
>>> A = SDM({0:{1: ZZ(2)}, 1:{0:ZZ(1)}}), (2, 2), ZZ)
>>> A.convert_to(QQ)
{0: {1: 2}, 1: {0: 1}}
```

**copy()**

Returns the copy of a *SDM* (page 2692) object

**Examples**

```
>>> from sympy.polys.matrices.sdm import SDM
>>> from sympy import QQ
>>> elemsdict = {0:{1:QQ(2)}, 1:{}}
>>> A = SDM(elemsdict, (2, 2), QQ)
>>> B = A.copy()
>>> B
{0: {1: 2}, 1: {}}
```

**det()**

Returns determinant of A

### Examples

```
>>> from sympy import QQ
>>> from sympy.polys.matrices.sdm import SDM
>>> A = SDM({0:{0:QQ(1), 1:QQ(2)}, 1:{0:QQ(3), 1:QQ(4)}}), (2, 2), QQ)
>>> A.det()
-2
```

**classmethod** `eye(shape, domain)`

Returns a identity *SDM* (page 2692) matrix of dimensions size x size, belonging to the specified domain

### Examples

```
>>> from sympy.polys.matrices.sdm import SDM
>>> from sympy import QQ
>>> I = SDM.eye((2, 2), QQ)
>>> I
{0: {0: 1}, 1: {1: 1}}
```

**classmethod** `from_ddm(ddm)`

converts object of *DDM* (page 2690) to *SDM* (page 2692)

### Examples

```
>>> from sympy.polys.matrices.ddm import DDM
>>> from sympy.polys.matrices.sdm import SDM
>>> from sympy import QQ
>>> ddm = DDM([[QQ(1, 2), 0], [0, QQ(3, 4)]], (2, 2), QQ)
>>> A = SDM.from_ddm(ddm)
>>> A
{0: {0: 1/2}, 1: {1: 3/4}}
```

**classmethod** `from_list(ddm, shape, domain)`

#### Parameters

**ddm:**

list of lists containing domain elements

**shape:**

Dimensions of *SDM* (page 2692) matrix

**domain:**

Represents *Domain* (page 2504) of *SDM* (page 2692) object

#### Returns

*SDM* (page 2692) containing elements of ddm

## Examples

```
>>> from sympy.polys.matrices.sdm import SDM
>>> from sympy import QQ
>>> ddm = [[QQ(1, 2), QQ(0)], [QQ(0), QQ(3, 4)]]
>>> A = SDM.from_list(ddm, (2, 2), QQ)
>>> A
{0: {0: 1/2}, 1: {1: 3/4}}
```

### hstack(\*B)

Horizontally stacks *SDM* (page 2692) matrices.

## Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices.sdm import SDM
```

```
>>> A = SDM({0: {0: ZZ(1), 1: ZZ(2)}, 1: {0: ZZ(3), 1: ZZ(4)}}), (2, 2), ZZ)
>>> B = SDM({0: {0: ZZ(5), 1: ZZ(6)}, 1: {0: ZZ(7), 1: ZZ(8)}}), (2, 2), ZZ)
>>> A.hstack(B)
{0: {0: 1, 1: 2, 2: 5, 3: 6}, 1: {0: 3, 1: 4, 2: 7, 3: 8}}
```

```
>>> C = SDM({0: {0: ZZ(9), 1: ZZ(10)}, 1: {0: ZZ(11), 1: ZZ(12)}}), (2, 2), ZZ)
>>> A.hstack(B, C)
{0: {0: 1, 1: 2, 2: 5, 3: 6, 4: 9, 5: 10}, 1: {0: 3, 1: 4, 2: 7, 3: 8, 4: 11, 5: 12}}
```

### inv()

Returns inverse of a matrix A

## Examples

```
>>> from sympy import QQ
>>> from sympy.polys.matrices.sdm import SDM
>>> A = SDM({0: {0: QQ(1), 1: QQ(2)}, 1: {0: QQ(3), 1: QQ(4)}}), (2, 2), QQ)
>>> A.inv()
{0: {0: -2, 1: 1}, 1: {0: 3/2, 1: -1/2}}
```

### is\_lower()

Says whether this matrix is lower-triangular. True can be returned even if the matrix is not square.

### is\_upper()

Says whether this matrix is upper-triangular. True can be returned even if the matrix is not square.

**is\_zero\_matrix()**

Says whether this matrix has all zero entries.

**lu()**

Returns LU decomposition for a matrix A

### Examples

```
>>> from sympy import QQ
>>> from sympy.polys.matrices.sdm import SDM
>>> A = SDM({0:{0:QQ(1), 1:QQ(2)}, 1:{0:QQ(3), 1:QQ(4)}}), (2, 2), QQ)
>>> A.lu()
({0: {0: 1}, 1: {0: 3, 1: 1}}, {0: {0: 1, 1: 2}, 1: {1: -2}}, [])
```

**lu\_solve(b)**Uses LU decomposition to solve  $Ax = b$ ,

### Examples

```
>>> from sympy import QQ
>>> from sympy.polys.matrices.sdm import SDM
>>> A = SDM({0:{0:QQ(1), 1:QQ(2)}, 1:{0:QQ(3), 1:QQ(4)}}), (2, 2), QQ)
>>> b = SDM({0:{0:QQ(1)}, 1:{0:QQ(2)}}), (2, 1), QQ)
>>> A.lu_solve(b)
{1: {0: 1/2}}
```

**matmul(B)**

Performs matrix multiplication of two SDM matrices

**Parameters****A, B: SDM to multiply****Returns**

SDM

SDM after multiplication

**Raises****DomainError**

If domain of A does not match with that of B

### Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices.sdm import SDM
>>> A = SDM({0:{1: ZZ(2)}, 1:{0:ZZ(1)}}), (2, 2), ZZ)
>>> B = SDM({0:{0:ZZ(2), 1:ZZ(3)}, 1:{0:ZZ(4)}}), (2, 2), ZZ)
>>> A.matmul(B)
{0: {0: 8}, 1: {0: 2, 1: 3}}
```



**mul(b)**

Multiplies each element of A with a scalar b

**Examples**

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices.sdm import SDM
>>> A = SDM({0:{1: ZZ(2)}, 1:{0:ZZ(1)}} , (2, 2), ZZ)
>>> A.mul(ZZ(3))
{0: {1: 6}, 1: {0: 3}}
```

**neg()**Returns the negative of a *SDM* (page 2692) matrix**Examples**

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices.sdm import SDM
>>> A = SDM({0:{1: ZZ(2)}, 1:{0:ZZ(1)}} , (2, 2), ZZ)
>>> A.neg()
{0: {1: -2}, 1: {0: -1}}
```

**classmethod new(sdm, shape, domain)****Parameters****sdm:** A dict of dicts for non-zero elements in SDM**shape:** tuple representing dimension of SDM**domain:** Represents :py:class:`~.Domain` of SDM**Returns**An *SDM* (page 2692) object**Examples**

```
>>> from sympy.polys.matrices.sdm import SDM
>>> from sympy import QQ
>>> elemsdict = {0:{1: QQ(2)}}
>>> A = SDM.new(elemsdict, (2, 2), QQ)
>>> A
{0: {1: 2}}
```

**nullspace()**Returns nullspace for a *SDM* (page 2692) matrix A

### Examples

```
>>> from sympy import QQ
>>> from sympy.polys.matrices.sdm import SDM
>>> A = SDM({0:{0:QQ(1), 1:QQ(2)}, 1:{0:QQ(2), 1:QQ(4)}}), (2, 2), QQ)
>>> A.nullspace()
({0: {0: -2, 1: 1}}, [1])
```

### rref()

Returns reduced-row echelon form and list of pivots for the [SDM](#) (page 2692)

### Examples

```
>>> from sympy import QQ
>>> from sympy.polys.matrices.sdm import SDM
>>> A = SDM({0:{0:QQ(1), 1:QQ(2)}, 1:{0:QQ(2), 1:QQ(4)}}), (2, 2), QQ)
>>> A.rref()
({0: {0: 1, 1: 2}}, [0])
```

### scc()

Strongly connected components of a square matrix A.

### Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices.sdm import SDM
>>> A = SDM({0:{0:ZZ(2)}, 1:{1:ZZ(1)}}), (2, 2), ZZ)
>>> A.scc()
[[0], [1]]
```

### See also:

[sympy.polys.matrices.domainmatrix.DomainMatrix.scc](#) (page 2684)

### sub(B)

Subtracts two [SDM](#) (page 2692) matrices

### Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices.sdm import SDM
>>> A = SDM({0:{1:ZZ(2)}, 1:{0:ZZ(1)}}), (2, 2), ZZ)
>>> B = SDM({0:{0:ZZ(3)}, 1:{1:ZZ(4)}}), (2, 2), ZZ)
>>> A.sub(B)
{0: {0: -3, 1: 2}, 1: {0: 1, 1: -4}}
```

### to\_ddm()

Convert a [SDM](#) (page 2692) object to a [DDM](#) (page 2690) object

## Examples

```
>>> from sympy.polys.matrices.sdm import SDM
>>> from sympy import QQ
>>> A = SDM({0:{1:QQ(2)}}, 1:{}}, (2, 2), QQ)
>>> A.to_ddm()
[[0, 2], [0, 0]]
```

### to\_list()

Converts a *SDM* (page 2692) object to a list

## Examples

```
>>> from sympy.polys.matrices.sdm import SDM
>>> from sympy import QQ
>>> elemsdict = {0:{1:QQ(2)}}, 1:{}}
>>> A = SDM(elemsdict, (2, 2), QQ)
>>> A.to_list()
[[0, 2], [0, 0]]
```

### transpose()

Returns the transpose of a *SDM* (page 2692) matrix

## Examples

```
>>> from sympy.polys.matrices.sdm import SDM
>>> from sympy import QQ
>>> A = SDM({0:{1:QQ(2)}}, 1:{}}, (2, 2), QQ)
>>> A.transpose()
{1: {0: 2}}
```

### vstack(\*B)

Vertically stacks *SDM* (page 2692) matrices.

## Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices.sdm import SDM

>>> A = SDM({0: {0: ZZ(1), 1: ZZ(2)}, 1: {0: ZZ(3), 1: ZZ(4)}}), (2, 2), ZZ)
>>> B = SDM({0: {0: ZZ(5), 1: ZZ(6)}, 1: {0: ZZ(7), 1: ZZ(8)}}), (2, 2), ZZ)
>>> A.vstack(B)
{0: {0: 1, 1: 2}, 1: {0: 3, 1: 4}, 2: {0: 5, 1: 6}, 3: {0: 7, 1: 8}}
```

```
>>> C = SDM({0: {0: ZZ(9), 1: ZZ(10)}, 1: {0: ZZ(11), 1: ZZ(12)}}), (2,
↳ 2), ZZ)
>>> A.vstack(B, C)
{0: {0: 1, 1: 2}, 1: {0: 3, 1: 4}, 2: {0: 5, 1: 6}, 3: {0: 7, 1: 8},
↳ 4: {0: 9, 1: 10}, 5: {0: 11, 1: 12}}
```

**classmethod zeros**(*shape, domain*)

Returns a [SDM](#) (page 2692) of size *shape*, belonging to the specified domain

In the example below we declare a matrix A where,

$$A := \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

```
>>> from sympy.polys.matrices.sdm import SDM
>>> from sympy import QQ
>>> A = SDM.zeros((2, 3), QQ)
>>> A
{}

```

## Normal Forms

`sympy.polys.matrices.normalforms.smith_normal_form(m)`

Return the Smith Normal Form of a matrix *m* over the ring *domain*. This will only work if the ring is a principal ideal domain.

## Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy.polys.matrices.normalforms import smith_normal_form
>>> m = DomainMatrix([[ZZ(12), ZZ(6), ZZ(4)],
... [ZZ(3), ZZ(9), ZZ(6)],
... [ZZ(2), ZZ(16), ZZ(14)]]), (3, 3), ZZ)
>>> print(smith_normal_form(m).to_Matrix())
Matrix([[1, 0, 0], [0, 10, 0], [0, 0, -30]])

```

`sympy.polys.matrices.normalforms.hermite_normal_form(A, *, D=None,  
check_rank=False)`

Compute the Hermite Normal Form of [DomainMatrix](#) (page 2671) *A* over [ZZ](#) (page 2525).

### Parameters

**A** :  $m \times n$  [DomainMatrix](#) over [ZZ](#) (page 2525).

**D** : [ZZ](#) (page 2525), optional

Let *W* be the HNF of *A*. If known in advance, a positive integer *D* being any multiple of  $\det(W)$  may be provided. In this case, if *A* also has rank *m*, then we may use an alternative algorithm that works mod *D* in order to prevent coefficient explosion.

**check\_rank** : boolean, optional (default=False)

The basic assumption is that, if you pass a value for  $D$ , then you already believe that  $A$  has rank  $m$ , so we do not waste time checking it for you. If you do want this to be checked (and the ordinary, non-modulo  $D$  algorithm to be used if the check fails), then set `check_rank` to `True`.

### Returns

[`DomainMatrix`](#) (page 2671)

The HNF of matrix  $A$ .

### Raises

#### **DMDomainError**

If the domain of the matrix is not [`ZZ`](#) (page 2525), or if  $D$  is given but is not in [`ZZ`](#) (page 2525).

#### **DMShapeError**

If the mod  $D$  algorithm is used but the matrix has more rows than columns.

### Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy.polys.matrices.normalforms import hermite_normal_form
>>> m = DomainMatrix([[ZZ(12), ZZ(6), ZZ(4)],
... [ZZ(3), ZZ(9), ZZ(6)],
... [ZZ(2), ZZ(16), ZZ(14)]]), (3, 3), ZZ)
>>> print(hermite_normal_form(m).to_Matrix())
Matrix([[10, 0, 2], [0, 15, 3], [0, 0, 2]])
```

### References

[R694]

## Number Fields

### Introduction

Like many other computations in algebraic number theory, the splitting of rational primes can be treated by *rational* methods only. This fact is very important if computation by automatic computing machinery is considered. Only the knowledge of the irreducible polynomial  $f(x)$ , a zero of which generates the field in question, is needed.

—Olga Taussky, 1953

Concepts like number fields and algebraic numbers are essential to our understanding of algebraic number theory, but to the computer the subject is all about polynomials: the ring  $\mathbb{Q}[x]$  reduced modulo irreducible polynomials  $f(x) \in \mathbb{Q}[x]$ . It thus finds a natural home under the [`polys`](#) (page 2360) module in SymPy.

Various authors (such as Taussky, Zimmer, Pohst and Zassenhaus, or Cohen) have articulated the main goals of computational algebraic number theory in different ways, but invariably the list centers around a certain essential set of tasks. As a goal for the `numberfields` module in SymPy, we may set the following list, based on [Cohen93], Sec. 4.9.3.

For a number field  $K = \mathbb{Q}(\theta)$ , whose ring of algebraic integers is denoted  $\mathbb{Z}_K$ , compute:

1. an integral basis of  $\mathbb{Z}_K$
2. the decomposition of rational primes in  $\mathbb{Z}_K$
3.  $p$ -adic valuations for ideals and elements
4. the Galois group of the Galois closure of  $K$
5. a system of fundamental units of  $K$
6. the regulator  $R(K)$
7. the class number
8. the structure of the class group  $Cl(K)$
9. decide whether a given ideal is principal, and if so compute a generator.

As a foundation, and to support our basic ability to define and work with number fields and algebraic numbers, we also set the following problems, following [Cohen93], Sec. 4.5.

10. Given an algebraic number – expressed by radicals and rational operations, or even as a special value of a transcendental function – determine its minimal polynomial over  $\mathbb{Q}$ .
11. The Subfield Problem: Given two number fields  $\mathbb{Q}(\alpha)$ ,  $\mathbb{Q}(\beta)$  via the minimal polynomials for their generators  $\alpha$  and  $\beta$ , decide whether one field is isomorphic to a subfield of the other, and if so exhibit an embedding.
12. The Field Membership Problem: Given two algebraic numbers  $\alpha$ ,  $\beta$ , decide whether  $\alpha \in \mathbb{Q}(\beta)$ , and if so write  $\alpha = f(\beta)$  for some  $f(x) \in \mathbb{Q}[x]$ .
13. The Primitive Element Problem: Given several algebraic numbers  $\alpha_1, \dots, \alpha_m$ , compute a single algebraic number  $\theta$  such that  $\mathbb{Q}(\alpha_1, \dots, \alpha_m) = \mathbb{Q}(\theta)$ .

At present only a subset of the tasks enumerated above is yet supported in SymPy, and if you are interested in expanding support, you are encouraged to contribute! An excellent source, providing solutions to all the remaining problems (as well as those already solved) is [Cohen93].

At time of writing, the existing solutions to the above problems are found in the following places:

| Task                         | Implementation                                                |
|------------------------------|---------------------------------------------------------------|
| (1) integral basis           | <a href="#"><code>round_two()</code></a> (page 2703)          |
| (2) prime decomposition      | <a href="#"><code>prime_decomp()</code></a> (page 2705)       |
| (3) p-adic valuation         | <a href="#"><code>prime_valuation()</code></a> (page 2709)    |
| (10) find minimal polynomial | <a href="#"><code>minimal_polynomial()</code></a> (page 2710) |
| (11) subfield                | <a href="#"><code>field_isomorphism()</code></a> (page 2711)  |
| (12) field membership        | <a href="#"><code>to_number_field()</code></a> (page 2714)    |
| (13) primitive element       | <a href="#"><code>primitive_element()</code></a> (page 2712)  |

## Solving the Main Problems

### Integral Basis

`sympy.polys.numberfields.basis.round_two(T, radicals=None)`

Zassenhaus’s “Round 2” algorithm.

#### Parameters

**T** : [\*Poly\*](#) (page 2378), [\*AlgebraicField\*](#) (page 2539)

Either (1) the irreducible monic polynomial over [\*ZZ\*](#) (page 2525) defining the number field, or (2) an [\*AlgebraicField\*](#) (page 2539) representing the number field itself.

**radicals** : dict, optional

This is a way for any  $p$ -radicals (if computed) to be returned by reference. If desired, pass an empty dictionary. If the algorithm reaches the point where it computes the nilradical mod  $p$  of the ring of integers  $Z_K$ , then an  $\mathbb{F}_p$ -basis for this ideal will be stored in this dictionary under the key  $p$ . This can be useful for other algorithms, such as prime decomposition.

#### Returns

Pair (ZK, dK), where:

ZK is a [\*Submodule\*](#) (page 2726) representing the maximal order.

dK is the discriminant of the field  $K = \mathbb{Q}[x]/(T(x))$ .

## Explanation

Carry out Zassenhaus's "Round 2" algorithm on a monic irreducible polynomial  $T$  over  $\mathbb{Z}$  (page 2525). This computes an integral basis and the discriminant for the field  $K = \mathbb{Q}[x]/(T(x))$ .

Alternatively, you may pass an *AlgebraicField* (page 2539) instance, in place of the polynomial  $T$ , in which case the algorithm is applied to the minimal polynomial for the field's primitive element.

Ordinarily this function need not be called directly, as one can instead access the *maximal\_order()* (page 2545), *integral\_basis()* (page 2544), and *discriminant()* (page 2543) methods of an *AlgebraicField* (page 2539).

## Examples

Working through an *AlgebraicField*:

```
>>> from sympy import Poly, QQ
>>> from sympy.abc import x
>>> T = Poly(x ** 3 + x ** 2 - 2 * x + 8)
>>> K = QQ.alg_field_from_poly(T, "theta")
>>> print(K.maximal_order())
Submodule[[2, 0, 0], [0, 2, 0], [0, 1, 1]]/2
>>> print(K.discriminant())
-503
>>> print(K.integral_basis(fmt='sympy'))
[1, theta, theta/2 + theta**2/2]
```

Calling directly:

```
>>> from sympy import Poly
>>> from sympy.abc import x
>>> from sympy.polys.numberfields.basis import round_two
>>> T = Poly(x ** 3 + x ** 2 - 2 * x + 8)
>>> print(round_two(T))
(Submodule[[2, 0, 0], [0, 2, 0], [0, 1, 1]]/2, -503)
```

The nilradicals mod  $p$  that are sometimes computed during the Round Two algorithm may be useful in further calculations. Pass a dictionary under *radicals* to receive these:

```
>>> T = Poly(x**3 + 3*x**2 + 5)
>>> rad = {}
>>> ZK, dK = round_two(T, radicals=rad)
>>> print(rad)
{3: Submodule[[-1, 1, 0], [-1, 0, 1]]}
```

### See also:

*AlgebraicField.maximal\_order* (page 2545), *AlgebraicField.integral\_basis* (page 2544), *AlgebraicField.discriminant* (page 2543)



## References

[R713]

## Prime Decomposition

`sympy.polys.numberfields.primes.prime_decomp(p, T=None, ZK=None, dK=None, radical=None)`

Compute the decomposition of rational prime  $p$  in a number field.

### Parameters

**p** : int

The rational prime whose decomposition is desired.

**T** : *Poly* (page 2378), optional

Monic irreducible polynomial defining the number field  $K$  in which to factor. NOTE: at least one of  $T$  or  $ZK$  must be provided.

**ZK** : *Submodule* (page 2726), optional

The maximal order for  $K$ , if already known. NOTE: at least one of  $T$  or  $ZK$  must be provided.

**dK** : int, optional

The discriminant of the field  $K$ , if already known.

**radical** : *Submodule* (page 2726), optional

The nilradical mod  $p$  in the integers of  $K$ , if already known.

### Returns

List of *PrimeIdeal* (page 2706) instances.

## Explanation

Ordinarily this should be accessed through the *primes\_above()* (page 2545) method of an *AlgebraicField* (page 2539).

## Examples

```
>>> from sympy import Poly, QQ
>>> from sympy.abc import x, theta
>>> T = Poly(x**3 + x**2 - 2*x + 8)
>>> K = QQ.algebraic_field((T, theta))
>>> print(K.primes_above(2))
[[(2, x**2 + 1) e=1, f=1], [(2, (x**2 + 3*x + 2)/2) e=1, f=1],
 [(2, (3*x**2 + 3*x)/2) e=1, f=1]]
```

## References

[R714]

**class** sympy.polys.numberfields.primes.**PrimeIdeal**(*ZK, p, alpha, f, e=None*)

A prime ideal in a ring of algebraic integers.

**\_\_init\_\_**(*ZK, p, alpha, f, e=None*)

### Parameters

**ZK** : [Submodule](#) (page 2726)

The maximal order where this ideal lives.

**p** : int

The rational prime this ideal divides.

**alpha** : [PowerBasisElement](#) (page 2732)

Such that the ideal is equal to  $p \cdot \text{ZK} + \alpha \cdot \text{ZK}$ .

**f** : int

The inertia degree.

**e** : int, None, optional

The ramification index, if already known. If None, we will compute it here.

**\_\_add\_\_**(*other*)

Convert to a [Submodule](#) (page 2726) and add to another [Submodule](#) (page 2726).

**See also:**

[as\\_submodule](#) (page 2706)

**\_\_mul\_\_**(*other*)

Convert to a [Submodule](#) (page 2726) and multiply by another [Submodule](#) (page 2726) or a rational number.

**See also:**

[as\\_submodule](#) (page 2706)

**as\_submodule**()

Represent this prime ideal as a [Submodule](#) (page 2726).

### Returns

[Submodule](#) (page 2726)

Will be equal to  $\text{self.p} * \text{self.ZK} + \text{self.alpha} * \text{self.ZK}$ .

## Explanation

The *PrimeIdeal* (page 2706) class serves to bundle information about a prime ideal, such as its inertia degree, ramification index, and two-generator representation, as well as to offer helpful methods like *valuation()* (page 2709) and *test\_factor()* (page 2709).

However, in order to be added and multiplied by other ideals or rational numbers, it must first be converted into a *Submodule* (page 2726), which is a class that supports these operations.

In many cases, the user need not perform this conversion deliberately, since it is automatically performed by the arithmetic operator methods *\_\_add\_\_()* (page 2706) and *\_\_mul\_\_()* (page 2706).

Raising a *PrimeIdeal* (page 2706) to a non-negative integer power is also supported.

## Examples

```
>>> from sympy import Poly, cyclotomic_poly, prime_decomp
>>> T = Poly(cyclotomic_poly(7))
>>> P0 = prime_decomp(7, T)[0]
>>> print(P0**6 == 7*P0.ZK)
True
```

Note that, on both sides of the equation above, we had a *Submodule* (page 2726). In the next equation we recall that adding ideals yields their GCD. This time, we need a deliberate conversion to *Submodule* (page 2726) on the right:

```
>>> print(P0 + 7*P0.ZK == P0.as_submodule())
True
```

### See also:

*\_\_add\_\_* (page 2706), *\_\_mul\_\_* (page 2706)

### property *is\_inert*

Say whether the rational prime we divide is inert, i.e. stays prime in our ring of integers.

### *reduce\_ANP(a)*

Reduce an *ANP* (page 2568) to a “small representative” modulo this prime ideal.

#### Parameters

**elt** : *ANP* (page 2568)

The element to be reduced.

#### Returns

*ANP* (page 2568)

The reduced element.

### See also:

*reduce\_element* (page 2708), *reduce\_alg\_num* (page 2707), *Submodule.reduce\_element* (page 2728)

**reduce\_alg\_num(*a*)**

Reduce an [AlgebraicNumber](#) (page 988) to a “small representative” modulo this prime ideal.

**Parameters**

**elt** : [AlgebraicNumber](#) (page 988)

The element to be reduced.

**Returns**

[AlgebraicNumber](#) (page 988)

The reduced element.

**See also:**

[reduce\\_element](#) (page 2708), [reduce\\_ANP](#) (page 2707), [Submodule.reduce\\_element](#) (page 2728)

**reduce\_element(*elt*)**

Reduce a [PowerBasisElement](#) (page 2732) to a “small representative” modulo this prime ideal.

**Parameters**

**elt** : [PowerBasisElement](#) (page 2732)

The element to be reduced.

**Returns**

[PowerBasisElement](#) (page 2732)

The reduced element.

**See also:**

[reduce\\_ANP](#) (page 2707), [reduce\\_alg\\_num](#) (page 2707), [Submodule.reduce\\_element](#) (page 2728)

**repr(*field\_gen=None, just\_gens=False*)**

Print a representation of this prime ideal.

**Parameters**

**field\_gen** : [Symbol](#) (page 976), None, optional (default=None)

The symbol to use for the generator of the field. This will appear in our representation of `self.alpha`. If None, we use the variable of the defining polynomial of `self.ZK`.

**just\_gens** : bool, optional (default=False)

If True, just print the “(p, alpha)” part, showing “just the generators” of the prime ideal. Otherwise, print a string of the form “[ (p, alpha) e=..., f=... ]”, giving the ramification index and inertia degree, along with the generators.

## Examples

```
>>> from sympy import cyclotomic_poly, QQ
>>> from sympy.abc import x, zeta
>>> T = cyclotomic_poly(7, x)
>>> K = QQ.algebraic_field((T, zeta))
>>> P = K.primes_above(11)
>>> print(P[0].repr())
[(11, x**3 + 5*x**2 + 4*x - 1) e=1, f=3]
>>> print(P[0].repr(field_gen=zeta))
[(11, zeta**3 + 5*zeta**2 + 4*zeta - 1) e=1, f=3]
>>> print(P[0].repr(field_gen=zeta, just_gens=True))
(11, zeta**3 + 5*zeta**2 + 4*zeta - 1)
```

### test\_factor()

Compute a test factor for this prime ideal.

## Explanation

Write  $\mathfrak{p}$  for this prime ideal,  $p$  for the rational prime it divides. Then, for computing  $p$ -adic valuations it is useful to have a number  $\beta \in \mathbb{Z}_K$  such that  $p/\mathfrak{p} = p\mathbb{Z}_K + \beta\mathbb{Z}_K$ .

Essentially, this is the same as the number  $\Psi$  (or the “reagent”) from Kummer’s 1847 paper (*Ueber die Zerlegung...*, Crelle vol. 35) in which ideal divisors were invented.

### valuation(I)

Compute the  $p$ -adic valuation of integral ideal  $I$  at this prime ideal.

#### Parameters

**I** : *Submodule* (page 2726)

#### See also:

*prime\_valuation* (page 2709)

## p-adic Valuation

`sympy.polys.numberfields.primes.prime_valuation(I, P)`

Compute the  $P$ -adic valuation for an integral ideal  $I$ .

#### Parameters

**I** : *Submodule* (page 2726)

An integral ideal whose valuation is desired.

**P** : *PrimeIdeal* (page 2706)

The prime at which to compute the valuation.

#### Returns

int

## Examples

```
>>> from sympy import QQ
>>> from sympy.polys.numberfields import prime_valuation
>>> K = QQ.cyclotomic_field(5)
>>> P = K.primes_above(5)
>>> ZK = K.maximal_order()
>>> print(prime_valuation(25*ZK, P[0]))
8
```

## See also:

[\*PrimeIdeal.valuation\*](#) (page 2709)

## References

[R715]

## Finding Minimal Polynomials

`sympy.polys.numberfields.minpoly.minimal_polynomial(ex, x=None, compose=True,  
 polys=False, domain=None)`

Computes the minimal polynomial of an algebraic element.

### Parameters

**ex** : Expr

Element or expression whose minimal polynomial is to be calculated.

**x** : Symbol, optional

Independent variable of the minimal polynomial

**compose** : boolean, optional (default=True)

Method to use for computing minimal polynomial. If `compose=True` (default) then `_minpoly_compose` is used, if `compose=False` then groebner bases are used.

**polys** : boolean, optional (default=False)

If True returns a Poly object else an Expr object.

**domain** : Domain, optional

Ground domain

## Notes

By default `compose=True`, the minimal polynomial of the subexpressions of `ex` are computed, then the arithmetic operations on them are performed using the resultant and factorization. If `compose=False`, a bottom-up algorithm is used with `groebner`. The default algorithm stalls less frequently.

If no ground domain is given, it will be generated automatically from the expression.

## Examples

```
>>> from sympy import minimal_polynomial, sqrt, solve, QQ
>>> from sympy.abc import x, y

>>> minimal_polynomial(sqrt(2), x)
x**2 - 2
>>> minimal_polynomial(sqrt(2), x, domain=QQ.algebraic_field(sqrt(2)))
x - sqrt(2)
>>> minimal_polynomial(sqrt(2) + sqrt(3), x)
x**4 - 10*x**2 + 1
>>> minimal_polynomial(solve(x**3 + x + 3)[0], x)
x**3 + x + 3
>>> minimal_polynomial(sqrt(y), x)
x**2 - y
```

`sympy.polys.numberfields.minpoly.minpoly(ex, x=None, compose=True, polys=False, domain=None)`

This is a synonym for [`minimal\_polynomial\(\)`](#) (page 2710).

## The Subfield Problem

Functions in `polys.numberfields.subfield` solve the “Subfield Problem” and allied problems, for algebraic number fields.

Following Cohen (see [Cohen93] Section 4.5), we can define the main problem as follows:

- **Subfield Problem:**

Given two number fields  $\mathbb{Q}(\alpha)$ ,  $\mathbb{Q}(\beta)$  via the minimal polynomials for their generators  $\alpha$  and  $\beta$ , decide whether one field is isomorphic to a subfield of the other.

From a solution to this problem flow solutions to the following problems as well:

- **Primitive Element Problem:**

Given several algebraic numbers  $\alpha_1, \dots, \alpha_m$ , compute a single algebraic number  $\theta$  such that  $\mathbb{Q}(\alpha_1, \dots, \alpha_m) = \mathbb{Q}(\theta)$ .

- **Field Isomorphism Problem:**

Decide whether two number fields  $\mathbb{Q}(\alpha)$ ,  $\mathbb{Q}(\beta)$  are isomorphic.

- **Field Membership Problem:**

Given two algebraic numbers  $\alpha$ ,  $\beta$ , decide whether  $\alpha \in \mathbb{Q}(\beta)$ , and if so write  $\alpha = f(\beta)$  for some  $f(x) \in \mathbb{Q}[x]$ .

`sympy.polys.numberfields.subfield.field_isomorphism(a, b, *, fast=True)`

Find an embedding of one number field into another.

#### Parameters

**a** : *Expr* (page 947)

Any expression representing an algebraic number.

**b** : *Expr* (page 947)

Any expression representing an algebraic number.

**fast** : boolean, optional (default=True)

If True, we first attempt a potentially faster way of computing the isomorphism, falling back on a slower method if this fails. If False, we go directly to the slower method, which is guaranteed to return a result.

#### Returns

List of rational numbers, or None

If  $\mathbb{Q}(a)$  is not isomorphic to some subfield of  $\mathbb{Q}(b)$ , then return None. Otherwise, return a list of rational numbers representing an element of  $\mathbb{Q}(b)$  to which  $a$  may be mapped, in order to define a monomorphism, i.e. an isomorphism from  $\mathbb{Q}(a)$  to some subfield of  $\mathbb{Q}(b)$ . The elements of the list are the coefficients of falling powers of  $b$ .

#### Explanation

This function looks for an isomorphism from  $\mathbb{Q}(a)$  onto some subfield of  $\mathbb{Q}(b)$ . Thus, it solves the Subfield Problem.

#### Examples

```
>>> from sympy import sqrt, field_isomorphism, I
>>> print(field_isomorphism(3, sqrt(2)))
[3]
>>> print(field_isomorphism(I*sqrt(3), I*sqrt(3)/2))
[2, 0]
```

`sympy.polys.numberfields.subfield.primitive_element(extension, x=None, *, ex=False, polys=False)`

Find a single generator for a number field given by several generators.

#### Parameters

**extension** : list of *Expr* (page 947)

Each expression must represent an algebraic number  $\alpha_i$ .

**x** : *Symbol* (page 976), optional (default=None)

The desired symbol to appear in the computed minimal polynomial for the primitive element  $\theta$ . If None, we use a dummy symbol.

**ex** : boolean, optional (default=False)



If and only if `True`, compute the representation of each  $\alpha_i$  as a  $\mathbb{Q}$ -linear combination over the powers of  $\theta$ .

**polys** : boolean, optional (default=False)

If `True`, return the minimal polynomial as a *Poly* (page 2378). Otherwise return it as an *Expr* (page 947).

### Returns

Pair (f, coeffs) or triple (f, coeffs, reps), where:

f is the minimal polynomial for the primitive element. coeffs gives the primitive element as a linear combination of the given generators. reps is present if and only if argument `ex=True` was passed, and is a list of lists of rational numbers. Each list gives the coefficients of falling powers of the primitive element, to recover one of the original, given generators.

### Explanation

The basic problem is this: Given several algebraic numbers  $\alpha_1, \alpha_2, \dots, \alpha_n$ , find a single algebraic number  $\theta$  such that  $\mathbb{Q}(\alpha_1, \alpha_2, \dots, \alpha_n) = \mathbb{Q}(\theta)$ .

This function actually guarantees that  $\theta$  will be a linear combination of the  $\alpha_i$ , with non-negative integer coefficients.

Furthermore, if desired, this function will tell you how to express each  $\alpha_i$  as a  $\mathbb{Q}$ -linear combination of the powers of  $\theta$ .

### Examples

```
>>> from sympy import primitive_element, sqrt, S, minpoly, simplify
>>> from sympy.abc import x
>>> f, lincomb, reps = primitive_element([sqrt(2), sqrt(3)], x, ex=True)
```

Then `lincomb` tells us the primitive element as a linear combination of the given generators `sqrt(2)` and `sqrt(3)`.

```
>>> print(lincomb)
[1, 1]
```

This means the primitive element is  $\sqrt{2} + \sqrt{3}$ . Meanwhile `f` is the minimal polynomial for this primitive element.

```
>>> print(f)
x**4 - 10*x**2 + 1
>>> print(minpoly(sqrt(2) + sqrt(3), x))
x**4 - 10*x**2 + 1
```

Finally, `reps` (which was returned only because we set keyword arg `ex=True`) tells us how to recover each of the generators  $\sqrt{2}$  and  $\sqrt{3}$  as  $\mathbb{Q}$ -linear combinations of the powers of the primitive element  $\sqrt{2} + \sqrt{3}$ .

```
>>> print([S(r) for r in reps[0]])
[1/2, 0, -9/2, 0]
>>> theta = sqrt(2) + sqrt(3)
>>> print(simplify(theta**3/2 - 9*theta/2))
sqrt(2)
>>> print([S(r) for r in reps[1]])
[-1/2, 0, 11/2, 0]
>>> print(simplify(-theta**3/2 + 11*theta/2))
sqrt(3)
```

`sympy.polys.numberfields.subfield.to_number_field(extension, theta=None, *,  
gen=None, alias=None)`

Express one algebraic number in the field generated by another.

#### Parameters

**extension** : *Expr* (page 947) or list of *Expr* (page 947)

Either the algebraic number that is to be expressed in the other field, or else a list of algebraic numbers, a primitive element for which is to be expressed in the other field.

**theta** : *Expr* (page 947), None, optional (default=None)

If an *Expr* (page 947) representing an algebraic number, behavior is as described under **Explanation**. If None, then this function reduces to a shorthand for calling *primitive\_element()* (page 2712) on *extension* and turning the computed primitive element into an *AlgebraicNumber* (page 988).

**gen** : *Symbol* (page 976), None, optional (default=None)

If provided, this will be used as the generator symbol for the minimal polynomial in the returned *AlgebraicNumber* (page 988).

**alias** : str, *Symbol* (page 976), None, optional (default=None)

If provided, this will be used as the alias symbol for the returned *AlgebraicNumber* (page 988).

#### Returns

*AlgebraicNumber*

Belonging to  $\mathbb{Q}(\theta)$  and equaling  $\eta$ .

#### Raises

**IsomorphismFailed**

If  $\eta \notin \mathbb{Q}(\theta)$ .

## Explanation

Given two algebraic numbers  $\eta, \theta$ , this function either expresses  $\eta$  as an element of  $\mathbb{Q}(\theta)$ , or else raises an exception if  $\eta \notin \mathbb{Q}(\theta)$ .

This function is essentially just a convenience, utilizing [field\\_isomorphism\(\)](#) (page 2711) (our solution of the Subfield Problem) to solve this, the Field Membership Problem.

As an additional convenience, this function allows you to pass a list of algebraic numbers  $\alpha_1, \alpha_2, \dots, \alpha_n$  instead of  $\eta$ . It then computes  $\eta$  for you, as a solution of the Primitive Element Problem, using [primitive\\_element\(\)](#) (page 2712) on the list of  $\alpha_i$ .

## Examples

```
>>> from sympy import sqrt, to_number_field
>>> eta = sqrt(2)
>>> theta = sqrt(2) + sqrt(3)
>>> a = to_number_field(eta, theta)
>>> print(type(a))
<class 'sympy.core.numbers.AlgebraicNumber'>
>>> a.root
sqrt(2) + sqrt(3)
>>> print(a)
sqrt(2)
>>> a.coeffs()
[1/2, 0, -9/2, 0]
```

We get an [AlgebraicNumber](#) (page 988), whose `.root` is  $\theta$ , whose value is  $\eta$ , and whose `.coeffs()` show how to write  $\eta$  as a  $\mathbb{Q}$ -linear combination in falling powers of  $\theta$ .

**See also:**

[field\\_isomorphism](#) (page 2711), [primitive\\_element](#) (page 2712)

## Internals

### Algebraic number fields

Algebraic number fields are represented in SymPy by the [AlgebraicField](#) (page 2539) class, which is a part of [the polynomial domains system](#) (page 2504).

### Representing algebraic numbers

There are several different ways to represent algebraic numbers, and different forms may be preferable for different computational tasks. See [Cohen93], Sec. 4.2.

## As number field elements

In SymPy, there is a distinction between number and expression classes defined in the [sympy.core.numbers](#) (page 981) module on the one hand, and domains and domain elements defined in the [polys](#) (page 2360) module on the other. This is explained in more detail [here](#) (page 2477).

When it comes to algebraic numbers, the [sympy.core.numbers](#) (page 981) module offers the [AlgebraicNumber](#) (page 988) class, while the [polys](#) (page 2360) module offers the [ANP](#) (page 2568) class. This is the type of domain elements belonging to the [AlgebraicField](#) (page 2539) domain.

## As elements of finitely-generated modules

In computational algebraic number theory, finitely-generated  $\mathbb{Z}$ -modules are of central importance. For example, every [order](#) and every [ideal](#) is such a module.

In particular, the maximal order – or [ring of integers](#) – in a number field is a finitely-generated  $\mathbb{Z}$ -module, whose generators form an [integral basis](#) for the field.

Classes allowing us to represent such modules, and their elements, are provided in the [modules](#) (page 2716) module. Here, the [ModuleElement](#) (page 2729) class provides another way to represent algebraic numbers.

## Finitely-generated modules

Modules in number fields.

The classes defined here allow us to work with finitely generated, free modules, whose generators are algebraic numbers.

There is an abstract base class called [Module](#) (page 2719), which has two concrete subclasses, [PowerBasis](#) (page 2725) and [Submodule](#) (page 2726).

Every module is defined by its basis, or set of generators:

- For a [PowerBasis](#) (page 2725), the generators are the first  $n$  powers (starting with the zeroth) of an algebraic integer  $\theta$  of degree  $n$ . The [PowerBasis](#) (page 2725) is constructed by passing either the minimal polynomial of  $\theta$ , or an [AlgebraicField](#) (page 2539) having  $\theta$  as its primitive element.
- For a [Submodule](#) (page 2726), the generators are a set of  $\mathbb{Q}$ -linear combinations of the generators of another module. That other module is then the “parent” of the [Submodule](#) (page 2726). The coefficients of the  $\mathbb{Q}$ -linear combinations may be given by an integer matrix, and a positive integer denominator. Each column of the matrix defines a generator.

```
>>> from sympy.polys import Poly, cyclotomic_poly, ZZ
>>> from sympy.abc import x
>>> from sympy.polys.matrices import DomainMatrix, DM
>>> from sympy.polys.numberfields.modules import PowerBasis
>>> T = Poly(cyclotomic_poly(5, x))
>>> A = PowerBasis(T)
>>> print(A)
```

(continues on next page)

(continued from previous page)

```

PowerBasis(x**4 + x**3 + x**2 + x + 1)
>>> B = A.submodule_from_matrix(2 * DomainMatrix.eye(4, ZZ), denom=3)
>>> print(B)
Submodule[[2, 0, 0, 0], [0, 2, 0, 0], [0, 0, 2, 0], [0, 0, 0, 2]]/3
>>> print(B.parent)
PowerBasis(x**4 + x**3 + x**2 + x + 1)

```

Thus, every module is either a [PowerBasis](#) (page 2725), or a [Submodule](#) (page 2726), some ancestor of which is a [PowerBasis](#) (page 2725). (If  $S$  is a [Submodule](#) (page 2726), then its ancestors are  $S.parent$ ,  $S.parent.parent$ , and so on).

The [ModuleElement](#) (page 2729) class represents a linear combination of the generators of any module. Critically, the coefficients of this linear combination are not restricted to be integers, but may be any rational numbers. This is necessary so that any and all algebraic integers be representable, starting from the power basis in a primitive element  $\theta$  for the number field in question. For example, in a quadratic field  $\mathbb{Q}(\sqrt{d})$  where  $d \equiv 1 \pmod{4}$ , a denominator of 2 is needed.

A [ModuleElement](#) (page 2729) can be constructed from an integer column vector and a denominator:

```

>>> U = Poly(x**2 - 5)
>>> M = PowerBasis(U)
>>> e = M(DM([[1]], [1]], ZZ), denom=2)
>>> print(e)
[1, 1]/2
>>> print(e.module)
PowerBasis(x**2 - 5)

```

The [PowerBasisElement](#) (page 2732) class is a subclass of [ModuleElement](#) (page 2729) that represents elements of a [PowerBasis](#) (page 2725), and adds functionality pertinent to elements represented directly over powers of the primitive element  $\theta$ .

## Arithmetic with module elements

While a [ModuleElement](#) (page 2729) represents a linear combination over the generators of a particular module, recall that every module is either a [PowerBasis](#) (page 2725) or a descendant (along a chain of [Submodule](#) (page 2726) objects) thereof, so that in fact every [ModuleElement](#) (page 2729) represents an algebraic number in some field  $\mathbb{Q}(\theta)$ , where  $\theta$  is the defining element of some [PowerBasis](#) (page 2725). It thus makes sense to talk about the number field to which a given [ModuleElement](#) (page 2729) belongs.

This means that any two [ModuleElement](#) (page 2729) instances can be added, subtracted, multiplied, or divided, provided they belong to the same number field. Similarly, since  $\mathbb{Q}$  is a subfield of every number field, any [ModuleElement](#) (page 2729) may be added, multiplied, etc. by any rational number.

```

>>> from sympy import QQ
>>> from sympy.polys.numberfields.modules import to_col
>>> T = Poly(cyclotomic_poly(5))
>>> A = PowerBasis(T)
>>> C = A.submodule_from_matrix(3 * DomainMatrix.eye(4, ZZ))

```

(continues on next page)

(continued from previous page)

```

>>> e = A(to_col([0, 2, 0, 0]), denom=3)
>>> f = A(to_col([0, 0, 0, 7]), denom=5)
>>> g = C(to_col([1, 1, 1, 1]))
>>> e + f
[0, 10, 0, 21]/15
>>> e - f
[0, 10, 0, -21]/15
>>> e - g
[-9, -7, -9, -9]/3
>>> e + QQ(7, 10)
[21, 20, 0, 0]/30
>>> e * f
[-14, -14, -14, -14]/15
>>> e ** 2
[0, 0, 4, 0]/9
>>> f // g
[7, 7, 7, 7]/15
>>> f * QQ(2, 3)
[0, 0, 0, 14]/15

```

However, care must be taken with arithmetic operations on *ModuleElement* (page 2729), because the module *C* to which the result will belong will be the nearest common ancestor (NCA) of the modules *A*, *B* to which the two operands belong, and *C* may be different from either or both of *A* and *B*.

```

>>> A = PowerBasis(T)
>>> B = A.submodule_from_matrix(2 * DomainMatrix.eye(4, ZZ))
>>> C = A.submodule_from_matrix(3 * DomainMatrix.eye(4, ZZ))
>>> print((B(0) * C(0)).module == A)
True

```

Before the arithmetic operation is performed, copies of the two operands are automatically converted into elements of the NCA (the operands themselves are not modified). This upward conversion along an ancestor chain is easy: it just requires the successive multiplication by the defining matrix of each *Submodule* (page 2726).

Conversely, downward conversion, i.e. representing a given *ModuleElement* (page 2729) in a submodule, is also supported – namely by the *represent()* (page 2729) method – but is not guaranteed to succeed in general, since the given element may not belong to the submodule. The main circumstance in which this issue tends to arise is with multiplication, since modules, while closed under addition, need not be closed under multiplication.

## Multiplication

Generally speaking, a module need not be closed under multiplication, i.e. need not form a ring. However, many of the modules we work with in the context of number fields are in fact rings, and our classes do support multiplication.

Specifically, any *Module* (page 2719) can attempt to compute its own multiplication table, but this does not happen unless an attempt is made to multiply two *ModuleElement* (page 2729) instances belonging to it.

```

>>> A = PowerBasis(T)
>>> print(A._mult_tab is None)
True
>>> a = A(0)*A(1)
>>> print(A._mult_tab is None)
False

```

Every *PowerBasis* (page 2725) is, by its nature, closed under multiplication, so instances of *PowerBasis* (page 2725) can always successfully compute their multiplication table.

When a *Submodule* (page 2726) attempts to compute its multiplication table, it converts each of its own generators into elements of its parent module, multiplies them there, in every possible pairing, and then tries to represent the results in itself, i.e. as  $\mathbb{Z}$ -linear combinations over its own generators. This will succeed if and only if the submodule is in fact closed under multiplication.

## Module Homomorphisms

Many important number theoretic algorithms require the calculation of the kernel of one or more module homomorphisms. Accordingly we have several lightweight classes, *ModuleHomomorphism* (page 2733), *ModuleEndomorphism* (page 2734), *InnerEndomorphism* (page 2734), and *EndomorphismRing* (page 2734), which provide the minimal necessary machinery to support this.

## Class Reference

**class** sympy.polys.numberfields.modules.**Module**

Generic finitely-generated module.

This is an abstract base class, and should not be instantiated directly. The two concrete subclasses are *PowerBasis* (page 2725) and *Submodule* (page 2726).

Every *Submodule* (page 2726) is derived from another module, referenced by its parent attribute. If *S* is a submodule, then we refer to *S.parent*, *S.parent.parent*, and so on, as the “ancestors” of *S*. Thus, every *Module* (page 2719) is either a *PowerBasis* (page 2725) or a *Submodule* (page 2726), some ancestor of which is a *PowerBasis* (page 2725).

**\_\_call\_\_**(*spec*, *denom*=1)

Generate a *ModuleElement* (page 2729) belonging to this module.

### Parameters

**spec** : *DomainMatrix* (page 2671), int

Specifies the numerators of the coefficients of the *ModuleElement* (page 2729). Can be either a column vector over *ZZ* (page 2525), whose length must equal the number *n* of generators of this module, or else an integer *j*,  $0 \leq j < n$ , which is a shorthand for column *j* of *I<sub>n</sub>*, the  $n \times n$  identity matrix.

**denom** : int, optional (default=1)

Denominator for the coefficients of the *ModuleElement* (page 2729).

### Returns

*ModuleElement* (page 2729)

The coefficients are the entries of the *spec* vector, divided by *denom*.

### Examples

```
>>> from sympy.polys import Poly, cyclotomic_poly
>>> from sympy.polys.numberfields.modules import PowerBasis, to_col
>>> T = Poly(cyclotomic_poly(5))
>>> A = PowerBasis(T)
>>> e = A(to_col([1, 2, 3, 4]), denom=3)
>>> print(e)
[1, 2, 3, 4]/3
>>> f = A(2)
>>> print(f)
[0, 0, 1, 0]
```

**ancestors**(*include\_self=False*)

Return the list of ancestor modules of this module, from the foundational *PowerBasis* (page 2725) downward, optionally including *self*.

**See also:**

*Module* (page 2719)

**basis\_elements**()

Get list of *ModuleElement* (page 2729) being the generators of this module.

**element\_from\_rational**(*a*)

Return a *ModuleElement* (page 2729) representing a rational number.

**Parameters**

**a** : int, *ZZ* (page 2525), *QQ* (page 2529)

**Returns**

*ModuleElement* (page 2729)

### Explanation

The returned *ModuleElement* (page 2729) will belong to the first module on this module's ancestor chain (including this module itself) that starts with unity.

### Examples

```
>>> from sympy.polys import Poly, cyclotomic_poly, QQ
>>> from sympy.polys.numberfields.modules import PowerBasis
>>> T = Poly(cyclotomic_poly(5))
>>> A = PowerBasis(T)
>>> a = A.element_from_rational(QQ(2, 3))
>>> print(a)
[2, 0, 0, 0]/3
```

**endomorphism\_ring**()

Form the *EndomorphismRing* (page 2734) for this module.



**is\_compat\_col**(*col*)

Say whether *col* is a suitable column vector for this module.

**mult\_tab**()

Get the multiplication table for this module (if closed under mult).

**Returns**

dict of dict of lists

**Raises**

**ClosureFailure**

If the module is not closed under multiplication.

## Explanation

Computes a dictionary *M* of dictionaries of lists, representing the upper triangular half of the multiplication table.

In other words, if  $0 \leq i \leq j < \text{self.n}$ , then *M*[*i*][*j*] is the list *c* of coefficients such that  $g[i] * g[j] == \sum(c[k]*g[k], k \text{ in range}(\text{self.n}))$ , where *g* is the list of generators of this module.

If  $j < i$  then *M*[*i*][*j*] is undefined.

## Examples

```
>>> from sympy.polys import Poly, cyclotomic_poly
>>> from sympy.polys.numberfields.modules import PowerBasis
>>> T = Poly(cyclotomic_poly(5))
>>> A = PowerBasis(T)
>>> print(A.mult_tab())
{0: {0: [1, 0, 0, 0], 1: [0, 1, 0, 0], 2: [0, 0, 1, 0], 3: [0, 0, 0, 1],
 ↪ 0, 1]},
 1: {1: [0, 0, 1, 0], 2: [0, 0, 0, 1], 3: [-1, -1, -1, -1]},
 ↪ 1, -1, -1]},
 2: {2: [-1, -1, -1, -1], 3: [1, 0, 0, 1]},
 ↪ 0, 0]},
 3: {3: [0, 1, 0, 1], 0: [1, 0, 0, 0], 1: [0, 1, 0, 0], 2: [0, 0, 1, 0]},
 ↪ 0, 0]}}
```

**property n**

The number of generators of this module.

**nearest\_common\_ancestor**(*other*)

Locate the nearest common ancestor of this module and another.

**Returns**

[Module](#) (page 2719), None

**See also:**

[Module](#) (page 2719)

**property number\_field**

Return the associated *AlgebraicField* (page 2539), if any.

**Returns**

*AlgebraicField* (page 2539), None

**Explanation**

A *PowerBasis* (page 2725) can be constructed on a *Poly* (page 2378)  $f$  or on an *AlgebraicField* (page 2539)  $K$ . In the latter case, the *PowerBasis* (page 2725) and all its descendant modules will return  $K$  as their `.number_field` property, while in the former case they will all return None.

**one()**

Return a *ModuleElement* (page 2729) representing unity, and belonging to the first ancestor of this module (including itself) that starts with unity.

**property parent**

The parent module, if any, for this module.

**Returns**

*Module* (page 2719), None

**Explanation**

For a *Submodule* (page 2726) this is its parent attribute; for a *PowerBasis* (page 2725) this is None.

**See also:**

*Module* (page 2719)

**power\_basis\_ancestor()**

Return the *PowerBasis* (page 2725) that is an ancestor of this module.

**See also:**

*Module* (page 2719)

**represent(elt)**

Represent a module element as an integer-linear combination over the generators of this module.

**Parameters**

**elt** : *ModuleElement* (page 2729)

The module element to be represented. Must belong to some ancestor module of this module (including this module itself).

**Returns**

*DomainMatrix* (page 2671) over *ZZ* (page 2525)

This will be a column vector, representing the coefficients of a linear combination of this module's generators, which equals the given element.

**Raises**

**ClosureFailure**

If the given element cannot be represented as a [ZZ](#) (page 2525)-linear combination over this module.

### Explanation

In our system, to “represent” always means to write a [ModuleElement](#) (page 2729) as a [ZZ](#) (page 2525)-linear combination over the generators of the present [Module](#) (page 2719). Furthermore, the incoming [ModuleElement](#) (page 2729) must belong to an ancestor of the present [Module](#) (page 2719) (or to the present [Module](#) (page 2719) itself).

The most common application is to represent a [ModuleElement](#) (page 2729) in a [Submodule](#) (page 2726). For example, this is involved in computing multiplication tables.

On the other hand, representing in a [PowerBasis](#) (page 2725) is an odd case, and one which tends not to arise in practice, except for example when using a [ModuleEndomorphism](#) (page 2734) on a [PowerBasis](#) (page 2725).

In such a case, (1) the incoming [ModuleElement](#) (page 2729) must belong to the [PowerBasis](#) (page 2725) itself (since the latter has no proper ancestors) and (2) it is “representable” iff it belongs to  $\mathbb{Z}[\theta]$  (although generally a [PowerBasisElement](#) (page 2732) may represent any element of  $\mathbb{Q}(\theta)$ , i.e. any algebraic number).

### Examples

```
>>> from sympy import Poly, cyclotomic_poly
>>> from sympy.polys.numberfields.modules import PowerBasis, to_col
>>> from sympy.abc import zeta
>>> T = Poly(cyclotomic_poly(5))
>>> A = PowerBasis(T)
>>> a = A(to_col([2, 4, 6, 8]))
```

The [ModuleElement](#) (page 2729) `a` has all even coefficients. If we represent `a` in the submodule `B = 2*A`, the coefficients in the column vector will be halved:

```
>>> B = A.submodule_from_gens([2*A(i) for i in range(4)])
>>> b = B.represent(a)
>>> print(b.transpose())
DomainMatrix([[1, 2, 3, 4]], (1, 4), ZZ)
```

However, the element of `B` so defined still represents the same algebraic number:

```
>>> print(a.poly(zeta).as_expr())
8*zeta**3 + 6*zeta**2 + 4*zeta + 2
>>> print(B(b).over_power_basis().poly(zeta).as_expr())
8*zeta**3 + 6*zeta**2 + 4*zeta + 2
```

### See also:

[Submodule.represent](#) (page 2729), [PowerBasis.represent](#) (page 2725)

### `starts_with_unity()`

Say whether the module’s first generator equals unity.

**submodule\_from\_gens**(*gens*, *hnf*=True, *hnf\_modulus*=None)

Form the submodule generated by a list of [ModuleElement](#) (page 2729) belonging to this module.

**Parameters**

**gens** : list of [ModuleElement](#) (page 2729) belonging to this module.

**hnf** : boolean, optional (default=True)

If True, we will reduce the matrix into Hermite Normal Form before forming the [Submodule](#) (page 2726).

**hnf\_modulus** : int, None, optional (default=None)

Modulus for use in the HNF reduction algorithm. See [hermite\\_normal\\_form\(\)](#) (page 2700).

**Returns**

[Submodule](#) (page 2726)

**Examples**

```
>>> from sympy.polys import Poly, cyclotomic_poly
>>> from sympy.polys.numberfields.modules import PowerBasis
>>> T = Poly(cyclotomic_poly(5))
>>> A = PowerBasis(T)
>>> gens = [A(0), 2*A(1), 3*A(2), 4*A(3)//5]
>>> B = A.submodule_from_gens(gens)
>>> print(B)
Submodule[[5, 0, 0, 0], [0, 10, 0, 0], [0, 0, 15, 0], [0, 0, 0, 4]]/5
```

**See also:**

[submodule\\_from\\_matrix](#) (page 2724)

**submodule\_from\_matrix**(*B*, *denom*=1)

Form the submodule generated by the elements of this module indicated by the columns of a matrix, with an optional denominator.

**Parameters**

**B** : [DomainMatrix](#) (page 2671) over [ZZ](#) (page 2525)

Each column gives the numerators of the coefficients of one generator of the submodule. Thus, the number of rows of *B* must equal the number of generators of the present module.

**denom** : int, optional (default=1)

Common denominator for all generators of the submodule.

**Returns**

[Submodule](#) (page 2726)

**Raises**

**ValueError**

If the given matrix *B* is not over [ZZ](#) (page 2525) or its number of rows does not equal the number of generators of the present module.

## Examples

```
>>> from sympy.polys import Poly, cyclotomic_poly, ZZ
>>> from sympy.polys.matrices import DM
>>> from sympy.polys.numberfields.modules import PowerBasis
>>> T = Poly(cyclotomic_poly(5))
>>> A = PowerBasis(T)
>>> B = A.submodule_from_matrix(DM([
... [0, 10, 0, 0],
... [0, 0, 7, 0],
...], ZZ).transpose(), denom=15)
>>> print(B)
Submodule[[0, 10, 0, 0], [0, 0, 7, 0]]/15
```

### See also:

[submodule\\_from\\_gens](#) (page 2723)

### `whole_submodule()`

Return a submodule equal to this entire module.

## Explanation

This is useful when you have a [PowerBasis](#) (page 2725) and want to turn it into a [Submodule](#) (page 2726) (in order to use methods belonging to the latter).

### `zero()`

Return a [ModuleElement](#) (page 2729) representing zero.

### `class sympy.polys.numberfields.modules.PowerBasis(T)`

The module generated by the powers of an algebraic integer.

### `__init__(T)`

#### Parameters

**T** : [Poly](#) (page 2378), [AlgebraicField](#) (page 2539)

Either (1) the monic, irreducible, univariate polynomial over [ZZ](#) (page 2525), a root of which is the generator of the power basis, or (2) an [AlgebraicField](#) (page 2539) whose primitive element is the generator of the power basis.

### `element_from_ANP(a)`

Convert an ANP into a PowerBasisElement.

### `element_from_alg_num(a)`

Convert an AlgebraicNumber into a PowerBasisElement.

### `element_from_poly(f)`

Produce an element of this module, representing  $f$  after reduction mod our defining minimal polynomial.

#### Parameters

**f** : [Poly](#) (page 2378) over [ZZ](#) (page 2525) in same var as our defining poly.

#### Returns

[PowerBasisElement](#) (page 2732)

**represent**(*elt*)

Represent a module element as an integer-linear combination over the generators of this module.

**See also:**

[Module.represent](#) (page 2722), [Submodule.represent](#) (page 2729)

**class** sympy.polys.numberfields.modules.**Submodule**(*parent, matrix, denom=1, mult\_tab=None*)

A submodule of another module.

**\_\_init\_\_**(*parent, matrix, denom=1, mult\_tab=None*)

**Parameters**

**parent** : [Module](#) (page 2719)

The module from which this one is derived.

**matrix** : [DomainMatrix](#) (page 2671) over [ZZ](#) (page 2525)

The matrix whose columns define this submodule's generators as linear combinations over the parent's generators.

**denom** : int, optional (default=1)

Denominator for the coefficients given by the matrix.

**mult\_tab** : dict, None, optional

If already known, the multiplication table for this module may be supplied.

**property QQ\_matrix**

[DomainMatrix](#) (page 2671) over [QQ](#) (page 2529), equal to `self.matrix / self.denom`, and guaranteed to be dense.

**Returns**

[DomainMatrix](#) (page 2671) over [QQ](#) (page 2529)

**Explanation**

Depending on how it is formed, a [DomainMatrix](#) (page 2671) may have an internal representation that is sparse or dense. We guarantee a dense representation here, so that tests for equivalence of submodules always come out as expected.

**Examples**

```
>>> from sympy.polys import Poly, cyclotomic_poly, ZZ
>>> from sympy.abc import x
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy.polys.numberfields.modules import PowerBasis
>>> T = Poly(cyclotomic_poly(5, x))
>>> A = PowerBasis(T)
>>> B = A.submodule_from_matrix(3*DomainMatrix.eye(4, ZZ), denom=6)
>>> C = A.submodule_from_matrix(DomainMatrix.eye(4, ZZ), denom=2)
```

(continues on next page)

(continued from previous page)

```
>>> print(B.QQ_matrix == C.QQ_matrix)
True
```

**add**(*other*, *hnf*=True, *hnf\_modulus*=None)

Add this [Submodule](#) (page 2726) to another.

**Parameters**

**other** : [Submodule](#) (page 2726)

**hnf** : boolean, optional (default=True)

If True, reduce the matrix of the combined module to its Hermite Normal Form.

**hnf\_modulus** : [ZZ](#) (page 2525), None, optional

If a positive integer is provided, use this as modulus in the HNF reduction. See [hermite\\_normal\\_form\(\)](#) (page 2700).

**Returns**

[Submodule](#) (page 2726)

**Explanation**

This represents the module generated by the union of the two modules' sets of generators.

**basis\_element\_pullbacks()**

Return list of this submodule's basis elements as elements of the submodule's parent module.

**discard\_before(r)**

Produce a new module by discarding all generators before a given index *r*.

**mul**(*other*, *hnf*=True, *hnf\_modulus*=None)

Multiply this [Submodule](#) (page 2726) by a rational number, a [ModuleElement](#) (page 2729), or another [Submodule](#) (page 2726).

**Parameters**

**other** : int, [ZZ](#) (page 2525), [QQ](#) (page 2529), [ModuleElement](#) (page 2729), [Submodule](#) (page 2726)

**hnf** : boolean, optional (default=True)

If True, reduce the matrix of the product module to its Hermite Normal Form.

**hnf\_modulus** : [ZZ](#) (page 2525), None, optional

If a positive integer is provided, use this as modulus in the HNF reduction. See [hermite\\_normal\\_form\(\)](#) (page 2700).

**Returns**

[Submodule](#) (page 2726)

### Explanation

To multiply by a rational number or [ModuleElement](#) (page 2729) means to form the submodule whose generators are the products of this quantity with all the generators of the present submodule.

To multiply by another [Submodule](#) (page 2726) means to form the submodule whose generators are all the products of one generator from the one submodule, and one generator from the other.

#### `reduce_element(elt)`

If this submodule  $B$  has defining matrix  $W$  in square, maximal-rank Hermite normal form, then, given an element  $x$  of the parent module  $A$ , we produce an element  $y \in A$  such that  $x - y \in B$ , and the  $i$ th coordinate of  $y$  satisfies  $0 \leq y_i < w_{i,i}$ . This representative  $y$  is unique, in the sense that every element of the coset  $x + B$  reduces to it under this procedure.

##### Parameters

**elt** : [ModuleElement](#) (page 2729)

An element of this submodule's parent module.

##### Returns

**elt** : [ModuleElement](#) (page 2729)

An element of this submodule's parent module.

##### Raises

###### **NotImplementedError**

If the given [ModuleElement](#) (page 2729) does not belong to this submodule's parent module.

###### **StructureError**

If this submodule's defining matrix is not in square, maximal-rank Hermite normal form.

### Explanation

In the special case where  $A$  is a power basis for a number field  $K$ , and  $B$  is a submodule representing an ideal  $I$ , this operation represents one of a few important ways of reducing an element of  $K$  modulo  $I$  to obtain a “small” representative. See [Cohen00] Section 1.4.3.

### Examples

```
>>> from sympy import QQ, Poly, symbols
>>> t = symbols('t')
>>> k = QQ.alg_field_from_poly(Poly(t**3 + t**2 - 2*t + 8))
>>> Zk = k.maximal_order()
>>> A = Zk.parent
>>> B = (A(2) - 3*A(0))*Zk
>>> B.reduce_element(A(2))
[3, 0, 0]
```



## References

[Cohen00]

### **reduced()**

Produce a reduced version of this submodule.

#### **Returns**

*Submodule* (page 2726)

## Explanation

In the reduced version, it is guaranteed that 1 is the only positive integer dividing both the submodule's denominator, and every entry in the submodule's matrix.

### **represent(elt)**

Represent a module element as an integer-linear combination over the generators of this module.

#### **See also:**

*Module.represent* (page 2722), *PowerBasis.represent* (page 2725)

**class** sympy.polys.numberfields.modules.**ModuleElement**(*module*, *col*, *denom*=1)

Represents an element of a *Module* (page 2719).

NOTE: Should not be constructed directly. Use the `__call__()` (page 2719) method or the `make_mod_elt()` (page 2733) factory function instead.

**\_\_init\_\_**(*module*, *col*, *denom*=1)

#### **Parameters**

**module** : *Module* (page 2719)

The module to which this element belongs.

**col** : *DomainMatrix* (page 2671) over *ZZ* (page 2525)

Column vector giving the numerators of the coefficients of this element.

**denom** : int, optional (default=1)

Denominator for the coefficients of this element.

**\_\_add\_\_**(*other*)

A *ModuleElement* (page 2729) can be added to a rational number, or to another *ModuleElement* (page 2729).

### Explanation

When the other summand is a rational number, it will be converted into a [ModuleElement](#) (page 2729) (belonging to the first ancestor of this module that starts with unity).

In all cases, the sum belongs to the nearest common ancestor (NCA) of the modules of the two summands. If the NCA does not exist, we return `NotImplemented`.

`__mul__(other)`

A [ModuleElement](#) (page 2729) can be multiplied by a rational number, or by another [ModuleElement](#) (page 2729).

### Explanation

When the multiplier is a rational number, the product is computed by operating directly on the coefficients of this [ModuleElement](#) (page 2729).

When the multiplier is another [ModuleElement](#) (page 2729), the product will belong to the nearest common ancestor (NCA) of the modules of the two operands, and that NCA must have a multiplication table. If the NCA does not exist, we return `NotImplemented`. If the NCA does not have a mult. table, `ClosureFailure` will be raised.

`__mod__(m)`

Reduce this [ModuleElement](#) (page 2729) mod a [Submodule](#) (page 2726).

#### Parameters

**m** : int, [ZZ](#) (page 2525), [QQ](#) (page 2529), [Submodule](#) (page 2726)

If a [Submodule](#) (page 2726), reduce `self` relative to this. If an integer or rational, reduce relative to the [Submodule](#) (page 2726) that is our own module times this constant.

#### See also:

[Submodule.reduce\\_element](#) (page 2728)

**property** `QQ_col`

[DomainMatrix](#) (page 2671) over [QQ](#) (page 2529), equal to `self.col / self.denom`, and guaranteed to be dense.

#### See also:

[Submodule.QQ\\_matrix](#) (page 2726)

**column**(*domain=None*)

Get a copy of this element's column, optionally converting to a domain.

**equiv**(*other*)

A [ModuleElement](#) (page 2729) may test as equivalent to a rational number or another [ModuleElement](#) (page 2729), if they represent the same algebraic number.

#### Parameters

**other** : int, [ZZ](#) (page 2525), [QQ](#) (page 2529), [ModuleElement](#) (page 2729)

#### Returns

bool

**Raises****UnificationFailed**

If `self` and `other` do not share a common [PowerBasis](#) (page 2725) ancestor.

**Explanation**

This method is intended to check equivalence only in those cases in which it is easy to test; namely, when `other` is either a [ModuleElement](#) (page 2729) that can be unified with this one (i.e. one which shares a common [PowerBasis](#) (page 2725) ancestor), or else a rational number (which is easy because every [PowerBasis](#) (page 2725) represents every rational number).

**classmethod** `from_int_list(module, coeffs, denom=1)`

Make a [ModuleElement](#) (page 2729) from a list of ints (instead of a column vector).

**is\_compat**(`other`)

Test whether `other` is another [ModuleElement](#) (page 2729) with same module.

**property** `n`

The length of this element's column.

**over\_power\_basis**()

Transform into a [PowerBasisElement](#) (page 2732) over our [PowerBasis](#) (page 2725) ancestor.

**reduced**()

Produce a reduced version of this [ModuleElement](#), i.e. one in which the gcd of the denominator together with all numerator coefficients is 1.

**reduced\_mod\_p**(`p`)

Produce a version of this [ModuleElement](#) (page 2729) in which all numerator coefficients have been reduced mod `p`.

**to\_ancestor**(`anc`)

Transform into a [ModuleElement](#) (page 2729) belonging to a given ancestor of this element's module.

**Parameters**

**anc** : [Module](#) (page 2719)

**to\_parent**()

Transform into a [ModuleElement](#) (page 2729) belonging to the parent of this element's module.

**unify**(`other`)

Try to make a compatible pair of [ModuleElement](#) (page 2729), one equivalent to this one, and one equivalent to the other.

**Returns**

Pair (`e1`, `e2`)

Each `ei` is a [ModuleElement](#) (page 2729), they belong to the same [Module](#) (page 2719), `e1` is equivalent to `self`, and `e2` is equivalent to `other`.

**Raises****UnificationFailed**

If self and other have no common ancestor module.

**Explanation**

We search for the nearest common ancestor module for the pair of elements, and represent each one there.

**class** sympy.polys.numberfields.modules.**PowerBasisElement**(*module, col, denom=1*)  
Subclass for *ModuleElement* (page 2729) instances whose module is a *PowerBasis* (page 2725).

**property T**

Access the defining polynomial of the *PowerBasis* (page 2725).

**as\_expr**(*x=None*)

Create a Basic expression from self.

**property generator**

Return a *Symbol* (page 976) to be used when expressing this element as a polynomial.

If we have an associated *AlgebraicField* (page 2539) whose primitive element has an alias symbol, we use that. Otherwise we use the variable of the minimal polynomial defining the power basis to which we belong.

**property is\_rational**

Say whether this element represents a rational number.

**norm**(*T=None*)

Compute the norm of this number.

**numerator**(*x=None*)

Obtain the numerator as a polynomial over *ZZ* (page 2525).

**poly**(*x=None*)

Obtain the number as a polynomial over *QQ* (page 2529).

**to\_ANP**()

Convert to an equivalent *ANP* (page 2568).

**to\_alg\_num**()

Try to convert to an equivalent *AlgebraicNumber* (page 988).

**Returns**

*AlgebraicNumber* (page 988)

**Raises****StructureError**

If the *PowerBasis* (page 2725) to which this element belongs does not have an associated *AlgebraicField* (page 2539).

## Explanation

In general, the conversion from an *AlgebraicNumber* (page 988) to a *PowerBasisElement* (page 2732) throws away information, because an *AlgebraicNumber* (page 988) specifies a complex embedding, while a *PowerBasisElement* (page 2732) does not. However, in some cases it is possible to convert a *PowerBasisElement* (page 2732) back into an *AlgebraicNumber* (page 988), namely when the associated *PowerBasis* (page 2725) has a reference to an *AlgebraicField* (page 2539).

`sympy.polys.numberfields.modules.make_mod_elt(module, col, denom=1)`

Factory function which builds a *ModuleElement* (page 2729), but ensures that it is a *PowerBasisElement* (page 2732) if the module is a *PowerBasis* (page 2725).

**class** `sympy.polys.numberfields.modules.ModuleHomomorphism(domain, codomain, mapping)`

A homomorphism from one module to another.

**\_\_init\_\_**(*domain*, *codomain*, *mapping*)

### Parameters

**domain** : *Module* (page 2719)

The domain of the mapping.

**codomain** : *Module* (page 2719)

The codomain of the mapping.

**mapping** : callable

An arbitrary callable is accepted, but should be chosen so as to represent an actual module homomorphism. In particular, should accept elements of *domain* and return elements of *codomain*.

## Examples

```
>>> from sympy import Poly, cyclotomic_poly
>>> from sympy.polys.numberfields.modules import PowerBasis, \
↳ ModuleHomomorphism
>>> T = Poly(cyclotomic_poly(5))
>>> A = PowerBasis(T)
>>> B = A.submodule_from_gens([2*A(j) for j in range(4)])
>>> phi = ModuleHomomorphism(A, B, lambda x: 6*x)
>>> print(phi.matrix())
DomainMatrix([[3, 0, 0, 0], [0, 3, 0, 0], [0, 0, 3, 0], [0, 0, 0, 3]],
↳ (4, 4), ZZ)
```

**kernel**(*modulus=None*)

Compute a Submodule representing the kernel of this homomorphism.

### Parameters

**modulus** : int, optional

A positive prime number  $p$  if the kernel should be computed mod  $p$ .

### Returns

*Submodule* (page 2726)

This submodule's generators span the kernel of this homomorphism over [ZZ](#) (page 2525), or else over [GF\(p\)](#) (page 2522) if a modulus was given.

**matrix**(*modulus=None*)

Compute the matrix of this homomorphism.

**Parameters**

**modulus** : int, optional

A positive prime number  $p$  if the matrix should be reduced mod  $p$ .

**Returns**

[DomainMatrix](#) (page 2671)

The matrix is over [ZZ](#) (page 2525), or else over [GF\(p\)](#) (page 2522) if a modulus was given.

**class** sympy.polys.numberfields.modules.**ModuleEndomorphism**(*domain, mapping*)

A homomorphism from one module to itself.

**\_\_init\_\_**(*domain, mapping*)

**Parameters**

**domain** : [Module](#) (page 2719)

The common domain and codomain of the mapping.

**mapping** : callable

An arbitrary callable is accepted, but should be chosen so as to represent an actual module endomorphism. In particular, should accept and return elements of *domain*.

**class** sympy.polys.numberfields.modules.**InnerEndomorphism**(*domain, multiplier*)

An inner endomorphism on a module, i.e. the endomorphism corresponding to multiplication by a fixed element.

**\_\_init\_\_**(*domain, multiplier*)

**Parameters**

**domain** : [Module](#) (page 2719)

The domain and codomain of the endomorphism.

**multiplier** : [ModuleElement](#) (page 2729)

The element  $a$  defining the mapping as  $x \mapsto ax$ .

**class** sympy.polys.numberfields.modules.**EndomorphismRing**(*domain*)

The ring of endomorphisms on a module.

**\_\_init\_\_**(*domain*)

**Parameters**

**domain** : [Module](#) (page 2719)

The domain and codomain of the endomorphisms.

**inner\_endomorphism**(*multiplier*)

Form an inner endomorphism belonging to this endomorphism ring.

**Parameters**

**multiplier** : [ModuleElement](#) (page 2729)

Element  $a$  defining the inner endomorphism  $x \mapsto ax$ .

### Returns

[\*InnerEndomorphism\*](#) (page 2734)

### `represent(element)`

Represent an element of this endomorphism ring, as a single column vector.

### Parameters

**element** : [\*ModuleEndomorphism\*](#) (page 2734) belonging to this ring.

### Returns

[\*DomainMatrix\*](#) (page 2671)

Column vector equalling the vertical stacking of all the columns of the matrix that represents the given *element* as a mapping.

## Explanation

Let  $M$  be a module, and  $E$  its ring of endomorphisms. Let  $N$  be another module, and consider a homomorphism  $\varphi : N \rightarrow E$ . In the event that  $\varphi$  is to be represented by a matrix  $A$ , each column of  $A$  must represent an element of  $E$ . This is possible when the elements of  $E$  are themselves representable as matrices, by stacking the columns of such a matrix into a single column.

This method supports calculating such matrices  $A$ , by representing an element of this endomorphism ring first as a matrix, and then stacking that matrix's columns into a single column.

## Examples

Note that in these examples we print matrix transposes, to make their columns easier to inspect.

```
>>> from sympy import Poly, cyclotomic_poly
>>> from sympy.polys.numberfields.modules import PowerBasis
>>> from sympy.polys.numberfields.modules import ModuleHomomorphism
>>> T = Poly(cyclotomic_poly(5))
>>> M = PowerBasis(T)
>>> E = M.endomorphism_ring()
```

Let  $\zeta$  be a primitive 5th root of unity, a generator of our field, and consider the inner endomorphism  $\tau$  on the ring of integers, induced by  $\zeta$ :

```
>>> zeta = M(1)
>>> tau = E.inner_endomorphism(zeta)
>>> tau.matrix().transpose()
DomainMatrix(
 [[0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1], [-1, -1, -1, -1]],
 (4, 4), ZZ)
```

The matrix representation of  $\tau$  is as expected. The first column shows that multiplying by  $\zeta$  carries 1 to  $\zeta$ , the second column that it carries  $\zeta$  to  $\zeta^2$ , and so forth.

The `represent` method of the endomorphism ring  $E$  stacks these into a single column:

```
>>> E.represent(tau).transpose()
DomainMatrix(
 [[0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, -1, -1, -1, -1]],
 (1, 16), ZZ)
```

This is useful when we want to consider a homomorphism  $\varphi$  having  $E$  as codomain:

```
>>> phi = ModuleHomomorphism(M, E, lambda x: E.inner_endomorphism(x))
```

and we want to compute the matrix of such a homomorphism:

```
>>> phi.matrix().transpose()
DomainMatrix(
 [[1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1],
 [0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, -1, -1, -1, -1],
 [0, 0, 1, 0, 0, 0, 0, 0, 1, -1, -1, -1, -1, 1, 0, 0, 0],
 [0, 0, 0, 1, -1, -1, -1, -1, 1, 0, 0, 0, 0, 1, 0, 0, 0]],
 (4, 16), ZZ)
```

Note that the stacked matrix of  $\tau$  occurs as the second column in this example. This is because  $\zeta$  is the second basis element of  $M$ , and  $\varphi(\zeta) = \tau$ .

`sympy.polys.numberfields.modules.find_min_poly(alpha, domain, x=None, powers=None)`

Find a polynomial of least degree (not necessarily irreducible) satisfied by an element of a finitely-generated ring with unity.

#### Parameters

**alpha** : *ModuleElement* (page 2729)

The element whose min poly is to be found, and whose module has multiplication and starts with unity.

**domain** : *Domain* (page 2504)

The desired domain of the polynomial.

**x** : *Symbol* (page 976), optional

The desired variable for the polynomial.

**powers** : list, optional

If desired, pass an empty list. The powers of *alpha* (as *ModuleElement* (page 2729) instances) from the zeroth up to the degree of the min poly will be recorded here, as we compute them.

#### Returns

*Poly* (page 2378), None

The minimal polynomial for *alpha*, or None if no polynomial could be found over the desired domain.

#### Raises

**MissingUnityError**

If the module to which *alpha* belongs does not start with unity.

**ClosureFailure**



If the module to which alpha belongs is not closed under multiplication.

## Examples

For the  $n$ th cyclotomic field,  $n$  an odd prime, consider the quadratic equation whose roots are the two periods of length  $(n-1)/2$ . Article 356 of Gauss tells us that we should get  $x^2 + x - (n-1)/4$  or  $x^2 + x + (n+1)/4$  according to whether  $n$  is 1 or 3 mod 4, respectively.

```
>>> from sympy import Poly, cyclotomic_poly, primitive_root, QQ
>>> from sympy.abc import x
>>> from sympy.polys.numberfields.modules import PowerBasis, find_min_
 poly
>>> n = 13
>>> g = primitive_root(n)
>>> C = PowerBasis(Poly(cyclotomic_poly(n, x)))
>>> ee = [g**(2*k+1) % n for k in range((n-1)//2)]
>>> eta = sum(C(e) for e in ee)
>>> print(find_min_poly(eta, QQ, x=x).as_expr())
x**2 + x - 3
>>> n = 19
>>> g = primitive_root(n)
>>> C = PowerBasis(Poly(cyclotomic_poly(n, x)))
>>> ee = [g**(2*k+2) % n for k in range((n-1)//2)]
>>> eta = sum(C(e) for e in ee)
>>> print(find_min_poly(eta, QQ, x=x).as_expr())
x**2 + x + 5
```

## Utilities

`sympy.polys.numberfields.utilities.is_rat(c)`

Test whether an argument is of an acceptable type to be used as a rational number.

### Explanation

Returns True on any argument of type `int`, `ZZ` (page 2525), or `QQ` (page 2529).

**See also:**

`is_int` (page 2737)

`sympy.polys.numberfields.utilities.is_int(c)`

Test whether an argument is of an acceptable type to be used as an integer.

### Explanation

Returns True on any argument of type int or [ZZ](#) (page 2525).

#### See also:

[is\\_rat](#) (page 2737)

`sympy.polys.numberfields.utilities.get_num_denom(c)`

Given any argument on which [is\\_rat\(\)](#) (page 2737) is True, return the numerator and denominator of this number.

#### See also:

[is\\_rat](#) (page 2737)

`sympy.polys.numberfields.utilities.extract_fundamental_discriminant(a)`

Extract a fundamental discriminant from an integer  $a$ .

#### Parameters

**a:** int, must be 0 or 1 mod 4

#### Returns

Pair (D, F) of dictionaries.

#### Raises

##### ValueError

If  $a$  is not 0 or 1 mod 4.

### Explanation

Given any rational integer  $a$  that is 0 or 1 mod 4, write  $a = df^2$ , where  $d$  is either 1 or a fundamental discriminant, and return a pair of dictionaries (D, F) giving the prime factorizations of  $d$  and  $f$  respectively, in the same format returned by [factorint\(\)](#) (page 1493).

A fundamental discriminant  $d$  is different from unity, and is either 1 mod 4 and squarefree, or is 0 mod 4 and such that  $d/4$  is squarefree and 2 or 3 mod 4. This is the same as being the discriminant of some quadratic field.

### Examples

```
>>> from sympy.polys.numberfields.utilities import extract_fundamental_
↪discriminant
>>> print(extract_fundamental_discriminant(-432))
({3: 1, -1: 1}, {2: 2, 3: 1})
```

For comparison:

```
>>> from sympy import factorint
>>> print(factorint(-432))
{2: 4, 3: 3, -1: 1}
```

## References

[R716]

**class** sympy.polys.numberfields.utilities.**AlgIntPowers**(*T*, *modulus=None*)

Compute the powers of an algebraic integer.

## Explanation

Given an algebraic integer  $\theta$  by its monic irreducible polynomial  $T$  over [ZZ](#) (page 2525), this class computes representations of arbitrarily high powers of  $\theta$ , as [ZZ](#) (page 2525)-linear combinations over  $\{1, \theta, \dots, \theta^{n-1}\}$ , where  $n = \deg(T)$ .

The representations are computed using the linear recurrence relations for powers of  $\theta$ , derived from the polynomial  $T$ . See [1], Sec. 4.2.2.

Optionally, the representations may be reduced with respect to a modulus.

## Examples

```
>>> from sympy import Poly, cyclotomic_poly
>>> from sympy.polys.numberfields.utilities import AlgIntPowers
>>> T = Poly(cyclotomic_poly(5))
>>> zeta_pow = AlgIntPowers(T)
>>> print(zeta_pow[0])
[1, 0, 0, 0]
>>> print(zeta_pow[1])
[0, 1, 0, 0]
>>> print(zeta_pow[4])
[-1, -1, -1, -1]
>>> print(zeta_pow[24])
[-1, -1, -1, -1]
```

## References

[R717]

**\_\_init\_\_**(*T*, *modulus=None*)

### Parameters

**T** : [Poly](#) (page 2378)

The monic irreducible polynomial over [ZZ](#) (page 2525) defining the algebraic integer.

**modulus** : int, None, optional

If not None, all representations will be reduced w.r.t. this.

sympy.polys.numberfields.utilities.**coeff\_search**(*m*, *R*)

Generate coefficients for searching through polynomials.

### Parameters

**m** : int

Length of coeff list.

**R** : int

Initial max abs val for coeffs (will increase as search proceeds).

**Returns**

generator

Infinite generator of lists of coefficients.

**Explanation**

Lead coeff is always non-negative. Explore all combinations with coeffs bounded in absolute value before increasing the bound. Skip the all-zero list, and skip any repeats. See examples.

**Examples**

```
>>> from sympy.polys.numberfields.utilities import coeff_search
>>> cs = coeff_search(2, 1)
>>> C = [next(cs) for i in range(13)]
>>> print(C)
[[1, 1], [1, 0], [1, -1], [0, 1], [2, 2], [2, 1], [2, 0], [2, -1], [2, -
-2],
 [1, 2], [1, -2], [0, 2], [3, 3]]
```

`sympy.polys.numberfields.utilities.supplement_a_subspace(M)`

Extend a basis for a subspace to a basis for the whole space.

**Parameters**

**M** : *DomainMatrix* (page 2671)

The columns give the basis for the subspace.

**Returns**

*DomainMatrix* (page 2671)

This matrix is invertible and its first  $r$  columns equal  $M$ .

**Raises**

**DMRankError**

If  $M$  was not of maximal rank.

**Explanation**

Given an  $n \times r$  matrix  $M$  of rank  $r$  (so  $r \leq n$ ), this function computes an invertible  $n \times n$  matrix  $B$  such that the first  $r$  columns of  $B$  equal  $M$ .

This operation can be interpreted as a way of extending a basis for a subspace, to give a basis for the whole space.

To be precise, suppose you have an  $n$ -dimensional vector space  $V$ , with basis  $\{v_1, v_2, \dots, v_n\}$ , and an  $r$ -dimensional subspace  $W$  of  $V$ , spanned by a basis  $\{w_1, w_2, \dots, w_r\}$ , where the  $w_j$  are given as linear combinations of the  $v_i$ . If the columns of  $M$  represent

the  $w_j$  as such linear combinations, then the columns of the matrix  $B$  computed by this function give a new basis  $\{u_1, u_2, \dots, u_n\}$  for  $V$ , again relative to the  $\{v_i\}$  basis, and such that  $u_j = w_j$  for  $1 \leq j \leq r$ .

## Examples

Note: The function works in terms of columns, so in these examples we print matrix transposes in order to make the columns easier to inspect.

```
>>> from sympy.polys.matrices import DM
>>> from sympy import QQ, FF
>>> from sympy.polys.numberfields.utilities import supplement_a_subspace
>>> M = DM([[1, 7, 0], [2, 3, 4]], QQ).transpose()
>>> print(supplement_a_subspace(M).to_Matrix().transpose())
Matrix([[1, 7, 0], [2, 3, 4], [1, 0, 0]])
```

```
>>> M2 = M.convert_to(FF(7))
>>> print(M2.to_Matrix().transpose())
Matrix([[1, 0, 0], [2, 3, -3]])
>>> print(supplement_a_subspace(M2).to_Matrix().transpose())
Matrix([[1, 0, 0], [2, 3, -3], [0, 1, 0]])
```

## References

[R718]

`sympy.polys.numberfields.utilities.isolate(alg, eps=None, fast=False)`

Find a rational isolating interval for a real algebraic number.

### Parameters

**alg** : str, int, *Expr* (page 947)

The algebraic number to be isolated. Must be a real number, to use this particular function. However, see also *Poly.intervals()* (page 2398), which isolates complex roots when you pass *all*=True.

**eps** : positive element of *QQ* (page 2529), None, optional (default=None)

Precision to be passed to *Poly.refine\_root()* (page 2412)

**fast** : boolean, optional (default=False)

Say whether fast refinement procedure should be used. (Will be passed to *Poly.refine\_root()* (page 2412).)

### Returns

Pair of rational numbers defining an isolating interval for the given algebraic number.

## Examples

```
>>> from sympy import isolate, sqrt, Rational
>>> print(isolate(sqrt(2)))
(1, 2)
>>> print(isolate(sqrt(2), eps=Rational(1, 100)))
(24/17, 17/12)
```

### See also:

*Poly.intervals* (page 2398)

## Category Theory

### Introduction

The category theory module for SymPy will allow manipulating diagrams within a single category, including drawing them in TikZ and deciding whether they are commutative or not.

The general reference work this module tries to follow is

The latest version of this book should be available for free download from

[katmat.math.uni-bremen.de/acc/acc.pdf](http://katmat.math.uni-bremen.de/acc/acc.pdf)

The module is still in its pre-embryonic stage.

### Base Class Reference

This section lists the classes which implement some of the basic notions in category theory: objects, morphisms, categories, and diagrams.

**class** sympy.categories.**Object**(*name*, *\*\*assumptions*)

The base class for any kind of object in an abstract category.

### Explanation

While technically any instance of *Basic* (page 927) will do, this class is the recommended way to create abstract objects in abstract categories.

**class** sympy.categories.**Morphism**(*domain*, *codomain*)

The base class for any morphism in an abstract category.

## Explanation

In abstract categories, a morphism is an arrow between two category objects. The object where the arrow starts is called the domain, while the object where the arrow ends is called the codomain.

Two morphisms between the same pair of objects are considered to be the same morphisms. To distinguish between morphisms between the same objects use *NamedMorphism* (page 2744).

It is prohibited to instantiate this class. Use one of the derived classes instead.

### See also:

*IdentityMorphism* (page 2746), *NamedMorphism* (page 2744), *CompositeMorphism* (page 2744)

### property codomain

Returns the codomain of the morphism.

## Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> f = NamedMorphism(A, B, "f")
>>> f.codomain
Object("B")
```

### compose(*other*)

Composes self with the supplied morphism.

The order of elements in the composition is the usual order, i.e., to construct  $g \circ f$  use `g.compose(f)`.

## Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> g * f
CompositeMorphism((NamedMorphism(Object("A"), Object("B"), "f"),
NamedMorphism(Object("B"), Object("C"), "g")))
>>> (g * f).domain
Object("A")
>>> (g * f).codomain
Object("C")
```

### property domain

Returns the domain of the morphism.

### Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> f = NamedMorphism(A, B, "f")
>>> f.domain
Object("A")
```

**class** sympy.categories.NamedMorphism(*domain, codomain, name*)

Represents a morphism which has a name.

### Explanation

Names are used to distinguish between morphisms which have the same domain and codomain: two named morphisms are equal if they have the same domains, codomains, and names.

### Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> f = NamedMorphism(A, B, "f")
>>> f
NamedMorphism(Object("A"), Object("B"), "f")
>>> f.name
'f'
```

**See also:**

[\*Morphism\*](#) (page 2742)

**property name**

Returns the name of the morphism.

### Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> f = NamedMorphism(A, B, "f")
>>> f.name
'f'
```

**class** sympy.categories.CompositeMorphism(*\*components*)

Represents a morphism which is a composition of other morphisms.



## Explanation

Two composite morphisms are equal if the morphisms they were obtained from (components) are the same and were listed in the same order.

The arguments to the constructor for this class should be listed in diagram order: to obtain the composition  $g \circ f$  from the instances of *Morphism* (page 2742) *g* and *f* use `CompositeMorphism(f, g)`.

## Examples

```
>>> from sympy.categories import Object, NamedMorphism, CompositeMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> g * f
CompositeMorphism((NamedMorphism(Object("A"), Object("B"), "f"),
NamedMorphism(Object("B"), Object("C"), "g")))
>>> CompositeMorphism(f, g) == g * f
True
```

### property codomain

Returns the codomain of this composite morphism.

The codomain of the composite morphism is the codomain of its last component.

## Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> (g * f).codomain
Object("C")
```

### property components

Returns the components of this composite morphism.

### Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> (g * f).components
(NamedMorphism(Object("A"), Object("B"), "f"),
 NamedMorphism(Object("B"), Object("C"), "g"))
```

### property domain

Returns the domain of this composite morphism.

The domain of the composite morphism is the domain of its first component.

### Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> (g * f).domain
Object("A")
```

### flatten(*new\_name*)

Forgets the composite structure of this morphism.

### Explanation

If *new\_name* is not empty, returns a [\*NamedMorphism\*](#) (page 2744) with the supplied name, otherwise returns a [\*Morphism\*](#) (page 2742). In both cases the domain of the new morphism is the domain of this composite morphism and the codomain of the new morphism is the codomain of this composite morphism.

### Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> (g * f).flatten("h")
NamedMorphism(Object("A"), Object("C"), "h")
```

**class** sympy.categories.**IdentityMorphism**(*domain*)

Represents an identity morphism.

### Explanation

An identity morphism is a morphism with equal domain and codomain, which acts as an identity with respect to composition.

### Examples

```
>>> from sympy.categories import Object, NamedMorphism, IdentityMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> f = NamedMorphism(A, B, "f")
>>> id_A = IdentityMorphism(A)
>>> id_B = IdentityMorphism(B)
>>> f * id_A == f
True
>>> id_B * f == f
True
```

**See also:**

[Morphism](#) (page 2742)

**class** sympy.categories.**Category**(*name*, *objects*=*EmptySet*,  
                                  *commutative\_diagrams*=*EmptySet*)

An (abstract) category.

### Explanation

A category [JoyOfCats] is a quadruple  $K = (O, \text{hom}, id, \circ)$  consisting of

- a (set-theoretical) class  $O$ , whose members are called  $K$ -objects,
- for each pair  $(A, B)$  of  $K$ -objects, a set  $\text{hom}(A, B)$  whose members are called  $K$ -morphisms from  $A$  to  $B$ ,
- for a each  $K$ -object  $A$ , a morphism  $id : A \rightarrow A$ , called the  $K$ -identity of  $A$ ,
- a composition law  $\circ$  associating with every  $K$ -morphisms  $f : A \rightarrow B$  and  $g : B \rightarrow C$  a  $K$ -morphism  $g \circ f : A \rightarrow C$ , called the composite of  $f$  and  $g$ .

Composition is associative,  $K$ -identities are identities with respect to composition, and the sets  $\text{hom}(A, B)$  are pairwise disjoint.

This class knows nothing about its objects and morphisms. Concrete cases of (abstract) categories should be implemented as classes derived from this one.

Certain instances of [Diagram](#) (page 2749) can be asserted to be commutative in a [Category](#) (page 2747) by supplying the argument `commutative_diagrams` in the constructor.

## Examples

```
>>> from sympy.categories import Object, NamedMorphism, Diagram, Category
>>> from sympy import FiniteSet
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g])
>>> K = Category("K", commutative_diagrams=[d])
>>> K.commutative_diagrams == FiniteSet(d)
True
```

### See also:

[Diagram](#) (page 2749)

### property `commutative_diagrams`

Returns the [FiniteSet](#) (page 1197) of diagrams which are known to be commutative in this category.

## Examples

```
>>> from sympy.categories import Object, NamedMorphism, Diagram, Category
>>> from sympy import FiniteSet
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g])
>>> K = Category("K", commutative_diagrams=[d])
>>> K.commutative_diagrams == FiniteSet(d)
True
```

### property `name`

Returns the name of this category.

## Examples

```
>>> from sympy.categories import Category
>>> K = Category("K")
>>> K.name
'K'
```

### property `objects`

Returns the class of objects of this category.

## Examples

```
>>> from sympy.categories import Object, Category
>>> from sympy import FiniteSet
>>> A = Object("A")
>>> B = Object("B")
>>> K = Category("K", FiniteSet(A, B))
>>> K.objects
Class({Object("A"), Object("B")})
```

**class** sympy.categories.**Diagram**(\*args)

Represents a diagram in a certain category.

## Explanation

Informally, a diagram is a collection of objects of a category and certain morphisms between them. A diagram is still a monoid with respect to morphism composition; i.e., identity morphisms, as well as all composites of morphisms included in the diagram belong to the diagram. For a more formal approach to this notion see [Pare1970].

The components of composite morphisms are also added to the diagram. No properties are assigned to such morphisms by default.

A commutative diagram is often accompanied by a statement of the following kind: “if such morphisms with such properties exist, then such morphisms which such properties exist and the diagram is commutative”. To represent this, an instance of *Diagram* (page 2749) includes a collection of morphisms which are the premises and another collection of conclusions. *premises* and *conclusions* associate morphisms belonging to the corresponding categories with the *FiniteSet* (page 1197)’s of their properties.

The set of properties of a composite morphism is the intersection of the sets of properties of its components. The domain and codomain of a conclusion morphism should be among the domains and codomains of the morphisms listed as the premises of a diagram.

No checks are carried out of whether the supplied object and morphisms do belong to one and the same category.

## Examples

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy import pprint, default_sort_key
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g])
>>> premises_keys = sorted(d.premises.keys(), key=default_sort_key)
>>> pprint(premises_keys, use_unicode=False)
[g*f:A-->C, id:A-->A, id:B-->B, id:C-->C, f:A-->B, g:B-->C]
>>> pprint(d.premises, use_unicode=False)
{g*f:A-->C: EmptySet, id:A-->A: EmptySet, id:B-->B: EmptySet, id:C-->C: }
```

(continues on next page)

(continued from previous page)

```

↳ EmptySet
et, f:A-->B: EmptySet, g:B-->C: EmptySet}
>>> d = Diagram([f, g], {g * f: "unique"})
>>> pprint(d.conclusions, use_unicode=False)
{g*f:A-->C: {unique}}

```

## References

[Pare1970] B. Pareigis: Categories and functors. Academic Press, 1970.

### property conclusions

Returns the conclusions of this diagram.

### Examples

```

>>> from sympy.categories import Object, NamedMorphism
>>> from sympy.categories import IdentityMorphism, Diagram
>>> from sympy import FiniteSet
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g])
>>> IdentityMorphism(A) in d.premises.keys()
True
>>> g * f in d.premises.keys()
True
>>> d = Diagram([f, g], {g * f: "unique"})
>>> d.conclusions[g * f] == FiniteSet("unique")
True

```

### hom(A, B)

Returns a 2-tuple of sets of morphisms between objects A and B: one set of morphisms listed as premises, and the other set of morphisms listed as conclusions.

### Examples

```

>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy import pretty
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g], {g * f: "unique"})
>>> print(pretty(d.hom(A, C), use_unicode=False))
({g*f:A-->C}, {g*f:A-->C})

```

**See also:**

[Object](#) (page 2742), [Morphism](#) (page 2742)

**is\_subdiagram**(*diagram*)

Checks whether *diagram* is a subdiagram of *self*. Diagram  $D'$  is a subdiagram of  $D$  if all premises (conclusions) of  $D'$  are contained in the premises (conclusions) of  $D$ . The morphisms contained both in  $D'$  and  $D$  should have the same properties for  $D'$  to be a subdiagram of  $D$ .

**Examples**

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g], {g * f: "unique"})
>>> d1 = Diagram([f])
>>> d.is_subdiagram(d1)
True
>>> d1.is_subdiagram(d)
False
```

**property objects**

Returns the [FiniteSet](#) (page 1197) of objects that appear in this diagram.

**Examples**

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g])
>>> d.objects
{Object("A"), Object("B"), Object("C")}
```

**property premises**

Returns the premises of this diagram.

### Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> from sympy.categories import IdentityMorphism, Diagram
>>> from sympy import pretty
>>> A = Object("A")
>>> B = Object("B")
>>> f = NamedMorphism(A, B, "f")
>>> id_A = IdentityMorphism(A)
>>> id_B = IdentityMorphism(B)
>>> d = Diagram([f])
>>> print(pretty(d.premises, use_unicode=False))
{id:A-->A: EmptySet, id:B-->B: EmptySet, f:A-->B: EmptySet}
```

#### `subdiagram_from_objects(objects)`

If `objects` is a subset of the objects of `self`, returns a diagram which has as premises all those premises of `self` which have a domains and codomains in `objects`, likewise for conclusions. Properties are preserved.

### Examples

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy import FiniteSet
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g], {f: "unique", g*f: "veryunique"})
>>> d1 = d.subdiagram_from_objects(FiniteSet(A, B))
>>> d1 == Diagram([f], {f: "unique"})
True
```

### Diagram Drawing

This section lists the classes which allow automatic drawing of diagrams.

**class** `sympy.categories.diagram_drawing.DiagramGrid`(*diagram*, *groups=None*,  
\*\**hints*)

Constructs and holds the fitting of the diagram into a grid.



## Explanation

The mission of this class is to analyse the structure of the supplied diagram and to place its objects on a grid such that, when the objects and the morphisms are actually drawn, the diagram would be “readable”, in the sense that there will not be many intersections of morphisms. This class does not perform any actual drawing. It does strive nevertheless to offer sufficient metadata to draw a diagram.

Consider the following simple diagram.

```
>>> from sympy.categories import Object, NamedMorphism
>>> from sympy.categories import Diagram, DiagramGrid
>>> from sympy import pprint
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g])
```

The simplest way to have a diagram laid out is the following:

```
>>> grid = DiagramGrid(diagram)
>>> (grid.width, grid.height)
(2, 2)
>>> pprint(grid)
A B
 C
```

Sometimes one sees the diagram as consisting of logical groups. One can advise `DiagramGrid` as to such groups by employing the `groups` keyword argument.

Consider the following diagram:

```
>>> D = Object("D")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> h = NamedMorphism(D, A, "h")
>>> k = NamedMorphism(D, B, "k")
>>> diagram = Diagram([f, g, h, k])
```

Lay it out with generic layout:

```
>>> grid = DiagramGrid(diagram)
>>> pprint(grid)
A B D
 C
```

Now, we can group the objects *A* and *D* to have them near one another:

```
>>> grid = DiagramGrid(diagram, groups=[[A, D], B, C])
>>> pprint(grid)
B C
```

(continues on next page)

(continued from previous page)

```
A D
```

Note how the positioning of the other objects changes.

Further indications can be supplied to the constructor of *DiagramGrid* (page 2752) using keyword arguments. The currently supported hints are explained in the following paragraphs.

*DiagramGrid* (page 2752) does not automatically guess which layout would suit the supplied diagram better. Consider, for example, the following linear diagram:

```
>>> E = Object("E")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> h = NamedMorphism(C, D, "h")
>>> i = NamedMorphism(D, E, "i")
>>> diagram = Diagram([f, g, h, i])
```

When laid out with the generic layout, it does not get to look linear:

```
>>> grid = DiagramGrid(diagram)
>>> pprint(grid)
A B
 C D
 E
```

To get it laid out in a line, use `layout="sequential"`:

```
>>> grid = DiagramGrid(diagram, layout="sequential")
>>> pprint(grid)
A B C D E
```

One may sometimes need to transpose the resulting layout. While this can always be done by hand, *DiagramGrid* (page 2752) provides a hint for that purpose:

```
>>> grid = DiagramGrid(diagram, layout="sequential", transpose=True)
>>> pprint(grid)
A
B
C
D
E
```

Separate hints can also be provided for each group. For an example, refer to `tests/test_drawing.py`, and see the different ways in which the five lemma [FiveLemma] can be laid out.

**See also:**[Diagram](#) (page 2749)**References**[\[FiveLemma\]](#)**property height**

Returns the number of rows in this diagram layout.

**Examples**

```

>>> from sympy.categories import Object, NamedMorphism
>>> from sympy.categories import Diagram, DiagramGrid
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g])
>>> grid = DiagramGrid(diagram)
>>> grid.height
2

```

**property morphisms**

Returns those morphisms (and their properties) which are sufficiently meaningful to be drawn.

**Examples**

```

>>> from sympy.categories import Object, NamedMorphism
>>> from sympy.categories import Diagram, DiagramGrid
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g])
>>> grid = DiagramGrid(diagram)
>>> grid.morphisms
{NamedMorphism(Object("A"), Object("B"), "f"): EmptySet,
 NamedMorphism(Object("B"), Object("C"), "g"): EmptySet}

```

**property width**

Returns the number of columns in this diagram layout.

## Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> from sympy.categories import Diagram, DiagramGrid
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g])
>>> grid = DiagramGrid(diagram)
>>> grid.width
2
```

```
class sympy.categories.diagram_drawing.ArrowStringDescription(unit, curving,
 curving_amount,
 looping_start,
 looping_end, horizontal_direction,
 vertical_direction,
 label_position,
 label)
```

Stores the information necessary for producing an Xy-pic description of an arrow.

The principal goal of this class is to abstract away the string representation of an arrow and to also provide the functionality to produce the actual Xy-pic string.

`unit` sets the unit which will be used to specify the amount of curving and other distances. `horizontal_direction` should be a string of "r" or "l" specifying the horizontal offset of the target cell of the arrow relatively to the current one. `vertical_direction` should specify the vertical offset using a series of either "d" or "u". `label_position` should be either "^", "\_", or "|" to specify that the label should be positioned above the arrow, below the arrow or just over the arrow, in a break. Note that the notions "above" and "below" are relative to arrow direction. `label` stores the morphism label.

This works as follows (disregard the yet unexplained arguments):

```
>>> from sympy.categories.diagram_drawing import ArrowStringDescription
>>> astr = ArrowStringDescription(
... unit="mm", curving=None, curving_amount=None,
... looping_start=None, looping_end=None, horizontal_direction="d",
... vertical_direction="r", label_position="_", label="f")
>>> print(str(astr))
\ar[dr]_{f}
```

`curving` should be one of "^", "\_" to specify in which direction the arrow is going to curve. `curving_amount` is a number describing how many unit's the morphism is going to curve:

```
>>> astr = ArrowStringDescription(
... unit="mm", curving="^", curving_amount=12,
... looping_start=None, looping_end=None, horizontal_direction="d",
... vertical_direction="r", label_position="_", label="f")
>>> print(str(astr))
\ar@/^12mm/[dr]_{f}
```

looping\_start and looping\_end are currently only used for loop morphisms, those which have the same domain and codomain. These two attributes should store a valid Xy-pic direction and specify, correspondingly, the direction the arrow gets out into and the direction the arrow gets back from:

```
>>> astr = ArrowStringDescription(
... unit="mm", curving=None, curving_amount=None,
... looping_start="u", looping_end="l", horizontal_direction="",
... vertical_direction="", label_position="_", label="f")
>>> print(str(astr))
\ar@(u,l)[]_{f}
```

label\_displacement controls how far the arrow label is from the ends of the arrow. For example, to position the arrow label near the arrow head, use ">":

```
>>> astr = ArrowStringDescription(
... unit="mm", curving="^", curving_amount=12,
... looping_start=None, looping_end=None, horizontal_direction="d",
... vertical_direction="r", label_position="_", label="f")
>>> astr.label_displacement = ">"
>>> print(str(astr))
\ar@/^12mm/[dr]_{>f}
```

Finally, arrow\_style is used to specify the arrow style. To get a dashed arrow, for example, use "{->}" as arrow style:

```
>>> astr = ArrowStringDescription(
... unit="mm", curving="^", curving_amount=12,
... looping_start=None, looping_end=None, horizontal_direction="d",
... vertical_direction="r", label_position="_", label="f")
>>> astr.arrow_style = "{->}"
>>> print(str(astr))
\ar@/^12mm/@{->}[dr]_{f}
```

## Notes

Instances of *ArrowStringDescription* (page 2756) will be constructed by *XypicDiagramDrawer* (page 2758) and provided for further use in formatters. The user is not expected to construct instances of *ArrowStringDescription* (page 2756) themselves.

To be able to properly utilise this class, the reader is encouraged to checkout the Xy-pic user guide, available at [Xypic].

### See also:

*XypicDiagramDrawer* (page 2758)

## References

[Xypic]

**class** sympy.categories.diagram\_drawing.XypicDiagramDrawer

Given a [Diagram](#) (page 2749) and the corresponding [DiagramGrid](#) (page 2752), produces the Xy-pic representation of the diagram.

The most important method in this class is `draw`. Consider the following triangle diagram:

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy.categories import DiagramGrid, XypicDiagramDrawer
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g], {g * f: "unique"})
```

To draw this diagram, its objects need to be laid out with a [DiagramGrid](#) (page 2752):

```
>>> grid = DiagramGrid(diagram)
```

Finally, the drawing:

```
>>> drawer = XypicDiagramDrawer()
>>> print(drawer.draw(diagram, grid))
\xymatrix{
A \ar[d]_{g\circ f} \ar[r]^{f} & B \ar[ld]^{g} \\
C & \\
}
```

For further details see the docstring of this method.

To control the appearance of the arrows, formatters are used. The dictionary `arrow_formatters` maps morphisms to formatter functions. A formatter accepts an [ArrowStringDescription](#) (page 2756) and is allowed to modify any of the arrow properties exposed thereby. For example, to have all morphisms with the property `unique` appear as dashed arrows, and to have their names prepended with  $\exists!$ , the following should be done:

```
>>> def formatter(astr):
... astr.label = r"\exists !" + astr.label
... astr.arrow_style = "{-->}"
>>> drawer.arrow_formatters["unique"] = formatter
>>> print(drawer.draw(diagram, grid))
\xymatrix{
A \ar@{->}[d]_{\exists !g\circ f} \ar[r]^{f} & B \ar[ld]^{g} \\
C & \\
}
```

To modify the appearance of all arrows in the diagram, set `default_arrow_formatter`. For example, to place all morphism labels a little bit farther from the arrow head so that they look more centred, do as follows:

```
>>> def default_formatter(astr):
... astr.label_displacement = "(0.45)"
>>> drawer.default_arrow_formatter = default_formatter
>>> print(drawer.draw(diagram, grid))
\xymatrix{
A \ar@{->}[d]_(0.45){\exists !g\circ f} \ar[r]^(0.45){f} & B \ar[l]^(0.
\rightarrow 45){g} \\
C &
}
```

In some diagrams some morphisms are drawn as curved arrows. Consider the following diagram:

```
>>> D = Object("D")
>>> E = Object("E")
>>> h = NamedMorphism(D, A, "h")
>>> k = NamedMorphism(D, B, "k")
>>> diagram = Diagram([f, g, h, k])
>>> grid = DiagramGrid(diagram)
>>> drawer = XypicDiagramDrawer()
>>> print(drawer.draw(diagram, grid))
\xymatrix{
A \ar[r]_f & B \ar[d]^g & D \ar[l]^{k} \ar@/_3mm/[ll]_h \\
& C &
}
```

To control how far the morphisms are curved by default, one can use the `unit` and `default_curving_amount` attributes:

```
>>> drawer.unit = "cm"
>>> drawer.default_curving_amount = 1
>>> print(drawer.draw(diagram, grid))
\xymatrix{
A \ar[r]_f & B \ar[d]^g & D \ar[l]^{k} \ar@/_1cm/[ll]_h \\
& C &
}
```

In some diagrams, there are multiple curved morphisms between the same two objects. To control by how much the curving changes between two such successive morphisms, use `default_curving_step`:

```
>>> drawer.default_curving_step = 1
>>> h1 = NamedMorphism(A, D, "h1")
>>> diagram = Diagram([f, g, h, k, h1])
>>> grid = DiagramGrid(diagram)
>>> print(drawer.draw(diagram, grid))
\xymatrix{
A \ar[r]_f \ar@/^1cm/[rr]^{h_1} & B \ar[d]^g & D \ar[l]^{k} \ar@/_
\rightarrow 2cm/[ll]_h \\
& C &
}
```

The default value of `default_curving_step` is 4 units.

See also:

[draw](#) (page 2760), [ArrowStringDescription](#) (page 2756)

**draw**(*diagram*, *grid*, *masked=None*, *diagram\_format=""*)

Returns the Xy-pic representation of diagram laid out in grid.

Consider the following simple triangle diagram.

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy.categories import DiagramGrid, XypicDiagramDrawer
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g], {g * f: "unique"})
```

To draw this diagram, its objects need to be laid out with a [DiagramGrid](#) (page 2752):

```
>>> grid = DiagramGrid(diagram)
```

Finally, the drawing:

```
>>> drawer = XypicDiagramDrawer()
>>> print(drawer.draw(diagram, grid))
\xymatrix{
A \ar[d]_{g\circ f} \ar[r]^{f} & B \ar[ld]^{g} \\
C & \\
}
```

The argument *masked* can be used to skip morphisms in the presentation of the diagram:

```
>>> print(drawer.draw(diagram, grid, masked=[g * f]))
\xymatrix{
A \ar[r]^{f} & B \ar[ld]^{g} \\
C & \\
}
```

Finally, the *diagram\_format* argument can be used to specify the format string of the diagram. For example, to increase the spacing by 1 cm, proceeding as follows:

```
>>> print(drawer.draw(diagram, grid, diagram_format="@+1cm"))
\xymatrix@+1cm{
A \ar[d]_{g\circ f} \ar[r]^{f} & B \ar[ld]^{g} \\
C & \\
}
```

`sympy.categories.diagram_drawing.xypic_draw_diagram`(*diagram*, *masked=None*,  
*diagram\_format=""*,  
*groups=None*, *\*\*hints*)

Provides a shortcut combining [DiagramGrid](#) (page 2752) and [XypicDiagramDrawer](#) (page 2758). Returns an Xy-pic presentation of diagram. The argument *masked* is a list of morphisms which will be not be drawn. The argument *diagram\_format* is the



format string inserted after “xymatrix”. groups should be a set of logical groups. The hints will be passed directly to the constructor of *DiagramGrid* (page 2752).

For more information about the arguments, see the docstrings of *DiagramGrid* (page 2752) and *XypicDiagramDrawer*.draw.

### Examples

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy.categories import xypic_draw_diagram
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g], {g * f: "unique"})
>>> print(xypic_draw_diagram(diagram))
\xymatrix{
A \ar[d]_{g \circ f} \ar[r]^{f} & B \ar[l]^{g} \\
C & \\
}
```

#### See also:

*XypicDiagramDrawer* (page 2758), *DiagramGrid* (page 2752)

```
sympy.categories.diagram_drawing.preview_diagram(diagram, masked=None,
 diagram_format="",
 groups=None, output='png',
 viewer=None, euler=True,
 **hints)
```

Combines the functionality of *xypic\_draw\_diagram* and *sympy.printing.preview*. The arguments *masked*, *diagram\_format*, *groups*, and *hints* are passed to *xypic\_draw\_diagram*, while *output*, *viewer*, and *euler* are passed to *preview*.

### Examples

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy.categories import preview_diagram
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g], {g * f: "unique"})
>>> preview_diagram(d)
```

#### See also:

*XypicDiagramDrawer* (page 2758)

## Cryptography

**Warning:** This module is intended for educational purposes only. Do not use the functions in this module for real cryptographic applications. If you wish to encrypt real data, we recommend using something like the [cryptography](#) module.

Encryption is the process of hiding a message and a cipher is a means of doing so. Included in this module are both block and stream ciphers:

- Shift cipher
- Affine cipher
- substitution ciphers
- Vigenere’s cipher
- Hill’s cipher
- Bifid ciphers
- RSA
- Kid RSA
- linear-feedback shift registers (for stream ciphers)
- ElGamal encryption

In a *substitution cipher* “units” (not necessarily single characters) of plaintext are replaced with ciphertext according to a regular system.

A *transposition cipher* is a method of encryption by which the positions held by “units” of plaintext are replaced by a permutation of the plaintext. That is, the order of the units is changed using a bijective function on the position of the characters to perform the encryption.

A *monoalphabetic cipher* uses fixed substitution over the entire message, whereas a *polyalphabetic cipher* uses a number of substitutions at different times in the message.

`sympy.crypto.crypto.AZ(s=None)`

Return the letters of `s` in uppercase. In case more than one string is passed, each of them will be processed and a list of upper case strings will be returned.

### Examples

```
>>> from sympy.crypto.crypto import AZ
>>> AZ('Hello, world!')
'HELLOWORLD'
>>> AZ('Hello, world!'.split())
['HELLO', 'WORLD']
```

### See also:

[check\\_and\\_join](#) (page 2763)

`sympy.crypto.crypto.padded_key(key, symbols)`

Return a string of the distinct characters of `symbols` with those of `key` appearing first. A `ValueError` is raised if a) there are duplicate characters in `symbols` or b) there are characters in `key` that are not in `symbols`.

### Examples

```
>>> from sympy.crypto.crypto import padded_key
>>> padded_key('PUPPY', 'OPQRSTUVWXYZ')
'PUYQIRSTVWX'
>>> padded_key('RSA', 'ARTIST')
Traceback (most recent call last):
...
ValueError: duplicate characters in symbols: T
```

`sympy.crypto.crypto.check_and_join(phrase, symbols=None, filter=None)`

Joins characters of `phrase` and if `symbols` is given, raises an error if any character in `phrase` is not in `symbols`.

#### Parameters

##### **phrase**

String or list of strings to be returned as a string.

##### **symbols**

Iterable of characters allowed in phrase.

If `symbols` is `None`, no checking is performed.

### Examples

```
>>> from sympy.crypto.crypto import check_and_join
>>> check_and_join('a phrase')
'a phrase'
>>> check_and_join('a phrase'.upper().split())
'APHRASE'
>>> check_and_join('a phrase!'.upper().split(), 'ARE', filter=True)
'ARAE'
>>> check_and_join('a phrase!'.upper().split(), 'ARE')
Traceback (most recent call last):
...
ValueError: characters in phrase but not symbols: "!HPS"
```

`sympy.crypto.crypto.cycle_list(k, n)`

Returns the elements of the list `range(n)` shifted to the left by `k` (so the list starts with `k (mod n)`).

## Examples

```
>>> from sympy.crypto.crypto import cycle_list
>>> cycle_list(3, 10)
[3, 4, 5, 6, 7, 8, 9, 0, 1, 2]
```

`sympy.crypto.crypto.encipher_shift(msg, key, symbols=None)`

Performs shift cipher encryption on plaintext msg, and returns the ciphertext.

### Parameters

**key** : int

The secret key.

**msg** : str

Plaintext of upper-case letters.

### Returns

str

Ciphertext of upper-case letters.

## Examples

```
>>> from sympy.crypto.crypto import encipher_shift, decipher_shift
>>> msg = "GONAVYBEATARMY"
>>> ct = encipher_shift(msg, 1); ct
'HPOBWZCFBUBSNZ'
```

To decipher the shifted text, change the sign of the key:

```
>>> encipher_shift(ct, -1)
'GONAVYBEATARMY'
```

There is also a convenience function that does this with the original key:

```
>>> decipher_shift(ct, 1)
'GONAVYBEATARMY'
```

## Notes

ALGORITHM:

### STEPS:

0. Number the letters of the alphabet from 0, ..., N
1. Compute from the string msg a list L1 of corresponding integers.
2. Compute from the list L1 a new list L2, given by adding  $(k \bmod 26)$  to each element in L1.
3. Compute from the list L2 a string ct of corresponding letters.

The shift cipher is also called the Caesar cipher, after Julius Caesar, who, according to Suetonius, used it with a shift of three to protect messages of military significance. Caesar's nephew Augustus reportedly used a similar cipher, but with a right shift of 1.

**See also:**

[\*decipher\\_shift\*](#) (page 2765)

## References

[R136], [R137]

`sympy.crypto.crypto.decipher_shift(msg, key, symbols=None)`

Return the text by shifting the characters of msg to the left by the amount given by key.

## Examples

```
>>> from sympy.crypto.crypto import encipher_shift, decipher_shift
>>> msg = "GONAVYBEATARMY"
>>> ct = encipher_shift(msg, 1); ct
'HPOBWZCFBUBSNZ'
```

To decipher the shifted text, change the sign of the key:

```
>>> encipher_shift(ct, -1)
'GONAVYBEATARMY'
```

Or use this function with the original key:

```
>>> decipher_shift(ct, 1)
'GONAVYBEATARMY'
```

`sympy.crypto.crypto.encipher_rot13(msg, symbols=None)`

Performs the ROT13 encryption on a given plaintext msg.

## Explanation

ROT13 is a substitution cipher which substitutes each letter in the plaintext message for the letter furthest away from it in the English alphabet.

Equivalently, it is just a Caesar (shift) cipher with a shift key of 13 (midway point of the alphabet).

**See also:**

[\*decipher\\_rot13\*](#) (page 2766), [\*encipher\\_shift\*](#) (page 2764)

## References

[R138]

`sympy.crypto.crypto.decipher_rot13(msg, symbols=None)`

Performs the ROT13 decryption on a given plaintext msg.

## Explanation

`decipher_rot13` is equivalent to `encipher_rot13` as both `decipher_shift` with a key of 13 and `encipher_shift` key with a key of 13 will return the same results. Nonetheless, `decipher_rot13` has nonetheless been explicitly defined here for consistency.

## Examples

```
>>> from sympy.crypto.crypto import encipher_rot13, decipher_rot13
>>> msg = 'GONAVYBEATARMY'
>>> ciphertext = encipher_rot13(msg);ciphertext
'TBANILORNGNEZL'
>>> decipher_rot13(ciphertext)
'GONAVYBEATARMY'
>>> encipher_rot13(msg) == decipher_rot13(msg)
True
>>> msg == decipher_rot13(ciphertext)
True
```

`sympy.crypto.crypto.encipher_affine(msg, key, symbols=None, _inverse=False)`

Performs the affine cipher encryption on plaintext msg, and returns the ciphertext.

### Parameters

**msg** : str

Characters that appear in symbols.

**a, b** : int, int

A pair integers, with  $\gcd(a, N) = 1$  (the secret key).

### symbols

String of characters (default = uppercase letters).

When no symbols are given, msg is converted to upper case letters and all other characters are ignored.

### Returns

ct

String of characters (the ciphertext message)

## Explanation

Encryption is based on the map  $x \rightarrow ax + b \pmod{N}$  where  $N$  is the number of characters in the alphabet. Decryption is based on the map  $x \rightarrow cx + d \pmod{N}$ , where  $c = a^{-1} \pmod{N}$  and  $d = -a^{-1}b \pmod{N}$ . In particular, for the map to be invertible, we need  $\gcd(a, N) = 1$  and an error will be raised if this is not true.

## Notes

ALGORITHM:

### STEPS:

0. Number the letters of the alphabet from 0, ...,  $N$
1. Compute from the string `msg` a list `L1` of corresponding integers.
2. Compute from the list `L1` a new list `L2`, given by replacing  $x$  by  $a*x + b \pmod{N}$ , for each element  $x$  in `L1`.
3. Compute from the list `L2` a string `ct` of corresponding letters.

This is a straightforward generalization of the shift cipher with the added complexity of requiring 2 characters to be deciphered in order to recover the key.

**See also:**

[`decipher\_affine`](#) (page 2767)

## References

[R139]

`sympy.crypto.crypto.decipher_affine(msg, key, symbols=None)`

Return the deciphered text that was made from the mapping,  $x \rightarrow ax + b \pmod{N}$ , where  $N$  is the number of characters in the alphabet. Deciphering is done by reciphering with a new key:  $x \rightarrow cx + d \pmod{N}$ , where  $c = a^{-1} \pmod{N}$  and  $d = -a^{-1}b \pmod{N}$ .

## Examples

```
>>> from sympy.crypto.crypto import encipher_affine, decipher_affine
>>> msg = "GO NAVY BEAT ARMY"
>>> key = (3, 1)
>>> encipher_affine(msg, key)
'TROBMVENBGBALV'
>>> decipher_affine(_, key)
'GONAVYBEATARMY'
```

**See also:**

[`encipher\_affine`](#) (page 2766)

`sympy.crypto.crypto.encipher_atbash(msg, symbols=None)`

Enciphers a given `msg` into its Atbash ciphertext and returns it.

### Explanation

Atbash is a substitution cipher originally used to encrypt the Hebrew alphabet. Atbash works on the principle of mapping each alphabet to its reverse / counterpart (i.e. a would map to z, b to y etc.)

Atbash is functionally equivalent to the affine cipher with  $a = 25$  and  $b = 25$

#### See also:

[\*decipher\\_atbash\*](#) (page 2768)

`sympy.crypto.crypto.decipher_atbash(msg, symbols=None)`

Deciphers a given msg using Atbash cipher and returns it.

### Explanation

`decipher_atbash` is functionally equivalent to `encipher_atbash`. However, it has still been added as a separate function to maintain consistency.

### Examples

```
>>> from sympy.crypto.crypto import encipher_atbash, decipher_atbash
>>> msg = 'GONAVYBEATARMY'
>>> encipher_atbash(msg)
'TLMZEBYVZGZINB'
>>> decipher_atbash(msg)
'TLMZEBYVZGZINB'
>>> encipher_atbash(msg) == decipher_atbash(msg)
True
>>> msg == encipher_atbash(encipher_atbash(msg))
True
```

#### See also:

[\*encipher\\_atbash\*](#) (page 2767)

### References

[R140]

`sympy.crypto.crypto.encipher_substitution(msg, old, new=None)`

Returns the ciphertext obtained by replacing each character that appears in `old` with the corresponding character in `new`. If `old` is a mapping, then `new` is ignored and the replacements defined by `old` are used.



## Explanation

This is a more general than the affine cipher in that the key can only be recovered by determining the mapping for each symbol. Though in practice, once a few symbols are recognized the mappings for other characters can be quickly guessed.

## Examples

```
>>> from sympy.crypto.crypto import encipher_substitution, AZ
>>> old = 'OEYAG'
>>> new = 'Q34^6'
>>> msg = AZ("go navy! beat army!")
>>> ct = encipher_substitution(msg, old, new); ct
'60N^V4B3^T^RM4'
```

To decrypt a substitution, reverse the last two arguments:

```
>>> encipher_substitution(ct, new, old)
'GONAVYBEATARMY'
```

In the special case where old and new are a permutation of order 2 (representing a transposition of characters) their order is immaterial:

```
>>> old = 'NAVY'
>>> new = 'ANYV'
>>> encipher = lambda x: encipher_substitution(x, old, new)
>>> encipher('NAVY')
'ANYV'
>>> encipher(_)
'NAVY'
```

The substitution cipher, in general, is a method whereby “units” (not necessarily single characters) of plaintext are replaced with ciphertext according to a regular system.

```
>>> ords = dict(zip('abc', ['\\%i' % ord(i) for i in 'abc']))
>>> print(encipher_substitution('abc', ords))
\97\98\99
```

## References

[R141]

`sympy.crypto.crypto.encipher_vigenere(msg, key, symbols=None)`

Performs the Vigenere cipher encryption on plaintext msg, and returns the ciphertext.

## Examples

```
>>> from sympy.crypto.crypto import encipher_vigenere, AZ
>>> key = "encrypt"
>>> msg = "meet me on monday"
>>> encipher_vigenere(msg, key)
'QRGKKTHRZQEBPR'
```

Section 1 of the Kryptos sculpture at the CIA headquarters uses this cipher and also changes the order of the alphabet [R143]. Here is the first line of that section of the sculpture:

```
>>> from sympy.crypto.crypto import decipher_vigenere, padded_key
>>> alp = padded_key('KRYPTOS', AZ())
>>> key = 'PALIMPSEST'
>>> msg = 'EMUFPHZLRFAXYUSDJKZLDKRNSHGNFIVJ'
>>> decipher_vigenere(msg, key, alp)
'BETWEENSUBTLESHADINGANDTHEABSENC'
```

## Explanation

The Vigenere cipher is named after Blaise de Vigenere, a sixteenth century diplomat and cryptographer, by a historical accident. Vigenere actually invented a different and more complicated cipher. The so-called *Vigenere cipher* was actually invented by Giovan Batista Belaso in 1553.

This cipher was used in the 1800's, for example, during the American Civil War. The Confederacy used a brass cipher disk to implement the Vigenere cipher (now on display in the NSA Museum in Fort Meade) [R142].

The Vigenere cipher is a generalization of the shift cipher. Whereas the shift cipher shifts each letter by the same amount (that amount being the key of the shift cipher) the Vigenere cipher shifts a letter by an amount determined by the key (which is a word or phrase known only to the sender and receiver).

For example, if the key was a single letter, such as "C", then the so-called Vigenere cipher is actually a shift cipher with a shift of 2 (since "C" is the 2nd letter of the alphabet, if you start counting at 0). If the key was a word with two letters, such as "CA", then the so-called Vigenere cipher will shift letters in even positions by 2 and letters in odd positions are left alone (shifted by 0, since "A" is the 0th letter, if you start counting at 0).

ALGORITHM:

INPUT:

msg: string of characters that appear in symbols (the plaintext)  
key: a string of characters that appear in symbols (the secret key)  
symbols: a string of letters defining the alphabet

OUTPUT:

ct: string of characters (the ciphertext message)

STEPS:

0. Number the letters of the alphabet from 0, ..., N

1. Compute from the string `key` a list `L1` of corresponding integers. Let `n1 = len(L1)`.
2. Compute from the string `msg` a list `L2` of corresponding integers. Let `n2 = len(L2)`.
3. Break `L2` up sequentially into sublists of size `n1`; the last sublist may be smaller than `n1`.
4. For each of these sublists `L` of `L2`, compute a new list `C` given by  $C[i] = L[i] + L1[i] \pmod{N}$  to the  $i$ -th element in the sublist, for each  $i$ .
5. Assemble these lists `C` by concatenation into a new list of length `n2`.
6. Compute from the new list a string `ct` of corresponding letters.

Once it is known that the key is, say,  $n$  characters long, frequency analysis can be applied to every  $n$ -th letter of the ciphertext to determine the plaintext. This method is called *Kasiski examination* (although it was first discovered by Babbage). If the key is as long as the message and is comprised of randomly selected characters – a one-time pad – the message is theoretically unbreakable.

The cipher Vigenere actually discovered is an “auto-key” cipher described as follows.

ALGORITHM:

INPUT:

key: a string of letters (the secret key)

msg: string of letters (the plaintext message)

OUTPUT:

ct: string of upper-case letters (the ciphertext message)

STEPS:

0. Number the letters of the alphabet from 0, ...,  $N$
1. Compute from the string `msg` a list `L2` of corresponding integers. Let `n2 = len(L2)`.
2. Let `n1` be the length of the key. Append to the string `key` the first `n2 - n1` characters of the plaintext message. Compute from this string (also of length `n2`) a list `L1` of integers corresponding to the letter numbers in the first step.
3. Compute a new list `C` given by  $C[i] = L1[i] + L2[i] \pmod{N}$ .
4. Compute from the new list a string `ct` of letters corresponding to the new integers.

To decipher the auto-key ciphertext, the key is used to decipher the first `n1` characters and then those characters become the key to decipher the next `n1` characters, etc...:

```
>>> m = AZ('go navy, beat army! yes you can'); m
'GONAVYBEATARMYYESYOU CAN'
>>> key = AZ('gold bug'); n1 = len(key); n2 = len(m)
>>> auto_key = key + m[:n2 - n1]; auto_key
'GOLDBUGGONAVYBEATARMYYE'
>>> ct = encipher_vigenere(m, auto_key); ct
```

(continues on next page)

(continued from previous page)

```
'MCYDWSHKOGAMKZCELYFGAYR'
>>> n1 = len(key)
>>> pt = []
>>> while ct:
... part, ct = ct[:n1], ct[n1:]
... pt.append(decipher_vigenere(part, key))
... key = pt[-1]
...
>>> ''.join(pt) == m
True
```

## References

[R142], [R143], [R144]

`sympy.crypto.crypto.decipher_vigenere(msg, key, symbols=None)`  
Decode using the Vigenere cipher.

## Examples

```
>>> from sympy.crypto.crypto import decipher_vigenere
>>> key = "encrypt"
>>> ct = "QRGK kt HRZQE BPR"
>>> decipher_vigenere(ct, key)
'MEETMEONMONDAY'
```

`sympy.crypto.crypto.encipher_hill(msg, key, symbols=None, pad='Q')`  
Return the Hill cipher encryption of msg.

### Parameters

#### **msg**

Plaintext message of  $n$  upper-case letters.

#### **key**

A  $k \times k$  invertible matrix  $K$ , all of whose entries are in  $Z_{26}$  (or whatever number of symbols are being used).

#### **pad**

Character (default "Q") to use to make length of text be a multiple of  $k$ .

### Returns

ct

Ciphertext of upper-case letters.

## Explanation

The Hill cipher [R145], invented by Lester S. Hill in the 1920's [R146], was the first polygraphic cipher in which it was practical (though barely) to operate on more than three symbols at once. The following discussion assumes an elementary knowledge of matrices.

First, each letter is first encoded as a number starting with 0. Suppose your message *msg* consists of  $n$  capital letters, with no spaces. This may be regarded an  $n$ -tuple  $M$  of elements of  $Z_{26}$  (if the letters are those of the English alphabet). A key in the Hill cipher is a  $k \times k$  matrix  $K$ , all of whose entries are in  $Z_{26}$ , such that the matrix  $K$  is invertible (i.e., the linear transformation  $K : Z_N^k \rightarrow Z_N^k$  is one-to-one).

## Notes

ALGORITHM:

### STEPS:

0. Number the letters of the alphabet from 0, ..., N
1. Compute from the string *msg* a list *L* of corresponding integers. Let  $n = \text{len}(L)$ .
2. Break the list *L* up into  $t = \text{ceiling}(n/k)$  sublists  $L_1, \dots, L_t$  of size  $k$  (with the last list "padded" to ensure its size is  $k$ ).
3. Compute new list  $C_1, \dots, C_t$  given by  $C[i] = K * L_i$  (arithmetic is done mod  $N$ ), for each  $i$ .
4. Concatenate these into a list  $C = C_1 + \dots + C_t$ .
5. Compute from *C* a string *ct* of corresponding letters. This has length  $k*t$ .

See also:

[\*decipher\\_hill\*](#) (page 2773)

## References

[R145], [R146]

`sympy.crypto.crypto.decipher_hill(msg, key, symbols=None)`

Deciphering is the same as enciphering but using the inverse of the key matrix.

## Examples

```
>>> from sympy.crypto.crypto import encipher_hill, decipher_hill
>>> from sympy import Matrix
```

```
>>> key = Matrix([[1, 2], [3, 5]])
>>> encipher_hill("meet me on monday", key)
'UEQDUEODOCTCWQ'
```

(continues on next page)

(continued from previous page)

```
>>> decipher_hill(_, key)
'MEETMEONMONDAY'
```

When the length of the plaintext (stripped of invalid characters) is not a multiple of the key dimension, extra characters will appear at the end of the enciphered and deciphered text. In order to decipher the text, those characters must be included in the text to be deciphered. In the following, the key has a dimension of 4 but the text is 2 short of being a multiple of 4 so two characters will be added.

```
>>> key = Matrix([[1, 1, 1, 2], [0, 1, 1, 0],
... [2, 2, 3, 4], [1, 1, 0, 1]])
>>> msg = "ST"
>>> encipher_hill(msg, key)
'HJEB'
>>> decipher_hill(_, key)
'STQQ'
>>> encipher_hill(msg, key, pad="Z")
'ISPK'
>>> decipher_hill(_, key)
'STZZ'
```

If the last two characters of the ciphertext were ignored in either case, the wrong plaintext would be recovered:

```
>>> decipher_hill("HD", key)
'ORMV'
>>> decipher_hill("IS", key)
'UIKY'
```

**See also:**

[\*encipher\\_hill\*](#) (page 2772)

`sympy.crypto.crypto.encipher_bifid(msg, key, symbols=None)`

Performs the Bifid cipher encryption on plaintext `msg`, and returns the ciphertext.

This is the version of the Bifid cipher that uses an  $n \times n$  Polybius square.

**Parameters****msg**

Plaintext string.

**key**

Short string for key.

Duplicate characters are ignored and then it is padded with the characters in `symbols` that were not in the short key.

**symbols**

$n \times n$  characters defining the alphabet.

(default is `string.printable`)

**Returns**

ciphertext

Ciphertext using Bifid5 cipher without spaces.

**See also:**

[`decipher\_bifid`](#) (page 2775), [`encipher\_bifid5`](#) (page 2776), [`encipher\_bifid6`](#) (page 2778)

## References

[R147]

`sympy.crypto.crypto.decipher_bifid(msg, key, symbols=None)`

Performs the Bifid cipher decryption on ciphertext `msg`, and returns the plaintext.

This is the version of the Bifid cipher that uses the  $n \times n$  Polybius square.

### Parameters

#### **msg**

Ciphertext string.

#### **key**

Short string for key.

Duplicate characters are ignored and then it is padded with the characters in `symbols` that were not in the short key.

#### **symbols**

$n \times n$  characters defining the alphabet.

(default=`string.printable`, a  $10 \times 10$  matrix)

### Returns

deciphered

Deciphered text.

## Examples

```
>>> from sympy.crypto.crypto import (
... encipher_bifid, decipher_bifid, AZ)
```

Do an encryption using the bifid5 alphabet:

```
>>> alp = AZ().replace('J', '')
>>> ct = AZ("meet me on monday!")
>>> key = AZ("gold bug")
>>> encipher_bifid(ct, key, alp)
'IEILHHFSTSFQYE'
```

When entering the text or ciphertext, spaces are ignored so it can be formatted as desired. Re-entering the ciphertext from the preceding, putting 4 characters per line and padding with an extra J, does not cause problems for the deciphering:

```
>>> decipher_bifid(''
... IEILH
... HFSTS
... FQYEJ'', key, alp)
'MEETMEONMONDAY'
```

When no alphabet is given, all 100 printable characters will be used:

```
>>> key = ''
>>> encipher_bifid('hello world!', key)
'bmtwmg-bIo*w'
>>> decipher_bifid(_, key)
'hello world!'
```

If the key is changed, a different encryption is obtained:

```
>>> key = 'gold bug'
>>> encipher_bifid('hello world!', 'gold_bug')
'hg2sfuei7t}w'
```

And if the key used to decrypt the message is not exact, the original text will not be perfectly obtained:

```
>>> decipher_bifid(_, 'gold pug')
'helldo~wor6d!'
```

`sympy.crypto.crypto.bifid5_square(key=None)`

5x5 Polybius square.

Produce the Polybius square for the  $5 \times 5$  Bifid cipher.

### Examples

```
>>> from sympy.crypto.crypto import bifid5_square
>>> bifid5_square("gold bug")
Matrix([
[G, O, L, D, B],
[U, A, C, E, F],
[H, I, K, M, N],
[P, Q, R, S, T],
[V, W, X, Y, Z]])
```

`sympy.crypto.crypto.encipher_bifid5(msg, key)`

Performs the Bifid cipher encryption on plaintext `msg`, and returns the ciphertext.

#### Parameters

**msg** : str

Plaintext string.

Converted to upper case and filtered of anything but all letters except J.

**key**



Short string for key; non-alphabetic letters, J and duplicated characters are ignored and then, if the length is less than 25 characters, it is padded with other letters of the alphabet (in alphabetical order).

### Returns

ct

Ciphertext (all caps, no spaces).

### Explanation

This is the version of the Bifid cipher that uses the  $5 \times 5$  Polybius square. The letter "J" is ignored so it must be replaced with something else (traditionally an "I") before encryption.

ALGORITHM: (5x5 case)

#### STEPS:

0. Create the  $5 \times 5$  Polybius square  $S$  associated to key as follows:
  - a) moving from left-to-right, top-to-bottom, place the letters of the key into a  $5 \times 5$  matrix,
  - b) if the key has less than 25 letters, add the letters of the alphabet not in the key until the  $5 \times 5$  square is filled.
1. Create a list  $P$  of pairs of numbers which are the coordinates in the Polybius square of the letters in  $msg$ .
2. Let  $L1$  be the list of all first coordinates of  $P$  (length of  $L1 = n$ ), let  $L2$  be the list of all second coordinates of  $P$  (so the length of  $L2$  is also  $n$ ).
3. Let  $L$  be the concatenation of  $L1$  and  $L2$  (length  $L = 2*n$ ), except that consecutive numbers are paired ( $L[2*i]$ ,  $L[2*i + 1]$ ). You can regard  $L$  as a list of pairs of length  $n$ .
4. Let  $C$  be the list of all letters which are of the form  $S[i, j]$ , for all  $(i, j)$  in  $L$ . As a string, this is the ciphertext of  $msg$ .

### Examples

```
>>> from sympy.crypto.crypto import (
... encipher_bifid5, decipher_bifid5)
```

"J" will be omitted unless it is replaced with something else:

```
>>> round_trip = lambda m, k: \
... decipher_bifid5(encipher_bifid5(m, k), k)
>>> key = 'a'
>>> msg = "JOSIE"
>>> round_trip(msg, key)
'OSIE'
>>> round_trip(msg.replace("J", "I"), key)
'IOSIE'
>>> j = "QIQ"
```

(continues on next page)

(continued from previous page)

```
>>> round_trip(msg.replace("J", j), key).replace(j, "J")
'JOSIE'
```

## Notes

The Bifid cipher was invented around 1901 by Felix Delastelle. It is a *fractional substitution* cipher, where letters are replaced by pairs of symbols from a smaller alphabet. The cipher uses a  $5 \times 5$  square filled with some ordering of the alphabet, except that “J” is replaced with “I” (this is a so-called Polybius square; there is a  $6 \times 6$  analog if you add back in “J” and also append onto the usual 26 letter alphabet, the digits 0, 1, ..., 9). According to Helen Gaines’ book *Cryptanalysis*, this type of cipher was used in the field by the German Army during World War I.

### See also:

[\*decipher\\_bifid5\*](#) (page 2778), [\*encipher\\_bifid\*](#) (page 2774)

`sympy.crypto.crypto.decipher_bifid5(msg, key)`

Return the Bifid cipher decryption of msg.

### Parameters

#### **msg**

Ciphertext string.

#### **key**

Short string for key; duplicated characters are ignored and if the length is less than 25 characters, it will be padded with other letters from the alphabet omitting “J”. Non-alphabetic characters are ignored.

### Returns

plaintext

Plaintext from Bifid5 cipher (all caps, no spaces).

## Explanation

This is the version of the Bifid cipher that uses the  $5 \times 5$  Polybius square; the letter “J” is ignored unless a key of length 25 is used.

## Examples

```
>>> from sympy.crypto.crypto import encipher_bifid5, decipher_bifid5
>>> key = "gold bug"
>>> encipher_bifid5('meet me on friday', key)
'IEILEHFSFSXEE'
>>> encipher_bifid5('meet me on monday', key)
'IEILHHFSFSFYE'
>>> decipher_bifid5(_, key)
'MEETMEONMONDAY'
```

`sympy.crypto.crypto.encipher_bifid6(msg, key)`

Performs the Bifid cipher encryption on plaintext `msg`, and returns the ciphertext.

This is the version of the Bifid cipher that uses the  $6 \times 6$  Polybius square.

#### Parameters

##### **msg**

Plaintext string (digits okay).

##### **key**

Short string for key (digits okay).

If key is less than 36 characters long, the square will be filled with letters A through Z and digits 0 through 9.

#### Returns

ciphertext

Ciphertext from Bifid cipher (all caps, no spaces).

#### See also:

[\*decipher\\_bifid6\*](#) (page 2779), [\*encipher\\_bifid\*](#) (page 2774)

`sympy.crypto.crypto.decipher_bifid6(msg, key)`

Performs the Bifid cipher decryption on ciphertext `msg`, and returns the plaintext.

This is the version of the Bifid cipher that uses the  $6 \times 6$  Polybius square.

#### Parameters

##### **msg**

Ciphertext string (digits okay); converted to upper case

##### **key**

Short string for key (digits okay).

If key is less than 36 characters long, the square will be filled with letters A through Z and digits 0 through 9. All letters are converted to uppercase.

#### Returns

plaintext

Plaintext from Bifid cipher (all caps, no spaces).

#### Examples

```
>>> from sympy.crypto.crypto import encipher_bifid6, decipher_bifid6
>>> key = "gold bug"
>>> encipher_bifid6('meet me on monday at 8am', key)
'KFKLJJHF5MMMKTFRGPL'
>>> decipher_bifid6(_, key)
'MEETMEONMONDAYAT8AM'
```

`sympy.crypto.crypto.bifid6_square(key=None)`

6x6 Polybius square.

Produces the Polybius square for the  $6 \times 6$  Bifid cipher. Assumes alphabet of symbols is "A", ..., "Z", "0", ..., "9".

### Examples

```
>>> from sympy.crypto.crypto import bifid6_square
>>> key = "gold bug"
>>> bifid6_square(key)
Matrix([
 [G, O, L, D, B, U],
 [A, C, E, F, H, I],
 [J, K, M, N, P, Q],
 [R, S, T, V, W, X],
 [Y, Z, 0, 1, 2, 3],
 [4, 5, 6, 7, 8, 9]])
```

`sympy.crypto.crypto.rsa_public_key(*args, **kwargs)`

Return the RSA *public* key pair,  $(n, e)$

#### Parameters

**args** : naturals

If specified as  $p, q, e$  where  $p$  and  $q$  are distinct primes and  $e$  is a desired public exponent of the RSA,  $n = pq$  and  $e$  will be verified against the totient  $\phi(n)$  (Euler totient) or  $\lambda(n)$  (Carmichael totient) to be  $\gcd(e, \phi(n)) = 1$  or  $\gcd(e, \lambda(n)) = 1$ .

If specified as  $p_1, p_2, \dots, p_n, e$  where  $p_1, p_2, \dots, p_n$  are specified as primes, and  $e$  is specified as a desired public exponent of the RSA, it will be able to form a multi-prime RSA, which is a more generalized form of the popular 2-prime RSA.

It can also be possible to form a single-prime RSA by specifying the argument as  $p, e$ , which can be considered a trivial case of a multi-prime RSA.

Furthermore, it can be possible to form a multi-power RSA by specifying two or more pairs of the primes to be same. However, unlike the two-distinct prime RSA or multi-prime RSA, not every numbers in the complete residue system  $(\mathbb{Z}_n)$  will be decryptable since the mapping  $\mathbb{Z}_n \rightarrow \mathbb{Z}_n$  will not be bijective. (Only except for the trivial case when  $e = 1$  or more generally,

$$e \in \{1 + k\lambda(n) \mid k \in \mathbb{Z} \wedge k \geq 0\}$$

when RSA reduces to the identity.) However, the RSA can still be decryptable for the numbers in the reduced residue system  $(\mathbb{Z}_n^\times)$ , since the mapping  $\mathbb{Z}_n^\times \rightarrow \mathbb{Z}_n^\times$  can still be bijective.

If you pass a non-prime integer to the arguments  $p_1, p_2, \dots, p_n$ , the particular number will be prime-factored and it will become either a multi-prime RSA or a multi-power RSA in its canonical form, depending on whether the product equals its radical or not.  $p_1 p_2 \dots p_n = \text{rad}(p_1 p_2 \dots p_n)$

**totient** : bool, optional

If 'Euler', it uses Euler's totient  $\phi(n)$  which is `sympy.ntheory.factor_.totient()` (page 1501) in SymPy.

If 'Carmichael', it uses Carmichael's totient  $\lambda(n)$  which is `sympy.ntheory.factor_.reduced_totient()` (page 1501) in SymPy.

Unlike private key generation, this is a trivial keyword for public key generation because  $\gcd(e, \phi(n)) = 1 \iff \gcd(e, \lambda(n)) = 1$ .

**index** : nonnegative integer, optional

Returns an arbitrary solution of a RSA public key at the index specified at 0, 1, 2, .... This parameter needs to be specified along with `totient='Carmichael'`.

Similarly to the non-uniqueness of a RSA private key as described in the `index` parameter documentation in `rsa_private_key()` (page 2782), RSA public key is also not unique and there is an infinite number of RSA public exponents which can behave in the same manner.

From any given RSA public exponent  $e$ , there can be another RSA public exponent  $e + k\lambda(n)$  where  $k$  is an integer,  $\lambda$  is a Carmichael's totient function.

However, considering only the positive cases, there can be a principal solution of a RSA public exponent  $e_0$  in  $0 < e_0 < \lambda(n)$ , and all the other solutions can be canonicalized in a form of  $e_0 + k\lambda(n)$ .

`index` specifies the  $k$  notation to yield any possible value an RSA public key can have.

An example of computing any arbitrary RSA public key:

```
>>> from sympy.crypto.crypto import rsa_public_key
>>> rsa_public_key(61, 53, 17, totient='Carmichael',
 ↪ index=0)
(3233, 17)
>>> rsa_public_key(61, 53, 17, totient='Carmichael',
 ↪ index=1)
(3233, 797)
>>> rsa_public_key(61, 53, 17, totient='Carmichael',
 ↪ index=2)
(3233, 1577)
```

**multipower** : bool, optional

Any pair of non-distinct primes found in the RSA specification will restrict the domain of the cryptosystem, as noted in the explanation of the parameter `args`.

SymPy RSA key generator may give a warning before dispatching it as a multi-power RSA, however, you can disable the warning if you pass `True` to this keyword.

## Returns

(**n**, **e**) : int, int

$n$  is a product of any arbitrary number of primes given as the argument.

$e$  is relatively prime (coprime) to the Euler totient  $\phi(n)$ .

False

Returned if less than two arguments are given, or  $e$  is not relatively prime to the modulus.

## Examples

```
>>> from sympy.crypto.crypto import rsa_public_key
```

A public key of a two-prime RSA:

```
>>> p, q, e = 3, 5, 7
>>> rsa_public_key(p, q, e)
(15, 7)
>>> rsa_public_key(p, q, 30)
False
```

A public key of a multiprime RSA:

```
>>> primes = [2, 3, 5, 7, 11, 13]
>>> e = 7
>>> args = primes + [e]
>>> rsa_public_key(*args)
(30030, 7)
```

## Notes

Although the RSA can be generalized over any modulus  $n$ , using two large primes had become the most popular specification because a product of two large primes is usually the hardest to factor relatively to the digits of  $n$  can have.

However, it may need further understanding of the time complexities of each prime-factoring algorithms to verify the claim.

### See also:

[rsa\\_private\\_key](#) (page 2782), [encipher\\_rsa](#) (page 2784), [decipher\\_rsa](#) (page 2785)

## References

[R148], [R149], [R150], [R151]

`sympy.crypto.crypto.rsa_private_key(*args, **kwargs)`

Return the RSA *private* key pair,  $(n, d)$

### Parameters

**args** : naturals

The keyword is identical to the args in [rsa\\_public\\_key\(\)](#) (page 2780).

**totient** : bool, optional

If 'Euler', it uses Euler's totient convention  $\phi(n)$  which is `sympy.ntheory.factor_.totient()` (page 1501) in SymPy.

If 'Carmichael', it uses Carmichael's totient convention  $\lambda(n)$  which is `sympy.ntheory.factor_.reduced_totient()` (page 1501) in SymPy.

There can be some output differences for private key generation as examples below.

Example using Euler's totient:

```
>>> from sympy.crypto.crypto import rsa_private_key
>>> rsa_private_key(61, 53, 17, totient='Euler')
(3233, 2753)
```

Example using Carmichael's totient:

```
>>> from sympy.crypto.crypto import rsa_private_key
>>> rsa_private_key(61, 53, 17, totient='Carmichael')
(3233, 413)
```

**index** : nonnegative integer, optional

Returns an arbitrary solution of a RSA private key at the index specified at 0,1,2,... This parameter needs to be specified along with `totient='Carmichael'`.

RSA private exponent is a non-unique solution of  $ed \bmod \lambda(n) = 1$  and it is possible in any form of  $d + k\lambda(n)$ , where  $d$  is an another already-computed private exponent, and  $\lambda$  is a Carmichael's totient function, and  $k$  is any integer.

However, considering only the positive cases, there can be a principal solution of a RSA private exponent  $d_0$  in  $0 < d_0 < \lambda(n)$ , and all the other solutions can be canonicalized in a form of  $d_0 + k\lambda(n)$ .

`index` specifies the  $k$  notation to yield any possible value an RSA private key can have.

An example of computing any arbitrary RSA private key:

```
>>> from sympy.crypto.crypto import rsa_private_key
>>> rsa_private_key(61, 53, 17, totient='Carmichael',
↪ index=0)
(3233, 413)
>>> rsa_private_key(61, 53, 17, totient='Carmichael',
↪ index=1)
(3233, 1193)
>>> rsa_private_key(61, 53, 17, totient='Carmichael',
↪ index=2)
(3233, 1973)
```

**multipower** : bool, optional

The keyword is identical to the `multipower` in `rsa_public_key()` (page 2780).

**Returns****(n, d)** : int, int

$n$  is a product of any arbitrary number of primes given as the argument.

$d$  is the inverse of  $e \pmod{\phi(n)}$  where  $e$  is the exponent given, and  $\phi$  is a Euler totient.

False

Returned if less than two arguments are given, or  $e$  is not relatively prime to the totient of the modulus.

**Examples**

```
>>> from sympy.crypto.crypto import rsa_private_key
```

A private key of a two-prime RSA:

```
>>> p, q, e = 3, 5, 7
>>> rsa_private_key(p, q, e)
(15, 7)
>>> rsa_private_key(p, q, 30)
False
```

A private key of a multiprime RSA:

```
>>> primes = [2, 3, 5, 7, 11, 13]
>>> e = 7
>>> args = primes + [e]
>>> rsa_private_key(*args)
(30030, 823)
```

**See also:**

[rsa\\_public\\_key](#) (page 2780), [encipher\\_rsa](#) (page 2784), [decipher\\_rsa](#) (page 2785)

**References**

[R152], [R153], [R154], [R155]

`sympy.crypto.crypto.encipher_rsa(i, key, factors=None)`

Encrypt the plaintext with RSA.

**Parameters****i** : integer

The plaintext to be encrypted for.

**key** : (n, e) where n, e are integers

$n$  is the modulus of the key and  $e$  is the exponent of the key. The encryption is computed by  $i^e \pmod n$ .

The key can either be a public key or a private key, however, the message encrypted by a public key can only be decrypted by a private key, and vice versa, as RSA is an asymmetric cryptography system.



**factors** : list of coprime integers

This is identical to the keyword `factors` in `decipher_rsa()` (page 2785).

## Notes

Some specifications may make the RSA not cryptographically meaningful.

For example, 0, 1 will remain always same after taking any number of exponentiation, thus, should be avoided.

Furthermore, if  $i^e < n$ ,  $i$  may easily be figured out by taking  $e$  th root.

And also, specifying the exponent as 1 or in more generalized form as  $1 + k\lambda(n)$  where  $k$  is an nonnegative integer,  $\lambda$  is a carmichael totient, the RSA becomes an identity mapping.

## Examples

```
>>> from sympy.crypto.crypto import encipher_rsa
>>> from sympy.crypto.crypto import rsa_public_key, rsa_private_key
```

Public Key Encryption:

```
>>> p, q, e = 3, 5, 7
>>> puk = rsa_public_key(p, q, e)
>>> msg = 12
>>> encipher_rsa(msg, puk)
3
```

Private Key Encryption:

```
>>> p, q, e = 3, 5, 7
>>> prk = rsa_private_key(p, q, e)
>>> msg = 12
>>> encipher_rsa(msg, prk)
3
```

Encryption using chinese remainder theorem:

```
>>> encipher_rsa(msg, prk, factors=[p, q])
3
```

`sympy.crypto.crypto.decipher_rsa(i, key, factors=None)`

Decrypt the ciphertext with RSA.

### Parameters

**i** : integer

The ciphertext to be decrypted for.

**key** : (*n*, *d*) where *n*, *d* are integers

*n* is the modulus of the key and *d* is the exponent of the key. The decryption is computed by  $i^d \bmod n$ .

The key can either be a public key or a private key, however, the message encrypted by a public key can only be decrypted by a private key, and vice versa, as RSA is an asymmetric cryptography system.

**factors** : list of coprime integers

As the modulus  $n$  created from RSA key generation is composed of arbitrary prime factors  $n = p_1^{k_1} p_2^{k_2} \dots p_n^{k_n}$  where  $p_1, p_2, \dots, p_n$  are distinct primes and  $k_1, k_2, \dots, k_n$  are positive integers, chinese remainder theorem can be used to compute  $i^d \bmod n$  from the fragmented modulo operations like

$$i^d \bmod p_1^{k_1}, i^d \bmod p_2^{k_2}, \dots, i^d \bmod p_n^{k_n}$$

or like

$$i^d \bmod p_1^{k_1} p_2^{k_2}, i^d \bmod p_3^{k_3}, \dots, i^d \bmod p_n^{k_n}$$

as long as every moduli does not share any common divisor each other.

The raw primes used in generating the RSA key pair can be a good option.

Note that the speed advantage of using this is only viable for very large cases (Like 2048-bit RSA keys) since the overhead of using pure Python implementation of [sympy.nttheory.modular.crt\(\)](#) (page 1509) may overcompensate the theoretical speed advantage.

## Notes

See the Notes section in the documentation of [encipher\\_rsa\(\)](#) (page 2784)

## Examples

```
>>> from sympy.crypto.crypto import decipher_rsa, encipher_rsa
>>> from sympy.crypto.crypto import rsa_public_key, rsa_private_key
```

Public Key Encryption and Decryption:

```
>>> p, q, e = 3, 5, 7
>>> prk = rsa_private_key(p, q, e)
>>> puk = rsa_public_key(p, q, e)
>>> msg = 12
>>> new_msg = encipher_rsa(msg, prk)
>>> new_msg
3
>>> decipher_rsa(new_msg, puk)
12
```

Private Key Encryption and Decryption:

```
>>> p, q, e = 3, 5, 7
>>> prk = rsa_private_key(p, q, e)
>>> puk = rsa_public_key(p, q, e)
>>> msg = 12
>>> new_msg = encipher_rsa(msg, puk)
>>> new_msg
3
>>> decipher_rsa(new_msg, prk)
12
```

Decryption using chinese remainder theorem:

```
>>> decipher_rsa(new_msg, prk, factors=[p, q])
12
```

**See also:**

[`encipher\_rsa`](#) (page 2784)

`sympy.crypto.crypto.kid_rsa_public_key(a, b, A, B)`

Kid RSA is a version of RSA useful to teach grade school children since it does not involve exponentiation.

### Explanation

Alice wants to talk to Bob. Bob generates keys as follows. Key generation:

- Select positive integers  $a, b, A, B$  at random.
- Compute  $M = ab - 1$ ,  $e = AM + a$ ,  $d = BM + b$ ,  $n = (ed - 1)/M$ .
- The *public key* is  $(n, e)$ . Bob sends these to Alice.
- The *private key* is  $(n, d)$ , which Bob keeps secret.

Encryption: If  $p$  is the plaintext message then the ciphertext is  $c = pe \pmod{n}$ .

Decryption: If  $c$  is the ciphertext message then the plaintext is  $p = cd \pmod{n}$ .

### Examples

```
>>> from sympy.crypto.crypto import kid_rsa_public_key
>>> a, b, A, B = 3, 4, 5, 6
>>> kid_rsa_public_key(a, b, A, B)
(369, 58)
```

`sympy.crypto.crypto.kid_rsa_private_key(a, b, A, B)`

Compute  $M = ab - 1$ ,  $e = AM + a$ ,  $d = BM + b$ ,  $n = (ed - 1)/M$ . The *private key* is  $d$ , which Bob keeps secret.

### Examples

```
>>> from sympy.crypto.crypto import kid_rsa_private_key
>>> a, b, A, B = 3, 4, 5, 6
>>> kid_rsa_private_key(a, b, A, B)
(369, 70)
```

`sympy.crypto.crypto.encipher_kid_rsa(msg, key)`  
Here msg is the plaintext and key is the public key.

### Examples

```
>>> from sympy.crypto.crypto import (
... encipher_kid_rsa, kid_rsa_public_key)
>>> msg = 200
>>> a, b, A, B = 3, 4, 5, 6
>>> key = kid_rsa_public_key(a, b, A, B)
>>> encipher_kid_rsa(msg, key)
161
```

`sympy.crypto.crypto.decipher_kid_rsa(msg, key)`  
Here msg is the plaintext and key is the private key.

### Examples

```
>>> from sympy.crypto.crypto import (
... kid_rsa_public_key, kid_rsa_private_key,
... decipher_kid_rsa, encipher_kid_rsa)
>>> a, b, A, B = 3, 4, 5, 6
>>> d = kid_rsa_private_key(a, b, A, B)
>>> msg = 200
>>> pub = kid_rsa_public_key(a, b, A, B)
>>> pri = kid_rsa_private_key(a, b, A, B)
>>> ct = encipher_kid_rsa(msg, pub)
>>> decipher_kid_rsa(ct, pri)
200
```

`sympy.crypto.crypto.encode_morse(msg, sep='|', mapping=None)`  
Encodes a plaintext into popular Morse Code with letters separated by sep and words by a double sep.

## Examples

```
>>> from sympy.crypto.crypto import encode_morse
>>> msg = 'ATTACK RIGHT FLANK'
>>> encode_morse(msg)
'.-|-|-|. -|. -..|. -..|.|..|..|...|-||...|. -..|. -|. -|. -..'
```

## References

[R156]

`sympy.crypto.crypto.decode_morse(msg, sep='|', mapping=None)`

Decodes a Morse Code with letters separated by `sep` (default is '|') and words by `word_sep` (default is '||') into plaintext.

## Examples

```
>>> from sympy.crypto.crypto import decode_morse
>>> mc = '-.-|---|...|.|||.|-|...|-'
>>> decode_morse(mc)
'MOVE EAST'
```

## References

[R157]

`sympy.crypto.crypto.lfsr_sequence(key, fill, n)`

This function creates an LFSR sequence.

### Parameters

**key** : list

A list of finite field elements,  $[c_0, c_1, \dots, c_k]$ .

**fill** : list

The list of the initial terms of the LFSR sequence,  $[x_0, x_1, \dots, x_k]$ .

**n**

Number of terms of the sequence that the function returns.

### Returns

L

The LFSR sequence defined by  $x_{n+1} = c_k x_n + \dots + c_0 x_{n-k}$ , for  $n \leq k$ .

## Notes

S. Golomb [G157] gives a list of three statistical properties a sequence of numbers  $a = \{a_n\}_{n=1}^{\infty}$ ,  $a_n \in \{0, 1\}$ , should display to be considered “random”. Define the autocorrelation of  $a$  to be

$$C(k) = C(k, a) = \lim_{N \rightarrow \infty} \frac{1}{N} \sum_{n=1}^N (-1)^{a_n + a_{n+k}}.$$

In the case where  $a$  is periodic with period  $P$  then this reduces to

$$C(k) = \frac{1}{P} \sum_{n=1}^P (-1)^{a_n + a_{n+k}}.$$

Assume  $a$  is periodic with period  $P$ .

- balance:

$$\left| \sum_{n=1}^P (-1)^{a_n} \right| \leq 1.$$

- low autocorrelation:

$$C(k) = \begin{cases} 1, & k = 0, \\ \epsilon, & k \neq 0. \end{cases}$$

(For sequences satisfying these first two properties, it is known that  $\epsilon = -1/P$  must hold.)

- proportional runs property: In each period, half the runs have length 1, one-fourth have length 2, etc. Moreover, there are as many runs of 1’s as there are of 0’s.

## Examples

```
>>> from sympy.crypto.crypto import lfsr_sequence
>>> from sympy.polys.domains import FF
>>> F = FF(2)
>>> fill = [F(1), F(1), F(0), F(1)]
>>> key = [F(1), F(0), F(0), F(1)]
>>> lfsr_sequence(key, fill, 10)
[1 mod 2, 1 mod 2, 0 mod 2, 1 mod 2, 0 mod 2,
1 mod 2, 1 mod 2, 0 mod 2, 0 mod 2, 1 mod 2]
```

## References

[G157]

`sympy.crypto.crypto.lfsr_autocorrelation(L, P, k)`

This function computes the LFSR autocorrelation function.

**Parameters****L**

A periodic sequence of elements of  $GF(2)$ . L must have length larger than P.

**P**

The period of L.

**k** : int

An integer  $k$  ( $0 < k < P$ ).

**Returns**

autocorrelation

The k-th value of the autocorrelation of the LFSR L.

**Examples**

```
>>> from sympy.crypto.crypto import (
... lfsr_sequence, lfsr_autocorrelation)
>>> from sympy.polys.domains import FF
>>> F = FF(2)
>>> fill = [F(1), F(1), F(0), F(1)]
>>> key = [F(1), F(0), F(0), F(1)]
>>> s = lfsr_sequence(key, fill, 20)
>>> lfsr_autocorrelation(s, 15, 7)
-1/15
>>> lfsr_autocorrelation(s, 15, 0)
1
```

`sympy.crypto.crypto.lfsr_connection_polynomial(s)`

This function computes the LFSR connection polynomial.

**Parameters****s**

A sequence of elements of even length, with entries in a finite field.

**Returns**

$C(x)$

The connection polynomial of a minimal LFSR yielding s.

This implements the algorithm in section 3 of J. L. Massey's article [\[M158\]](#).

## Examples

```
>>> from sympy.crypto.crypto import (
... lfsr_sequence, lfsr_connection_polynomial)
>>> from sympy.polys.domains import FF
>>> F = FF(2)
>>> fill = [F(1), F(1), F(0), F(1)]
>>> key = [F(1), F(0), F(0), F(1)]
>>> s = lfsr_sequence(key, fill, 20)
>>> lfsr_connection_polynomial(s)
x**4 + x + 1
>>> fill = [F(1), F(0), F(0), F(1)]
>>> key = [F(1), F(1), F(0), F(1)]
>>> s = lfsr_sequence(key, fill, 20)
>>> lfsr_connection_polynomial(s)
x**3 + 1
>>> fill = [F(1), F(0), F(1)]
>>> key = [F(1), F(1), F(0)]
>>> s = lfsr_sequence(key, fill, 20)
>>> lfsr_connection_polynomial(s)
x**3 + x**2 + 1
>>> fill = [F(1), F(0), F(1)]
>>> key = [F(1), F(0), F(1)]
>>> s = lfsr_sequence(key, fill, 20)
>>> lfsr_connection_polynomial(s)
x**3 + x + 1
```

## References

[M158]

`sympy.crypto.crypto.elgamal_public_key(key)`

Return three number tuple as public key.

### Parameters

**key** : (p, r, e)

Tuple generated by `elgamal_private_key`.

### Returns

**tuple** : (p, r, e)

$e = r ** d \bmod p$

$d$  is a random number in private key.



## Examples

```
>>> from sympy.crypto.crypto import elgamal_public_key
>>> elgamal_public_key((1031, 14, 636))
(1031, 14, 212)
```

`sympy.crypto.crypto.elgamal_private_key(digit=10, seed=None)`

Return three number tuple as private key.

### Parameters

**digit** : int

Minimum number of binary digits for key.

### Returns

**tuple** : (p, r, d)

p = prime number.

r = primitive root.

d = random number.

## Explanation

Elgamal encryption is based on the mathematical problem called the Discrete Logarithm Problem (DLP). For example,

$$a^b \equiv c \pmod{p}$$

In general, if a and b are known, ct is easily calculated. If b is unknown, it is hard to use a and ct to get b.

## Notes

For testing purposes, the seed parameter may be set to control the output of this routine. See `sympy.core.random._randrange`.

## Examples

```
>>> from sympy.crypto.crypto import elgamal_private_key
>>> from sympy.ntheory import is_primitive_root, isprime
>>> a, b, _ = elgamal_private_key()
>>> isprime(a)
True
>>> is_primitive_root(b, a)
True
```

`sympy.crypto.crypto.encipher_elgamal(i, key, seed=None)`

Encrypt message with public key.

### Parameters

**msg**

int of encoded message.

**key**

Public key.

**Returns**

**tuple** : (c1, c2)

Encipher into two number.

**Explanation**

i is a plaintext message expressed as an integer. key is public key (p, r, e). In order to encrypt a message, a random number a in `range(2, p)` is generated and the encrypted message is returned as  $c_1$  and  $c_2$  where:

$$c_1 \equiv r^a \pmod{p}$$

$$c_2 \equiv me^a \pmod{p}$$

**Notes**

For testing purposes, the seed parameter may be set to control the output of this routine. See `sympy.core.random._randrange`.

**Examples**

```
>>> from sympy.crypto.crypto import encipher_elgamal, elgamal_private_
 ↪key, elgamal_public_key
>>> pri = elgamal_private_key(5, seed=[3]); pri
(37, 2, 3)
>>> pub = elgamal_public_key(pri); pub
(37, 2, 8)
>>> msg = 36
>>> encipher_elgamal(msg, pub, seed=[3])
(8, 6)
```

`sympy.crypto.crypto.decipher_elgamal(msg, key)`

Decrypt message with private key.

$$msg = (c_1, c_2)$$

$$key = (p, r, d)$$

According to extended Eucliden theorem,  $uc_1^d + pn = 1$

$$u \equiv 1/c_1^d \pmod{p}$$

$$uc_2 \equiv \frac{1}{c_1^d} c_2 \equiv \frac{1}{r^{ad}} c_2 \pmod{p}$$

$$\frac{1}{r^{ad}} me^a \equiv \frac{1}{r^{ad}} mr^{da} \equiv m \pmod{p}$$

## Examples

```
>>> from sympy.crypto.crypto import decipher_elgamal
>>> from sympy.crypto.crypto import encipher_elgamal
>>> from sympy.crypto.crypto import elgamal_private_key
>>> from sympy.crypto.crypto import elgamal_public_key

>>> pri = elgamal_private_key(5, seed=[3])
>>> pub = elgamal_public_key(pri); pub
(37, 2, 8)
>>> msg = 17
>>> decipher_elgamal(encipher_elgamal(msg, pub), pri) == msg
True
```

`sympy.crypto.crypto.dh_public_key(key)`

Return three number tuple as public key.

This is the tuple that Alice sends to Bob.

### Parameters

**key** : (p, g, a)

A tuple generated by `dh_private_key`.

### Returns

**tuple** : int, int, int

A tuple of  $(p, g, g^a \bmod p)$  with  $p$ ,  $g$  and  $a$  given as parameters.s

## Examples

```
>>> from sympy.crypto.crypto import dh_private_key, dh_public_key
>>> p, g, a = dh_private_key();
>>> _p, _g, x = dh_public_key((p, g, a))
>>> p == _p and g == _g
True
>>> x == pow(g, a, p)
True
```

`sympy.crypto.crypto.dh_private_key(digit=10, seed=None)`

Return three integer tuple as private key.

### Parameters

**digit**

Minimum number of binary digits required in key.

### Returns

**tuple** : (p, g, a)

p = prime number.

g = primitive root of p.

a = random number from 2 through p - 1.

## Explanation

Diffie-Hellman key exchange is based on the mathematical problem called the Discrete Logarithm Problem (see ElGamal).

Diffie-Hellman key exchange is divided into the following steps:

- Alice and Bob agree on a base that consist of a prime  $p$  and a primitive root of  $p$  called  $g$
- Alice choses a number  $a$  and Bob choses a number  $b$  where  $a$  and  $b$  are random numbers in range  $[2, p)$ . These are their private keys.
- Alice then publicly sends Bob  $g^a \pmod{p}$  while Bob sends Alice  $g^b \pmod{p}$
- They both raise the received value to their secretly chosen number ( $a$  or  $b$ ) and now have both as their shared key  $g^{ab} \pmod{p}$

## Notes

For testing purposes, the seed parameter may be set to control the output of this routine. See `sympy.core.random._randrange`.

## Examples

```
>>> from sympy.crypto.crypto import dh_private_key
>>> from sympy.ntheory import isprime, is_primitive_root
>>> p, g, _ = dh_private_key()
>>> isprime(p)
True
>>> is_primitive_root(g, p)
True
>>> p, g, _ = dh_private_key(5)
>>> isprime(p)
True
>>> is_primitive_root(g, p)
True
```

`sympy.crypto.crypto.dh_shared_key(key, b)`

Return an integer that is the shared key.

This is what Bob and Alice can both calculate using the public keys they received from each other and their private keys.

### Parameters

**key** :  $(p, g, x)$

Tuple  $(p, g, x)$  generated by `dh_public_key`.

**b**

Random number in the range of 2 to  $p - 1$  (Chosen by second key exchange member (Bob)).

### Returns

int

A shared key.

## Examples

```
>>> from sympy.crypto.crypto import (
... dh_private_key, dh_public_key, dh_shared_key)
>>> prk = dh_private_key();
>>> p, g, x = dh_public_key(prk);
>>> sk = dh_shared_key((p, g, x), 1000)
>>> sk == pow(x, 1000, p)
True
```

`sympy.crypto.crypto.gm_public_key(p, q, a=None, seed=None)`

Compute public keys for p and q. Note that in Goldwasser-Micali Encryption, public keys are randomly selected.

### Parameters

**p, q, a** : int, int, int

Initialization variables.

### Returns

**tuple** : (a, N)

a is the input a if it is not None otherwise some random integer co-prime to p and q.

N is the product of p and q.

`sympy.crypto.crypto.gm_private_key(p, q, a=None)`

Check if p and q can be used as private keys for the Goldwasser-Micali encryption. The method works roughly as follows.

### Parameters

**p, q, a**

Initialization variables.

### Returns

**tuple** : (p, q)

The input value p and q.

### Raises

#### ValueError

If p and q are not distinct odd primes.

**Explanation**

1. Pick two large primes  $p$  and  $q$ .
2. Call their product  $N$ .
3. Given a message as an integer  $i$ , write  $i$  in its bit representation  $b_0, \dots, b_n$ .
4. For each  $k$ ,  
  
    **if**  $b_k = 0$ :  
        let  $a_k$  be a random square (quadratic residue) modulo  $pq$  such that  
        `jacobi_symbol(a, p*q) = 1`  
  
    **if**  $b_k = 1$ :  
        let  $a_k$  be a random non-square (non-quadratic residue) modulo  $pq$  such that  
        `jacobi_symbol(a, p*q) = -1`

returns  $[a_1, a_2, \dots]$

$b_k$  can be recovered by checking whether or not  $a_k$  is a residue. And from the  $b_k$ 's, the message can be reconstructed.

The idea is that, while `jacobi_symbol(a, p*q)` can be easily computed (and when it is equal to  $-1$  will tell you that  $a$  is not a square mod  $pq$ ), quadratic residuosity modulo a composite number is hard to compute without knowing its factorization.

Moreover, approximately half the numbers coprime to  $pq$  have `jacobi_symbol()` (page 1522) equal to 1. And among those, approximately half are residues and approximately half are not. This maximizes the entropy of the code.

`sympy.crypto.crypto.encipher_gm(i, key, seed=None)`

Encrypt integer 'i' using public\_key 'key' Note that gm uses random encryption.

**Parameters**

**i** : int

The message to encrypt.

**key** : (a, N)

The public key.

**Returns**

**list** : list of int

The randomized encrypted message.

`sympy.crypto.crypto.decipher_gm(message, key)`

Decrypt message 'message' using public\_key 'key'.

**Parameters**

**message** : list of int

The randomized encrypted message.

**key** : (p, q)

The private key.

**Returns**

int

The encrypted message.

`sympy.crypto.crypto.encipher_railfence(message, rails)`

Performs Railfence Encryption on plaintext and returns ciphertext

**Parameters**

**message** : string, the message to encrypt.

**rails** : int, the number of rails.

**Returns**

The Encrypted string message.

**Examples**

```
>>> from sympy.crypto.crypto import encipher_railfence
>>> message = "hello world"
>>> encipher_railfence(message,3)
'horel ollwd'
```

**References**

[R160]

`sympy.crypto.crypto.decipher_railfence(ciphertext, rails)`

Decrypt the message using the given rails

**Parameters**

**message** : string, the message to encrypt.

**rails** : int, the number of rails.

**Returns**

The Decrypted string message.

**Examples**

```
>>> from sympy.crypto.crypto import decipher_railfence
>>> decipher_railfence("horel ollwd",3)
'hello world'
```

## Differential Geometry

### Introduction

### Base Class Reference

`class sympy.diffgeom.Manifold(name, dim, **kwargs)`

A mathematical manifold.

**Parameters**

**name** : str

The name of the manifold.

**dim** : int

The dimension of the manifold.

### Explanation

A manifold is a topological space that locally resembles Euclidean space near each point [1]. This class does not provide any means to study the topological characteristics of the manifold that it represents, though.

### Examples

```
>>> from sympy.diffgeom import Manifold
>>> m = Manifold('M', 2)
>>> m
M
>>> m.dim
2
```

### References

[R161]

**class** sympy.diffgeom.**Patch**(*name*, *manifold*, *\*\*kwargs*)

A patch on a manifold.

#### Parameters

**name** : str

The name of the patch.

**manifold** : Manifold

The manifold on which the patch is defined.

### Explanation

Coordinate patch, or patch in short, is a simply-connected open set around a point in the manifold [1]. On a manifold one can have many patches that do not always include the whole manifold. On these patches coordinate charts can be defined that permit the parameterization of any point on the patch in terms of a tuple of real numbers (the coordinates).

This class does not provide any means to study the topological characteristics of the patch that it represents.



## Examples

```
>>> from sympy.diffgeom import Manifold, Patch
>>> m = Manifold('M', 2)
>>> p = Patch('P', m)
>>> p
P
>>> p.dim
2
```

## References

[R162]

```
class sympy.diffgeom.CoordSystem(name, patch, symbols=None, relations={},
 **kwargs)
```

A coordinate system defined on the patch.

### Parameters

**name** : str

The name of the coordinate system.

**patch** : Patch

The patch where the coordinate system is defined.

**symbols** : list of Symbols, optional

Defines the names and assumptions of coordinate symbols.

**relations** : dict, optional

Key is a tuple of two strings, who are the names of the systems where the coordinates transform from and transform to. Value is a tuple of the symbols before transformation and a tuple of the expressions after transformation.

## Explanation

Coordinate system is a system that uses one or more coordinates to uniquely determine the position of the points or other geometric elements on a manifold [1].

By passing Symbols to *symbols* parameter, user can define the name and assumptions of coordinate symbols of the coordinate system. If not passed, these symbols are generated automatically and are assumed to be real valued.

By passing *relations* parameter, user can define the tranform relations of coordinate systems. Inverse transformation and indirect transformation can be found automatically. If this parameter is not passed, coordinate transformation cannot be done.

## Examples

We define two-dimensional Cartesian coordinate system and polar coordinate system.

```
>>> from sympy import symbols, pi, sqrt, atan2, cos, sin
>>> from sympy.diffgeom import Manifold, Patch, CoordSystem
>>> m = Manifold('M', 2)
>>> p = Patch('P', m)
>>> x, y = symbols('x y', real=True)
>>> r, theta = symbols('r theta', nonnegative=True)
>>> relation_dict = {
... ('Car2D', 'Pol'): [(x, y), (sqrt(x**2 + y**2), atan2(y, x))],
... ('Pol', 'Car2D'): [(r, theta), (r*cos(theta), r*sin(theta))]
... }
>>> Car2D = CoordSystem('Car2D', p, (x, y), relation_dict)
>>> Pol = CoordSystem('Pol', p, (r, theta), relation_dict)
```

`symbols` property returns `CoordinateSymbol` instances. These symbols are not same with the symbols used to construct the coordinate system.

```
>>> Car2D
Car2D
>>> Car2D.dim
2
>>> Car2D.symbols
(x, y)
>>> _[0].func
<class 'sympy.diffgeom.diffgeom.CoordinateSymbol'>
```

`transformation()` method returns the transformation function from one coordinate system to another. `transform()` method returns the transformed coordinates.

```
>>> Car2D.transformation(Pol)
Lambda((x, y), Matrix([
[sqrt(x**2 + y**2)],
[atan2(y, x)]]))
>>> Car2D.transform(Pol)
Matrix([
[sqrt(x**2 + y**2)],
[atan2(y, x)]])
>>> Car2D.transform(Pol, [1, 2])
Matrix([
[sqrt(5)],
[atan(2)]])
```

`jacobian()` method returns the Jacobian matrix of coordinate transformation between two systems. `jacobian_determinant()` method returns the Jacobian determinant of coordinate transformation between two systems.

```
>>> Pol.jacobian(Car2D)
Matrix([
[cos(theta), -r*sin(theta)],
[sin(theta), r*cos(theta)]]
>>> Pol.jacobian(Car2D, [1, pi/2])
```

(continues on next page)

(continued from previous page)

```

Matrix([
[0, -1],
[1, 0]])
>>> Car2D.jacobian_determinant(Pol)
1/sqrt(x**2 + y**2)
>>> Car2D.jacobian_determinant(Pol, [1,0])
1

```

## References

[R163]

**base\_oneform**(*coord\_index*)

Return a basis 1-form field. The basis one-form field for this coordinate system. It is also an operator on vector fields.

**base\_oneforms**()

Returns a list of all base oneforms. For more details see the `base_oneform` method of this class.

**base\_scalar**(*coord\_index*)

Return `BaseScalarField` that takes a point and returns one of the coordinates.

**base\_scalars**()

Returns a list of all coordinate functions. For more details see the `base_scalar` method of this class.

**base\_vector**(*coord\_index*)

Return a basis vector field. The basis vector field for this coordinate system. It is also an operator on scalar fields.

**base\_vectors**()

Returns a list of all base vectors. For more details see the `base_vector` method of this class.

**coord\_function**(*coord\_index*)

Return `BaseScalarField` that takes a point and returns one of the coordinates.

**coord\_functions**()

Returns a list of all coordinate functions. For more details see the `base_scalar` method of this class.

**coord\_tuple\_transform\_to**(*to\_sys, coords*)

Transform *coords* to coord system *to\_sys*.

**jacobian**(*sys, coordinates=None*)

Return the jacobian matrix of a transformation on given coordinates. If coordinates are not given, coordinate symbols of *self* are used.

### Parameters

**sys** : `CoordSystem`

**coordinates** : Any iterable, optional.

### Returns

`sympy.ImmutableDenseMatrix`

## Examples

```
>>> from sympy.diffgeom.rn import R2_r, R2_p
>>> R2_p.jacobian(R2_r)
Matrix([
[cos(theta), -rho*sin(theta)],
[sin(theta), rho*cos(theta)]]
>>> R2_p.jacobian(R2_r, [1, 0])
Matrix([
[1, 0],
[0, 1]])
```

**jacobian\_determinant**(*sys*, *coordinates=None*)

Return the jacobian determinant of a transformation on given coordinates. If coordinates are not given, coordinate symbols of *self* are used.

### Parameters

**sys** : CoordSystem

**coordinates** : Any iterable, optional.

### Returns

sympy.Expr

## Examples

```
>>> from sympy.diffgeom.rn import R2_r, R2_p
>>> R2_r.jacobian_determinant(R2_p)
1/sqrt(x**2 + y**2)
>>> R2_r.jacobian_determinant(R2_p, [1, 0])
1
```

**jacobian\_matrix**(*sys*, *coordinates=None*)

Return the jacobian matrix of a transformation on given coordinates. If coordinates are not given, coordinate symbols of *self* are used.

### Parameters

**sys** : CoordSystem

**coordinates** : Any iterable, optional.

### Returns

sympy.ImmutableDenseMatrix

## Examples

```
>>> from sympy.diffgeom.rn import R2_r, R2_p
>>> R2_p.jacobian(R2_r)
Matrix([
[cos(theta), -rho*sin(theta)],
[sin(theta), rho*cos(theta)]]
>>> R2_p.jacobian(R2_r, [1, 0])
```

(continues on next page)

(continued from previous page)

```
Matrix([
[1, 0],
[0, 1]])
```

**point**(*coords*)

Create a Point with coordinates given in this coord system.

**point\_to\_coords**(*point*)

Calculate the coordinates of a point in this coord system.

**transform**(*sys*, *coordinates=None*)

Return the result of coordinate transformation from *self* to *sys*. If coordinates are not given, coordinate symbols of *self* are used.

**Parameters**

**sys** : CoordSystem

**coordinates** : Any iterable, optional.

**Returns**

sympy.ImmutableDenseMatrix containing CoordinateSymbol

### Examples

```
>>> from sympy.diffgeom.rn import R2_r, R2_p
>>> R2_r.transform(R2_p)
Matrix([
[sqrt(x**2 + y**2)],
[atan2(y, x)]]))
>>> R2_r.transform(R2_p, [0, 1])
Matrix([
[1],
[pi/2]])
```

**transformation**(*sys*)

Return coordinate transformation function from *self* to *sys*.

**Parameters**

**sys** : CoordSystem

**Returns**

sympy.Lambda

### Examples

```
>>> from sympy.diffgeom.rn import R2_r, R2_p
>>> R2_r.transformation(R2_p)
Lambda((x, y), Matrix([
[sqrt(x**2 + y**2)],
[atan2(y, x)]]))
```

**class** sympy.diffgeom.CoordinateSymbol(*coord\_sys*, *index*, *\*\*assumptions*)

A symbol which denotes an abstract value of *i*-th coordinate of the coordinate system with given context.

**Parameters**

**coord\_sys** : CoordSystem

**index** : integer

**Explanation**

Each coordinates in coordinate system are represented by unique symbol, such as *x*, *y*, *z* in Cartesian coordinate system.

You may not construct this class directly. Instead, use *symbols* method of CoordSystem.

**Examples**

```
>>> from sympy import symbols, Lambda, Matrix, sqrt, atan2, cos, sin
>>> from sympy.diffgeom import Manifold, Patch, CoordSystem
>>> m = Manifold('M', 2)
>>> p = Patch('P', m)
>>> x, y = symbols('x y', real=True)
>>> r, theta = symbols('r theta', nonnegative=True)
>>> relation_dict = {
... ('Car2D', 'Pol'): Lambda((x, y), Matrix([sqrt(x**2 + y**2), atan2(y,
↳ x)])),
... ('Pol', 'Car2D'): Lambda((r, theta), Matrix([r*cos(theta),
↳ r*sin(theta)]))
... }
>>> Car2D = CoordSystem('Car2D', p, [x, y], relation_dict)
>>> Pol = CoordSystem('Pol', p, [r, theta], relation_dict)
>>> x, y = Car2D.symbols
```

CoordinateSymbol contains its coordinate symbol and index.

```
>>> x.name
'x'
>>> x.coord_sys == Car2D
True
>>> x.index
0
>>> x.is_real
True
```

You can transform CoordinateSymbol into other coordinate system using *rewrite()* method.

```
>>> x.rewrite(Pol)
r*cos(theta)
>>> sqrt(x**2 + y**2).rewrite(Pol).simplify()
r
```

**class** sympy.diffgeom.**Point**(*coord\_sys*, *coords*, **\*\*kwargs**)

Point defined in a coordinate system.

**Parameters**

**coord\_sys** : CoordSystem

**coords** : list

The coordinates of the point.

## Explanation

Mathematically, point is defined in the manifold and does not have any coordinates by itself. Coordinate system is what imbues the coordinates to the point by coordinate chart. However, due to the difficulty of realizing such logic, you must supply a coordinate system and coordinates to define a Point here.

The usage of this object after its definition is independent of the coordinate system that was used in order to define it, however due to limitations in the simplification routines you can arrive at complicated expressions if you use inappropriate coordinate systems.

## Examples

```
>>> from sympy import pi
>>> from sympy.diffgeom import Point
>>> from sympy.diffgeom.rn import R2, R2_r, R2_p
>>> rho, theta = R2_p.symbols
```

```
>>> p = Point(R2_p, [rho, 3*pi/4])
```

```
>>> p.manifold == R2
True
```

```
>>> p.coords()
Matrix([
[rho],
[3*pi/4]])
>>> p.coords(R2_r)
Matrix([
[-sqrt(2)*rho/2],
[sqrt(2)*rho/2]])
```

**coords**(*sys=None*)

Coordinates of the point in given coordinate system. If coordinate system is not passed, it returns the coordinates in the coordinate system in which the poin was defined.

**class** sympy.diffgeom.**BaseScalarField**(*coord\_sys*, *index*, **\*\*kwargs**)

Base scalar field over a manifold for a given coordinate system.

**Parameters**

**coord\_sys** : CoordSystem

**index** : integer

## Explanation

A scalar field takes a point as an argument and returns a scalar. A base scalar field of a coordinate system takes a point and returns one of the coordinates of that point in the coordinate system in question.

To define a scalar field you need to choose the coordinate system and the index of the coordinate.

The use of the scalar field after its definition is independent of the coordinate system in which it was defined, however due to limitations in the simplification routines you may arrive at more complicated expression if you use unappropriate coordinate systems. You can build complicated scalar fields by just building up SymPy expressions containing BaseScalarField instances.

## Examples

```
>>> from sympy import Function, pi
>>> from sympy.diffgeom import BaseScalarField
>>> from sympy.diffgeom.rn import R2_r, R2_p
>>> rho, _ = R2_p.symbols
>>> point = R2_p.point([rho, 0])
>>> fx, fy = R2_r.base_scalars()
>>> ftheta = BaseScalarField(R2_r, 1)
```

```
>>> fx(point)
rho
>>> fy(point)
0
```

```
>>> (fx**2+fy**2).rcall(point)
rho**2
```

```
>>> g = Function('g')
>>> fg = g(ftheta-pi)
>>> fg.rcall(point)
g(-pi)
```

**class** sympy.diffgeom.BaseVectorField(*coord\_sys*, *index*, *\*\*kwargs*)

Base vector field over a manifold for a given coordinate system.

### Parameters

**coord\_sys** : CoordSystem

**index** : integer



## Explanation

A vector field is an operator taking a scalar field and returning a directional derivative (which is also a scalar field). A base vector field is the same type of operator, however the derivation is specifically done with respect to a chosen coordinate.

To define a base vector field you need to choose the coordinate system and the index of the coordinate.

The use of the vector field after its definition is independent of the coordinate system in which it was defined, however due to limitations in the simplification routines you may arrive at more complicated expression if you use inappropriate coordinate systems.

## Examples

```
>>> from sympy import Function
>>> from sympy.diffgeom.rn import R2_p, R2_r
>>> from sympy.diffgeom import BaseVectorField
>>> from sympy import pprint
```

```
>>> x, y = R2_r.symbols
>>> rho, theta = R2_p.symbols
>>> fx, fy = R2_r.base_scalars()
>>> point_p = R2_p.point([rho, theta])
>>> point_r = R2_r.point([x, y])
```

```
>>> g = Function('g')
>>> s_field = g(fx, fy)
```

```
>>> v = BaseVectorField(R2_r, 1)
>>> pprint(v(s_field))
/ d \
|---(g(x, xi))|
\dx /xi=y
>>> pprint(v(s_field).rcall(point_r).doit())
d
--(g(x, y))
dy
>>> pprint(v(s_field).rcall(point_p))
/ d \
|---(g(rho*cos(theta), xi))|
\dx /xi=rho*sin(theta)
```

**class** sympy.diffgeom.Commutator(v1, v2)  
Commutator of two vector fields.

### Explanation

The commutator of two vector fields  $v_1$  and  $v_2$  is defined as the vector field  $[v_1, v_2]$  that evaluated on each scalar field  $f$  is equal to  $v_1(v_2(f)) - v_2(v_1(f))$ .

### Examples

```
>>> from sympy.diffgeom.rn import R2_p, R2_r
>>> from sympy.diffgeom import Commutator
>>> from sympy import simplify
```

```
>>> fx, fy = R2_r.base_scalars()
>>> e_x, e_y = R2_r.base_vectors()
>>> e_r = R2_p.base_vector(0)
```

```
>>> c_xy = Commutator(e_x, e_y)
>>> c_xr = Commutator(e_x, e_r)
>>> c_xy
0
```

Unfortunately, the current code is not able to compute everything:

```
>>> c_xr
Commutator(e_x, e_rho)
>>> simplify(c_xr(fy**2))
-2*cos(theta)*y**2/(x**2 + y**2)
```

**class** sympy.diffgeom.Differential(*form\_field*)

Return the differential (exterior derivative) of a form field.

### Explanation

The differential of a form (i.e. the exterior derivative) has a complicated definition in the general case. The differential  $df$  of the 0-form  $f$  is defined for any vector field  $v$  as  $df(v) = v(f)$ .

### Examples

```
>>> from sympy import Function
>>> from sympy.diffgeom.rn import R2_r
>>> from sympy.diffgeom import Differential
>>> from sympy import pprint
```

```
>>> fx, fy = R2_r.base_scalars()
>>> e_x, e_y = R2_r.base_vectors()
>>> g = Function('g')
>>> s_field = g(fx, fy)
>>> dg = Differential(s_field)
```

```

>>> dg
d(g(x, y))
>>> pprint(dg(e_x))
/ d \
|---(g(xi, y))|
\dx /|xi=x
>>> pprint(dg(e_y))
/ d \
|---(g(x, xi))|
\dx /|xi=y

```

Applying the exterior derivative operator twice always results in:

```

>>> Differential(dg)
0

```

**class** `sympy.diffgeom.TensorProduct(*args)`

Tensor product of forms.

### Explanation

The tensor product permits the creation of multilinear functionals (i.e. higher order tensors) out of lower order fields (e.g. 1-forms and vector fields). However, the higher tensors thus created lack the interesting features provided by the other type of product, the wedge product, namely they are not antisymmetric and hence are not form fields.

### Examples

```

>>> from sympy.diffgeom.rn import R2_r
>>> from sympy.diffgeom import TensorProduct

```

```

>>> fx, fy = R2_r.base_scalars()
>>> e_x, e_y = R2_r.base_vectors()
>>> dx, dy = R2_r.base_oneforms()

```

```

>>> TensorProduct(dx, dy)(e_x, e_y)
1
>>> TensorProduct(dx, dy)(e_y, e_x)
0
>>> TensorProduct(dx, fx*dy)(fx*e_x, e_y)
x**2
>>> TensorProduct(e_x, e_y)(fx**2, fy**2)
4*x*y
>>> TensorProduct(e_y, dx)(fy)
dx

```

You can nest tensor products.

```
>>> tp1 = TensorProduct(dx, dy)
>>> TensorProduct(tp1, dx)(e_x, e_y, e_x)
1
```

You can make partial contraction for instance when ‘raising an index’. Putting `None` in the second argument of `rcall` means that the respective position in the tensor product is left as it is.

```
>>> TP = TensorProduct
>>> metric = TP(dx, dx) + 3*TP(dy, dy)
>>> metric.rcall(e_y, None)
3*dy
```

Or automatically pad the args with `None` without specifying them.

```
>>> metric.rcall(e_y)
3*dy
```

**class** `sympy.diffgeom.WedgeProduct(*args)`

Wedge product of forms.

### Explanation

In the context of integration only completely antisymmetric forms make sense. The wedge product permits the creation of such forms.

### Examples

```
>>> from sympy.diffgeom.rn import R2_r
>>> from sympy.diffgeom import WedgeProduct
```

```
>>> fx, fy = R2_r.base_scalars()
>>> e_x, e_y = R2_r.base_vectors()
>>> dx, dy = R2_r.base_oneforms()
```

```
>>> WedgeProduct(dx, dy)(e_x, e_y)
1
>>> WedgeProduct(dx, dy)(e_y, e_x)
-1
>>> WedgeProduct(dx, fx*dy)(fx*e_x, e_y)
x**2
>>> WedgeProduct(e_x, e_y)(fy, None)
-e_x
```

You can nest wedge products.

```
>>> wp1 = WedgeProduct(dx, dy)
>>> WedgeProduct(wp1, dx)(e_x, e_y, e_x)
0
```

**class** `sympy.diffgeom.LieDerivative(v_field, expr)`  
 Lie derivative with respect to a vector field.

### Explanation

The transport operator that defines the Lie derivative is the pushforward of the field to be derived along the integral curve of the field with respect to which one derives.

### Examples

```
>>> from sympy.diffgeom.rn import R2_r, R2_p
>>> from sympy.diffgeom import (LieDerivative, TensorProduct)
```

```
>>> fx, fy = R2_r.base_scalars()
>>> e_x, e_y = R2_r.base_vectors()
>>> e_rho, e_theta = R2_p.base_vectors()
>>> dx, dy = R2_r.base_oneforms()
```

```
>>> LieDerivative(e_x, fy)
0
>>> LieDerivative(e_x, fx)
1
>>> LieDerivative(e_x, e_x)
0
```

The Lie derivative of a tensor field by another tensor field is equal to their commutator:

```
>>> LieDerivative(e_x, e_rho)
Commutator(e_x, e_rho)
>>> LieDerivative(e_x + e_y, fx)
1
```

```
>>> tp = TensorProduct(dx, dy)
>>> LieDerivative(e_x, tp)
LieDerivative(e_x, TensorProduct(dx, dy))
>>> LieDerivative(e_x, tp)
LieDerivative(e_x, TensorProduct(dx, dy))
```

**class** `sympy.diffgeom.BaseCovarDerivativeOp(coord_sys, index, christoffel)`  
 Covariant derivative operator with respect to a base vector.

### Examples

```
>>> from sympy.diffgeom.rn import R2_r
>>> from sympy.diffgeom import BaseCovarDerivativeOp
>>> from sympy.diffgeom import metric_to_Christoffel_2nd, TensorProduct
```

```
>>> TP = TensorProduct
>>> fx, fy = R2_r.base_scalars()
>>> e_x, e_y = R2_r.base_vectors()
>>> dx, dy = R2_r.base_oneforms()
```

```
>>> ch = metric_to_Christoffel_2nd(TP(dx, dx) + TP(dy, dy))
>>> ch
[[[0, 0], [0, 0]], [[0, 0], [0, 0]]]
>>> cvd = BaseCovarDerivativeOp(R2_r, 0, ch)
>>> cvd(fx)
1
>>> cvd(fx*e_x)
e_x
```

**class** sympy.diffgeom.CovarDerivativeOp(*wrt, christoffel*)  
Covariant derivative operator.

### Examples

```
>>> from sympy.diffgeom.rn import R2_r
>>> from sympy.diffgeom import CovarDerivativeOp
>>> from sympy.diffgeom import metric_to_Christoffel_2nd, TensorProduct
>>> TP = TensorProduct
>>> fx, fy = R2_r.base_scalars()
>>> e_x, e_y = R2_r.base_vectors()
>>> dx, dy = R2_r.base_oneforms()
>>> ch = metric_to_Christoffel_2nd(TP(dx, dx) + TP(dy, dy))
```

```
>>> ch
[[[0, 0], [0, 0]], [[0, 0], [0, 0]]]
>>> cvd = CovarDerivativeOp(fx*e_x, ch)
>>> cvd(fx)
x
>>> cvd(fx*e_x)
x*e_x
```

sympy.diffgeom.intcurve\_series(*vector\_field, param, start\_point, n=6, coord\_sys=None, coeffs=False*)

Return the series expansion for an integral curve of the field.

#### Parameters

##### **vector\_field**

the vector field for which an integral curve will be given

##### **param**

the argument of the function  $\gamma$  from  $R$  to the curve

**start\_point**

the point which corresponds to  $\gamma(0)$

**n**

the order to which to expand

**coord\_sys**

the coordinate system in which to expand coeffs (default False) - if True return a list of elements of the expansion

## Explanation

Integral curve is a function  $\gamma$  taking a parameter in  $R$  to a point in the manifold. It verifies the equation:

$$V(f)(\gamma(t)) = \frac{d}{dt}f(\gamma(t))$$

where the given `vector_field` is denoted as  $V$ . This holds for any value  $t$  for the parameter and any scalar field  $f$ .

This equation can also be decomposed of a basis of coordinate functions  $V(f_i)(\gamma(t)) = \frac{d}{dt}f_i(\gamma(t)) \quad \forall i$

This function returns a series expansion of  $\gamma(t)$  in terms of the coordinate system `coord_sys`. The equations and expansions are necessarily done in coordinate-system-dependent way as there is no other way to represent movement between points on the manifold (i.e. there is no such thing as a difference of points for a general manifold).

## Examples

Use the predefined  $R^2$  manifold:

```
>>> from sympy.abc import t, x, y
>>> from sympy.diffgeom.rn import R2_p, R2_r
>>> from sympy.diffgeom import intcurve_series
```

Specify a starting point and a vector field:

```
>>> start_point = R2_r.point([x, y])
>>> vector_field = R2_r.e_x
```

Calculate the series:

```
>>> intcurve_series(vector_field, t, start_point, n=3)
Matrix([
[t + x],
[y]])
```

Or get the elements of the expansion in a list:

```
>>> series = intcurve_series(vector_field, t, start_point, n=3,
 ↪ coeffs=True)
>>> series[0]
Matrix([
[x],
[y]])
>>> series[1]
Matrix([
[t],
[0]])
>>> series[2]
Matrix([
[0],
[0]])
```

The series in the polar coordinate system:

```
>>> series = intcurve_series(vector_field, t, start_point,
... n=3, coord_sys=R2_p, coeffs=True)
>>> series[0]
Matrix([
[sqrt(x**2 + y**2)],
[atan2(y, x)]]
>>> series[1]
Matrix([
[t*x/sqrt(x**2 + y**2)],
[-t*y/(x**2 + y**2)]]
>>> series[2]
Matrix([
[t**2*(-x**2/(x**2 + y**2)**(3/2) + 1/sqrt(x**2 + y**2))/2],
[t**2*x*y/(x**2 + y**2)**2]])
```

**See also:**

[\*intcurve\\_diffegu\*](#) (page 2816)

`sympy.diffgeom.intcurve_diffegu(vector_field, param, start_point, coord_sys=None)`

Return the differential equation for an integral curve of the field.

**Parameters**

**vector\_field**

the vector field for which an integral curve will be given

**param**

the argument of the function  $\gamma$  from  $\mathbb{R}$  to the curve

**start\_point**

the point which corresponds to  $\gamma(0)$

**coord\_sys**

the coordinate system in which to give the equations

**Returns**

a tuple of (equations, initial conditions)



## Explanation

Integral curve is a function  $\gamma$  taking a parameter in  $R$  to a point in the manifold. It verifies the equation:

$$V(f)(\gamma(t)) = \frac{d}{dt}f(\gamma(t))$$

where the given `vector_field` is denoted as  $V$ . This holds for any value  $t$  for the parameter and any scalar field  $f$ .

This function returns the differential equation of  $\gamma(t)$  in terms of the coordinate system `coord_sys`. The equations and expansions are necessarily done in coordinate-system-dependent way as there is no other way to represent movement between points on the manifold (i.e. there is no such thing as a difference of points for a general manifold).

## Examples

Use the predefined R2 manifold:

```
>>> from sympy.abc import t
>>> from sympy.diffgeom.rn import R2, R2_p, R2_r
>>> from sympy.diffgeom import intcurve_diffequ
```

Specify a starting point and a vector field:

```
>>> start_point = R2_r.point([0, 1])
>>> vector_field = -R2.y*R2.e_x + R2.x*R2.e_y
```

Get the equation:

```
>>> equations, init_cond = intcurve_diffequ(vector_field, t, start_point)
>>> equations
[f_1(t) + Derivative(f_0(t), t), -f_0(t) + Derivative(f_1(t), t)]
>>> init_cond
[f_0(0), f_1(0) - 1]
```

The series in the polar coordinate system:

```
>>> equations, init_cond = intcurve_diffequ(vector_field, t, start_point,
→ R2_p)
>>> equations
[Derivative(f_0(t), t), Derivative(f_1(t), t) - 1]
>>> init_cond
[f_0(0) - 1, f_1(0) - pi/2]
```

### See also:

[`intcurve\_series`](#) (page 2814)

`sympy.diffgeom.vectors_in_basis(expr, to_sys)`

Transform all base vectors in base vectors of a specified coord basis. While the new base vectors are in the new coordinate system basis, any coefficients are kept in the old system.

### Examples

```
>>> from sympy.diffgeom import vectors_in_basis
>>> from sympy.diffgeom.rn import R2_r, R2_p
```

```
>>> vectors_in_basis(R2_r.e_x, R2_p)
-y*e_theta/(x**2 + y**2) + x*e_rho/sqrt(x**2 + y**2)
>>> vectors_in_basis(R2_p.e_r, R2_r)
sin(theta)*e_y + cos(theta)*e_x
```

`sympy.diffgeom.twoform_to_matrix(expr)`

Return the matrix representing the twoform.

For the twoform  $w$  return the matrix  $M$  such that  $M[i, j] = w(e_i, e_j)$ , where  $e_i$  is the  $i$ -th base vector field for the coordinate system in which the expression of  $w$  is given.

### Examples

```
>>> from sympy.diffgeom.rn import R2
>>> from sympy.diffgeom import twoform_to_matrix, TensorProduct
>>> TP = TensorProduct
```

```
>>> twoform_to_matrix(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
Matrix([
[1, 0],
[0, 1]])
>>> twoform_to_matrix(R2.x*TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
Matrix([
[x, 0],
[0, 1]])
>>> twoform_to_matrix(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy) - TP(R2.dx, R2.
→dy)/2)
Matrix([
[1, 0],
[-1/2, 1]])
```

`sympy.diffgeom.metric_to_Christoffel_1st(expr)`

Return the nested list of Christoffel symbols for the given metric. This returns the Christoffel symbol of first kind that represents the Levi-Civita connection for the given metric.

### Examples

```
>>> from sympy.diffgeom.rn import R2
>>> from sympy.diffgeom import metric_to_Christoffel_1st, TensorProduct
>>> TP = TensorProduct
```

```
>>> metric_to_Christoffel_1st(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
[[[0, 0], [0, 0]], [[0, 0], [0, 0]]]
```

(continues on next page)

(continued from previous page)

```
>>> metric_to_Christoffel_1st(R2.x*TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
[[[1/2, 0], [0, 0]], [[0, 0], [0, 0]]]
```

`sympy.diffgeom.metric_to_Christoffel_2nd(expr)`

Return the nested list of Christoffel symbols for the given metric. This returns the Christoffel symbol of second kind that represents the Levi-Civita connection for the given metric.

### Examples

```
>>> from sympy.diffgeom.rn import R2
>>> from sympy.diffgeom import metric_to_Christoffel_2nd, TensorProduct
>>> TP = TensorProduct
```

```
>>> metric_to_Christoffel_2nd(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
[[[0, 0], [0, 0]], [[0, 0], [0, 0]]]
>>> metric_to_Christoffel_2nd(R2.x*TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
[[[1/(2*x), 0], [0, 0]], [[0, 0], [0, 0]]]
```

`sympy.diffgeom.metric_to_Riemann_components(expr)`

Return the components of the Riemann tensor expressed in a given basis.

Given a metric it calculates the components of the Riemann tensor in the canonical basis of the coordinate system in which the metric expression is given.

### Examples

```
>>> from sympy import exp
>>> from sympy.diffgeom.rn import R2
>>> from sympy.diffgeom import metric_to_Riemann_components,
↳ TensorProduct
>>> TP = TensorProduct
```

```
>>> metric_to_Riemann_components(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
[[[[0, 0], [0, 0]], [[0, 0], [0, 0]]], [[[[0, 0], [0, 0]], [[0, 0], [0,
↳ 0]]]]]
>>> non_trivial_metric = exp(2*R2.r)*TP(R2.dr, R2.dr) + R2.
↳ r**2*TP(R2.dtheta, R2.dtheta)
>>> non_trivial_metric
exp(2*rho)*TensorProduct(drho, drho) + rho**2*TensorProduct(dtheta,
↳ dtheta)
>>> riemann = metric_to_Riemann_components(non_trivial_metric)
>>> riemann[0, :, :, :]
[[[0, 0], [0, 0]], [[0, exp(-2*rho)*rho], [-exp(-2*rho)*rho, 0]]]
>>> riemann[1, :, :, :]
[[[0, -1/rho], [1/rho, 0]], [[0, 0], [0, 0]]]
```

`sympy.diffgeom.metric_to_Ricci_components(expr)`

Return the components of the Ricci tensor expressed in a given basis.

Given a metric it calculates the components of the Ricci tensor in the canonical basis of the coordinate system in which the metric expression is given.

### Examples

```
>>> from sympy import exp
>>> from sympy.diffgeom.Rn import R2
>>> from sympy.diffgeom import metric_to_Ricci_components, TensorProduct
>>> TP = TensorProduct
```

```
>>> metric_to_Ricci_components(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
[[0, 0], [0, 0]]
>>> non_trivial_metric = exp(2*R2.r)*TP(R2.dr, R2.dr) +
→ R2.r**2*TP(R2.dtheta, R2.dtheta)
>>> non_trivial_metric
exp(2*rho)*TensorProduct(drho, drho) + rho**2*TensorProduct(dtheta,
→dtheta)
>>> metric_to_Ricci_components(non_trivial_metric)
[[1/rho, 0], [0, exp(-2*rho)*rho]]
```

## Plotting

### Introduction

The plotting module allows you to make 2-dimensional and 3-dimensional plots. Presently the plots are rendered using `matplotlib` as a backend. It is also possible to plot 2-dimensional plots using a *TextBackend* (page 2869) if you do not have `matplotlib`.

The plotting module has the following functions:

- `plot()` (page 2825): Plots 2D line plots.
- `plot_parametric()` (page 2828): Plots 2D parametric plots.
- `plot_implicit()` (page 2847): Plots 2D implicit and region plots.
- `plot3d()` (page 2838): Plots 3D plots of functions in two variables.
- `plot3d_parametric_line()` (page 2839): Plots 3D line plots, defined by a parameter.
- `plot3d_parametric_surface()` (page 2846): Plots 3D parametric surface plots.

The above functions are only for convenience and ease of use. It is possible to plot any plot by passing the corresponding *Series* class to *Plot* (page 2821) as argument.

## Plot Class

```
class sympy.plotting.plot.Plot(*args, title=None, xlabel=None, ylabel=None,
 ylabel=None, aspect_ratio='auto', xlim=None,
 ylim=None, axis_center='auto', axis=True,
 xscale='linear', yscale='linear', legend=False,
 autoscale=True, margin=0, annotations=None,
 markers=None, rectangles=None, fill=None,
 backend='default', size=None, **kwargs)
```

The central class of the plotting module.

## Explanation

For interactive work the function `plot()` (page 2825) is better suited.

This class permits the plotting of SymPy expressions using numerous backends (`matplotlib`, `textplot`, the old `pyglet` module for SymPy, Google charts api, etc).

The figure can contain an arbitrary number of plots of SymPy expressions, lists of coordinates of points, etc. Plot has a private attribute `_series` that contains all data series to be plotted (expressions for lines or surfaces, lists of points, etc (all subclasses of `BaseSeries`)). Those data series are instances of classes not imported by `from sympy import *`.

The customization of the figure is on two levels. Global options that concern the figure as a whole (e.g. title, xlabel, scale, etc) and per-data series options (e.g. name) and aesthetics (e.g. color, point shape, line type, etc.).

The difference between options and aesthetics is that an aesthetic can be a function of the coordinates (or parameters in a parametric plot). The supported values for an aesthetic are:

- None (the backend uses default values)
- a constant
- a function of one variable (the first coordinate or parameter)
- a function of two variables (the first and second coordinate or parameters)
- a function of three variables (only in nonparametric 3D plots)

Their implementation depends on the backend so they may not work in some backends.

If the plot is parametric and the arity of the aesthetic function permits it the aesthetic is calculated over parameters and not over coordinates. If the arity does not permit calculation over parameters the calculation is done over coordinates.

Only cartesian coordinates are supported for the moment, but you can use the parametric plots to plot in polar, spherical and cylindrical coordinates.

The arguments for the constructor `Plot` must be subclasses of `BaseSeries`.

Any global option can be specified as a keyword argument.

The global options for a figure are:

- title : str
- xlabel : str or Symbol

- `ylabel` : str or Symbol
- `zlabel` : str or Symbol
- `legend` : bool
- `xscale` : {'linear', 'log'}
- `yscale` : {'linear', 'log'}
- `axis` : bool
- `axis_center` : tuple of two floats or {'center', 'auto'}
- `xlim` : tuple of two floats
- `ylim` : tuple of two floats
- `aspect_ratio` : tuple of two floats or {'auto'}
- `autoscale` : bool
- `margin` : float in [0, 1]
- `backend` : {'default', 'matplotlib', 'text'} or a subclass of BaseBackend
- `size` : optional tuple of two floats, (width, height); default: None

The per data series options and aesthetics are: There are none in the base series. See below for options for subclasses.

Some data series support additional aesthetics or options:

[LineOver1DRangeSeries](#) (page 2866), [Parametric2DLineSeries](#) (page 2866), and [Parametric3DLineSeries](#) (page 2867) support the following:

Aesthetics:

- **line\_color**  
[string, or float, or function, optional] Specifies the color for the plot, which depends on the backend being used.

For example, if MatplotlibBackend is being used, then Matplotlib string colors are acceptable ("red", "r", "cyan", "c", ...). Alternatively, we can use a float number,  $0 < \text{color} < 1$ , wrapped in a string (for example, `line_color="0.5"`) to specify grayscale colors. Alternatively, We can specify a function returning a single float value: this will be used to apply a color-loop (for example, `line_color=lambda x: math.cos(x)`).

Note that by setting `line_color`, it would be applied simultaneously to all the series.

Options:

- `label` : str
- `steps` : bool
- `integers_only` : bool

[SurfaceOver2DRangeSeries](#) (page 2867) and [ParametricSurfaceSeries](#) (page 2867) support the following:

Aesthetics:

- `surface_color` : function which returns a float.

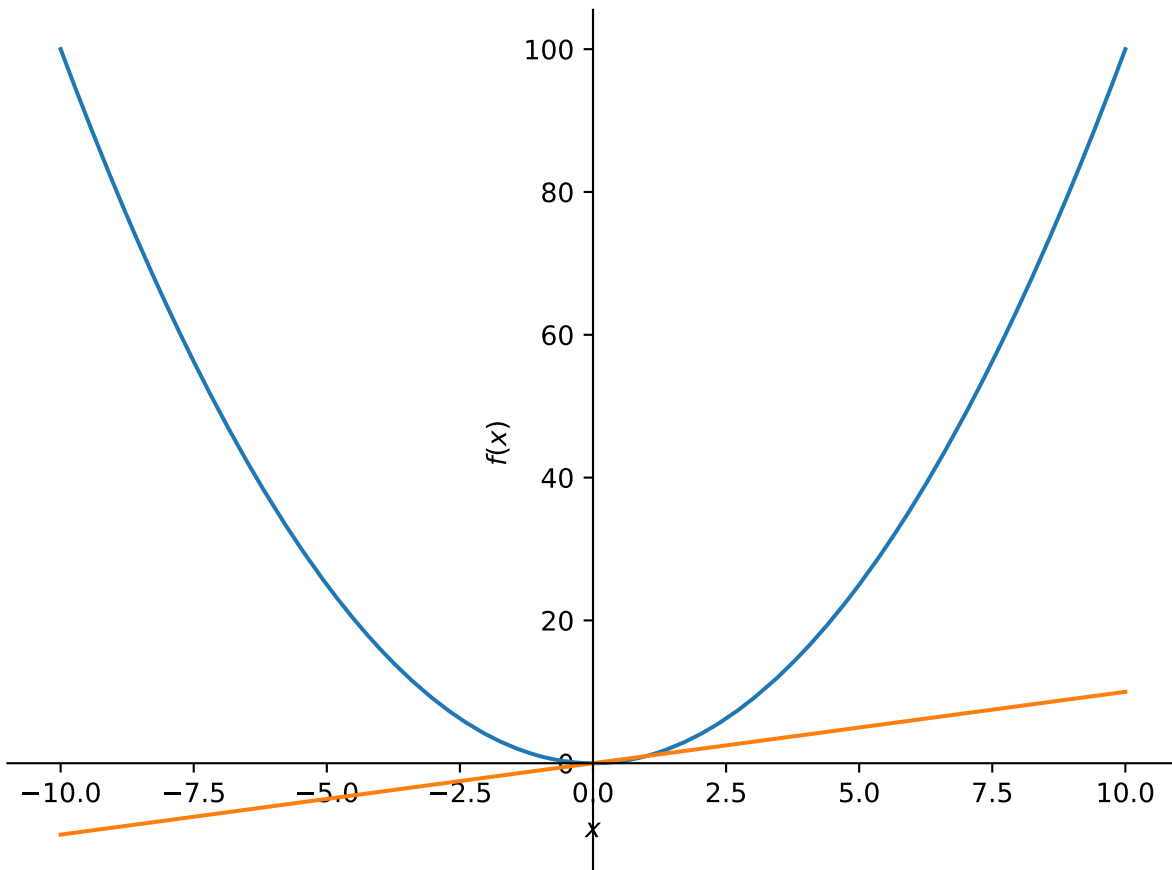
**append(*arg*)**

Adds an element from a plot's series to an existing plot.

**Examples**

Consider two Plot objects, p1 and p2. To add the second plot's first series object to the first, use the append method, like so:

```
>>> from sympy import symbols
>>> from sympy.plotting import plot
>>> x = symbols('x')
>>> p1 = plot(x*x, show=False)
>>> p2 = plot(x, show=False)
>>> p1.append(p2[0])
>>> p1
Plot object containing:
[0]: cartesian line: x**2 for x over (-10.0, 10.0)
[1]: cartesian line: x for x over (-10.0, 10.0)
>>> p1.show()
```

**See also:**

[extend](#) (page 2823)

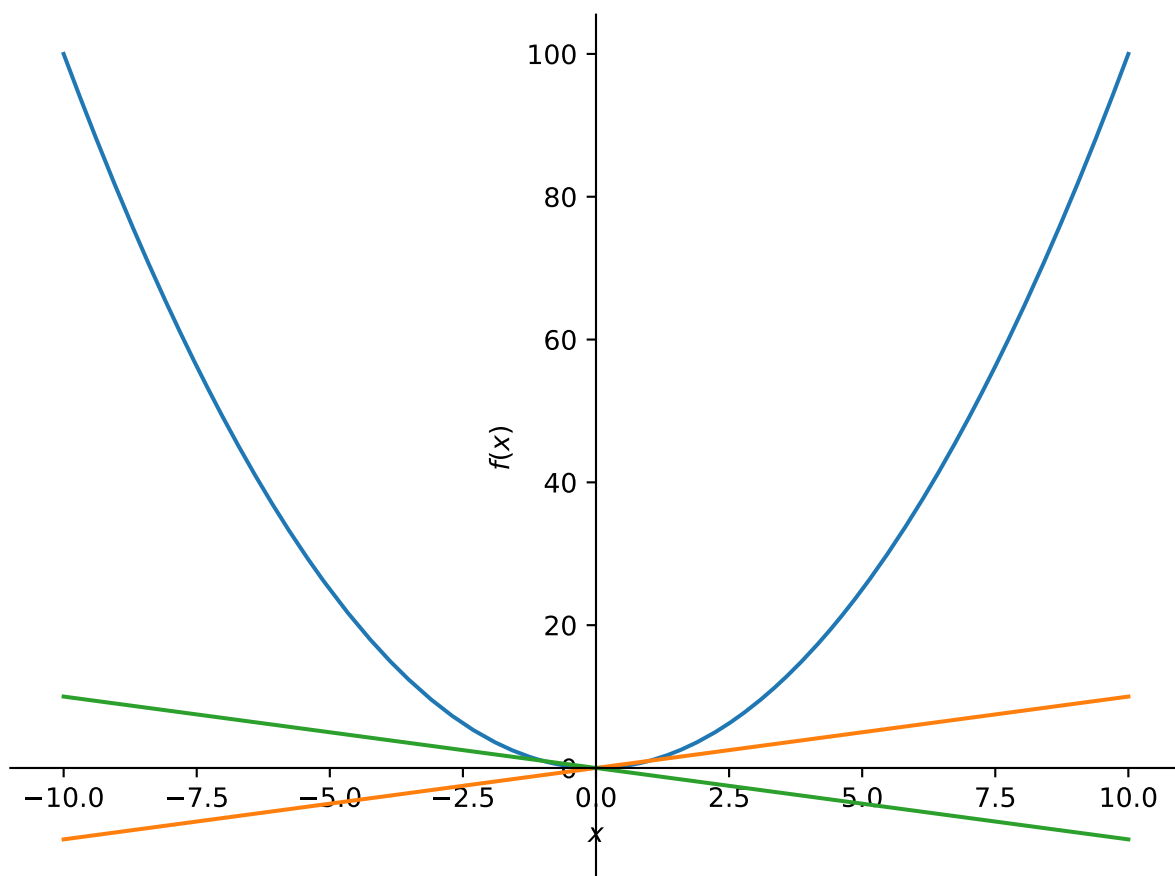
**extend(*arg*)**

Adds all series from another plot.

**Examples**

Consider two Plot objects, p1 and p2. To add the second plot to the first, use the extend method, like so:

```
>>> from sympy import symbols
>>> from sympy.plotting import plot
>>> x = symbols('x')
>>> p1 = plot(x**2, show=False)
>>> p2 = plot(x, -x, show=False)
>>> p1.extend(p2)
>>> p1
Plot object containing:
[0]: cartesian line: x**2 for x over (-10.0, 10.0)
[1]: cartesian line: x for x over (-10.0, 10.0)
[2]: cartesian line: -x for x over (-10.0, 10.0)
>>> p1.show()
```





## Plotting Function Reference

`sympy.plotting.plot.plot(*args, show=True, **kwargs)`

Plots a function of a single variable as a curve.

### Parameters

#### **args :**

The first argument is the expression representing the function of single variable to be plotted.

The last argument is a 3-tuple denoting the range of the free variable. e.g.  $(x, 0, 5)$

Typical usage examples are in the followings:

- **Plotting a single expression with a single range.**  
`plot(expr, range, **kwargs)`
- **Plotting a single expression with the default range (-10, 10).**  
`plot(expr, **kwargs)`
- **Plotting multiple expressions with a single range.**  
`plot(expr1, expr2, ..., range, **kwargs)`
- **Plotting multiple expressions with multiple ranges.**  
`plot((expr1, range1), (expr2, range2), ..., **kwargs)`

It is best practice to specify range explicitly because default range may change in the future if a more advanced default range detection algorithm is implemented.

**show** : bool, optional

The default value is set to True. Set show to False and the function will not display the plot. The returned instance of the Plot class can then be used to save or display the plot by calling the `save()` and `show()` methods respectively.

**line\_color** : string, or float, or function, optional

Specifies the color for the plot. See Plot to see how to set color for the plots. Note that by setting `line_color`, it would be applied simultaneously to all the series.

**title** : str, optional

Title of the plot. It is set to the latex representation of the expression, if the plot has only one expression.

**label** : str, optional

The label of the expression in the plot. It will be used when called with `legend`. Default is the name of the expression. e.g.  $\sin(x)$

**xlabel** : str or expression, optional

Label for the x-axis.

**ylabel** : str or expression, optional

Label for the y-axis.

**xscale** : 'linear' or 'log', optional

Sets the scaling of the x-axis.

**yscale** : 'linear' or 'log', optional

Sets the scaling of the y-axis.

**axis\_center** : (float, float), optional

Tuple of two floats denoting the coordinates of the center or {'center', 'auto'}

**xlim** : (float, float), optional

Denotes the x-axis limits, (min, max)`.

**ylim** : (float, float), optional

Denotes the y-axis limits, (min, max)`.

**annotations** : list, optional

A list of dictionaries specifying the type of annotation required. The keys in the dictionary should be equivalent to the arguments of the `matplotlib`'s `annotate()` method.

**markers** : list, optional

A list of dictionaries specifying the type the markers required. The keys in the dictionary should be equivalent to the arguments of the `matplotlib`'s `plot()` function along with the marker related key-worded arguments.

**rectangles** : list, optional

A list of dictionaries specifying the dimensions of the rectangles to be plotted. The keys in the dictionary should be equivalent to the arguments of the `matplotlib`'s `Rectangle` class.

**fill** : dict, optional

A dictionary specifying the type of color filling required in the plot. The keys in the dictionary should be equivalent to the arguments of the `matplotlib`'s `fill_between()` method.

**adaptive** : bool, optional

The default value is set to True. Set adaptive to False and specify `nb_of_points` if uniform sampling is required.

The plotting uses an adaptive algorithm which samples recursively to accurately plot. The adaptive algorithm uses a random point near the midpoint of two points that has to be further sampled. Hence the same plots can appear slightly different.

**depth** : int, optional

Recursion depth of the adaptive algorithm. A depth of value  $n$  samples a maximum of  $2^n$  points.

If the adaptive flag is set to False, this will be ignored.

**nb\_of\_points** : int, optional

Used when the adaptive is set to False. The function is uniformly sampled at `nb_of_points` number of points.

If the adaptive flag is set to True, this will be ignored.

**size** : (float, float), optional

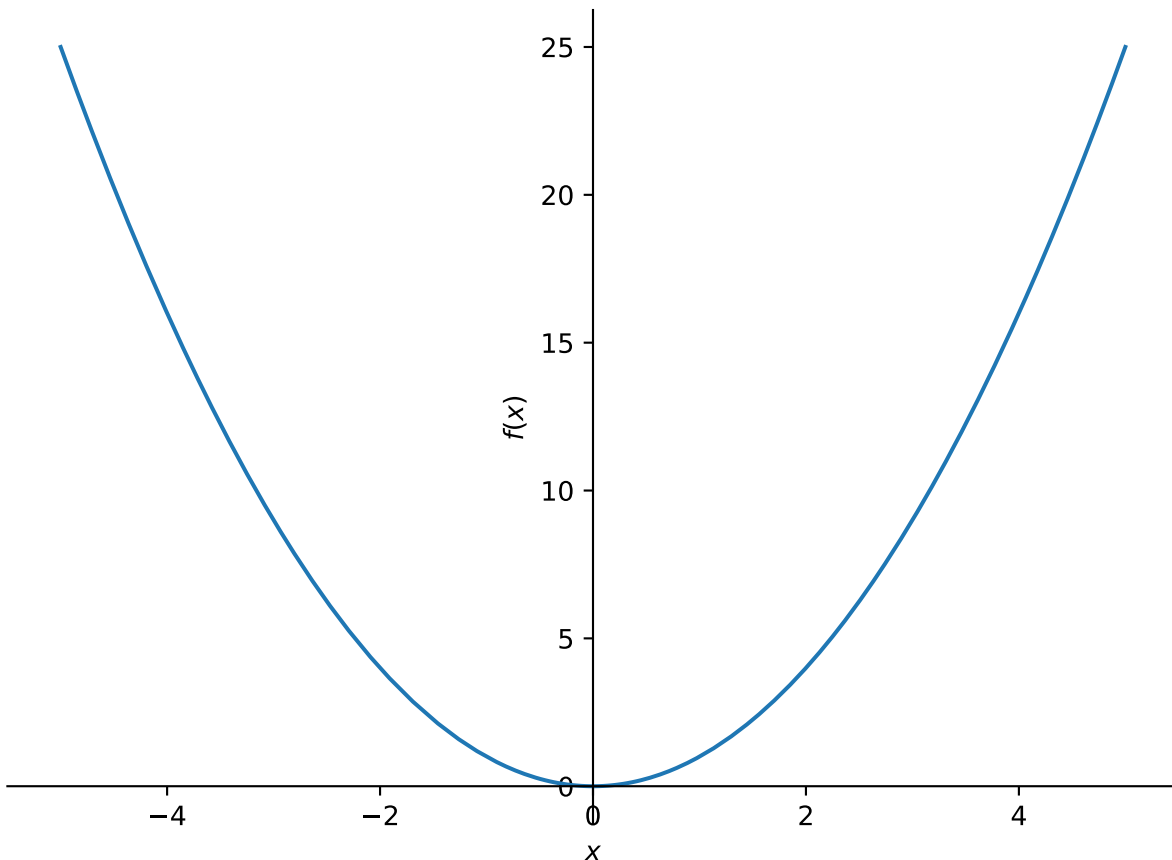
A tuple in the form (width, height) in inches to specify the size of the overall figure. The default value is set to None, meaning the size will be set by the default backend.

## Examples

```
>>> from sympy import symbols
>>> from sympy.plotting import plot
>>> x = symbols('x')
```

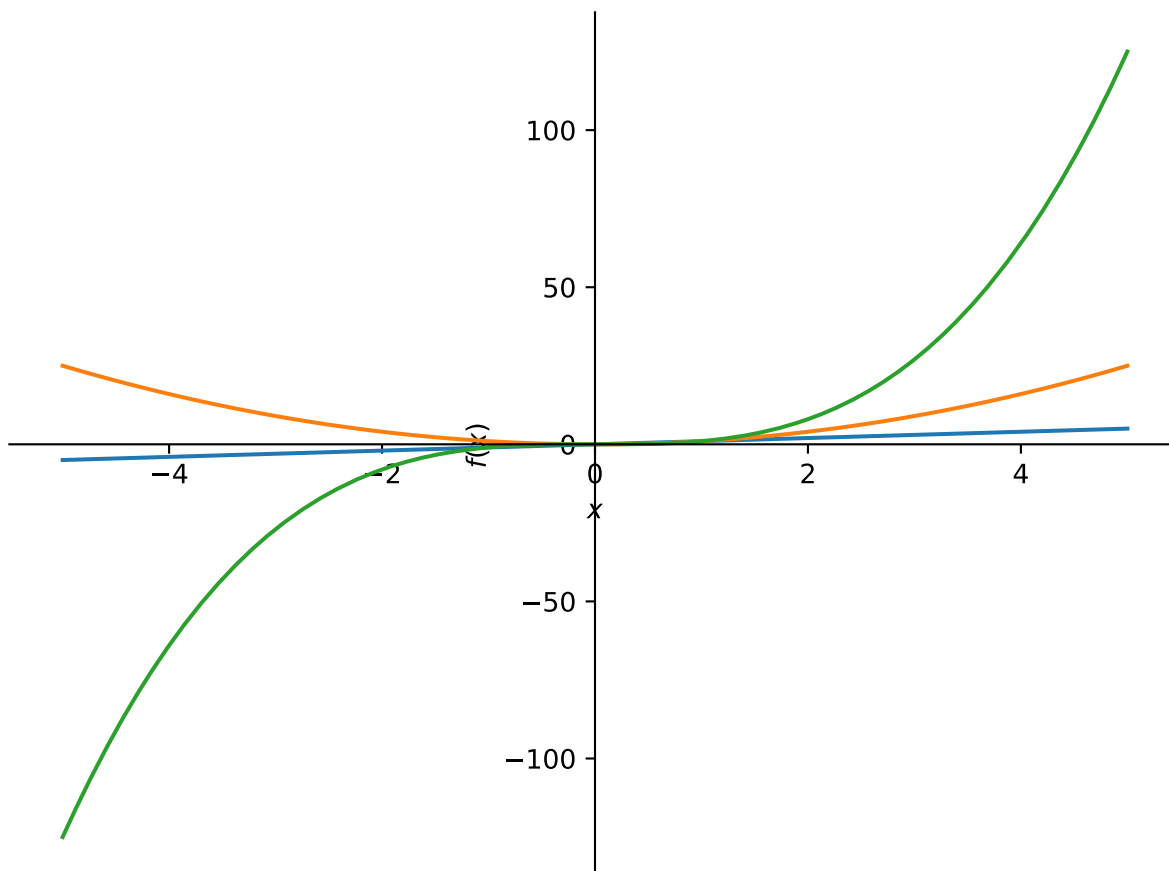
Single Plot

```
>>> plot(x**2, (x, -5, 5))
Plot object containing:
[0]: cartesian line: x**2 for x over (-5.0, 5.0)
```



Multiple plots with single range.

```
>>> plot(x, x**2, x**3, (x, -5, 5))
Plot object containing:
[0]: cartesian line: x for x over (-5.0, 5.0)
[1]: cartesian line: x**2 for x over (-5.0, 5.0)
[2]: cartesian line: x**3 for x over (-5.0, 5.0)
```



Multiple plots with different ranges.

```
>>> plot((x**2, (x, -6, 6)), (x, (x, -5, 5)))
Plot object containing:
[0]: cartesian line: x**2 for x over (-6.0, 6.0)
[1]: cartesian line: x for x over (-5.0, 5.0)
```

No adaptive sampling.

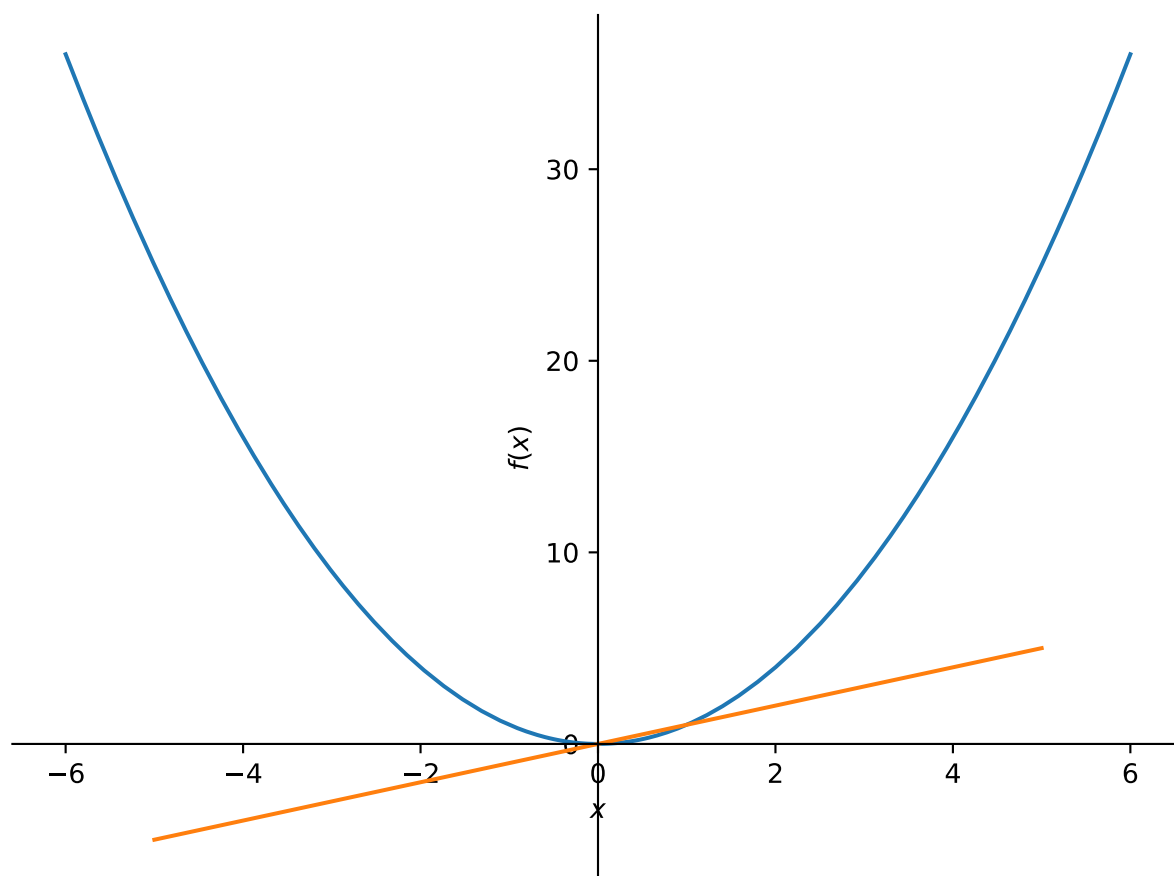
```
>>> plot(x**2, adaptive=False, nb_of_points=400)
Plot object containing:
[0]: cartesian line: x**2 for x over (-10.0, 10.0)
```

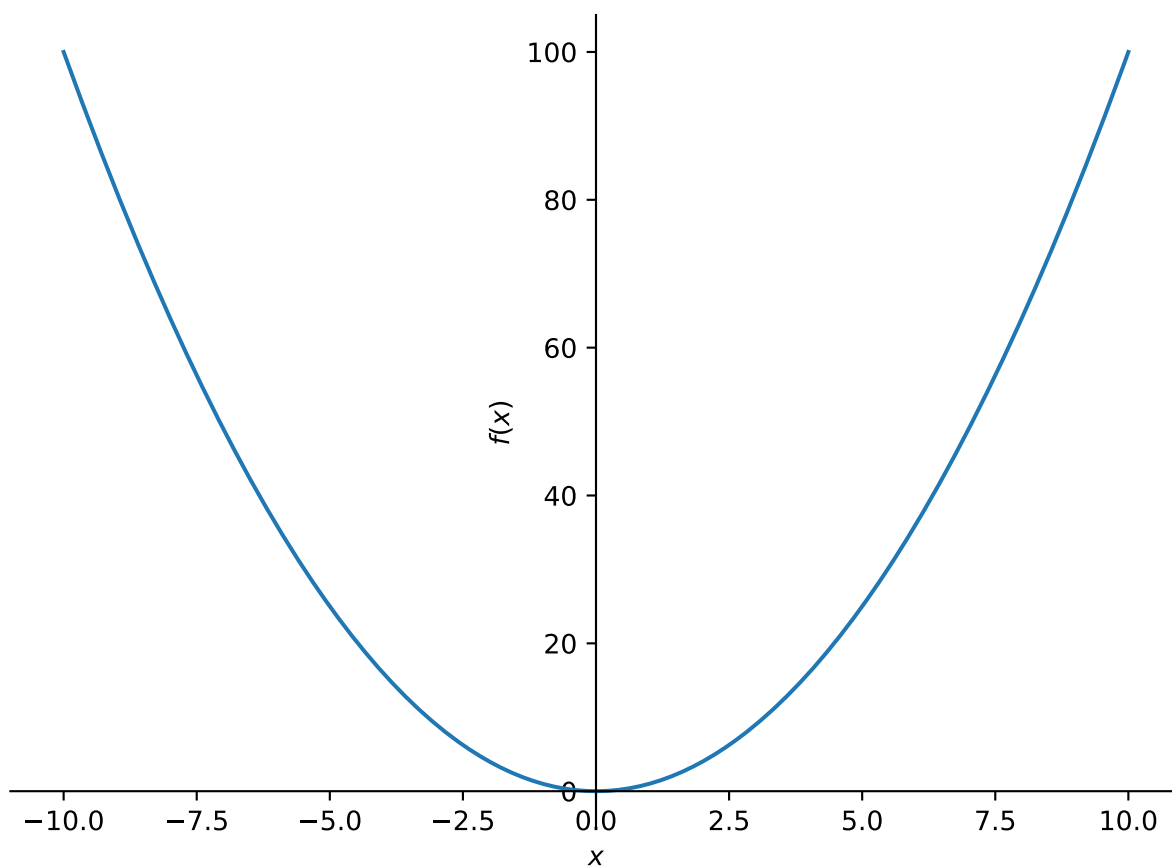
**See also:**

[Plot](#) (page 2821), [LineOver1DRangeSeries](#) (page 2866)

`sympy.plotting.plot.plot_parametric(*args, show=True, **kwargs)`

Plots a 2D parametric curve.





**Parameters****args**

Common specifications are:

- **Plotting a single parametric curve with a range**  
`plot_parametric((expr_x, expr_y), range)`
- **Plotting multiple parametric curves with the same range**  
`plot_parametric((expr_x, expr_y), ..., range)`
- **Plotting multiple parametric curves with different ranges**  
`plot_parametric((expr_x, expr_y, range), ...)`

`expr_x` is the expression representing  $x$  component of the parametric function.

`expr_y` is the expression representing  $y$  component of the parametric function.

`range` is a 3-tuple denoting the parameter symbol, start and stop. For example, `(u, 0, 5)`.

If the range is not specified, then a default range of `(-10, 10)` is used.

However, if the arguments are specified as `(expr_x, expr_y, range), ...,` you must specify the ranges for each expressions manually.

Default range may change in the future if a more advanced algorithm is implemented.

**adaptive** : bool, optional

Specifies whether to use the adaptive sampling or not.

The default value is set to `True`. Set `adaptive` to `False` and specify `nb_of_points` if uniform sampling is required.

**depth** : int, optional

The recursion depth of the adaptive algorithm. A depth of value  $n$  samples a maximum of  $2^n$  points.

**nb\_of\_points** : int, optional

Used when the `adaptive` flag is set to `False`.

Specifies the number of the points used for the uniform sampling.

**line\_color** : string, or float, or function, optional

Specifies the color for the plot. See `Plot` to see how to set color for the plots. Note that by setting `line_color`, it would be applied simultaneously to all the series.

**label** : str, optional

The label of the expression in the plot. It will be used when called with `legend`. Default is the name of the expression. e.g. `sin(x)`

**xlabel** : str, optional

Label for the x-axis.

**ylabel** : str, optional

Label for the y-axis.

**xscale** : 'linear' or 'log', optional

Sets the scaling of the x-axis.

**yscale** : 'linear' or 'log', optional

Sets the scaling of the y-axis.

**axis\_center** : (float, float), optional

Tuple of two floats denoting the coordinates of the center or {'center', 'auto'}

**xlim** : (float, float), optional

Denotes the x-axis limits, (min, max)`.

**ylim** : (float, float), optional

Denotes the y-axis limits, (min, max)`.

**size** : (float, float), optional

A tuple in the form (width, height) in inches to specify the size of the overall figure. The default value is set to None, meaning the size will be set by the default backend.

## Examples

```
>>> from sympy import plot_parametric, symbols, cos, sin
>>> u = symbols('u')
```

A parametric plot with a single expression:

```
>>> plot_parametric((cos(u), sin(u)), (u, -5, 5))
Plot object containing:
[0]: parametric cartesian line: (cos(u), sin(u)) for u over (-5.0, 5.0)
```

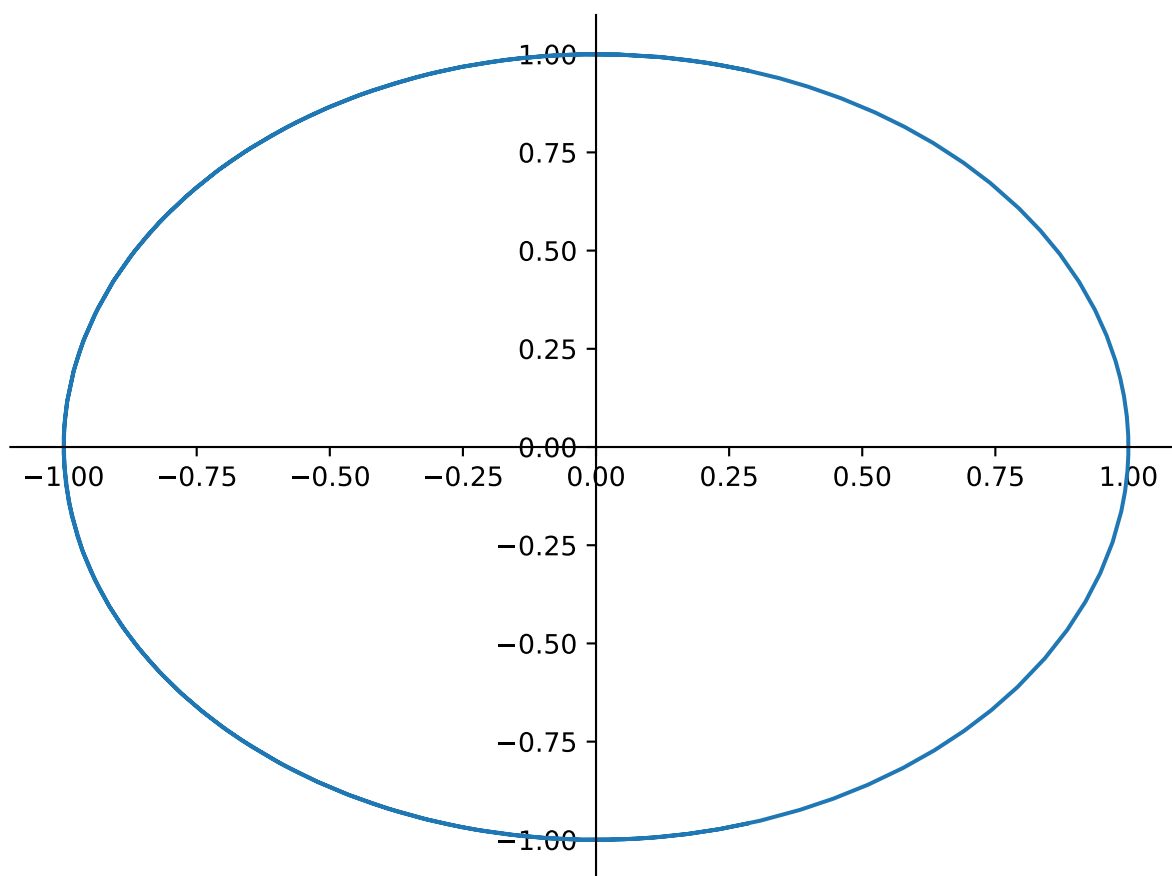
A parametric plot with multiple expressions with the same range:

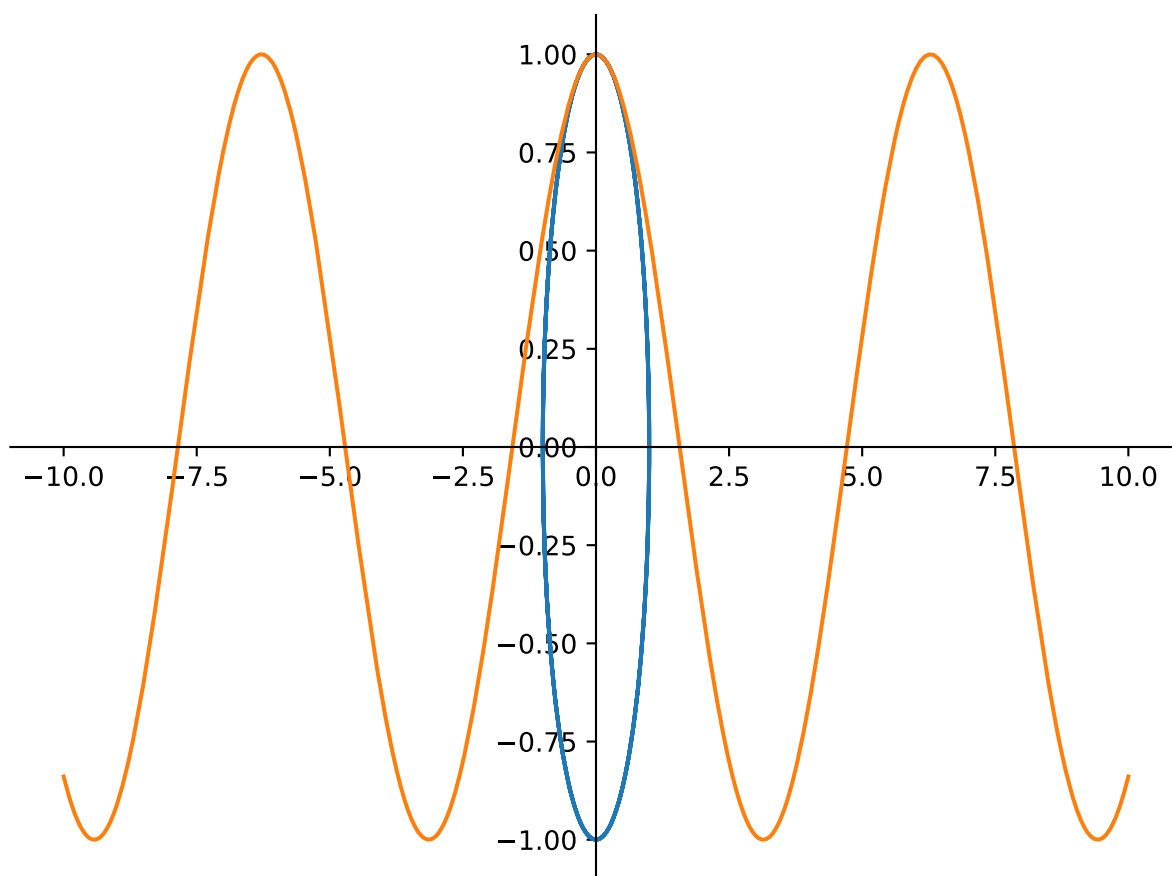
```
>>> plot_parametric((cos(u), sin(u)), (u, cos(u)), (u, -10, 10))
Plot object containing:
[0]: parametric cartesian line: (cos(u), sin(u)) for u over (-10.0, 10.0)
[1]: parametric cartesian line: (u, cos(u)) for u over (-10.0, 10.0)
```

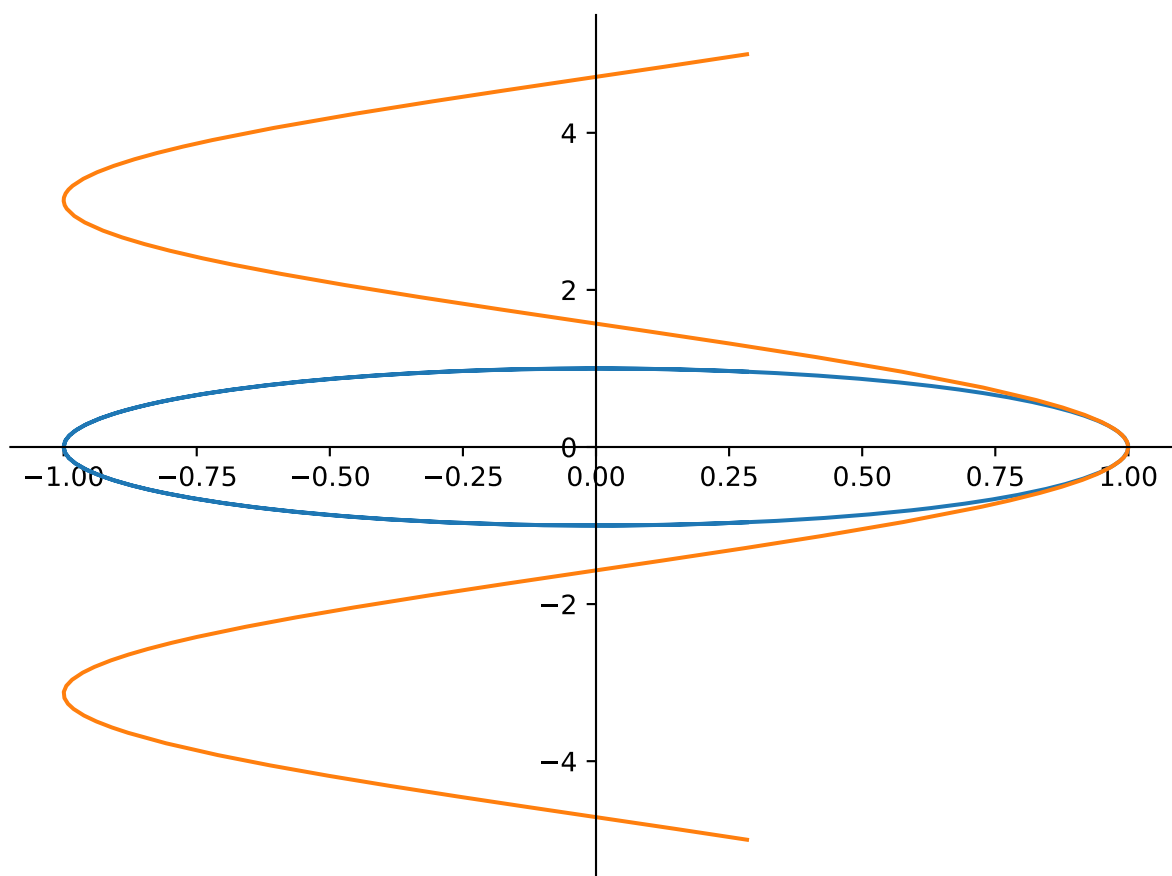
A parametric plot with multiple expressions with different ranges for each curve:

```
>>> plot_parametric((cos(u), sin(u), (u, -5, 5)),
... (cos(u), u, (u, -5, 5)))
Plot object containing:
[0]: parametric cartesian line: (cos(u), sin(u)) for u over (-5.0, 5.0)
[1]: parametric cartesian line: (cos(u), u) for u over (-5.0, 5.0)
```









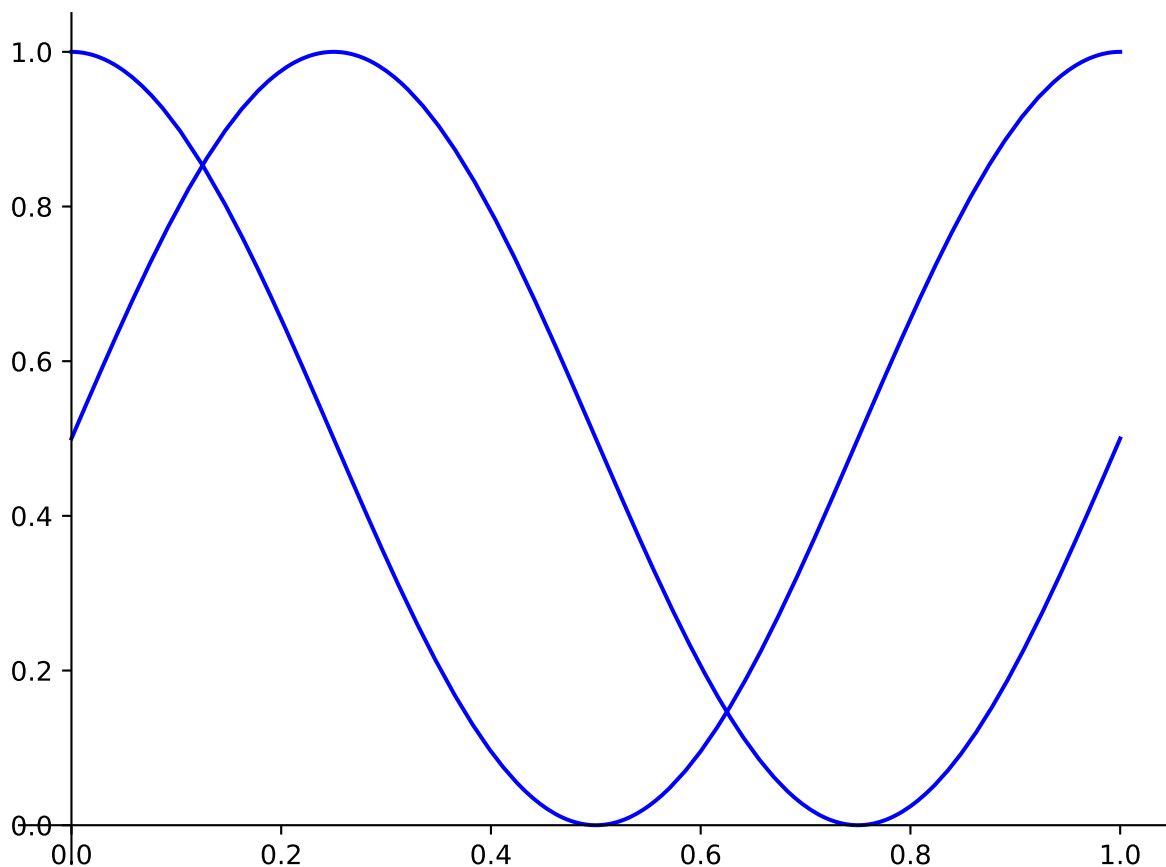
## Notes

The plotting uses an adaptive algorithm which samples recursively to accurately plot the curve. The adaptive algorithm uses a random point near the midpoint of two points that has to be further sampled. Hence, repeating the same plot command can give slightly different results because of the random sampling.

If there are multiple plots, then the same optional arguments are applied to all the plots drawn in the same canvas. If you want to set these options separately, you can index the returned Plot object and set it.

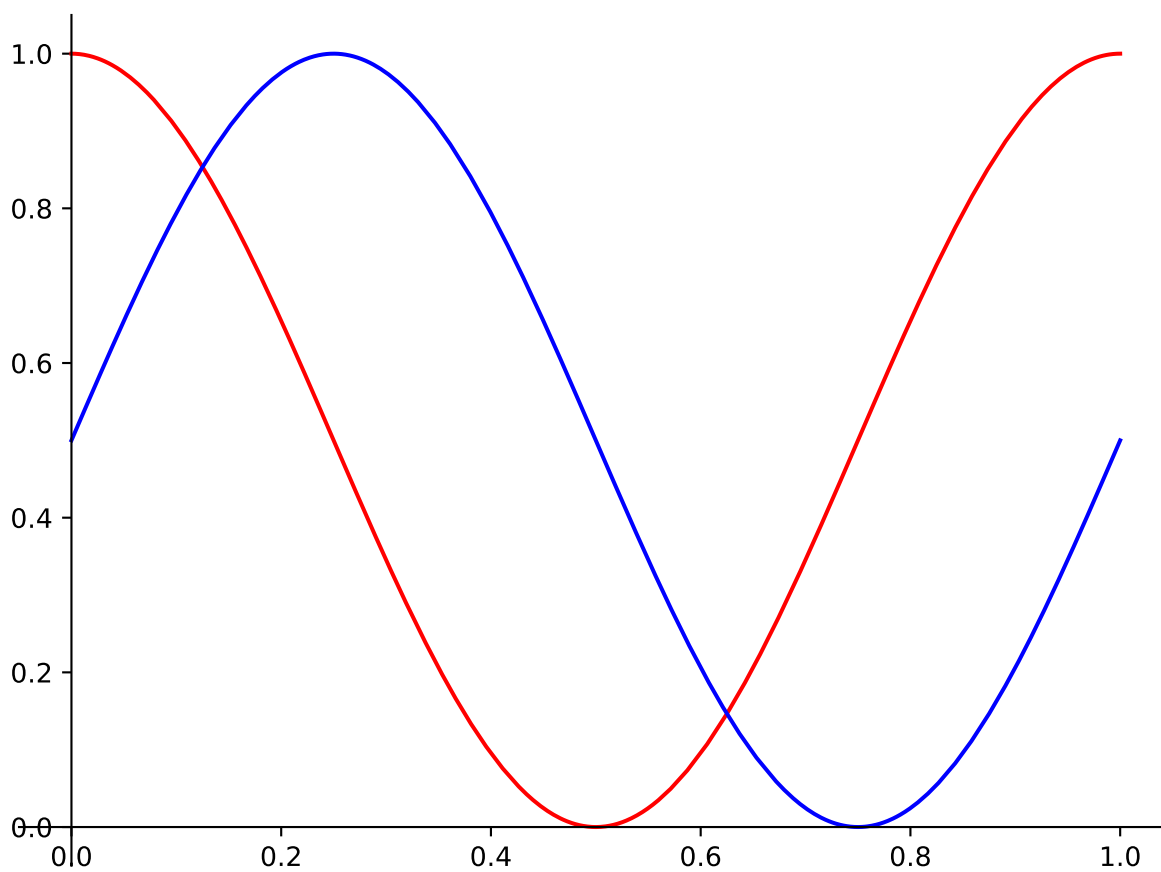
For example, when you specify `line_color` once, it would be applied simultaneously to both series.

```
>>> from sympy import pi
>>> expr1 = (u, cos(2*pi*u)/2 + 1/2)
>>> expr2 = (u, sin(2*pi*u)/2 + 1/2)
>>> p = plot_parametric(expr1, expr2, (u, 0, 1), line_color='blue')
```



If you want to specify the line color for the specific series, you should index each item and apply the property manually.

```
>>> p[0].line_color = 'red'
>>> p.show()
```



**See also:**

[Plot](#) (page 2821), [Parametric2DLineSeries](#) (page 2866)

`sympy.plotting.plot.plot3d(*args, show=True, **kwargs)`

Plots a 3D surface plot.

**Usage**

Single plot

`plot3d(expr, range_x, range_y, **kwargs)`

If the ranges are not specified, then a default range of (-10, 10) is used.

Multiple plot with the same range.

`plot3d(expr1, expr2, range_x, range_y, **kwargs)`

If the ranges are not specified, then a default range of (-10, 10) is used.

Multiple plots with different ranges.

`plot3d((expr1, range_x, range_y), (expr2, range_x, range_y), ..., **kwargs)`

Ranges have to be specified for every expression.

Default range may change in the future if a more advanced default range detection algorithm is implemented.

**Arguments**

`expr` : Expression representing the function along x.

**`range_x`**

[[Symbol](#) (page 976), float, float)] A 3-tuple denoting the range of the x variable, e.g. (x, 0, 5).

**`range_y`**

[[Symbol](#) (page 976), float, float)] A 3-tuple denoting the range of the y variable, e.g. (y, 0, 5).

**Keyword Arguments**

Arguments for `SurfaceOver2DRangeSeries` class:

**`nb_of_points_x`**

[int] The x range is sampled uniformly at `nb_of_points_x` of points.

**`nb_of_points_y`**

[int] The y range is sampled uniformly at `nb_of_points_y` of points.

Aesthetics:

**`surface_color`**

[Function which returns a float] Specifies the color for the surface of the plot. See [Plot](#) (page 2821) for more details.

If there are multiple plots, then the same series arguments are applied to all the plots. If you want to set these options separately, you can index the returned Plot object and set it.

Arguments for Plot class:

**title**

[str] Title of the plot.

**size**

[(float, float), optional] A tuple in the form (width, height) in inches to specify the size of the overall figure. The default value is set to None, meaning the size will be set by the default backend.

## Examples

```
>>> from sympy import symbols
>>> from sympy.plotting import plot3d
>>> x, y = symbols('x y')
```

Single plot

```
>>> plot3d(x*y, (x, -5, 5), (y, -5, 5))
Plot object containing:
[0]: cartesian surface: x*y for x over (-5.0, 5.0) and y over (-5.0, 5.0)
```

Multiple plots with same range

```
>>> plot3d(x*y, -x*y, (x, -5, 5), (y, -5, 5))
Plot object containing:
[0]: cartesian surface: x*y for x over (-5.0, 5.0) and y over (-5.0, 5.0)
[1]: cartesian surface: -x*y for x over (-5.0, 5.0) and y over (-5.0, 5.0)
→0)
```

Multiple plots with different ranges.

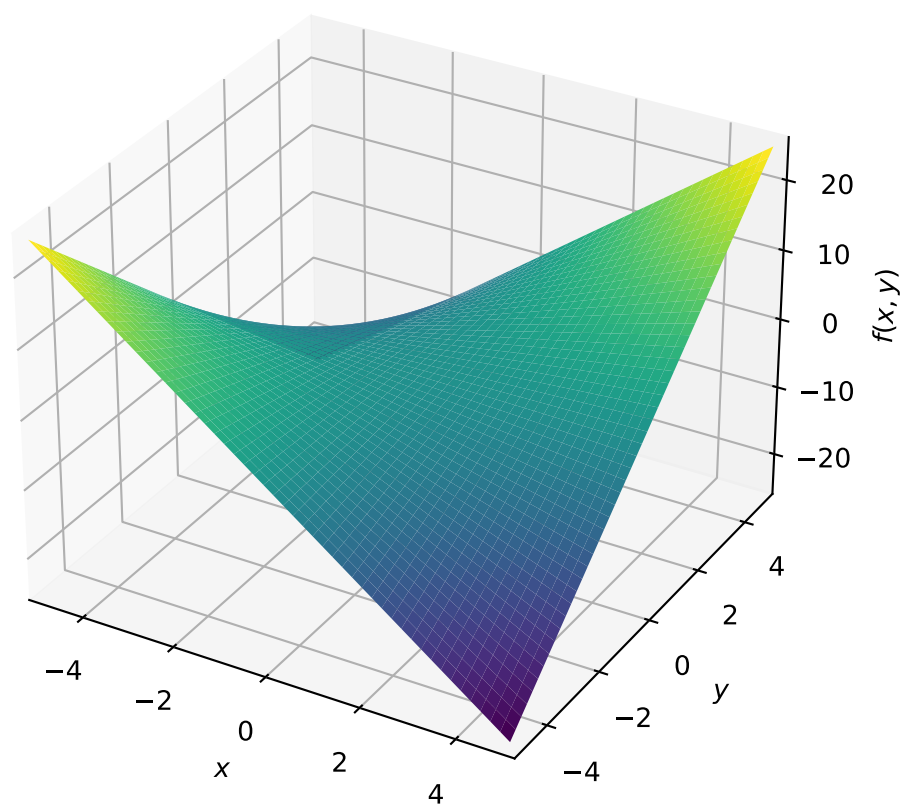
```
>>> plot3d((x**2 + y**2, (x, -5, 5), (y, -5, 5)),
... (x*y, (x, -3, 3), (y, -3, 3)))
Plot object containing:
[0]: cartesian surface: x**2 + y**2 for x over (-5.0, 5.0) and y over (-5.0, 5.0)
→5.0, 5.0)
[1]: cartesian surface: x*y for x over (-3.0, 3.0) and y over (-3.0, 3.0)
```

**See also:**

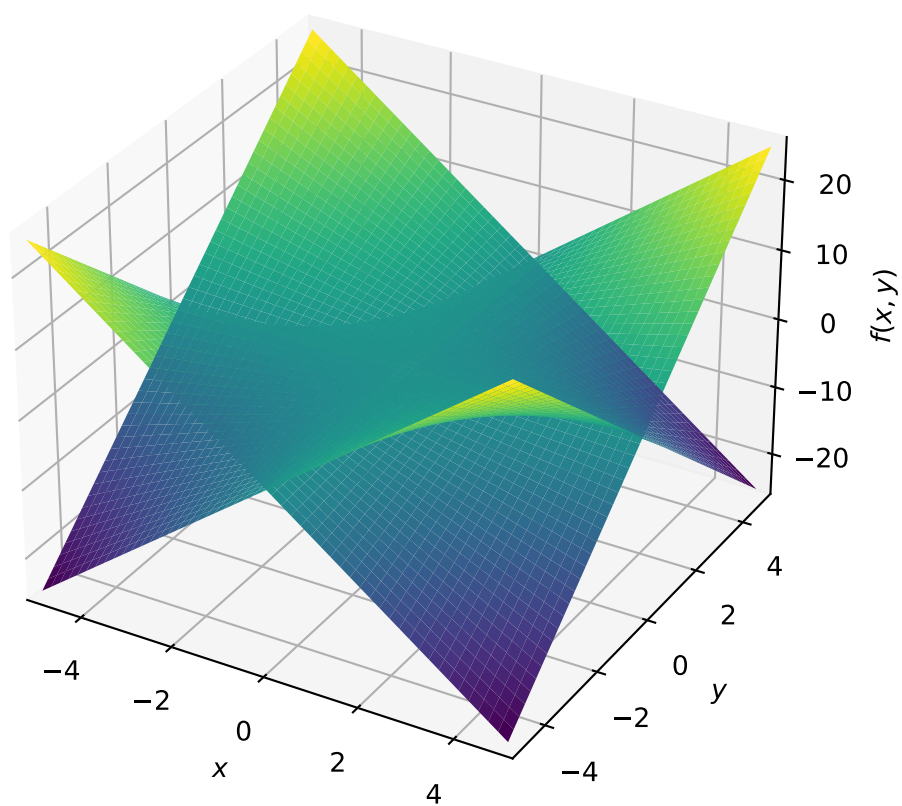
[Plot](#) (page 2821), [SurfaceOver2DRangeSeries](#) (page 2867)

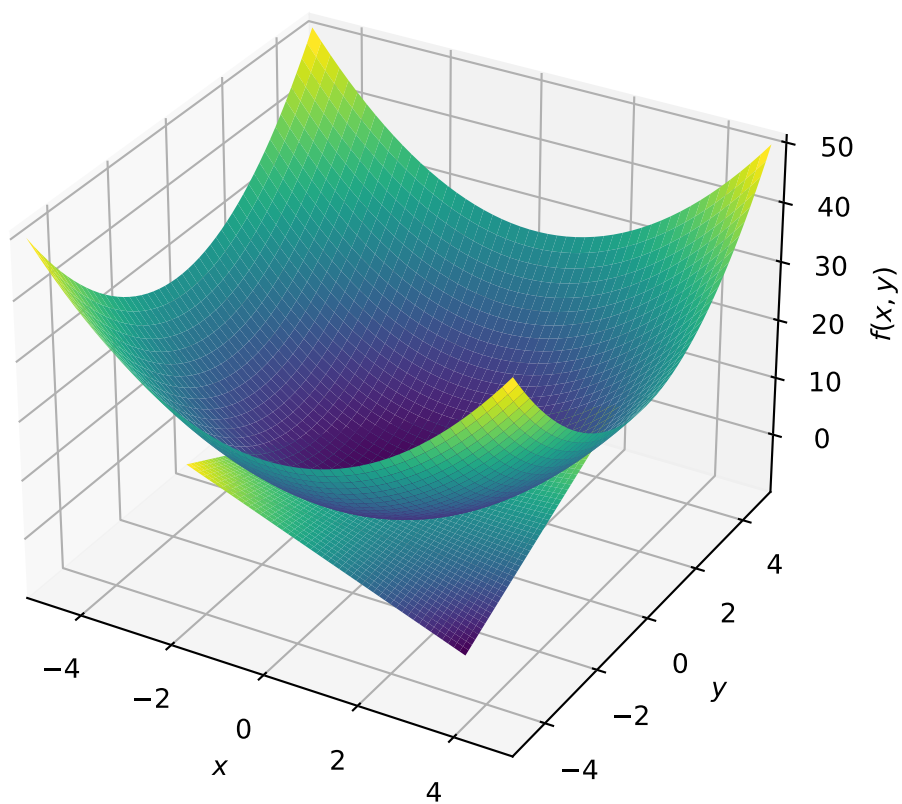
`sympy.plotting.plot.plot3d_parametric_line(*args, show=True, **kwargs)`

Plots a 3D parametric line plot.









## Usage

Single plot:

```
plot3d_parametric_line(expr_x, expr_y, expr_z, range, **kwargs)
```

If the range is not specified, then a default range of (-10, 10) is used.

Multiple plots.

```
plot3d_parametric_line((expr_x, expr_y, expr_z, range), ..., **kwargs)
```

Ranges have to be specified for every expression.

Default range may change in the future if a more advanced default range detection algorithm is implemented.

## Arguments

`expr_x` : Expression representing the function along x.

`expr_y` : Expression representing the function along y.

`expr_z` : Expression representing the function along z.

### **range**

`[(Symbol (page 976), float, float)]` A 3-tuple denoting the range of the parameter variable, e.g., (u, 0, 5).

## Keyword Arguments

Arguments for Parametric3DLineSeries class.

`nb_of_points` : The range is uniformly sampled at `nb_of_points` number of points.

Aesthetics:

### **line\_color**

`[string, or float, or function, optional]` Specifies the color for the plot. See Plot to see how to set color for the plots. Note that by setting `line_color`, it would be applied simultaneously to all the series.

### **label**

`[str]` The label to the plot. It will be used when called with `legend=True` to denote the function with the given label in the plot.

If there are multiple plots, then the same series arguments are applied to all the plots. If you want to set these options separately, you can index the returned Plot object and set it.

Arguments for Plot class.

### **title**

`[str]` Title of the plot.

### **size**

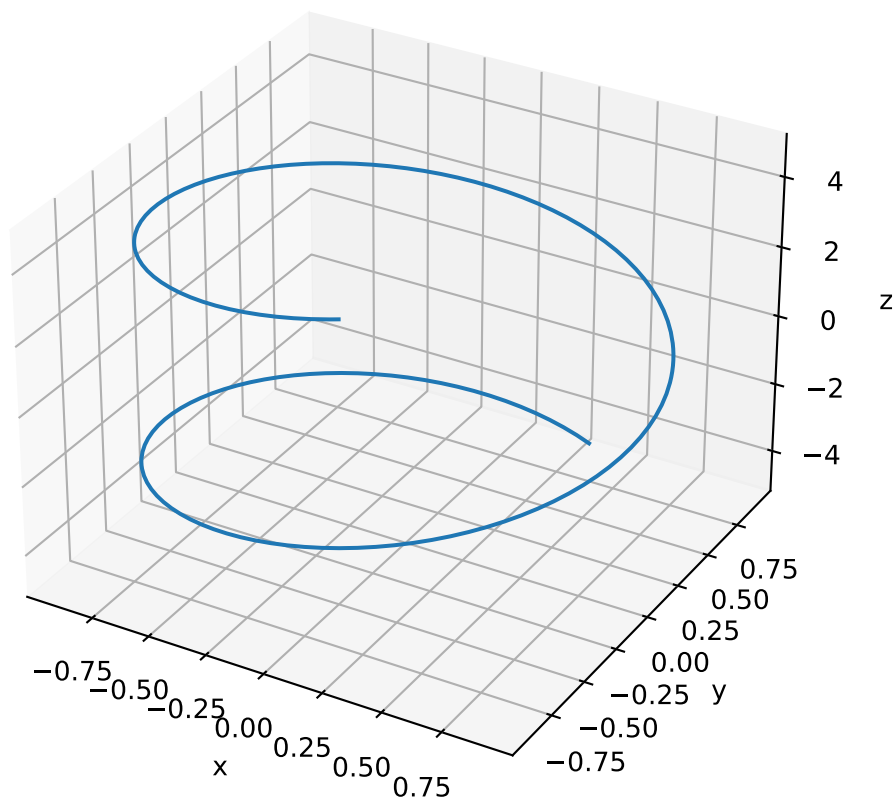
`[(float, float), optional]` A tuple in the form (width, height) in inches to specify the size of the overall figure. The default value is set to None, meaning the size will be set by the default backend.

## Examples

```
>>> from sympy import symbols, cos, sin
>>> from sympy.plotting import plot3d_parametric_line
>>> u = symbols('u')
```

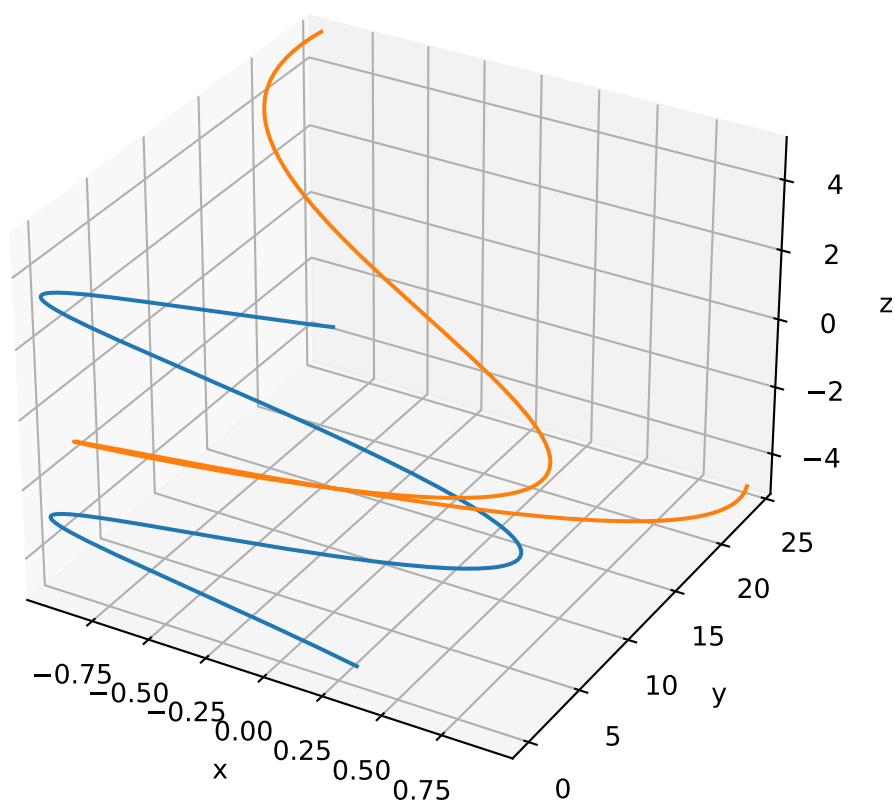
Single plot.

```
>>> plot3d_parametric_line(cos(u), sin(u), u, (u, -5, 5))
Plot object containing:
[0]: 3D parametric cartesian line: (cos(u), sin(u), u) for u over (-5.0, 5.0)
```



Multiple plots.

```
>>> plot3d_parametric_line((cos(u), sin(u), u, (u, -5, 5)),
... (sin(u), u**2, u, (u, -5, 5)))
Plot object containing:
[0]: 3D parametric cartesian line: (cos(u), sin(u), u) for u over (-5.0, 5.0)
[1]: 3D parametric cartesian line: (sin(u), u**2, u) for u over (-5.0, 5.0)
```



**See also:**

[Plot](#) (page 2821), [Parametric3DLineSeries](#) (page 2867)

`sympy.plotting.plot.plot3d_parametric_surface(*args, show=True, **kwargs)`

Plots a 3D parametric surface plot.

**Explanation**

Single plot.

```
plot3d_parametric_surface(expr_x, expr_y, expr_z, range_u, range_v,
**kwargs)
```

If the ranges is not specified, then a default range of (-10, 10) is used.

Multiple plots.

```
plot3d_parametric_surface((expr_x, expr_y, expr_z, range_u, range_v), ...
, **kwargs)
```

Ranges have to be specified for every expression.

Default range may change in the future if a more advanced default range detection algorithm is implemented.

**Arguments**

`expr_x` : Expression representing the function along x.

`expr_y` : Expression representing the function along y.

`expr_z` : Expression representing the function along z.

**`range_u`**

[Symbol](#) (page 976), float, float)] A 3-tuple denoting the range of the u variable, e.g. (u, 0, 5).

**`range_v`**

[Symbol](#) (page 976), float, float)] A 3-tuple denoting the range of the v variable, e.g. (v, 0, 5).

**Keyword Arguments**

Arguments for ParametricSurfaceSeries class:

**`nb_of_points_u`**

[int] The u range is sampled uniformly at nb\_of\_points\_v of points

**`nb_of_points_y`**

[int] The v range is sampled uniformly at nb\_of\_points\_y of points

Aesthetics:

**`surface_color`**

[Function which returns a float] Specifies the color for the surface of the plot. See [Plot](#) (page 2821) for more details.

If there are multiple plots, then the same series arguments are applied for all the plots. If you want to set these options separately, you can index the returned Plot object and set it.

Arguments for Plot class:

#### title

[str] Title of the plot.

#### size

[(float, float), optional] A tuple in the form (width, height) in inches to specify the size of the overall figure. The default value is set to None, meaning the size will be set by the default backend.

### Examples

```
>>> from sympy import symbols, cos, sin
>>> from sympy.plotting import plot3d_parametric_surface
>>> u, v = symbols('u v')
```

Single plot.

```
>>> plot3d_parametric_surface(cos(u + v), sin(u - v), u - v,
... (u, -5, 5), (v, -5, 5))
Plot object containing:
[0]: parametric cartesian surface: (cos(u + v), sin(u - v), u - v) for u
→ over (-5.0, 5.0) and v over (-5.0, 5.0)
```

#### See also:

[Plot](#) (page 2821), [ParametricSurfaceSeries](#) (page 2867)

```
sympy.plotting.plot_implicit.plot_implicit(expr, x_var=None, y_var=None,
 adaptive=True, depth=0, points=300,
 line_color='blue', show=True, **kwargs)
```

A plot function to plot implicit equations / inequalities.

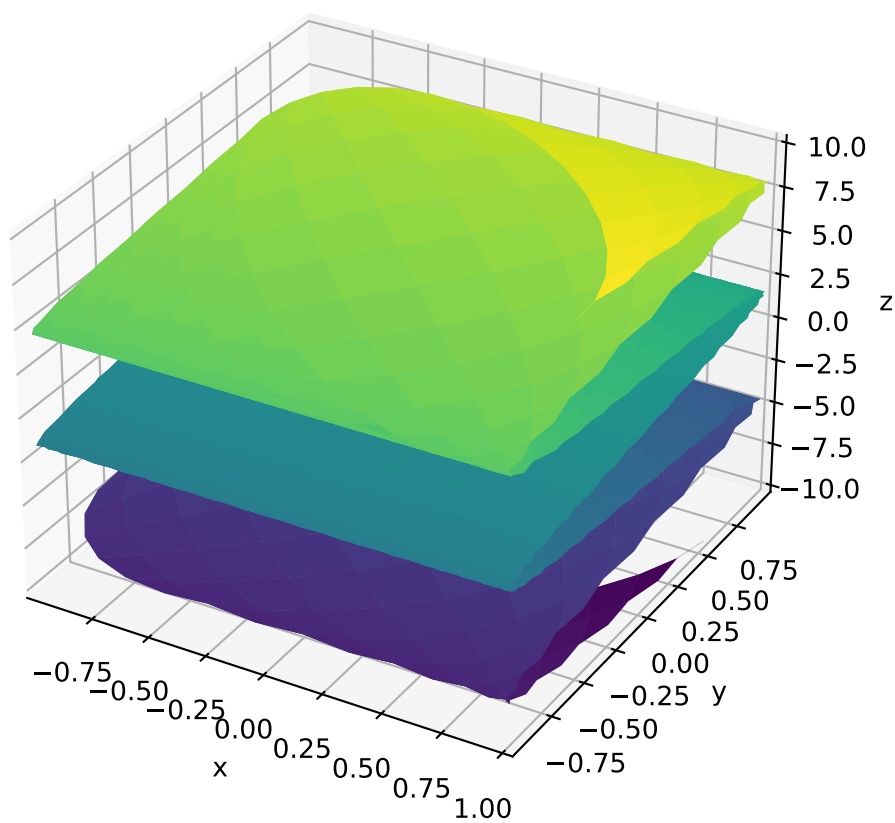
### Arguments

- `expr` : The equation / inequality that is to be plotted.
- `x_var` (optional) : symbol to plot on x-axis or tuple giving symbol and range as (symbol, xmin, xmax)
- `y_var` (optional) : symbol to plot on y-axis or tuple giving symbol and range as (symbol, ymin, ymax)

If neither `x_var` nor `y_var` are given then the free symbols in the expression will be assigned in the order they are sorted.

The following keyword arguments can also be used:

- **adaptive Boolean. The default value is set to True. It has to be** set to False if you want to use a mesh grid.
- **depth integer. The depth of recursion for adaptive mesh grid.** Default value is 0. Takes value in the range (0, 4).





- **points integer.** The number of points if adaptive mesh grid is not used. Default value is 300.
- **show Boolean.** Default value is True. If set to False, the plot will not be shown. See Plot for further information.
- **title string.** The title for the plot.
- **xlabel string.** The label for the x-axis
- **ylabel string.** The label for the y-axis

Aesthetics options:

- **line\_color: float or string.** Specifies the color for the plot.  
See Plot to see how to set color for the plots. Default value is “Blue”

plot\_implicit, by default, uses interval arithmetic to plot functions. If the expression cannot be plotted using interval arithmetic, it defaults to a generating a contour using a mesh grid of fixed number of points. By setting adaptive to False, you can force plot\_implicit to use the mesh grid. The mesh grid method can be effective when adaptive plotting using interval arithmetic, fails to plot with small line width.

## Examples

Plot expressions:

```
>>> from sympy import plot_implicit, symbols, Eq, And
>>> x, y = symbols('x y')
```

Without any ranges for the symbols in the expression:

```
>>> p1 = plot_implicit(Eq(x**2 + y**2, 5))
```

With the range for the symbols:

```
>>> p2 = plot_implicit(
... Eq(x**2 + y**2, 3), (x, -3, 3), (y, -3, 3))
```

With depth of recursion as argument:

```
>>> p3 = plot_implicit(
... Eq(x**2 + y**2, 5), (x, -4, 4), (y, -4, 4), depth = 2)
```

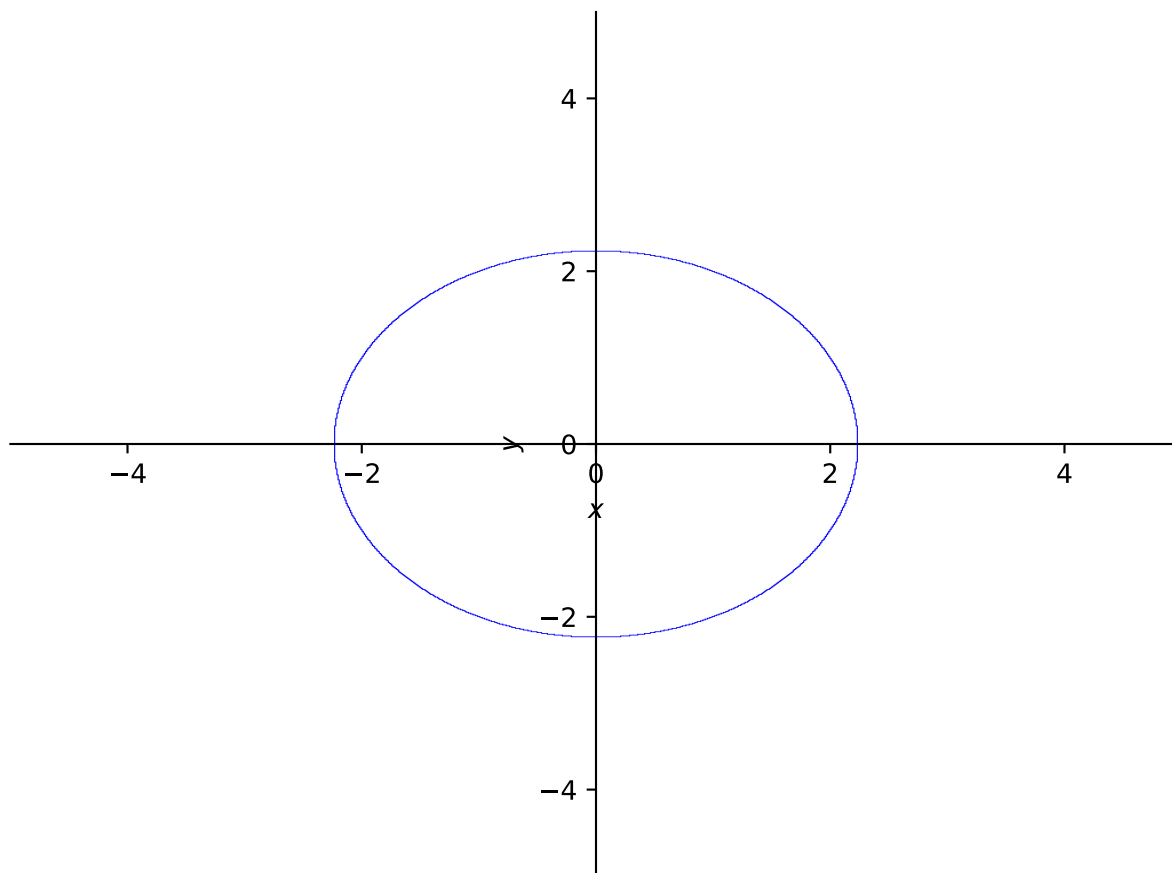
Using mesh grid and not using adaptive meshing:

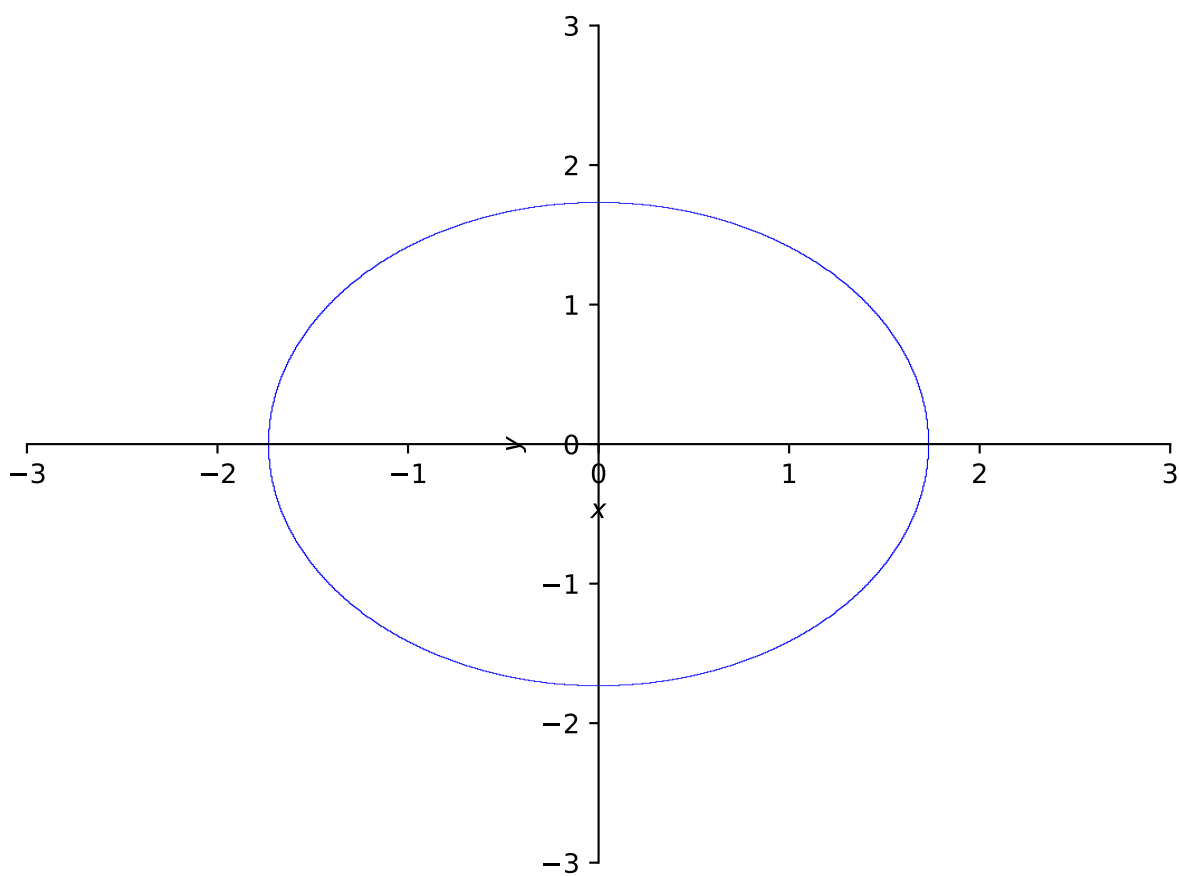
```
>>> p4 = plot_implicit(
... Eq(x**2 + y**2, 5), (x, -5, 5), (y, -2, 2),
... adaptive=False)
```

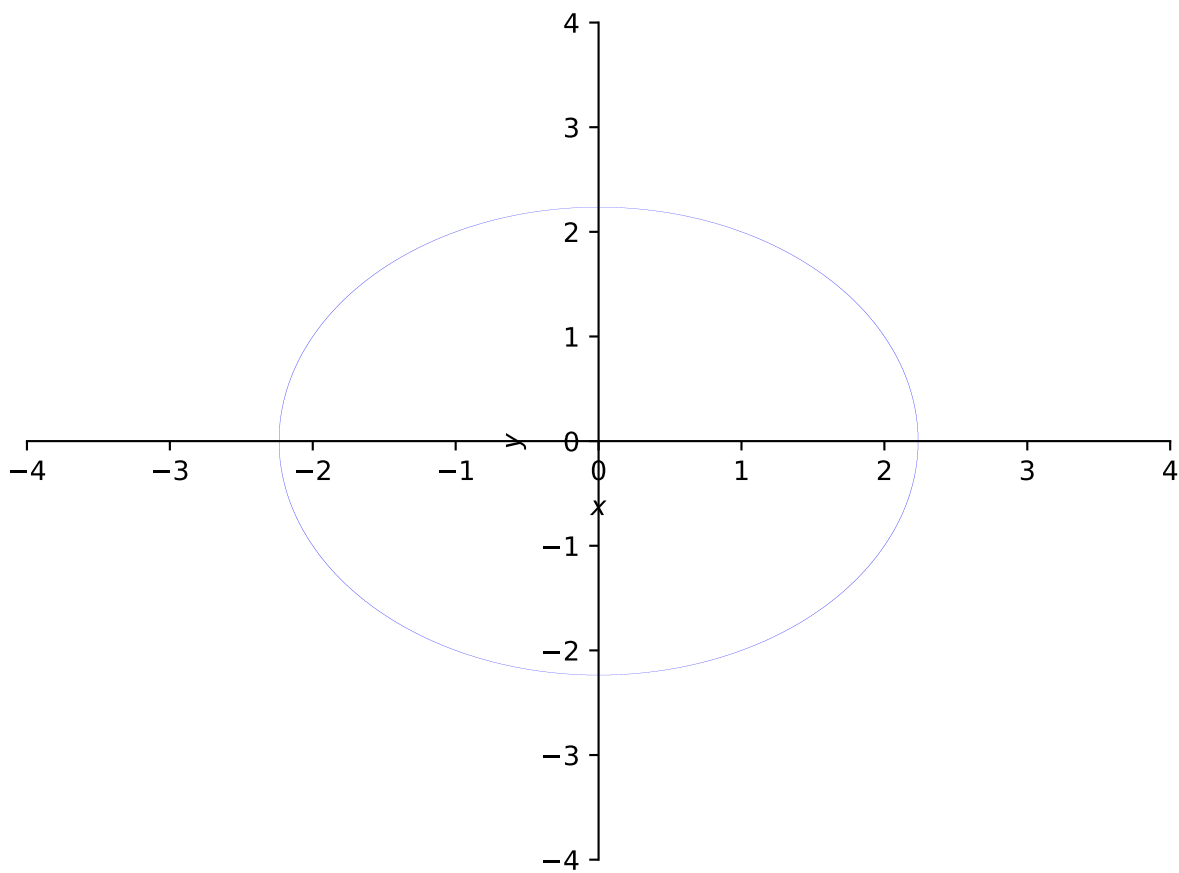
Using mesh grid without using adaptive meshing with number of points specified:

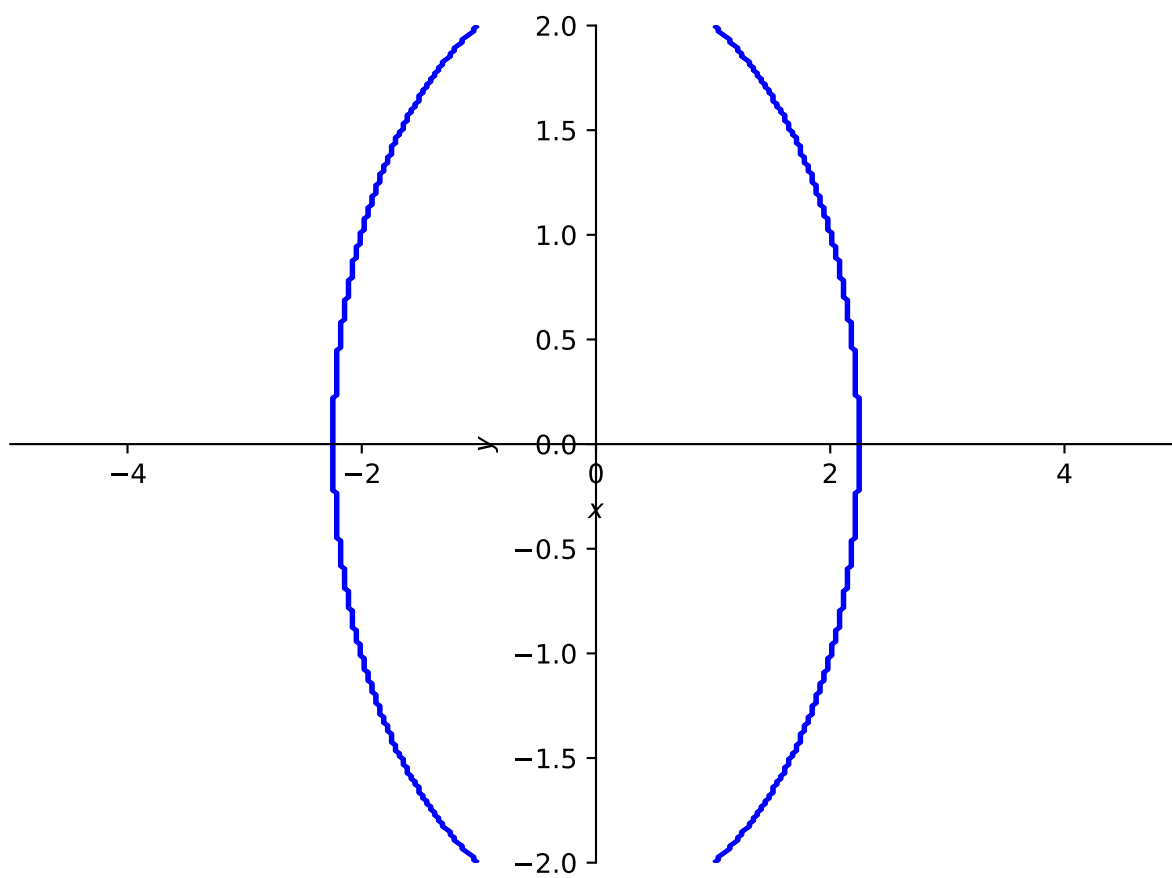
```
>>> p5 = plot_implicit(
... Eq(x**2 + y**2, 5), (x, -5, 5), (y, -2, 2),
... adaptive=False, points=400)
```

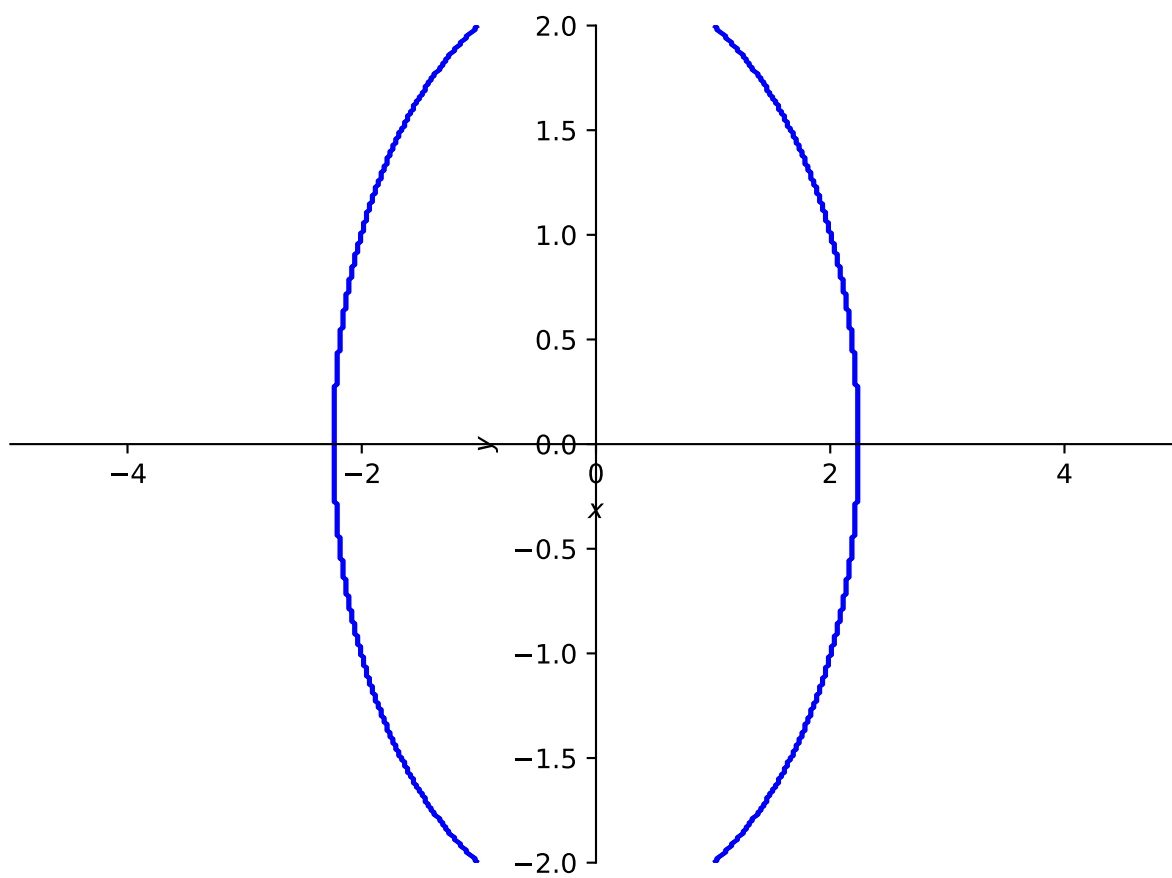
Plotting regions:



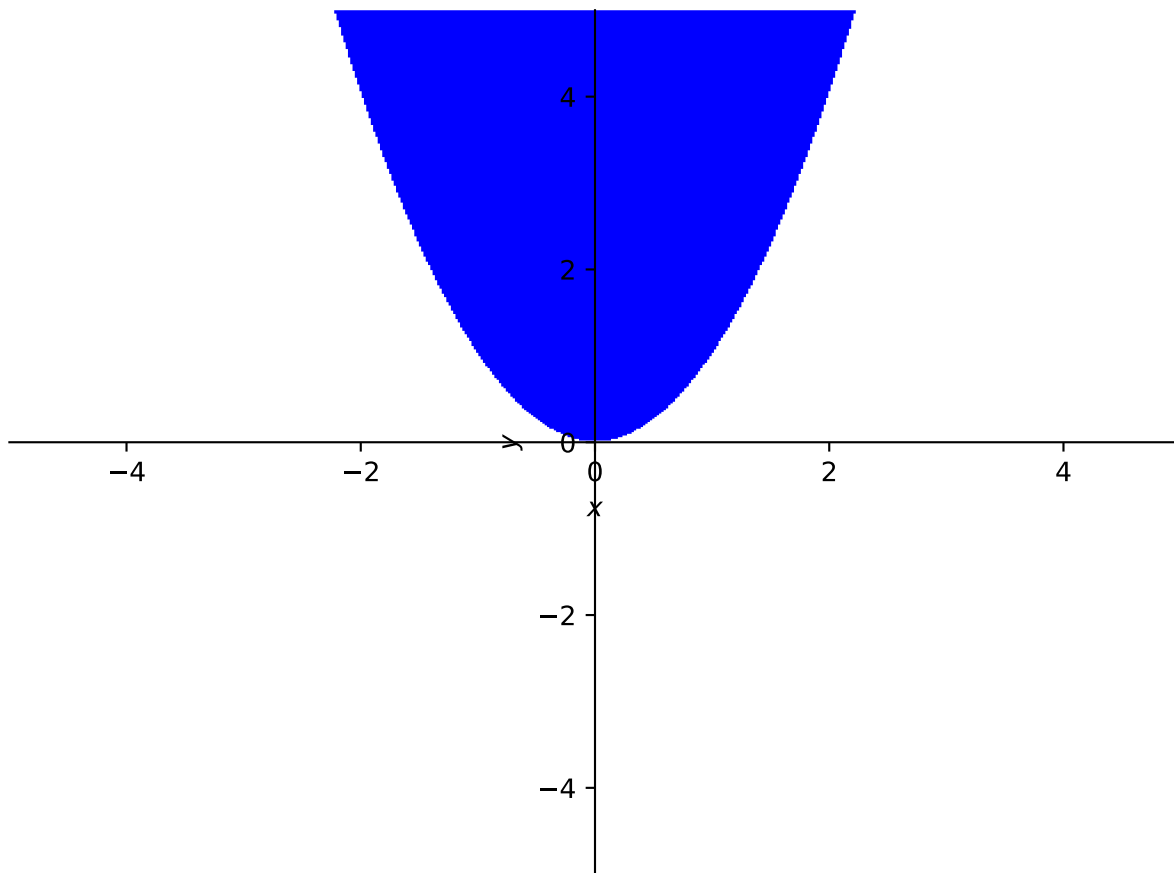








```
>>> p6 = plot_implicit(y > x**2)
```



Plotting Using boolean conjunctions:

```
>>> p7 = plot_implicit(And(y > x, y > -x))
```

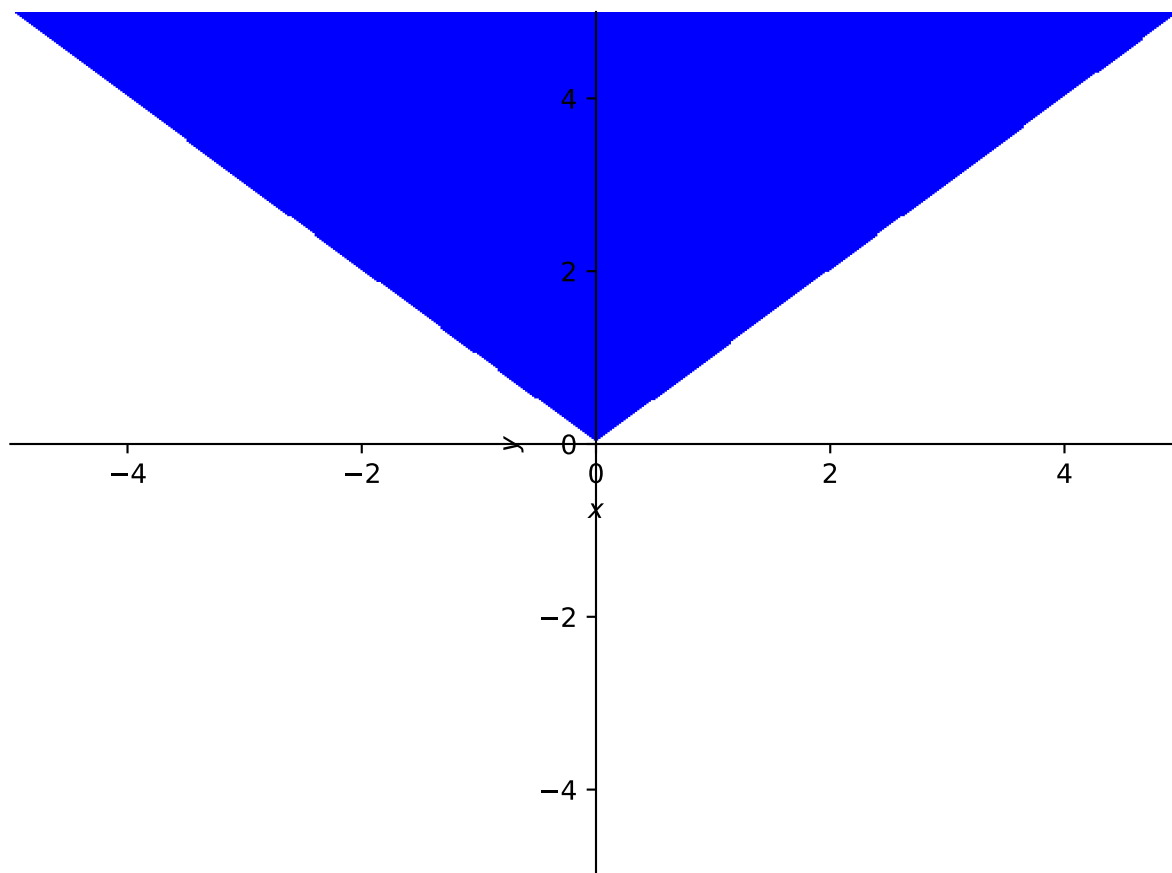
When plotting an expression with a single variable ( $y - 1$ , for example), specify the  $x$  or the  $y$  variable explicitly:

```
>>> p8 = plot_implicit(y - 1, y_var=y)
>>> p9 = plot_implicit(x - 1, x_var=x)
```

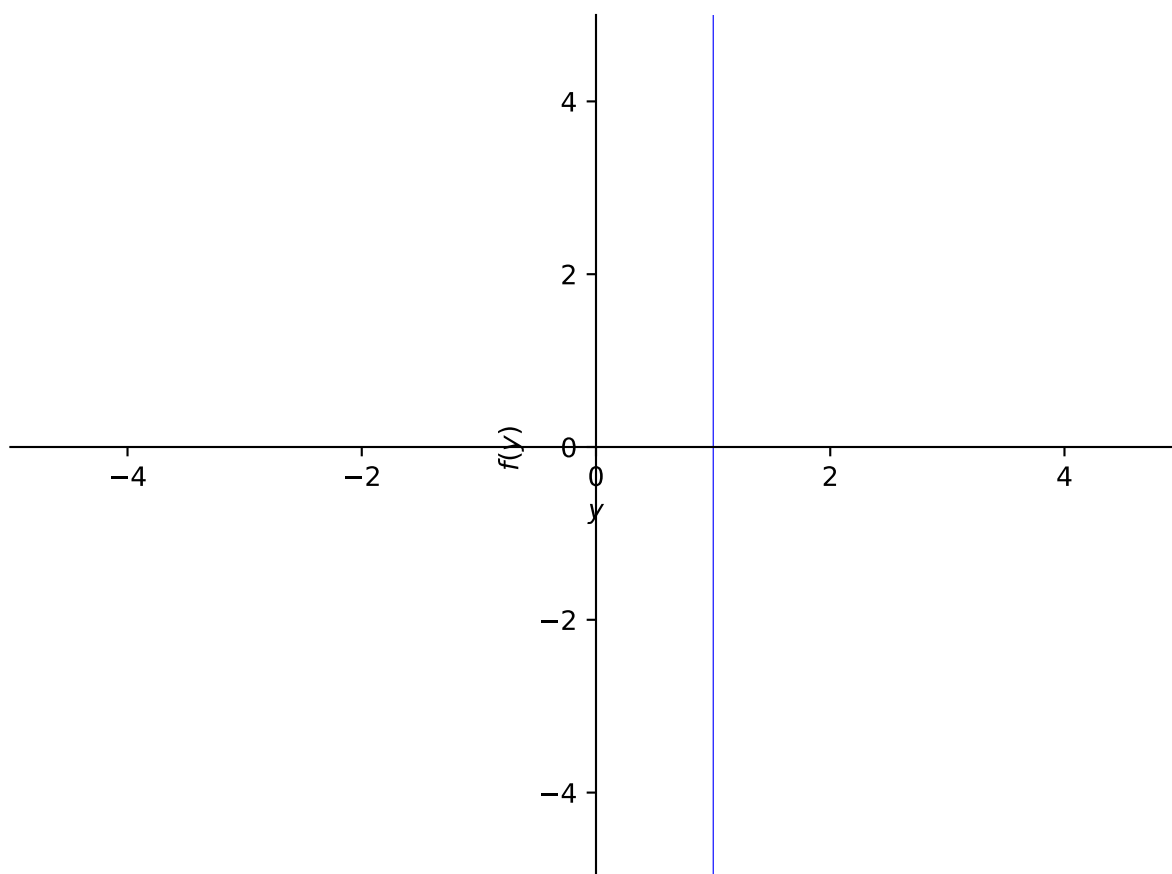
## PlotGrid Class

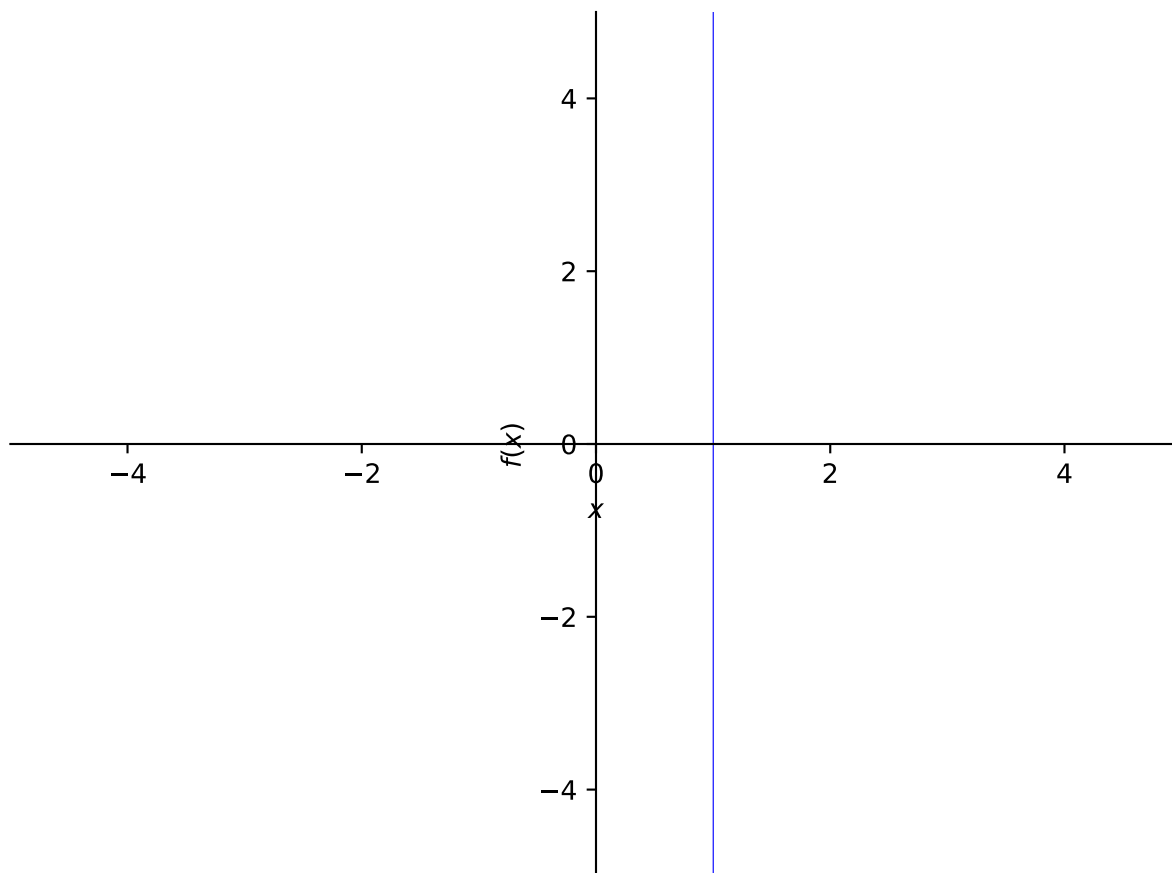
```
class sympy.plotting.plot.PlotGrid(nrows, ncolumns, *args, show=True, size=None,
 **kwargs)
```

This class helps to plot subplots from already created SymPy plots in a single figure.



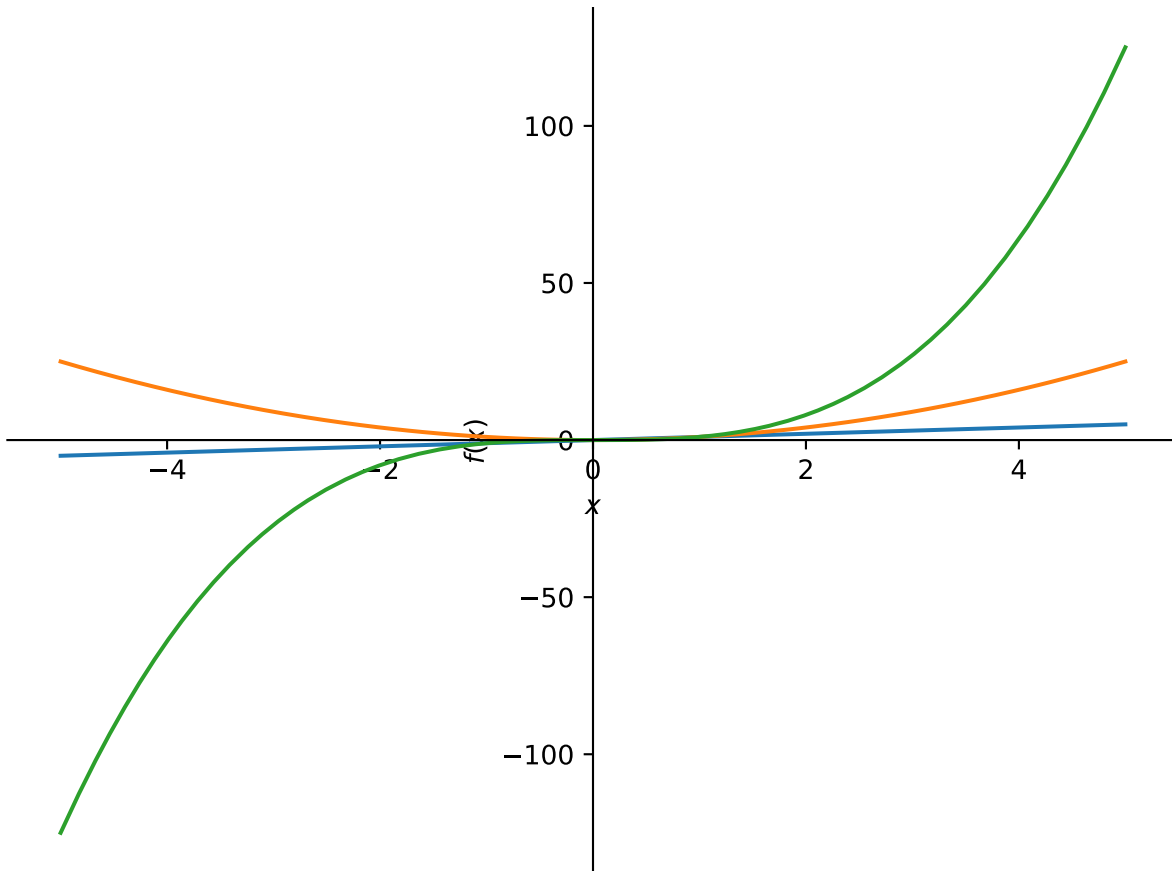






## Examples

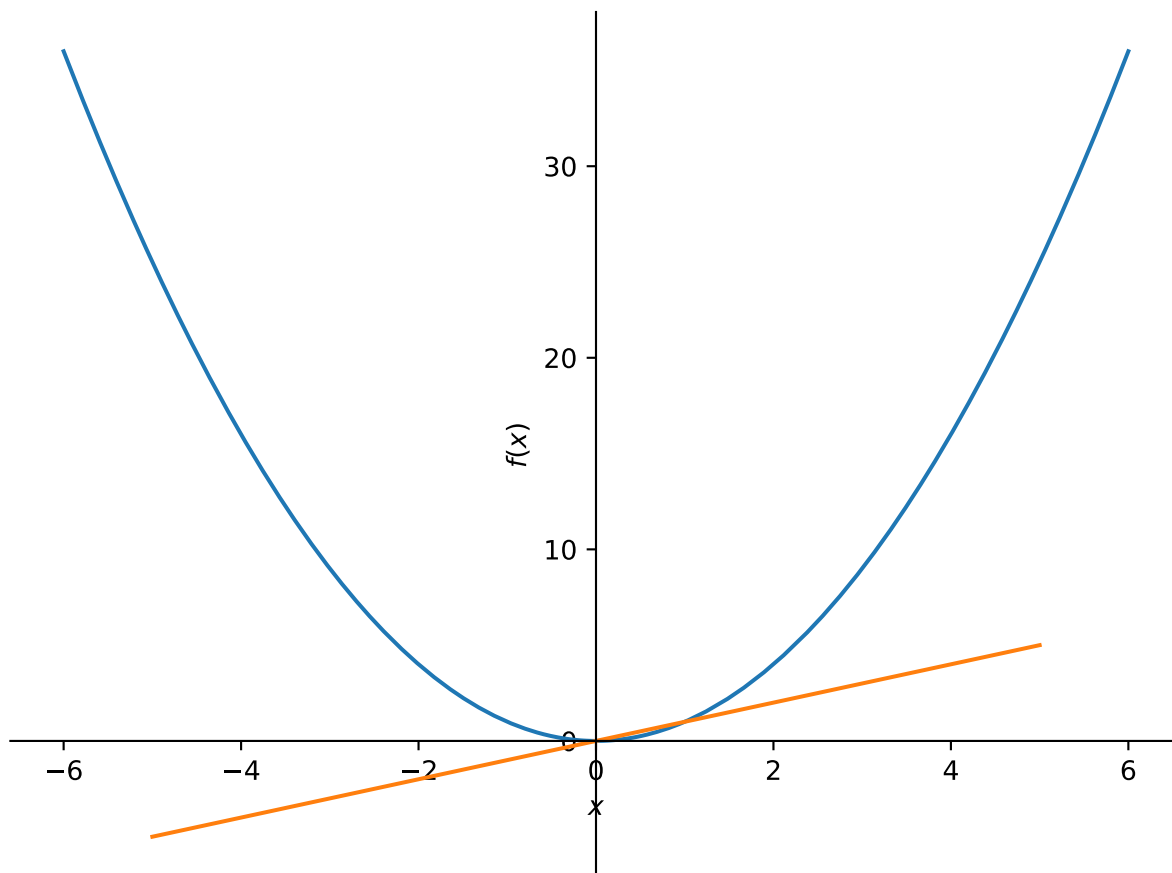
```
>>> from sympy import symbols
>>> from sympy.plotting import plot, plot3d, PlotGrid
>>> x, y = symbols('x, y')
>>> p1 = plot(x, x**2, x**3, (x, -5, 5))
>>> p2 = plot((x**2, (x, -6, 6)), (x, (x, -5, 5)))
>>> p3 = plot(x**3, (x, -5, 5))
>>> p4 = plot3d(x*y, (x, -5, 5), (y, -5, 5))
```

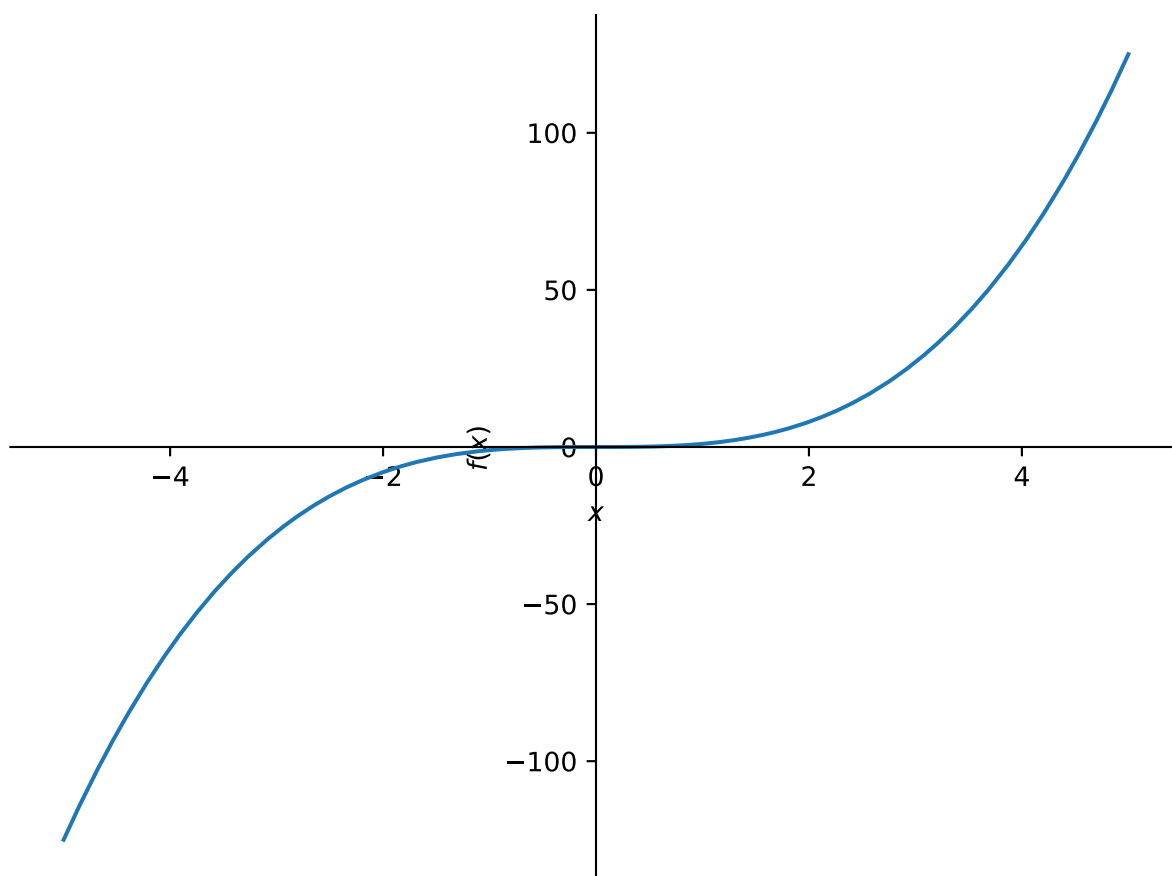


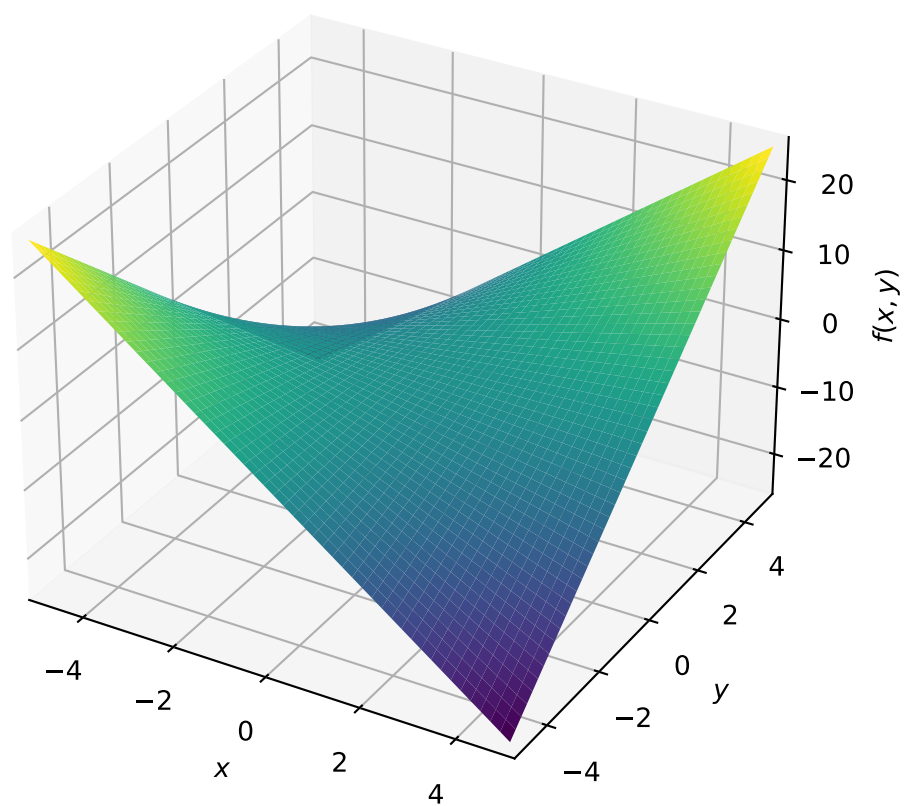
Plotting vertically in a single line:

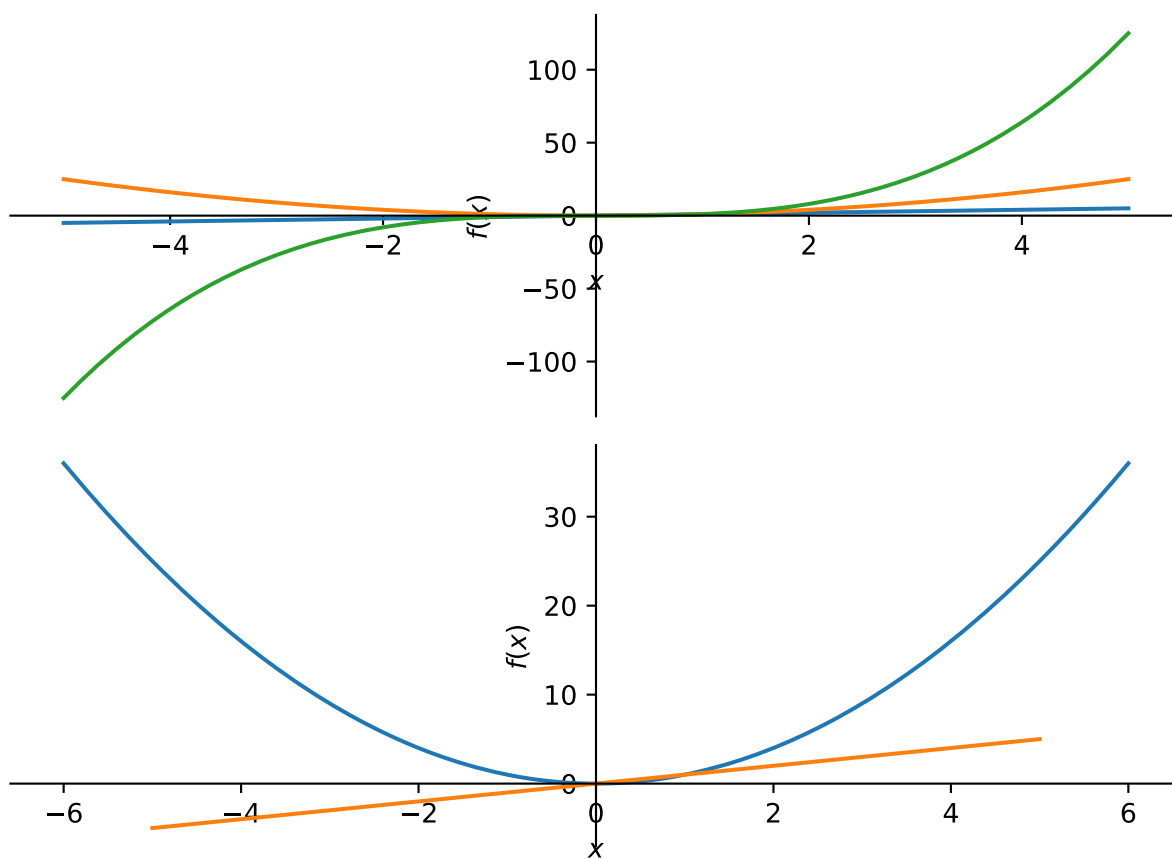
```
>>> PlotGrid(2, 1, p1, p2)
PlotGrid object containing:
Plot[0]:Plot object containing:
[0]: cartesian line: x for x over (-5.0, 5.0)
[1]: cartesian line: x**2 for x over (-5.0, 5.0)
[2]: cartesian line: x**3 for x over (-5.0, 5.0)
Plot[1]:Plot object containing:
[0]: cartesian line: x**2 for x over (-6.0, 6.0)
[1]: cartesian line: x for x over (-5.0, 5.0)
```

Plotting horizontally in a single line:

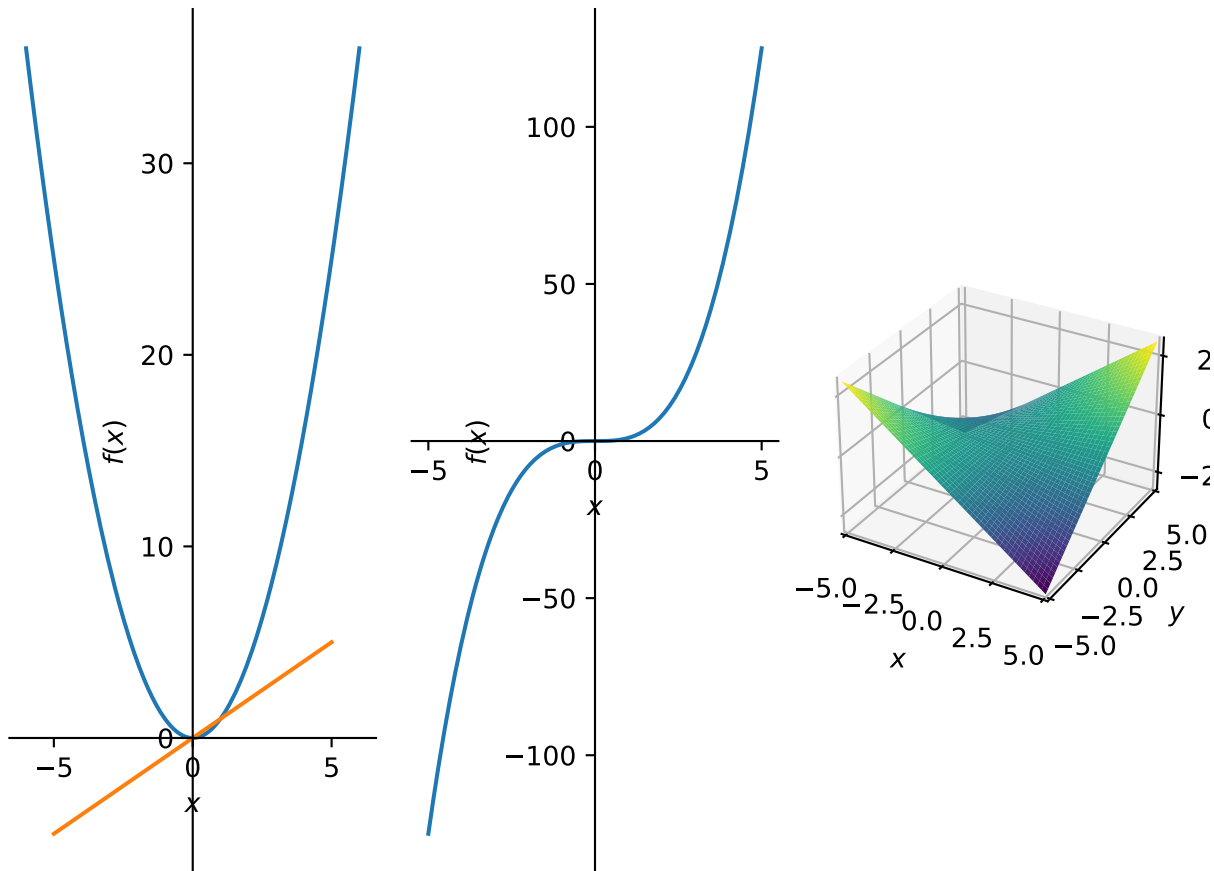








```
>>> PlotGrid(1, 3, p2, p3, p4)
PlotGrid object containing:
Plot[0]:Plot object containing:
[0]: cartesian line: x^2 for x over $(-6.0, 6.0)$
[1]: cartesian line: x for x over $(-5.0, 5.0)$
Plot[1]:Plot object containing:
[0]: cartesian line: x^3 for x over $(-5.0, 5.0)$
Plot[2]:Plot object containing:
[0]: cartesian surface: $x*y$ for x over $(-5.0, 5.0)$ and y over $(-5.0, 5.0)$
```



Plotting in a grid form:

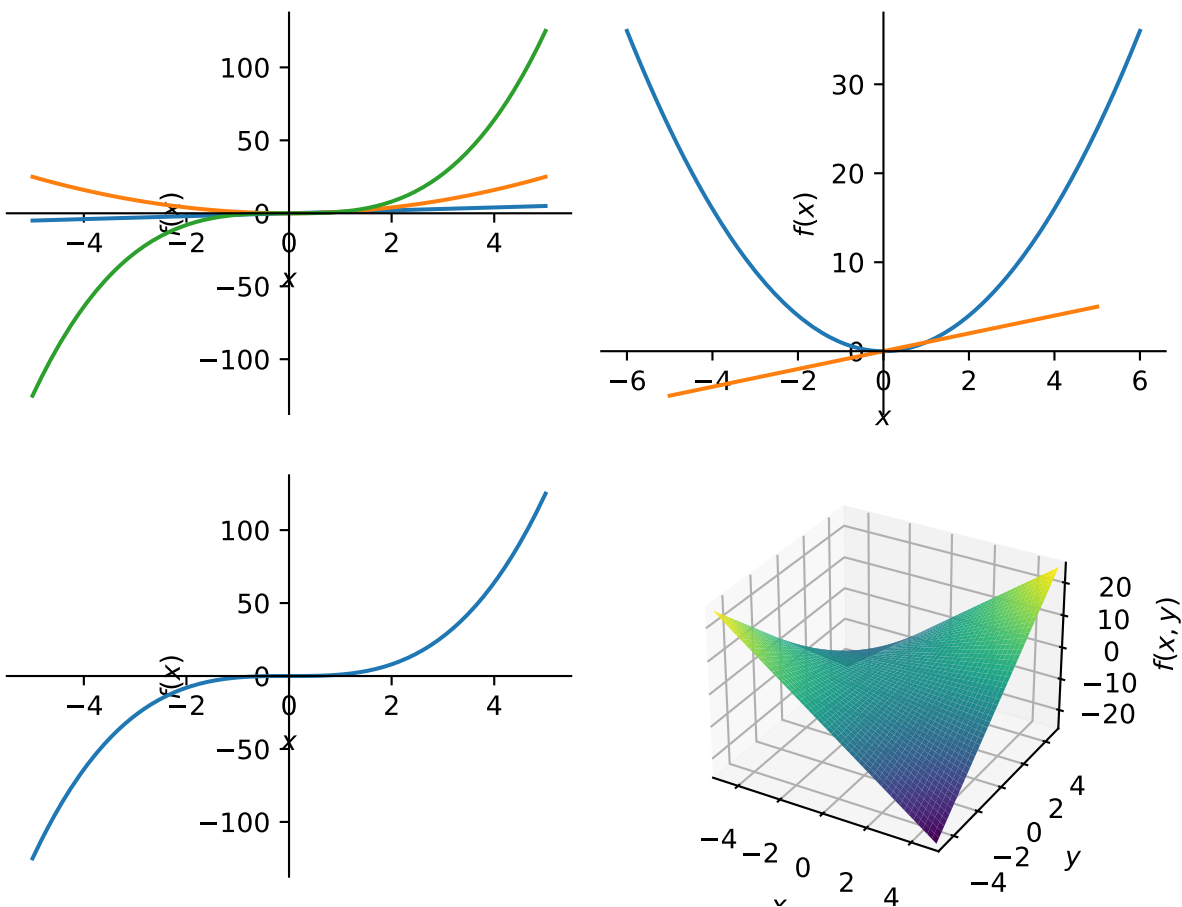
```
>>> PlotGrid(2, 2, p1, p2, p3, p4)
PlotGrid object containing:
Plot[0]:Plot object containing:
[0]: cartesian line: x for x over $(-5.0, 5.0)$
[1]: cartesian line: x^2 for x over $(-5.0, 5.0)$
[2]: cartesian line: x^3 for x over $(-5.0, 5.0)$
Plot[1]:Plot object containing:
[0]: cartesian line: x^2 for x over $(-6.0, 6.0)$
[1]: cartesian line: x for x over $(-5.0, 5.0)$
Plot[2]:Plot object containing:
[0]: cartesian line: x^3 for x over $(-5.0, 5.0)$
```

(continues on next page)



(continued from previous page)

```
Plot[3]:Plot object containing:
[0]: cartesian surface: x*y for x over (-5.0, 5.0) and y over (-5.0, 5.0)
```



## Series Classes

**class** sympy.plotting.plot.BaseSeries

Base class for the data objects containing stuff to be plotted.

## Explanation

The backend should check if it supports the data series that is given. (e.g. TextBackend supports only LineOver1DRangeSeries). It is the backend responsibility to know how to use the class of data series that is given.

Some data series classes are grouped (using a class attribute like `is_2Dline`) according to the api they present (based only on convention). The backend is not obliged to use that api (e.g. `LineOver1DRangeSeries` belongs to the `is_2Dline` group and presents the `get_points` method, but the `TextBackend` does not use the `get_points` method).

**class** sympy.plotting.plot.Line2DBaseSeries

A base class for 2D lines.

- adding the label, steps and only\_integers options
- making is\_2Dline true
- defining get\_segments and get\_color\_array

**get\_data()**

Return lists of coordinates for plotting the line.

**Returns**

**x** : list

List of x-coordinates

**y**

[list] List of y-coordinates

**z**

[list] List of z-coordinates in case of Parametric3DLineSeries

**class** sympy.plotting.plot.LineOver1DRangeSeries(*expr, var\_start\_end, \*\*kwargs*)

Representation for a line consisting of a SymPy expression over a range.

**get\_points()**

Return lists of coordinates for plotting. Depending on the `adaptive` option, this function will either use an adaptive algorithm or it will uniformly sample the expression over the provided range.

**Returns**

**x** : list

List of x-coordinates

**y**

[list] List of y-coordinates

**Explanation**

The adaptive sampling is done by recursively checking if three points are almost collinear. If they are not collinear, then more points are added between those points.

**References**

[R692]

**class** sympy.plotting.plot.Parametric2DLineSeries(*expr\_x, expr\_y, var\_start\_end, \*\*kwargs*)

Representation for a line consisting of two parametric SymPy expressions over a range.

**get\_points()**

Return lists of coordinates for plotting. Depending on the `adaptive` option, this function will either use an adaptive algorithm or it will uniformly sample the expression over the provided range.

**Returns**

**x** : list

List of x-coordinates

**y**

[list] List of y-coordinates

**Explanation**

The adaptive sampling is done by recursively checking if three points are almost collinear. If they are not collinear, then more points are added between those points.

**References**

[R693]

**class** `sympy.plotting.plot.Line3DBaseSeries`

A base class for 3D lines.

Most of the stuff is derived from `Line2DBaseSeries`.

**class** `sympy.plotting.plot.Parametric3DLineSeries(expr_x, expr_y, expr_z, var_start_end, **kwargs)`

Representation for a 3D line consisting of three parametric SymPy expressions and a range.

**class** `sympy.plotting.plot.SurfaceBaseSeries`

A base class for 3D surfaces.

**class** `sympy.plotting.plot.SurfaceOver2DRangeSeries(expr, var_start_end_x, var_start_end_y, **kwargs)`

Representation for a 3D surface consisting of a SymPy expression and 2D range.

**class** `sympy.plotting.plot.ParametricSurfaceSeries(expr_x, expr_y, expr_z, var_start_end_u, var_start_end_v, **kwargs)`

Representation for a 3D surface consisting of three parametric SymPy expressions and a range.

**class** `sympy.plotting.plot_implicit.ImplicitSeries(expr, var_start_end_x, var_start_end_y, has_equality, use_interval_math, depth, nb_of_points, line_color)`

Representation for Implicit plot

## Backends

**class** `sympy.plotting.plot.BaseBackend(parent)`

Base class for all backends. A backend represents the plotting library, which implements the necessary functionalities in order to use SymPy plotting functions.

How the plotting module works:

1. **Whenever a plotting function is called, the provided expressions are** processed and a list of instances of the [BaseSeries](#) (page 2865) class is created, containing the necessary information to plot the expressions (e.g. the expression, ranges, series name, ...). Eventually, these objects will generate the numerical data to be plotted.
2. **A [Plot](#) (page 2821) object is instantiated, which stores the list of** series and the main attributes of the plot (e.g. axis labels, title, ...).
3. **When the show command is executed, a new backend is instantiated,** which loops through each series object to generate and plot the numerical data. The backend is also going to set the axis labels, title, ..., according to the values stored in the Plot instance.

The backend should check if it supports the data series that it is given (e.g. [TextBackend](#) (page 2869) supports only [LineOver1DRangeSeries](#) (page 2866)).

It is the backend responsibility to know how to use the class of data series that it's given. Note that the current implementation of the \*Series classes is "matplotlib-centric": the numerical data returned by the `get_points` and `get_meshes` methods is meant to be used directly by Matplotlib. Therefore, the new backend will have to pre-process the numerical data to make it compatible with the chosen plotting library. Keep in mind that future SymPy versions may improve the \*Series classes in order to return numerical data "non-matplotlib-centric", hence if you code a new backend you have the responsibility to check if its working on each SymPy release.

Please explore the [MatplotlibBackend](#) (page 2868) source code to understand how a backend should be coded.

**See also:**

[MatplotlibBackend](#) (page 2868)

**class** `sympy.plotting.plot.MatplotlibBackend(parent)`

This class implements the functionalities to use Matplotlib with SymPy plotting functions.

**static** `get_segments(x, y, z=None)`

Convert two list of coordinates to a list of segments to be used with Matplotlib's [LineCollection](#).

**Parameters**

**x** : list

List of x-coordinates

**y**

[list] List of y-coordinates

**z**

[list] List of z-coordinates for a 3D line.

**process\_series()**

Iterates over every Plot object and further calls `_process_series()`

**class** `sympy.plotting.plot.TextBackend(parent)`

**Pyglet Plotting**

This is the documentation for the old plotting module that uses pyglet. This module has some limitations and is not actively developed anymore. For an alternative you can look at the new plotting module.

The pyglet plotting module can do nice 2D and 3D plots that can be controlled by console commands as well as keyboard and mouse, with the only dependency being [pyglet](#).

Here is the simplest usage:

```
>>> from sympy import var
>>> from sympy.plotting.pygletplot import PygletPlot as Plot
>>> var('x y z')
>>> Plot(x*y**3-y*x**3)
```

To see lots of plotting examples, see `examples/pyglet_plotting.py` and try running it in interactive mode (`python -i plotting.py`):

```
$ python -i examples/pyglet_plotting.py
```

And type for instance `example(7)` or `example(11)`.

See also the [Plotting Module](#) wiki page for screenshots.

**Plot Window Controls**

| Camera                 | Keys                                      |
|------------------------|-------------------------------------------|
| Sensitivity Modifier   | SHIFT                                     |
| Zoom                   | R and F, Page Up and Down, Numpad + and - |
| Rotate View X,Y axis   | Arrow Keys, A,S,D,W, Numpad 4,6,8,2       |
| Rotate View Z axis     | Q and E, Numpad 7 and 9                   |
| Rotate Ordinate Z axis | Z and C, Numpad 1 and 3                   |
| View XY                | F1                                        |
| View XZ                | F2                                        |
| View YZ                | F3                                        |
| View Perspective       | F4                                        |
| Reset                  | X, Numpad 5                               |

| Axes           | Keys |
|----------------|------|
| Toggle Visible | F5   |
| Toggle Colors  | F6   |

| Window     | Keys   |
|------------|--------|
| Close      | ESCAPE |
| Screenshot | F8     |

The mouse can be used to rotate, zoom, and translate by dragging the left, middle, and right mouse buttons respectively.

## Coordinate Modes

Plot supports several curvilinear coordinate modes, and they are independent for each plotted function. You can specify a coordinate mode explicitly with the 'mode' named argument, but it can be automatically determined for cartesian or parametric plots, and therefore must only be specified for polar, cylindrical, and spherical modes.

Specifically, `Plot(function arguments)` and `Plot.__setitem__(i, function arguments)` (accessed using array-index syntax on the Plot instance) will interpret your arguments as a cartesian plot if you provide one function and a parametric plot if you provide two or three functions. Similarly, the arguments will be interpreted as a curve if one variable is used, and a surface if two are used.

Supported mode names by number of variables:

- 1 (curves): parametric, cartesian, polar
- 2 (surfaces): parametric, cartesian, cylindrical, spherical

```
>>> Plot(1, 'mode=spherical; color=zfade4')
```

Note that function parameters are given as option strings of the form "key1=value1; key2 = value2" (spaces are truncated). Keyword arguments given directly to plot apply to the plot itself.

## Specifying Intervals for Variables

The basic format for variable intervals is [var, min, max, steps]. However, the syntax is quite flexible, and arguments not specified are taken from the defaults for the current coordinate mode:

```
>>> Plot(x**2) # implies [x, -5, 5, 100]
>>> Plot(x**2, [], []) # [x, -1, 1, 40], [y, -1, 1, 40]
>>> Plot(x**2-y**2, [100], [100]) # [x, -1, 1, 100], [y, -1, 1, 100]
>>> Plot(x**2, [x, -13, 13, 100])
>>> Plot(x**2, [-13, 13]) # [x, -13, 13, 100]
>>> Plot(x**2, [x, -13, 13]) # [x, -13, 13, 100]
>>> Plot(1*x, [], [x], 'mode=cylindrical') # [unbound_theta, 0, 2*Pi, 40], [x, -1,
↪ 1, 20]
```

## Using the Interactive Interface

```
>>> p = Plot(visible=False)
>>> f = x**2
>>> p[1] = f
>>> p[2] = f.diff(x)
>>> p[3] = f.diff(x).diff(x)
>>> p
[1]: x**2, 'mode=cartesian'
[2]: 2*x, 'mode=cartesian'
[3]: 2, 'mode=cartesian'
>>> p.show()
>>> p.clear()
>>> p
<blank plot>
>>> p[1] = x**2+y**2
>>> p[1].style = 'solid'
>>> p[2] = -x**2-y**2
>>> p[2].style = 'wireframe'
>>> p[1].color = z, (0.4,0.4,0.9), (0.9,0.4,0.4)
>>> p[1].style = 'both'
>>> p[2].style = 'both'
>>> p.close()
```

## Using Custom Color Functions

The following code plots a saddle and color it by the magnitude of its gradient:

```
>>> fz = x**2-y**2
>>> Fx, Fy, Fz = fz.diff(x), fz.diff(y), 0
>>> p[1] = fz, 'style=solid'
>>> p[1].color = (Fx**2 + Fy**2 + Fz**2)**(0.5)
```

The coloring algorithm works like this:

1. Evaluate the color function(s) across the curve or surface.
2. Find the minimum and maximum value of each component.
3. Scale each component to the color gradient.

When not specified explicitly, the default color gradient is  $f(0.0) = (0.4, 0.4, 0.4) \rightarrow f(1.0) = (0.9, 0.9, 0.9)$ . In our case, everything is gray-scale because we have applied the default color gradient uniformly for each color component. When defining a color scheme in this way, you might want to supply a color gradient as well:

```
>>> p[1].color = (Fx**2 + Fy**2 + Fz**2)**(0.5), (0.1,0.1,0.9), (0.9,0.1,0.1)
```

Here's a color gradient with four steps:

```
>>> gradient = [0.0, (0.1,0.1,0.9), 0.3, (0.1,0.9,0.1),
... 0.7, (0.9,0.9,0.1), 1.0, (1.0,0.0,0.0)]
>>> p[1].color = (Fx**2 + Fy**2 + Fz**2)**(0.5), gradient
```

The other way to specify a color scheme is to give a separate function for each component r, g, b. With this syntax, the default color scheme is defined:

```
>>> p[1].color = z,y,x, (0.4,0.4,0.4), (0.9,0.9,0.9)
```

This maps z->red, y->green, and x->blue. In some cases, you might prefer to use the following alternative syntax:

```
>>> p[1].color = z,(0.4,0.9), y,(0.4,0.9), x,(0.4,0.9)
```

You can still use multi-step gradients with three-function color schemes.

## Plotting Geometric Entities

The plotting module is capable of plotting some 2D geometric entities like line, circle and ellipse. The following example plots a circle centred at origin and of radius 2 units.

```
>>> from sympy import *
>>> x,y = symbols('x y')
>>> plot_implicit(Eq(x**2+y**2, 4))
```

Similarly, `plot_implicit()` (page 2847) may be used to plot any 2-D geometric structure from its implicit equation.

Plotting polygons (Polygon, RegularPolygon, Triangle) are not supported directly.

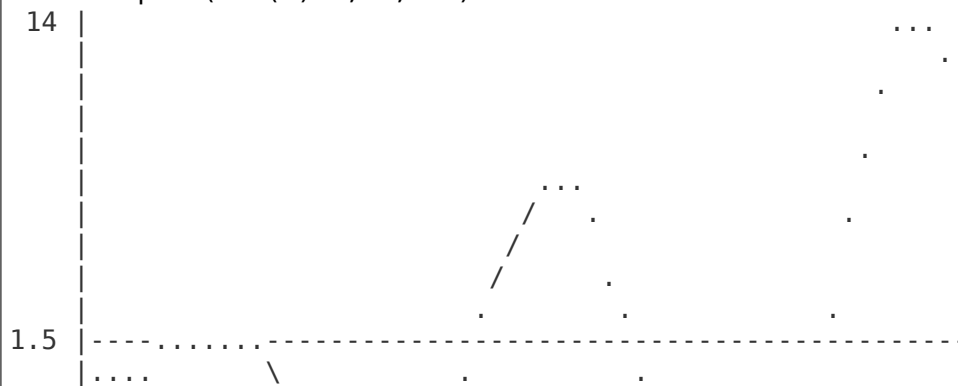
## Plotting with ASCII art

`sympy.plotting.textplot.textplot(expr, a, b, W=55, H=21)`

Print a crude ASCII art plot of the SymPy expression 'expr' (which should contain a single symbol, e.g. x or something else) over the interval [a, b].

### Examples

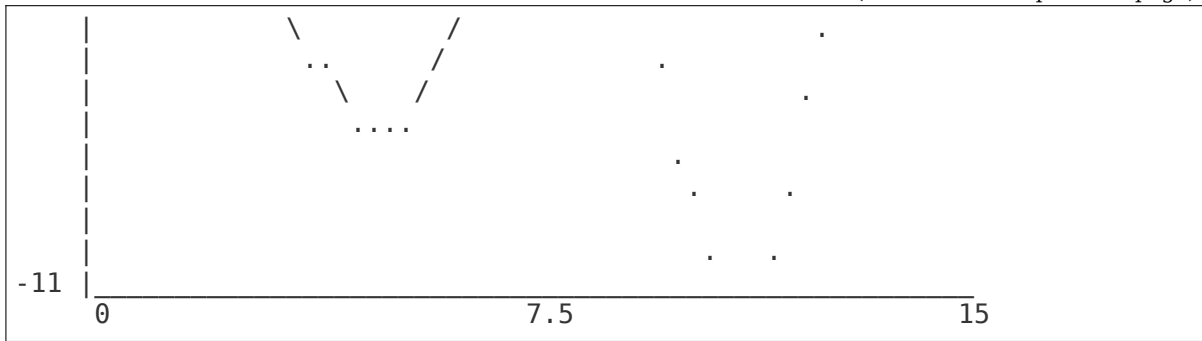
```
>>> from sympy import Symbol, sin
>>> from sympy.plotting import textplot
>>> t = Symbol('t')
>>> textplot(sin(t)*t, 0, 15)
```



(continues on next page)



(continued from previous page)



## Stats

SymPy statistics module

Introduces a random variable type into the SymPy language.

Random variables may be declared using prebuilt functions such as Normal, Exponential, Coin, Die, etc... or built with functions like FiniteRV.

Queries on random expressions can be made using the functions

| Expression                           | Meaning                      |
|--------------------------------------|------------------------------|
| $P(\text{condition})$                | Probability                  |
| $E(\text{expression})$               | Expected value               |
| $H(\text{expression})$               | Entropy                      |
| $\text{variance}(\text{expression})$ | Variance                     |
| $\text{density}(\text{expression})$  | Probability Density Function |
| $\text{sample}(\text{expression})$   | Produce a realization        |
| $\text{where}(\text{condition})$     | Where the condition is true  |

## Examples

```
>>> from sympy.stats import P, E, variance, Die, Normal
>>> from sympy import simplify
>>> X, Y = Die('X', 6), Die('Y', 6) # Define two six sided dice
>>> Z = Normal('Z', 0, 1) # Declare a Normal random variable with mean 0, std.
 ↪ 1
>>> P(X>3) # Probability X is greater than 3
1/2
>>> E(X+Y) # Expectation of the sum of two dice
7
>>> variance(X+Y) # Variance of the sum of two dice
35/6
>>> simplify(P(Z>1)) # Probability of Z being greater than 1
1/2 - erf(sqrt(2)/2)/2
```

One could also create custom distribution and define custom random variables as follows:

1. If you want to create a Continuous Random Variable:

```
>>> from sympy.stats import ContinuousRV, P, E
>>> from sympy import exp, Symbol, Interval, oo
>>> x = Symbol('x')
>>> pdf = exp(-x) # pdf of the Continuous Distribution
>>> Z = ContinuousRV(x, pdf, set=Interval(0, oo))
>>> E(Z)
1
>>> P(Z > 5)
exp(-5)
```

1.1 To create an instance of Continuous Distribution:

```
>>> from sympy.stats import ContinuousDistributionHandmade
>>> from sympy import Lambda
>>> dist = ContinuousDistributionHandmade(Lambda(x, pdf), set=Interval(0, oo))
>>> dist.pdf(x)
exp(-x)
```

2. If you want to create a Discrete Random Variable:

```
>>> from sympy.stats import DiscreteRV, P, E
>>> from sympy import Symbol, S
>>> p = S(1)/2
>>> x = Symbol('x', integer=True, positive=True)
>>> pdf = p*(1 - p)**(x - 1)
>>> D = DiscreteRV(x, pdf, set=S.Naturals)
>>> E(D)
2
>>> P(D > 3)
1/8
```

2.1 To create an instance of Discrete Distribution:

```
>>> from sympy.stats import DiscreteDistributionHandmade
>>> from sympy import Lambda
>>> dist = DiscreteDistributionHandmade(Lambda(x, pdf), set=S.Naturals)
>>> dist.pdf(x)
2**(1 - x)/2
```

3. If you want to create a Finite Random Variable:

```
>>> from sympy.stats import FiniteRV, P, E
>>> from sympy import Rational, Eq
>>> pmf = {1: Rational(1, 3), 2: Rational(1, 6), 3: Rational(1, 4), 4:
↳ Rational(1, 4)}
>>> X = FiniteRV('X', pmf)
>>> E(X)
29/12
>>> P(X > 3)
1/4
```

3.1 To create an instance of Finite Distribution:

```
>>> from sympy.stats import FiniteDistributionHandmade
>>> dist = FiniteDistributionHandmade(pmf)
>>> dist.pmf(x)
Lambda(x, Piecewise((1/3, Eq(x, 1)), (1/6, Eq(x, 2)), (1/4, Eq(x, 3) | Eq(x, 4)), (0, True)))
```

## Random Variable Types

### Finite Types

`sympy.stats.DiscreteUniform(name, items)`

Create a Finite Random Variable representing a uniform distribution over the input set.

#### Parameters

**items** : list/tuple

Items over which Uniform distribution is to be made

#### Returns

RandomSymbol

### Examples

```
>>> from sympy.stats import DiscreteUniform, density
>>> from sympy import symbols
```

```
>>> X = DiscreteUniform('X', symbols('a b c')) # equally likely over a, b, c
>>> density(X).dict
{a: 1/3, b: 1/3, c: 1/3}
```

```
>>> Y = DiscreteUniform('Y', list(range(5))) # distribution over a range
>>> density(Y).dict
{0: 1/5, 1: 1/5, 2: 1/5, 3: 1/5, 4: 1/5}
```

### References

[R803], [R804]

`sympy.stats.Die(name, sides=6)`

Create a Finite Random Variable representing a fair die.

#### Parameters

**sides** : Integer

Represents the number of sides of the Die, by default is 6

#### Returns

RandomSymbol

## Examples

```
>>> from sympy.stats import Die, density
>>> from sympy import Symbol
```

```
>>> D6 = Die('D6', 6) # Six sided Die
>>> density(D6).dict
{1: 1/6, 2: 1/6, 3: 1/6, 4: 1/6, 5: 1/6, 6: 1/6}
```

```
>>> D4 = Die('D4', 4) # Four sided Die
>>> density(D4).dict
{1: 1/4, 2: 1/4, 3: 1/4, 4: 1/4}
```

```
>>> n = Symbol('n', positive=True, integer=True)
>>> Dn = Die('Dn', n) # n sided Die
>>> density(Dn).dict
Density(DieDistribution(n))
>>> density(Dn).dict.subs(n, 4).doit()
{1: 1/4, 2: 1/4, 3: 1/4, 4: 1/4}
```

`sympy.stats.Bernoulli(name, p, succ=1, fail=0)`

Create a Finite Random Variable representing a Bernoulli process.

### Parameters

**p** : Rational number between 0 and 1

Represents probability of success

**succ** : Integer/symbol/string

Represents event of success

**fail** : Integer/symbol/string

Represents event of failure

### Returns

RandomSymbol

## Examples

```
>>> from sympy.stats import Bernoulli, density
>>> from sympy import S
```

```
>>> X = Bernoulli('X', S(3)/4) # 1-0 Bernoulli variable, probability = 3/
↪4
>>> density(X).dict
{0: 1/4, 1: 3/4}
```

```
>>> X = Bernoulli('X', S.Half, 'Heads', 'Tails') # A fair coin toss
>>> density(X).dict
{Heads: 1/2, Tails: 1/2}
```

## References

[R805], [R806]

`sympy.stats.Coin(name, p=1 / 2)`

Create a Finite Random Variable representing a Coin toss.

### Parameters

**p** : Rational Numeber between 0 and 1

Represents probability of getting “Heads”, by default is Half

### Returns

RandomSymbol

## Examples

```
>>> from sympy.stats import Coin, density
>>> from sympy import Rational
```

```
>>> C = Coin('C') # A fair coin toss
>>> density(C).dict
{H: 1/2, T: 1/2}
```

```
>>> C2 = Coin('C2', Rational(3, 5)) # An unfair coin
>>> density(C2).dict
{H: 3/5, T: 2/5}
```

### See also:

[`sympy.stats.Binomial`](#) (page 2877)

## References

[R807]

`sympy.stats.Binomial(name, n, p, succ=1, fail=0)`

Create a Finite Random Variable representing a binomial distribution.

### Parameters

**n** : Positive Integer

Represents number of trials

**p** : Rational Number between 0 and 1

Represents probability of success

**succ** : Integer/symbol/string

Represents event of success, by default is 1

**fail** : Integer/symbol/string

Represents event of failure, by default is 0

### Returns

RandomSymbol

## Examples

```
>>> from sympy.stats import Binomial, density
>>> from sympy import S, Symbol
```

```
>>> X = Binomial('X', 4, S.Half) # Four "coin flips"
>>> density(X).dict
{0: 1/16, 1: 1/4, 2: 3/8, 3: 1/4, 4: 1/16}
```

```
>>> n = Symbol('n', positive=True, integer=True)
>>> p = Symbol('p', positive=True)
>>> X = Binomial('X', n, S.Half) # n "coin flips"
>>> density(X).dict
Density(BinomialDistribution(n, 1/2, 1, 0))
>>> density(X).dict.subs(n, 4).doit()
{0: 1/16, 1: 1/4, 2: 3/8, 3: 1/4, 4: 1/16}
```

## References

[R808], [R809]

`sympy.stats.BetaBinomial(name, n, alpha, beta)`

Create a Finite Random Variable representing a Beta-binomial distribution.

### Parameters

**n** : Positive Integer

Represents number of trials

**alpha** : Real positive number

**beta** : Real positive number

### Returns

RandomSymbol

## Examples

```
>>> from sympy.stats import BetaBinomial, density
```

```
>>> X = BetaBinomial('X', 2, 1, 1)
>>> density(X).dict
{0: 1/3, 1: 2*beta(2, 2), 2: 1/3}
```

## References

[R810], [R811]

`sympy.stats.Hypergeometric(name, N, m, n)`

Create a Finite Random Variable representing a hypergeometric distribution.

### Parameters

**N** : Positive Integer

Represents finite population of size N.

**m** : Positive Integer

Represents number of trials with required feature.

**n** : Positive Integer

Represents numbers of draws.

### Returns

RandomSymbol

## Examples

```
>>> from sympy.stats import Hypergeometric, density
```

```
>>> X = Hypergeometric('X', 10, 5, 3) # 10 marbles, 5 white (success), 3
↳ draws
>>> density(X).dict
{0: 1/12, 1: 5/12, 2: 5/12, 3: 1/12}
```

## References

[R812], [R813]

`sympy.stats.FiniteRV(name, density, **kwargs)`

Create a Finite Random Variable given a dict representing the density.

### Parameters

**name** : Symbol

Represents name of the random variable.

**density** : dict

Dictionary containing the pdf of finite distribution

**check** : bool

If True, it will check whether the given density integrates to 1 over the given set. If False, it will not perform this check. Default is False.

### Returns

RandomSymbol

### Examples

```
>>> from sympy.stats import FiniteRV, P, E
```

```
>>> density = {0: .1, 1: .2, 2: .3, 3: .4}
>>> X = FiniteRV('X', density)
```

```
>>> E(X)
2.0000000000000000
>>> P(X >= 2)
0.7000000000000000
```

`sympy.stats.Rademacher(name)`

Create a Finite Random Variable representing a Rademacher distribution.

#### Returns

RandomSymbol

### Examples

```
>>> from sympy.stats import Rademacher, density
```

```
>>> X = Rademacher('X')
>>> density(X).dict
{-1: 1/2, 1: 1/2}
```

#### See also:

[`sympy.stats.Bernoulli`](#) (page 2876)

### References

[R814]

## Discrete Types

`sympy.stats.Geometric(name, p)`

Create a discrete random variable with a Geometric distribution.

#### Parameters

**p** : A probability between 0 and 1

#### Returns

RandomSymbol



## Explanation

The density of the Geometric distribution is given by

$$f(k) := p(1 - p)^{k-1}$$

## Examples

```
>>> from sympy.stats import Geometric, density, E, variance
>>> from sympy import Symbol, S
```

```
>>> p = S.One / 5
>>> z = Symbol("z")
```

```
>>> X = Geometric("x", p)
```

```
>>> density(X)(z)
(4/5)**(z - 1)/5
```

```
>>> E(X)
5
```

```
>>> variance(X)
20
```

## References

[R815], [R816]

`sympy.stats.Hermite(name, a1, a2)`

Create a discrete random variable with a Hermite distribution.

### Parameters

**a1** : A Positive number greater than equal to 0.

**a2** : A Positive number greater than equal to 0.

### Returns

RandomSymbol

## Explanation

The density of the Hermite distribution is given by

$$f(x) := e^{-a_1 - a_2} \sum_{j=0}^{\lfloor x/2 \rfloor} \frac{a_1^{x-2j} a_2^j}{(x-2j)! j!}$$

## Examples

```
>>> from sympy.stats import Hermite, density, E, variance
>>> from sympy import Symbol
```

```
>>> a1 = Symbol("a1", positive=True)
>>> a2 = Symbol("a2", positive=True)
>>> x = Symbol("x")
```

```
>>> H = Hermite("H", a1=5, a2=4)
```

```
>>> density(H)(2)
33*exp(-9)/2
```

```
>>> E(H)
13
```

```
>>> variance(H)
21
```

## References

[R817]

`sympy.stats.Poisson(name, lamda)`

Create a discrete random variable with a Poisson distribution.

### Parameters

**lamda** : Positive number, a rate

### Returns

RandomSymbol

## Explanation

The density of the Poisson distribution is given by

$$f(k) := \frac{\lambda^k e^{-\lambda}}{k!}$$

## Examples

```
>>> from sympy.stats import Poisson, density, E, variance
>>> from sympy import Symbol, simplify
```

```
>>> rate = Symbol("lambda", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Poisson("x", rate)
```

```
>>> density(X)(z)
lambda**z*exp(-lambda)/factorial(z)
```

```
>>> E(X)
lambda
```

```
>>> simplify(variance(X))
lambda
```

## References

[R818], [R819]

`sympy.stats.Logarithmic(name, p)`

Create a discrete random variable with a Logarithmic distribution.

### Parameters

**p** : A value between 0 and 1

### Returns

RandomSymbol

## Explanation

The density of the Logarithmic distribution is given by

$$f(k) := \frac{-p^k}{k \ln(1-p)}$$

## Examples

```
>>> from sympy.stats import Logarithmic, density, E, variance
>>> from sympy import Symbol, S
```

```
>>> p = S.One / 5
>>> z = Symbol("z")
```

```
>>> X = Logarithmic("x", p)
```

```
>>> density(X)(z)
-1/(5**z*z*log(4/5))
```

```
>>> E(X)
-1/(-4*log(5) + 8*log(2))
```

```
>>> variance(X)
-1/((-4*log(5) + 8*log(2))*(-2*log(5) + 4*log(2))) + 1/(-
→ 64*log(2)*log(5) + 64*log(2)**2 + 16*log(5)**2) - 10/(-32*log(5) +
→ 64*log(2))
```

## References

[R820], [R821]

`sympy.stats.NegativeBinomial(name, r, p)`

Create a discrete random variable with a Negative Binomial distribution.

### Parameters

**r** : A positive value

**p** : A value between 0 and 1

### Returns

RandomSymbol

## Explanation

The density of the Negative Binomial distribution is given by

$$f(k) := \binom{k+r-1}{k} (1-p)^r p^k$$

## Examples

```
>>> from sympy.stats import NegativeBinomial, density, E, variance
>>> from sympy import Symbol, S
```

```
>>> r = 5
>>> p = S.One / 5
>>> z = Symbol("z")
```

```
>>> X = NegativeBinomial("x", r, p)
```

```
>>> density(X)(z)
1024*binomial(z + 4, z)/(3125*5**z)
```

```
>>> E(X)
5/4
```

```
>>> variance(X)
25/16
```

## References

[R822], [R823]

`sympy.stats.Skellam(name, mu1, mu2)`

Create a discrete random variable with a Skellam distribution.

### Parameters

**mu1** : A non-negative value

**mu2** : A non-negative value

### Returns

RandomSymbol

## Explanation

The Skellam is the distribution of the difference  $N1 - N2$  of two statistically independent random variables  $N1$  and  $N2$  each Poisson-distributed with respective expected values  $\mu1$  and  $\mu2$ .

The density of the Skellam distribution is given by

$$f(k) := e^{-(\mu_1 + \mu_2)} \left( \frac{\mu_1}{\mu_2} \right)^{k/2} I_k(2\sqrt{\mu_1 \mu_2})$$

## Examples

```
>>> from sympy.stats import Skellam, density, E, variance
>>> from sympy import Symbol, pprint
```

```
>>> z = Symbol("z", integer=True)
>>> mu1 = Symbol("mu1", positive=True)
>>> mu2 = Symbol("mu2", positive=True)
>>> X = Skellam("x", mu1, mu2)
```

```
>>> pprint(density(X)(z), use_unicode=False)
 z
 -
 2
/mu1\ -mu1 - mu2 /
|---| *e *besseli\z, 2*\ / mu1 * \ / mu2 /
\mu2/
>>> E(X)
mu1 - mu2
>>> variance(X).expand()
mu1 + mu2
```

## References

[R824]

`sympy.stats.YuleSimon(name, rho)`

Create a discrete random variable with a Yule-Simon distribution.

### Parameters

**rho** : A positive value

### Returns

RandomSymbol

## Explanation

The density of the Yule-Simon distribution is given by

$$f(k) := \rho B(k, \rho + 1)$$

## Examples

```
>>> from sympy.stats import YuleSimon, density, E, variance
>>> from sympy import Symbol, simplify
```

```
>>> p = 5
>>> z = Symbol("z")
```

```
>>> X = YuleSimon("x", p)
```

```
>>> density(X)(z)
5*beta(z, 6)
```

```
>>> simplify(E(X))
5/4
```

```
>>> simplify(variance(X))
25/48
```

## References

[R825]

`sympy.stats.Zeta(name, s)`

Create a discrete random variable with a Zeta distribution.

### Parameters

**s** : A value greater than 1

### Returns

RandomSymbol

## Explanation

The density of the Zeta distribution is given by

$$f(k) := \frac{1}{k^s \zeta(s)}$$

## Examples

```
>>> from sympy.stats import Zeta, density, E, variance
>>> from sympy import Symbol
```

```
>>> s = 5
>>> z = Symbol("z")
```

```
>>> X = Zeta("x", s)
```

```
>>> density(X)(z)
1/(z**5*zeta(5))
```

```
>>> E(X)
pi**4/(90*zeta(5))
```

```
>>> variance(X)
-pi**8/(8100*zeta(5)**2) + zeta(3)/zeta(5)
```

## References

[R826]

## Continuous Types

`sympy.stats.Arcsin(name, a=0, b=1)`

Create a Continuous Random Variable with an arcsin distribution.

The density of the arcsin distribution is given by

$$f(x) := \frac{1}{\pi \sqrt{(x-a)(b-x)}}$$

with  $x \in (a, b)$ . It must hold that  $-\infty < a < b < \infty$ .

### Parameters

**a** : Real number, the left interval boundary

**b** : Real number, the right interval boundary

### Returns

RandomSymbol

## Examples

```
>>> from sympy.stats import Arcsin, density, cdf
>>> from sympy import Symbol
```

```
>>> a = Symbol("a", real=True)
>>> b = Symbol("b", real=True)
>>> z = Symbol("z")
```

```
>>> X = Arcsin("x", a, b)
```

```
>>> density(X)(z)
1/(pi*sqrt((-a + z)*(b - z)))
```

```
>>> cdf(X)(z)
Piecewise((0, a > z),
 (2*asin(sqrt((-a + z)/(-a + b)))/pi, b >= z),
 (1, True))
```

## References

[R827]

`sympy.stats.Benini(name, alpha, beta, sigma)`

Create a Continuous Random Variable with a Benini distribution.

The density of the Benini distribution is given by

$$f(x) := e^{-\alpha \log \frac{x}{\sigma} - \beta \log^2 \left[ \frac{x}{\sigma} \right]} \left( \frac{\alpha}{x} + \frac{2\beta \log \frac{x}{\sigma}}{x} \right)$$

This is a heavy-tailed distribution and is also known as the log-Rayleigh distribution.

### Parameters

**alpha** : Real number,  $\alpha > 0$ , a shape

**beta** : Real number,  $\beta > 0$ , a shape

**sigma** : Real number,  $\sigma > 0$ , a scale

### Returns

RandomSymbol

## Examples

```
>>> from sympy.stats import Benini, density, cdf
>>> from sympy import Symbol, pprint
```

```
>>> alpha = Symbol("alpha", positive=True)
>>> beta = Symbol("beta", positive=True)
>>> sigma = Symbol("sigma", positive=True)
>>> z = Symbol("z")
```



```
>>> X = Benini("x", alpha, beta, sigma)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
/
| 2*beta*log|-----| - alpha*log|-----| - beta*log 2/ z \
|alpha \|sigma/ \|sigma/ \|sigma/
|----- + -----|*e
\ z z /
```

```
>>> cdf(X)(z)
Piecewise((1 - exp(-alpha*log(z/sigma) - beta*log(z/sigma)**2), sigma <=
→ z),
 (0, True))
```

## References

[R828], [R829]

`sympy.stats.Beta(name, alpha, beta)`

Create a Continuous Random Variable with a Beta distribution.

The density of the Beta distribution is given by

$$f(x) := \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha, \beta)}$$

with  $x \in [0, 1]$ .

### Parameters

**alpha** : Real number,  $\alpha > 0$ , a shape

**beta** : Real number,  $\beta > 0$ , a shape

### Returns

RandomSymbol

## Examples

```
>>> from sympy.stats import Beta, density, E, variance
>>> from sympy import Symbol, simplify, pprint, factor
```

```
>>> alpha = Symbol("alpha", positive=True)
>>> beta = Symbol("beta", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Beta("x", alpha, beta)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
alpha - 1 beta - 1
```

(continues on next page)

(continued from previous page)

$$\frac{z^{alpha-1} (1-z)^{beta-1}}{B(alpha, beta)}$$

```
>>> simplify(E(X))
alpha/(alpha + beta)
```

```
>>> factor(simplify(variance(X)))
alpha*beta/((alpha + beta)**2*(alpha + beta + 1))
```

## References

[R830], [R831]

`sympy.stats.BetaNoncentral`(*name, alpha, beta, lamda*)

Create a Continuous Random Variable with a Type I Noncentral Beta distribution.

The density of the Noncentral Beta distribution is given by

$$f(x) := \sum_{k=0}^{\infty} e^{-\lambda/2} \frac{(\lambda/2)^k}{k!} \frac{x^{\alpha+k-1} (1-x)^{\beta-1}}{B(\alpha+k, \beta)}$$

with  $x \in [0, 1]$ .

### Parameters

**alpha** : Real number,  $\alpha > 0$ , a shape

**beta** : Real number,  $\beta > 0$ , a shape

**lamda** : Real number,  $\lambda \geq 0$ , noncentrality parameter

### Returns

RandomSymbol

## Examples

```
>>> from sympy.stats import BetaNoncentral, density, cdf
>>> from sympy import Symbol, pprint
```

```
>>> alpha = Symbol("alpha", positive=True)
>>> beta = Symbol("beta", positive=True)
>>> lamda = Symbol("lamda", nonnegative=True)
>>> z = Symbol("z")
```

```
>>> X = BetaNoncentral("x", alpha, beta, lamda)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
oo
```

(continues on next page)

(continued from previous page)

$$\frac{z^{k+\alpha-1} \lambda^{\alpha-1} \Gamma(\alpha) \Gamma(\beta)}{\Gamma(\alpha+\beta)} \frac{\lambda^k}{2^k} (1-z)^{\beta-1} e^{-\lambda z} = \frac{B(k+\alpha, \beta) k!}{B(k+\alpha, \beta) k!}$$

$k = 0$

Compute cdf with specific 'x', 'alpha', 'beta' and 'lamda' values as follows:

```
>>> cdf(BetaNoncentral("x", 1, 1, 1), evaluate=False)(2).doit()
2*exp(1/2)
```

The argument `evaluate=False` prevents an attempt at evaluation of the sum for general `x`, before the argument `2` is passed.

## References

[R832], [R833]

`sympy.stats.BetaPrime(name, alpha, beta)`

Create a continuous random variable with a Beta prime distribution.

The density of the Beta prime distribution is given by

$$f(x) := \frac{x^{\alpha-1} (1+x)^{-\alpha-\beta}}{B(\alpha, \beta)}$$

with  $x > 0$ .

### Parameters

**alpha** : Real number,  $\alpha > 0$ , a shape

**beta** : Real number,  $\beta > 0$ , a shape

### Returns

RandomSymbol

## Examples

```
>>> from sympy.stats import BetaPrime, density
>>> from sympy import Symbol, pprint
```

```
>>> alpha = Symbol("alpha", positive=True)
>>> beta = Symbol("beta", positive=True)
>>> z = Symbol("z")
```

```
>>> X = BetaPrime("x", alpha, beta)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
 alpha - 1 -alpha - beta
 z *(z + 1)

 B(alpha, beta)
```

## References

[R834], [R835]

`sympy.stats.BoundedPareto`(*name*, *alpha*, *left*, *right*)

Create a continuous random variable with a Bounded Pareto distribution.

The density of the Bounded Pareto distribution is given by

$$f(x) := \frac{\alpha L^\alpha x^{-\alpha-1}}{1 - (\frac{L}{H})^\alpha}$$

### Parameters

**alpha** : Real Number,  $\alpha > 0$

Shape parameter

**left** : Real Number,  $left > 0$

Location parameter

**right** : Real Number,  $right > left$

Location parameter

### Returns

RandomSymbol

## Examples

```
>>> from sympy.stats import BoundedPareto, density, cdf, E
>>> from sympy import symbols
>>> L, H = symbols('L, H', positive=True)
>>> X = BoundedPareto('X', 2, L, H)
>>> x = symbols('x')
>>> density(X)(x)
2*L**2/(x**3*(1 - L**2/H**2))
>>> cdf(X)(x)
Piecewise((-H**2*L**2/(x**2*(H**2 - L**2)) + H**2/(H**2 - L**2), L <= x),
 (0, True))
>>> E(X).simplify()
2*H*L/(H + L)
```

## References

[R836]

`sympy.stats.Cauchy(name, x0, gamma)`

Create a continuous random variable with a Cauchy distribution.

The density of the Cauchy distribution is given by

$$f(x) := \frac{1}{\pi\gamma[1 + (\frac{x-x_0}{\gamma})^2]}$$

### Parameters

**x0** : Real number, the location

**gamma** : Real number,  $\gamma > 0$ , a scale

### Returns

RandomSymbol

## Examples

```
>>> from sympy.stats import Cauchy, density
>>> from sympy import Symbol
```

```
>>> x0 = Symbol("x0")
>>> gamma = Symbol("gamma", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Cauchy("x", x0, gamma)
```

```
>>> density(X)(z)
1/(pi*gamma*(1 + (-x0 + z)**2/gamma**2))
```

## References

[R837], [R838]

`sympy.stats.Chi(name, k)`

Create a continuous random variable with a Chi distribution.

The density of the Chi distribution is given by

$$f(x) := \frac{2^{1-k/2} x^{k-1} e^{-x^2/2}}{\Gamma(k/2)}$$

with  $x \geq 0$ .

### Parameters

**k** : Positive integer, The number of degrees of freedom

### Returns

RandomSymbol

## Examples

```
>>> from sympy.stats import Chi, density, E
>>> from sympy import Symbol, simplify
```

```
>>> k = Symbol("k", integer=True)
>>> z = Symbol("z")
```

```
>>> X = Chi("x", k)
```

```
>>> density(X)(z)
2**(1 - k/2)*z**(k - 1)*exp(-z**2/2)/gamma(k/2)
```

```
>>> simplify(E(X))
sqrt(2)*gamma(k/2 + 1/2)/gamma(k/2)
```

## References

[R839], [R840]

`sympy.stats.ChiNoncentral`(*name*, *k*, *l*)

Create a continuous random variable with a non-central Chi distribution.

### Parameters

**k** : A positive Integer,  $k > 0$

The number of degrees of freedom.

**lambda** : Real number,  $\lambda > 0$

Shift parameter.

### Returns

RandomSymbol

## Explanation

The density of the non-central Chi distribution is given by

$$f(x) := \frac{e^{-(x^2+\lambda^2)/2} x^k \lambda}{(\lambda x)^{k/2}} I_{k/2-1}(\lambda x)$$

with  $x \geq 0$ . Here,  $I_\nu(x)$  is the *modified Bessel function of the first kind* (page 499).

## Examples

```
>>> from sympy.stats import ChiNoncentral, density
>>> from sympy import Symbol
```

```
>>> k = Symbol("k", integer=True)
>>> l = Symbol("l")
>>> z = Symbol("z")
```

```
>>> X = ChiNoncentral("x", k, l)
```

```
>>> density(X)(z)
l*z**k*exp(-l**2/2 - z**2/2)*besseli(k/2 - 1, l*z)/(l*z)**(k/2)
```

## References

[R841]

`sympy.stats.ChiSquared(name, k)`

Create a continuous random variable with a Chi-squared distribution.

### Parameters

**k** : Positive integer

The number of degrees of freedom.

### Returns

RandomSymbol

## Explanation

The density of the Chi-squared distribution is given by

$$f(x) := \frac{1}{2^{\frac{k}{2}} \Gamma(\frac{k}{2})} x^{\frac{k}{2}-1} e^{-\frac{x}{2}}$$

with  $x \geq 0$ .

## Examples

```
>>> from sympy.stats import ChiSquared, density, E, variance, moment
>>> from sympy import Symbol
```

```
>>> k = Symbol("k", integer=True, positive=True)
>>> z = Symbol("z")
```

```
>>> X = ChiSquared("x", k)
```

```
>>> density(X)(z)
z**(k/2 - 1)*exp(-z/2)/(2**(k/2)*gamma(k/2))
```

```
>>> E(X)
k
```

```
>>> variance(X)
2*k
```

```
>>> moment(X, 3)
k**3 + 6*k**2 + 8*k
```

## References

[R842], [R843]

`sympy.stats.Dagum(name, p, a, b)`

Create a continuous random variable with a Dagum distribution.

### Parameters

**p** : Real number

$p > 0$ , a shape.

**a** : Real number

$a > 0$ , a shape.

**b** : Real number

$b > 0$ , a scale.

### Returns

RandomSymbol

## Explanation

The density of the Dagum distribution is given by

$$f(x) := \frac{ap}{x} \left( \frac{\left(\frac{x}{b}\right)^{ap}}{\left(\left(\frac{x}{b}\right)^a + 1\right)^{p+1}} \right)$$

with  $x > 0$ .

## Examples

```
>>> from sympy.stats import Dagum, density, cdf
>>> from sympy import Symbol
```

```
>>> p = Symbol("p", positive=True)
>>> a = Symbol("a", positive=True)
>>> b = Symbol("b", positive=True)
>>> z = Symbol("z")
```



```
>>> X = Dagum("x", p, a, b)
```

```
>>> density(X)(z)
a*p*(z/b)**(a*p)*((z/b)**a + 1)**(-p - 1)/z
```

```
>>> cdf(X)(z)
Piecewise(((1 + (z/b)**(-a))**(-p), z >= 0), (0, True))
```

## References

[R844]

`sympy.stats.Erlang(name, k, l)`

Create a continuous random variable with an Erlang distribution.

### Parameters

**k** : Positive integer

**l** : Real number,  $\lambda > 0$ , the rate

### Returns

RandomSymbol

## Explanation

The density of the Erlang distribution is given by

$$f(x) := \frac{\lambda^k x^{k-1} e^{-\lambda x}}{(k-1)!}$$

with  $x \in [0, \infty]$ .

## Examples

```
>>> from sympy.stats import Erlang, density, cdf, E, variance
>>> from sympy import Symbol, simplify, pprint
```

```
>>> k = Symbol("k", integer=True, positive=True)
>>> l = Symbol("l", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Erlang("x", k, l)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
 k k - 1 -l*z
l *z *e

 Gamma(k)
```

```
>>> C = cdf(X)(z)
>>> pprint(C, use_unicode=False)
/lowergamma(k, l*z)
|----- for z > 0
| Gamma(k)
|
| 0 otherwise
\
```

```
>>> E(X)
k/l
```

```
>>> simplify(variance(X))
k/l**2
```

## References

[R845], [R846]

`sympy.stats.ExGaussian(name, mean, std, rate)`

Create a continuous random variable with an Exponentially modified Gaussian (EMG) distribution.

### Parameters

**name** : A string giving a name for this distribution

**mean** : A Real number, the mean of Gaussian component

**std** : A positive Real number,

**math**

$\sigma^2 > 0$  the variance of Gaussian component

**rate** : A positive Real number,

**math**

$\lambda > 0$  the rate of Exponential component

### Returns

RandomSymbol

## Explanation

The density of the exponentially modified Gaussian distribution is given by

$$f(x) := \frac{\lambda}{2} e^{\frac{\lambda}{2}(2\mu + \lambda\sigma^2 - 2x)} \operatorname{erfc}\left(\frac{\mu + \lambda\sigma^2 - x}{\sqrt{2}\sigma}\right)$$

with  $x > 0$ . Note that the expected value is  $1/\lambda$ .

## Examples

```
>>> from sympy.stats import ExGaussian, density, cdf, E
>>> from sympy.stats import variance, skewness
>>> from sympy import Symbol, pprint, simplify
```

```
>>> mean = Symbol("mu")
>>> std = Symbol("sigma", positive=True)
>>> rate = Symbol("lamda", positive=True)
>>> z = Symbol("z")
>>> X = ExGaussian("x", mean, std, rate)
```

```
>>> pprint(density(X)(z), use_unicode=False)
 / 2 \
lamda*\lamda*sigma + 2*mu - 2*z/
----- / ____ / 2
-> \\\
 2 |\ / 2 *\lamda*sigma + mu -
-> z/|
lamda*e *erfc|-----
-> --|
 \ 2*sigma
-> /

-> ---
 2
```

```
>>> cdf(X)(z)
-(erf(sqrt(2)*(-lamda**2*sigma**2 + lamda*(-mu + z))/(2*lamda*sigma))/2
-> + 1/2)*exp(lamda**2*sigma**2/2 - lamda*(-mu + z)) + erf(sqrt(2)*(-mu +
-> z)/(2*sigma))/2 + 1/2
```

```
>>> E(X)
(lamda*mu + 1)/lamda
```

```
>>> simplify(variance(X))
sigma**2 + lamda**(-2)
```

```
>>> simplify(skewness(X))
2/(lamda**2*sigma**2 + 1)**(3/2)
```

## References

[R847]

`sympy.stats.Exponential(name, rate)`

Create a continuous random variable with an Exponential distribution.

### Parameters

**rate** : A positive Real number,  $\lambda > 0$ , the rate (or inverse scale/inverse mean)

### Returns

RandomSymbol

## Explanation

The density of the exponential distribution is given by

$$f(x) := \lambda \exp(-\lambda x)$$

with  $x > 0$ . Note that the expected value is  $1/\lambda$ .

## Examples

```
>>> from sympy.stats import Exponential, density, cdf, E
>>> from sympy.stats import variance, std, skewness, quantile
>>> from sympy import Symbol
```

```
>>> l = Symbol("lambda", positive=True)
>>> z = Symbol("z")
>>> p = Symbol("p")
>>> X = Exponential("x", l)
```

```
>>> density(X)(z)
lambda*exp(-lambda*z)
```

```
>>> cdf(X)(z)
Piecewise((1 - exp(-lambda*z), z >= 0), (0, True))
```

```
>>> quantile(X)(p)
-log(1 - p)/lambda
```

```
>>> E(X)
1/lambda
```

```
>>> variance(X)
lambda**(-2)
```

```
>>> skewness(X)
2
```

```
>>> X = Exponential('x', 10)
```

```
>>> density(X)(z)
10*exp(-10*z)
```

```
>>> E(X)
1/10
```

```
>>> std(X)
1/10
```

## References

[R848], [R849]

`sympy.stats.FDistribution(name, d1, d2)`

Create a continuous random variable with a F distribution.

### Parameters

**d1** :  $d_1 > 0$ , where  $d_1$  is the degrees of freedom ( $n_1 - 1$ )

**d2** :  $d_2 > 0$ , where  $d_2$  is the degrees of freedom ( $n_2 - 1$ )

### Returns

RandomSymbol

## Explanation

The density of the F distribution is given by

$$f(x) := \frac{\sqrt{\frac{(d_1 x)^{d_1} d_2^{d_2}}{(d_1 x + d_2)^{d_1 + d_2}}}}{x B\left(\frac{d_1}{2}, \frac{d_2}{2}\right)}$$

with  $x > 0$ .

## Examples

```
>>> from sympy.stats import FDistribution, density
>>> from sympy import Symbol, pprint
```

```
>>> d1 = Symbol("d1", positive=True)
>>> d2 = Symbol("d2", positive=True)
>>> z = Symbol("z")
```

```
>>> X = FDistribution("x", d1, d2)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
d2
--
2 / d1 -d1 - d2
d2 * \ / (d1*z) * (d1*z + d2)

 /d1 d2\
 z*B|--, --|
 \2 2 /
```

## References

[R850], [R851]

`sympy.stats.FisherZ(name, d1, d2)`

Create a Continuous Random Variable with an Fisher's Z distribution.

### Parameters

**d1** :  $d_1 > 0$

Degree of freedom.

**d2** :  $d_2 > 0$

Degree of freedom.

### Returns

RandomSymbol

## Explanation

The density of the Fisher's Z distribution is given by

$$f(x) := \frac{2d_1^{d_1/2} d_2^{d_2/2}}{B(d_1/2, d_2/2)} \frac{e^{d_1 z}}{(d_1 e^{2z} + d_2)^{(d_1+d_2)/2}}$$

## Examples

```
>>> from sympy.stats import FisherZ, density
>>> from sympy import Symbol, pprint
```

```
>>> d1 = Symbol("d1", positive=True)
>>> d2 = Symbol("d2", positive=True)
>>> z = Symbol("z")
```

```
>>> X = FisherZ("x", d1, d2)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)

 d1 d2 d1 d2
 -- -- - -- - --
 2 2 2 2
2*d1 *d2 /\ 2*z \ d1*z
*\d1*e + d2/ *e

 /d1 d2\
 B|--, --|
 \2 2 /
```

## References

[R852], [R853]

`sympy.stats.Frechet(name, a, s=1, m=0)`

Create a continuous random variable with a Frechet distribution.

### Parameters

**a** : Real number,  $a \in (0, \infty)$  the shape

**s** : Real number,  $s \in (0, \infty)$  the scale

**m** : Real number,  $m \in (-\infty, \infty)$  the minimum

### Returns

RandomSymbol

## Explanation

The density of the Frechet distribution is given by

$$f(x) := \frac{\alpha}{s} \left( \frac{x-m}{s} \right)^{-1-\alpha} e^{-\left( \frac{x-m}{s} \right)^{-\alpha}}$$

with  $x \geq m$ .

## Examples

```
>>> from sympy.stats import Frechet, density, cdf
>>> from sympy import Symbol
```

```
>>> a = Symbol("a", positive=True)
>>> s = Symbol("s", positive=True)
>>> m = Symbol("m", real=True)
>>> z = Symbol("z")
```

```
>>> X = Frechet("x", a, s, m)
```

```
>>> density(X)(z)
a*((-m + z)/s)**(-a - 1)*exp(-1/((-m + z)/s)**a)/s
```

```
>>> cdf(X)(z)
Piecewise((exp(-1/((-m + z)/s)**a), m <= z), (0, True))
```

## References

[R854]

`sympy.stats.Gamma(name, k, theta)`

Create a continuous random variable with a Gamma distribution.

### Parameters

**k** : Real number,  $k > 0$ , a shape

**theta** : Real number,  $\theta > 0$ , a scale

### Returns

RandomSymbol

## Explanation

The density of the Gamma distribution is given by

$$f(x) := \frac{1}{\Gamma(k)\theta^k} x^{k-1} e^{-\frac{x}{\theta}}$$

with  $x \in [0, 1]$ .

## Examples

```
>>> from sympy.stats import Gamma, density, cdf, E, variance
>>> from sympy import Symbol, pprint, simplify
```

```
>>> k = Symbol("k", positive=True)
>>> theta = Symbol("theta", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Gamma("x", k, theta)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
 -z

 -k k - 1 theta
 theta *z *e

 Gamma(k)
```



```
>>> C = cdf(X, meijerg=True)(z)
>>> pprint(C, use_unicode=False)
/
| k*lowergamma|k, ----|
| \ theta/
<----- for z >= 0
| Gamma(k + 1)
|
\ 0 otherwise
```

```
>>> E(X)
k*theta
```

```
>>> V = simplify(variance(X))
>>> pprint(V, use_unicode=False)
 2
k*theta
```

## References

[R855], [R856]

`sympy.stats.GammaInverse(name, a, b)`

Create a continuous random variable with an inverse Gamma distribution.

### Parameters

**a** : Real number,  $a > 0$ , a shape

**b** : Real number,  $b > 0$ , a scale

### Returns

RandomSymbol

## Explanation

The density of the inverse Gamma distribution is given by

$$f(x) := \frac{\beta^\alpha}{\Gamma(\alpha)} x^{-\alpha-1} \exp\left(\frac{-\beta}{x}\right)$$

with  $x > 0$ .

## Examples

```
>>> from sympy.stats import GammaInverse, density, cdf
>>> from sympy import Symbol, pprint
```

```
>>> a = Symbol("a", positive=True)
>>> b = Symbol("b", positive=True)
>>> z = Symbol("z")
```

```
>>> X = GammaInverse("x", a, b)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
 -b

 a -a - 1 z
b *z *e

 Gamma(a)
```

```
>>> cdf(X)(z)
Piecewise((uppergamma(a, b/z)/gamma(a), z > 0), (0, True))
```

## References

[R857]

`sympy.stats.Gompertz(name, b, eta)`

Create a Continuous Random Variable with Gompertz distribution.

### Parameters

**b** : Real number,  $b > 0$ , a scale

**eta** : Real number,  $\eta > 0$ , a shape

### Returns

RandomSymbol

## Explanation

The density of the Gompertz distribution is given by

$$f(x) := b\eta e^{bx} e^{\eta} \exp(-\eta e^{bx})$$

with  $x \in [0, \infty)$ .

## Examples

```
>>> from sympy.stats import Gompertz, density
>>> from sympy import Symbol
```

```
>>> b = Symbol("b", positive=True)
>>> eta = Symbol("eta", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Gompertz("x", b, eta)
```

```
>>> density(X)(z)
b*eta*exp(eta)*exp(b*z)*exp(-eta*exp(b*z))
```

## References

[R858]

`sympy.stats.Gumbel(name, beta, mu, minimum=False)`

Create a Continuous Random Variable with Gumbel distribution.

### Parameters

**mu** : Real number,  $\mu$ , a location

**beta** : Real number,  $\beta > 0$ , a scale

**minimum** : Boolean, by default False, set to True for enabling minimum distribution

### Returns

RandomSymbol

## Explanation

The density of the Gumbel distribution is given by

For Maximum

$$f(x) := \frac{1}{\beta} \exp\left(-\frac{x-\mu}{\beta} - \exp\left(-\frac{x-\mu}{\beta}\right)\right)$$

with  $x \in [-\infty, \infty]$ .

For Minimum

$$f(x) := \frac{e^{-e^{-\frac{-\mu+x}{\beta}} + \frac{-\mu+x}{\beta}}}{\beta}$$

with  $x \in [-\infty, \infty]$ .

## Examples

```
>>> from sympy.stats import Gumbel, density, cdf
>>> from sympy import Symbol
>>> x = Symbol("x")
>>> mu = Symbol("mu")
>>> beta = Symbol("beta", positive=True)
>>> X = Gumbel("x", beta, mu)
>>> density(X)(x)
exp(-exp(-(-mu + x)/beta) - (-mu + x)/beta)/beta
>>> cdf(X)(x)
exp(-exp(-(-mu + x)/beta))
```

## References

[R859], [R860], [R861], [R862]

`sympy.stats.Kumaraswamy(name, a, b)`

Create a Continuous Random Variable with a Kumaraswamy distribution.

### Parameters

**a** : Real number,  $a > 0$ , a shape

**b** : Real number,  $b > 0$ , a shape

### Returns

RandomSymbol

## Explanation

The density of the Kumaraswamy distribution is given by

$$f(x) := abx^{a-1}(1-x^a)^{b-1}$$

with  $x \in [0, 1]$ .

## Examples

```
>>> from sympy.stats import Kumaraswamy, density, cdf
>>> from sympy import Symbol, pprint
```

```
>>> a = Symbol("a", positive=True)
>>> b = Symbol("b", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Kumaraswamy("x", a, b)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
 b - 1
 a - 1 / a\
a*b*z *\1 - z /
```

```
>>> cdf(X)(z)
Piecewise((0, z < 0), (1 - (1 - z**a)**b, z <= 1), (1, True))
```

## References

[R863]

`sympy.stats.Laplace(name, mu, b)`

Create a continuous random variable with a Laplace distribution.

### Parameters

**mu** : Real number or a list/matrix, the location (mean) or the location vector

**b** : Real number or a positive definite matrix, representing a scale or the covariance matrix.

### Returns

RandomSymbol

## Explanation

The density of the Laplace distribution is given by

$$f(x) := \frac{1}{2b} \exp\left(-\frac{|x - \mu|}{b}\right)$$

## Examples

```
>>> from sympy.stats import Laplace, density, cdf
>>> from sympy import Symbol, pprint
```

```
>>> mu = Symbol("mu")
>>> b = Symbol("b", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Laplace("x", mu, b)
```

```
>>> density(X)(z)
exp(-Abs(mu - z)/b)/(2*b)
```

```
>>> cdf(X)(z)
Piecewise((exp((-mu + z)/b)/2, mu > z), (1 - exp((mu - z)/b)/2, True))
```

```
>>> L = Laplace('L', [1, 2], [[1, 0], [0, 1]])
>>> pprint(density(L)(1, 2), use_unicode=False)
 5 / \
e *besselk\0, \ / 35 /

 pi
```

## References

[R864], [R865]

`sympy.stats.Levy(name, mu, c)`

Create a continuous random variable with a Levy distribution.

The density of the Levy distribution is given by

$$f(x) := \sqrt{\frac{c}{2\pi}} \frac{\exp - \frac{c}{2(x-\mu)}}{(x-\mu)^{3/2}}$$

### Parameters

**mu** : Real number

The location parameter.

**c** : Real number,  $c > 0$

A scale parameter.

### Returns

RandomSymbol

## Examples

```
>>> from sympy.stats import Levy, density, cdf
>>> from sympy import Symbol
```

```
>>> mu = Symbol("mu", real=True)
>>> c = Symbol("c", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Levy("x", mu, c)
```

```
>>> density(X)(z)
sqrt(2)*sqrt(c)*exp(-c/(-2*mu + 2*z))/(2*sqrt(pi)*(-mu + z)**(3/2))
```

```
>>> cdf(X)(z)
erfc(sqrt(c)*sqrt(1/(-2*mu + 2*z)))
```

## References

[R866], [R867]

`sympy.stats.Logistic(name, mu, s)`

Create a continuous random variable with a logistic distribution.

### Parameters

**mu** : Real number, the location (mean)

**s** : Real number,  $s > 0$ , a scale

### Returns

RandomSymbol

## Explanation

The density of the logistic distribution is given by

$$f(x) := \frac{e^{-(x-\mu)/s}}{s(1 + e^{-(x-\mu)/s})^2}$$

## Examples

```
>>> from sympy.stats import Logistic, density, cdf
>>> from sympy import Symbol
```

```
>>> mu = Symbol("mu", real=True)
>>> s = Symbol("s", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Logistic("x", mu, s)
```

```
>>> density(X)(z)
exp((mu - z)/s)/(s*(exp((mu - z)/s) + 1)**2)
```

```
>>> cdf(X)(z)
1/(exp((mu - z)/s) + 1)
```

## References

[R868], [R869]

`sympy.stats.LogLogistic(name, alpha, beta)`

Create a continuous random variable with a log-logistic distribution. The distribution is unimodal when  $\beta > 1$ .

### Parameters

**alpha** : Real number,  $\alpha > 0$ , scale parameter and median of distribution

**beta** : Real number,  $\beta > 0$ , a shape parameter

### Returns

RandomSymbol

## Explanation

The density of the log-logistic distribution is given by

$$f(x) := \frac{(\frac{\beta}{\alpha})(\frac{x}{\alpha})^{\beta-1}}{(1 + (\frac{x}{\alpha})^\beta)^2}$$

## Examples

```
>>> from sympy.stats import LogLogistic, density, cdf, quantile
>>> from sympy import Symbol, pprint
```

```
>>> alpha = Symbol("alpha", positive=True)
>>> beta = Symbol("beta", positive=True)
>>> p = Symbol("p")
>>> z = Symbol("z", positive=True)
```

```
>>> X = LogLogistic("x", alpha, beta)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
 beta - 1
 / z \
beta*|-----|
 \alpha/

 / beta \
 | / z \ |
alpha*| |-----| + 1 |
 \|alpha/ /
```

```
>>> cdf(X)(z)
1/(1 + (z/alpha)**(-beta))
```

```
>>> quantile(X)(p)
alpha*(p/(1 - p))**(1/beta)
```

## References

[R870]

`sympy.stats.LogNormal`(*name, mean, std*)

Create a continuous random variable with a log-normal distribution.

### Parameters

**mu** : Real number

The log-scale.

**sigma** : Real number

A shape. ( $\sigma^2 > 0$ )

### Returns

RandomSymbol



## Explanation

The density of the log-normal distribution is given by

$$f(x) := \frac{1}{x\sqrt{2\pi\sigma^2}} e^{-\frac{(\ln x - \mu)^2}{2\sigma^2}}$$

with  $x \geq 0$ .

## Examples

```
>>> from sympy.stats import LogNormal, density
>>> from sympy import Symbol, pprint
```

```
>>> mu = Symbol("mu", real=True)
>>> sigma = Symbol("sigma", positive=True)
>>> z = Symbol("z")
```

```
>>> X = LogNormal("x", mu, sigma)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
```

```
 2
 -(-mu + log(z))

 2
 2*sigma
\ / 2 *e

2*\ / pi *sigma*z
```

```
>>> X = LogNormal('x', 0, 1) # Mean 0, standard deviation 1
```

```
>>> density(X)(z)
sqrt(2)*exp(-log(z)**2/2)/(2*sqrt(pi)*z)
```

## References

[R871], [R872]

`sympy.stats.Lomax(name, alpha, lamda)`

Create a continuous random variable with a Lomax distribution.

### Parameters

**alpha** : Real Number,  $\alpha > 0$

Shape parameter

**lamda** : Real Number,  $\lambda > 0$

Scale parameter

**Returns**

RandomSymbol

**Explanation**

The density of the Lomax distribution is given by

$$f(x) := \frac{\alpha}{\lambda} \left[ 1 + \frac{x}{\lambda} \right]^{-(\alpha+1)}$$

**Examples**

```
>>> from sympy.stats import Lomax, density, cdf, E
>>> from sympy import symbols
>>> a, l = symbols('a, l', positive=True)
>>> X = Lomax('X', a, l)
>>> x = symbols('x')
>>> density(X)(x)
a*(1 + x/l)**(-a - 1)/l
>>> cdf(X)(x)
Piecewise((1 - 1/(1 + x/l)**a, x >= 0), (0, True))
>>> a = 2
>>> X = Lomax('X', a, l)
>>> E(X)
l
```

**References**[\[R873\]](#)

`sympy.stats.Maxwell(name, a)`

Create a continuous random variable with a Maxwell distribution.

**Parameters**

**a** : Real number,  $a > 0$

**Returns**

RandomSymbol

**Explanation**

The density of the Maxwell distribution is given by

$$f(x) := \sqrt{\frac{2}{\pi}} \frac{x^2 e^{-x^2/(2a^2)}}{a^3}$$

with  $x \geq 0$ .

## Examples

```
>>> from sympy.stats import Maxwell, density, E, variance
>>> from sympy import Symbol, simplify
```

```
>>> a = Symbol("a", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Maxwell("x", a)
```

```
>>> density(X)(z)
sqrt(2)*z**2*exp(-z**2/(2*a**2))/(sqrt(pi)*a**3)
```

```
>>> E(X)
2*sqrt(2)*a/sqrt(pi)
```

```
>>> simplify(variance(X))
a**2*(-8 + 3*pi)/pi
```

## References

[R874], [R875]

`sympy.stats.Moyal`(*name*, *mu*, *sigma*)

Create a continuous random variable with a Moyal distribution.

### Parameters

**mu** : Real number

Location parameter

**sigma** : Real positive number

Scale parameter

### Returns

RandomSymbol

## Explanation

The density of the Moyal distribution is given by

$$f(x) := \frac{\exp -\frac{1}{2} \exp -\frac{x-\mu}{\sigma} - \frac{x-\mu}{2\sigma}}{\sqrt{2\pi}\sigma}$$

with  $x \in \mathbb{R}$ .

## Examples

```
>>> from sympy.stats import Moyal, density, cdf
>>> from sympy import Symbol, simplify
>>> mu = Symbol("mu", real=True)
>>> sigma = Symbol("sigma", positive=True, real=True)
>>> z = Symbol("z")
>>> X = Moyal("x", mu, sigma)
>>> density(X)(z)
sqrt(2)*exp(-exp((mu - z)/sigma)/2 - (-mu + z)/(2*sigma))/
 (2*sqrt(pi)*sigma)
>>> simplify(cdf(X)(z))
1 - erf(sqrt(2)*exp((mu - z)/(2*sigma))/2)
```

## References

[R876], [R877]

`sympy.stats.Nakagami(name, mu, omega)`

Create a continuous random variable with a Nakagami distribution.

### Parameters

**mu** : Real number,  $\mu \geq \frac{1}{2}$ , a shape

**omega** : Real number,  $\omega > 0$ , the spread

### Returns

RandomSymbol

## Explanation

The density of the Nakagami distribution is given by

$$f(x) := \frac{2\mu^\mu}{\Gamma(\mu)\omega^\mu} x^{2\mu-1} \exp\left(-\frac{\mu}{\omega}x^2\right)$$

with  $x > 0$ .

## Examples

```
>>> from sympy.stats import Nakagami, density, E, variance, cdf
>>> from sympy import Symbol, simplify, pprint
```

```
>>> mu = Symbol("mu", positive=True)
>>> omega = Symbol("omega", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Nakagami("x", mu, omega)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
 2
 -mu*z

 mu -mu 2*mu - 1 omega
2*mu *omega *z *e

 Gamma(mu)
```

```
>>> simplify(E(X))
sqrt(mu)*sqrt(omega)*gamma(mu + 1/2)/gamma(mu + 1)
```

```
>>> V = simplify(variance(X))
>>> pprint(V, use_unicode=False)
 2
 omega*Gamma (mu + 1/2)
omega - -----
 Gamma(mu)*Gamma(mu + 1)
```

```
>>> cdf(X)(z)
Piecewise((lowergamma(mu, mu*z**2/omega)/gamma(mu), z > 0),
 (0, True))
```

## References

[R878]

`sympy.stats.Normal(name, mean, std)`

Create a continuous random variable with a Normal distribution.

### Parameters

**mu** : Real number or a list representing the mean or the mean vector

**sigma** : Real number or a positive definite square matrix,

$\sigma^2 > 0$ , the variance

### Returns

RandomSymbol

## Explanation

The density of the Normal distribution is given by

$$f(x) := \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

## Examples

```
>>> from sympy.stats import Normal, density, E, std, cdf, skewness, quantile, marginal_distribution
>>> from sympy import Symbol, simplify, pprint
```

```
>>> mu = Symbol("mu")
>>> sigma = Symbol("sigma", positive=True)
>>> z = Symbol("z")
>>> y = Symbol("y")
>>> p = Symbol("p")
>>> X = Normal("x", mu, sigma)
```

```
>>> density(X)(z)
sqrt(2)*exp(-(-mu + z)**2/(2*sigma**2))/(2*sqrt(pi)*sigma)
```

```
>>> C = simplify(cdf(X))(z) # it needs a little more help...
>>> pprint(C, use_unicode=False)
 / \
 | \ / 2 * (-mu + z) |
erf|-----|
 \ 2*sigma / 1
----- + -
 2 2
```

```
>>> quantile(X)(p)
mu + sqrt(2)*sigma*erfinv(2*p - 1)
```

```
>>> simplify(skewness(X))
0
```

```
>>> X = Normal("x", 0, 1) # Mean 0, standard deviation 1
>>> density(X)(z)
sqrt(2)*exp(-z**2/2)/(2*sqrt(pi))
```

```
>>> E(2*X + 1)
1
```

```
>>> simplify(std(2*X + 1))
2
```

```
>>> m = Normal('X', [1, 2], [[2, 1], [1, 2]])
>>> pprint(density(m)(y, z), use_unicode=False)
 2 2
 y y*z z
 - --- + --- - --- + z - 1
 3 3 3
\ / 3 * e

 6*pi
```

```
>>> marginal_distribution(m, m[0])(1)
1/(2*sqrt(pi))
```

## References

[R879], [R880]

`sympy.stats.Pareto(name, xm, alpha)`

Create a continuous random variable with the Pareto distribution.

### Parameters

**xm** : Real number,  $x_m > 0$ , a scale

**alpha** : Real number,  $\alpha > 0$ , a shape

### Returns

RandomSymbol

## Explanation

The density of the Pareto distribution is given by

$$f(x) := \frac{\alpha x_m^\alpha}{x^{\alpha+1}}$$

with  $x \in [x_m, \infty]$ .

## Examples

```
>>> from sympy.stats import Pareto, density
>>> from sympy import Symbol
```

```
>>> xm = Symbol("xm", positive=True)
>>> beta = Symbol("beta", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Pareto("x", xm, beta)
```

```
>>> density(X)(z)
beta*xm**beta*z**(-beta - 1)
```

## References

[R881], [R882]

`sympy.stats.PowerFunction(name, alpha, a, b)`

Creates a continuous random variable with a Power Function Distribution.

**Parameters**

**alpha** : Positive number,  $0 < \alpha$ , the shape paramater

**a** : Real number,  $-\infty < a$ , the left boundary

**b** : Real number,  $a < b < \infty$ , the right boundary

**Returns**

RandomSymbol

**Explanation**

The density of PowerFunction distribution is given by

$$f(x) := \frac{\alpha(x-a)^{\alpha-1}}{(b-a)^\alpha}$$

with  $x \in [a, b]$ .

**Examples**

```
>>> from sympy.stats import PowerFunction, density, cdf, E, variance
>>> from sympy import Symbol
>>> alpha = Symbol("alpha", positive=True)
>>> a = Symbol("a", real=True)
>>> b = Symbol("b", real=True)
>>> z = Symbol("z")
```

```
>>> X = PowerFunction("X", 2, a, b)
```

```
>>> density(X)(z)
(-2*a + 2*z)/(-a + b)**2
```

```
>>> cdf(X)(z)
Piecewise((a**2/(a**2 - 2*a*b + b**2) - 2*a*z/(a**2 - 2*a*b + b**2) +
z**2/(a**2 - 2*a*b + b**2), a <= z), (0, True))
```

```
>>> alpha = 2
>>> a = 0
>>> b = 1
>>> Y = PowerFunction("Y", alpha, a, b)
```

```
>>> E(Y)
2/3
```

```
>>> variance(Y)
1/18
```



## References

[R883]

`sympy.stats.QuadraticU(name, a, b)`

Create a Continuous Random Variable with a U-quadratic distribution.

### Parameters

**a** : Real number

**b** : Real number,  $a < b$

### Returns

RandomSymbol

## Explanation

The density of the U-quadratic distribution is given by

$$f(x) := \alpha(x - \beta)^2$$

with  $x \in [a, b]$ .

## Examples

```
>>> from sympy.stats import QuadraticU, density
>>> from sympy import Symbol, pprint
```

```
>>> a = Symbol("a", real=True)
>>> b = Symbol("b", real=True)
>>> z = Symbol("z")
```

```
>>> X = QuadraticU("x", a, b)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
/
| / a b \
|12*|- - - - + z|
| \ 2 2 /
<----- for And(b >= z, a <= z)
|
| 3
| (-a + b)
|
| 0 otherwise
\
```

## References

[R884]

`sympy.stats.RaisedCosine(name, mu, s)`

Create a Continuous Random Variable with a raised cosine distribution.

### Parameters

**mu** : Real number

**s** : Real number,  $s > 0$

### Returns

RandomSymbol

## Explanation

The density of the raised cosine distribution is given by

$$f(x) := \frac{1}{2s} \left( 1 + \cos \left( \frac{x - \mu}{s} \pi \right) \right)$$

with  $x \in [\mu - s, \mu + s]$ .

## Examples

```
>>> from sympy.stats import RaisedCosine, density
>>> from sympy import Symbol, pprint
```

```
>>> mu = Symbol("mu", real=True)
>>> s = Symbol("s", positive=True)
>>> z = Symbol("z")
```

```
>>> X = RaisedCosine("x", mu, s)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
/ /pi*(-mu + z)\
|cos|-----| + 1
| \ s /
<----- for And(z >= mu - s, z <= mu + s)
| 2*s
|
\ 0 otherwise
```

## References

[R885]

`sympy.stats.Rayleigh(name, sigma)`

Create a continuous random variable with a Rayleigh distribution.

### Parameters

**sigma** : Real number,  $\sigma > 0$

### Returns

RandomSymbol

## Explanation

The density of the Rayleigh distribution is given by

$$f(x) := \frac{x}{\sigma^2} e^{-x^2/2\sigma^2}$$

with  $x > 0$ .

## Examples

```
>>> from sympy.stats import Rayleigh, density, E, variance
>>> from sympy import Symbol
```

```
>>> sigma = Symbol("sigma", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Rayleigh("x", sigma)
```

```
>>> density(X)(z)
z*exp(-z**2/(2*sigma**2))/sigma**2
```

```
>>> E(X)
sqrt(2)*sqrt(pi)*sigma/2
```

```
>>> variance(X)
-pi*sigma**2/2 + 2*sigma**2
```

## References

[R886], [R887]

`sympy.stats.Reciprocal(name, a, b)`

Creates a continuous random variable with a reciprocal distribution.

### Parameters

**a** : Real number,  $0 < a$

**b** : Real number,  $a < b$

**Returns**

RandomSymbol

**Examples**

```
>>> from sympy.stats import Reciprocal, density, cdf
>>> from sympy import symbols
>>> a, b, x = symbols('a, b, x', positive=True)
>>> R = Reciprocal('R', a, b)
```

```
>>> density(R)(x)
1/(x*(-log(a) + log(b)))
>>> cdf(R)(x)
Piecewise((log(a)/(log(a) - log(b)) - log(x)/(log(a) - log(b)), a <= x),
 (0, True))
```

**Reference**`sympy.stats.StudentT(name, nu)`

Create a continuous random variable with a student's t distribution.

**Parameters****nu** : Real number,  $\nu > 0$ , the degrees of freedom**Returns**

RandomSymbol

**Explanation**

The density of the student's t distribution is given by

$$f(x) := \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\nu\pi}\Gamma\left(\frac{\nu}{2}\right)} \left(1 + \frac{x^2}{\nu}\right)^{-\frac{\nu+1}{2}}$$

**Examples**

```
>>> from sympy.stats import StudentT, density, cdf
>>> from sympy import Symbol, pprint
```

```
>>> nu = Symbol("nu", positive=True)
>>> z = Symbol("z")
```

```
>>> X = StudentT("x", nu)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
 nu 1
```

(continues on next page)

(continued from previous page)

$$\frac{\sqrt{\frac{z}{1 + \frac{z^2}{\nu}}}}{\sqrt{\nu} \cdot B\left(\frac{1}{2}, \frac{\nu}{2}\right)}$$

```
>>> cdf(X)(z)
1/2 + z*gamma(nu/2 + 1/2)*hyper((1/2, nu/2 + 1/2), (3/2,),
-z**2/nu)/(sqrt(pi)*sqrt(nu)*gamma(nu/2))
```

## References

[R889], [R890]

`sympy.stats.ShiftedGompertz(name, b, eta)`

Create a continuous random variable with a Shifted Gompertz distribution.

### Parameters

**b** : Real number,  $b > 0$ , a scale

**eta** : Real number,  $\eta > 0$ , a shape

### Returns

RandomSymbol

## Explanation

The density of the Shifted Gompertz distribution is given by

$$f(x) := be^{-bx}e^{-\eta \exp(-bx)} \left[1 + \eta(1 - e^{-bx})\right]$$

with  $x \in [0, \infty)$ .

## Examples

```
>>> from sympy.stats import ShiftedGompertz, density
>>> from sympy import Symbol
```

```
>>> b = Symbol("b", positive=True)
>>> eta = Symbol("eta", positive=True)
>>> x = Symbol("x")
```

```
>>> X = ShiftedGompertz("x", b, eta)
```

```
>>> density(X)(x)
b*(eta*(1 - exp(-b*x)) + 1)*exp(-b*x)*exp(-eta*exp(-b*x))
```

## References

[R891]

`sympy.stats.Trapezoidal(name, a, b, c, d)`

Create a continuous random variable with a trapezoidal distribution.

### Parameters

**a** : Real number,  $a < d$

**b** : Real number,  $a \leq b < c$

**c** : Real number,  $b < c \leq d$

**d** : Real number

### Returns

RandomSymbol

## Explanation

The density of the trapezoidal distribution is given by

$$f(x) := \begin{cases} 0 & \text{for } x < a, \\ \frac{2(x-a)}{(b-a)(d+c-a-b)} & \text{for } a \leq x < b, \\ \frac{2}{d+c-a-b} & \text{for } b \leq x < c, \\ \frac{2(d-x)}{(d-c)(d+c-a-b)} & \text{for } c \leq x < d, \\ 0 & \text{for } d < x. \end{cases}$$

## Examples

```
>>> from sympy.stats import Trapezoidal, density
>>> from sympy import Symbol, pprint
```

```
>>> a = Symbol("a")
>>> b = Symbol("b")
>>> c = Symbol("c")
>>> d = Symbol("d")
>>> z = Symbol("z")
```

```
>>> X = Trapezoidal("x", a,b,c,d)
```

```
>>> pprint(density(X)(z), use_unicode=False)
/
-2*a + 2*z
|----- for And(a <= z, b > z)
|(-a + b)*(-a - b + c + d)
```

(continues on next page)

(continued from previous page)

$$\begin{aligned}
 & \frac{2}{-a - b + c + d} && \text{for } \text{And}(b \leq z, c > z) \\
 & \frac{2*d - 2*z}{(-c + d)*(-a - b + c + d)} && \text{for } \text{And}(d \geq z, c \leq z) \\
 & 0 && \text{otherwise}
 \end{aligned}$$

## References

[R892]

`sympy.stats.Triangular(name, a, b, c)`

Create a continuous random variable with a triangular distribution.

### Parameters

**a** : Real number,  $a \in (-\infty, \infty)$

**b** : Real number,  $a < b$

**c** : Real number,  $a \leq c \leq b$

### Returns

RandomSymbol

## Explanation

The density of the triangular distribution is given by

$$f(x) := \begin{cases} 0 & \text{for } x < a, \\ \frac{2(x-a)}{(b-a)(c-a)} & \text{for } a \leq x < c, \\ \frac{2}{b-a} & \text{for } x = c, \\ \frac{2(b-x)}{(b-a)(b-c)} & \text{for } c < x \leq b, \\ 0 & \text{for } b < x. \end{cases}$$

## Examples

```
>>> from sympy.stats import Triangular, density
>>> from sympy import Symbol, pprint
```

```
>>> a = Symbol("a")
>>> b = Symbol("b")
>>> c = Symbol("c")
>>> z = Symbol("z")
```

```
>>> X = Triangular("x", a,b,c)
```

```
>>> pprint(density(X)(z), use_unicode=False)
/ -2*a + 2*z
|----- for And(a <= z, c > z)
| (-a + b)*(-a + c)
|
| 2
|----- for c = z
< -a + b
|
| 2*b - 2*z
|----- for And(b >= z, c < z)
| (-a + b)*(b - c)
|
\ 0
 otherwise
```

## References

[R893], [R894]

`sympy.stats.Uniform(name, left, right)`

Create a continuous random variable with a uniform distribution.

### Parameters

**a** : Real number,  $-\infty < a$ , the left boundary

**b** : Real number,  $a < b < \infty$ , the right boundary

### Returns

RandomSymbol

## Explanation

The density of the uniform distribution is given by

$$f(x) := \begin{cases} \frac{1}{b-a} & \text{for } x \in [a, b] \\ 0 & \text{otherwise} \end{cases}$$

with  $x \in [a, b]$ .

## Examples

```
>>> from sympy.stats import Uniform, density, cdf, E, variance
>>> from sympy import Symbol, simplify
```

```
>>> a = Symbol("a", negative=True)
>>> b = Symbol("b", positive=True)
>>> z = Symbol("z")
```



```
>>> X = Uniform("x", a, b)
```

```
>>> density(X)(z)
Piecewise((1/(-a + b), (b >= z) & (a <= z)), (0, True))
```

```
>>> cdf(X)(z)
Piecewise((0, a > z), ((-a + z)/(-a + b), b >= z), (1, True))
```

```
>>> E(X)
a/2 + b/2
```

```
>>> simplify(variance(X))
a**2/12 - a*b/6 + b**2/12
```

## References

[R895], [R896]

`sympy.stats.UniformSum(name, n)`

Create a continuous random variable with an Irwin-Hall distribution.

### Parameters

**n** : A positive integer,  $n > 0$

### Returns

RandomSymbol

## Explanation

The probability distribution function depends on a single parameter  $n$  which is an integer.

The density of the Irwin-Hall distribution is given by

$$f(x) := \frac{1}{(n-1)!} \sum_{k=0}^{\lfloor x \rfloor} (-1)^k \binom{n}{k} (x-k)^{n-1}$$

## Examples

```
>>> from sympy.stats import UniformSum, density, cdf
>>> from sympy import Symbol, pprint
```

```
>>> n = Symbol("n", integer=True)
>>> z = Symbol("z")
```

```
>>> X = UniformSum("x", n)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
floor(z)
```

$$\sum_{k=0}^{\lfloor z \rfloor} \frac{(-1)^k (-k + z)^{n-1} \sqrt{n}}{\sqrt{k}}$$

-----

$$(n-1)!$$

```
>>> cdf(X)(z)
Piecewise((0, z < 0), (Sum((-1)**_k*(-_k + z)**n*binomial(n, _k),
(_k, 0, floor(z)))/factorial(n), n >= z), (1, True))
```

Compute cdf with specific 'x' and 'n' values as follows : `>>> cdf(UniformSum("x", 5), evaluate=False)(2).doit()` 9/40

The argument `evaluate=False` prevents an attempt at evaluation of the sum for general `n`, before the argument 2 is passed.

## References

[R897], [R898]

`sympy.stats.VonMises(name, mu, k)`

Create a Continuous Random Variable with a von Mises distribution.

### Parameters

**mu** : Real number

Measure of location.

**k** : Real number

Measure of concentration.

### Returns

RandomSymbol

## Explanation

The density of the von Mises distribution is given by

$$f(x) := \frac{e^{\kappa \cos(x-\mu)}}{2\pi I_0(\kappa)}$$

with  $x \in [0, 2\pi]$ .

## Examples

```
>>> from sympy.stats import VonMises, density
>>> from sympy import Symbol, pprint
```

```
>>> mu = Symbol("mu")
>>> k = Symbol("k", positive=True)
>>> z = Symbol("z")
```

```
>>> X = VonMises("x", mu, k)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
 k*cos(mu - z)
 e

2*pi*besseli(0, k)
```

## References

[R899], [R900]

`sympy.stats.Wald(name, mean, shape)`

Create a continuous random variable with an Inverse Gaussian distribution. Inverse Gaussian distribution is also known as Wald distribution.

### Parameters

**mu :**

Positive number representing the mean.

**lambda :**

Positive number representing the shape parameter.

### Returns

RandomSymbol

## Explanation

The density of the Inverse Gaussian distribution is given by

$$f(x) := \sqrt{\frac{\lambda}{2\pi x^3}} e^{-\frac{\lambda(x-\mu)^2}{2x\mu^2}}$$

## Examples

```
>>> from sympy.stats import GaussianInverse, density, E, std, skewness
>>> from sympy import Symbol, pprint
```

```
>>> mu = Symbol("mu", positive=True)
>>> lamda = Symbol("lambda", positive=True)
>>> z = Symbol("z", positive=True)
>>> X = GaussianInverse("x", mu, lamda)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
 2
 -lambda*(-mu + z)

 2
 2*mu *z
\ / 2 * \ / lambda * e

 3/2
2*\ / pi *z
```

```
>>> E(X)
mu
```

```
>>> std(X).expand()
mu**(3/2)/sqrt(lambda)
```

```
>>> skewness(X).expand()
3*sqrt(mu)/sqrt(lambda)
```

## References

[R901], [R902]

`sympy.stats.Weibull(name, alpha, beta)`

Create a continuous random variable with a Weibull distribution.

### Parameters

**lambda** : Real number,  $\lambda > 0$ , a scale

**k** : Real number,  $k > 0$ , a shape

### Returns

RandomSymbol

## Explanation

The density of the Weibull distribution is given by

$$f(x) := \begin{cases} \frac{k}{\lambda} \left(\frac{x}{\lambda}\right)^{k-1} e^{-(x/\lambda)^k} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

## Examples

```
>>> from sympy.stats import Weibull, density, E, variance
>>> from sympy import Symbol, simplify
```

```
>>> l = Symbol("lambda", positive=True)
>>> k = Symbol("k", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Weibull("x", l, k)
```

```
>>> density(X)(z)
k*(z/lambda)**(k - 1)*exp(-(z/lambda)**k)/lambda
```

```
>>> simplify(E(X))
lambda*gamma(1 + 1/k)
```

```
>>> simplify(variance(X))
lambda**2*(-gamma(1 + 1/k)**2 + gamma(1 + 2/k))
```

## References

[R903], [R904]

`sympy.stats.WignerSemicircle(name, R)`

Create a continuous random variable with a Wigner semicircle distribution.

### Parameters

**R** : Real number,  $R > 0$ , the radius

### Returns

A RandomSymbol.

## Explanation

The density of the Wigner semicircle distribution is given by

$$f(x) := \frac{2}{\pi R^2} \sqrt{R^2 - x^2}$$

with  $x \in [-R, R]$ .

## Examples

```
>>> from sympy.stats import WignerSemicircle, density, E
>>> from sympy import Symbol
```

```
>>> R = Symbol("R", positive=True)
>>> z = Symbol("z")
```

```
>>> X = WignerSemicircle("x", R)
```

```
>>> density(X)(z)
2*sqrt(R**2 - z**2)/(pi*R**2)
```

```
>>> E(X)
0
```

## References

[R905], [R906]

`sympy.stats.ContinuousRV(symbol, density, set=Interval(-oo, oo), **kwargs)`

Create a Continuous Random Variable given the following:

### Parameters

**symbol** : Symbol

Represents name of the random variable.

**density** : Expression containing symbol

Represents probability density function.

**set** : set/Interval

Represents the region where the pdf is valid, by default is real line.

**check** : bool

If True, it will check whether the given density integrates to 1 over the given set. If False, it will not perform this check. Default is False.

### Returns

RandomSymbol

Many common continuous random variable types are already implemented.

This function should be necessary only very rarely.

## Examples

```
>>> from sympy import Symbol, sqrt, exp, pi
>>> from sympy.stats import ContinuousRV, P, E
```

```
>>> x = Symbol("x")
```

```
>>> pdf = sqrt(2)*exp(-x**2/2)/(2*sqrt(pi)) # Normal distribution
>>> X = ContinuousRV(x, pdf)
```

```
>>> E(X)
0
>>> P(X>0)
1/2
```

## Joint Types

`sympy.stats.JointRV(symbol, pdf, _set=None)`

Create a Joint Random Variable where each of its component is continuous, given the following:

### Parameters

**symbol** : Symbol

Represents name of the random variable.

**pdf** : A PDF in terms of indexed symbols of the symbol given as the first argument

### Returns

RandomSymbol

## Note

As of now, the set for each component for a JointRV is equal to the set of all integers, which cannot be changed.

## Examples

```
>>> from sympy import exp, pi, Indexed, S
>>> from sympy.stats import density, JointRV
>>> x1, x2 = (Indexed('x', i) for i in (1, 2))
>>> pdf = exp(-x1**2/2 + x1 - x2**2/2 - S(1)/2)/(2*pi)
>>> N1 = JointRV('x', pdf) #Multivariate Normal distribution
>>> density(N1)(1, 2)
exp(-2)/(2*pi)
```

`sympy.stats.marginal_distribution(rv, *indices)`

Marginal distribution function of a joint random variable.

**Parameters**

**rv** : A random variable with a joint probability distribution.

**indices** : Component indices or the indexed random symbol  
for which the joint distribution is to be calculated

**Returns**

A Lambda expression in *sym*.

**Examples**

```
>>> from sympy.stats import MultivariateNormal, marginal_distribution
>>> m = MultivariateNormal('X', [1, 2], [[2, 1], [1, 2]])
>>> marginal_distribution(m, m[0])(1)
1/(2*sqrt(pi))
```

`sympy.stats.MultivariateNormal(name, mu, sigma)`

Creates a continuous random variable with Multivariate Normal Distribution.

The density of the multivariate normal distribution can be found at [1].

**Parameters**

**mu** : List representing the mean or the mean vector

**sigma** : Positive semidefinite square matrix

Represents covariance Matrix. If  $\sigma$  is noninvertible then only sampling is supported currently

**Returns**

RandomSymbol

**Examples**

```
>>> from sympy.stats import MultivariateNormal, density, marginal_
->distribution
>>> from sympy import symbols, MatrixSymbol
>>> X = MultivariateNormal('X', [3, 4], [[2, 1], [1, 2]])
>>> y, z = symbols('y z')
>>> density(X)(y, z)
sqrt(3)*exp(-y**2/3 + y*z/3 + 2*y/3 - z**2/3 + 5*z/3 - 13/3)/(6*pi)
>>> density(X)(1, 2)
sqrt(3)*exp(-4/3)/(6*pi)
>>> marginal_distribution(X, X[1])(y)
exp(-(y - 4)**2/4)/(2*sqrt(pi))
>>> marginal_distribution(X, X[0])(y)
exp(-(y - 3)**2/4)/(2*sqrt(pi))
```

The example below shows that it is also possible to use symbolic parameters to define the MultivariateNormal class.



```
>>> n = symbols('n', integer=True, positive=True)
>>> Sg = MatrixSymbol('Sg', n, n)
>>> mu = MatrixSymbol('mu', n, 1)
>>> obs = MatrixSymbol('obs', n, 1)
>>> X = MultivariateNormal('X', mu, Sg)
```

The density of a multivariate normal can be calculated using a matrix argument, as shown below.

```
>>> density(X)(obs)
(exp(((1/2)*mu.T - (1/2)*obs.T)*Sg**(-1)*(-mu + obs)))/
→sqrt((2*pi)**n*Determinant(Sg)))[0, 0]
```

## References

[R907]

`sympy.stats.MultivariateLaplace(name, mu, sigma)`

Creates a continuous random variable with Multivariate Laplace Distribution.

The density of the multivariate Laplace distribution can be found at [1].

### Parameters

**mu** : List representing the mean or the mean vector

**sigma** : Positive definite square matrix

Represents covariance Matrix

### Returns

RandomSymbol

## Examples

```
>>> from sympy.stats import MultivariateLaplace, density
>>> from sympy import symbols
>>> y, z = symbols('y z')
>>> X = MultivariateLaplace('X', [2, 4], [[3, 1], [1, 3]])
>>> density(X)(y, z)
sqrt(2)*exp(y/4 + 5*z/4)*besselk(0, sqrt(15*y*(3*y/8 - z/8)/2 + 15*z*(-y/
→8 + 3*z/8)/2))/(4*pi)
>>> density(X)(1, 2)
sqrt(2)*exp(11/4)*besselk(0, sqrt(165)/4)/(4*pi)
```

## References

[R908]

`sympy.stats.GeneralizedMultivariateLogGamma(syms, delta, v, lamda, mu)`

Creates a joint random variable with generalized multivariate log gamma distribution.

The joint pdf can be found at [1].

### Parameters

**syms** : list/tuple/set of symbols for identifying each component

**delta** : A constant in range  $[0, 1]$

**v** : Positive real number

**lamda** : List of positive real numbers

**mu** : List of positive real numbers

### Returns

RandomSymbol

## Examples

```
>>> from sympy.stats import density
>>> from sympy.stats.joint_rv_types import GeneralizedMultivariateLogGamma
>>> from sympy import symbols, S
>>> v = 1
>>> l, mu = [1, 1, 1], [1, 1, 1]
>>> d = S.Half
>>> y = symbols('y_1:4', positive=True)
>>> Gd = GeneralizedMultivariateLogGamma('G', d, v, l, mu)
>>> density(Gd)(y[0], y[1], y[2])
Sum(exp((n + 1)*(y_1 + y_2 + y_3) - exp(y_1) - exp(y_2) - exp(y_3))/(2**n*gamma(n + 1)**3), (n, 0, oo))/2
```

## Note

If the `GeneralizedMultivariateLogGamma` is too long to type use,

```
>>> from sympy.stats.joint_rv_types import GeneralizedMultivariateLogGamma as GMVLG
>>> Gd = GMVLG('G', d, v, l, mu)
```

If you want to pass the matrix `omega` instead of the constant `delta`, then use `GeneralizedMultivariateLogGammaOmega`.

## References

[R909], [R910]

`sympy.stats.GeneralizedMultivariateLogGammaOmega(syms, omega, v, lamda, mu)`

Extends GeneralizedMultivariateLogGamma.

### Parameters

**syms** : list/tuple/set of symbols

For identifying each component

**omega** : A square matrix

Every element of square matrix must be absolute value of square root of correlation coefficient

**v** : Positive real number

**lamda** : List of positive real numbers

**mu** : List of positive real numbers

### Returns

RandomSymbol

## Examples

```
>>> from sympy.stats import density
>>> from sympy.stats.joint_rv_types import _
↳ GeneralizedMultivariateLogGammaOmega
>>> from sympy import Matrix, symbols, S
>>> omega = Matrix([[1, S.Half, S.Half], [S.Half, 1, S.Half], [S.Half, S.
↳ Half, 1]])
>>> v = 1
>>> l, mu = [1, 1, 1], [1, 1, 1]
>>> G = GeneralizedMultivariateLogGammaOmega('G', omega, v, l, mu)
>>> y = symbols('y_1:4', positive=True)
>>> density(G)(y[0], y[1], y[2])
sqrt(2)*Sum((1 - sqrt(2)/2)**n*exp((n + 1)*(y_1 + y_2 + y_3) - exp(y_1) -
exp(y_2) - exp(y_3))/gamma(n + 1)**3, (n, 0, oo))/2
```

## Notes

If the GeneralizedMultivariateLogGammaOmega is too long to type use,

```
>>> from sympy.stats.joint_rv_types import _
↳ GeneralizedMultivariateLogGammaOmega as GMVLGO
>>> G = GMVLGO('G', omega, v, l, mu)
```

## References

[R911], [R912]

`sympy.stats.Multinomial(syms, n, *p)`

Creates a discrete random variable with Multinomial Distribution.

The density of the said distribution can be found at [1].

### Parameters

**n** : Positive integer

Represents number of trials

**p** : List of event probabilities

Must be in the range of [0, 1].

### Returns

RandomSymbol

## Examples

```
>>> from sympy.stats import density, Multinomial, marginal_distribution
>>> from sympy import symbols
>>> x1, x2, x3 = symbols('x1, x2, x3', nonnegative=True, integer=True)
>>> p1, p2, p3 = symbols('p1, p2, p3', positive=True)
>>> M = Multinomial('M', 3, p1, p2, p3)
>>> density(M)(x1, x2, x3)
Piecewise((6*p1**x1*p2**x2*p3**x3/
 ↳ (factorial(x1)*factorial(x2)*factorial(x3)),
Eq(x1 + x2 + x3, 3)), (0, True))
>>> marginal_distribution(M, M[0])(x1).subs(x1, 1)
3*p1*p2**2 + 6*p1*p2*p3 + 3*p1*p3**2
```

## References

[R913], [R914]

`sympy.stats.MultivariateBeta(syms, *alpha)`

Creates a continuous random variable with Dirichlet/Multivariate Beta Distribution.

The density of the Dirichlet distribution can be found at [1].

### Parameters

**alpha** : Positive real numbers

Signifies concentration numbers.

### Returns

RandomSymbol

## Examples

```
>>> from sympy.stats import density, MultivariateBeta, marginal_
 ↳distribution
>>> from sympy import Symbol
>>> a1 = Symbol('a1', positive=True)
>>> a2 = Symbol('a2', positive=True)
>>> B = MultivariateBeta('B', [a1, a2])
>>> C = MultivariateBeta('C', a1, a2)
>>> x = Symbol('x')
>>> y = Symbol('y')
>>> density(B)(x, y)
x**(a1 - 1)*y**(a2 - 1)*gamma(a1 + a2)/(gamma(a1)*gamma(a2))
>>> marginal_distribution(C, C[0])(x)
x**(a1 - 1)*gamma(a1 + a2)/(a2*gamma(a1)*gamma(a2))
```

## References

[R915], [R916]

`sympy.stats.MultivariateEwens(syms, n, theta)`

Creates a discrete random variable with Multivariate Ewens Distribution.

The density of the said distribution can be found at [1].

### Parameters

**n** : Positive integer

Size of the sample or the integer whose partitions are considered

**theta** : Positive real number

Denotes Mutation rate

### Returns

RandomSymbol

## Examples

```
>>> from sympy.stats import density, marginal_distribution,
 ↳MultivariateEwens
>>> from sympy import Symbol
>>> a1 = Symbol('a1', positive=True)
>>> a2 = Symbol('a2', positive=True)
>>> ed = MultivariateEwens('E', 2, 1)
>>> density(ed)(a1, a2)
Piecewise((1/(2**a2*factorial(a1)*factorial(a2)), Eq(a1 + 2*a2, 2)), (0,
 ↳True))
>>> marginal_distribution(ed, ed[0])(a1)
Piecewise((1/factorial(a1), Eq(a1, 2)), (0, True))
```

## References

[R917], [R918]

`sympy.stats.MultivariateT(syms, mu, sigma, v)`

Creates a joint random variable with multivariate T-distribution.

### Parameters

**syms** : A symbol/str

For identifying the random variable.

**mu** : A list/matrix

Representing the location vector

**sigma** : The shape matrix for the distribution

### Returns

RandomSymbol

## Examples

```
>>> from sympy.stats import density, MultivariateT
>>> from sympy import Symbol
```

```
>>> x = Symbol("x")
>>> X = MultivariateT("x", [1, 1], [[1, 0], [0, 1]], 2)
```

```
>>> density(X)(1, 2)
2/(9*pi)
```

`sympy.stats.NegativeMultinomial(syms, k0, *p)`

Creates a discrete random variable with Negative Multinomial Distribution.

The density of the said distribution can be found at [1].

### Parameters

**k0** : positive integer

Represents number of failures before the experiment is stopped

**p** : List of event probabilities

Must be in the range of [0, 1]

### Returns

RandomSymbol

## Examples

```
>>> from sympy.stats import density, NegativeMultinomial, marginal_
 ↪distribution
>>> from sympy import symbols
>>> x1, x2, x3 = symbols('x1, x2, x3', nonnegative=True, integer=True)
>>> p1, p2, p3 = symbols('p1, p2, p3', positive=True)
>>> N = NegativeMultinomial('M', 3, p1, p2, p3)
>>> N_c = NegativeMultinomial('M', 3, 0.1, 0.1, 0.1)
>>> density(N)(x1, x2, x3)
p1**x1*p2**x2*p3**x3*(-p1 - p2 - p3 + 1)**3*gamma(x1 + x2 +
x3 + 3)/(2*factorial(x1)*factorial(x2)*factorial(x3))
>>> marginal_distribution(N_c, N_c[0])(1).evalf().round(2)
0.25
```

## References

[R919], [R920]

`sympy.stats.NormalGamma(sym, mu, lamda, alpha, beta)`

Creates a bivariate joint random variable with multivariate Normal gamma distribution.

### Parameters

- sym** : A symbol/str  
For identifying the random variable.
- mu** : A real number  
The mean of the normal distribution
- lamda** : A positive integer  
Parameter of joint distribution
- alpha** : A positive integer  
Parameter of joint distribution
- beta** : A positive integer  
Parameter of joint distribution

### Returns

RandomSymbol

## Examples

```
>>> from sympy.stats import density, NormalGamma
>>> from sympy import symbols
```

```
>>> X = NormalGamma('x', 0, 1, 2, 3)
>>> y, z = symbols('y z')
```

```
>>> density(X)(y, z)
9*sqrt(2)*z**(3/2)*exp(-3*z)*exp(-y**2*z/2)/(2*sqrt(pi))
```

## References

[R921]

## Stochastic Processes

**class** `sympy.stats.DiscreteMarkovChain`(*sym*, *state\_space=None*, *trans\_probs=None*)

Represents a finite discrete time-homogeneous Markov chain.

This type of Markov Chain can be uniquely characterised by its (ordered) state space and its one-step transition probability matrix.

### Parameters

**sym:**

The name given to the Markov Chain

**state\_space:**

Optional, by default, `Range(n)`

**trans\_probs:**

Optional, by default, `MatrixSymbol('T', n, n)`

## Examples

```
>>> from sympy.stats import DiscreteMarkovChain, TransitionMatrixOf, P, E
>>> from sympy import Matrix, MatrixSymbol, Eq, symbols
>>> T = Matrix([[0.5, 0.2, 0.3],[0.2, 0.5, 0.3],[0.2, 0.3, 0.5]])
>>> Y = DiscreteMarkovChain("Y", [0, 1, 2], T)
>>> YS = DiscreteMarkovChain("Y")
```

```
>>> Y.state_space
{0, 1, 2}
>>> Y.transition_probabilities
Matrix([
[0.5, 0.2, 0.3],
[0.2, 0.5, 0.3],
[0.2, 0.3, 0.5]])
>>> TS = MatrixSymbol('T', 3, 3)
>>> P(Eq(YS[3], 2), Eq(YS[1], 1) & TransitionMatrixOf(YS, TS))
T[0, 2]*T[1, 0] + T[1, 1]*T[1, 2] + T[1, 2]*T[2, 2]
>>> P(Eq(Y[3], 2), Eq(Y[1], 1)).round(2)
0.36
```

Probabilities will be calculated based on indexes rather than state names. For example, with the Sunny-Cloudy-Rainy model with string state names:

```
>>> from sympy.core.symbol import Str
>>> Y = DiscreteMarkovChain("Y", [Str('Sunny'), Str('Cloudy'), Str('Rainy
→')], T)
>>> P(Eq(Y[3], 2), Eq(Y[1], 1)).round(2)
0.36
```



This gives the same answer as the `[0, 1, 2]` state space. Currently, there is no support for state names within probability and expectation statements. Here is a work-around using `Str`:

```
>>> P(Eq(Str('Rainy'), Y[3]), Eq(Y[1], Str('Cloudy'))).round(2)
0.36
```

Symbol state names can also be used:

```
>>> sunny, cloudy, rainy = symbols('Sunny, Cloudy, Rainy')
>>> Y = DiscreteMarkovChain("Y", [sunny, cloudy, rainy], T)
>>> P(Eq(Y[3], rainy), Eq(Y[1], cloudy)).round(2)
0.36
```

Expectations will be calculated as follows:

```
>>> E(Y[3], Eq(Y[1], cloudy))
0.38*Cloudy + 0.36*Rainy + 0.26*Sunny
```

Probability of expressions with multiple `RandomIndexedSymbols` can also be calculated provided there is only 1 `RandomIndexedSymbol` in the given condition. It is always better to use `Rational` instead of floating point numbers for the probabilities in the transition matrix to avoid errors.

```
>>> from sympy import Gt, Le, Rational
>>> T = Matrix([[Rational(5, 10), Rational(3, 10), Rational(2, 10)],
↳ [Rational(2, 10), Rational(7, 10), Rational(1, 10)], [Rational(3, 10),
↳ Rational(3, 10), Rational(4, 10)]]
>>> Y = DiscreteMarkovChain("Y", [0, 1, 2], T)
>>> P(Eq(Y[3], Y[1]), Eq(Y[0], 0)).round(3)
0.409
>>> P(Gt(Y[3], Y[1]), Eq(Y[0], 0)).round(2)
0.36
>>> P(Le(Y[15], Y[10]), Eq(Y[8], 2)).round(7)
0.6963328
```

Symbolic probability queries are also supported

```
>>> a, b, c, d = symbols('a b c d')
>>> T = Matrix([[Rational(1, 10), Rational(4, 10), Rational(5, 10)],
↳ [Rational(3, 10), Rational(4, 10), Rational(3, 10)], [Rational(7, 10),
↳ Rational(2, 10), Rational(1, 10)]]
>>> Y = DiscreteMarkovChain("Y", [0, 1, 2], T)
>>> query = P(Eq(Y[a], b), Eq(Y[c], d))
>>> query.subs({a:10, b:2, c:5, d:1}).round(4)
0.3096
>>> P(Eq(Y[10], 2), Eq(Y[5], 1)).evalf().round(4)
0.3096
>>> query_gt = P(Gt(Y[a], b), Eq(Y[c], d))
>>> query_gt.subs({a:21, b:0, c:5, d:0}).evalf().round(5)
0.64705
>>> P(Gt(Y[21], 0), Eq(Y[5], 0)).round(5)
0.64705
```

There is limited support for arbitrarily sized states:

```

>>> n = symbols('n', nonnegative=True, integer=True)
>>> T = MatrixSymbol('T', n, n)
>>> Y = DiscreteMarkovChain("Y", trans_probs=T)
>>> Y.state_space
Range(0, n, 1)
>>> query = P(Eq(Y[a], b), Eq(Y[c], d))
>>> query.subs({a:10, b:2, c:5, d:1})
(T**5)[1, 2]

```

## References

[R922], [R923]

### absorbing\_probabilities()

Computes the absorbing probabilities, i.e. the  $ij$ -th entry of the matrix denotes the probability of Markov chain being absorbed in state  $j$  starting from state  $i$ .

**canonical\_form()**  $\rightarrow$  Tuple[List[[Basic](#) (page 927)], [ImmutableDenseMatrix](#) (page 1369)]

Reorders the one-step transition matrix so that recurrent states appear first and transient states appear last. Other representations include inserting transient states first and recurrent states last.

#### Returns

states, P\_new

states is the list that describes the order of the new states in the matrix so that the  $i$ th element in states is the state of the  $i$ th row of A. P\_new is the new transition matrix in canonical form.

## Examples

```

>>> from sympy.stats import DiscreteMarkovChain
>>> from sympy import Matrix, S

```

You can convert your chain into canonical form:

```

>>> T = Matrix([[S(1)/2, S(1)/2, 0, 0, 0],
... [S(2)/5, S(1)/5, S(2)/5, 0, 0],
... [0, 0, 1, 0, 0],
... [0, 0, S(1)/2, S(1)/2, 0],
... [S(1)/2, 0, 0, 0, S(1)/2]])
>>> X = DiscreteMarkovChain('X', list(range(1, 6)), trans_probs=T)
>>> states, new_matrix = X.canonical_form()
>>> states
[3, 1, 2, 4, 5]

```

```

>>> new_matrix
Matrix([
[1, 0, 0, 0, 0],
[0, 1/2, 1/2, 0, 0],

```

(continues on next page)

(continued from previous page)

```
[2/5, 2/5, 1/5, 0, 0],
[1/2, 0, 0, 1/2, 0],
[0, 1/2, 0, 0, 1/2]])
```

The new states are [3, 1, 2, 4, 5] and you can create a new chain with this and its canonical form will remain the same (since it is already in canonical form).

```
>>> X = DiscreteMarkovChain('X', states, new_matrix)
>>> states, new_matrix = X.canonical_form()
>>> states
[3, 1, 2, 4, 5]
```

```
>>> new_matrix
Matrix([
[1, 0, 0, 0, 0],
[0, 1/2, 1/2, 0, 0],
[2/5, 2/5, 1/5, 0, 0],
[1/2, 0, 0, 1/2, 0],
[0, 1/2, 0, 0, 1/2]])
```

This is not limited to absorbing chains:

```
>>> T = Matrix([[0, 5, 5, 0, 0],
... [0, 0, 0, 10, 0],
... [5, 0, 5, 0, 0],
... [0, 10, 0, 0, 0],
... [0, 3, 0, 3, 4]])/10
>>> X = DiscreteMarkovChain('X', trans_probs=T)
>>> states, new_matrix = X.canonical_form()
>>> states
[1, 3, 0, 2, 4]
```

```
>>> new_matrix
Matrix([
[0, 1, 0, 0, 0],
[1, 0, 0, 0, 0],
[1/2, 0, 0, 1/2, 0],
[0, 0, 1/2, 1/2, 0],
[3/10, 3/10, 0, 0, 2/5]])
```

#### See also:

[\*sympy.stats.DiscreteMarkovChain.communication\\_classes\*](#) (page 2948),  
[\*sympy.stats.DiscreteMarkovChain.decompose\*](#) (page 2949)

## References

[R924], [R925]

**communication\_classes()** → List[Tuple[List[[Basic](#) (page 927)], [Boolean](#) (page 1163), [Integer](#) (page 987)]]

Returns the list of communication classes that partition the states of the markov chain.

A communication class is defined to be a set of states such that every state in that set is reachable from every other state in that set. Due to its properties this forms a class in the mathematical sense. Communication classes are also known as recurrence classes.

### Returns

classes

The classes are a list of tuples. Each tuple represents a single communication class with its properties. The first element in the tuple is the list of states in the class, the second element is whether the class is recurrent and the third element is the period of the communication class.

## Examples

```
>>> from sympy.stats import DiscreteMarkovChain
>>> from sympy import Matrix
>>> T = Matrix([[0, 1, 0],
... [1, 0, 0],
... [1, 0, 0]])
>>> X = DiscreteMarkovChain('X', [1, 2, 3], T)
>>> classes = X.communication_classes()
>>> for states, is_recurrent, period in classes:
... states, is_recurrent, period
([1, 2], True, 2)
([3], False, 1)
```

From this we can see that states 1 and 2 communicate, are recurrent and have a period of 2. We can also see state 3 is transient with a period of 1.

## Notes

The algorithm used is of order  $O(n^2)$  where  $n$  is the number of states in the markov chain. It uses Tarjan's algorithm to find the classes themselves and then it uses a breadth-first search algorithm to find each class's periodicity. Most of the algorithm's components approach  $O(n)$  as the matrix becomes more and more sparse.

## References

[R926], [R927], [R928], [R929]

**decompose()** → Tuple[List[*Basic* (page 927)], *ImmutableDenseMatrix* (page 1369), *ImmutableDenseMatrix* (page 1369), *ImmutableDenseMatrix* (page 1369)]

Decomposes the transition matrix into submatrices with special properties.

The transition matrix can be decomposed into 4 submatrices: - A - the submatrix from recurrent states to recurrent states. - B - the submatrix from transient to recurrent states. - C - the submatrix from transient to transient states. - O - the submatrix of zeros for recurrent to transient states.

### Returns

states, A, B, C

states - a list of state names with the first being the recurrent states and the last being the transient states in the order of the row names of A and then the row names of C. A - the submatrix from recurrent states to recurrent states. B - the submatrix from transient to recurrent states. C - the submatrix from transient to transient states.

## Examples

```
>>> from sympy.stats import DiscreteMarkovChain
>>> from sympy import Matrix, S
```

One can decompose this chain for example:

```
>>> T = Matrix([[S(1)/2, S(1)/2, 0, 0, 0],
... [S(2)/5, S(1)/5, S(2)/5, 0, 0],
... [0, 0, 1, 0, 0],
... [0, 0, S(1)/2, S(1)/2, 0],
... [S(1)/2, 0, 0, 0, S(1)/2]])
>>> X = DiscreteMarkovChain('X', trans_probs=T)
>>> states, A, B, C = X.decompose()
>>> states
[2, 0, 1, 3, 4]
```

```
>>> A # recurrent to recurrent
Matrix([[1]])
```

```
>>> B # transient to recurrent
Matrix([
[0],
[2/5],
[1/2],
[0]])
```

```
>>> C # transient to transient
Matrix([
[1/2, 1/2, 0, 0],
[2/5, 1/5, 0, 0],
```

(continues on next page)

(continued from previous page)

```
[0, 0, 1/2, 0],
[1/2, 0, 0, 1/2]])
```

This means that state 2 is the only absorbing state (since A is a 1x1 matrix). B is a 4x1 matrix since the 4 remaining transient states all merge into recurrent state 2. And C is the 4x4 matrix that shows how the transient states 0, 1, 3, 4 all interact.

**See also:**

[\*sympy.stats.DiscreteMarkovChain.communication\\_classes\*](#) (page 2948),  
[\*sympy.stats.DiscreteMarkovChain.canonical\\_form\*](#) (page 2946)

**References**

[R930], [R931]

**fixed\_row\_vector()**

A wrapper for `stationary_distribution()`.

**fundamental\_matrix()**

Each entry fundamental matrix can be interpreted as the expected number of times the chains is in state j if it started in state i.

**References**

[R932]

**property limiting\_distribution**

The fixed row vector is the limiting distribution of a discrete Markov chain.

**sample()****Returns**

sample: iterator object

iterator object containing the sample

**stationary\_distribution**(*condition\_set=False*) → Union[[\*ImmutableDenseMatrix\*](#) (page 1369), [\*ConditionSet\*](#) (page 1215), [\*Lambda\*](#) (page 1039)]

The stationary distribution is any row vector,  $p$ , that solves  $p = pP$ , is row stochastic and each element in  $p$  must be nonnegative. That means in matrix form:  $(P - I)^T p^T = 0$  and  $(1, \dots, 1)p = 1$  where  $P$  is the one-step transition matrix.

All time-homogeneous Markov Chains with a finite state space have at least one stationary distribution. In addition, if a finite time-homogeneous Markov Chain is irreducible, the stationary distribution is unique.

**Parameters**

**condition\_set** : bool

If the chain has a symbolic size or transition matrix, it will return a `Lambda` if `False` and return a `ConditionSet` if `True`.

## Examples

```
>>> from sympy.stats import DiscreteMarkovChain
>>> from sympy import Matrix, S
```

An irreducible Markov Chain

```
>>> T = Matrix([[S(1)/2, S(1)/2, 0],
... [S(4)/5, S(1)/5, 0],
... [1, 0, 0]])
>>> X = DiscreteMarkovChain('X', trans_probs=T)
>>> X.stationary_distribution()
Matrix([[8/13, 5/13, 0]])
```

A reducible Markov Chain

```
>>> T = Matrix([[S(1)/2, S(1)/2, 0],
... [S(4)/5, S(1)/5, 0],
... [0, 0, 1]])
>>> X = DiscreteMarkovChain('X', trans_probs=T)
>>> X.stationary_distribution()
Matrix([[8/13 - 8*tau0/13, 5/13 - 5*tau0/13, tau0]])
```

```
>>> Y = DiscreteMarkovChain('Y')
>>> Y.stationary_distribution()
Lambda((wm, _T), Eq(wm*_T, wm))
```

```
>>> Y.stationary_distribution(condition_set=True)
ConditionSet(wm, Eq(wm*_T, wm))
```

**See also:**

[\*sympy.stats.DiscreteMarkovChain.limiting\\_distribution\*](#) (page 2950)

## References

[R933], [R934]

### property `transition_probabilities`

Transition probabilities of discrete Markov chain, either an instance of `Matrix` or `MatrixSymbol`.

**class** `sympy.stats.ContinuousMarkovChain(sym, state_space=None, gen_mat=None)`

Represents continuous time Markov chain.

### Parameters

**sym** : Symbol/str

**state\_space** : Set

Optional, by default, `S.Reals`

**gen\_mat** : Matrix/ImmutableMatrix/MatrixSymbol

Optional, by default, `None`

## Examples

```
>>> from sympy.stats import ContinuousMarkovChain, P
>>> from sympy import Matrix, S, Eq, Gt
>>> G = Matrix([[S(1), S(1)], [S(1), -S(1)]])
>>> C = ContinuousMarkovChain('C', state_space=[0, 1], gen_mat=G)
>>> C.limiting_distribution()
Matrix([[1/2, 1/2]])
>>> C.state_space
{0, 1}
>>> C.generator_matrix
Matrix([
[-1, 1],
[1, -1]])
```

Probability queries are supported

```
>>> P(Eq(C(1.96), 0), Eq(C(0.78), 1)).round(5)
0.45279
>>> P(Gt(C(1.7), 0), Eq(C(0.82), 1)).round(5)
0.58602
```

Probability of expressions with multiple RandomIndexedSymbols can also be calculated provided there is only 1 RandomIndexedSymbol in the given condition. It is always better to use Rational instead of floating point numbers for the probabilities in the generator matrix to avoid errors.

```
>>> from sympy import Gt, Le, Rational
>>> G = Matrix([[S(1), Rational(1, 10), Rational(9, 10)], [Rational(2, 5), S(1), Rational(3, 5)], [Rational(1, 2), Rational(1, 2), -S(1)]])
>>> C = ContinuousMarkovChain('C', state_space=[0, 1, 2], gen_mat=G)
>>> P(Eq(C(3.92), C(1.75)), Eq(C(0.46), 0)).round(5)
0.37933
>>> P(Gt(C(3.92), C(1.75)), Eq(C(0.46), 0)).round(5)
0.34211
>>> P(Le(C(1.57), C(3.14)), Eq(C(1.22), 1)).round(4)
0.7143
```

Symbolic probability queries are also supported

```
>>> from sympy import symbols
>>> a,b,c,d = symbols('a b c d')
>>> G = Matrix([[S(1), Rational(1, 10), Rational(9, 10)], [Rational(2, 5), S(1), Rational(3, 5)], [Rational(1, 2), Rational(1, 2), -S(1)]])
>>> C = ContinuousMarkovChain('C', state_space=[0, 1, 2], gen_mat=G)
>>> query = P(Eq(C(a), b), Eq(C(c), d))
>>> query.subs({a:3.65, b:2, c:1.78, d:1}).evalf().round(10)
0.4002723175
>>> P(Eq(C(3.65), 2), Eq(C(1.78), 1)).round(10)
0.4002723175
>>> query_gt = P(Gt(C(a), b), Eq(C(c), d))
>>> query_gt.subs({a:43.2, b:0, c:3.29, d:2}).evalf().round(10)
0.6832579186
```

(continues on next page)



(continued from previous page)

```
>>> P(Gt(C(43.2), 0), Eq(C(3.29), 2)).round(10)
0.6832579186
```

## References

[R935], [R936]

**class** `sympy.stats.BernoulliProcess(sym, p, success=1, failure=0)`

The Bernoulli process consists of repeated independent Bernoulli process trials with the same parameter  $p$ . It's assumed that the probability  $p$  applies to every trial and that the outcomes of each trial are independent of all the rest. Therefore Bernoulli Process is Discrete State and Discrete Time Stochastic Process.

### Parameters

**sym** : Symbol/str

**success** : Integer/str

The event which is considered to be success. Default: 1.

**failure**: Integer/str

The event which is considered to be failure. Default: 0.

**p** : Real Number between 0 and 1

Represents the probability of getting success.

## Examples

```
>>> from sympy.stats import BernoulliProcess, P, E
>>> from sympy import Eq, Gt
>>> B = BernoulliProcess("B", p=0.7, success=1, failure=0)
>>> B.state_space
{0, 1}
>>> (B.p).round(2)
0.70
>>> B.success
1
>>> B.failure
0
>>> X = B[1] + B[2] + B[3]
>>> P(Eq(X, 0)).round(2)
0.03
>>> P(Eq(X, 2)).round(2)
0.44
>>> P(Eq(X, 4)).round(2)
0
>>> P(Gt(X, 1)).round(2)
0.78
>>> P(Eq(B[1], 0) & Eq(B[2], 1) & Eq(B[3], 0) & Eq(B[4], 1)).round(2)
0.04
>>> B.joint_distribution(B[1], B[2])
```

(continues on next page)

(continued from previous page)

```
JointDistributionHandmade(Lambda((B[1], B[2]), Piecewise((0.7, Eq(B[1], 1)),
↳ 1)),
(0.3, Eq(B[1], 0)), (0, True))*Piecewise((0.7, Eq(B[2], 1)), (0.3, 1),
↳ Eq(B[2], 0)),
(0, True))))
>>> E(2*B[1] + B[2]).round(2)
2.10
>>> P(B[1] < 1).round(2)
0.30
```

## References

[R937], [R938]

**expectation**(*expr*, *condition=None*, *evaluate=True*, *\*\*kwargs*)

Computes expectation.

### Parameters

**expr** : RandomIndexedSymbol, Relational, Logic

Condition for which expectation has to be computed. Must contain a RandomIndexedSymbol of the process.

**condition** : Relational, Logic

The given conditions under which computations should be done.

### Returns

Expectation of the RandomIndexedSymbol.

**probability**(*condition*, *given\_condition=None*, *evaluate=True*, *\*\*kwargs*)

Computes probability.

### Parameters

**condition** : Relational

Condition for which probability has to be computed. Must contain a RandomIndexedSymbol of the process.

**given\_condition** : Relational, Logic

The given conditions under which computations should be done.

### Returns

Probability of the condition.

**class** `sympy.stats.PoissonProcess`(*sym*, *lamda*)

The Poisson process is a counting process. It is usually used in scenarios where we are counting the occurrences of certain events that appear to happen at a certain rate, but completely at random.

### Parameters

**sym** : Symbol/str

**lamda** : Positive number

Rate of the process,  $\lambda > 0$

## Examples

```
>>> from sympy.stats import PoissonProcess, P, E
>>> from sympy import symbols, Eq, Ne, Contains, Interval
>>> X = PoissonProcess("X", lamda=3)
>>> X.state_space
Naturals0
>>> X.lamda
3
>>> t1, t2 = symbols('t1 t2', positive=True)
>>> P(X(t1) < 4)
(9*t1**3/2 + 9*t1**2/2 + 3*t1 + 1)*exp(-3*t1)
>>> P(Eq(X(t1), 2) | Ne(X(t1), 4), Contains(t1, Interval.Ropen(2, 4)))
1 - 36*exp(-6)
>>> P(Eq(X(t1), 2) & Eq(X(t2), 3), Contains(t1, Interval.Lopen(0, 2))
... & Contains(t2, Interval.Lopen(2, 4)))
648*exp(-12)
>>> E(X(t1))
3*t1
>>> E(X(t1)**2 + 2*X(t2), Contains(t1, Interval.Lopen(0, 1))
... & Contains(t2, Interval.Lopen(1, 2)))
18
>>> P(X(3) < 1, Eq(X(1), 0))
exp(-6)
>>> P(Eq(X(4), 3), Eq(X(2), 3))
exp(-6)
>>> P(X(2) <= 3, X(1) > 1)
5*exp(-3)
```

### Merging two Poisson Processes

```
>>> Y = PoissonProcess("Y", lamda=4)
>>> Z = X + Y
>>> Z.lamda
7
```

### Splitting a Poisson Process into two independent Poisson Processes

```
>>> N, M = Z.split(l1=2, l2=5)
>>> N.lamda, M.lamda
(2, 5)
```

## References

[R939], [R940]

**class** sympy.stats.WienerProcess(sym)

The Wiener process is a real valued continuous-time stochastic process. In physics it is used to study Brownian motion and it is often also called Brownian motion due to its historical connection with physical process of the same name originally observed by Scottish botanist Robert Brown.

**Parameters****sym** : Symbol/str**Examples**

```
>>> from sympy.stats import WienerProcess, P, E
>>> from sympy import symbols, Contains, Interval
>>> X = WienerProcess("X")
>>> X.state_space
Reals
>>> t1, t2 = symbols('t1 t2', positive=True)
>>> P(X(t1) < 7).simplify()
erf(7*sqrt(2)/(2*sqrt(t1)))/2 + 1/2
>>> P((X(t1) > 2) | (X(t1) < 4), Contains(t1, Interval.Ropen(2, 4))).
↪simplify()
-erf(1)/2 + erf(2)/2 + 1
>>> E(X(t1))
0
>>> E(X(t1) + 2*X(t2), Contains(t1, Interval.Lopen(0, 1))
... & Contains(t2, Interval.Lopen(1, 2)))
0
```

**References**[\[R941\]](#), [\[R942\]](#)**class** sympy.stats.GammaProcess(sym, lamda, gamma)

A Gamma process is a random process with independent gamma distributed increments. It is a pure-jump increasing Levy process.

**Parameters****sym** : Symbol/str**lamda** : Positive numberJump size of the process,  $\text{lamda} > 0$ **gamma** : Positive numberRate of jump arrivals,  $\gamma > 0$ **Examples**

```
>>> from sympy.stats import GammaProcess, E, P, variance
>>> from sympy import symbols, Contains, Interval, Not
>>> t, d, x, l, g = symbols('t d x l g', positive=True)
>>> X = GammaProcess("X", l, g)
>>> E(X(t))
g*t/l
>>> variance(X(t)).simplify()
g*t/l**2
>>> X = GammaProcess('X', 1, 2)
```

(continues on next page)

(continued from previous page)

```

>>> P(X(t) < 1).simplify()
lowergamma(2*t, 1)/gamma(2*t)
>>> P(Not((X(t) < 5) & (X(d) > 3)), Contains(t, Interval.Ropen(2, 4)) &
... Contains(d, Interval.Lopen(7, 8))).simplify()
-4*exp(-3) + 472*exp(-8)/3 + 1
>>> E(X(2) + x*E(X(5)))
10*x + 4

```

## References

[R943]

## Matrix Distributions

`sympy.stats.MatrixGamma(symbol, alpha, beta, scale_matrix)`  
 Creates a random variable with Matrix Gamma Distribution.  
 The density of the said distribution can be found at [1].

### Parameters

**alpha: Positive Real number**

Shape Parameter

**beta: Positive Real number**

Scale Parameter

**scale\_matrix: Positive definite real square matrix**

Scale Matrix

### Returns

RandomSymbol

## Examples

```

>>> from sympy.stats import density, MatrixGamma
>>> from sympy import MatrixSymbol, symbols
>>> a, b = symbols('a b', positive=True)
>>> M = MatrixGamma('M', a, b, [[2, 1], [1, 2]])
>>> X = MatrixSymbol('X', 2, 2)
>>> density(M)(X).doit()
exp(Trace(Matrix([
[-2/3, 1/3],
[1/3, -2/3]])*X)/b)*Determinant(X)**(a - 3/2)/
(3**a*sqrt(pi)*b**(2*a)*gamma(a)*gamma(a - 1/2))
>>> density(M)([[1, 0], [0, 1]]).doit()
exp(-4/(3*b))/(3**a*sqrt(pi)*b**(2*a)*gamma(a)*gamma(a - 1/2))

```

## References

[R944]

`sympy.stats.Wishart(symbol, n, scale_matrix)`

Creates a random variable with Wishart Distribution.

The density of the said distribution can be found at [1].

### Parameters

**n: Positive Real number**

Represents degrees of freedom

**scale\_matrix: Positive definite real square matrix**

Scale Matrix

### Returns

RandomSymbol

## Examples

```
>>> from sympy.stats import density, Wishart
>>> from sympy import MatrixSymbol, symbols
>>> n = symbols('n', positive=True)
>>> W = Wishart('W', n, [[2, 1], [1, 2]])
>>> X = MatrixSymbol('X', 2, 2)
>>> density(W)(X).doit()
exp(Trace(Matrix([
[-1/3, 1/6],
[1/6, -1/3]])*X))*Determinant(X)**(n/2 - 3/2)/(2**n*3**(n/
- 2)*sqrt(pi)*gamma(n/2)*gamma(n/2 - 1/2))
>>> density(W)([[1, 0], [0, 1]]).doit()
exp(-2/3)/(2**n*3**(n/2)*sqrt(pi)*gamma(n/2)*gamma(n/2 - 1/2))
```

## References

[R945]

`sympy.stats.MatrixNormal(symbol, location_matrix, scale_matrix_1, scale_matrix_2)`

Creates a random variable with Matrix Normal Distribution.

The density of the said distribution can be found at [1].

### Parameters

**location\_matrix: Real ``n x p`` matrix**

Represents degrees of freedom

**scale\_matrix\_1: Positive definite matrix**

Scale Matrix of shape  $n \times n$

**scale\_matrix\_2: Positive definite matrix**

Scale Matrix of shape  $p \times p$

**Returns**

RandomSymbol

**Examples**

```
>>> from sympy import MatrixSymbol
>>> from sympy.stats import density, MatrixNormal
>>> M = MatrixNormal('M', [[1, 2]], [1], [[1, 0], [0, 1]])
>>> X = MatrixSymbol('X', 1, 2)
>>> density(M)(X).doit()
2*exp(-Trace((Matrix([
[-1],
[-2]]) + X.T*(Matrix([[-1, -2]]) + X))/2)/pi
>>> density(M)([[3, 4]]).doit()
2*exp(-4)/pi
```

**References**

[R946]

**Compound Distribution**

**class** sympy.stats.compound\_rv.**CompoundDistribution**(*dist*)

Class for Compound Distributions.

**Parameters**

**dist** : Distribution

Distribution must contain a random parameter

**Examples**

```
>>> from sympy.stats.compound_rv import CompoundDistribution
>>> from sympy.stats.crv_types import NormalDistribution
>>> from sympy.stats import Normal
>>> from sympy.abc import x
>>> X = Normal('X', 2, 4)
>>> N = NormalDistribution(X, 4)
>>> C = CompoundDistribution(N)
>>> C.set
Interval(-oo, oo)
>>> C.pdf(x, evaluate=True).simplify()
exp(-x**2/64 + x/16 - 1/16)/(8*sqrt(pi))
```

## References

[R947]

## Interface

`sympy.stats.P(condition, given_condition=None, numsamples=None, evaluate=True, **kwargs)`

Probability that a condition is true, optionally given a second condition.

### Parameters

**condition** : Combination of Relationals containing RandomSymbols

The condition of which you want to compute the probability

**given\_condition** : Combination of Relationals containing RandomSymbols

A conditional expression.  $P(X > 1, X > 0)$  is expectation of  $X > 1$  given  $X > 0$

**numsamples** : int

Enables sampling and approximates the probability with this many samples

**evaluate** : Bool (defaults to True)

In case of continuous systems return unevaluated integral

## Examples

```
>>> from sympy.stats import P, Die
>>> from sympy import Eq
>>> X, Y = Die('X', 6), Die('Y', 6)
>>> P(X > 3)
1/2
>>> P(Eq(X, 5), X > 2) # Probability that X == 5 given that X > 2
1/4
>>> P(X > Y)
5/12
```

**class** `sympy.stats.Probability(prob, condition=None, **kwargs)`

Symbolic expression for the probability.

## Examples

```
>>> from sympy.stats import Probability, Normal
>>> from sympy import Integral
>>> X = Normal("X", 0, 1)
>>> prob = Probability(X > 1)
>>> prob
Probability(X > 1)
```



Integral representation:

```
>>> prob.rewrite(Integral)
Integral(sqrt(2)*exp(-_z**2/2)/(2*sqrt(pi)), (_z, 1, oo))
```

Evaluation of the integral:

```
>>> prob.evaluate_integral()
sqrt(2)*(-sqrt(2)*sqrt(pi)*erf(sqrt(2)/2) + sqrt(2)*sqrt(pi))/
→ (4*sqrt(pi))
```

`sympy.stats.E(expr, condition=None, numsamples=None, evaluate=True, **kwargs)`

Returns the expected value of a random expression.

#### Parameters

**expr** : Expr containing RandomSymbols

The expression of which you want to compute the expectation value

**given** : Expr containing RandomSymbols

A conditional expression.  $E(X, X > 0)$  is expectation of  $X$  given  $X > 0$

**numsamples** : int

Enables sampling and approximates the expectation with this many samples

**evalf** : Bool (defaults to True)

If sampling return a number rather than a complex expression

**evaluate** : Bool (defaults to True)

In case of continuous systems return unevaluated integral

#### Examples

```
>>> from sympy.stats import E, Die
>>> X = Die('X', 6)
>>> E(X)
7/2
>>> E(2*X + 1)
8
```

```
>>> E(X, X > 3) # Expectation of X given that it is above 3
5
```

**class** `sympy.stats.Expectation(expr, condition=None, **kwargs)`

Symbolic expression for the expectation.

## Examples

```
>>> from sympy.stats import Expectation, Normal, Probability, Poisson
>>> from sympy import symbols, Integral, Sum
>>> mu = symbols("mu")
>>> sigma = symbols("sigma", positive=True)
>>> X = Normal("X", mu, sigma)
>>> Expectation(X)
Expectation(X)
>>> Expectation(X).evaluate_integral().simplify()
mu
```

To get the integral expression of the expectation:

```
>>> Expectation(X).rewrite(Integral)
Integral(sqrt(2)*X*exp(-(X - mu)**2/(2*sigma**2))/(2*sqrt(pi)*sigma), (X,
↪ -oo, oo))
```

The same integral expression, in more abstract terms:

```
>>> Expectation(X).rewrite(Probability)
Integral(x*Probability(Eq(X, x)), (x, -oo, oo))
```

To get the Summation expression of the expectation for discrete random variables:

```
>>> lamda = symbols('lamda', positive=True)
>>> Z = Poisson('Z', lamda)
>>> Expectation(Z).rewrite(Sum)
Sum(Z*lamda**Z*exp(-lamda)/factorial(Z), (Z, 0, oo))
```

This class is aware of some properties of the expectation:

```
>>> from sympy.abc import a
>>> Expectation(a*X)
Expectation(a*X)
>>> Y = Normal("Y", 1, 2)
>>> Expectation(X + Y)
Expectation(X + Y)
```

To expand the Expectation into its expression, use `expand()`:

```
>>> Expectation(X + Y).expand()
Expectation(X) + Expectation(Y)
>>> Expectation(a*X + Y).expand()
a*Expectation(X) + Expectation(Y)
>>> Expectation(a*X + Y)
Expectation(a*X + Y)
>>> Expectation((X + Y)*(X - Y)).expand()
Expectation(X**2) - Expectation(Y**2)
```

To evaluate the Expectation, use `doit()`:

```
>>> Expectation(X + Y).doit()
mu + 1
```

(continues on next page)

(continued from previous page)

```
>>> Expectation(X + Expectation(Y + Expectation(2*X))).doit()
3*mu + 1
```

To prevent evaluating nested Expectation, use `doit(deep=False)`

```
>>> Expectation(X + Expectation(Y)).doit(deep=False)
mu + Expectation(Expectation(Y))
>>> Expectation(X + Expectation(Y + Expectation(2*X))).doit(deep=False)
mu + Expectation(Expectation(Y + Expectation(2*X)))
```

`sympy.stats.density(expr, condition=None, evaluate=True, numsamples=None, **kwargs)`

Probability density of a random expression, optionally given a second condition.

#### Parameters

**expr** : Expr containing RandomSymbols

The expression of which you want to compute the density value

**condition** : Relational containing RandomSymbols

A conditional expression. `density(X > 1, X > 0)` is density of  $X > 1$  given  $X > 0$

**numsamples** : int

Enables sampling and approximates the density with this many samples

#### Explanation

This density will take on different forms for different types of probability spaces. Discrete variables produce Dicts. Continuous variables produce Lambdas.

#### Examples

```
>>> from sympy.stats import density, Die, Normal
>>> from sympy import Symbol
```

```
>>> x = Symbol('x')
>>> D = Die('D', 6)
>>> X = Normal(x, 0, 1)
```

```
>>> density(D).dict
{1: 1/6, 2: 1/6, 3: 1/6, 4: 1/6, 5: 1/6, 6: 1/6}
>>> density(2*D).dict
{2: 1/6, 4: 1/6, 6: 1/6, 8: 1/6, 10: 1/6, 12: 1/6}
>>> density(X)(x)
sqrt(2)*exp(-x**2/2)/(2*sqrt(pi))
```

`sympy.stats.entropy(expr, condition=None, **kwargs)`

Calculates entropy of a probability distribution.

**Parameters**

**expression** : the random expression whose entropy is to be calculated

**condition** : optional, to specify conditions on random expression

**b: base of the logarithm, optional**

By default, it is taken as Euler's number

**Returns**

**result** : Entropy of the expression, a constant

**Examples**

```
>>> from sympy.stats import Normal, Die, entropy
>>> X = Normal('X', 0, 1)
>>> entropy(X)
log(2)/2 + 1/2 + log(pi)/2
```

```
>>> D = Die('D', 4)
>>> entropy(D)
log(4)
```

**References**

[R948], [R949], [R950]

`sympy.stats.given(expr, condition=None, **kwargs)`

Conditional Random Expression.

**Explanation**

From a random expression and a condition on that expression creates a new probability space from the condition and returns the same expression on that conditional probability space.

**Examples**

```
>>> from sympy.stats import given, density, Die
>>> X = Die('X', 6)
>>> Y = given(X, X > 3)
>>> density(Y).dict
{4: 1/3, 5: 1/3, 6: 1/3}
```

Following convention, if the condition is a random symbol then that symbol is considered fixed.

```
>>> from sympy.stats import Normal
>>> from sympy import pprint
>>> from sympy.abc import z
```

```
>>> X = Normal('X', 0, 1)
>>> Y = Normal('Y', 0, 1)
>>> pprint(density(X + Y, Y)(z), use_unicode=False)
 2
 -(-Y + z)

 2
 \ / 2 *e

 2*\ / pi
```

`sympy.stats.where(condition, given_condition=None, **kwargs)`  
Returns the domain where a condition is True.

### Examples

```
>>> from sympy.stats import where, Die, Normal
>>> from sympy import And
```

```
>>> D1, D2 = Die('a', 6), Die('b', 6)
>>> a, b = D1.symbol, D2.symbol
>>> X = Normal('x', 0, 1)
```

```
>>> where(X**2<1)
Domain: (-1 < x) & (x < 1)
```

```
>>> where(X**2<1).set
Interval.open(-1, 1)
```

```
>>> where(And(D1<=D2, D2<3))
Domain: (Eq(a, 1) & Eq(b, 1)) | (Eq(a, 1) & Eq(b, 2)) | (Eq(a, 2) & Eq(b,
↪ 2))
```

`sympy.stats.variance(X, condition=None, **kwargs)`  
Variance of a random expression.

$$\text{variance}(X) = E((X - E(X))^2)$$

### Examples

```
>>> from sympy.stats import Die, Bernoulli, variance
>>> from sympy import simplify, Symbol
```

```
>>> X = Die('X', 6)
>>> p = Symbol('p')
>>> B = Bernoulli('B', p, 1, 0)
```

```
>>> variance(2*X)
35/3
```

```
>>> simplify(variance(B))
p*(1 - p)
```

**class** `sympy.stats.Variance`(*arg*, *condition=None*, *\*\*kwargs*)  
 Symbolic expression for the variance.

## Examples

```
>>> from sympy import symbols, Integral
>>> from sympy.stats import Normal, Expectation, Variance, Probability
>>> mu = symbols("mu", positive=True)
>>> sigma = symbols("sigma", positive=True)
>>> X = Normal("X", mu, sigma)
>>> Variance(X)
Variance(X)
>>> Variance(X).evaluate_integral()
sigma**2
```

Integral representation of the underlying calculations:

```
>>> Variance(X).rewrite(Integral)
Integral(sqrt(2)*(X - Integral(sqrt(2)*X*exp(-(X - mu)**2/(2*sigma**2))/
→ (2*sqrt(pi)*sigma), (X, -oo, oo)))**2*exp(-(X - mu)**2/(2*sigma**2))/
→ (2*sqrt(pi)*sigma), (X, -oo, oo))
```

Integral representation, without expanding the PDF:

```
>>> Variance(X).rewrite(Probability)
-Integral(x*Probability(Eq(X, x)), (x, -oo, oo))**2 +
→ Integral(x**2*Probability(Eq(X, x)), (x, -oo, oo))
```

Rewrite the variance in terms of the expectation

```
>>> Variance(X).rewrite(Expectation)
-Expectation(X)**2 + Expectation(X**2)
```

Some transformations based on the properties of the variance may happen:

```
>>> from sympy.abc import a
>>> Y = Normal("Y", 0, 1)
>>> Variance(a*X)
Variance(a*X)
```

To expand the variance in its expression, use `expand()`:

```
>>> Variance(a*X).expand()
a**2*Variance(X)
>>> Variance(X + Y)
Variance(X + Y)
```

(continues on next page)

(continued from previous page)

```
>>> Variance(X + Y).expand()
2*Covariance(X, Y) + Variance(X) + Variance(Y)
```

`sympy.stats.covariance(X, Y, condition=None, **kwargs)`  
Covariance of two random expressions.

### Explanation

The expectation that the two variables will rise and fall together

$$\text{covariance}(X, Y) = E((X - E(X))(Y - E(Y)))$$

### Examples

```
>>> from sympy.stats import Exponential, covariance
>>> from sympy import Symbol
```

```
>>> rate = Symbol('lambda', positive=True, real=True)
>>> X = Exponential('X', rate)
>>> Y = Exponential('Y', rate)
```

```
>>> covariance(X, X)
lambda**(-2)
>>> covariance(X, Y)
0
>>> covariance(X, Y + rate*X)
1/lambda
```

**class** `sympy.stats.Covariance(arg1, arg2, condition=None, **kwargs)`  
Symbolic expression for the covariance.

### Examples

```
>>> from sympy.stats import Covariance
>>> from sympy.stats import Normal
>>> X = Normal("X", 3, 2)
>>> Y = Normal("Y", 0, 1)
>>> Z = Normal("Z", 0, 1)
>>> W = Normal("W", 0, 1)
>>> cexpr = Covariance(X, Y)
>>> cexpr
Covariance(X, Y)
```

Evaluate the covariance,  $X$  and  $Y$  are independent, therefore zero is the result:

```
>>> cexpr.evaluate_integral()
0
```

Rewrite the covariance expression in terms of expectations:

```
>>> from sympy.stats import Expectation
>>> cexpr.rewrite(Expectation)
Expectation(X*Y) - Expectation(X)*Expectation(Y)
```

In order to expand the argument, use `expand()`:

```
>>> from sympy.abc import a, b, c, d
>>> Covariance(a*X + b*Y, c*Z + d*W)
Covariance(a*X + b*Y, c*Z + d*W)
>>> Covariance(a*X + b*Y, c*Z + d*W).expand()
a*c*Covariance(X, Z) + a*d*Covariance(W, X) + b*c*Covariance(Y, Z) +
→ b*d*Covariance(W, Y)
```

This class is aware of some properties of the covariance:

```
>>> Covariance(X, X).expand()
Variance(X)
>>> Covariance(a*X, b*Y).expand()
a*b*Covariance(X, Y)
```

`sympy.stats.coskewness(X, Y, Z, condition=None, **kwargs)`

Calculates the co-skewness of three random variables.

#### Parameters

**X** : RandomSymbol

Random Variable used to calculate coskewness

**Y** : RandomSymbol

Random Variable used to calculate coskewness

**Z** : RandomSymbol

Random Variable used to calculate coskewness

**condition** : Expr containing RandomSymbols

A conditional expression

#### Returns

**coskewness** : The coskewness of the three random variables

#### Explanation

Mathematically Coskewness is defined as

$$\text{coskewness}(X, Y, Z) = \frac{E[(X - E[X]) * (Y - E[Y]) * (Z - E[Z])]}{\sigma_X \sigma_Y \sigma_Z}$$



## Examples

```
>>> from sympy.stats import coskewness, Exponential, skewness
>>> from sympy import symbols
>>> p = symbols('p', positive=True)
>>> X = Exponential('X', p)
>>> Y = Exponential('Y', 2*p)
>>> coskewness(X, Y, Y)
0
>>> coskewness(X, Y + X, Y + 2*X)
16*sqrt(85)/85
>>> coskewness(X + 2*Y, Y + X, Y + 2*X, X > 3)
9*sqrt(170)/85
>>> coskewness(Y, Y, Y) == skewness(Y)
True
>>> coskewness(X, Y + p*X, Y + 2*p*X)
4/(sqrt(1 + 1/(4*p**2))*sqrt(4 + 1/(4*p**2)))
```

## References

[R951]

`sympy.stats.median(X, evaluate=True, **kwargs)`

Calculates the median of the probability distribution.

### Parameters

**X:** The random expression whose median is to be calculated.

### Returns

The FiniteSet or an Interval which contains the median of the random expression.

## Explanation

Mathematically, median of Probability distribution is defined as all those values of  $m$  for which the following condition is satisfied

$$P(X \leq m) \geq \frac{1}{2} \text{ and } P(X \geq m) \geq \frac{1}{2}$$

## Examples

```
>>> from sympy.stats import Normal, Die, median
>>> N = Normal('N', 3, 1)
>>> median(N)
{3}
>>> D = Die('D')
>>> median(D)
{3, 4}
```

## References

[R952]

`sympy.stats.std(X, condition=None, **kwargs)`

Standard Deviation of a random expression

$$\text{std}(X) = \sqrt{E((X - E(X))^2)}$$

## Examples

```
>>> from sympy.stats import Bernoulli, std
>>> from sympy import Symbol, simplify
```

```
>>> p = Symbol('p')
>>> B = Bernoulli('B', p, 1, 0)
```

```
>>> simplify(std(B))
sqrt(p*(1 - p))
```

`sympy.stats.quantile(expr, evaluate=True, **kwargs)`

Return the  $p^{\text{th}}$  order quantile of a probability distribution.

## Explanation

Quantile is defined as the value at which the probability of the random variable is less than or equal to the given probability.

$$Q(p) = \inf\{x \in (-\infty, \infty) : p \leq F(x)\}$$

## Examples

```
>>> from sympy.stats import quantile, Die, Exponential
>>> from sympy import Symbol, pprint
>>> p = Symbol("p")
```

```
>>> l = Symbol("lambda", positive=True)
>>> X = Exponential("x", l)
>>> quantile(X)(p)
-log(1 - p)/lambda
```

```
>>> D = Die("d", 6)
>>> pprint(quantile(D)(p), use_unicode=False)
/nan for 0<(p > 1, p < 0)
|
| 1 for p <= 1/6
|
| 2 for p <= 1/3
```

(continues on next page)

(continued from previous page)

```
|
< 3 for p <= 1/2
|
| 4 for p <= 2/3
|
| 5 for p <= 5/6
|
\ 6 for p <= 1
```

```
sympy.stats.sample(expr, condition=None, size=(), library='scipy', numsamples=1,
 seed=None, **kwargs)
```

A realization of the random expression.

### Parameters

**expr** : Expression of random variables

Expression from which sample is extracted

**condition** : Expr containing RandomSymbols

A conditional expression

**size** : int, tuple

Represents size of each sample in numsamples

**library** : str

- 'scipy' : Sample using scipy
- 'numpy' : Sample using numpy
- 'pymc' : Sample using PyMC

Choose any of the available options to sample from as string, by default is 'scipy'

**numsamples** : int

Number of samples, each with size as size.

Deprecated since version 1.9.

The numsamples parameter is deprecated and is only provided for compatibility with v1.8. Use a list comprehension or an additional dimension in size instead. See [sympy.stats.sample\(numsamples=n\)](#) (page 168) for details.

**seed** :

An object to be used as seed by the given external library for sampling *expr*. Following is the list of possible types of object for the supported libraries,

- 'scipy': int, numpy.random.RandomState,  
numpy.random.Generator
- 'numpy': int, numpy.random.RandomState,  
numpy.random.Generator
- 'pymc': int

Optional, by default None, in which case seed settings related to the given library will be used. No modifications to environment's global seed settings are done by this argument.

**Returns**

sample: float/list/numpy.ndarray

one sample or a collection of samples of the random expression.

- sample(X) returns float/numpy.float64/numpy.int64 object.
- sample(X, size=int/tuple) returns numpy.ndarray object.

**Examples**

```
>>> from sympy.stats import Die, sample, Normal, Geometric
>>> X, Y, Z = Die('X', 6), Die('Y', 6), Die('Z', 6) # Finite Random
↳Variable
>>> die_roll = sample(X + Y + Z)
>>> die_roll
3
>>> N = Normal('N', 3, 4) # Continuous Random Variable
>>> samp = sample(N)
>>> samp in N.pspace.domain.set
True
>>> samp = sample(N, N>0)
>>> samp > 0
True
>>> samp_list = sample(N, size=4)
>>> [sam in N.pspace.domain.set for sam in samp_list]
[True, True, True, True]
>>> sample(N, size = (2,3))
array([[5.42519758, 6.40207856, 4.94991743],
 [1.85819627, 6.83403519, 1.9412172]])
>>> G = Geometric('G', 0.5) # Discrete Random Variable
>>> samp_list = sample(G, size=3)
>>> samp_list
[1, 3, 2]
>>> [sam in G.pspace.domain.set for sam in samp_list]
[True, True, True]
>>> MN = Normal("MN", [3, 4], [[2, 1], [1, 2]]) # Joint Random Variable
>>> samp_list = sample(MN, size=4)
>>> samp_list
[array([2.85768055, 3.38954165]),
 array([4.11163337, 4.3176591]),
 array([0.79115232, 1.63232916]),
 array([4.01747268, 3.96716083])]
>>> [tuple(sam) in MN.pspace.domain.set for sam in samp_list]
[True, True, True, True]
```

Changed in version 1.7.0: sample used to return an iterator containing the samples instead of value.

Changed in version 1.9.0: sample returns values or array of values instead of an iterator and numsamples is deprecated.

`sympy.stats.sample_iter(expr, condition=None, size=(), library='scipy',  
numsamples=oo, seed=None, **kwargs)`

Returns an iterator of realizations from the expression given a condition.

#### Parameters

**expr: Expr**

Random expression to be realized

**condition: Expr, optional**

A conditional expression

**size : int, tuple**

Represents size of each sample in numsamples

**numsamples: integer, optional**

Length of the iterator (defaults to infinity)

**seed :**

An object to be used as seed by the given external library for sampling *expr*. Following is the list of possible types of object for the supported libraries,

- 'scipy': int, numpy.random.RandomState,  
numpy.random.Generator
- 'numpy': int, numpy.random.RandomState,  
numpy.random.Generator
- 'pymc': int

Optional, by default None, in which case seed settings related to the given library will be used. No modifications to environment's global seed settings are done by this argument.

#### Returns

sample\_iter: iterator object

iterator object containing the sample/samples of given expr

#### Examples

```
>>> from sympy.stats import Normal, sample_iter
>>> X = Normal('X', 0, 1)
>>> expr = X*X + 3
>>> iterator = sample_iter(expr, numsamples=3)
>>> list(iterator)
[12, 4, 7]
```

#### See also:

[sample](#) (page 2971), [sampling\\_P](#) (page 2976), [sampling\\_E](#) (page 2976)

`sympy.stats.factorial_moment(X, n, condition=None, **kwargs)`

The factorial moment is a mathematical quantity defined as the expectation or average of the falling factorial of a random variable.

$$\text{factorial-moment}(X, n) = E(X(X-1)(X-2)\dots(X-n+1))$$

**Parameters****n**: A natural number, n-th factorial moment.**condition** : Expr containing RandomSymbols

A conditional expression.

**Examples**

```
>>> from sympy.stats import factorial_moment, Poisson, Binomial
>>> from sympy import Symbol, S
>>> lamda = Symbol('lamda')
>>> X = Poisson('X', lamda)
>>> factorial_moment(X, 2)
lamda**2
>>> Y = Binomial('Y', 2, S.Half)
>>> factorial_moment(Y, 2)
1/2
>>> factorial_moment(Y, 2, Y > 1) # find factorial moment for Y > 1
2
```

**References**[\[R953\]](#), [\[R954\]](#)`sympy.stats.kurtosis(X, condition=None, **kwargs)`

Characterizes the tails/outliers of a probability distribution.

**Parameters****condition** : Expr containing RandomSymbolsA conditional expression. `kurtosis(X, X>0)` is kurtosis of X given  $X > 0$ **Explanation**

Kurtosis of any univariate normal distribution is 3. Kurtosis less than 3 means that the distribution produces fewer and less extreme outliers than the normal distribution.

$$kurtosis(X) = E(((X - E(X))/\sigma_X)^4)$$

**Examples**

```
>>> from sympy.stats import kurtosis, Exponential, Normal
>>> from sympy import Symbol
>>> X = Normal('X', 0, 1)
>>> kurtosis(X)
3
>>> kurtosis(X, X > 0) # find kurtosis given X > 0
(-4/pi - 12/pi**2 + 3)/(1 - 2/pi)**2
```

```
>>> rate = Symbol('lamda', positive=True, real=True)
>>> Y = Exponential('Y', rate)
>>> kurtosis(Y)
9
```

## References

[R955], [R956]

`sympy.stats.skewness(X, condition=None, **kwargs)`

Measure of the asymmetry of the probability distribution.

### Parameters

**condition** : Expr containing RandomSymbols

A conditional expression. `skewness(X, X>0)` is skewness of X given  $X > 0$

## Explanation

Positive skew indicates that most of the values lie to the right of the mean.

$$\text{skewness}(X) = E(((X - E(X))/\sigma_X)^3)$$

## Examples

```
>>> from sympy.stats import skewness, Exponential, Normal
>>> from sympy import Symbol
>>> X = Normal('X', 0, 1)
>>> skewness(X)
0
>>> skewness(X, X > 0) # find skewness given X > 0
(-sqrt(2)/sqrt(pi) + 4*sqrt(2)/pi**(3/2))/(1 - 2/pi)**(3/2)
```

```
>>> rate = Symbol('lambda', positive=True, real=True)
>>> Y = Exponential('Y', rate)
>>> skewness(Y)
2
```

`sympy.stats.correlation(X, Y, condition=None, **kwargs)`

Correlation of two random expressions, also known as correlation coefficient or Pearson's correlation.

## Explanation

The normalized expectation that the two variables will rise and fall together

$$\text{correlation}(X, Y) = E((X - E(X))(Y - E(Y)) / (\sigma_x \sigma_y))$$

## Examples

```
>>> from sympy.stats import Exponential, correlation
>>> from sympy import Symbol
```

```
>>> rate = Symbol('lambda', positive=True, real=True)
>>> X = Exponential('X', rate)
>>> Y = Exponential('Y', rate)
```

```
>>> correlation(X, X)
1
>>> correlation(X, Y)
0
>>> correlation(X, Y + rate*X)
1/sqrt(1 + lambda**(-2))
```

`sympy.stats.rv.sampling_density(expr, given_condition=None, library='scipy', numsamples=1, seed=None, **kwargs)`

Sampling version of density.

### See also:

[density](#) (page 2963), [sampling\\_P](#) (page 2976), [sampling\\_E](#) (page 2976)

`sympy.stats.rv.sampling_P(condition, given_condition=None, library='scipy', numsamples=1, evalf=True, seed=None, **kwargs)`

Sampling version of P.

### See also:

[P](#) (page 2960), [sampling\\_E](#) (page 2976), [sampling\\_density](#) (page 2976)

`sympy.stats.rv.sampling_E(expr, given_condition=None, library='scipy', numsamples=1, evalf=True, seed=None, **kwargs)`

Sampling version of E.

### See also:

[P](#) (page 2960), [sampling\\_P](#) (page 2976), [sampling\\_density](#) (page 2976)

**class** `sympy.stats.Moment(X, n, c=0, condition=None, **kwargs)`

Symbolic class for Moment



## Examples

```
>>> from sympy import Symbol, Integral
>>> from sympy.stats import Normal, Expectation, Probability, Moment
>>> mu = Symbol('mu', real=True)
>>> sigma = Symbol('sigma', positive=True)
>>> X = Normal('X', mu, sigma)
>>> M = Moment(X, 3, 1)
```

To evaluate the result of Moment use *doit*:

```
>>> M.doit()
mu**3 - 3*mu**2 + 3*mu*sigma**2 + 3*mu - 3*sigma**2 - 1
```

Rewrite the Moment expression in terms of Expectation:

```
>>> M.rewrite(Expectation)
Expectation((X - 1)**3)
```

Rewrite the Moment expression in terms of Probability:

```
>>> M.rewrite(Probability)
Integral((x - 1)**3*Probability(Eq(X, x)), (x, -oo, oo))
```

Rewrite the Moment expression in terms of Integral:

```
>>> M.rewrite(Integral)
Integral(sqrt(2)*(X - 1)**3*exp(-(X - mu)**2/(2*sigma**2))/
 ↳ (2*sqrt(pi)*sigma), (X, -oo, oo))
```

`sympy.stats.moment(X, n, c=0, condition=None, *, evaluate=True, **kwargs)`

Return the nth moment of a random expression about c.

$$\text{moment}(X, c, n) = E((X - c)^n)$$

Default value of c is 0.

## Examples

```
>>> from sympy.stats import Die, moment, E
>>> X = Die('X', 6)
>>> moment(X, 1, 6)
-5/2
>>> moment(X, 2)
91/6
>>> moment(X, 1) == E(X)
True
```

`class sympy.stats.CentralMoment(X, n, condition=None, **kwargs)`

Symbolic class Central Moment

## Examples

```
>>> from sympy import Symbol, Integral
>>> from sympy.stats import Normal, Expectation, Probability,
↳ CentralMoment
>>> mu = Symbol('mu', real=True)
>>> sigma = Symbol('sigma', positive=True)
>>> X = Normal('X', mu, sigma)
>>> CM = CentralMoment(X, 4)
```

To evaluate the result of CentralMoment use *doit*:

```
>>> CM.doit().simplify()
3*sigma**4
```

Rewrite the CentralMoment expression in terms of Expectation:

```
>>> CM.rewrite(Expectation)
Expectation((X - Expectation(X))**4)
```

Rewrite the CentralMoment expression in terms of Probability:

```
>>> CM.rewrite(Probability)
Integral((x - Integral(x*Probability(True), (x, -oo,
↳ oo)))**4*Probability(Eq(X, x)), (x, -oo, oo))
```

Rewrite the CentralMoment expression in terms of Integral:

```
>>> CM.rewrite(Integral)
Integral(sqrt(2)*(X - Integral(sqrt(2)*X*exp(-(X - mu)**2/(2*sigma**2))/
↳ (2*sqrt(pi)*sigma), (X, -oo, oo)))**4*exp(-(X - mu)**2/(2*sigma**2))/
↳ (2*sqrt(pi)*sigma), (X, -oo, oo))
```

`sympy.stats.cmoment(X, n, condition=None, *, evaluate=True, **kwargs)`

Return the *n*th central moment of a random expression about its mean.

$$cmoment(X, n) = E((X - E(X))^n)$$

## Examples

```
>>> from sympy.stats import Die, cmoment, variance
>>> X = Die('X', 6)
>>> cmoment(X, 3)
0
>>> cmoment(X, 2)
35/12
>>> cmoment(X, 2) == variance(X)
True
```

**class** `sympy.stats.ExpectationMatrix(expr, condition=None)`

Expectation of a random matrix expression.

## Examples

```
>>> from sympy.stats import ExpectationMatrix, Normal
>>> from sympy.stats.rv import RandomMatrixSymbol
>>> from sympy import symbols, MatrixSymbol, Matrix
>>> k = symbols("k")
>>> A, B = MatrixSymbol("A", k, k), MatrixSymbol("B", k, k)
>>> X, Y = RandomMatrixSymbol("X", k, 1), RandomMatrixSymbol("Y", k, 1)
>>> ExpectationMatrix(X)
ExpectationMatrix(X)
>>> ExpectationMatrix(A*X).shape
(k, 1)
```

To expand the expectation in its expression, use `expand()`:

```
>>> ExpectationMatrix(A*X + B*Y).expand()
A*ExpectationMatrix(X) + B*ExpectationMatrix(Y)
>>> ExpectationMatrix((X + Y)*(X - Y).T).expand()
ExpectationMatrix(X*X.T) - ExpectationMatrix(X*Y.T) +
↳ ExpectationMatrix(Y*X.T) - ExpectationMatrix(Y*Y.T)
```

To evaluate the `ExpectationMatrix`, use `doit()`:

```
>>> N11, N12 = Normal('N11', 11, 1), Normal('N12', 12, 1)
>>> N21, N22 = Normal('N21', 21, 1), Normal('N22', 22, 1)
>>> M11, M12 = Normal('M11', 1, 1), Normal('M12', 2, 1)
>>> M21, M22 = Normal('M21', 3, 1), Normal('M22', 4, 1)
>>> x1 = Matrix([[N11, N12], [N21, N22]])
>>> x2 = Matrix([[M11, M12], [M21, M22]])
>>> ExpectationMatrix(x1 + x2).doit()
Matrix([
[12, 14],
[24, 26]])
```

**class** `sympy.stats.VarianceMatrix`(*arg*, *condition=None*)

Variance of a random matrix probability expression. Also known as Covariance matrix, auto-covariance matrix, dispersion matrix, or variance-covariance matrix.

## Examples

```
>>> from sympy.stats import VarianceMatrix
>>> from sympy.stats.rv import RandomMatrixSymbol
>>> from sympy import symbols, MatrixSymbol
>>> k = symbols("k")
>>> A, B = MatrixSymbol("A", k, k), MatrixSymbol("B", k, k)
>>> X, Y = RandomMatrixSymbol("X", k, 1), RandomMatrixSymbol("Y", k, 1)
>>> VarianceMatrix(X)
VarianceMatrix(X)
>>> VarianceMatrix(X).shape
(k, k)
```

To expand the variance in its expression, use `expand()`:

```
>>> VarianceMatrix(A*X).expand()
A*VarianceMatrix(X)*A.T
>>> VarianceMatrix(A*X + B*Y).expand()
2*A*CrossCovarianceMatrix(X, Y)*B.T + A*VarianceMatrix(X)*A.T +
↳ B*VarianceMatrix(Y)*B.T
```

**class** `sympy.stats.CrossCovarianceMatrix(arg1, arg2, condition=None)`  
Covariance of a random matrix probability expression.

### Examples

```
>>> from sympy.stats import CrossCovarianceMatrix
>>> from sympy.stats.rv import RandomMatrixSymbol
>>> from sympy import symbols, MatrixSymbol
>>> k = symbols("k")
>>> A, B = MatrixSymbol("A", k, k), MatrixSymbol("B", k, k)
>>> C, D = MatrixSymbol("C", k, k), MatrixSymbol("D", k, k)
>>> X, Y = RandomMatrixSymbol("X", k, 1), RandomMatrixSymbol("Y", k, 1)
>>> Z, W = RandomMatrixSymbol("Z", k, 1), RandomMatrixSymbol("W", k, 1)
>>> CrossCovarianceMatrix(X, Y)
CrossCovarianceMatrix(X, Y)
>>> CrossCovarianceMatrix(X, Y).shape
(k, k)
```

To expand the covariance in its expression, use `expand()`:

```
>>> CrossCovarianceMatrix(X + Y, Z).expand()
CrossCovarianceMatrix(X, Z) + CrossCovarianceMatrix(Y, Z)
>>> CrossCovarianceMatrix(A*X, Y).expand()
A*CrossCovarianceMatrix(X, Y)
>>> CrossCovarianceMatrix(A*X, B.T*Y).expand()
A*CrossCovarianceMatrix(X, Y)*B
>>> CrossCovarianceMatrix(A*X + B*Y, C.T*Z + D.T*W).expand()
A*CrossCovarianceMatrix(X, W)*D + A*CrossCovarianceMatrix(X, Z)*C +
↳ B*CrossCovarianceMatrix(Y, W)*D + B*CrossCovarianceMatrix(Y, Z)*C
```

## Mechanics

SymPy Stats employs a relatively complex class hierarchy.

RandomDomains are a mapping of variables to possible values. For example, we might say that the symbol `Symbol('x')` can take on the values  $\{1, 2, 3, 4, 5, 6\}$ .

**class** `sympy.stats.rv.RandomDomain`

A PSpace, or Probability Space, combines a RandomDomain with a density to provide probabilistic information. For example the above domain could be enhanced by a finite density  $\{1:1/6, 2:1/6, 3:1/6, 4:1/6, 5:1/6, 6:1/6\}$  to fully define the roll of a fair die named `x`.

**class** `sympy.stats.rv.PSpace`

A `RandomSymbol` represents the `PSpace`'s symbol 'x' inside of SymPy expressions.

```
class sympy.stats.rv.RandomSymbol
```

The `RandomDomain` and `PSpace` classes are almost never directly instantiated. Instead they are subclassed for a variety of situations.

`RandomDomains` and `PSpaces` must be sufficiently general to represent domains and spaces of several variables with arbitrarily complex densities. This generality is often unnecessary. Instead we often build `SingleDomains` and `SinglePSpaces` to represent single, univariate events and processes such as a single die or a single normal variable.

```
class sympy.stats.rv.SinglePSpace
```

```
class sympy.stats.rv.SingleDomain
```

Another common case is to collect together a set of such univariate random variables. A collection of independent `SinglePSpaces` or `SingleDomains` can be brought together to form a `ProductDomain` or `ProductPSpace`. These objects would be useful in representing three dice rolled together for example.

```
class sympy.stats.rv.ProductDomain
```

```
class sympy.stats.rv.ProductPSpace
```

The `Conditional` adjective is added whenever we add a global condition to a `RandomDomain` or `PSpace`. A common example would be three independent dice where we know their sum to be greater than 12.

```
class sympy.stats.rv.ConditionalDomain
```

We specialize further into `Finite` and `Continuous` versions of these classes to represent finite (such as dice) and continuous (such as normals) random variables.

```
class sympy.stats.frv.FiniteDomain
```

```
class sympy.stats.frv.FinitePSpace
```

```
class sympy.stats.crv.ContinuousDomain
```

```
class sympy.stats.crv.ContinuousPSpace
```

Additionally there are a few specialized classes that implement certain common random variable types. There is for example a `DiePSpace` that implements `SingleFinitePSpace` and a `NormalPSpace` that implements `SingleContinuousPSpace`.

```
class sympy.stats.frv_types.DiePSpace
```

```
class sympy.stats.crv_types.NormalPSpace
```

`RandomVariables` can be extracted from these objects using the `PSpace.values` method.

As previously mentioned SymPy Stats employs a relatively complex class structure. Inheritance is widely used in the implementation of end-level classes. This tactic was chosen to balance between the need to allow SymPy to represent arbitrarily defined random variables and optimizing for common cases. This complicates the code but is structured to only be important to those working on extending SymPy Stats to other random variable types.

Users will not use this class structure. Instead these mechanics are exposed through variable creation functions `Die`, `Coin`, `FiniteRV`, `Normal`, `Exponential`, etc.... These build the appropriate `SinglePSpaces` and return the corresponding `RandomVariable`. `Conditional` and `Product`

spaces are formed in the natural construction of SymPy expressions and the use of interface functions `E`, `Given`, `Density`, etc....

`sympy.stats.Die()`

`sympy.stats.Normal()`

There are some additional functions that may be useful. They are largely used internally.

`sympy.stats.rv.random_symbols(expr)`

Returns all RandomSymbols within a SymPy Expression.

`sympy.stats.rv.pspace(expr)`

Returns the underlying Probability Space of a random expression.

For internal use.

### Examples

```
>>> from sympy.stats import pspace, Normal
>>> X = Normal('X', 0, 1)
>>> pspace(2*X + 1) == X.pspace
True
```

`sympy.stats.rv.rs_swap(a, b)`

Build a dictionary to swap RandomSymbols based on their underlying symbol.

i.e. if  $X = ('x', \text{pspace1})$  and  $Y = ('x', \text{pspace2})$  then  $X$  and  $Y$  match and the key, value pair  $\{X:Y\}$  will appear in the result

Inputs: collections  $a$  and  $b$  of random variables which share common symbols Output: dict mapping RVs in  $a$  to RVs in  $b$

## CONTRIBUTING

The contributing guide goes over the details necessary to contribute to SymPy. See also the full [Development Workflow](#) guide on the SymPy wiki.

### Content

## 6.1 Development Environment Setup

The first step to contributing to the code base is creating your development environment.

### 6.1.1 Git Setup

SymPy is available on [GitHub](#) and uses [Git](#) for source control. The workflow is such that code is pulled and pushed to and from the main repository. Install the respective version of Git for your operating system to start development.

---

**Note:** Refer to the installation instructions in the [Git installation instructions](#). Learn about the basic git commands in this [Git Handbook](#) or any other sources on the internet.

---

### 6.1.2 Get the SymPy Code

It is recommended practice to create a fork of the SymPy project for your development purposes. Create your own fork of the SymPy project (if you have not yet). Go to the SymPy GitHub repository:

```
https://github.com/sympy/sympy
```

You will now have a fork at `<https://github.com/<your-user-name>/sympy>`.

Then, on your machine browse to where you would like to store SymPy, and clone (download) the latest code from SymPy's original repository (about 77 MiB):

```
$ git clone https://github.com/<your-user-name>/sympy
```

You must [configure the remote repositories](#) for collaboration with the upstream project:

```
$ cd sympy
$ git remote add upstream https://github.com/sympy/sympy
```

After the configuration, your setup should be similar to this:

```
$ git remote -v
origin https://github.com/<your-user-name>/sympy (fetch)
origin https://github.com/<your-user-name>/sympy (push)
upstream https://github.com/sympy/sympy (fetch)
upstream https://github.com/sympy/sympy (push)
```

For further development, it is recommended to create a development branch.

```
$ git checkout -b dev-branch
```

The new branch can be of any name.

### 6.1.3 Virtual Environment Setup

You may want to take advantage of using virtual environments to isolate your development version of SymPy from any system wide installed versions, e.g. from `apt-get install python-sympy`.

We recommend using `conda` to create a virtual environment:

```
$ conda create -n sympy-dev python=3 mpmath flake8
```

You now have a environment that you can use for testing your development copy of SymPy. For example, clone your SymPy fork from Github:

```
$ git clone git@github.com:<your-github-username>/sympy.git
$ cd sympy
```

Now activate the environment:

```
$ conda activate sympy-dev
```

### 6.1.4 Run the Tests

There are several ways of running SymPy tests but the easiest is to use the `bin/test` script, consult 'the wiki details on running tests <<https://github.com/sympy/sympy/wiki/Running-tests>>'.

The script takes a number of options and arguments and then passes them to `sympy.test(*paths, **kwargs)`. Run `bin/test --help` for all supported arguments.

Run all tests by using the command:

```
$ bin/test
```

To run tests for a specific file, use:



```
$ bin/test test_basic
```

Where `test_basic` is from file `sympy/core/basic.py`.

To run tests for modules, use:

```
$ bin/test /core /utilities
```

This will run tests for the `core` and `utilities` modules.

Similarly, run quality tests with:

```
$ bin/test code_quality
```

## 6.2 Dependencies

This page lists the hard and optional dependencies of SymPy.

There are several packages that, when installed, can enable certain additional SymPy functionality. Most users and contributors will not need to install any of the packages mentioned below (except for the hard dependencies), unless they intend to use or contribute to the parts of SymPy that can use those packages.

Every dependency listed below can be installed with `conda` via [conda-forge](#), and most can also be installed with `pip`.

This page does not list packages which themselves depend on SymPy, only those packages that SymPy depends on. An incomplete list of packages that depend on SymPy can be found on the [main SymPy webpage](#), and a more complete list can be found on [GitHub](#) or [libraries.io](#).

### 6.2.1 Hard Dependencies

SymPy only has one hard dependency, which is required for it to work: `mpmath`.

- **mpmath:** [mpmath](#) is a pure Python package for arbitrary precision arithmetic. It is used under the hood whenever SymPy calculates the floating-point value of a function, e.g., when using [evalf](#) (page 1065).

SymPy cannot function without `mpmath` and will fail to import if it is not installed. If you get an error like

```
ImportError: SymPy now depends on mpmath as an external library. See
https://docs.sympy.org/latest/install.html#mpmath for more information.
```

this means that you did not install `mpmath` correctly. [This page](#) (page 2) explains how to install it.

Most methods of installing SymPy, such as the ones outlined in the [installation](#) (page 1) guide, will install `mpmath` automatically. You typically only need to install `mpmath` manually if you did not actually install SymPy, e.g., if you are developing directly on SymPy in the git repository.

## 6.2.2 Optional Dependencies

These dependencies are not required to use SymPy. The vast majority of SymPy functions do not require them, however, a few functions such as plotting and automatic wrapping of code generated functions require additional dependencies to function.

Additionally, as a contributor, when running the SymPy tests, some tests will be skipped if a dependency they require is not installed. The [GitHub Actions CI](#) which is run on every SymPy pull request will automatically install these dependencies in the “optional-dependencies” build, but you may wish to install them locally if you are working on a part of SymPy that uses them.

### Recommended Optional Dependencies

These dependencies are not required for SymPy to function, but it is recommended that all users install them if they can, as they will improve the general performance of SymPy.

- **gmpy2:** [gmpy2](#) is a Python wrapper for the [GMP multiple-precision library](#). It provides large integers that are faster than the built-in Python `int`. When `gmpy2` is installed, it is used automatically by certain core functions that operate on integers, such as the [polys](#) (page 2341). See [Reference docs for the Poly Domains](#) (page 2504) for more details. SymPy uses `gmpy2` automatically when it is installed. No further action is required to enable it.

The polys themselves are used by many parts of SymPy, such as the integration algorithms, simplification algorithms like `collect()` and `factor()`, the matrices, and some parts of the core. Thus, installing `gmpy2` can speed up many parts of SymPy. It is not a required dependency of SymPy because it makes use of a non-Python library (GMP), which is also non-BSD licensed. However, we recommended all users who are able to to install `gmpy2` to get a better SymPy experience.

### Interactive Use

SymPy is designed to be used both interactively and as a library. When used interactively, SymPy is able to interface with IPython and Jupyter notebooks.

- **IPython:** The [init\\_session\(\)](#) (page 2117) function and `isympy` command will automatically start IPython if it is installed. In addition to the usual benefits of using [IPython](#), this enables interactive plotting with `matplotlib`. Also some flags such as `auto_symbols` and `auto_int_to_Integer` will only work in IPython.

The IPython package is required to run some of the tests in `sympy/interactive`.

- **Jupyter Notebook and Qt Console:** SymPy expressions automatically print using MathJax in the [Jupyter Notebook](#) and with LaTeX [Qt Console](#) (if [LaTeX](#) (page 2987) is installed).

## Printing

The `preview()` (page 2181) function automatically converts SymPy expressions into images rendered with LaTeX. `preview()` can either save the image to a file or show it with a viewer.

- **LaTeX:** A  $\text{\LaTeX}$  distributions such as [TeXLive](#) or [MiKTeX](#) is required for `preview()` (page 2181) to function.

## Parsing

Several functions in the `sympy.parsing` (page 2122) submodule require external dependencies to function. Note that not all parsers require external modules at this time. The Python (`parse_expr()` (page 2122)), Mathematica (`parse_mathematica()` (page 2125)), and Maxima (`parse_maxima()` (page 2125)) parsers do not require any external dependencies.

- **antlr-python-runtime:** [Antlr](#) is used for the [LaTeX parser](#) (page 2129) and [Autolev](#) (page 1724) parsers. They both require the Antlr Python runtime to be installed. The package for this is called `antlr-python-runtime` with conda and `antlr4-python3-runtime` with pip). Also be aware that the version of the Antlr Python runtime must match the version that was used to compile the LaTeX and Autolev parsers (4.10).
- **Clang Python Bindings:** The C parser (`sympy.parsing.c.parse_c`) requires the Clang Python bindings. The package for this is called `python-clang` with conda and `clang` with pip.
- **lfortran:** The Fortran parser (in `sympy.parsing.fortran`) requires [LFortran](#).

## Logic

The `satisfiable()` (page 1184) function includes a pure Python implementation of the DPLL satisfiability algorithm. But it can optionally use faster C SAT solvers if they are installed. Note that `satisfiable()` is also used by `ask()` (page 191).

- **pycosat:** [Pycosat](#) is used automatically if it is installed. The use of pycosat can be forced by using `satisfiable(algorithm='pycosat')`.
- **pysat:** [Pysat](#) is a library which wraps many SAT solvers. It can also be used as a backend to `satisfiable()`. Presently, only [Minisat](#) is implemented, using `satisfiable(algorithm=minisat22')`.

## Plotting

The `sympy.plotting.plot` (page 2820) module makes heavy use of external plotting libraries to render plots. The primary plotting module that is supported is Matplotlib.

- **matplotlib:** Most plotting functionality requires the [Matplotlib](#) plotting library. Without Matplotlib installed, most plotting functions will either fail or give rudimentary [text plots](#) (page 2872).
- **pyglet:** SymPy has a submodule `sympy.plotting.pygletplot` (page 2869) that can be used to interface with the [pyglet](#) module to do 2D and 3D plotting.

## lambdify

`lambdify()` (page 2100) is a function that converts SymPy expressions into functions that can be evaluated numerically using various libraries as backends. `lambdify` is the primary vehicle by which users interface between SymPy and these libraries. It is the standard way to convert a symbolic SymPy expression into an evaluable numeric function.

In principle, `lambdify` can interface with any external library if the user passes in an appropriate namespace dictionary as the third argument, but by default, `lambdify` is aware of several popular numeric Python libraries. These libraries are enabled as backends in `lambdify` with build-in translations to convert SymPy expressions into the appropriate functions for those libraries.

- **NumPy:** By default, if it is installed, `lambdify` creates functions using `NumPy` (if `NumPy` is not installed, `lambdify` produces functions using the standard library `math` module, although this behavior is primarily provided for backwards compatibility).
- **SciPy:** If `SciPy` is installed, `lambdify` will use it automatically. `SciPy` is needed to `lambdify` certain [special functions](#) that are not included in `NumPy`.
- **CuPy:** `CuPy` is a library that provides a `NumPy` compatible interface for CUDA GPUs. `lambdify` can produce `CuPy` compatible functions using `lambdify(modules='cupy')`.
- **Jax:** `JAX` is a library that uses XLA to compile and run `NumPy` programs on GPUs and TPUs. `lambdify` can produce `JAX` compatible functions using `lambdify(modules='jax')`.
- **TensorFlow:** `TensorFlow` is a popular machine learning library. `lambdify` can produce `TensorFlow` compatible functions using `lambdify(modules='tensorflow')`.
- **NumExpr:** `NumExpr` is a fast numerical expression evaluator for `NumPy`. `lambdify` can produce `NumExpr` compatible functions using `lambdify(modules='numexpr')`.
- **mpmath:** `lambdify` can also produce `mpmath` compatible functions. Note that `mpmath` is already a [required dependency](#) (page 2985) of SymPy. This functionality is useful for converting a SymPy expression to a function for use with pure `mpmath`.

## Code Generation

SymPy can [generate code](#) (page 1108) for a large number of languages by converting SymPy expressions into valid code for those languages. It also has functionality for some languages to automatically compile and run the code.

Note that the dependencies below are **not** a list of supported languages that SymPy can generate code for. Rather it is a list of packages that SymPy can interface with in some way. For most languages that SymPy supports code generation, it simply generates a string representing the code for that language, so no dependency on that language is required to use the code generation functionality. A dependency is typically only required for features that automatically take the generated code and compile it to a function that can be used within Python. Note that `lambdify()` (page 2100) is a special case of this, but its dependencies are listed [above](#) (page 2988).

## Autowrap

- **NumPy:** NumPy and, optionally, its subpackage `f2py`, can be used to generate Python functions using the `autowrap()` (page 2041) or `ufuncify()` (page 2043) functions.
- **Cython:** Cython can be used as a backend for `autowrap()` (page 2041) or `ufuncify()` (page 2043). Cython is also used in some of the `sympy.codegen` tests to compile some examples.
- **Compilers:** `autowrap()` (page 2041), `ufuncify()` (page 2043), and related functions rely on a compiler to compile the generated code to a function. Most standard C, C++, and Fortran compilers are supported, including Clang/LLVM, GCC, and ifort.

## Code Printers

Most code printers generate Python strings, and therefore do not require the given library or language compiler as a dependency. However, a few code printers generate Python functions instead of strings:

- **Aesara:** The `sympy.printing.aesaracode` (page 2166) module contains functions to convert SymPy expressions into a functions using the Aesara (previously Theano) library. The Aesara code generation functions return Aesara graph objects.
- **llvmlite:** The `sympy.printing.llvmjitcode` module supports generating LLVM Jit from a SymPy expression. The functions make use of `llvmlite`, a Python wrapper around LLVM. The `llvm_callable()` function generates callable functions.
- **TensorFlow:** The `sympy.printing.tensorflow` module supports generating functions using the TensorFlow, a popular machine learning library. Unlike the above two examples, `tensorflow_code()` function **does** generate Python strings. However, `tensorflow` is imported if available in order to automatically detect the TensorFlow version. If it is not installed, the `tensorflow_code()` function assumes the latest supported version of TensorFlow.

## Testing-Only Dependencies

- **Wurlitzer:** Wurlitzer is a Python package that allows capturing output from C extensions. It is used by some of the tests in the `sympy.codegen` submodule. It is only used by the test suite. It is not used by any end-user functionality. If it is not installed, some tests will be skipped.
- **Cython:** Cython is also used in some of the `sympy.codegen` tests to compile some examples.
- **Compilers:** The various *compilers* (page 2989) mentioned above are used in some of the codegen and autowrap tests if they are installed.

## Statistics

The `sympy.stats.sample()` (page 2971) function uses an external library to produce samples from the given distribution. At least one of the following libraries is required to use the sampling functionality of `sympy.stats`.

- **SciPy**: `sample(library='scipy')` is the default. This uses `scipy.stats`.
- **NumPy**: `sample(library='numpy')` uses the `NumPy random` module.
- **pymc**: `sample(library='pymc')` uses `PyMC` to do sampling.

## Optional SymEngine Backend

- **python-symengine**: `SymEngine` is a fast symbolic manipulation library, written in C++. The SymEngine Python bindings may be used as an optional backend for SymPy core. To do this, first install the SymEngine Python bindings (with `pip install symengine` or `conda install -c conda-forge python-symengine`) and run SymPy with the `USE_SYMENGINE=1` environment variable.

Presently, the SymEngine backend is only used by the `sympy.physics.mechanics` (page 1675) and `sympy.liealgebras` (page 2326) modules, although you can also interface with SymPy's SymEngine backend directly by importing things from `sympy.core.backend`:

```
>>> from sympy.core.backend import Symbol
>>> # This will create a SymEngine Symbol object if the USE_SYMENGINE
>>> # environment variable is configured. Otherwise it will be an
↳ ordinary
>>> # SymPy Symbol object.
>>> x = Symbol('x')
```

SymEngine backend support is still experimental, so certain SymPy functions may not work correctly when it is enabled.

## Experimental Rubi Integrator

- **MatchPy**: `MatchPy` is a library for doing pattern matching. It is used in the experimental `sympy.integrals.rubi` module, but presently, it is not used anywhere else in SymPy. SymPy and MatchPy are able to interface with each other.

## Sage

`Sage` is an open source mathematics software that incorporates a large number of open source mathematics libraries. SymPy is one of the libraries used by Sage.

Most of the code that interfaces between SymPy and Sage is in Sage itself, but a few `_sage_` methods in SymPy that do some very basic setting up of the Sage/SymPy wrappers. These methods should typically only be called by Sage itself.

### 6.2.3 Development Dependencies

Typical development on SymPy does not require any additional dependencies beyond Python and mpmath.

#### Getting the Source Code

- **git:** The [SymPy source code](#) uses the [git](#) version control system. See the [installation guide](#) (page 1) and [development workflow](#) for instructions on how to get the development version of SymPy from git.

#### Running the Tests

The base SymPy tests do not require any additional dependencies, however most of the above dependencies may be required for some tests to run. Tests that depend on optional dependencies should be skipped when they are not installed, either by using the `sympy.testing.pytest.skip()` function or by setting `skip = True` to skip the entire test file. Optional modules in tests and SymPy library code should be imported with `import_module()`.

- **pytest:** [Pytest](#) is not a required dependency for the SymPy test suite. SymPy has its own test runner, which can be accessed via the `bin/test` script in the SymPy source directory or the [test\(\)](#) (page 2035) function.

However, if you prefer to use `pytest`, you can use it to run the tests instead of the SymPy test runner. Tests in SymPy should use the wrappers in [sympy.testing.pytest](#) (page 2027) instead of using `pytest` functions directly.

- **Cloudpickle:** The [cloudpickle](#) package can be used to more effectively pickle SymPy objects than the built-in Python [pickle](#). Some tests in `sympy.utilities.tests.test_pickling.py` depend on `cloudpickle` to run. It is not otherwise required for any SymPy function.

#### Building the Documentation

Building the documentation requires several additional dependencies. [This page](#) (page 2992) outlines these dependencies and how to install them. It is only necessary to install these dependencies if you are contributing documentation to SymPy and want to check that the HTML or PDF documentation renders correctly. If you only want to view the documentation for the development version of SymPy, development builds of the docs are hosted online at <https://docs.sympy.org/dev/index.html>.

#### Running the Benchmarks

The benchmarks for SymPy are hosted at [https://github.com/sympy/sympy\\_benchmarks](https://github.com/sympy/sympy_benchmarks). The [README](#) in that repository explains how to run the benchmarks.

Note that the benchmarks are also run automatically on the [GitHub Actions CI](#), so it is generally not necessary to run them yourself as a contributor unless you want to reproduce the benchmarks results on your computer or add a new benchmark to the suite.

- **asv:** [Airspeed Velocity](#) is the package used for running the benchmarks. Note that the package name that you install is called `asv`.

## 6.3 Build the Documentation

Start by installing the required dependencies for the documentation.

### 6.3.1 Required dependencies

You can either install the dependencies locally on your machine, or you can build a Docker image containing them.

#### Docker

If you have [Docker](#), then instead of following the OS-specific installation instructions below, you may choose to build a Docker image:

```
cd doc
docker build -f Dockerfile.html doc -t sympy_html doc .
```

If you choose this option, you can now skip down to the “Build the Docs” section below.

#### Debian/Ubuntu

For Debian/Ubuntu:

```
apt-get install python3-sphinx texlive-latex-recommended dvipng librsvg2-bin
→imagemagick docbook2x graphviz
python -m pip install -r doc/requirements.txt
```

If you get mpmath error, install python-mpmath package:

```
apt-get install python-mpmath
```

If you get matplotlib error, install python-matplotlib package:

```
apt-get install python-matplotlib
```

#### Fedora

For Fedora (and maybe other RPM-based distributions), install the prerequisites:

```
dnf install python3-sphinx librsvg2 ImageMagick docbook2X texlive-dvipng-bin
texlive-scheme-medium librsvg2-tools
python -m pip install -r doc/requirements.txt
```

If you get mpmath error, install python3-mpmath package:

```
dnf install python3-mpmath
```

If you get matplotlib error, install python3-matplotlib package:



```
dnf install python3-matplotlib
```

## Mac

For Mac, first install homebrew: <https://brew.sh/>

Then install these packages with homebrew:

```
brew install imagemagick graphviz docbook librsvg
```

Install the docs dependencies with either pip or conda:

```
python -m pip install -r requirements.txt
```

Or:

```
conda install -c conda-forge --file requirements.txt
```

Making your Sphinx build successful on the Windows system is tricky because some dependencies like dvipng or docbook2x are not available.

## Windows 10

For Windows 10, however, the Windows Subsystem for Linux can be a possible workaround solution, and you can install Ubuntu shell on your Windows system after following the tutorial below:

<https://github.com/MicrosoftDocs/WSL/blob/live/WSL/install-win10.md>

In your command prompt, run `ubuntu` to transfer to Linux terminal, and follow the Debian/Ubuntu tutorial above to install the dependencies, and then you can run `make html` to build. (Note that you also have to install `make` via `apt-get install make`.)

If you want to change the directory in your prompt to your working folder of SymPy in the Windows file system, you can prepend `cd /mnt/` to your file path in Windows, and run in your shell to navigate to the folder. (Also note that Linux uses `/` instead of `\` for file paths.)

This method provides better compatibility than Cygwin or MSYS2 and more convenience than a virtual machine if you partially need a Linux environment for your workflow, however this method is only viable for Windows 10 64-bit users.

or

Follow [instruction](#) to install Chocolatey

Install `make` and other dependencies:

```
choco install make graphviz rsvg-convert imagemagick
```

Install python dependencies:

```
pip install -r doc/requirements.txt
```

## 6.3.2 Build the Docs

### Docker

If you chose to build using Docker, and followed the instructions above to build the `sympy_html doc` image, then you can build the docs with:

```
docker run --rm -v /absolute/path/to/sympy:/sympy sympy_html doc
```

(Be sure to substitute the actual absolute filesystem path to sympy!) This command can be run from any directory.

### Local Installation

If you chose to follow OS-specific instructions above and installed the required dependencies locally, the documentation can be built by running the makefile in the `doc` subdirectory:

```
cd doc
make html
```

### SymPy Logos

SymPy has a collection of official logos, which can be generated from `sympy.svg` in your local copy of SymPy by:

```
$ cd doc
$ make logo # will be stored in the _build/logo subdirectory
```

The license of all the logos is the same as SymPy: BSD. See the [LICENSE file](#) for more information.

## 6.3.3 View the Docs

Once you have built the docs, the generated files will be found under `doc/_build/html`. To view them in your preferred web browser, use the drop down menu and select “open file”, navigate into the `sympy/doc/_build/html` folder, and open the `index.html` file.

## 6.3.4 Auto-Rebuild with the Live Server

The instructions given above told you how to build the docs once, and load them in the browser. After you make changes to the document sources, you’ll have to manually repeat the build step, and reload the pages in the browser.

There is an alternative approach that sets up a live server, which will monitor the docs directory, automatically rebuild when changes are detected, and automatically reload the page you are viewing in the browser.

If you want to use this option, the procedure again depends on whether you are using Docker, or a local installation.

## Docker

To start the live server with Docker, you can use:

```
docker run --rm -it \
 -v /absolute/path/to/sympy:/sympy \
 -p 8000:80 \
 sympy_html doc live
```

and then navigate your browser to `localhost:8000`. You can use a different port by changing the 8000 in the command. Again, be sure to substitute the actual absolute filesystem path to sympy.

When finished, you can stop the server with `ctrl-c` in the terminal.

Alternatively, you may run the server in detached mode, using:

```
docker run --rm -d --name=sympy-livehtml \
 -v /absolute/path/to/sympy:/sympy \
 -p 8000:80 \
 sympy_html doc live
```

and then stop it with:

```
docker stop sympy-livehtml
```

## Local Installation

If you installed the build dependencies locally, then simply use:

```
cd doc
make livehtml
```

to start the server. Your web browser should then automatically open a new tab, showing the index page of the SymPy docs.

When you are finished, you can use `ctrl-c` in the terminal to stop the server.

### 6.3.5 PDF Documentation

---

**Note:** It is not necessary for the majority of contributors to build the PDF documentation. The PDF documentation will be built automatically on GitHub Actions on pull requests. PDF documentation for each release is included on the [GitHub releases page](#).

If the PDF documentation build fails on GitHub Actions, 99% of the time this is due to bad LaTeX math formatting. Double check that any math you have added is formatted correctly, and make sure you use ``double backticks`` for code (single backticks will render as math, not code). See the resources in the [style guide](#) (page 3013) for tips on formatting LaTeX math.

---

Building the PDF documentation requires a few extra dependencies. First you will need to have a TeXLive installation that includes XeLaTeX and latexmk. You will also need to have Chrome or Chromium installed, as it is used to convert some SVG files for the PDF.

On Ubuntu, you can install these with:

```
apt-get install chromium-browser texlive texlive-xetex texlive-fonts-
→recommended texlive-latex-extra latexmk lmodern
```

On Mac, you can use:

```
brew install texlive
brew install --cask chromium
brew tap homebrew/cask-fonts
brew install font-dejavu
```

On Windows 10, you can use:

```
choco install chromium strawberryperl miktex dejavufonts
```

If DejaVu fonts are not installed in C:\Windows\Fonts, then open ~\AppData\Local\Microsoft\Windows\Fonts, select all DejaVu fonts, right-click and click Install for all users.

To build the pdf docs run:

```
cd doc
make pdf
```

The resulting PDF will be in:

```
_build/latex/sympy-<version>.pdf
```

where <version> is the SymPy version (e.g., sympy-1.10.dev.pdf).

## 6.4 Debugging

To start sympy in debug mode set the SYMPY\_DEBUG variable. For instance in a unix-like system you would do

```
$ SYMPY_DEBUG=True bin/isympy
```

or in Windows

```
> set SYMPY_DEBUG=True > python bin/isympy
```

Now just use for example the `limit()` function. You will get a nice printed tree, which is very useful for debugging.

## 6.5 SymPy Docstrings Style Guide

### 6.5.1 General Guidelines

To contribute to SymPy's docstrings, please read these guidelines in full.

A documentation string (docstring) is a string literal that occurs as the first statement in a module, function, class, or method definition. Such a docstring becomes the `__doc__` special attribute of that object.

#### Example

Here is a basic docstring:

```
def fdiff(self, argindex=1):
 """
 Returns the first derivative of a Heaviside Function.

 Examples
 =====

 >>> from sympy import Heaviside, diff
 >>> from sympy.abc import x

 >>> Heaviside(x).fdiff()
 DiracDelta(x)

 >>> Heaviside(x**2 - 1).fdiff()
 DiracDelta(x**2 - 1)

 >>> diff(Heaviside(x)).fdiff()
 DiracDelta(x, 1)

 """
```

Every public function, class, method, and module should have a docstring that describes what it does. Documentation that is specific to a function or class in the module should be in the docstring of that function or class. The module level docstring should discuss the purpose and scope of the module, and give a high-level example of how to use the functions or classes in the module. A module docstring is the docstring at the very top of the file, for example, the docstring for [solvers.ode](#).

A public function is one that is intended to be used by the end- user, or the public. Documentation is important for public functions because they will be seen and used by many people.

A private function, on the other hand, is one that is only intended to be used in the code in SymPy itself. Although it is less important to document private functions, it also helps to have docstrings on private functions to help other SymPy developers understand how to use the function.

It may not always be clear what is a public function and what is a private function. If a function begins with an underscore, it is private, and if a function is included in `__init__.py` it is public, but the converse is not always true, so sometimes you have to decide based on context. In general, if you are unsure, having documentation on a function is better than not having documentation, regardless if it is public or private.

Docstrings should contain information aimed at users of the function. Comments specific to the code or other notes that would only distract users should go in comments in the code, not in docstrings.

Every docstring should have examples that show how the function works. Examples are the most important part of a docstring. A single example showing input and output to a function can be more helpful than a paragraph of descriptive text.

Remember that the primary consumers of docstrings are other human beings, not machines, so it is important to describe what the function does in plain English. Likewise, examples of how to use the function should be designed for human readers, not just for the doctest machinery.

Keep in mind that while Sphinx is the primary way users consume docstrings, and therefore the first platform to keep in mind while writing docstrings (especially for public functions), it is not the only way users consume docstrings. You can also view docstrings using `help()` or `?` in IPython. When using `help()`, for instance, it will show you all of the docstrings on private methods. Additionally, anyone reading the source code directly will see every docstring.

All public functions, classes, and methods and their corresponding docstrings should be imported into the Sphinx docs, instructions on which can be found at the end of this guide.

## 6.5.2 Formatting

Docstrings are written in [reStructuredText](#) format extended by [Sphinx](#). Here is a concise guide to [Quick reStructuredText](#). More in-depth information about using reStructuredText can be found in the [Sphinx Documentation](#).

In order for Sphinx to render docstrings nicely in the HTML documentation, some formatting guidelines should be followed when writing docstrings:

- Always use `"""triple double quotes"""` around docstrings. Use `r"""raw triple double quotes"""` if you use any backslashes in your docstrings.
- Include a blank line before the docstring's closing quotes.
- Lines should not be longer than 80 characters.
- Always write class-level docstrings under the class definition line, as that is better to read in the source code.
- The various methods on the class can be mentioned in the docstring or examples if they are important, but details on them should go in the docstring for the method itself.
- Be aware that `::` creates code blocks, which are rarely used in the docstrings. Any code example with example Python should be put in a doctest. Always check that the final version as rendered by Sphinx looks correct in the HTML.
- In order to make section underlining work nicely in docstrings, [numpydoc Sphinx extension](#) is used.
- Always double check that you have formatted your docstring correctly:

1. Make sure that your docstring is imported into Sphinx.
2. Build the Sphinx docs (`cd doc; make html`).
3. Make sure that Sphinx doesn't output any errors.
4. Open the page in `_build/html` and make sure that it is formatted correctly.

### 6.5.3 Sections

In SymPy’s docstrings, it is preferred that function, class, and method docstrings consist of the following sections in this order:

1. Single-Sentence Summary
2. Explanation
3. Examples
4. Parameters
5. See Also
6. References

The Single-Sentence Summary and Examples sections are **required** for every docstring. Docstrings will not pass review if these sections are not included.

Do not change the names of these supported sections, for example, the heading “Examples” as a plural should be used even if there is only one example.

SymPy will continue to support all of the section headings listed in the [NumPy Docstring Guide](#).

Headings should be underlined with the same length in equals signs.

If a section is not required and that information for the function in question is unnecessary, do not use it. Unnecessary sections and cluttered docstrings can make a function harder to understand. Aim for the minimal amount of information required to understand the function.

#### 1. Single-Sentence Summary

This section is **required** for every docstring. A docstring will not pass review if it is not included. No heading is necessary for this section.

This section consists of a one-line sentence ending in a period that describes the function, class, or method’s effect.

Deprecation warnings should go directly after the Single-Sentence Summary, so as to notify users right away. Deprecation warnings should be written as a deprecated in the Sphinx directive:

```
.. deprecated:: 1.1

 The ``simplify_this`` function is deprecated. Use :func:`simplify`
 instead. See its documentation for more information.
```

See [Documenting a deprecation](#) (page 3022) for more details.

## 2. Explanation Section

This section is encouraged. If you choose to include an Explanation section in your docstring, it should be labeled with the heading “Explanation” underlined with the same length in equals signs.

```
Explanation
=====
```

This section consists of a more elaborate description of what the function, class, or method does when the concise Single-Sentence Summary will not suffice. This section should be used to clarify functionality in several sentences or paragraphs.

## 3. Examples Section

This section is **required** for every docstring. A docstring will not pass review if it is not included. It should be labeled with the heading “Examples” (even if there is only one example) underlined with the same length in equals signs.

```
Examples
=====
```

This section consists of examples that show how the function works, called doctests. Doctests should be complicated enough that they fully demonstrate the API and functionality of the function, but simple enough that a user can understand them without too much thought. The perfect doctest tells the user exactly what they need to know about the function without reading any other part of the docstring.

There should always be a blank line before the doctest. When multiple examples are provided, they should be separated by blank lines. Comments explaining the examples should have blank lines both above and below them.

Do not think of doctests as tests. Think of them as examples that happen to be tested. They should demonstrate the API of the function to the user (i.e., what the input parameters look like, what the output looks like, and what it does). If you only want to test something, add a test to the relevant `test_*.py` file.

You can use the `./bin/coverage_doctest.py` script to test the doctest coverage of a file or module. Run the doctests with `./bin/doctest`.

You should only skip the testing of an example if it is impossible to test it. If necessary, testing of an example can be skipped by adding a special comment.

### Example

```
>>> import random
>>> random.random()
0.6868680200532414
```

If an example is longer than 80 characters, it should be line wrapped. Examples should be line wrapped so that they are still valid Python code, using `...` continuation as in a Python prompt. For example, from the ODE module documentation:

### Example



```
>>> from sympy import Function, dsolve, cos, sin
>>> from sympy.abc import x
>>> f = Function('f')
>>> dsolve(cos(f(x)) - (x*sin(f(x)) - f(x)**2)*f(x).diff(x),
... f(x), hint='1st_exact')
Eq(x*cos(f(x)) + f(x)**3/3, C1)
```

Here `dsolve(cos(f(x)) - (x*sin(f(x)) - f(x)**2)*f(x).diff(x), f(x), hint='1st_exact')` is too long, so we line break it after a comma so that it is readable, and put `...` on the continuation lines. If this is not done correctly, the doctests will fail.

The output of a command can also be line wrapped. No `...` should be used in this case. The doctester automatically accepts output that is line wrapped.

### Example

```
>>> list(range(30))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29]
```

In a doctest, write imports like `sympy import ...` instead of `import sympy` or `from sympy import *`. To define symbols, use `from sympy.abc import x`, unless the name is not in `sympy.abc` (for instance, if it has assumptions), in which case use symbols like `x`, `y = symbols('x y')`.

In general, you should run `./bin/doctest` to make sure your examples run correctly, and fix them if they do not.

## 4. Parameters Section

This section is encouraged. If you choose to include a Parameters section in your docstring, it should be labeled with the heading “Parameters” underlined with the same length in equals signs.

```
Parameters
=====
```

If you have parameters listed in parentheses after a function, class, or method name, you must include a parameters section.

This section consists of descriptions of the function arguments, keywords, and their respective types.

Enclose variables in double backticks. The colon must be preceded by a space, or omitted if the type is absent. For the parameter types, be as precise as possible. If it is not necessary to specify a keyword argument, use `optional`. Optional keyword parameters have default values, which are displayed as part of the function signature. They can also be detailed in the description.

When a parameter can only assume one of a fixed set of values, those values can be listed in braces, with the default appearing first. When two or more input parameters have exactly the same type, shape, and description, they can be combined.

If the Parameters section is not formatted correctly, the documentation build will render incorrectly.

If you wish to include a Returns section, write it as its own section with its own heading.

### Example

Here is an example of a correctly formatted Parameters section:

```
def opt_cse(exprs, order='canonical'):
 """
 Find optimization opportunities in Adds, Muls, Pows and negative
 coefficient Muls.

 Parameters
 =====

 exprs : list of sympy expressions
 The expressions to optimize.
 order : string, 'none' or 'canonical'
 The order by which Mul and Add arguments are processed. For large
 expressions where speed is a concern, use the setting order='none'.

 """
```

## 5. See Also Section

This section is encouraged. If you choose to include a See Also section in your docstring, it should be labeled with the heading “See Also” underlined with the same length in equals signs.

```
See Also
=====
```

This section consists of a listing of related functions, classes, and methods. The related items can be described with a concise fragment (not a full sentence) if desired, but this is not required. If the description spans more than one line, subsequent lines must be indented.

The See Also section should only be used to reference other SymPy objects. Anything that is a link should be embedded as a hyperlink in the text of the docstring instead; see the References section for details.

Do not reference classes with `class:Classname`, `class:`Classname``, or `:class:`Classname``, but rather only by their class name.

### Examples

Here is a correctly formatted See Also section with concise descriptions:

```
class erf(Function):
 r"""
 The Gauss error function.

 See Also
 =====

 erfc: Complementary error function.
 erfi: Imaginary error function.
```

(continues on next page)

(continued from previous page)

```

erf2: Two-argument error function.
erfinv: Inverse error function.
erfcinv: Inverse Complementary error function.
erf2inv: Inverse two-argument error function.

"""

```

Here is a correctly formatted See Also section with just a list of names:

```

class besselj(BesselBase):
 r"""
 Bessel function of the first kind.

 See Also
 =====

 bessely, besseli, bess elk

 """

```

## 6. References Section

This section is encouraged. If you choose to include a References section in your docstring, it should be labeled with the heading “References” underlined with the same length in equals signs.

```

References
=====

```

This section consists of a list of references cited anywhere in the previous sections. Any reference to other SymPy objects should go in the See Also section instead.

The References section should include online resources, paper citations, and/or any other printed resource giving general information about the function. References are meant to augment the docstring, but should not be required to understand it. References are numbered, starting from one, in the order in which they are cited.

For online resources, only link to freely accessible and stable online resources such as Wikipedia, Wolfram MathWorld, and the NIST Digital Library of Mathematical Functions (DLMF), which are unlikely to suffer from hyperlink rot.

References for papers should include, in this order: reference citation, author name, title of work, journal or publication, year published, page number.

If there is a DOI (Digital Object Identifier), include it in the citation and make sure it is a clickable hyperlink.

### Examples

Here is a References section that cites a printed resource:

```

References
=====

```

(continues on next page)

(continued from previous page)

```
.. [1] [Kozen89] D. Kozen, S. Landau, Polynomial Decomposition Algorithms,
 Journal of Symbolic Computation 7 (1989), pp. 445-456
```

Here is a References section that cites printed and online resources:

#### References

```
=====

.. [1] Abramowitz, Milton; Stegun, Irene A., "Chapter 9," Handbook of
 Mathematical Functions with Formulas, Graphs, and Mathematical
 Tables, eds. (1965)
.. [2] Luke, Y. L., The Special Functions and Their Approximations,
 Volume 1, (1969)
.. [3] https://en.wikipedia.org/wiki/Bessel_function
.. [4] http://functions.wolfram.com/Bessel-TypeFunctions/BesselJ/
```

### 6.5.4 Sample Docstring

Here is an example of a correctly formatted docstring:

```
class gamma(Function):
 r"""
 The gamma function

 .. math::
 \Gamma(x) := \int^{\infty}_{0} t^{x-1} e^{-t} \mathrm{d}t.

 Explanation
 =====

 The ``gamma`` function implements the function which passes through the
 values of the factorial function (i.e., $\Gamma(n) = (n - 1)!$), when n
 is an integer. More generally, $\Gamma(z)$ is defined in the whole
 complex plane except at the negative integers where there are simple
 poles.

 Examples
 =====

 >>> from sympy import S, I, pi, oo, gamma
 >>> from sympy.abc import x

 Several special values are known:

 >>> gamma(1)
 1
 >>> gamma(4)
 6
 >>> gamma(S(3)/2)
 sqrt(pi)/2
```

(continues on next page)

(continued from previous page)

The ``gamma`` function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(gamma(x))
gamma(conjugate(x))
```

Differentiation with respect to  $x$  is supported:

```
>>> from sympy import diff
>>> diff(gamma(x), x)
gamma(x)*polygamma(0, x)
```

Series expansion is also supported:

```
>>> from sympy import series
>>> series(gamma(x), x, 0, 3)
1/x - EulerGamma + x*(EulerGamma**2/2 + pi**2/12) + x**2*(-
→ EulerGamma*pi**2/12 +
polygamma(2, 1)/6 - EulerGamma**3/6) + O(x**3)
```

We can numerically evaluate the ``gamma`` function to arbitrary precision on the whole complex plane:

```
>>> gamma(pi).evalf(40)
2.288037795340032417959588909060233922890
>>> gamma(1+I).evalf(20)
0.49801566811835604271 - 0.15494982830181068512*I
```

See Also

=====

lowergamma: Lower incomplete gamma function.  
 uppgamma: Upper incomplete gamma function.  
 polygamma: Polygamma function.  
 loggamma: Log Gamma function.  
 digamma: Digamma function.  
 trigamma: Trigamma function.  
 beta: Euler Beta function.

References

=====

```
.. [1] https://en.wikipedia.org/wiki/Gamma_function
.. [2] http://dlmf.nist.gov/5
.. [3] http://mathworld.wolfram.com/GammaFunction.html
.. [4] http://functions.wolfram.com/GammaBetaErf/Gamma/
```

"""

### 6.5.5 Docstrings for Classes that are Mathematical Functions

SymPy is unusual in that it also has classes that are mathematical functions. The docstrings for classes that are mathematical functions should include details specific to this kind of class, as noted below:

- The Explanation section should include a mathematical definition of the function. This should use LaTeX math. Use `$$` for *inline math* (page 3013) and `.. math::` for display math, which should be used for the main definition. The variable names in the formulas should match the names of the parameters, and the LaTeX formatting should match the LaTeX pretty printing used by SymPy. As relevant, the mathematical definitions should mention their domain of definition, especially if the domain is different from the complex numbers.
- If there are multiple conventions in the literature for a function, make sure to clearly specify which convention SymPy uses.
- The Explanation section may also include some important mathematical facts about the function. These can alternately be demonstrated in the Examples section. Mathematical discussions should not be too long, as users can check the references for more details.
- The docstring does not need to discuss every implementation detail such as at which operations are defined on the function or at which points it evaluates in the “eval” method. Important or illuminating instances of these can be shown in the Examples section.
- The docstring should go on the class level (right under the line that has “class”). The “eval” method should not have a docstring.
- Private methods on the class, that is, any method that starts with an underscore, do not need to be documented. They can still be documented if you like, but note that these docstrings are not pulled into the Sphinx documentation, so they will only be seen by developers who are reading the code, so if there is anything very important that you want to mention here, it should go in the class-level docstring as well.

### 6.5.6 Best Practices for Writing Docstrings

When writing docstrings, please follow all of the same formatting, style, and tone preferences as when writing narrative documentation. For guidelines, see [Best Practices for Writing Documentation](#) (page 3013), Formatting, Style, and Tone.

### 6.5.7 Importing Docstrings into the Sphinx Documentation

Here are excerpts from the `doc/src/modules/geometry` directory that imports the relevant docstrings from geometry module into documentation:

```
Utils
=====

.. module:: sympy.geometry.util

.. autofunction:: intersection

.. autofunction:: convex_hull
```

(continues on next page)

(continued from previous page)

```
.. autofunction:: are_similar

Points
=====

.. module:: sympy.geometry.point

.. autoclass:: Point
 :members:

Lines
=====

.. module:: sympy.geometry.line

.. autoclass:: LinearEntity
 :members:

.. autoclass:: Line
 :members:

.. autoclass:: Ray
 :members:

.. autoclass:: Segment
 :members:

Curves
=====

.. module:: sympy.geometry.curve

.. autoclass:: Curve
 :members:

Ellipses
=====

.. module:: sympy.geometry.ellipse

.. autoclass:: Ellipse
 :members:

.. autoclass:: Circle
 :members:

Polygons
=====

.. module:: sympy.geometry.polygon

.. autoclass:: Polygon
```

(continues on next page)

(continued from previous page)

```
:members:

.. autoclass:: RegularPolygon
 :members:

.. autoclass:: Triangle
 :members:
```

First namespace is set to particular submodule (file) with `.. module::` directive, then docstrings are imported with `.. autoclass::` or `.. autofunction::` relative to that submodule (file). Other methods are either cumbersome to use (using full paths for all objects) or break something (importing relative to main module using `.. module:: sympy.geometry` breaks viewcode Sphinx extension). All files in `doc/src/modules/` should use this format.

### 6.5.8 Cross-Referencing

Any text that references another SymPy function should be formatted so that a cross-reference link to that function’s documentation is created automatically. This is done using the RST cross-reference syntax. There are two different kinds of objects that have conventions here:

1. Objects that are included in `from sympy import *`, for example, `sympy.acos`.

For these, use `:obj:`~.acos()``. The `~` makes it so that the text in the rendered HTML only shows `acos` instead of the fully qualified name `sympy.functions.elementary.trigonometric.acos`. (This will encourage importing names from the global `sympy` namespace instead of a specific submodule.) The `.` makes it so that the function name is found automatically. (If Sphinx gives a warning that there are multiple names found, replace the `.` with the full name. For example, `:obj:`~sympy.solvers.solvers.solve()``.) Adding a trailing pair of parentheses is a convention for indicating the name is a function, method, or class.

You may also use a more specific type indicator instead of `obj` (see <https://www.sphinx-doc.org/en/master/usage/restructuredtext/domains.html#cross-referencing-python-objects>).

However, `obj` will always work, and sometimes SymPy names are not the type you might expect them to be. For example, mathematical function objects such as `sin` are not actually a Python function, rather they are a Python class, therefore `:func:`~.sin`` will not work.

2. Objects that are not included in `from sympy import *`, for example, `sympy.physics.vector.dynamicsymbols`.

This can be public API objects from submodules that are not included in the main `sympy/__init__.py`, such as the `physics` submodule, or private API objects that are not necessarily intended to be used by end-users (but should still be documented). In this case, you must show the fully qualified name, so do not use the `~.` syntax. For example, `:obj:`~sympy.physics.vector.dynamicsymbols()``.

You may also write custom text that links to the documentation for something using the following syntax `:obj:`custom text<object>``. For example, `:obj:`the sine function <.sin>`` produces the text “the sine function” that links to the documentation for `sin`. Note that the `~` character should not be used here.

Note that references in the *See Also* (page 3002) section of the docstrings do not require the `:obj:` syntax.



If the resulting cross reference is written incorrectly, Sphinx will error when building the docs with an error like:

WARNING: py:obj reference target **not** found: expand

Here are some troubleshooting tips to fix the errors:

- Make sure you have used the correct syntax, as described above.
- Make sure you spelled the function name correctly.
- Check if the function you are trying to cross-reference is actually included in the Sphinx documentation. If it is not, Sphinx will not be able to create a reference for it. In that case, you should add it to the appropriate RST file as described in the [Docstring Guidelines](#) (page 2997).
- If the function or object is not included in `from sympy import *`, you will need to use the fully qualified name, like `sympy.submodule.submodule.function` instead of just `function`.
- A fully qualified name must include the full submodule for a function all the way down to the file. For example, `sympy.physics.vector.ReferenceFrame` will not work (even though you can access it that way in code). It has to be `sympy.physics.vector.frame.ReferenceFrame`.
- If the thing you are referring to does not actually have somewhere to link to, do not use the `:obj:` syntax. Instead, mark it as code using double backticks. Examples of things that cannot be linked to are Python built in functions like `int` or `NotImplementedError`, functions from other modules outside of SymPy like `matplotlib.plot`, and variable or parameter names that are specific to the text at hand. In general, if the object cannot be accessed as `sympy.something.something.object`, it cannot be cross- referenced and you should not use the `:obj:` syntax.
- If you are using one of the [type specific](#) identifiers like `:func:`, be sure that the type for it is correct. `:func:` only refers to Python functions. For classes, you need to use `:class:`, and for methods on a class you need to use `:method:`. In general, it is recommended to use `:obj:`, as this will work for any type of object.
- If you cannot get the cross-referencing syntax to work, go ahead and submit the pull request as is and ask the reviewers for help.

You may also see errors like:

WARNING: more than one target found **for** cross-reference `'subs()'`:  
`sympy.core.basic.Basic.subs`, `sympy.matrices.common.MatrixCommon.subs`,  
`sympy.physics.vector.vector.Vector.subs`,  
`sympy.physics.vector.dyadic.Dyadic.subs`

for instance, from using `:obj:`~.subs``. This means that the `.` is not sufficient to find the function, because there are multiple names in SymPy named `subs`. In this case, you need to use the fully qualified name. You can still use `~` to make it shortened in the final text, like `:obj:`~~sympy.core.basic.Basic.subs``.

The line numbers for warnings in Python files are relative to the top of the docstring, not the file itself. The line numbers are often not completely correct, so you will generally have to search the docstring for the part that the warning is referring to. This is due to a bug in Sphinx.

## 6.6 Documentation Style Guide

### 6.6.1 General Guidelines

Documentation is one of the most highly valued aspects of an open source project. Documentation teaches users and contributors how to use a project, how to contribute, and the standards of conduct within an open source community. But according to GitHub's [Open Source Survey](#), incomplete or confusing documentation is the most commonly encountered problem in open source. This style guide aims to change that.

The purpose of this style guide is to provide the SymPy community with a set of style and formatting guidelines that can be utilized and followed when writing SymPy documentation. Adhering to the guidelines offered in this style guide will bring greater consistency and clarity to SymPy's documentation, supporting its mission to become a full-featured, open source computer algebra system (CAS).

The SymPy documentation found at [docs.sympy.org](https://docs.sympy.org) is generated from docstrings in the source code and dedicated narrative documentation files in the `doc/src` directory. Both are written in `reStructuredText` format extended by `Sphinx`.

The documentation contained in the `doc/src` directory and the docstrings embedded in the Python source code are processed by Sphinx and various Sphinx extensions. This means that the documentation source format is specified by the documentation processing tools. The SymPy Documentation Style Guide provides both the essential elements for writing SymPy documentation as well as any deviations in style we specify relative to these documentation processing tools. The following lists the processing tools:

- `reStructuredText`: Narrative documentation files and documentation strings embedded in Python code follow the `reStructuredText` format. Advanced features not described in this document can be found at <http://docutils.sourceforge.net/rst.html>.
- `Sphinx`: Sphinx includes additional default features for the `reStructuredText` specification that are described at: <http://www.sphinx-doc.org/>.
- Sphinx extensions included with Sphinx that we enable:
  - `sphinx.ext.autodoc`: Processes Python source code files for the associated documentation strings to automatically generate pages containing the Application Programming Interface (API). See section on calling autodoc directives in this document to get started. More information is at: <https://www.sphinx-doc.org/en/master/usage/extensions/autodoc.html>.
  - `sphinx.ext.graphviz`: Provides a directive for adding Graphviz graphics. See <https://www.sphinx-doc.org/en/master/usage/extensions/graphviz.html> for more info.
  - `sphinx.ext.mathjax`: Causes math written in LaTeX to display using MathJax in the HTML version of the documentation. More information is at: <https://www.sphinx-doc.org/en/master/usage/extensions/math.html#module-sphinx.ext.mathjax>. *No bearing on documentation source format.*
  - `sphinx.ext.linkcode`: Causes links to source code to direct to the related files on Github. More information is at: <https://www.sphinx-doc.org/en/master/usage/extensions/linkcode.html>. *No bearing on documentation source format.*
- Sphinx extensions that are not included with Sphinx that we enable:

- `numpydoc`: Processes docstrings written in the “numpydoc” format, see <https://numpydoc.readthedocs.io>. We recommend the subset of numpydoc formatting features in this document. (Note that we currently use an older modified fork of numpydoc, which is included in the SymPy source code.)
- `sphinx_math_dollar`: Allows math to be delimited with dollar signs instead of reStructuredText directives (e.g.,  $a^2$  instead of `:math:`a^2``). See <https://www.sympy.org/sphinx-math-dollar/> for more info.
- `matplotlib.sphinxext.plot_directive`: Provides directives for included matplotlib generated figures in reStructuredText. See [https://matplotlib.org/devel/plot\\_directive.html](https://matplotlib.org/devel/plot_directive.html) for more info.

Everything supported by the above processing tools is available for use in the SymPy documentation, but this style guide supersedes any recommendations made in the above documents. Note that we do not follow PEP 257 or the [www.python.org](http://www.python.org) documentation recommendations.

If you are contributing to SymPy for the first time, please read our [Introduction to Contributing](#) page as well as this guide.

## 6.6.2 Types of Documentation

There are four main locations where SymPy’s documentation can be found:

**SymPy Website** <https://sympy.org>

The SymPy website’s primary function is to advertise the software to users and developers. It also serves as an initial location to point viewers to other relevant resources on the web. The SymPy website has basic information on SymPy and how to obtain it, as well as examples to advertise it to users, but it does not have technical documentation. The source files are located in the SymPy [webpage directory](#). Appropriate items for the website are:

- General descriptions of what SymPy and the SymPy community are
- Explanations/demonstrations of major software features
- Listings of other major software that uses SymPy
- Getting started info for users (download and install instructions)
- Getting started info for developers
- Where users can get help and support on using SymPy
- News about SymPy

**SymPy Documentation** <https://docs.sympy.org>

This is the main place where users go to learn how to use SymPy. It contains a tutorial for SymPy as well as technical documentation for all of the modules. The source files are hosted in the main SymPy repository in the [doc directory](#) at and are built using the [Sphinx site generator](#) and uploaded to the [docs.sympy.org](https://docs.sympy.org) site automatically. There are two primary types of pages that are generated from different source files in the docs directory:

- **Narrative Pages**: reStructuredText files that correspond to manually written documentation pages not present in the Python source code. Examples are the [tutorial RST files](#). In general, if your documentation is not API documentation it belongs in a narrative page.

- API Documentation Pages: reStructuredText files that contain directives that generate the Application Programming Interface documentation. These are automatically generated from the SymPy Python source code.

### SymPy Source Code <https://github.com/sympy/sympy>

Most functions and classes have documentation written inside it in the form of a docstring, which explains the function and includes examples called doctests. The purpose of these docstrings are to explain the API of that class or function. The doctests examples are tested as part of the test suite, so that we know that they always produce the output that they say that they do. Here is an [example docstring](#). Most docstrings are also automatically included in the Sphinx documentation above, so that they appear on the SymPy Documentation website. Here is that [same docstring](#) (page 383) on the SymPy website. The docstrings are formatted in a specific way so that Sphinx can render them correctly for the docs website. The SymPy sources all contain sparse technical documentation in the form of source code comments, although this does not generally constitute anything substantial and is not displayed on the documentation website.

### SymPy Wiki <https://github.com/sympy/sympy/wiki>

The SymPy Wiki can be edited by anyone without review. It contains various types of documentation, including:

- High-level developer documentation (for example: <https://github.com/sympy/sympy/wiki/Args-Invariant>)
- Guides for new contributors (for example: <https://github.com/sympy/sympy/wiki/Introduction-to-contributing>)
- Development policies (for example: <https://github.com/sympy/sympy/wiki/Python-version-support-policy>)
- Release notes (for example: <https://github.com/sympy/sympy/wiki/Release-Notes-for-1.5>)
- Various pages that different contributors have added

## 6.6.3 Narrative Documentation Guidelines

Extensive documentation, or documentation that is not centered around an API reference, should be written as a narrative document in the Sphinx docs (located in the [doc/src directory](#)). The narrative documents do not reside in the Python source files, but as standalone restructured files in the doc/src directory. SymPy's narrative documentation is defined as the collective documents, tutorials, and guides that teach users how to use SymPy. Reference documentation should go in the docstrings and be pulled into the RST with autodoc. The RST itself should only have narrative style documentation that is not a reference for a single specific function.

### 6.6.4 Documentation using Markdown

Narrative documentation can be written using either Restructured Text (`.rst`) or Markdown (`.md`). Markdown documentation uses [MyST](#). See [this guide](#) for more information on how to write documents in Markdown. Markdown is only supported for narrative documentation. Docstrings should continue to use RST syntax. Any part of this style guide that is not specific to RST syntax should still apply to Markdown documents.

### 6.6.5 Best Practices for Writing Documentation

Please follow these formatting, style, and tone preferences when writing documentation.

#### Formatting Preferences

In order for math and code to render correctly on the SymPy website, please follow these formatting guidelines.

#### Math

Text that is surrounded by dollar signs `$ _ $` will be rendered as LaTeX math. Any text that is meant to appear as LaTeX math should be written as `$math$`. In the HTML version of the docs, MathJax will render the math.

#### Example

The Bessel  $J_\nu$  function of order  $\nu$  is defined to be the function satisfying Bessel's differential equation.

#### LaTeX Recommendations

- If a docstring has any LaTeX, be sure to make it “raw.” See the [Docstring Formatting](#) (page 2998) section for details.
- If you are not sure how to render something, you can use the SymPy [`latex\(\)`](#) (page 2170) function. But be sure to strip out the unimportant parts (the bullet points below).
- Avoid unnecessary `\left` and `\right` (but be sure to use them when they are required).
- Avoid unnecessary `{}`. (For example, write `x^2` instead of `x^{2}`.)
- Use whitespace in a way that makes the equation easiest to read.
- Always check the final rendering to make sure it looks the way you expect it to.
- The HTML documentation build will not fail if there is invalid math, but rather it will show as an error on the page. However, the PDF build, which is run on GitHub Actions on pull requests, will fail. If the LaTeX PDF build fails on CI, there is likely an issue with LaTeX math somewhere.

#### Examples

Correct:

```
\int \sin(x)\,dx
```

Incorrect:

```
\int \sin{\left(x\right)}\, dx
```

For more in-depth resources on how to write math in LaTeX, see:

- <https://math.meta.stackexchange.com/questions/5020/mathjax-basic-tutorial-and-quick-reference>
- <https://en.wikibooks.org/wiki/LaTeX/Mathematics>
- [https://www.overleaf.com/learn/latex/Mathematical\\_expressions](https://www.overleaf.com/learn/latex/Mathematical_expressions)

## Code

Text that should be printed verbatim, such as code, should be surrounded by a set of double backticks like this.

### Example

```
To use this class, define the ``_rewrite()`` and ``_expand()`` methods.
```

Sometimes a variable will be the same in both math and code, and can even appear in the same paragraph, making it difficult to know if it should be formatted as math or code. If the sentence in question is discussing mathematics, then LaTeX should be used, but if the sentence is discussing the SymPy implementation specifically, then code should be used.

In general, the rule of thumb is to consider if the variable in question were something that rendered differently in code and in math. For example, the Greek letter  $\alpha$  would be written as `alpha` in code and  `$\alpha$`  in LaTeX. The reason being that  `$\alpha$`  cannot be used in contexts referring to Python code because it is not valid Python, and conversely `alpha` would be incorrect in a math context because it does not render as the Greek letter ( $\alpha$ ).

### Example

```
class loggamma(Function):
 r"""
 The ``loggamma`` function implements the logarithm of the gamma
 function (i.e., $\log\Gamma(x)$).

 """
```

Variables listed in the parameters after the function name should, in written text, be italicized using Sphinx emphasis with asterisks like `*this*`.

### Example

```
def stirling(n, k, d=None, kind=2, signed=False):
 """
 ...

 The first kind of Stirling number counts the number of permutations of
 n distinct items that have *k* cycles; the second kind counts the
 ways in which *n* distinct items can be partitioned into *k* parts.
 If *d* is given, the "reduced Stirling number of the second kind" is
```

(continues on next page)

(continued from previous page)

```

returned: $S^{\{d\}}(n, k) = S(n - d + 1, k - d + 1)$ with $n \geq k \geq d$.
This counts the ways to partition n consecutive integers into k
groups with no pairwise difference less than d .

```

```

"""

```

Note that in the above example, the first instances of  $n$  and  $k$  are referring to the input parameters of the function `stirling`. Because they are Python variables but also parameters listed by themselves, they are formatted as parameters in italics. The last instances of  $n$  and  $k$  are talking about mathematical expressions, so they are formatted as math.

If a variable is code, but is also a parameter written by itself, the parameter formatting takes precedence and it should be rendered in italics. However, if a parameter appears in a larger code expression it should be within double backticks to be rendered as code. If a variable is only code and not a parameter as well, it should be in double backticks to be rendered as code.

Please note that references to other functions in SymPy are handled differently from parameters or code. If something is referencing another function in SymPy, the cross-reference reStructuredText syntax should be used. See the section on [Cross-Referencing](#) (page 3008) for more information.

## Headings

Section headings in reStructuredText files are created by underlining (and optionally overlining) the section title with a punctuation character at least as long as the text.

Normally, there are no heading levels assigned to certain characters as the structure is determined from the succession of headings. However, for SymPy's documentation, here is a suggested convention:

```

=== with overline: title (top level heading)

```

```

=== heading 1

```

```

- - - heading 2

```

```

^^^ heading 3

```

```

~~~ heading 4

```

```

""" heading 5

```

## Style Preferences

### Spelling and Punctuation

All narrative writing in SymPy follows American spelling and punctuation standards. For example, "color" is preferred over "colour" and commas should be placed inside of quotation marks.

### Examples

If the ``line\_color`` aesthetic is a function of arity 1, then the coloring is a function of the x value of a point.

The term "unrestricted necklace," or "bracelet," is used to imply an object that can be turned over or a sequence that can be reversed.

If there is any ambiguity about the spelling of a word, such as in the case of a function named after a person, refer to the spelling of the actual SymPy function.

For example, Chebyshev polynomials are named after Pafnuty Lvovich Tchebychev, whose name is sometimes transliterated from Russian to be spelled with a "T," but in SymPy it should always be spelled "Chebyshev" to refer to the SymPy function.

### Example

```
class chebyshevt(OrthogonalPolynomial):
    r"""
    Chebyshev polynomial of the first kind,  $T_n(x)$ 
    ...
    """
```

## Capitalization

Title case capitalization is preferred in all SymPy headings.

### Example

```
What is Symbolic Computation?
-----
```

## Tone Preferences

Across SymPy documentation, please write in:

- The present tense (e.g., In the following section, we are going to learn...)
- The first-person inclusive plural (e.g., We did this the long way, but now we can try it the short way...)
- Use the generic pronoun "you" instead of "one." Or use "the reader" or "the user." (e.g., You can access this function by... The user can then access this function by...)
- Use the gender-neutral pronoun "they" instead of "he" or "she." (e.g., A good docstring tells the user exactly what they need to know.)

Avoid extraneous or belittling words such as "obviously," "easily," "simply," "just," or "straightforward."

Avoid unwelcoming or judgement-based phrases like "That is wrong." Instead use friendly and inclusive language like "A common mistake is..."

Avoid extraneous phrases like, "we just have to do one more thing."



## 6.7 Making a Contribution

For in-depth instructions on how to contribute to SymPy's code base including coding conventions, creating your environment, picking an issue to fix, and opening a pull request, please read our full [Development Workflow](#) guide.

## 6.8 Deprecation Policy

This page outlines SymPy's policy on doing deprecations, and describes the steps developers should take to properly deprecate code.

A list of all currently active deprecations in SymPy can be found at [List of active deprecations](#) (page 163).

### 6.8.1 What is a deprecation?

A deprecation is a way to make backwards incompatible changes in a way that allows users to update their code. Deprecated code continues to work as it used to, but whenever someone uses it, it prints a warning to the screen indicating that it will be removed in a future version of SymPy, and indicating what the user should be using instead.

This allows users a chance to update their code without it completely breaking. It also gives SymPy an opportunity to give users an informative message on how to update their code, rather than making their code simply error or start giving wrong answers.

### 6.8.2 Try to avoid backwards incompatible changes in the first place

Backwards incompatible API changes should not be made lightly. Any backwards compatibility break means that users will need to fix their code. Whenever you want to make a breaking change, you should consider whether this is worth the pain for users. Users who have to update their code to match new APIs with every SymPy release will become frustrated with the library, and may go seek a more stable alternative. Consider whether the behavior you want can be done in a way that is compatible with existing APIs. New APIs do not necessarily need to completely supplant old ones. It is sometimes possible for old APIs to exist alongside newer, better designed ones without removing them. For example, the newer [`solveset`](#) (page 858) API was designed to be a superior replacement for the older [`solve`](#) (page 836) API. But the older `solve()` function remains intact and is still supported.

It is important to try to be cognizant of API design whenever adding new functionality. Try to consider what a function may do in the future, and design the API in a way that it can do so without having to make a breaking change. For example, if you add a property to an object `A.attr`, it is impossible to later convert that property into a method `A.attr()` so that it can take arguments, except by doing so in a backwards incompatible way. If you are unsure about your API design for a new functionality, one option is to mark the new functionality as explicitly private or as experimental.

With that being said, it may be decided that the API of SymPy must change in some incompatible way. Some reasons APIs are changed can include:

- The existing API is confusing.

- There is unnecessary redundancy in the API.
- The existing API limits what is possible.

Because one of the core use-cases of SymPy is to be usable as a library, we take API breakage very seriously. Whenever an API breakage is necessary, the following steps should be taken:

- Discuss the API change with the community. Be sure that the improved API is indeed better, and worth a breakage. It is important to get API right so that we will not need to break the API again to “fix” it a second time.
- If it is possible, deprecate the old API. The technical steps for doing this are described [below](#) (page 3020).
- Document the change so that users know how to update their code. The documentation that should added is described [below](#) (page 3022).

### 6.8.3 When does a change require deprecation?

When considering whether a change requires a deprecation, two things must be considered:

- Is the change backwards incompatible?
- Is the behavior changing public API?

A change is backwards incompatible if user code making use of it would stop working after the change.

What counts as “public API” needs to be considered on a case-by-case basis. The exact rules for what does and doesn’t constitute public API for SymPy are still not yet fully codified. Cleaning up the distinction between public and private APIs, as well as the categorization in the reference documentation is currently an [open issue for SymPy](#).

Here are some thing that constitute public API. *Note: these are just general guidelines. This list is not exhaustive, and there are always exceptions to the rules.*

---

#### Public API

- Function names.
  - Keyword argument names.
  - Keyword argument default values.
  - Positional argument order.
  - Submodule names.
  - The mathematical conventions used to define a function.
- 

And here are some things that generally aren’t public API, and therefore don’t require deprecations to change (again, this list is only a general set of guidelines).

---

#### Not Public API

- The precise form of an expression. In general, functions may be changed to return a different but mathematically equivalent form of the same expression. This includes a function returning a value which it was not able to compute previously.

- Functions and methods that are private, i.e., for internal use only. Such things should generally be prefixed with an underscore `_`, although this convention is not currently universally adhered to in the SymPy codebase.
- Anything explicitly marked as “experimental”.
- Changes to behavior that were mathematically incorrect previously (in general, bug fixes are not considered breaking changes, because despite the saying, bugs in SymPy are not features).
- Anything that was added before the most recent release. Code that has not yet made it into a release does not need to be deprecated. If you are going to change the API of new code, it is best to do it before a release is made so that no deprecations are necessary for future releases.

---

Note: both public and private API functions are included in the [reference documentation](#) (page 185), and many functions are not included there which should be, or are not documented at all which should be, so this should not be used to determine whether something public or not.

If you’re unsure, there is no harm in deprecating something even if it might not actually be “public API”.

### 6.8.4 The purpose of deprecation

Deprecation has several purposes:

- To allow existing code to continue to work for a while, giving people a chance to upgrade SymPy without fixing all deprecation issues immediately.
- To warn users that their code will break in a future version.
- To inform users how to fix their code so that it will continue work in future versions.

All deprecation warnings should be something that users can remove by updating their code. Deprecation warnings that fire unconditionally, even when using the “correct” newer APIs, should be avoided.

This also means all deprecated code must have a completely functioning replacement. If there is no way for users to update their code, then this means API in question is not yet ready to be deprecated. The deprecation warning should inform users of a way to change their code so that it works in the same version of SymPy, as well as all future versions, and, if possible, previous versions of SymPy as well. See [below](#) (page 3022).

Deprecations should always

1. Allow users to continue to use the existing APIs unchanged during the deprecation period (with a warning, which can be [silenced](#) (page 164) with `warnings.filterwarnings`).
2. Allow users to always fix their code so that it stops giving the warning.
3. After users fix their code, it should continue to work after the deprecated code is removed.

The third point is important. We do not want the “new” method to itself cause another API break when the deprecation period is over. Doing this would completely defeat the purpose of doing a deprecation.

### When it is not technically possible to deprecate

In some cases, this is not technically possible to make a deprecation that follows the above three rules. API changes of this nature should be considered the most heavily, as they will break people's code immediately without warning. Consideration into how easy it will be for users to support multiple versions of SymPy, one with the change and one without, should also be taken into account.

If you decide that the change is nonetheless worth making, there are two options:

- Make the non-deprecatable change immediately, with no warnings. This will break user code.
- Warn that the code will change in the future. There won't be a way for users to fix their code until a version is released with the breaking change, but they will at least be aware that changes are coming.

Which of the two to make should be decided on a case-by-case basis.

### 6.8.5 How long should deprecations last?

Deprecations should remain intact for **at least 1 year** after the first major release is made with the deprecation. This is only a minimum period: deprecations are allowed to remain intact for longer than this. If a change is especially hard for users to migrate, the deprecation period should be lengthened. The period may also be lengthened for deprecated features that do not impose a significant maintenance burden to keep around.

The deprecation period policy is time-based rather than release-based for a few reasons. Firstly, SymPy does not have a regular release schedule. Sometimes multiple releases will be made in a year, and some years only a single release will be made. Being time-based assures that users have sufficient opportunity to update their code regardless of how often releases happen to be made.

Secondly, SymPy does not make use of a rigorous versioning scheme like semantic versioning. The API surface of SymPy and number of contributions are both large enough that virtually every major release has some deprecations and backwards incompatible changes made in some submodule. Encoding this into the version number would be virtually impossible. The development team also does not backport changes to prior major releases, except in extreme cases. Thus a time-based deprecation scheme is more accurate to SymPy's releasing model than a version-based one would be.

Finally, a time-based scheme removes any temptation to "fudge" a deprecation period down by releasing early. The best way for the developers to accelerate the removal of deprecated functionality is to make a release containing the deprecation as early as possible.

### 6.8.6 How to deprecate code

#### Checklist

Here is a checklist for doing a deprecation. See below for details on each step.

Discuss the backwards incompatible change with the community. Ensure the change is really worth making as per the discussion above.

Remove all instance of the deprecated code from everywhere in the codebase (including doctest examples).

Add `sympy_deprecation_warning()` (page 2067) to the code.

Write a descriptive message for the `sympy_deprecation_warning()` (page 2067). Make sure the message explains both what is deprecated and what to replace it with. The message may be a multiline string and contain examples.

Set `deprecated_since_version` to the version in `sympy/release.py` (without the `.dev`).

Set `active_deprecations_target` to the target used in the `active-deprecations.md` file.

Make sure `stacklevel` is set to the right value so that the deprecation warning shows the user line of code.

Visually confirm the deprecation warning looks good in the console.

Add a `.. deprecated:: <version>` note to the top of the relevant docstring(s).

Add a section to the `doc/src/explanation/active-deprecations.md` file.

Add a cross-reference target (`deprecation-xyz`)= before the section header (this is the same reference used by `active_deprecations_target` above).

Explain what is deprecated and what to replace it with.

Explain *why* the given thing is deprecated.

Add a test using `warns_deprecated_sympy()` (page 2029) that tests that the deprecation warning is issued properly. This test should be the only place in the code that actually uses the deprecated functionality.

Run the test suite to ensure the above test works and that no other code uses the deprecated code, which will cause the tests to fail.

In your PR, add a **BREAKING CHANGE** entry to the release notes for the deprecation.

Once the PR is merged, manually add the change to the “Backwards compatibility breaks and deprecations” section of the release notes on the wiki.

## Adding the deprecation to the code

All deprecations should use `sympy.utilities.exceptions.sympy_deprecation_warning()` (page 2067). If an entire function or method is deprecated, you can use the `sympy.utilities.decorator.deprecated()` (page 2057) decorator. The `deprecated_since_version` and `active_deprecations_target` flags are required. Do not use the `SymPyDeprecationWarning` class directly to issue a deprecation warning. Please see the docstring of `sympy_deprecation_warning()` (page 2067) for more information. See *below* (page 3022) for an example.

Add a test for the deprecated behavior. Use the `sympy.testing.pytest.warns_deprecated_sympy()` (page 2029) context manager.

```
from sympy.testing.pytest import warns_deprecated_sympy

with warns_deprecated_sympy():
    <deprecated behavior>
```

---

**Note:** `warns_deprecated_sympy` is only intended to be used internally by the SymPy test suite. Users of SymPy should use the `warnings` module directly to filter SymPy deprecation warnings. See *Silencing SymPy Deprecation Warnings* (page 164).

---

This has two purposes: to test that the warning is emitted correctly, and to test that the deprecated behavior still actually functions.

If you want to test multiple things and assert that each emits a warning then use separate with blocks for each:

```
with warns_deprecated_sympy():  
    <deprecated behavior1>  
with warns_deprecated_sympy():  
    <deprecated behavior2>
```

This should be the only part of the codebase and test suite that uses the deprecated behavior. Everything else should be changed to use the new, non-deprecated behavior. The SymPy test suite is configured to fail if a `SymPyDeprecationWarning` is issued anywhere except in a `warns_deprecated_sympy()` block. You should not use this function or a `warnings.filterwarnings(SymPyDeprecationWarning)` anywhere except in the test for the deprecation. This includes the documentation examples. The documentation for a deprecated function should just have a note pointing to the non-deprecated alternative. If you want to show a deprecated function in a doctest use `# doctest: +SKIP`. The only exception to this rule is that you may use `ignore_warnings(SymPyDeprecationWarning)` to prevent the exact same warning from triggering twice, i.e., if a deprecated function calls another function that issues the same or a similar warning.

If it is not possible to remove the deprecated behavior somewhere, that is a sign that it is not ready to be deprecated yet. Consider that users may not be able to replace the deprecated behavior for exact same reason.

## Documenting a deprecation

All deprecations should be documented. Every deprecation needs to be documented in three primary places:

- The `sympy_deprecation_warning()` (page 2067) warning text. This text is allowed to be long enough to describe the deprecation, but it should not be more than one paragraph. The primary purpose of the warning text should be *to inform users how to update their code*. The warning text should *not* discuss why a feature was deprecated or unnecessary internal technical details. This discussion can go in the other sections mentioned below. Do not include information in the message that is already part of the metadata provided to the keyword arguments to `sympy_deprecation_warning()`, like the version number or a link to the active deprecations document. Remember that the warning text will be shown in plain-text, so do not use RST or Markdown markup in the text. Code blocks should be clearly delineated with newlines so that they are easy to read. All text in the warning message should be wrapped to 80 characters, except for code examples that cannot be wrapped.

Always include full context of what is deprecated in the message. For example, write “the `abc` keyword to `func()` is deprecated” instead of just “the `abc` keyword is deprecated”. That way if a user has a larger line of code that is using the deprecated functionality, it will be easier for them to see exactly which part is causing the warning.

- A deprecation note in the relevant docstring(s). This should use the `deprecated` Sphinx directive. This uses the syntax `.. deprecated:: <version>`. If the entire function is deprecated, this should be placed at the top of the docstring, right below the first line. Otherwise, if only part of a function is deprecated (e.g., a single keyword argument), it should be placed near the part of the docstring that discusses that feature, e.g., in the parameters list.

The text in the deprecation should be short (no more than a paragraph), explaining what is deprecated and what users should use instead. If you want, you may use the same text here as in the `sympy_deprecation_warning()` (page 2067). Be sure to use RST formatting, including cross-references to the new function if relevant, and a cross-reference to the longer description in the `active-deprecations.md` document (see [below](#) (page 3024)).

If the documentation for the feature is otherwise the same as the replaced feature (i.e., the deprecation is just a renaming of a function or argument), you may replace the rest of the documentation with a note like “see the documentation for <new feature>”. Otherwise, the documentation for the deprecated functionality should be left intact.

Here are some (imaginary) examples:

```
@deprecated("""\
The simplify_this(expr) function is deprecated. Use simplify(expr)
instead.""", deprecated_since_version="1.1",
active_deprecations_target='simplify-this-deprecation')
def simplify_this(expr):
    """
    Simplify ``expr``.

    .. deprecated:: 1.1

    The ``simplify_this`` function is deprecated. Use :func:`simplify`
    instead. See its documentation for more information. See
    :ref:`simplify-this-deprecation` for details.

    """
    return simplify(expr)
```

```
def is_this_zero(x, y=0):
    """
    Determine if  $x = 0$ .

    Parameters
    =====

    x : Expr
        The expression to check.

    y : Expr, optional
        If provided, check if  $x = y$ .

    .. deprecated:: 1.1

    The ``y`` argument to ``is_this_zero`` is deprecated. Use
    ``is_this_zero(x - y)`` instead. See
    :ref:`is-this-zero-y-deprecated` for more details.

    """
    if y != 0:
        sympy_deprecation_warning("""\
The y argument to is_zero() is deprecated. Use is_zero(x - y) instead.""")
```

(continues on next page)



(continued from previous page)

```

→",
        deprecated_since_version="1.1",
        active_deprecations_target='is-this-zero-y-deprecation')
    return simplify(x - y) == 0

```

- A longer description of the deprecation should be added to *the page listing all currently active deprecations* (page 163) in the documentation (in `doc/src/explanation/active-deprecations.md`).

This page is where you can go into more detail about the technical details of a deprecation. Here you should also list *why* a feature was deprecated. You may link to relevant issues, pull requests, and mailing list discussions about the deprecation, but these discussion should be summarized so that users can get the basic idea of why the deprecation without having to read through pages of old discussions. You may also give longer examples here that would not fit in the `sympy_deprecation_warning()` (page 2067) message or `.. deprecated::` text.

Every deprecation should have a cross-reference target (using `(target-name)=` above the section header) so that the `.. deprecated::` note in the relevant docstring can refer to it. This target should also be passed to the `active_deprecations_target` option of `sympy_deprecation_warning()` (page 2067) or `@deprecated` (page 2057). This will automatically put a link to the page in the documentation in the warning message. The target name should include the word “deprecation” or “deprecated” (target names are global in Sphinx, so the target name needs to be unique across the entire documentation).

The section header name should be the thing that is deprecated, and should be a level 3 header under the corresponding version (typically it should be added to the top of the file).

If multiple deprecations are related to one another, they can all share a single section on this page.

If the deprecated function is not included in the top-level `sympy/__init__.py` be sure to clearly indicate which submodule the object is referring to. If you refer to anything that is documented in the Sphinx module reference, cross-reference it, like `{func}~.func_name``.

Note that examples here are helpful, but you generally should not use doctests to show the deprecated functionality, as this will itself raise the deprecation warning and fail the doctest. Instead you may use `# doctest: +SKIP`, or just show the example as a code block instead of a doctest.

Here are examples corresponding to the (imaginary) examples above:

```

(simplify-this-deprecation)=
### `simplify_this()`

The `sympy.simplify.simplify_this()` function is deprecated. It has been
replaced with the {func}~.simplify` function. Code using `simplify_
→this()`
can be fixed by replacing `simplfiy_this(expr)` with `simplify(expr)`.
→The
behavior of the two functions is otherwise identical.

This change was made because `simplify` is a much more Pythonic name than
`simplify_this`.

```



```
(is-this-zero-y-deprecation)=
### `is_this_zero()` second argument
The second argument to {func}~.is_this_zero() is deprecated. Previously
`is_this_zero(x, y)` would check if  $x = y$ . However, this was removed
→ because
it is trivially equivalent to `is_this_zero(x - y)`. Furthermore,
→ allowing
to check  $x=y$  in addition to just  $x=0$  is confusing given the
→ function
is named "is this zero".
```

In particular, replace

```
```py
is_this_zero(expr1, expr2)
```
```

with

```
```py
is_this_zero(expr1 - expr2)
```
```

In addition to the above examples, there are dozens of examples of existing deprecations which can be found by searching for `sympy_deprecation_warning` in the SymPy codebase

## Release notes entry

In the pull request, document the breaking change in the release notes section with **BREAKING CHANGE**.

Once the PR is merged, you should also add it to the “Backwards compatibility breaks and deprecations” section of the release notes for the upcoming release. This needs to be done manually, in addition to the change from the bot. See <https://github.com/sympy/sympy/wiki/Writing-Release-Notes#backwards-compatibility-breaks-and-deprecations>

Whenever a deprecated functionality is removed entirely after its deprecation period, this also needs to be marked as a **BREAKING CHANGE** and added to the “Backwards compatibility breaks and deprecations” section of the release notes.



## BIBLIOGRAPHY

[commutative] [https://en.wikipedia.org/wiki/Commutative\\_property](https://en.wikipedia.org/wiki/Commutative_property)  
[infinite] <https://en.wikipedia.org/wiki/Infinity>  
[antihermitian] [https://en.wikipedia.org/wiki/Skew-Hermitian\\_matrix](https://en.wikipedia.org/wiki/Skew-Hermitian_matrix)  
[complex] [https://en.wikipedia.org/wiki/Complex\\_number](https://en.wikipedia.org/wiki/Complex_number)  
[algebraic] [https://en.wikipedia.org/wiki/Algebraic\\_number](https://en.wikipedia.org/wiki/Algebraic_number)  
[transcendental] [https://en.wikipedia.org/wiki/Transcendental\\_number](https://en.wikipedia.org/wiki/Transcendental_number)  
[extended\_real] [https://en.wikipedia.org/wiki/Extended\\_real\\_number\\_line](https://en.wikipedia.org/wiki/Extended_real_number_line)  
[real] [https://en.wikipedia.org/wiki/Real\\_number](https://en.wikipedia.org/wiki/Real_number)  
[imaginary] [https://en.wikipedia.org/wiki/Imaginary\\_number](https://en.wikipedia.org/wiki/Imaginary_number)  
[rational] [https://en.wikipedia.org/wiki/Rational\\_number](https://en.wikipedia.org/wiki/Rational_number)  
[irrational] [https://en.wikipedia.org/wiki/Irrational\\_number](https://en.wikipedia.org/wiki/Irrational_number)  
[integer] <https://en.wikipedia.org/wiki/Integer>  
[parity] [https://en.wikipedia.org/wiki/Parity\\_\(mathematics\)](https://en.wikipedia.org/wiki/Parity_(mathematics))  
[prime] [https://en.wikipedia.org/wiki/Prime\\_number](https://en.wikipedia.org/wiki/Prime_number)  
[composite] [https://en.wikipedia.org/wiki/Composite\\_number](https://en.wikipedia.org/wiki/Composite_number)  
[zero] <https://en.wikipedia.org/wiki/0>  
[positive] [https://en.wikipedia.org/wiki/Positive\\_real\\_numbers](https://en.wikipedia.org/wiki/Positive_real_numbers)  
[negative] [https://en.wikipedia.org/wiki/Negative\\_number](https://en.wikipedia.org/wiki/Negative_number)  
[R5] [https://en.wikipedia.org/wiki/Predicate\\_\(mathematical\\_logic\)](https://en.wikipedia.org/wiki/Predicate_(mathematical_logic))  
[R6] [https://en.wikipedia.org/wiki/Sexy\\_prime](https://en.wikipedia.org/wiki/Sexy_prime)  
[R3] [https://en.wikipedia.org/wiki/Predicate\\_\(mathematical\\_logic\)](https://en.wikipedia.org/wiki/Predicate_(mathematical_logic))  
[R4] [https://en.wikipedia.org/wiki/Sexy\\_prime](https://en.wikipedia.org/wiki/Sexy_prime)  
[R7] <https://en.wikipedia.org/wiki/Finite>  
[R8] [https://en.wikipedia.org/wiki/Symmetric\\_matrix](https://en.wikipedia.org/wiki/Symmetric_matrix)  
[R9] [https://en.wikipedia.org/wiki/Invertible\\_matrix](https://en.wikipedia.org/wiki/Invertible_matrix)  
[R10] [https://en.wikipedia.org/wiki/Orthogonal\\_matrix](https://en.wikipedia.org/wiki/Orthogonal_matrix)  
[R11] [https://en.wikipedia.org/wiki/Unitary\\_matrix](https://en.wikipedia.org/wiki/Unitary_matrix)

- [R12] [https://en.wikipedia.org/wiki/Positive-definite\\_matrix](https://en.wikipedia.org/wiki/Positive-definite_matrix)
- [R13] <http://mathworld.wolfram.com/UpperTriangularMatrix.html>
- [R14] <http://mathworld.wolfram.com/LowerTriangularMatrix.html>
- [R15] [https://en.wikipedia.org/wiki/Diagonal\\_matrix](https://en.wikipedia.org/wiki/Diagonal_matrix)
- [R16] [https://en.wikipedia.org/wiki/Square\\_matrix](https://en.wikipedia.org/wiki/Square_matrix)
- [R17] <http://mathworld.wolfram.com/SingularMatrix.html>
- [R18] [https://en.wikipedia.org/wiki/Normal\\_matrix](https://en.wikipedia.org/wiki/Normal_matrix)
- [R19] [https://en.wikipedia.org/wiki/Triangular\\_matrix](https://en.wikipedia.org/wiki/Triangular_matrix)
- [R20] <https://en.wikipedia.org/wiki/Integer>
- [R21] [https://en.wikipedia.org/wiki/Rational\\_number](https://en.wikipedia.org/wiki/Rational_number)
- [R22] [https://en.wikipedia.org/wiki/Irrational\\_number](https://en.wikipedia.org/wiki/Irrational_number)
- [R23] [https://en.wikipedia.org/wiki/Real\\_number](https://en.wikipedia.org/wiki/Real_number)
- [R24] <http://mathworld.wolfram.com/HermitianOperator.html>
- [R25] [https://en.wikipedia.org/wiki/Complex\\_number](https://en.wikipedia.org/wiki/Complex_number)
- [R26] [https://en.wikipedia.org/wiki/Imaginary\\_number](https://en.wikipedia.org/wiki/Imaginary_number)
- [R27] <http://mathworld.wolfram.com/HermitianOperator.html>
- [R28] [https://en.wikipedia.org/wiki/Algebraic\\_number](https://en.wikipedia.org/wiki/Algebraic_number)
- [R29] [https://en.wikipedia.org/wiki/Euler%E2%80%93Lagrange\\_equation](https://en.wikipedia.org/wiki/Euler%E2%80%93Lagrange_equation)
- [R30] [https://en.wikipedia.org/wiki/Mathematical\\_singularity](https://en.wikipedia.org/wiki/Mathematical_singularity)
- [R31] Generation of Finite Difference Formulas on Arbitrarily Spaced Grids, Bengt Fornberg; Mathematics of computation; 51; 184; (1988); 699-706; doi:10.1090/S0025-5718-1988-0935077-0
- [R32] [https://en.wikipedia.org/wiki/Convex\\_function](https://en.wikipedia.org/wiki/Convex_function)
- [R33] [http://www.ifp.illinois.edu/~angelia/L3\\_convfunc.pdf](http://www.ifp.illinois.edu/~angelia/L3_convfunc.pdf)
- [R34] [https://en.wikipedia.org/wiki/Logarithmically\\_convex\\_function](https://en.wikipedia.org/wiki/Logarithmically_convex_function)
- [R35] [https://en.wikipedia.org/wiki/Logarithmically\\_concave\\_function](https://en.wikipedia.org/wiki/Logarithmically_concave_function)
- [R36] [https://en.wikipedia.org/wiki/Concave\\_function](https://en.wikipedia.org/wiki/Concave_function)
- [R50] [https://en.wikipedia.org/wiki/Partition\\_%28number\\_theory%29](https://en.wikipedia.org/wiki/Partition_%28number_theory%29)
- [R68] Skiena, S. 'Permutations.' 1.1 in Implementing Discrete Mathematics Combinatorics and Graph Theory with Mathematica. Reading, MA: Addison-Wesley, pp. 3-16, 1990.
- [R69] Knuth, D. E. The Art of Computer Programming, Vol. 4: Combinatorial Algorithms, 1st ed. Reading, MA: Addison-Wesley, 2011.
- [R70] Wendy Myrvold and Frank Ruskey. 2001. Ranking and unranking permutations in linear time. Inf. Process. Lett. 79, 6 (September 2001), 281-284. DOI=10.1016/S0020-0190(01)00141-7
- [R71] D. L. Kreher, D. R. Stinson 'Combinatorial Algorithms' CRC Press, 1999
- [R72] Graham, R. L.; Knuth, D. E.; and Patashnik, O. Concrete Mathematics: A Foundation for Computer Science, 2nd ed. Reading, MA: Addison-Wesley, 1994.

- [R73] [https://en.wikipedia.org/wiki/Permutation#Product\\_and\\_inverse](https://en.wikipedia.org/wiki/Permutation#Product_and_inverse)
- [R74] [https://en.wikipedia.org/wiki/Lehmer\\_code](https://en.wikipedia.org/wiki/Lehmer_code)
- [R75] <https://en.wikipedia.org/wiki/Commutator>
- [R76] <http://www.cp.eng.chula.ac.th/~piak/teaching/algo/algo2008/count-inv.htm>
- [R77] [https://en.wikipedia.org/wiki/Flavius\\_Josephus](https://en.wikipedia.org/wiki/Flavius_Josephus)
- [R78] [https://en.wikipedia.org/wiki/Josephus\\_problem](https://en.wikipedia.org/wiki/Josephus_problem)
- [R79] <http://www.wou.edu/~burtonl/josephus.html>
- [R80] [https://en.wikipedia.org/wiki/Transposition\\_%28mathematics%29#Properties](https://en.wikipedia.org/wiki/Transposition_%28mathematics%29#Properties)
- [R51] Holt, D., Eick, B., O'Brien, E. "Handbook of Computational Group Theory"
- [R52] Seress, A. "Permutation Group Algorithms"
- [R53] [https://en.wikipedia.org/wiki/Schreier\\_vector](https://en.wikipedia.org/wiki/Schreier_vector)
- [R54] [https://en.wikipedia.org/wiki/Nielsen\\_transformation#Product\\_replacement\\_algorithm](https://en.wikipedia.org/wiki/Nielsen_transformation#Product_replacement_algorithm)
- [R55] Frank Celler, Charles R. Leedham-Green, Scott H. Murray, Alice C. Niemeyer, and E.A. O'Brien. "Generating Random Elements of a Finite Group"
- [R56] [https://en.wikipedia.org/wiki/Block\\_%28permutation\\_group\\_theory%29](https://en.wikipedia.org/wiki/Block_%28permutation_group_theory%29)
- [R57] [http://www.algorithmist.com/index.php/Union\\_Find](http://www.algorithmist.com/index.php/Union_Find)
- [R58] [https://en.wikipedia.org/wiki/Multiply\\_transitive\\_group#Multiply\\_transitive\\_groups](https://en.wikipedia.org/wiki/Multiply_transitive_group#Multiply_transitive_groups)
- [R59] [https://en.wikipedia.org/wiki/Center\\_%28group\\_theory%29](https://en.wikipedia.org/wiki/Center_%28group_theory%29)
- [R60] Holt, D., Eick, B., O'Brien, E. "Handbook of computational group theory"
- [R66] [http://www.algorithmist.com/index.php/Union\\_Find](http://www.algorithmist.com/index.php/Union_Find)
- [R62] Holt, D., Eick, B., O'Brien, E. "Handbook of computational group theory"
- [R68] [http://www.algorithmist.com/index.php/Union\\_Find](http://www.algorithmist.com/index.php/Union_Find)
- [R64] The Implementation of Various Algorithms for Permutation Groups in the Computer Algebra System: AXIOM, N.J. Doye, M.Sc. Thesis
- [R65] 1978: John S. Rose: A Course on Group Theory, Introduction to Finite Group Theory: 1.4
- [R81] <http://mathworld.wolfram.com/PolyhedralGroup.html>
- [R82] <http://mathworld.wolfram.com/LabeledTree.html>
- [R83] <https://hamberg.no/erlend/posts/2010-11-06-prufer-sequence-compact-tree-representation.html>
- [R39] Nijenhuis, A. and Wilf, H.S. (1978). Combinatorial Algorithms. Academic Press.
- [R40] Knuth, D. (2011). The Art of Computer Programming, Vol 4 Addison Wesley
- [R41] Knuth, D. (2011). The Art of Computer Programming, Vol 4, Addison Wesley
- [R42] <http://statweb.stanford.edu/~susan/courses/s208/node12.html>
- [R46] [https://en.wikipedia.org/wiki/Symmetric\\_group#Generators\\_and\\_relations](https://en.wikipedia.org/wiki/Symmetric_group#Generators_and_relations)
- [R47] [https://en.wikipedia.org/wiki/Dihedral\\_group](https://en.wikipedia.org/wiki/Dihedral_group)

- [R48] Armstrong, M. "Groups and Symmetry"
- [R49] [http://groupprops.subwiki.org/wiki/Structure\\_theorem\\_for\\_finitely\\_generated\\_abelian\\_groups](http://groupprops.subwiki.org/wiki/Structure_theorem_for_finitely_generated_abelian_groups)
- [R43] Pakianathan, J., Shankar, K., *Nilpotent Numbers*, The American Mathematical Monthly, 107(7), 631-634.
- [R44] Pakianathan, J., Shankar, K., *Nilpotent Numbers*, The American Mathematical Monthly, 107(7), 631-634.
- [R45] Pakianathan, J., Shankar, K., *Nilpotent Numbers*, The American Mathematical Monthly, 107(7), 631-634.
- [R84] Holt, D., Eick, B., O'Brien, E. "Handbook of computational group theory"
- [R85] Holt, D., Eick, B., O'Brien, E. "Handbook of computational group theory"
- [R86] Holt, D., Eick, B., O'Brien, E. "Handbook of computational group theory"
- [CDHW73] John J. Cannon, Lucien A. Dimino, George Havas, and Jane M. Watson. Implementation and analysis of the Todd-Coxeter algorithm. *Math. Comp.*, 27:463- 490, 1973.
- [Ho05] Derek F. Holt, Handbook of Computational Group Theory. In the series 'Discrete Mathematics and its Applications', Chapman & Hall/CRC 2005, xvi + 514 p.
- [Hav91] George Havas, Coset enumeration strategies. In Proceedings of the International Symposium on Symbolic and Algebraic Computation (ISSAC'91), Bonn 1991, pages 191-199. ACM Press, 1991.
- [Ho05] Derek F. Holt, Handbook of Computational Group Theory. In the series 'Discrete Mathematics and its Applications', Chapman & Hall/CRC 2005, xvi + 514 p.
- [R238] [https://en.wikipedia.org/wiki/Complex\\_conjugation](https://en.wikipedia.org/wiki/Complex_conjugation)
- [R239] [https://en.wikipedia.org/wiki/Trigonometric\\_functions](https://en.wikipedia.org/wiki/Trigonometric_functions)
- [R240] <http://dlmf.nist.gov/4.14>
- [R241] <http://functions.wolfram.com/ElementaryFunctions/Sin>
- [R242] <http://mathworld.wolfram.com/TrigonometryAngles.html>
- [R243] [https://en.wikipedia.org/wiki/Trigonometric\\_functions](https://en.wikipedia.org/wiki/Trigonometric_functions)
- [R244] <http://dlmf.nist.gov/4.14>
- [R245] <http://functions.wolfram.com/ElementaryFunctions/Cos>
- [R246] [https://en.wikipedia.org/wiki/Trigonometric\\_functions](https://en.wikipedia.org/wiki/Trigonometric_functions)
- [R247] <http://dlmf.nist.gov/4.14>
- [R248] <http://functions.wolfram.com/ElementaryFunctions/Tan>
- [R249] [https://en.wikipedia.org/wiki/Trigonometric\\_functions](https://en.wikipedia.org/wiki/Trigonometric_functions)
- [R250] <http://dlmf.nist.gov/4.14>
- [R251] <http://functions.wolfram.com/ElementaryFunctions/Cot>
- [R252] [https://en.wikipedia.org/wiki/Trigonometric\\_functions](https://en.wikipedia.org/wiki/Trigonometric_functions)
- [R253] <http://dlmf.nist.gov/4.14>
- [R254] <http://functions.wolfram.com/ElementaryFunctions/Sec>

- [R255] [https://en.wikipedia.org/wiki/Trigonometric\\_functions](https://en.wikipedia.org/wiki/Trigonometric_functions)
- [R256] <http://dlmf.nist.gov/4.14>
- [R257] <http://functions.wolfram.com/ElementaryFunctions/Csc>
- [R258] [https://en.wikipedia.org/wiki/Sinc\\_function](https://en.wikipedia.org/wiki/Sinc_function)
- [R259] [https://en.wikipedia.org/wiki/Inverse\\_trigonometric\\_functions](https://en.wikipedia.org/wiki/Inverse_trigonometric_functions)
- [R260] <http://dlmf.nist.gov/4.23>
- [R261] <http://functions.wolfram.com/ElementaryFunctions/ArcSin>
- [R262] [https://en.wikipedia.org/wiki/Inverse\\_trigonometric\\_functions](https://en.wikipedia.org/wiki/Inverse_trigonometric_functions)
- [R263] <http://dlmf.nist.gov/4.23>
- [R264] <http://functions.wolfram.com/ElementaryFunctions/ArcCos>
- [R265] [https://en.wikipedia.org/wiki/Inverse\\_trigonometric\\_functions](https://en.wikipedia.org/wiki/Inverse_trigonometric_functions)
- [R266] <http://dlmf.nist.gov/4.23>
- [R267] <http://functions.wolfram.com/ElementaryFunctions/ArcTan>
- [R268] <http://dlmf.nist.gov/4.23>
- [R269] <http://functions.wolfram.com/ElementaryFunctions/ArcCot>
- [R270] [https://en.wikipedia.org/wiki/Inverse\\_trigonometric\\_functions](https://en.wikipedia.org/wiki/Inverse_trigonometric_functions)
- [R271] <http://dlmf.nist.gov/4.23>
- [R272] <http://functions.wolfram.com/ElementaryFunctions/ArcSec>
- [R273] <http://reference.wolfram.com/language/ref/ArcSec.html>
- [R274] [https://en.wikipedia.org/wiki/Inverse\\_trigonometric\\_functions](https://en.wikipedia.org/wiki/Inverse_trigonometric_functions)
- [R275] <http://dlmf.nist.gov/4.23>
- [R276] <http://functions.wolfram.com/ElementaryFunctions/ArcCsc>
- [R277] [https://en.wikipedia.org/wiki/Inverse\\_trigonometric\\_functions](https://en.wikipedia.org/wiki/Inverse_trigonometric_functions)
- [R278] <https://en.wikipedia.org/wiki/Atan2>
- [R279] <http://functions.wolfram.com/ElementaryFunctions/ArcTan2>
- [R280] [https://en.wikipedia.org/wiki/Hyperbolic\\_function](https://en.wikipedia.org/wiki/Hyperbolic_function)
- [R281] <http://dlmf.nist.gov/4.37>
- [R282] <http://functions.wolfram.com/ElementaryFunctions/ArcSech/>
- [R283] [https://en.wikipedia.org/wiki/Hyperbolic\\_function](https://en.wikipedia.org/wiki/Hyperbolic_function)
- [R284] <http://dlmf.nist.gov/4.37>
- [R285] <http://functions.wolfram.com/ElementaryFunctions/ArcCsch/>
- [R286] "Concrete mathematics" by Graham, pp. 87
- [R287] <http://mathworld.wolfram.com/CeilingFunction.html>
- [R288] "Concrete mathematics" by Graham, pp. 87
- [R289] <http://mathworld.wolfram.com/FloorFunction.html>
- [R290] [https://en.wikipedia.org/wiki/Fractional\\_part](https://en.wikipedia.org/wiki/Fractional_part)

- [R291] <http://mathworld.wolfram.com/FractionalPart.html>
- [R292] [https://en.wikipedia.org/wiki/Lambert\\_W\\_function](https://en.wikipedia.org/wiki/Lambert_W_function)
- [R293] [https://en.wikipedia.org/wiki/Directed\\_complete\\_partial\\_order](https://en.wikipedia.org/wiki/Directed_complete_partial_order)
- [R294] [https://en.wikipedia.org/wiki/Lattice\\_%28order%29](https://en.wikipedia.org/wiki/Lattice_%28order%29)
- [R295] [https://en.wikipedia.org/wiki/Square\\_root](https://en.wikipedia.org/wiki/Square_root)
- [R296] [https://en.wikipedia.org/wiki/Real\\_root](https://en.wikipedia.org/wiki/Real_root)
- [R297] [https://en.wikipedia.org/wiki/Root\\_of\\_unity](https://en.wikipedia.org/wiki/Root_of_unity)
- [R298] [https://en.wikipedia.org/wiki/Principal\\_value](https://en.wikipedia.org/wiki/Principal_value)
- [R299] <http://mathworld.wolfram.com/CubeRoot.html>
- [R300] [https://en.wikipedia.org/wiki/Square\\_root](https://en.wikipedia.org/wiki/Square_root)
- [R301] [https://en.wikipedia.org/wiki/Principal\\_value](https://en.wikipedia.org/wiki/Principal_value)
- [R302] [https://en.wikipedia.org/wiki/Cube\\_root](https://en.wikipedia.org/wiki/Cube_root)
- [R303] [https://en.wikipedia.org/wiki/Principal\\_value](https://en.wikipedia.org/wiki/Principal_value)
- [R193] [https://en.wikipedia.org/wiki/Bell\\_number](https://en.wikipedia.org/wiki/Bell_number)
- [R194] <http://mathworld.wolfram.com/BellNumber.html>
- [R195] <http://mathworld.wolfram.com/BellPolynomial.html>
- [R196] [https://en.wikipedia.org/wiki/Bernoulli\\_number](https://en.wikipedia.org/wiki/Bernoulli_number)
- [R197] [https://en.wikipedia.org/wiki/Bernoulli\\_polynomial](https://en.wikipedia.org/wiki/Bernoulli_polynomial)
- [R198] <http://mathworld.wolfram.com/BernoulliNumber.html>
- [R199] <http://mathworld.wolfram.com/BernoulliPolynomial.html>
- [R200] [https://www.johndcook.com/blog/binomial\\_coefficients/](https://www.johndcook.com/blog/binomial_coefficients/)
- [R201] [https://en.wikipedia.org/wiki/Catalan\\_number](https://en.wikipedia.org/wiki/Catalan_number)
- [R202] <http://mathworld.wolfram.com/CatalanNumber.html>
- [R203] <http://functions.wolfram.com/GammaBetaErf/CatalanNumber/>
- [R204] <http://geometer.org/mathcircles/catalan.pdf>
- [R205] [https://en.wikipedia.org/wiki/Euler\\_numbers](https://en.wikipedia.org/wiki/Euler_numbers)
- [R206] <http://mathworld.wolfram.com/EulerNumber.html>
- [R207] [https://en.wikipedia.org/wiki/Alternating\\_permutation](https://en.wikipedia.org/wiki/Alternating_permutation)
- [R208] <http://mathworld.wolfram.com/AlternatingPermutation.html>
- [R209] <http://dlmf.nist.gov/24.2#ii>
- [R210] <https://en.wikipedia.org/wiki/Subfactorial>
- [R211] <http://mathworld.wolfram.com/Subfactorial.html>
- [R212] [https://en.wikipedia.org/wiki/Double\\_factorial](https://en.wikipedia.org/wiki/Double_factorial)
- [R213] <http://mathworld.wolfram.com/FallingFactorial.html>
- [R214] Peter Paule, "Greatest Factorial Factorization and Symbolic Summation", Journal of Symbolic Computation, vol. 20, pp. 235-268, 1995.



- [R216] [https://en.wikipedia.org/wiki/Fibonacci\\_number](https://en.wikipedia.org/wiki/Fibonacci_number)
- [R217] <http://mathworld.wolfram.com/FibonacciNumber.html>
- [R218] [https://en.wikipedia.org/wiki/Generalizations\\_of\\_Fibonacci\\_numbers#Tribonacci\\_numbers](https://en.wikipedia.org/wiki/Generalizations_of_Fibonacci_numbers#Tribonacci_numbers)
- [R219] <http://mathworld.wolfram.com/TribonacciNumber.html>
- [R220] <https://oeis.org/A000073>
- [R221] [https://en.wikipedia.org/wiki/Harmonic\\_number](https://en.wikipedia.org/wiki/Harmonic_number)
- [R222] <http://functions.wolfram.com/GammaBetaErf/HarmonicNumber/>
- [R223] <http://functions.wolfram.com/GammaBetaErf/HarmonicNumber2/>
- [R224] [https://en.wikipedia.org/wiki/Lucas\\_number](https://en.wikipedia.org/wiki/Lucas_number)
- [R225] <http://mathworld.wolfram.com/LucasNumber.html>
- [R226] [https://en.wikipedia.org/wiki/Genocchi\\_number](https://en.wikipedia.org/wiki/Genocchi_number)
- [R227] <http://mathworld.wolfram.com/GenocchiNumber.html>
- [R228] [https://en.wikipedia.org/wiki/Partition\\_\(number\\_theory%29](https://en.wikipedia.org/wiki/Partition_(number_theory%29)
- [R229] [https://en.wikipedia.org/wiki/Pentagonal\\_number\\_theorem](https://en.wikipedia.org/wiki/Pentagonal_number_theorem)
- [R230] [https://en.wikipedia.org/wiki/Pochhammer\\_symbol](https://en.wikipedia.org/wiki/Pochhammer_symbol)
- [R231] Peter Paule, "Greatest Factorial Factorization and Symbolic Summation", Journal of Symbolic Computation, vol. 20, pp. 235-268, 1995.
- [R232] [https://en.wikipedia.org/wiki/Stirling\\_numbers\\_of\\_the\\_first\\_kind](https://en.wikipedia.org/wiki/Stirling_numbers_of_the_first_kind)
- [R233] [https://en.wikipedia.org/wiki/Stirling\\_numbers\\_of\\_the\\_second\\_kind](https://en.wikipedia.org/wiki/Stirling_numbers_of_the_second_kind)
- [R234] <https://en.wikipedia.org/wiki/Combination>
- [R235] <http://tinyurl.com/cep849r>
- [R236] <https://en.wikipedia.org/wiki/Permutation>
- [R237] <http://undergraduate.csse.uwa.edu.au/units/CITS7209/partition.pdf>
- [R304] <http://mathworld.wolfram.com/DeltaFunction.html>
- [R305] <http://mathworld.wolfram.com/HeavisideStepFunction.html>
- [R306] <http://dlmf.nist.gov/1.16#iv>
- [R307] [https://en.wikipedia.org/wiki/Singularity\\_function](https://en.wikipedia.org/wiki/Singularity_function)
- [R308] [https://en.wikipedia.org/wiki/Gamma\\_function](https://en.wikipedia.org/wiki/Gamma_function)
- [R309] <http://dlmf.nist.gov/5>
- [R310] <http://mathworld.wolfram.com/GammaFunction.html>
- [R311] <http://functions.wolfram.com/GammaBetaErf/Gamma/>
- [R312] [https://en.wikipedia.org/wiki/Gamma\\_function](https://en.wikipedia.org/wiki/Gamma_function)
- [R313] <http://dlmf.nist.gov/5>
- [R314] <http://mathworld.wolfram.com/LogGammaFunction.html>
- [R315] <http://functions.wolfram.com/GammaBetaErf/LogGamma/>
- [R316] [https://en.wikipedia.org/wiki/Polygamma\\_function](https://en.wikipedia.org/wiki/Polygamma_function)

- [R317] <http://mathworld.wolfram.com/PolygammaFunction.html>
- [R318] <http://functions.wolfram.com/GammaBetaErf/PolyGamma/>
- [R319] <http://functions.wolfram.com/GammaBetaErf/PolyGamma2/>
- [R320] [https://en.wikipedia.org/wiki/Digamma\\_function](https://en.wikipedia.org/wiki/Digamma_function)
- [R321] <http://mathworld.wolfram.com/DigammaFunction.html>
- [R322] <http://functions.wolfram.com/GammaBetaErf/PolyGamma2/>
- [R323] [https://en.wikipedia.org/wiki/Trigamma\\_function](https://en.wikipedia.org/wiki/Trigamma_function)
- [R324] <http://mathworld.wolfram.com/TrigammaFunction.html>
- [R325] <http://functions.wolfram.com/GammaBetaErf/PolyGamma2/>
- [R326] [https://en.wikipedia.org/wiki/Incomplete\\_gamma\\_function#Upper\\_incomplete\\_Gamma\\_function](https://en.wikipedia.org/wiki/Incomplete_gamma_function#Upper_incomplete_Gamma_function)
- [R327] Abramowitz, Milton; Stegun, Irene A., eds. (1965), Chapter 6, Section 5, Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables
- [R328] <http://dlmf.nist.gov/8>
- [R329] <http://functions.wolfram.com/GammaBetaErf/Gamma2/>
- [R330] <http://functions.wolfram.com/GammaBetaErf/Gamma3/>
- [R331] [https://en.wikipedia.org/wiki/Exponential\\_integral#Relation\\_with\\_other\\_functions](https://en.wikipedia.org/wiki/Exponential_integral#Relation_with_other_functions)
- [R332] [https://en.wikipedia.org/wiki/Incomplete\\_gamma\\_function#Lower\\_incomplete\\_Gamma\\_function](https://en.wikipedia.org/wiki/Incomplete_gamma_function#Lower_incomplete_Gamma_function)
- [R333] Abramowitz, Milton; Stegun, Irene A., eds. (1965), Chapter 6, Section 5, Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables
- [R334] <http://dlmf.nist.gov/8>
- [R335] <http://functions.wolfram.com/GammaBetaErf/Gamma2/>
- [R336] <http://functions.wolfram.com/GammaBetaErf/Gamma3/>
- [R337] [https://en.wikipedia.org/wiki/Multivariate\\_gamma\\_function](https://en.wikipedia.org/wiki/Multivariate_gamma_function)
- [R338] [https://en.wikipedia.org/wiki/Beta\\_function](https://en.wikipedia.org/wiki/Beta_function)
- [R339] <http://mathworld.wolfram.com/BetaFunction.html>
- [R340] <http://dlmf.nist.gov/5.12>
- [R341] [https://en.wikipedia.org/wiki/Error\\_function](https://en.wikipedia.org/wiki/Error_function)
- [R342] <http://dlmf.nist.gov/7>
- [R343] <http://mathworld.wolfram.com/Erf.html>
- [R344] <http://functions.wolfram.com/GammaBetaErf/Erf>
- [R345] [https://en.wikipedia.org/wiki/Error\\_function](https://en.wikipedia.org/wiki/Error_function)
- [R346] <http://dlmf.nist.gov/7>
- [R347] <http://mathworld.wolfram.com/Erfc.html>
- [R348] <http://functions.wolfram.com/GammaBetaErf/Erfc>
- [R349] [https://en.wikipedia.org/wiki/Error\\_function](https://en.wikipedia.org/wiki/Error_function)

- [R350] <http://mathworld.wolfram.com/Erfi.html>
- [R351] <http://functions.wolfram.com/GammaBetaErf/Erfi>
- [R352] <http://functions.wolfram.com/GammaBetaErf/Erf2/>
- [R353] [https://en.wikipedia.org/wiki/Error\\_function#Inverse\\_functions](https://en.wikipedia.org/wiki/Error_function#Inverse_functions)
- [R354] <http://functions.wolfram.com/GammaBetaErf/InverseErf/>
- [R355] [https://en.wikipedia.org/wiki/Error\\_function#Inverse\\_functions](https://en.wikipedia.org/wiki/Error_function#Inverse_functions)
- [R356] <http://functions.wolfram.com/GammaBetaErf/InverseErfc/>
- [R357] <http://functions.wolfram.com/GammaBetaErf/InverseErf2/>
- [R358] [https://en.wikipedia.org/wiki/Fresnel\\_integral](https://en.wikipedia.org/wiki/Fresnel_integral)
- [R359] <http://dlmf.nist.gov/7>
- [R360] <http://mathworld.wolfram.com/FresnelIntegrals.html>
- [R361] <http://functions.wolfram.com/GammaBetaErf/FresnelS>
- [R362] The converging factors for the fresnel integrals by John W. Wrench Jr. and Vicki Alley
- [R363] [https://en.wikipedia.org/wiki/Fresnel\\_integral](https://en.wikipedia.org/wiki/Fresnel_integral)
- [R364] <http://dlmf.nist.gov/7>
- [R365] <http://mathworld.wolfram.com/FresnelIntegrals.html>
- [R366] <http://functions.wolfram.com/GammaBetaErf/FresnelC>
- [R367] The converging factors for the fresnel integrals by John W. Wrench Jr. and Vicki Alley
- [R368] <http://dlmf.nist.gov/6.6>
- [R369] [https://en.wikipedia.org/wiki/Exponential\\_integral](https://en.wikipedia.org/wiki/Exponential_integral)
- [R370] Abramowitz & Stegun, section 5: [http://people.math.sfu.ca/~cbm/aands/page\\_228.htm](http://people.math.sfu.ca/~cbm/aands/page_228.htm)
- [R371] <http://dlmf.nist.gov/8.19>
- [R372] <http://functions.wolfram.com/GammaBetaErf/ExpIntegralE/>
- [R373] [https://en.wikipedia.org/wiki/Exponential\\_integral](https://en.wikipedia.org/wiki/Exponential_integral)
- [R374] [https://en.wikipedia.org/wiki/Logarithmic\\_integral](https://en.wikipedia.org/wiki/Logarithmic_integral)
- [R375] <http://mathworld.wolfram.com/LogarithmicIntegral.html>
- [R376] <http://dlmf.nist.gov/6>
- [R377] <http://mathworld.wolfram.com/SoldnersConstant.html>
- [R378] [https://en.wikipedia.org/wiki/Logarithmic\\_integral](https://en.wikipedia.org/wiki/Logarithmic_integral)
- [R379] <http://mathworld.wolfram.com/LogarithmicIntegral.html>
- [R380] <http://dlmf.nist.gov/6>
- [R381] [https://en.wikipedia.org/wiki/Trigonometric\\_integral](https://en.wikipedia.org/wiki/Trigonometric_integral)
- [R382] [https://en.wikipedia.org/wiki/Trigonometric\\_integral](https://en.wikipedia.org/wiki/Trigonometric_integral)
- [R383] [https://en.wikipedia.org/wiki/Trigonometric\\_integral](https://en.wikipedia.org/wiki/Trigonometric_integral)

- [R384] [https://en.wikipedia.org/wiki/Trigonometric\\_integral](https://en.wikipedia.org/wiki/Trigonometric_integral)
- [R385] Abramowitz, Milton; Stegun, Irene A., eds. (1965), "Chapter 9", Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables
- [R386] Luke, Y. L. (1969), The Special Functions and Their Approximations, Volume 1
- [R387] [https://en.wikipedia.org/wiki/Bessel\\_function](https://en.wikipedia.org/wiki/Bessel_function)
- [R388] <http://functions.wolfram.com/Bessel-TypeFunctions/BesselJ/>
- [R389] <http://functions.wolfram.com/Bessel-TypeFunctions/BesselY/>
- [R390] <http://functions.wolfram.com/Bessel-TypeFunctions/BesselI/>
- [R391] <http://functions.wolfram.com/Bessel-TypeFunctions/BesselK/>
- [R392] <http://functions.wolfram.com/Bessel-TypeFunctions/HankelH1/>
- [R393] <http://functions.wolfram.com/Bessel-TypeFunctions/HankelH2/>
- [R394] <http://dlmf.nist.gov/10.47>
- [R395] <http://dlmf.nist.gov/10.47>
- [R396] [https://en.wikipedia.org/wiki/Marcum\\_Q-function](https://en.wikipedia.org/wiki/Marcum_Q-function)
- [R397] <http://mathworld.wolfram.com/MarcumQ-Function.html>
- [R398] [https://en.wikipedia.org/wiki/Airy\\_function](https://en.wikipedia.org/wiki/Airy_function)
- [R399] <http://dlmf.nist.gov/9>
- [R400] [http://www.encyclopediaofmath.org/index.php/Airy\\_functions](http://www.encyclopediaofmath.org/index.php/Airy_functions)
- [R401] <http://mathworld.wolfram.com/AiryFunctions.html>
- [R402] [https://en.wikipedia.org/wiki/Airy\\_function](https://en.wikipedia.org/wiki/Airy_function)
- [R403] <http://dlmf.nist.gov/9>
- [R404] [http://www.encyclopediaofmath.org/index.php/Airy\\_functions](http://www.encyclopediaofmath.org/index.php/Airy_functions)
- [R405] <http://mathworld.wolfram.com/AiryFunctions.html>
- [R406] [https://en.wikipedia.org/wiki/Airy\\_function](https://en.wikipedia.org/wiki/Airy_function)
- [R407] <http://dlmf.nist.gov/9>
- [R408] [http://www.encyclopediaofmath.org/index.php/Airy\\_functions](http://www.encyclopediaofmath.org/index.php/Airy_functions)
- [R409] <http://mathworld.wolfram.com/AiryFunctions.html>
- [R410] [https://en.wikipedia.org/wiki/Airy\\_function](https://en.wikipedia.org/wiki/Airy_function)
- [R411] <http://dlmf.nist.gov/9>
- [R412] [http://www.encyclopediaofmath.org/index.php/Airy\\_functions](http://www.encyclopediaofmath.org/index.php/Airy_functions)
- [R413] <http://mathworld.wolfram.com/AiryFunctions.html>
- [R414] <https://en.wikipedia.org/wiki/B-spline>
- [R415] <http://dlmf.nist.gov/25.11>
- [R416] [https://en.wikipedia.org/wiki/Hurwitz\\_zeta\\_function](https://en.wikipedia.org/wiki/Hurwitz_zeta_function)
- [R417] [https://en.wikipedia.org/wiki/Dirichlet\\_eta\\_function](https://en.wikipedia.org/wiki/Dirichlet_eta_function)
- [R418] Bateman, H.; Erdelyi, A. (1953), Higher Transcendental Functions, Vol. I, New York: McGraw-Hill. Section 1.11.

- [R419] <http://dlmf.nist.gov/25.14>
- [R420] [https://en.wikipedia.org/wiki/Lerch\\_transcendent](https://en.wikipedia.org/wiki/Lerch_transcendent)
- [R421] [https://en.wikipedia.org/wiki/Stieltjes\\_constants](https://en.wikipedia.org/wiki/Stieltjes_constants)
- [R422] Luke, Y. L. (1969), The Special Functions and Their Approximations, Volume 1
- [R423] [https://en.wikipedia.org/wiki/Generalized\\_hypergeometric\\_function](https://en.wikipedia.org/wiki/Generalized_hypergeometric_function)
- [R424] Luke, Y. L. (1969), The Special Functions and Their Approximations, Volume 1
- [R425] [https://en.wikipedia.org/wiki/Meijer\\_G-function](https://en.wikipedia.org/wiki/Meijer_G-function)
- [R426] [https://en.wikipedia.org/wiki/Appell\\_series](https://en.wikipedia.org/wiki/Appell_series)
- [R427] <http://functions.wolfram.com/HypergeometricFunctions/AppellF1/>
- [R428] [https://en.wikipedia.org/wiki/Elliptic\\_integrals](https://en.wikipedia.org/wiki/Elliptic_integrals)
- [R429] <http://functions.wolfram.com/EllipticIntegrals/EllipticK>
- [R430] [https://en.wikipedia.org/wiki/Elliptic\\_integrals](https://en.wikipedia.org/wiki/Elliptic_integrals)
- [R431] <http://functions.wolfram.com/EllipticIntegrals/EllipticF>
- [R432] [https://en.wikipedia.org/wiki/Elliptic\\_integrals](https://en.wikipedia.org/wiki/Elliptic_integrals)
- [R433] <http://functions.wolfram.com/EllipticIntegrals/EllipticE2>
- [R434] <http://functions.wolfram.com/EllipticIntegrals/EllipticE>
- [R435] [https://en.wikipedia.org/wiki/Elliptic\\_integrals](https://en.wikipedia.org/wiki/Elliptic_integrals)
- [R436] <http://functions.wolfram.com/EllipticIntegrals/EllipticPi3>
- [R437] <http://functions.wolfram.com/EllipticIntegrals/EllipticPi>
- [R438] [https://en.wikipedia.org/wiki/Mathieu\\_function](https://en.wikipedia.org/wiki/Mathieu_function)
- [R439] <http://dlmf.nist.gov/28>
- [R440] <http://mathworld.wolfram.com/MathieuBase.html>
- [R441] <http://functions.wolfram.com/MathieuandSpheroidalFunctions/MathieuS/>
- [R442] [https://en.wikipedia.org/wiki/Mathieu\\_function](https://en.wikipedia.org/wiki/Mathieu_function)
- [R443] <http://dlmf.nist.gov/28>
- [R444] <http://mathworld.wolfram.com/MathieuBase.html>
- [R445] <http://functions.wolfram.com/MathieuandSpheroidalFunctions/MathieuC/>
- [R446] [https://en.wikipedia.org/wiki/Mathieu\\_function](https://en.wikipedia.org/wiki/Mathieu_function)
- [R447] <http://dlmf.nist.gov/28>
- [R448] <http://mathworld.wolfram.com/MathieuBase.html>
- [R449] <http://functions.wolfram.com/MathieuandSpheroidalFunctions/MathieuSPrime/>
- [R450] [https://en.wikipedia.org/wiki/Mathieu\\_function](https://en.wikipedia.org/wiki/Mathieu_function)
- [R451] <http://dlmf.nist.gov/28>
- [R452] <http://mathworld.wolfram.com/MathieuBase.html>
- [R453] <http://functions.wolfram.com/MathieuandSpheroidalFunctions/MathieuCPrime/>
- [R454] [https://en.wikipedia.org/wiki/Jacobi\\_polynomials](https://en.wikipedia.org/wiki/Jacobi_polynomials)

- [R455] <http://mathworld.wolfram.com/JacobiPolynomial.html>
- [R456] <http://functions.wolfram.com/Polynomials/JacobiP/>
- [R457] [https://en.wikipedia.org/wiki/Jacobi\\_polynomials](https://en.wikipedia.org/wiki/Jacobi_polynomials)
- [R458] <http://mathworld.wolfram.com/JacobiPolynomial.html>
- [R459] <http://functions.wolfram.com/Polynomials/JacobiP/>
- [R460] [https://en.wikipedia.org/wiki/Gegenbauer\\_polynomials](https://en.wikipedia.org/wiki/Gegenbauer_polynomials)
- [R461] <http://mathworld.wolfram.com/GegenbauerPolynomial.html>
- [R462] <http://functions.wolfram.com/Polynomials/GegenbauerC3/>
- [R463] [https://en.wikipedia.org/wiki/Chebyshev\\_polynomial](https://en.wikipedia.org/wiki/Chebyshev_polynomial)
- [R464] <http://mathworld.wolfram.com/ChebyshevPolynomialoftheFirstKind.html>
- [R465] <http://mathworld.wolfram.com/ChebyshevPolynomialoftheSecondKind.html>
- [R466] <http://functions.wolfram.com/Polynomials/ChebyshevT/>
- [R467] <http://functions.wolfram.com/Polynomials/ChebyshevU/>
- [R468] [https://en.wikipedia.org/wiki/Chebyshev\\_polynomial](https://en.wikipedia.org/wiki/Chebyshev_polynomial)
- [R469] <http://mathworld.wolfram.com/ChebyshevPolynomialoftheFirstKind.html>
- [R470] <http://mathworld.wolfram.com/ChebyshevPolynomialoftheSecondKind.html>
- [R471] <http://functions.wolfram.com/Polynomials/ChebyshevT/>
- [R472] <http://functions.wolfram.com/Polynomials/ChebyshevU/>
- [R473] [https://en.wikipedia.org/wiki/Legendre\\_polynomial](https://en.wikipedia.org/wiki/Legendre_polynomial)
- [R474] <http://mathworld.wolfram.com/LegendrePolynomial.html>
- [R475] <http://functions.wolfram.com/Polynomials/LegendreP/>
- [R476] <http://functions.wolfram.com/Polynomials/LegendreP2/>
- [R477] [https://en.wikipedia.org/wiki/Associated\\_Legendre\\_polynomials](https://en.wikipedia.org/wiki/Associated_Legendre_polynomials)
- [R478] <http://mathworld.wolfram.com/LegendrePolynomial.html>
- [R479] <http://functions.wolfram.com/Polynomials/LegendreP/>
- [R480] <http://functions.wolfram.com/Polynomials/LegendreP2/>
- [R481] [https://en.wikipedia.org/wiki/Hermite\\_polynomial](https://en.wikipedia.org/wiki/Hermite_polynomial)
- [R482] <http://mathworld.wolfram.com/HermitePolynomial.html>
- [R483] <http://functions.wolfram.com/Polynomials/HermiteH/>
- [R484] [https://en.wikipedia.org/wiki/Laguerre\\_polynomial](https://en.wikipedia.org/wiki/Laguerre_polynomial)
- [R485] <http://mathworld.wolfram.com/LaguerrePolynomial.html>
- [R486] <http://functions.wolfram.com/Polynomials/LaguerreL/>
- [R487] <http://functions.wolfram.com/Polynomials/LaguerreL3/>
- [R488] [https://en.wikipedia.org/wiki/Laguerre\\_polynomial#Generalized\\_Laguerre\\_polynomials](https://en.wikipedia.org/wiki/Laguerre_polynomial#Generalized_Laguerre_polynomials)
- [R489] <http://mathworld.wolfram.com/AssociatedLaguerrePolynomial.html>

- [R490] <http://functions.wolfram.com/Polynomials/LaguerreL/>
- [R491] <http://functions.wolfram.com/Polynomials/LaguerreL3/>
- [R492] [https://en.wikipedia.org/wiki/Spherical\\_harmonics](https://en.wikipedia.org/wiki/Spherical_harmonics)
- [R493] <http://mathworld.wolfram.com/SphericalHarmonic.html>
- [R494] <http://functions.wolfram.com/Polynomials/SphericalHarmonicY/>
- [R495] <http://dlmf.nist.gov/14.30>
- [R496] [https://en.wikipedia.org/wiki/Spherical\\_harmonics](https://en.wikipedia.org/wiki/Spherical_harmonics)
- [R497] <http://mathworld.wolfram.com/SphericalHarmonic.html>
- [R498] <http://functions.wolfram.com/Polynomials/SphericalHarmonicY/>
- [R499] [https://en.wikipedia.org/wiki/Spherical\\_harmonics](https://en.wikipedia.org/wiki/Spherical_harmonics)
- [R500] <http://mathworld.wolfram.com/SphericalHarmonic.html>
- [R501] <http://functions.wolfram.com/Polynomials/SphericalHarmonicY/>
- [R502] [https://en.wikipedia.org/wiki/Kronecker\\_delta](https://en.wikipedia.org/wiki/Kronecker_delta)
- [BlogPost] <https://nessgrh.wordpress.com/2011/07/07/tricky-branch-cuts/>
- [R525] M. Bronstein, Symbolic Integration I: Transcendental Functions, Second Edition, Springer-Verlag, 2005, pp. 35-70
- [R526] [http://en.wikibooks.org/wiki/Calculus/Integration\\_techniques](http://en.wikibooks.org/wiki/Calculus/Integration_techniques)
- [R527] <http://www-sop.inria.fr/cafe/Manuel.Bronstein/pmint/index.html>
- [R528] K. Geddes, L. Stefanus, On the Risch-Norman Integration Method and its Implementation in Maple, Proceedings of ISSAC'89, ACM Press, 212-217.
- [R529] J. H. Davenport, On the Parallel Risch Algorithm (I), Proceedings of EURO-CAM'82, LNCS 144, Springer, 144-157.
- [R530] J. H. Davenport, On the Parallel Risch Algorithm (III): Use of Tangents, SIGSAM Bulletin 16 (1982), 3-6.
- [R531] J. H. Davenport, B. M. Trager, On the Parallel Risch Algorithm (II), ACM Transactions on Mathematical Software 11 (1985), 356-362.
- [R532] [https://en.wikipedia.org/wiki/Riemann\\_sum#Methods](https://en.wikipedia.org/wiki/Riemann_sum#Methods)
- [R533] [https://en.wikipedia.org/wiki/Cauchy\\_principal\\_value](https://en.wikipedia.org/wiki/Cauchy_principal_value)
- [R534] <http://mathworld.wolfram.com/CauchyPrincipalValue.html>
- [R535] [https://en.wikipedia.org/wiki/Gaussian\\_quadrature](https://en.wikipedia.org/wiki/Gaussian_quadrature)
- [R536] [http://people.sc.fsu.edu/~jburkardt/cpp\\_src/legendre\\_rule/legendre\\_rule.html](http://people.sc.fsu.edu/~jburkardt/cpp_src/legendre_rule/legendre_rule.html)
- [R537] [https://en.wikipedia.org/wiki/Gauss%E2%80%93Laguerre\\_quadrature](https://en.wikipedia.org/wiki/Gauss%E2%80%93Laguerre_quadrature)
- [R538] [http://people.sc.fsu.edu/~jburkardt/cpp\\_src/laguerre\\_rule/laguerre\\_rule.html](http://people.sc.fsu.edu/~jburkardt/cpp_src/laguerre_rule/laguerre_rule.html)
- [R539] [https://en.wikipedia.org/wiki/Gauss-Hermite\\_Quadrature](https://en.wikipedia.org/wiki/Gauss-Hermite_Quadrature)
- [R540] [http://people.sc.fsu.edu/~jburkardt/cpp\\_src/hermite\\_rule/hermite\\_rule.html](http://people.sc.fsu.edu/~jburkardt/cpp_src/hermite_rule/hermite_rule.html)
- [R541] [http://people.sc.fsu.edu/~jburkardt/cpp\\_src/gen\\_hermite\\_rule/gen\\_hermite\\_rule.html](http://people.sc.fsu.edu/~jburkardt/cpp_src/gen_hermite_rule/gen_hermite_rule.html)
- [R542] [https://en.wikipedia.org/wiki/Gauss%E2%80%93Laguerre\\_quadrature](https://en.wikipedia.org/wiki/Gauss%E2%80%93Laguerre_quadrature)



- [R543] [http://people.sc.fsu.edu/~jburkardt/cpp\\_src/gen\\_laguerre\\_rule/gen\\_laguerre\\_rule.html](http://people.sc.fsu.edu/~jburkardt/cpp_src/gen_laguerre_rule/gen_laguerre_rule.html)
- [R544] [https://en.wikipedia.org/wiki/Chebyshev%E2%80%93Gauss\\_quadrature](https://en.wikipedia.org/wiki/Chebyshev%E2%80%93Gauss_quadrature)
- [R545] [http://people.sc.fsu.edu/~jburkardt/cpp\\_src/chebyshev1\\_rule/chebyshev1\\_rule.html](http://people.sc.fsu.edu/~jburkardt/cpp_src/chebyshev1_rule/chebyshev1_rule.html)
- [R546] [https://en.wikipedia.org/wiki/Chebyshev%E2%80%93Gauss\\_quadrature](https://en.wikipedia.org/wiki/Chebyshev%E2%80%93Gauss_quadrature)
- [R547] [http://people.sc.fsu.edu/~jburkardt/cpp\\_src/chebyshev2\\_rule/chebyshev2\\_rule.html](http://people.sc.fsu.edu/~jburkardt/cpp_src/chebyshev2_rule/chebyshev2_rule.html)
- [R548] [https://en.wikipedia.org/wiki/Gauss%E2%80%93Jacobi\\_quadrature](https://en.wikipedia.org/wiki/Gauss%E2%80%93Jacobi_quadrature)
- [R549] [http://people.sc.fsu.edu/~jburkardt/cpp\\_src/jacobi\\_rule/jacobi\\_rule.html](http://people.sc.fsu.edu/~jburkardt/cpp_src/jacobi_rule/jacobi_rule.html)
- [R550] [http://people.sc.fsu.edu/~jburkardt/cpp\\_src/gegenbauer\\_rule/gegenbauer\\_rule.html](http://people.sc.fsu.edu/~jburkardt/cpp_src/gegenbauer_rule/gegenbauer_rule.html)
- [R551] [https://en.wikipedia.org/wiki/Gaussian\\_quadrature#Gauss.E2.80.93Lobatto\\_rules](https://en.wikipedia.org/wiki/Gaussian_quadrature#Gauss.E2.80.93Lobatto_rules)
- [R552] [http://people.math.sfu.ca/~cbm/aands/page\\_888.htm](http://people.math.sfu.ca/~cbm/aands/page_888.htm)
- [R742] Big O notation
- [R743] [https://en.wikipedia.org/wiki/Residue\\_theorem](https://en.wikipedia.org/wiki/Residue_theorem)
- [R737] [https://en.wikipedia.org/wiki/Gibbs\\_phenomenon](https://en.wikipedia.org/wiki/Gibbs_phenomenon)
- [R738] [https://en.wikipedia.org/wiki/Sigma\\_approximation](https://en.wikipedia.org/wiki/Sigma_approximation)
- [R739] <https://mathworld.wolfram.com/FourierSeries.html>
- [R731] Comtet, Louis: Advanced combinatorics; the art of finite and infinite expansions. Reidel, 1974.
- [R732] Comtet, Louis: Advanced combinatorics; the art of finite and infinite expansions. Reidel, 1974.
- [R733] Formal Power Series - Dominik Gruntz, Wolfram Koepf
- [R734] Power Series in Computer Algebra - Wolfram Koepf
- [R735] Formal Power Series - Dominik Gruntz, Wolfram Koepf
- [R736] Power Series in Computer Algebra - Wolfram Koepf
- [R740] <https://reference.wolfram.com/language/ref/DifferenceDelta.html>
- [R741] Computing Limits of Sequences - Manuel Kauers
- [R762] M. Monagan, R. Pearce, Rational Simplification Modulo a Polynomial Ideal, <http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.163.6984> (specifically, the second algorithm)
- [R763] <http://researcher.watson.ibm.com/researcher/files/us-fagin/symb85.pdf>
- [R764] D. J. Jeffrey and A. D. Rich, 'Simplifying Square Roots of Square Roots by Denesting' (available at <http://www.cybertester.com/data/denest.pdf>)
- [Roach1996] Kelly B. Roach. Hypergeometric Function Representations. In: Proceedings of the 1996 International Symposium on Symbolic and Algebraic Computation, pages 301-308, New York, 1996. ACM.



- [Roach1997] Kelly B. Roach. Meijer G Function Representations. In: Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation, pages 205-211, New York, 1997. ACM.
- [Luke1969] Luke, Y. L. (1969), The Special Functions and Their Approximations, Volume 1.
- [Prudnikov1990] A. P. Prudnikov, Yu. A. Brychkov and O. I. Marichev (1990). Integrals and Series: More Special Functions, Vol. 3, Gordon and Breach Science Publisher.
- [R759] [https://en.wikipedia.org/wiki/Morrie%27s\\_law](https://en.wikipedia.org/wiki/Morrie%27s_law)
- [R760] [https://en.wikipedia.org/wiki/List\\_of\\_trigonometric\\_identities#Power-reduction\\_formulae](https://en.wikipedia.org/wiki/List_of_trigonometric_identities#Power-reduction_formulae)
- [R761] <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.657.2478&rep=rep1&type=pdf>
- [R765] Methods to solve  $Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$ , [online], Available: <http://www.alpertron.com.ar/METHODS.HTM>
- [R766] Solving the equation  $ax^2 + bxy + cy^2 + dx + ey + f = 0$ , [online], Available: <https://web.archive.org/web/20160323033111/http://www.jpr2718.org/ax2p.pdf>
- [R767] Solving the generalized Pell equation  $x^2 - Dy^2 = N$ , John P. Robertson, July 31, 2004, Pages 16 - 17. [online], Available: <https://web.archive.org/web/20160323033128/http://www.jpr2718.org/pell.pdf>
- [R768] A. Nitaj, "L'algorithme de Cornacchia"
- [R769] Solving the diophantine equation  $ax^2 + by^2 = m$  by Cornacchia's method, [online], Available: <http://www.numbertheory.org/php/cornacchia.html>
- [R770] Solving the generalized Pell equation  $x^2 - Dy^2 = N$ , John P. Robertson, July 31, 2004, Page 15. <https://web.archive.org/web/20160323033128/http://www.jpr2718.org/pell.pdf>
- [R771] Solving the equation  $ax^2 + bxy + cy^2 + dx + ey + f = 0$ , John P. Robertson, May 8, 2003, Page 7 - 11. <https://web.archive.org/web/20160323033111/http://www.jpr2718.org/ax2p.pdf>
- [R772] Solving the equation  $ax^2 + bxy + cy^2 + dx + ey + f = 0$ , John P. Robertson, May 8, 2003, Page 7 - 11. <https://web.archive.org/web/20160323033111/http://www.jpr2718.org/ax2p.pdf>
- [R773] Efficient Solution of Rational Conics, J. E. Cremona and D. Rusin, Mathematics of Computation, Volume 00, Number 0.
- [R774] Representing an integer as a sum of three squares, [online], Available: [http://www.proofwiki.org/wiki/Integer\\_as\\_Sum\\_of\\_Three\\_Squares](http://www.proofwiki.org/wiki/Integer_as_Sum_of_Three_Squares)
- [R775] Representing a number as a sum of three squares, [online], Available: <http://schorn.ch/lagrange.html>
- [R776] Representing a number as a sum of four squares, [online], Available: <http://schorn.ch/lagrange.html>
- [R777] Solving the generalized Pell equation  $x^2 - Dy^2 = N$ , John P. Robertson, July 31, 2004, Pages 4 - 8. <https://web.archive.org/web/20160323033128/http://www.jpr2718.org/pell.pdf>
- [R778] Solving the generalized Pell equation  $x^2 - Dy^2 = N$ , John P. Robertson, July 31, 2004, Page 12. <https://web.archive.org/web/20160323033128/http://www.jpr2718.org/pell.pdf>

- [R779] The algorithmic resolution of Diophantine equations, Nigel P. Smart, London Mathematical Society Student Texts 41, Cambridge University Press, Cambridge, 1998.
- [R780] The algorithmic resolution of Diophantine equations, Nigel P. Smart, London Mathematical Society Student Texts 41, Cambridge University Press, Cambridge, 1998.
- [R781] Efficient Solution of Rational Conics, J. E. Cremona and D. Rusin, [online], Available: <http://eprints.nottingham.ac.uk/60/1/kvxefz87.pdf>
- [R782] Gaussian lattice Reduction [online]. Available: <http://home.ie.cuhk.edu.hk/~wkshum/wordpress/?p=404>
- [R783] Efficient Solution of Rational Conics, J. E. Cremona and D. Rusin, Mathematics of Computation, Volume 00, Number 0.
- [R784] Efficient Solution of Rational Conics, J. E. Cremona and D. Rusin, Mathematics of Computation, Volume 00, Number 0.
- [R785] Diophantine Equations, L. J. Mordell, page 48.
- [R786] Representing a number as a sum of four squares, [online], Available: <http://schorn.ch/lagrange.html>
- [R787] Legendre's Theorem, Lagrange's Descent, [http://public.csusm.edu/aitken\\_html/notes/legendre.pdf](http://public.csusm.edu/aitken_html/notes/legendre.pdf)
- [R788] Methods to solve  $Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$ , [online], Available: <http://www.alpertron.com.ar/METHODS.HTM>
- [R789] Solving the equation  $ax^2 + bxy + cy^2 + dx + ey + f = 0$ , [online], Available: <https://web.archive.org/web/20160323033111/http://www.jpr2718.org/ax2p.pdf>
- [R790] Representing an integer as a sum of three squares, [online], Available: [http://www.proofwiki.org/wiki/Integer\\_as\\_Sum\\_of\\_Three\\_Squares](http://www.proofwiki.org/wiki/Integer_as_Sum_of_Three_Squares)
- [R791] [https://en.wikipedia.org/wiki/Matrix\\_differential\\_equation](https://en.wikipedia.org/wiki/Matrix_differential_equation)
- [R792] [https://en.wikipedia.org/wiki/Defective\\_matrix](https://en.wikipedia.org/wiki/Defective_matrix)
- [R793] [https://en.wikipedia.org/wiki/Jordan\\_matrix](https://en.wikipedia.org/wiki/Jordan_matrix)
- [R794] [https://en.wikipedia.org/wiki/Jordan\\_normal\\_form](https://en.wikipedia.org/wiki/Jordan_normal_form)
- [R795] [https://en.wikipedia.org/wiki/Jordan\\_normal\\_form](https://en.wikipedia.org/wiki/Jordan_normal_form)
- [R796] [https://en.wikipedia.org/wiki/Matrix\\_exponential](https://en.wikipedia.org/wiki/Matrix_exponential)
- [R797] S. A. Abramov, M. Bronstein and M. Petkovsek, On polynomial solutions of linear operator equations, in: T. Levelt, ed., Proc. ISSAC '95, ACM Press, New York, 1995, 290-296.
- [R798] M. Petkovsek, Hypergeometric solutions of linear recurrences with polynomial coefficients, J. Symbolic Computation, 14 (1992), 243-264.
- [R799] M. Petkovsek, H. S. Wilf, D. Zeilberger, A = B, 1996.
- [R800] S. A. Abramov, Rational solutions of linear difference and q-difference equations with polynomial coefficients, in: T. Levelt, ed., Proc. ISSAC '95, ACM Press, New York, 1995, 285-289
- [R801] M. Petkovsek, Hypergeometric solutions of linear recurrences with polynomial coefficients, J. Symbolic Computation, 14 (1992), 243-264.

- [R802] M. Petkovsek, H. S. Wilf, D. Zeilberger,  $A = B$ , 1996.
- [R1] <http://www.euclideanspace.com/maths/algebra/realNormedAlgebra/quaternions/>
- [R2] <https://en.wikipedia.org/wiki/Quaternion>
- [R87] Michael Karr, "Summation in Finite Terms", Journal of the ACM, Volume 28 Issue 2, April 1981, Pages 305-350 <http://dl.acm.org/citation.cfm?doid=322248.322255>
- [R88] [https://en.wikipedia.org/wiki/Summation#Capital-sigma\\_notation](https://en.wikipedia.org/wiki/Summation#Capital-sigma_notation)
- [R89] [https://en.wikipedia.org/wiki/Empty\\_sum](https://en.wikipedia.org/wiki/Empty_sum)
- [R90] [https://en.wikipedia.org/wiki/Absolute\\_convergence](https://en.wikipedia.org/wiki/Absolute_convergence)
- [R91] [https://en.wikipedia.org/wiki/Convergence\\_tests](https://en.wikipedia.org/wiki/Convergence_tests)
- [R92] Michael Karr, "Summation in Finite Terms", Journal of the ACM, Volume 28 Issue 2, April 1981, Pages 305-350 <http://dl.acm.org/citation.cfm?doid=322248.322255>
- [R93] Michael Karr, "Summation in Finite Terms", Journal of the ACM, Volume 28 Issue 2, April 1981, Pages 305-350 <http://dl.acm.org/citation.cfm?doid=322248.322255>
- [R94] [https://en.wikipedia.org/wiki/Multiplication#Capital\\_Pi\\_notation](https://en.wikipedia.org/wiki/Multiplication#Capital_Pi_notation)
- [R95] [https://en.wikipedia.org/wiki/Empty\\_product](https://en.wikipedia.org/wiki/Empty_product)
- [R96] [https://en.wikipedia.org/wiki/Infinite\\_product](https://en.wikipedia.org/wiki/Infinite_product)
- [R97] Michael Karr, "Summation in Finite Terms", Journal of the ACM, Volume 28 Issue 2, April 1981, Pages 305-350 <http://dl.acm.org/citation.cfm?doid=322248.322255>
- [R98] Marko Petkovsek, Herbert S. Wilf, Doron Zeilberger,  $A = B$ , AK Peters, Ltd., Wellesley, MA, USA, 1997, pp. 73-100
- [R99] [https://en.wikipedia.org/wiki/Negative\\_number](https://en.wikipedia.org/wiki/Negative_number)
- [R100] [https://en.wikipedia.org/wiki/Parity\\_%28mathematics%29](https://en.wikipedia.org/wiki/Parity_%28mathematics%29)
- [R101] [https://en.wikipedia.org/wiki/Imaginary\\_number](https://en.wikipedia.org/wiki/Imaginary_number)
- [R102] [https://en.wikipedia.org/wiki/Composite\\_number](https://en.wikipedia.org/wiki/Composite_number)
- [R103] [https://en.wikipedia.org/wiki/Irrational\\_number](https://en.wikipedia.org/wiki/Irrational_number)
- [R104] [https://en.wikipedia.org/wiki/Prime\\_number](https://en.wikipedia.org/wiki/Prime_number)
- [R105] <https://en.wikipedia.org/wiki/Finite>
- [R106] <https://docs.python.org/3/library/math.html#math.isfinite>
- [R107] <http://docs.scipy.org/doc/numpy/reference/generated/numpy.isfinite.html>
- [R108] Gruntz, Dominik. A new algorithm for computing asymptotic series. In: Proc. 1993 Int. Symp. Symbolic and Algebraic Computation. 1993. pp. 239-244.
- [R109] Gruntz thesis - p90
- [R110] [http://en.wikipedia.org/wiki/Asymptotic\\_expansion](http://en.wikipedia.org/wiki/Asymptotic_expansion)
- [R111] [https://en.wikipedia.org/wiki/Algebraic\\_expression](https://en.wikipedia.org/wiki/Algebraic_expression)
- [R112] [https://en.wikipedia.org/wiki/Euclidean\\_algorithm](https://en.wikipedia.org/wiki/Euclidean_algorithm)

- [R113] <https://en.wikipedia.org/wiki/Zero>
- [R114] [https://en.wikipedia.org/wiki/1\\_%28number%29](https://en.wikipedia.org/wiki/1_%28number%29)
- [R115] [https://en.wikipedia.org/wiki/%E2%88%921\\_%28number%29](https://en.wikipedia.org/wiki/%E2%88%921_%28number%29)
- [R116] [https://en.wikipedia.org/wiki/One\\_half](https://en.wikipedia.org/wiki/One_half)
- [R117] <https://en.wikipedia.org/wiki/NaN>
- [R118] <https://en.wikipedia.org/wiki/Infinity>
- [R119] [https://en.wikipedia.org/wiki/E\\_%28mathematical\\_constant%29](https://en.wikipedia.org/wiki/E_%28mathematical_constant%29)
- [R120] [https://en.wikipedia.org/wiki/Imaginary\\_unit](https://en.wikipedia.org/wiki/Imaginary_unit)
- [R121] <https://en.wikipedia.org/wiki/Pi>
- [R122] [https://en.wikipedia.org/wiki/Euler%E2%80%93Mascheroni\\_constant](https://en.wikipedia.org/wiki/Euler%E2%80%93Mascheroni_constant)
- [R123] [https://en.wikipedia.org/wiki/Catalan%27s\\_constant](https://en.wikipedia.org/wiki/Catalan%27s_constant)
- [R124] [https://en.wikipedia.org/wiki/Golden\\_ratio](https://en.wikipedia.org/wiki/Golden_ratio)
- [R125] [https://en.wikipedia.org/wiki/Generalizations\\_of\\_Fibonacci\\_numbers#Tribonacci\\_numbers](https://en.wikipedia.org/wiki/Generalizations_of_Fibonacci_numbers#Tribonacci_numbers)
- [R126] [https://en.wikipedia.org/wiki/Modular\\_multiplicative\\_inverse](https://en.wikipedia.org/wiki/Modular_multiplicative_inverse)
- [R127] [https://en.wikipedia.org/wiki/Extended\\_Euclidean\\_algorithm](https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm)
- [R128] <https://en.wikipedia.org/wiki/Exponentiation>
- [R129] [https://en.wikipedia.org/wiki/Exponentiation#Zero\\_to\\_the\\_power\\_of\\_zero](https://en.wikipedia.org/wiki/Exponentiation#Zero_to_the_power_of_zero)
- [R130] [https://en.wikipedia.org/wiki/Indeterminate\\_forms](https://en.wikipedia.org/wiki/Indeterminate_forms)
- [R131] 

This implementation detail is that Python provides no reliable method to determine that a chained inequality is being built. Chained comparison operators are evaluated pairwise, using “and” logic (see <http://docs.python.org/reference/expressions.html#not-in>). This is done in an efficient way, so that each object being compared is only evaluated once and the comparison can short-circuit. For example,  $1 > 2 > 3$  is evaluated by Python as  $(1 > 2)$  and  $(2 > 3)$ . The and operator coerces each side into a bool, returning the object itself when it short-circuits. The bool of the `-Than` operators will raise `TypeError` on purpose, because SymPy cannot determine the mathematical ordering of symbolic expressions. Thus, if we were to compute  $x > y > z$ , with  $x$ ,  $y$ , and  $z$  being Symbols, Python converts the statement (roughly) into these steps:

  - (1)  $x > y > z$
  - (2)  $(x > y)$  and  $(y > z)$
  - (3) `GreaterThanObject` and  $(y > z)$
  - (4) `GreaterThanObject.__bool__()` and  $(y > z)$
  - (5) `TypeError`

Because of the and added at step 2, the statement gets turned into a weak ternary statement, and the first object’s `__bool__` method will raise `TypeError`. Thus, creating a chained inequality is not possible.

In Python, there is no way to override the and operator, or to control how it short circuits, so it is impossible to make something like  $x > y >$

z work. There was a PEP to change this, [PEP 335](#), but it was officially closed in March, 2012.

- [R132] This implementation detail is that Python provides no reliable method to determine that a chained inequality is being built. Chained comparison operators are evaluated pairwise, using “and” logic (see <http://docs.python.org/reference/expressions.html#not-in>). This is done in an efficient way, so that each object being compared is only evaluated once and the comparison can short-circuit. For example,  $1 > 2 > 3$  is evaluated by Python as  $(1 > 2)$  and  $(2 > 3)$ . The and operator coerces each side into a bool, returning the object itself when it short-circuits. The bool of the -Than operators will raise TypeError on purpose, because SymPy cannot determine the mathematical ordering of symbolic expressions. Thus, if we were to compute  $x > y > z$ , with x, y, and z being Symbols, Python converts the statement (roughly) into these steps:

- (1)  $x > y > z$
- (2)  $(x > y)$  and  $(y > z)$
- (3) (GreaterThanObject) and  $(y > z)$
- (4) (GreaterThanObject.\_\_bool\_\_()) and  $(y > z)$
- (5) TypeError

Because of the and added at step 2, the statement gets turned into a weak ternary statement, and the first object’s \_\_bool\_\_ method will raise TypeError. Thus, creating a chained inequality is not possible.

In Python, there is no way to override the and operator, or to control how it short circuits, so it is impossible to make something like  $x > y > z$  work. There was a PEP to change this, [PEP 335](#), but it was officially closed in March, 2012.

- [R133] This implementation detail is that Python provides no reliable method to determine that a chained inequality is being built. Chained comparison operators are evaluated pairwise, using “and” logic (see <http://docs.python.org/reference/expressions.html#not-in>). This is done in an efficient way, so that each object being compared is only evaluated once and the comparison can short-circuit. For example,  $1 > 2 > 3$  is evaluated by Python as  $(1 > 2)$  and  $(2 > 3)$ . The and operator coerces each side into a bool, returning the object itself when it short-circuits. The bool of the -Than operators will raise TypeError on purpose, because SymPy cannot determine the mathematical ordering of symbolic expressions. Thus, if we were to compute  $x > y > z$ , with x, y, and z being Symbols, Python converts the statement (roughly) into these steps:

- (1)  $x > y > z$
- (2)  $(x > y)$  and  $(y > z)$
- (3) (GreaterThanObject) and  $(y > z)$
- (4) (GreaterThanObject.\_\_bool\_\_()) and  $(y > z)$
- (5) TypeError

Because of the and added at step 2, the statement gets turned into a weak ternary statement, and the first object’s \_\_bool\_\_ method will raise TypeError. Thus, creating a chained inequality is not possible.

In Python, there is no way to override the and operator, or to control how it short circuits, so it is impossible to make something like  $x > y >$

z work. There was a PEP to change this, [PEP 335](#), but it was officially closed in March, 2012.

[R134] This implementation detail is that Python provides no reliable method to determine that a chained inequality is being built. Chained comparison operators are evaluated pairwise, using “and” logic (see <http://docs.python.org/reference/expressions.html#not-in>). This is done in an efficient way, so that each object being compared is only evaluated once and the comparison can short-circuit. For example,  $1 > 2 > 3$  is evaluated by Python as  $(1 > 2)$  and  $(2 > 3)$ . The and operator coerces each side into a bool, returning the object itself when it short-circuits. The bool of the `-Than` operators will raise `TypeError` on purpose, because SymPy cannot determine the mathematical ordering of symbolic expressions. Thus, if we were to compute  $x > y > z$ , with  $x$ ,  $y$ , and  $z$  being Symbols, Python converts the statement (roughly) into these steps:

- (1)  $x > y > z$
- (2)  $(x > y)$  and  $(y > z)$
- (3) `GreaterThanObject` and  $(y > z)$
- (4) `GreaterThanObject.__bool__()` and  $(y > z)$
- (5) `TypeError`

Because of the and added at step 2, the statement gets turned into a weak ternary statement, and the first object’s `__bool__` method will raise `TypeError`. Thus, creating a chained inequality is not possible.

In Python, there is no way to override the and operator, or to control how it short circuits, so it is impossible to make something like  $x > y > z$  work. There was a PEP to change this, [PEP 335](#), but it was officially closed in March, 2012.

- [R135] [http://reference.wolfram.com/legacy/v5\\_2/Built-inFunctions/AlgebraicComputation/Calculus/D.html](http://reference.wolfram.com/legacy/v5_2/Built-inFunctions/AlgebraicComputation/Calculus/D.html)
- [R164] [https://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey\\_FFT\\_algorithm](https://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm)
- [R165] <http://mathworld.wolfram.com/FastFourierTransform.html>
- [R166] [https://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey\\_FFT\\_algorithm](https://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm)
- [R167] <http://mathworld.wolfram.com/FastFourierTransform.html>
- [R168] <http://www.apfloat.org/ntt.html>
- [R169] <http://mathworld.wolfram.com/NumberTheoreticTransform.html>
- [R170] [https://en.wikipedia.org/wiki/Discrete\\_Fourier\\_transform\\_\(general%29](https://en.wikipedia.org/wiki/Discrete_Fourier_transform_(general%29)
- [R171] <http://www.apfloat.org/ntt.html>
- [R172] <http://mathworld.wolfram.com/NumberTheoreticTransform.html>
- [R173] [https://en.wikipedia.org/wiki/Discrete\\_Fourier\\_transform\\_\(general%29](https://en.wikipedia.org/wiki/Discrete_Fourier_transform_(general%29)
- [R174] [https://en.wikipedia.org/wiki/Hadamard\\_transform](https://en.wikipedia.org/wiki/Hadamard_transform)
- [R175] [https://en.wikipedia.org/wiki/Fast\\_Walsh%E2%80%93Hadamard\\_transform](https://en.wikipedia.org/wiki/Fast_Walsh%E2%80%93Hadamard_transform)
- [R176] [https://en.wikipedia.org/wiki/Hadamard\\_transform](https://en.wikipedia.org/wiki/Hadamard_transform)
- [R177] [https://en.wikipedia.org/wiki/Fast\\_Walsh%E2%80%93Hadamard\\_transform](https://en.wikipedia.org/wiki/Fast_Walsh%E2%80%93Hadamard_transform)
- [R178] [https://en.wikipedia.org/wiki/M%C3%B6bius\\_inversion\\_formula](https://en.wikipedia.org/wiki/M%C3%B6bius_inversion_formula)



- [R179] <https://people.csail.mit.edu/rrw/presentations/subset-conv.pdf>
- [R180] <https://arxiv.org/pdf/1211.0189.pdf>
- [R181] [https://en.wikipedia.org/wiki/M%C3%B6bius\\_inversion\\_formula](https://en.wikipedia.org/wiki/M%C3%B6bius_inversion_formula)
- [R182] <https://people.csail.mit.edu/rrw/presentations/subset-conv.pdf>
- [R183] <https://arxiv.org/pdf/1211.0189.pdf>
- [R184] [https://en.wikipedia.org/wiki/Convolution\\_theorem](https://en.wikipedia.org/wiki/Convolution_theorem)
- [R185] [https://en.wikipedia.org/wiki/Discrete\\_Fourier\\_transform\\_\(general%29](https://en.wikipedia.org/wiki/Discrete_Fourier_transform_(general%29)
- [R186] [https://en.wikipedia.org/wiki/Convolution\\_theorem](https://en.wikipedia.org/wiki/Convolution_theorem)
- [R187] [https://en.wikipedia.org/wiki/Discrete\\_Fourier\\_transform\\_\(general%29](https://en.wikipedia.org/wiki/Discrete_Fourier_transform_(general%29)
- [R188] [https://www.radioeng.cz/fulltexts/2002/02\\_03\\_40\\_42.pdf](https://www.radioeng.cz/fulltexts/2002/02_03_40_42.pdf)
- [R189] [https://en.wikipedia.org/wiki/Hadamard\\_transform](https://en.wikipedia.org/wiki/Hadamard_transform)
- [R190] <https://people.csail.mit.edu/rrw/presentations/subset-conv.pdf>
- [R191] <https://people.csail.mit.edu/rrw/presentations/subset-conv.pdf>
- [R192] <https://people.csail.mit.edu/rrw/presentations/subset-conv.pdf>
- [R37] <https://docs.scipy.org/doc/numpy/user/basics.types.html>
- [R38] [https://en.wikipedia.org/wiki/Newton%27s\\_method](https://en.wikipedia.org/wiki/Newton%27s_method)
- [R557] [https://en.wikipedia.org/wiki/Quine-McCluskey\\_algorithm](https://en.wikipedia.org/wiki/Quine-McCluskey_algorithm)
- [R558] [https://en.wikipedia.org/wiki/Quine-McCluskey\\_algorithm](https://en.wikipedia.org/wiki/Quine-McCluskey_algorithm)
- [R559] [https://en.wikipedia.org/wiki/Zhegalkin\\_polynomial](https://en.wikipedia.org/wiki/Zhegalkin_polynomial)
- [R560] [https://en.wikipedia.org/wiki/Canonical\\_normal\\_form#Indexing\\_maxterms](https://en.wikipedia.org/wiki/Canonical_normal_form#Indexing_maxterms)
- [R561] [https://en.wikipedia.org/wiki/Canonical\\_normal\\_form#Indexing\\_minterms](https://en.wikipedia.org/wiki/Canonical_normal_form#Indexing_minterms)
- [R744] [https://en.wikipedia.org/wiki/Disjoint\\_sets](https://en.wikipedia.org/wiki/Disjoint_sets)
- [R745] [https://en.wikipedia.org/wiki/Power\\_set](https://en.wikipedia.org/wiki/Power_set)
- [R746] [https://en.wikipedia.org/wiki/Symmetric\\_difference](https://en.wikipedia.org/wiki/Symmetric_difference)
- [R747] [https://en.wikipedia.org/wiki/Interval\\_%28mathematics%29](https://en.wikipedia.org/wiki/Interval_%28mathematics%29)
- [R748] [https://en.wikipedia.org/wiki/Finite\\_set](https://en.wikipedia.org/wiki/Finite_set)
- [R749] [https://en.wikipedia.org/wiki/Union\\_%28set\\_theory%29](https://en.wikipedia.org/wiki/Union_%28set_theory%29)
- [R750] [https://en.wikipedia.org/wiki/Intersection\\_%28set\\_theory%29](https://en.wikipedia.org/wiki/Intersection_%28set_theory%29)
- [R751] [https://en.wikipedia.org/wiki/Cartesian\\_product](https://en.wikipedia.org/wiki/Cartesian_product)
- [R752] <http://mathworld.wolfram.com/ComplementSet.html>
- [R753] [https://en.wikipedia.org/wiki/Symmetric\\_difference](https://en.wikipedia.org/wiki/Symmetric_difference)
- [R754] [https://en.wikipedia.org/wiki/Empty\\_set](https://en.wikipedia.org/wiki/Empty_set)
- [R755] [https://en.wikipedia.org/wiki/Universal\\_set](https://en.wikipedia.org/wiki/Universal_set)
- [R756] [https://en.wikipedia.org/wiki/Power\\_set](https://en.wikipedia.org/wiki/Power_set)
- [R757] [https://en.wikipedia.org/wiki/Axiom\\_of\\_power\\_set](https://en.wikipedia.org/wiki/Axiom_of_power_set)
- [R758] [https://en.wikipedia.org/wiki/Element\\_%28mathematics%29](https://en.wikipedia.org/wiki/Element_%28mathematics%29)

- [R566] Prof. Frank Ben's notes: <https://math.berkeley.edu/~bernd/ban275.pdf>
- [R567] Wikipedia article on Permanent: [https://en.wikipedia.org/wiki/Permanent\\_\(mathematics\)](https://en.wikipedia.org/wiki/Permanent_(mathematics))
- [R568] <https://reference.wolfram.com/language/ref/Permanent.html>
- [R569] Permanent of a rectangular matrix : <https://arxiv.org/pdf/0904.3251.pdf>
- [R570] [https://en.wikipedia.org/wiki/Gram%E2%80%93Schmidt\\_process](https://en.wikipedia.org/wiki/Gram%E2%80%93Schmidt_process)
- [R571] Algorithm 5.4.2, Matrix computations by Golub and Van Loan, 4th edition
- [R572] Complex Matrix Bidiagonalization, <https://github.com/vslobody/Householder-Bidiagonalization>
- [R573] Algorithm 5.4.2, Matrix computations by Golub and Van Loan, 4th edition
- [R574] Complex Matrix Bidiagonalization : <https://github.com/vslobody/Householder-Bidiagonalization>
- [R575] [https://en.wikipedia.org/wiki/Definiteness\\_of\\_a\\_matrix#Eigenvalues](https://en.wikipedia.org/wiki/Definiteness_of_a_matrix#Eigenvalues)
- [R576] <http://mathworld.wolfram.com/PositiveDefiniteMatrix.html>
- [R577] Johnson, C. R. "Positive Definite Matrices." Amer. Math. Monthly 77, 259-264 1970.
- [R578] [https://en.wikipedia.org/wiki/Definiteness\\_of\\_a\\_matrix#Eigenvalues](https://en.wikipedia.org/wiki/Definiteness_of_a_matrix#Eigenvalues)
- [R579] <http://mathworld.wolfram.com/PositiveDefiniteMatrix.html>
- [R580] Johnson, C. R. "Positive Definite Matrices." Amer. Math. Monthly 77, 259-264 1970.
- [R581] [https://en.wikipedia.org/wiki/Definiteness\\_of\\_a\\_matrix#Eigenvalues](https://en.wikipedia.org/wiki/Definiteness_of_a_matrix#Eigenvalues)
- [R582] <http://mathworld.wolfram.com/PositiveDefiniteMatrix.html>
- [R583] Johnson, C. R. "Positive Definite Matrices." Amer. Math. Monthly 77, 259-264 1970.
- [R584] [https://en.wikipedia.org/wiki/Definiteness\\_of\\_a\\_matrix#Eigenvalues](https://en.wikipedia.org/wiki/Definiteness_of_a_matrix#Eigenvalues)
- [R585] <http://mathworld.wolfram.com/PositiveDefiniteMatrix.html>
- [R586] Johnson, C. R. "Positive Definite Matrices." Amer. Math. Monthly 77, 259-264 1970.
- [R587] [https://en.wikipedia.org/wiki/Definiteness\\_of\\_a\\_matrix#Eigenvalues](https://en.wikipedia.org/wiki/Definiteness_of_a_matrix#Eigenvalues)
- [R588] <http://mathworld.wolfram.com/PositiveDefiniteMatrix.html>
- [R589] Johnson, C. R. "Positive Definite Matrices." Amer. Math. Monthly 77, 259-264 1970.
- [R590] W. Zhou & D.J. Jeffrey, "Fraction-free matrix factors: new forms for LU and QR factors". Frontiers in Computer Science in China, Vol 2, no. 1, pp. 67-80, 2008.
- [R591] [https://en.wikipedia.org/wiki/Gaussian\\_elimination](https://en.wikipedia.org/wiki/Gaussian_elimination)
- [R592] [https://en.wikipedia.org/wiki/Moore-Penrose\\_pseudoinverse](https://en.wikipedia.org/wiki/Moore-Penrose_pseudoinverse)
- [R593] [https://en.wikipedia.org/wiki/Moore-Penrose\\_pseudoinverse#Obtaining\\_all\\_solutions\\_of\\_a\\_linear\\_system](https://en.wikipedia.org/wiki/Moore-Penrose_pseudoinverse#Obtaining_all_solutions_of_a_linear_system)
- [R594] [https://en.wikipedia.org/wiki/Rank\\_factorization](https://en.wikipedia.org/wiki/Rank_factorization)



- [R595] Piziak, R.; Odell, P. L. (1 June 1999). "Full Rank Factorization of Matrices". *Mathematics Magazine*. 72 (3): 193. doi:10.2307/2690882
- [R596] [https://en.wikipedia.org/wiki/Hessian\\_matrix](https://en.wikipedia.org/wiki/Hessian_matrix)
- [R597] [https://en.wikipedia.org/wiki/Gram%E2%80%93Schmidt\\_process](https://en.wikipedia.org/wiki/Gram%E2%80%93Schmidt_process)
- [R562] [https://en.wikipedia.org/wiki/Jordan\\_matrix](https://en.wikipedia.org/wiki/Jordan_matrix)
- [R563] <https://blogs.mathworks.com/cleve/2013/04/15/wilkinsons-matrices-2/>
- [R564] J. H. Wilkinson, *The Algebraic Eigenvalue Problem*, Claredon Press, Oxford, 1965, 662 pp.
- [R565] Wikipedia Article on Schur Component : [https://en.wikipedia.org/wiki/Schur\\_complement](https://en.wikipedia.org/wiki/Schur_complement)
- [R598] Cohen, H. *A Course in Computational Algebraic Number Theory*. (See Algorithms 2.4.5 and 2.4.8.)
- [WikiDel] <https://en.wikipedia.org/wiki/Del>
- [R978] [https://en.wikipedia.org/wiki/Dyadic\\_tensor](https://en.wikipedia.org/wiki/Dyadic_tensor)
- [R979] Kane, T., Levinson, D. *Dynamics Theory and Applications*. 1985 McGraw-Hill
- [Dyadics] <https://en.wikipedia.org/wiki/Dyadics>
- [DyadicProducts] [https://en.wikipedia.org/wiki/Dyadic\\_product](https://en.wikipedia.org/wiki/Dyadic_product)
- [DelOperator] <https://en.wikipedia.org/wiki/Del>
- [R599] [https://en.wikipedia.org/wiki/Prime\\_number\\_theorem#Table\\_of\\_.CF.80.28x.29.2C\\_x.2F\\_log\\_x.2C\\_and\\_li.28x.29](https://en.wikipedia.org/wiki/Prime_number_theorem#Table_of_.CF.80.28x.29.2C_x.2F_log_x.2C_and_li.28x.29)
- [R600] [https://en.wikipedia.org/wiki/Prime\\_number\\_theorem#Approximations\\_for\\_the\\_nth\\_prime\\_number](https://en.wikipedia.org/wiki/Prime_number_theorem#Approximations_for_the_nth_prime_number)
- [R601] [https://en.wikipedia.org/wiki/Skewes%27\\_number](https://en.wikipedia.org/wiki/Skewes%27_number)
- [R602] [https://en.wikipedia.org/wiki/Prime\\_number](https://en.wikipedia.org/wiki/Prime_number)
- [R603] <http://primes.utm.edu/notes/gaps.html>
- [R604] [https://en.wikipedia.org/wiki/Bertrand's\\_postulate](https://en.wikipedia.org/wiki/Bertrand's_postulate)
- [R605] Richard Crandall & Carl Pomerance (2005), "Prime Numbers: A Computational Perspective", Springer, 2nd edition, 229-231
- [R606] Richard Crandall & Carl Pomerance (2005), "Prime Numbers: A Computational Perspective", Springer, 2nd edition, 236-238
- [R607] <http://modular.math.washington.edu/edu/2007/spring/ent/ent-html/node81.html>
- [R608] <https://www.cs.toronto.edu/~yuvalf/Factorization.pdf>
- [R609] [https://en.wikipedia.org/wiki/Unitary\\_divisor](https://en.wikipedia.org/wiki/Unitary_divisor)
- [R610] <http://mathworld.wolfram.com/UnitaryDivisor.html>
- [R611] <http://mathworld.wolfram.com/UnitaryDivisorFunction.html>
- [R612] definition is described in <https://oeis.org/A066272/a066272a.html>
- [R613] formula from <https://oeis.org/A066272>
- [R614] [https://en.wikipedia.org/wiki/Euler%27s\\_totient\\_function](https://en.wikipedia.org/wiki/Euler%27s_totient_function)
- [R615] <http://mathworld.wolfram.com/TotientFunction.html>

- [R616] [https://en.wikipedia.org/wiki/Carmichael\\_function](https://en.wikipedia.org/wiki/Carmichael_function)
- [R617] <http://mathworld.wolfram.com/CarmichaelFunction.html>
- [R618] [https://en.wikipedia.org/wiki/Divisor\\_function](https://en.wikipedia.org/wiki/Divisor_function)
- [R619] <http://mathworld.wolfram.com/UnitaryDivisorFunction.html>
- [R620] [https://en.wikipedia.org/wiki/Square-free\\_integer#Squarefree\\_core](https://en.wikipedia.org/wiki/Square-free_integer#Squarefree_core)
- [R621] <http://mathworld.wolfram.com/PrimeFactor.html>
- [R622] <http://mathworld.wolfram.com/PrimeFactor.html>
- [R623] <http://mathworld.wolfram.com/PerfectNumber.html>
- [R624] [https://en.wikipedia.org/wiki/Perfect\\_number](https://en.wikipedia.org/wiki/Perfect_number)
- [R625] <http://mathworld.wolfram.com/MersennePrime.html>
- [R626] <http://mathworld.wolfram.com/AbundantNumber.html>
- [R627] <http://mathworld.wolfram.com/DeficientNumber.html>
- [R628] [https://en.wikipedia.org/wiki/Amicable\\_numbers](https://en.wikipedia.org/wiki/Amicable_numbers)
- [R629] <http://mathworld.wolfram.com/PartitionFunctionP.html>
- [R630] [https://en.wikipedia.org/wiki/Euler\\_pseudoprime](https://en.wikipedia.org/wiki/Euler_pseudoprime)
- [R631] <http://mersenneforum.org/showpost.php?p=110896>
- [R632] Richard Crandall & Carl Pomerance (2005), "Prime Numbers: A Computational Perspective", Springer, 2nd edition, 135-138
- [R633] [https://oeis.org/wiki/Gaussian\\_primes](https://oeis.org/wiki/Gaussian_primes)
- [R634] W. Stein "Elementary Number Theory" (2011), page 44
- [R635] P. Hackman "Elementary Number Theory" (2009), Chapter C
- [R636] P. Hackman "Elementary Number Theory" (2009), page 76
- [R637] <http://mathworld.wolfram.com/DiscreteLogarithm.html>
- [R638] "Handbook of applied cryptography", Menezes, A. J., Van, O. P. C., & Vanstone, S. A. (1997).
- [R639] [https://en.wikipedia.org/wiki/Continued\\_fraction](https://en.wikipedia.org/wiki/Continued_fraction)
- [R640] [https://en.wikipedia.org/wiki/Periodic\\_continued\\_fraction](https://en.wikipedia.org/wiki/Periodic_continued_fraction)
- [R641] K. Rosen. Elementary Number theory and its applications. Addison-Wesley, 3 Sub edition, pages 379-381, January 1992.
- [R642] [https://en.wikipedia.org/wiki/M%C3%B6bius\\_function](https://en.wikipedia.org/wiki/M%C3%B6bius_function)
- [R643] Thomas Koshy "Elementary Number Theory with Applications"
- [R644] [https://en.wikipedia.org/wiki/Egyptian\\_fraction](https://en.wikipedia.org/wiki/Egyptian_fraction)
- [R645] [https://en.wikipedia.org/wiki/Greedy\\_algorithm\\_for\\_Egyptian\\_fractions](https://en.wikipedia.org/wiki/Greedy_algorithm_for_Egyptian_fractions)
- [R646] <https://www.ics.uci.edu/~eppstein/numth/egypt/conflict.html>
- [R647] [http://ami.ektf.hu/uploads/papers/finalpdf/AMI\\_42\\_from129to134.pdf](http://ami.ektf.hu/uploads/papers/finalpdf/AMI_42_from129to134.pdf)
- [R648] <http://www.numberworld.org/digits/Pi/>

- [R649] <https://pdfs.semanticscholar.org/5c52/8a975c1405bd35c65993abf5a4edb667c1db.pdf>
- [R650] [https://www.rieselprime.de/ziki/Self-initializing\\_quadratic\\_sieve](https://www.rieselprime.de/ziki/Self-initializing_quadratic_sieve)
- [R658] [https://en.wikipedia.org/wiki/Gamma\\_matrices](https://en.wikipedia.org/wiki/Gamma_matrices)
- [R659] [https://en.wikipedia.org/wiki/Pauli\\_matrices](https://en.wikipedia.org/wiki/Pauli_matrices)
- [R668] [https://en.wikipedia.org/wiki/Pauli\\_matrices](https://en.wikipedia.org/wiki/Pauli_matrices)
- [R691] [https://en.wikipedia.org/wiki/Kronecker\\_delta](https://en.wikipedia.org/wiki/Kronecker_delta)
- [Regge58] ‘Symmetry Properties of Clebsch-Gordan Coefficients’, T. Regge, Nuovo Cimento, Volume 10, pp. 544 (1958)
- [Regge59] ‘Symmetry Properties of Racah Coefficients’, T. Regge, Nuovo Cimento, Volume 11, pp. 116 (1959)
- [Edmonds74] A. R. Edmonds. Angular momentum in quantum mechanics. Investigations in physics, 4.; Investigations in physics, no. 4. Princeton, N.J., Princeton University Press, 1957.
- [Rasch03] J. Rasch and A. C. H. Yu, ‘Efficient Storage Scheme for Pre-calculated Wigner 3j, 6j and Gaunt Coefficients’, SIAM J. Sci. Comput. Volume 25, Issue 4, pp. 1416-1428 (2003)
- [Liberatodebrito82] ‘FORTRAN program for the integral of three spherical harmonics’, A. Liberato de Brito, Comput. Phys. Commun., Volume 25, pp. 81-85 (1982)
- [Page52] C. H. Page, [Classes of units in the SI](#), Am. J. of Phys. 20, 1 (1952): 1.
- [Page78] C. H. Page, [Units and Dimensions in Physics](#), Am. J. of Phys. 46, 1 (1978): 78.
- [deBoer79] J. de Boer, [Group properties of quantities and units](#), Am. J. of Phys. 47, 9 (1979): 818.
- [LevyLeblond77] J.-M. Lévy-Leblond, [On the Conceptual Nature of the Physical Constants](#), La Rivista Del Nuovo Cimento 7, no. 2 (1977): 187-214.
- [NIST] [NIST reference on constants, units and uncertainties](#).
- [WikiDyadics] “Dyadics.” Wikipedia, the Free Encyclopedia. Web. 05 Aug. 2011. <<https://en.wikipedia.org/wiki/Dyadics>>.
- [WikiDyadicProducts] “Dyadic Product.” Wikipedia, the Free Encyclopedia. Web. 05 Aug. 2011. <[https://en.wikipedia.org/wiki/Dyadic\\_product](https://en.wikipedia.org/wiki/Dyadic_product)>.
- [Likins1973] Likins, Peter W. Elements of Engineering Mechanics. McGraw-Hill, Inc. 1973. Print.
- [Blajer1994] Blajer, Wojciech, Werner Schiehlen, and Walter Schirm. “A projective criterion to the coordinate partitioning method for multibody dynamics.” Archive of Applied Mechanics 64: 86-98. Print.
- [Kane1983] Kane, Thomas R., Peter W. Likins, and David A. Levinson. Spacecraft Dynamics. New York: McGraw-Hill Book, 1983. Print.
- [Kane1985] Kane, Thomas R., and David A. Levinson. Dynamics, Theory and Applications. New York: McGraw-Hill, 1985. Print.
- [Meijaard2007] Meijaard, J.P., Jim M. Papadopoulos, Andy Ruina, and A.L. Schwab. “Linearized Dynamics Equations for the Balance and Steer of a Bicycle: a Benchmark and Review.” Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences 463.2084 (2007): 1955-982. Print.

- [R669] <https://en.wikipedia.org/wiki/Commutator>
- [R670] Varshalovich, D A, Quantum Theory of Angular Momentum. 1988.
- [R671] [Clebsch-Gordan Coefficients, Spherical Harmonics, and d Functions](#) in P.A. Zyla *et al.* (Particle Data Group), Prog. Theor. Exp. Phys. 2020, 083C01 (2020).
- [R672] Varshalovich, D A, Quantum Theory of Angular Momentum. 1988.
- [R673] Varshalovich, D A, Quantum Theory of Angular Momentum. 1988.
- [R674] <https://en.wikipedia.org/wiki/Commutator>
- [R675] [https://en.wikipedia.org/wiki/Planck\\_constant](https://en.wikipedia.org/wiki/Planck_constant)
- [R676] [https://en.wikipedia.org/wiki/Hermitian\\_adjoint](https://en.wikipedia.org/wiki/Hermitian_adjoint)
- [R677] [https://en.wikipedia.org/wiki/Hermitian\\_transpose](https://en.wikipedia.org/wiki/Hermitian_transpose)
- [R683] [https://en.wikipedia.org/wiki/Inner\\_product](https://en.wikipedia.org/wiki/Inner_product)
- [R678] [https://en.wikipedia.org/wiki/Hilbert\\_space#Direct\\_sums](https://en.wikipedia.org/wiki/Hilbert_space#Direct_sums)
- [R679] [https://en.wikipedia.org/wiki/Fock\\_space](https://en.wikipedia.org/wiki/Fock_space)
- [R680] [https://en.wikipedia.org/wiki/Hilbert\\_space](https://en.wikipedia.org/wiki/Hilbert_space)
- [R681] [https://en.wikipedia.org/wiki/Hilbert\\_space#Tensor\\_products](https://en.wikipedia.org/wiki/Hilbert_space#Tensor_products)
- [R682] [https://en.wikipedia.org/wiki/Hilbert\\_space#Tensor\\_products](https://en.wikipedia.org/wiki/Hilbert_space#Tensor_products)
- [R684] [https://en.wikipedia.org/wiki/Operator\\_%28physics%29](https://en.wikipedia.org/wiki/Operator_%28physics%29)
- [R685] <https://en.wikipedia.org/wiki/Observable>
- [R686] [https://en.wikipedia.org/wiki/Outer\\_product](https://en.wikipedia.org/wiki/Outer_product)
- [R687] Varshalovich, D A, Quantum Theory of Angular Momentum. 1988.
- [R688] Varshalovich, D A, Quantum Theory of Angular Momentum. 1988.
- [R689] [https://en.wikipedia.org/wiki/Bra-ket\\_notation](https://en.wikipedia.org/wiki/Bra-ket_notation)
- [R690] [https://en.wikipedia.org/wiki/Bra-ket\\_notation](https://en.wikipedia.org/wiki/Bra-ket_notation)
- [R660] [https://en.wikipedia.org/wiki/Complex\\_beam\\_parameter](https://en.wikipedia.org/wiki/Complex_beam_parameter)
- [R661] [https://en.wikipedia.org/wiki/Gaussian\\_beam](https://en.wikipedia.org/wiki/Gaussian_beam)
- [R662] [https://en.wikipedia.org/wiki/Ray\\_transfer\\_matrix\\_analysis](https://en.wikipedia.org/wiki/Ray_transfer_matrix_analysis)
- [R663] [https://en.wikipedia.org/wiki/Optical\\_medium](https://en.wikipedia.org/wiki/Optical_medium)
- [R664] [https://en.wikipedia.org/wiki/Jones\\_calculus](https://en.wikipedia.org/wiki/Jones_calculus)
- [R665] [https://en.wikipedia.org/wiki/Mueller\\_calculus](https://en.wikipedia.org/wiki/Mueller_calculus)
- [R666] [https://en.wikipedia.org/wiki/Stokes\\_parameters](https://en.wikipedia.org/wiki/Stokes_parameters)
- [R667] [https://en.wikipedia.org/wiki/Fresnel\\_equations](https://en.wikipedia.org/wiki/Fresnel_equations)
- [R656] [https://en.wikipedia.org/wiki/Transfer\\_function](https://en.wikipedia.org/wiki/Transfer_function)
- [R657] [https://en.wikipedia.org/wiki/Laplace\\_transform](https://en.wikipedia.org/wiki/Laplace_transform)
- [R652] [https://en.wikipedia.org/wiki/Pole%E2%80%93zero\\_plot](https://en.wikipedia.org/wiki/Pole%E2%80%93zero_plot)
- [R653] <https://www.mathworks.com/help/control/ref/lti.impulse.html>
- [R654] <https://www.mathworks.com/help/control/ref/lti.step.html>

- [R655] [https://en.wikipedia.org/wiki/Ramp\\_function](https://en.wikipedia.org/wiki/Ramp_function)
- [R651] <http://homes.civil.aau.dk/jc/FemteSemester/Beams3D.pdf>
- [R957] <http://docs.scipy.org/doc/numpy/reference/ufuncs.html>
- [AOCP] Algorithm 7.1.2.5M in Volume 4A, Combinatorial Algorithms, Part 1, of The Art of Computer Programming, by Donald Knuth.
- [Factorisatio] On a Problem of Oppenheim concerning “Factorisatio Numerorum” E. R. Canfield, Paul Erdos, Carl Pomerance, JOURNAL OF NUMBER THEORY, Vol. 17, No. 1. August 1983. See section 7 for a description of an algorithm similar to Knuth’s.
- [Yorgey] Generating Multiset Partitions, Brent Yorgey, The Monad.Reader, Issue 8, September 2007.
- [R958] TAOCP 4, section 7.2.1.5, problem 64
- [R959] [https://en.wikipedia.org/wiki/Connected\\_component\\_\(graph\\_theory\)](https://en.wikipedia.org/wiki/Connected_component_(graph_theory))
- [R960] [https://en.wikipedia.org/wiki/Tarjan%27s\\_strongly\\_connected\\_components\\_algorithm](https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm)
- [R961] [https://en.wikipedia.org/wiki/Method\\_ringing](https://en.wikipedia.org/wiki/Method_ringing)
- [R962] <https://stackoverflow.com/questions/4856615/recursive-permutation/4857018>
- [R963] <http://programminggeeks.com/bell-algorithm-for-permutation/>
- [R964] [https://en.wikipedia.org/wiki/Steinhaus%E2%80%93Johnson%E2%80%93Trotter\\_algorithm](https://en.wikipedia.org/wiki/Steinhaus%E2%80%93Johnson%E2%80%93Trotter_algorithm)
- [R965] Generating involutions, derangements, and relatives by ECO Vincent Vajnovszki, DMTCS vol 1 issue 12, 2010
- [R966] <http://mathworld.wolfram.com/PermutationInvolution.html>
- [R967] T. Beyer and S.M. Hedetniemi: constant time generation of rooted trees, SIAM J. Computing Vol. 9, No. 4, November 1980
- [R968] <https://stackoverflow.com/questions/1633833/oriented-forest-taocp-algorithm-in-python>
- [R969] [https://en.wikipedia.org/wiki/Lexicographically\\_minimal\\_string\\_rotation](https://en.wikipedia.org/wiki/Lexicographically_minimal_string_rotation)
- [R970] <http://mathworld.wolfram.com/Necklace.html>
- [R971] Generating Integer Partitions, [online], Available: <https://jeromekelleher.net/generating-integer-partitions.html>
- [R972] Jerome Kelleher and Barry O’Sullivan, “Generating All Partitions: A Comparison Of Two Encodings”, [online], Available: <https://arxiv.org/pdf/0909.2331v2.pdf>
- [R973] modified from Tim Peter’s version to allow for k and m values: <http://code.activestate.com/recipes/218332-generator-for-integer-partitions/>
- [R974] [https://en.wikipedia.org/wiki/Strongly\\_connected\\_component](https://en.wikipedia.org/wiki/Strongly_connected_component)
- [R975] [https://en.wikipedia.org/wiki/Tarjan%27s\\_strongly\\_connected\\_components\\_algorithm](https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm)
- [R976] [https://en.wikipedia.org/wiki/Topological\\_sorting](https://en.wikipedia.org/wiki/Topological_sorting)
- [R977] <https://stackoverflow.com/questions/6116978/python-replace-multiple-strings>
- [R553] [https://en.wikibooks.org/wiki/LaTeX/Colors#The\\_68\\_standard\\_colors\\_known\\_to\\_dvips](https://en.wikibooks.org/wiki/LaTeX/Colors#The_68_standard_colors_known_to_dvips)

- [R554] <https://ipython.readthedocs.io/en/stable/config/details.html#terminal-colors>
- [WikiPappus] “Pappus’s Hexagon Theorem” Wikipedia, the Free Encyclopedia. Web. 26 Apr. 2013. <[https://en.wikipedia.org/wiki/Pappus's\\_hexagon\\_theorem](https://en.wikipedia.org/wiki/Pappus's_hexagon_theorem)>
- [R521] [https://en.wikipedia.org/wiki/Graham\\_scan](https://en.wikipedia.org/wiki/Graham_scan)
- [R522] Andrew’s Monotone Chain Algorithm (A.M. Andrew, “Another Efficient Algorithm for Convex Hulls in Two Dimensions”, 1979) <http://geomalgorithms.com/a10-hull-1.html>
- [R503] [https://en.wikipedia.org/wiki/Director\\_circle](https://en.wikipedia.org/wiki/Director_circle)
- [R504] <https://math.stackexchange.com/questions/108270/what-is-the-equation-of-an-ellipse-that-is-n>
- [R505] [https://en.wikipedia.org/wiki/Ellipse#Equation\\_of\\_a\\_shifted\\_ellipse](https://en.wikipedia.org/wiki/Ellipse#Equation_of_a_shifted_ellipse)
- [R506] [https://en.wikipedia.org/wiki/Polar\\_moment\\_of\\_inertia](https://en.wikipedia.org/wiki/Polar_moment_of_inertia)
- [R507] [https://en.wikipedia.org/wiki/List\\_of\\_second\\_moments\\_of\\_area](https://en.wikipedia.org/wiki/List_of_second_moments_of_area)
- [R508] [https://en.wikipedia.org/wiki/Section\\_modulus](https://en.wikipedia.org/wiki/Section_modulus)
- [R509] <http://mathworld.wolfram.com/SemilatusRectum.html>
- [R510] [https://en.wikipedia.org/wiki/Ellipse#Semi-latus\\_rectum](https://en.wikipedia.org/wiki/Ellipse#Semi-latus_rectum)
- [R511] <https://github.com/sympy/sympy/wiki/A-method-to-return-a-cut-section-of-any-polygon-geom>
- [R512] <http://paulbourke.net/geometry/polygonmesh/#insidepoly>
- [R513] <https://skyciv.com/docs/tutorials/section-tutorials/calculating-the-statical-or-first-moment-of-a?cc=BMD>
- [R514] <https://mechanicalc.com/reference/cross-sections>
- [R515] [https://en.wikipedia.org/wiki/Polar\\_moment\\_of\\_inertia](https://en.wikipedia.org/wiki/Polar_moment_of_inertia)
- [R516] [https://en.wikipedia.org/wiki/Second\\_moment\\_of\\_area](https://en.wikipedia.org/wiki/Second_moment_of_area)
- [R517] [https://en.wikipedia.org/wiki/Section\\_modulus](https://en.wikipedia.org/wiki/Section_modulus)
- [R518] <http://mathworld.wolfram.com/Excircles.html>
- [R519] <http://mathworld.wolfram.com/Exradius.html>
- [R520] <http://mathworld.wolfram.com/Excircles.html>
- [R523] <https://hal.inria.fr/inria-00070025/document>
- [R524] [http://www.risc.jku.at/publications/download/risc\\_2244/DIPLFORM.pdf](http://www.risc.jku.at/publications/download/risc_2244/DIPLFORM.pdf)
- [R555] [https://en.wikipedia.org/wiki/Root\\_system](https://en.wikipedia.org/wiki/Root_system)
- [R556] Lie Algebras and Representation Theory - Humphreys
- [Wester1999] Michael J. Wester, A Critique of the Mathematical Abilities of CA Systems, 1999, <http://www.math.unm.edu/~wester/cas/book/Wester.pdf>
- [R719] J.C. Faugere, P. Gianni, D. Lazard, T. Mora (1994). Efficient Computation of Zero-dimensional Groebner Bases by Change of Ordering
- [R720] [https://en.wikipedia.org/wiki/Cubic\\_function#Trigonometric\\_28and\\_hyperbolic\\_29\\_method](https://en.wikipedia.org/wiki/Cubic_function#Trigonometric_28and_hyperbolic_29_method)
- [R721] [Bronstein93]
- [R722] [ManWright94]

- [R723]     [\[Koepf98\]](#)
- [R724]     [\[Abramov71\]](#)
- [R725]     [\[Man93\]](#)
- [R726]     [\[ManWright94\]](#)
- [R727]     [\[Koepf98\]](#)
- [R728]     [\[Abramov71\]](#)
- [R729]     [\[Man93\]](#)
- [R695]     [\[Kozen89\]](#)
- [R696]     [\[Monagan93\]](#)
- [R697]     [\[Gathen99\]](#)
- [R698]     [\[Gathen99\]](#)
- [R699]     [\[Gathen99\]](#)
- [R700]     [\[Gathen92\]](#)
- [R701]     [\[Geddes92\]](#)
- [R702]     [\[Gathen99\]](#)
- [R703]     ‘An introduction to the Theory of Numbers’ 5th Edition by Ivan Niven, Zuckerman and Montgomery.
- [R704]     [\[Liao95\]](#)
- [R705]     [\[Gathen99\]](#)
- [R706]     [\[Gathen99\]](#)
- [R707]     [\[Weisstein09\]](#)
- [R708]     [\[Gathen99\]](#)
- [R709]     [\[Wang78\]](#)
- [R710]     [\[Geddes92\]](#)
- [R711]     [\[Gathen99\]](#)
- [R712]     J.C. Faugere, P. Gianni, D. Lazard, T. Mora (1994). Efficient Computation of Zero-dimensional Groebner Bases by Change of Ordering
- [R730]     A. Bostan, P. Flajolet, B. Salvy and E. Schost “Fast Computation with Two Algebraic Numbers”, (2002) Research Report 4579, Institut National de Recherche en Informatique et en Automatique
- [Kozen89]   D. Kozen, S. Landau, Polynomial decomposition algorithms, Journal of Symbolic Computation 7 (1989), pp. 445-456
- [Liao95]    Hsin-Chao Liao, R. Fateman, Evaluation of the heuristic polynomial GCD, International Symposium on Symbolic and Algebraic Computation (ISSAC), ACM Press, Montreal, Quebec, Canada, 1995, pp. 240-247
- [Gathen99]   J. von zur Gathen, J. Gerhard, Modern Computer Algebra, First Edition, Cambridge University Press, 1999
- [Weisstein09] Eric W. Weisstein, Cyclotomic Polynomial, From MathWorld - A Wolfram Web Resource, <http://mathworld.wolfram.com/CyclotomicPolynomial.html>



- [Wang78] P. S. Wang, An Improved Multivariate Polynomial Factoring Algorithm, *Math. of Computation* 32, 1978, pp. 1215-1231
- [Geddes92] K. Geddes, S. R. Czapor, G. Labahn, *Algorithms for Computer Algebra*, Springer, 1992
- [Monagan93] Michael Monagan, In-place Arithmetic for Polynomials over  $\mathbb{Z}_n$ , *Proceedings of DISCO '92*, Springer-Verlag LNCS, 721, 1993, pp. 22-34
- [Kaltofen98] E. Kaltofen, V. Shoup, Subquadratic-time Factoring of Polynomials over Finite Fields, *Mathematics of Computation*, Volume 67, Issue 223, 1998, pp. 1179-1197
- [Shoup95] V. Shoup, A New Polynomial Factorization Algorithm and its Implementation, *Journal of Symbolic Computation*, Volume 20, Issue 4, 1995, pp. 363-397
- [Gathen92] J. von zur Gathen, V. Shoup, Computing Frobenius Maps and Factoring Polynomials, *ACM Symposium on Theory of Computing*, 1992, pp. 187-224
- [Shoup91] V. Shoup, A Fast Deterministic Algorithm for Factoring Polynomials over Finite Fields of Small Characteristic, In *Proceedings of International Symposium on Symbolic and Algebraic Computation*, 1991, pp. 14-21
- [Cox97] D. Cox, J. Little, D. O'Shea, *Ideals, Varieties and Algorithms*, Springer, Second Edition, 1997
- [Ajwa95] I.A. Ajwa, Z. Liu, P.S. Wang, Groebner Bases Algorithm, <https://citeseer.ist.psu.edu/myciteseer/login>, 1995
- [Bose03] N.K. Bose, B. Buchberger, J.P. Guiver, *Multidimensional Systems Theory and Applications*, Springer, 2003
- [Giovini91] A. Giovini, T. Mora, "One sugar cube, please" or Selection strategies in Buchberger algorithm, *ISSAC '91*, ACM
- [Bronstein93] M. Bronstein, B. Salvy, Full partial fraction decomposition of rational functions, *Proceedings ISSAC '93*, ACM Press, Kiev, Ukraine, 1993, pp. 157-160
- [Buchberger01] B. Buchberger, Groebner Bases: A Short Introduction for Systems Theorists, In: R. Moreno-Diaz, B. Buchberger, J. L. Freire, *Proceedings of EUROCAST'01*, February, 2001
- [Davenport88] J.H. Davenport, Y. Siret, E. Tournier, *Computer Algebra Systems and Algorithms for Algebraic Computation*, Academic Press, London, 1988, pp. 124-128
- [Greuel2008] G.-M. Greuel, Gerhard Pfister, *A Singular Introduction to Commutative Algebra*, Springer, 2008
- [Atiyah69] M.F. Atiyah, I.G. MacDonald, *Introduction to Commutative Algebra*, Addison-Wesley, 1969
- [Collins67] G.E. Collins, Subresultants and Reduced Polynomial Remainder Sequences. *J. ACM* 14 (1967) 128-142
- [BrownTraub71] W.S. Brown, J.F. Traub, On Euclid's Algorithm and the Theory of Subresultants. *J. ACM* 18 (1971) 505-514
- [Brown78] W.S. Brown, The Subresultant PRS Algorithm. *ACM Transaction of Mathematical Software* 4 (1978) 237-249
- [Monagan00] M. Monagan and A. Wittkopf, On the Design and Implementation of Brown's Algorithm over the Integers and Number Fields, *Proceedings of ISSAC 2000*, pp. 225-233, ACM, 2000.



- [Brown71] W.S. Brown, On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors, J. ACM 18, 4, pp. 478-504, 1971.
- [Hoeij04] M. van Hoeij and M. Monagan, Algorithms for polynomial GCD computation over algebraic function fields, Proceedings of ISSAC 2004, pp. 297-304, ACM, 2004.
- [Wang81] P.S. Wang, A p-adic algorithm for univariate partial fractions, Proceedings of SYMSAC 1981, pp. 212-217, ACM, 1981.
- [Hoeij02] M. van Hoeij and M. Monagan, A modular GCD algorithm over number fields presented with multiple extensions, Proceedings of ISSAC 2002, pp. 109-116, ACM, 2002
- [ManWright94] Yiu-Kwong Man and Francis J. Wright, "Fast Polynomial Dispersion Computation and its Application to Indefinite Summation", Proceedings of the International Symposium on Symbolic and Algebraic Computation, 1994, Pages 175-180 <http://dl.acm.org/citation.cfm?doid=190347.190413>
- [Koepf98] Wolfram Koepf, "Hypergeometric Summation: An Algorithmic Approach to Summation and Special Function Identities", Advanced lectures in mathematics, Vieweg, 1998
- [Abramov71] S. A. Abramov, "On the Summation of Rational Functions", USSR Computational Mathematics and Mathematical Physics, Volume 11, Issue 4, 1971, Pages 324-330
- [Man93] Yiu-Kwong Man, "On Computing Closed Forms for Indefinite Summations", Journal of Symbolic Computation, Volume 16, Issue 4, 1993, Pages 355-376 <http://www.sciencedirect.com/science/article/pii/S0747717183710539>
- [Kapur1994] Deepak Kapur, Tushar Saxena, and Lu Yang. "Algebraic and geometric reasoning using Dixon resultants", In Proceedings of the international symposium on Symbolic and algebraic computation (ISSAC '94), 1994, pages 99-107. [https://www.researchgate.net/publication/2514261\\_Algebraic\\_and\\_Geometric\\_Reasoning\\_using\\_Dixon\\_Resultants](https://www.researchgate.net/publication/2514261_Algebraic_and_Geometric_Reasoning_using_Dixon_Resultants)
- [Palancz08] B Paláncz, P Zaletnyik, JL Awange, EW Grafarend. "Dixon resultant's solution of systems of geodetic polynomial equations", Journal of Geodesy, 2008, Springer, [https://www.researchgate.net/publication/225607735\\_Dixon\\_resultant's\\_solution\\_of\\_systems\\_o](https://www.researchgate.net/publication/225607735_Dixon_resultant's_solution_of_systems_o)
- [Bruce97] Bruce Randall Donald, Deepak Kapur, and Joseph L. Mundy (Eds.). "Symbolic and Numerical Computation for Artificial Intelligence", Chapter 2, Academic Press, Inc., Orlando, FL, USA, 1997, <https://www2.cs.duke.edu/donaldlab/Books/SymbolicNumericalComputation/045-087.pdf>.
- [Stiller96] P Stiller. "An introduction to the theory of resultants", Mathematics and Computer Science, T&M University, 1996, Citeseer, <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.590.2021&rep=rep1&type=pdf>.
- [Cohen93] Henri Cohen. "A Course in Computational Algebraic Number Theory", Springer, 1993.
- [R694] Cohen, H. *A Course in Computational Algebraic Number Theory*. (See Algorithms 2.4.5 and 2.4.8.)
- [R713] Cohen, H. *A Course in Computational Algebraic Number Theory*.
- [R714] Cohen, H. *A Course in Computational Algebraic Number Theory*. (See Algorithm 6.2.9.)
- [R715] Cohen, H. *A Course in Computational Algebraic Number Theory*. (See Algorithm 4.8.17.)

- [Cohen00] Cohen, H. *Advanced Topics in Computational Number Theory*.
- [R716] Cohen, H. *A Course in Computational Algebraic Number Theory*. (See Prop. 5.1.3)
- [R717] Cohen, H. *A Course in Computational Algebraic Number Theory*.
- [R718] Cohen, H. *A Course in Computational Algebraic Number Theory* (See Sec. 2.3.2.)
- [JoyOfCats] J. Adamek, H. Herrlich. G. E. Strecker: Abstract and Concrete Categories. The Joy of Cats.
- [FiveLemma] [https://en.wikipedia.org/wiki/Five\\_lemma](https://en.wikipedia.org/wiki/Five_lemma)
- [Xypic] <http://xy-pic.sourceforge.net/>
- [R136] [https://en.wikipedia.org/wiki/Caesar\\_cipher](https://en.wikipedia.org/wiki/Caesar_cipher)
- [R137] <http://mathworld.wolfram.com/CaesarsMethod.html>
- [R138] <https://en.wikipedia.org/wiki/ROT13>
- [R139] [https://en.wikipedia.org/wiki/Affine\\_cipher](https://en.wikipedia.org/wiki/Affine_cipher)
- [R140] <https://en.wikipedia.org/wiki/Atbash>
- [R141] [https://en.wikipedia.org/wiki/Substitution\\_cipher](https://en.wikipedia.org/wiki/Substitution_cipher)
- [R142] [https://en.wikipedia.org/wiki/Vigenere\\_cipher](https://en.wikipedia.org/wiki/Vigenere_cipher)
- [R143] <http://web.archive.org/web/20071116100808/>
- [R144] <http://filebox.vt.edu/users/batman/kryptos.html> (short URL: <https://goo.gl/ijr22d>)
- [R145] [https://en.wikipedia.org/wiki/Hill\\_cipher](https://en.wikipedia.org/wiki/Hill_cipher)
- [R146] Lester S. Hill, Cryptography in an Algebraic Alphabet, The American Mathematical Monthly Vol.36, June-July 1929, pp.306-312.
- [R147] [https://en.wikipedia.org/wiki/Bifid\\_cipher](https://en.wikipedia.org/wiki/Bifid_cipher)
- [R148] [https://en.wikipedia.org/wiki/RSA\\_%28cryptosystem%29](https://en.wikipedia.org/wiki/RSA_%28cryptosystem%29)
- [R149] <http://cacr.uwaterloo.ca/techreports/2006/cacr2006-16.pdf>
- [R150] <https://link.springer.com/content/pdf/10.1007%2FBFb0055738.pdf>
- [R151] <http://www.itiis.org/digital-library/manuscript/1381>
- [R152] [https://en.wikipedia.org/wiki/RSA\\_%28cryptosystem%29](https://en.wikipedia.org/wiki/RSA_%28cryptosystem%29)
- [R153] <http://cacr.uwaterloo.ca/techreports/2006/cacr2006-16.pdf>
- [R154] <https://link.springer.com/content/pdf/10.1007%2FBFb0055738.pdf>
- [R155] <http://www.itiis.org/digital-library/manuscript/1381>
- [R156] [https://en.wikipedia.org/wiki/Morse\\_code](https://en.wikipedia.org/wiki/Morse_code)
- [R157] [https://en.wikipedia.org/wiki/Morse\\_code](https://en.wikipedia.org/wiki/Morse_code)
- [G157] Solomon Golomb, Shift register sequences, Aegean Park Press, Laguna Hills, Ca, 1967
- [M158] James L. Massey, "Shift-Register Synthesis and BCH Decoding." IEEE Trans. on Information Theory, vol. 15(1), pp. 122-127, Jan 1969.
- [R160] [https://en.wikipedia.org/wiki/Rail\\_fence\\_cipher](https://en.wikipedia.org/wiki/Rail_fence_cipher)
- [R161] <https://en.wikipedia.org/wiki/Manifold>

- [R162] G. Sussman, J. Wisdom, W. Farr, Functional Differential Geometry (2013)
- [R163] [https://en.wikipedia.org/wiki/Coordinate\\_system](https://en.wikipedia.org/wiki/Coordinate_system)
- [R692] Adaptive polygonal approximation of parametric curves, Luiz Henrique de Figueiredo.
- [R693] Adaptive polygonal approximation of parametric curves, Luiz Henrique de Figueiredo.
- [R803] [https://en.wikipedia.org/wiki/Discrete\\_uniform\\_distribution](https://en.wikipedia.org/wiki/Discrete_uniform_distribution)
- [R804] <http://mathworld.wolfram.com/DiscreteUniformDistribution.html>
- [R805] [https://en.wikipedia.org/wiki/Bernoulli\\_distribution](https://en.wikipedia.org/wiki/Bernoulli_distribution)
- [R806] <http://mathworld.wolfram.com/BernoulliDistribution.html>
- [R807] [https://en.wikipedia.org/wiki/Coin\\_flipping](https://en.wikipedia.org/wiki/Coin_flipping)
- [R808] [https://en.wikipedia.org/wiki/Binomial\\_distribution](https://en.wikipedia.org/wiki/Binomial_distribution)
- [R809] <http://mathworld.wolfram.com/BinomialDistribution.html>
- [R810] [https://en.wikipedia.org/wiki/Beta-binomial\\_distribution](https://en.wikipedia.org/wiki/Beta-binomial_distribution)
- [R811] <http://mathworld.wolfram.com/BetaBinomialDistribution.html>
- [R812] [https://en.wikipedia.org/wiki/Hypergeometric\\_distribution](https://en.wikipedia.org/wiki/Hypergeometric_distribution)
- [R813] <http://mathworld.wolfram.com/HypergeometricDistribution.html>
- [R814] [https://en.wikipedia.org/wiki/Rademacher\\_distribution](https://en.wikipedia.org/wiki/Rademacher_distribution)
- [R815] [https://en.wikipedia.org/wiki/Geometric\\_distribution](https://en.wikipedia.org/wiki/Geometric_distribution)
- [R816] <http://mathworld.wolfram.com/GeometricDistribution.html>
- [R817] [https://en.wikipedia.org/wiki/Hermite\\_distribution](https://en.wikipedia.org/wiki/Hermite_distribution)
- [R818] [https://en.wikipedia.org/wiki/Poisson\\_distribution](https://en.wikipedia.org/wiki/Poisson_distribution)
- [R819] <http://mathworld.wolfram.com/PoissonDistribution.html>
- [R820] [https://en.wikipedia.org/wiki/Logarithmic\\_distribution](https://en.wikipedia.org/wiki/Logarithmic_distribution)
- [R821] <http://mathworld.wolfram.com/LogarithmicDistribution.html>
- [R822] [https://en.wikipedia.org/wiki/Negative\\_binomial\\_distribution](https://en.wikipedia.org/wiki/Negative_binomial_distribution)
- [R823] <http://mathworld.wolfram.com/NegativeBinomialDistribution.html>
- [R824] [https://en.wikipedia.org/wiki/Skellam\\_distribution](https://en.wikipedia.org/wiki/Skellam_distribution)
- [R825] [https://en.wikipedia.org/wiki/Yule%E2%80%93Simon\\_distribution](https://en.wikipedia.org/wiki/Yule%E2%80%93Simon_distribution)
- [R826] [https://en.wikipedia.org/wiki/Zeta\\_distribution](https://en.wikipedia.org/wiki/Zeta_distribution)
- [R827] [https://en.wikipedia.org/wiki/Arcsine\\_distribution](https://en.wikipedia.org/wiki/Arcsine_distribution)
- [R828] [https://en.wikipedia.org/wiki/Benini\\_distribution](https://en.wikipedia.org/wiki/Benini_distribution)
- [R829] <http://reference.wolfram.com/legacy/v8/ref/BeniniDistribution.html>
- [R830] [https://en.wikipedia.org/wiki/Beta\\_distribution](https://en.wikipedia.org/wiki/Beta_distribution)
- [R831] <http://mathworld.wolfram.com/BetaDistribution.html>
- [R832] [https://en.wikipedia.org/wiki/Noncentral\\_beta\\_distribution](https://en.wikipedia.org/wiki/Noncentral_beta_distribution)
- [R833] [https://reference.wolfram.com/language/ref/NoncentralBetaDistribution.html](http://reference.wolfram.com/language/ref/NoncentralBetaDistribution.html)

- [R834] [https://en.wikipedia.org/wiki/Beta\\_prime\\_distribution](https://en.wikipedia.org/wiki/Beta_prime_distribution)
- [R835] <http://mathworld.wolfram.com/BetaPrimeDistribution.html>
- [R836] [https://en.wikipedia.org/wiki/Pareto\\_distribution#Bounded\\_Pareto\\_distribution](https://en.wikipedia.org/wiki/Pareto_distribution#Bounded_Pareto_distribution)
- [R837] [https://en.wikipedia.org/wiki/Cauchy\\_distribution](https://en.wikipedia.org/wiki/Cauchy_distribution)
- [R838] <http://mathworld.wolfram.com/CauchyDistribution.html>
- [R839] [https://en.wikipedia.org/wiki/Chi\\_distribution](https://en.wikipedia.org/wiki/Chi_distribution)
- [R840] <http://mathworld.wolfram.com/ChiDistribution.html>
- [R841] [https://en.wikipedia.org/wiki/Noncentral\\_chi\\_distribution](https://en.wikipedia.org/wiki/Noncentral_chi_distribution)
- [R842] [https://en.wikipedia.org/wiki/Chi\\_squared\\_distribution](https://en.wikipedia.org/wiki/Chi_squared_distribution)
- [R843] <http://mathworld.wolfram.com/Chi-SquaredDistribution.html>
- [R844] [https://en.wikipedia.org/wiki/Dagum\\_distribution](https://en.wikipedia.org/wiki/Dagum_distribution)
- [R845] [https://en.wikipedia.org/wiki/Erlang\\_distribution](https://en.wikipedia.org/wiki/Erlang_distribution)
- [R846] <http://mathworld.wolfram.com/ErlangDistribution.html>
- [R847] [https://en.wikipedia.org/wiki/Exponentially\\_modified\\_Gaussian\\_distribution](https://en.wikipedia.org/wiki/Exponentially_modified_Gaussian_distribution)
- [R848] [https://en.wikipedia.org/wiki/Exponential\\_distribution](https://en.wikipedia.org/wiki/Exponential_distribution)
- [R849] <http://mathworld.wolfram.com/ExponentialDistribution.html>
- [R850] <https://en.wikipedia.org/wiki/F-distribution>
- [R851] <http://mathworld.wolfram.com/F-Distribution.html>
- [R852] [https://en.wikipedia.org/wiki/Fisher%27s\\_z-distribution](https://en.wikipedia.org/wiki/Fisher%27s_z-distribution)
- [R853] <http://mathworld.wolfram.com/Fishersz-Distribution.html>
- [R854] [https://en.wikipedia.org/wiki/Fr%C3%A9chet\\_distribution](https://en.wikipedia.org/wiki/Fr%C3%A9chet_distribution)
- [R855] [https://en.wikipedia.org/wiki/Gamma\\_distribution](https://en.wikipedia.org/wiki/Gamma_distribution)
- [R856] <http://mathworld.wolfram.com/GammaDistribution.html>
- [R857] [https://en.wikipedia.org/wiki/Inverse-gamma\\_distribution](https://en.wikipedia.org/wiki/Inverse-gamma_distribution)
- [R858] [https://en.wikipedia.org/wiki/Gompertz\\_distribution](https://en.wikipedia.org/wiki/Gompertz_distribution)
- [R859] <http://mathworld.wolfram.com/GumbelDistribution.html>
- [R860] [https://en.wikipedia.org/wiki/Gumbel\\_distribution](https://en.wikipedia.org/wiki/Gumbel_distribution)
- [R861] [http://www.mathwave.com/help/easyfit/html/analyses/distributions/gumbel\\_max.html](http://www.mathwave.com/help/easyfit/html/analyses/distributions/gumbel_max.html)
- [R862] [http://www.mathwave.com/help/easyfit/html/analyses/distributions/gumbel\\_min.html](http://www.mathwave.com/help/easyfit/html/analyses/distributions/gumbel_min.html)
- [R863] [https://en.wikipedia.org/wiki/Kumaraswamy\\_distribution](https://en.wikipedia.org/wiki/Kumaraswamy_distribution)
- [R864] [https://en.wikipedia.org/wiki/Laplace\\_distribution](https://en.wikipedia.org/wiki/Laplace_distribution)
- [R865] <http://mathworld.wolfram.com/LaplaceDistribution.html>
- [R866] [https://en.wikipedia.org/wiki/L%C3%A9vy\\_distribution](https://en.wikipedia.org/wiki/L%C3%A9vy_distribution)
- [R867] <http://mathworld.wolfram.com/LevyDistribution.html>
- [R868] [https://en.wikipedia.org/wiki/Logistic\\_distribution](https://en.wikipedia.org/wiki/Logistic_distribution)

- [R869] <http://mathworld.wolfram.com/LogisticDistribution.html>
- [R870] [https://en.wikipedia.org/wiki/Log-logistic\\_distribution](https://en.wikipedia.org/wiki/Log-logistic_distribution)
- [R871] <https://en.wikipedia.org/wiki/Lognormal>
- [R872] <http://mathworld.wolfram.com/LogNormalDistribution.html>
- [R873] [https://en.wikipedia.org/wiki/Lomax\\_distribution](https://en.wikipedia.org/wiki/Lomax_distribution)
- [R874] [https://en.wikipedia.org/wiki/Maxwell\\_distribution](https://en.wikipedia.org/wiki/Maxwell_distribution)
- [R875] <http://mathworld.wolfram.com/MaxwellDistribution.html>
- [R876] <https://reference.wolfram.com/language/ref/MoyalDistribution.html>
- [R877] <http://www.stat.rice.edu/~dobelman/textfiles/DistributionsHandbook.pdf>
- [R878] [https://en.wikipedia.org/wiki/Nakagami\\_distribution](https://en.wikipedia.org/wiki/Nakagami_distribution)
- [R879] [https://en.wikipedia.org/wiki/Normal\\_distribution](https://en.wikipedia.org/wiki/Normal_distribution)
- [R880] <http://mathworld.wolfram.com/NormalDistributionFunction.html>
- [R881] [https://en.wikipedia.org/wiki/Pareto\\_distribution](https://en.wikipedia.org/wiki/Pareto_distribution)
- [R882] <http://mathworld.wolfram.com/ParetoDistribution.html>
- [R883] [http://www.mathwave.com/help/easyfit/html/analyses/distributions/power\\_func.html](http://www.mathwave.com/help/easyfit/html/analyses/distributions/power_func.html)
- [R884] [https://en.wikipedia.org/wiki/U-quadratic\\_distribution](https://en.wikipedia.org/wiki/U-quadratic_distribution)
- [R885] [https://en.wikipedia.org/wiki/Raised\\_cosine\\_distribution](https://en.wikipedia.org/wiki/Raised_cosine_distribution)
- [R886] [https://en.wikipedia.org/wiki/Rayleigh\\_distribution](https://en.wikipedia.org/wiki/Rayleigh_distribution)
- [R887] <http://mathworld.wolfram.com/RayleighDistribution.html>
- [R888] [https://en.wikipedia.org/wiki/Reciprocal\\_distribution](https://en.wikipedia.org/wiki/Reciprocal_distribution)
- [R889] [https://en.wikipedia.org/wiki/Student\\_t-distribution](https://en.wikipedia.org/wiki/Student_t-distribution)
- [R890] <http://mathworld.wolfram.com/Studentst-Distribution.html>
- [R891] [https://en.wikipedia.org/wiki/Shifted\\_Gompertz\\_distribution](https://en.wikipedia.org/wiki/Shifted_Gompertz_distribution)
- [R892] [https://en.wikipedia.org/wiki/Trapezoidal\\_distribution](https://en.wikipedia.org/wiki/Trapezoidal_distribution)
- [R893] [https://en.wikipedia.org/wiki/Triangular\\_distribution](https://en.wikipedia.org/wiki/Triangular_distribution)
- [R894] <http://mathworld.wolfram.com/TriangularDistribution.html>
- [R895] [https://en.wikipedia.org/wiki/Uniform\\_distribution\\_%28continuous%29](https://en.wikipedia.org/wiki/Uniform_distribution_%28continuous%29)
- [R896] <http://mathworld.wolfram.com/UniformDistribution.html>
- [R897] [https://en.wikipedia.org/wiki/Uniform\\_sum\\_distribution](https://en.wikipedia.org/wiki/Uniform_sum_distribution)
- [R898] <http://mathworld.wolfram.com/UniformSumDistribution.html>
- [R899] [https://en.wikipedia.org/wiki/Von\\_Mises\\_distribution](https://en.wikipedia.org/wiki/Von_Mises_distribution)
- [R900] <http://mathworld.wolfram.com/vonMisesDistribution.html>
- [R901] [https://en.wikipedia.org/wiki/Inverse\\_Gaussian\\_distribution](https://en.wikipedia.org/wiki/Inverse_Gaussian_distribution)
- [R902] <http://mathworld.wolfram.com/InverseGaussianDistribution.html>
- [R903] [https://en.wikipedia.org/wiki/Weibull\\_distribution](https://en.wikipedia.org/wiki/Weibull_distribution)

- [R904] <http://mathworld.wolfram.com/WeibullDistribution.html>
- [R905] [https://en.wikipedia.org/wiki/Wigner\\_semicircle\\_distribution](https://en.wikipedia.org/wiki/Wigner_semicircle_distribution)
- [R906] <http://mathworld.wolfram.com/WignersSemicircleLaw.html>
- [R907] [https://en.wikipedia.org/wiki/Multivariate\\_normal\\_distribution](https://en.wikipedia.org/wiki/Multivariate_normal_distribution)
- [R908] [https://en.wikipedia.org/wiki/Multivariate\\_Laplace\\_distribution](https://en.wikipedia.org/wiki/Multivariate_Laplace_distribution)
- [R909] [https://en.wikipedia.org/wiki/Generalized\\_multivariate\\_log-gamma\\_distribution](https://en.wikipedia.org/wiki/Generalized_multivariate_log-gamma_distribution)
- [R910] [https://www.researchgate.net/publication/234137346\\_On\\_a\\_multivariate\\_log-gamma\\_distribution\\_and\\_the\\_use\\_of\\_the\\_distribution\\_in\\_the\\_Bayesian\\_analysis](https://www.researchgate.net/publication/234137346_On_a_multivariate_log-gamma_distribution_and_the_use_of_the_distribution_in_the_Bayesian_analysis)
- [R911] [https://en.wikipedia.org/wiki/Generalized\\_multivariate\\_log-gamma\\_distribution](https://en.wikipedia.org/wiki/Generalized_multivariate_log-gamma_distribution)
- [R912] [https://www.researchgate.net/publication/234137346\\_On\\_a\\_multivariate\\_log-gamma\\_distribution\\_and\\_the\\_use\\_of\\_the\\_distribution\\_in\\_the\\_Bayesian\\_analysis](https://www.researchgate.net/publication/234137346_On_a_multivariate_log-gamma_distribution_and_the_use_of_the_distribution_in_the_Bayesian_analysis)
- [R913] [https://en.wikipedia.org/wiki/Multinomial\\_distribution](https://en.wikipedia.org/wiki/Multinomial_distribution)
- [R914] <http://mathworld.wolfram.com/MultinomialDistribution.html>
- [R915] [https://en.wikipedia.org/wiki/Dirichlet\\_distribution](https://en.wikipedia.org/wiki/Dirichlet_distribution)
- [R916] <http://mathworld.wolfram.com/DirichletDistribution.html>
- [R917] [https://en.wikipedia.org/wiki/Ewens%27s\\_sampling\\_formula](https://en.wikipedia.org/wiki/Ewens%27s_sampling_formula)
- [R918] <http://www.stat.rutgers.edu/home/hcrane/Papers/STS529.pdf>
- [R919] [https://en.wikipedia.org/wiki/Negative\\_multinomial\\_distribution](https://en.wikipedia.org/wiki/Negative_multinomial_distribution)
- [R920] <http://mathworld.wolfram.com/NegativeBinomialDistribution.html>
- [R921] [https://en.wikipedia.org/wiki/Normal-gamma\\_distribution](https://en.wikipedia.org/wiki/Normal-gamma_distribution)
- [R922] [https://en.wikipedia.org/wiki/Markov\\_chain#Discrete-time\\_Markov\\_chain](https://en.wikipedia.org/wiki/Markov_chain#Discrete-time_Markov_chain)
- [R923] [https://www.dartmouth.edu/~chance/teaching\\_aids/books\\_articles/probability\\_book/Chapter11.pdf](https://www.dartmouth.edu/~chance/teaching_aids/books_articles/probability_book/Chapter11.pdf)
- [R924] <https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470316887.app1>
- [R925] <http://www.columbia.edu/~ww2040/6711F12/lect1023big.pdf>
- [R926] <http://www.columbia.edu/~ww2040/4701Sum07/4701-06-Notes-MCII.pdf>
- [R927] <http://cecas.clemson.edu/~shierd/Shier/markov.pdf>
- [R928] <https://ujcontent.uj.ac.za/vital/access/services/Download/uj:7506/CONTENT1>
- [R929] <https://www.mathworks.com/help/econ/dtmc.classify.html>
- [R930] [https://en.wikipedia.org/wiki/Absorbing\\_Markov\\_chain](https://en.wikipedia.org/wiki/Absorbing_Markov_chain)
- [R931] [http://people.brandeis.edu/~igusa/Math56aS08/Math56a\\_S08\\_notes015.pdf](http://people.brandeis.edu/~igusa/Math56aS08/Math56a_S08_notes015.pdf)
- [R932] <https://lips.cs.princeton.edu/the-fundamental-matrix-of-a-finite-markov-chain/>
- [R933] [https://www.probabilitycourse.com/chapter11/11\\_2\\_6\\_stationary\\_and\\_limiting\\_distributions.php](https://www.probabilitycourse.com/chapter11/11_2_6_stationary_and_limiting_distributions.php)
- [R934] [https://galton.uchicago.edu/~yibi/teaching/stat317/2014/Lectures/Lecture4\\_6up.pdf](https://galton.uchicago.edu/~yibi/teaching/stat317/2014/Lectures/Lecture4_6up.pdf)

- [R935] [https://en.wikipedia.org/wiki/Markov\\_chain#Continuous-time\\_Markov\\_chain](https://en.wikipedia.org/wiki/Markov_chain#Continuous-time_Markov_chain)
- [R936] <http://u.math.biu.ac.il/~amirgi/CTMCnotes.pdf>
- [R937] [https://en.wikipedia.org/wiki/Bernoulli\\_process](https://en.wikipedia.org/wiki/Bernoulli_process)
- [R938] <https://mathcs.clarku.edu/~djoyce/ma217/bernoulli.pdf>
- [R939] [https://www.probabilitycourse.com/chapter11/11\\_0\\_0\\_intro.php](https://www.probabilitycourse.com/chapter11/11_0_0_intro.php)
- [R940] [https://en.wikipedia.org/wiki/Poisson\\_point\\_process](https://en.wikipedia.org/wiki/Poisson_point_process)
- [R941] [https://www.probabilitycourse.com/chapter11/11\\_4\\_0\\_brownian\\_motion\\_wiener\\_process.php](https://www.probabilitycourse.com/chapter11/11_4_0_brownian_motion_wiener_process.php)
- [R942] [https://en.wikipedia.org/wiki/Wiener\\_process](https://en.wikipedia.org/wiki/Wiener_process)
- [R943] [https://en.wikipedia.org/wiki/Gamma\\_process](https://en.wikipedia.org/wiki/Gamma_process)
- [R944] [https://en.wikipedia.org/wiki/Matrix\\_gamma\\_distribution](https://en.wikipedia.org/wiki/Matrix_gamma_distribution)
- [R945] [https://en.wikipedia.org/wiki/Wishart\\_distribution](https://en.wikipedia.org/wiki/Wishart_distribution)
- [R946] [https://en.wikipedia.org/wiki/Matrix\\_normal\\_distribution](https://en.wikipedia.org/wiki/Matrix_normal_distribution)
- [R947] [https://en.wikipedia.org/wiki/Compound\\_probability\\_distribution](https://en.wikipedia.org/wiki/Compound_probability_distribution)
- [R948] [https://en.wikipedia.org/wiki/Entropy\\_\(information\\_theory\)](https://en.wikipedia.org/wiki/Entropy_(information_theory))
- [R949] [https://www.crmarsch.com/static/pdf/Charles\\_Marsh\\_Continuous\\_Entropy.pdf](https://www.crmarsch.com/static/pdf/Charles_Marsh_Continuous_Entropy.pdf)
- [R950] <http://www.math.uconn.edu/~kconrad/blurbs/analysis/entropypost.pdf>
- [R951] <https://en.wikipedia.org/wiki/Coskewness>
- [R952] [https://en.wikipedia.org/wiki/Median#Probability\\_distributions](https://en.wikipedia.org/wiki/Median#Probability_distributions)
- [R953] [https://en.wikipedia.org/wiki/Factorial\\_moment](https://en.wikipedia.org/wiki/Factorial_moment)
- [R954] <http://mathworld.wolfram.com/FactorialMoment.html>
- [R955] <https://en.wikipedia.org/wiki/Kurtosis>
- [R956] <http://mathworld.wolfram.com/Kurtosis.html>





## PYTHON MODULE INDEX

### S

sympy, ??  
sympy.abc, 884  
sympy.algebras, 885  
sympy.assumptions, 186  
sympy.assumptions.ask, 191  
sympy.assumptions.assume, 193  
sympy.assumptions.predicates, 200  
sympy.assumptions.refine, 197  
sympy.calculus, 233  
sympy.calculus.euler, 233  
sympy.calculus.finite\_diff, 240  
sympy.calculus.singularities, 234  
sympy.calculus.util, 244  
sympy.categories, 2742  
sympy.categories.diagram\_drawing, 2752  
sympy.codegen.algorithms, 1157  
sympy.codegen.approximations, 1122  
sympy.codegen.ast, 1124  
sympy.codegen.cfunctions, 1143  
sympy.codegen.cnodes, 1148  
sympy.codegen.cutils, 1159  
sympy.codegen.cxxnodes, 1149  
sympy.codegen.fnodes, 1149  
sympy.codegen.futils, 1159  
sympy.codegen.matrix\_nodes, 1121  
sympy.codegen.pyutils, 1159  
sympy.codegen.rewriting, 1118  
sympy.combinatorics.generators, 287  
sympy.combinatorics.graycode, 347  
sympy.combinatorics.group\_constructs, 363  
sympy.combinatorics.group\_numbers, 355  
sympy.combinatorics.named\_groups, 352  
sympy.combinatorics.partitions, 251  
sympy.combinatorics.perm\_groups, 288  
sympy.combinatorics.permutations, 257  
sympy.combinatorics.polyhedron, 332  
sympy.combinatorics.prufer, 335  
sympy.combinatorics.subsets, 340  
sympy.combinatorics.tensor\_can, 365  
sympy.combinatorics.testutil, 363  
sympy.combinatorics.util, 356  
sympy.core.add, 1013  
sympy.core.assumptions, 923  
sympy.core.basic, 927  
sympy.core.cache, 927  
sympy.core.containers, 1068  
sympy.core.core, 945  
sympy.core.evalf, 1065  
sympy.core.expr, 947  
sympy.core.exprtools, 1071  
sympy.core.function, 1039  
sympy.core.kind, 1073  
sympy.core.mod, 1018  
sympy.core.mul, 1009  
sympy.core.multidimensional, 1039  
sympy.core.numbers, 981  
sympy.core.power, 1005  
sympy.core.random, 1077  
sympy.core.relational, 1018  
sympy.core.singleton, 945  
sympy.core.symbol, 976  
sympy.core.sympify, 918  
sympy.core.traversal, 1080  
sympy.crypto.crypto, 2762  
sympy.diffgeom, 2799  
sympy.discrete, 1083  
sympy.discrete.convolutions, 1089  
sympy.discrete.transforms, 1083  
sympy.functions, 381  
sympy.functions.special.bessel, 497  
sympy.functions.special.beta\_functions, 470  
sympy.functions.special.elliptic\_integrals, 526  
sympy.functions.special.error\_functions, 471  
sympy.functions.special.gamma\_functions, 459  
sympy.functions.special.mathieu\_functions, 529  
sympy.functions.special.polynomials, 532

`sympy.functions.special.singularity_functions`, 456  
`sympy.functions.special.zeta_functions`, 514  
`sympy.geometry.curve`, 2247  
`sympy.geometry.ellipse`, 2253  
`sympy.geometry.entity`, 2194  
`sympy.geometry.line`, 2216  
`sympy.geometry.plane`, 2303  
`sympy.geometry.point`, 2202  
`sympy.geometry.polygon`, 2273  
`sympy.geometry.util`, 2197  
`sympy.holonomic`, 2312  
`sympy.integrals`, 575  
`sympy.integrals.intpoly`, 617  
`sympy.integrals.meijerint`, 568  
`sympy.integrals.meijerint_doc`, 564  
`sympy.integrals.transforms`, 576  
`sympy.interactive`, 2116  
`sympy.interactive.printing`, 2119  
`sympy.interactive.session`, 2116  
`sympy.liealgebras`, 2326  
`sympy.logic`, 1160  
`sympy.logic.inference`, 1184  
`sympy.matrices`, 1217  
`sympy.matrices.common`, 1327  
`sympy.matrices.expressions`, 1369  
`sympy.matrices.expressions.blockmatrix`, 1380  
`sympy.matrices.immutable`, 1369  
`sympy.matrices.matrices`, 1218  
`sympy.matrices.sparse`, 1364  
`sympy.matrices.sparsetools`, 1365  
`sympy.ntheory.bbp_pi`, 1531  
`sympy.ntheory.continued_fraction`, 1523  
`sympy.ntheory.digits`, 1526  
`sympy.ntheory.ecm`, 1531  
`sympy.ntheory.egyptian_fraction`, 1529  
`sympy.ntheory.factor_`, 1486  
`sympy.ntheory.generate`, 1476  
`sympy.ntheory.modular`, 1509  
`sympy.ntheory.multinomial`, 1511  
`sympy.ntheory.partitions_`, 1513  
`sympy.ntheory.primetest`, 1513  
`sympy.ntheory.qs`, 1532  
`sympy.ntheory.residue_ntheory`, 1518  
`sympy.parsing`, 2122  
`sympy.parsing.sym_expr`, 2130  
`sympy.physics`, 1533  
`sympy.physics.continuum_mechanics.beam`, 1949  
`sympy.physics.continuum_mechanics.truss`, 2021  
`sympy.physics.control`, 1890  
`sympy.physics.control.lti`, 1891  
`sympy.physics.hep.gamma_matrices`, 1589  
`sympy.physics.hydrogen`, 1533  
`sympy.physics.matrices`, 1537  
`sympy.physics.mechanics`, 1675  
`sympy.physics.mechanics.body`, 1756  
`sympy.physics.mechanics.joint`, 1770  
`sympy.physics.mechanics.jointsmethod`, 1780  
`sympy.physics.mechanics.kane`, 1764  
`sympy.physics.mechanics.lagrange`, 1767  
`sympy.physics.mechanics.linearize`, 1787  
`sympy.physics.mechanics.particle`, 1743  
`sympy.physics.mechanics.rigidbody`, 1746  
`sympy.physics.mechanics.system`, 1783  
`sympy.physics.optics.gaussopt`, 1857  
`sympy.physics.optics.medium`, 1870  
`sympy.physics.optics.polarization`, 1871  
`sympy.physics.optics.utils`, 1880  
`sympy.physics.optics.waves`, 1887  
`sympy.physics.paulialgebra`, 1539  
`sympy.physics.qho_ld`, 1539  
`sympy.physics.quantum.anticommutator`, 1791  
`sympy.physics.quantum.cartesian`, 1801  
`sympy.physics.quantum.cg`, 1792  
`sympy.physics.quantum.circuitplot`, 1840  
`sympy.physics.quantum.commutator`, 1795  
`sympy.physics.quantum.constants`, 1796  
`sympy.physics.quantum.dagger`, 1797  
`sympy.physics.quantum.gate`, 1842  
`sympy.physics.quantum.grover`, 1847  
`sympy.physics.quantum.hilbert`, 1802  
`sympy.physics.quantum.innerproduct`, 1798  
`sympy.physics.quantum.operator`, 1807  
`sympy.physics.quantum.operatorset`, 1812  
`sympy.physics.quantum.piab`, 1857  
`sympy.physics.quantum.qapply`, 1814  
`sympy.physics.quantum.qft`, 1850  
`sympy.physics.quantum.qubit`, 1850  
`sympy.physics.quantum.represent`, 1815  
`sympy.physics.quantum.shor`, 1856  
`sympy.physics.quantum.spin`, 1820  
`sympy.physics.quantum.state`, 1832  
`sympy.physics.quantum.tensorproduct`, 1799  
`sympy.physics.secondquant`, 1542  
`sympy.physics.sho`, 1541  
`sympy.physics.units.dimensions`, 1584  
`sympy.physics.units.prefixes`, 1586  
`sympy.physics.units.quantities`, 1587  
`sympy.physics.units.unitsystem`, 1587  
`sympy.physics.units.util`, 1588

---

[sympy.physics.vector](#), 1592  
[sympy.physics.vector.functions](#), 1659  
[sympy.physics.vector.point](#), 1652  
[sympy.physics.wigner](#), 1566  
[sympy.plotting.plot](#), 2820  
[sympy.plotting.pygletplot](#), 2869  
[sympy.polys](#), 2360  
[sympy.polys.numberfields.modules](#), 2716  
[sympy.polys.numberfields.subfield](#), 2711  
[sympy.polys.polyconfig](#), 2643  
[sympy.polys.polyoptions](#), 2642  
[sympy.polys.solvers](#), 2666  
[sympy.printing.aesaracode](#), 2166  
[sympy.printing.c](#), 2140  
[sympy.printing.codeprinter](#), 2183  
[sympy.printing.conventions](#), 2183  
[sympy.printing.cxx](#), 2144  
[sympy.printing.fortran](#), 2147  
[sympy.printing.gtk](#), 2170  
[sympy.printing.jscode](#), 2155  
[sympy.printing.julia](#), 2157  
[sympy.printing.lambdarepr](#), 2170  
[sympy.printing.latex](#), 2170  
[sympy.printing.maple](#), 2154  
[sympy.printing.mathematica](#), 2152  
[sympy.printing.mathml](#), 2175  
[sympy.printing.octave](#), 2160  
[sympy.printing.precedence](#), 2184  
[sympy.printing.pretty](#), 2139  
[sympy.printing.pretty.pretty](#), 2139  
[sympy.printing.pretty.pretty\\_symbology](#), 2184  
[sympy.printing.pretty.stringpict](#), 2186  
[sympy.printing.preview](#), 2181  
[sympy.printing.printer](#), 2135  
[sympy.printing.pycode](#), 2176  
[sympy.printing.python](#), 2177  
[sympy.printing.rcode](#), 2144  
[sympy.printing.repr](#), 2177  
[sympy.printing.rust](#), 2164  
[sympy.printing.str](#), 2178  
[sympy.printing.tree](#), 2178  
[sympy.sets.conditionset](#), 1215  
[sympy.sets.fancysets](#), 1203  
[sympy.sets.powerset](#), 1213  
[sympy.sets.sets](#), 1185  
[sympy.simplify.combsimp](#), 683  
[sympy.simplify.cse\\_main](#), 685  
[sympy.simplify.epathtools](#), 688  
[sympy.simplify.hyperexpand](#), 687  
[sympy.simplify.hyperexpand\\_doc](#), 699  
[sympy.simplify.powsimp](#), 680  
[sympy.simplify.radsimp](#), 670  
[sympy.simplify.ratsimp](#), 678  
[sympy.simplify.simplify](#), 154  
[sympy.simplify.sqrtdenest](#), 684  
[sympy.simplify.trigsimp](#), 679  
[sympy.solvers](#), 836  
[sympy.solvers.inequalities](#), 750  
[sympy.solvers.ode](#), 755  
[sympy.solvers.ode.ode](#), 822  
[sympy.solvers.pde](#), 835  
[sympy.solvers.recurr](#), 853  
[sympy.solvers.solve](#), 858  
[sympy.stats](#), 2873  
[sympy.stats.crv](#), 2981  
[sympy.stats.crv\\_types](#), 2981  
[sympy.stats.frv](#), 2981  
[sympy.stats.frv\\_types](#), 2981  
[sympy.stats.rv](#), 2980  
[sympy.tensor](#), 1387  
[sympy.tensor.array](#), 1387  
[sympy.tensor.array.expressions](#), 1395  
[sympy.tensor.index\\_methods](#), 1406  
[sympy.tensor.indexed](#), 1398  
[sympy.tensor.tensor](#), 1409  
[sympy.tensor.toperators](#), 1423  
[sympy.testing](#), 2027  
[sympy.testing.pytest](#), 2027  
[sympy.testing.randtest](#), 2030  
[sympy.testing.runtests](#), 2030  
[sympy.utilities](#), 2038  
[sympy.utilities.autowrap](#), 2039  
[sympy.utilities.codegen](#), 2046  
[sympy.utilities.decorator](#), 2057  
[sympy.utilities.enumerative](#), 2060  
[sympy.utilities.exceptions](#), 2066  
[sympy.utilities.iterables](#), 2071  
[sympy.utilities.lambdify](#), 2099  
[sympy.utilities.memoization](#), 2110  
[sympy.utilities.misc](#), 2110  
[sympy.utilities.pkgdata](#), 2115  
[sympy.utilities.source](#), 2116  
[sympy.utilities.timeutils](#), 2116  
[sympy.vector](#), 1425



# INDEX

## Symbols

\_\_CoeffExpValueError, 568  
 \_\_TensorManager (class in `sympy.tensor.tensor`), 1422  
 \_\_abs\_\_ () (`sympy.matrices.common.MatrixCommon` method), 1327  
 \_\_add\_\_ () (`sympy.matrices.common.MatrixCommon` method), 1327  
 \_\_add\_\_ () (`sympy.polys.numberfields.modules.ModuleElement` method), 2729  
 \_\_add\_\_ () (`sympy.polys.numberfields.primes.PrimeIdeal` method), 2706  
 \_\_cacheit\_\_ (in module `sympy.core.cache`), 927  
 \_\_call\_\_ () (`sympy.polys.numberfields.modules.ModuleElement` method), 2719  
 \_\_contains\_\_ () (`sympy.combinatorics.perm_groups.PermutationGroup` method), 289  
 \_\_getitem\_\_ () (`sympy.matrices.common.MatrixCommon` method), 1327  
 \_\_init\_\_ () (`sympy.polys.numberfields.modules.Endomorphism` method), 2734  
 \_\_init\_\_ () (`sympy.polys.numberfields.modules.InnerEndomorphism` method), 2734  
 \_\_init\_\_ () (`sympy.polys.numberfields.modules.ModuleElement` method), 2729  
 \_\_init\_\_ () (`sympy.polys.numberfields.modules.ModuleEndomorphism` method), 2734  
 \_\_init\_\_ () (`sympy.polys.numberfields.modules.ModuleHomomorphism` method), 2733  
 \_\_init\_\_ () (`sympy.polys.numberfields.modules.PowerBasis` method), 2725  
 \_\_init\_\_ () (`sympy.polys.numberfields.modules.Submodule` method), 2726  
 \_\_init\_\_ () (`sympy.polys.numberfields.primes.PrimeIdeal` method), 2706  
 \_\_init\_\_ () (`sympy.polys.numberfields.utilities.AlgIntPower` method), 2739  
 \_\_init\_\_ () (`sympy.vector.coordsysrect.CoordSys3D` method), 1448  
 \_\_init\_\_ () (`sympy.vector.orienters.AxisOrienter` method), 1466  
 \_\_init\_\_ () (`sympy.vector.orienters.BodyOrienter` method), 1467  
 \_\_init\_\_ () (`sympy.vector.orienters.QuaternionOrienter` method), 1469  
 \_\_init\_\_ () (`sympy.vector.orienters.SpaceOrienter` method), 1468  
 \_\_len\_\_ () (`sympy.matrices.common.MatrixCommon` method), 1327  
 \_\_mod\_\_ () (`sympy.polys.numberfields.modules.ModuleElement` method), 2730  
 \_\_mul\_\_ () (`sympy.combinatorics.perm_groups.PermutationGroup` method), 289  
 \_\_mul\_\_ () (`sympy.matrices.common.MatrixCommon` method), 1327  
 \_\_mul\_\_ () (`sympy.polys.numberfields.modules.ModuleElement` method), 2730  
 \_\_mul\_\_ () (`sympy.polys.numberfields.primes.PrimeIdeal` method), 2706  
 \_\_new\_\_ () (`sympy.combinatorics.perm_groups.PermutationGroup` static method), 290  
 \_\_new\_\_ () (`sympy.core.numbers.AlgebraicNumber` static method), 988  
 \_\_pow\_\_ () (`sympy.matrices.common.MatrixCommon` method), 1328  
 \_\_weakref\_\_ (`sympy.combinatorics.perm_groups.PermutationGroup` attribute), 290  
 \_\_weakref\_\_ (`sympy.matrices.common.MatrixCommon` attribute), 1328  
 af\_parity() (in module `sympy.combinatorics.permutations`), 286  
 \_\_all\_roots\_\_ () (`sympy.polys.rootoftools.ComplexRootOf` class method), 2434  
 \_\_base\_ordering\_\_ (in module `sympy.combinatorics.util`), 356  
 \_\_check\_antecedents\_\_ (in module `sympy.integrals.meijerint`), 568  
 \_\_check\_antecedents\_1\_\_ (in module `sympy.integrals.meijerint`), 568  
 \_\_check\_antecedents\_inversion\_\_ (in module `sympy.integrals.meijerint`), 569  
 \_\_check\_cycles\_alt\_sym\_\_ (in module

`sympy.combinatorics.util`), 357

`_cmp_perm_lists()` (in module `sympy.combinatorics.testutil`), 363

`_complexes_index()` (`sympy.polys.rootoftools.ComplexRootOf` class method), 2434

`_complexes_sorted()` (`sympy.polys.rootoftools.ComplexRootOf` class method), 2434

`_condsimp()` (in module `sympy.integrals.meijerint`), 569

`_convert_poly_rat_alg()` (in module `sympy.holonomic.holonomic`), 2326

`_coset_representative()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 290

`_count_roots()` (`sympy.polys.rootoftools.ComplexRootOf` class method), 2434

`_create_lookup_table()` (in module `sympy.integrals.meijerint`), 569

`_create_table()` (in module `sympy.holonomic.holonomic`), 2326

`_csrtodok()` (in module `sympy.matrices.sparsetools`), 1365

`_diff_wrt` (`sympy.core.function.Derivative` property), 1045

`_distinct_primes_lemma()` (`sympy.combinatorics.perm_groups.PermutationGroup` class method), 290

`_distribute_gens_by_base()` (in module `sympy.combinatorics.util`), 357

`_doktocsr()` (in module `sympy.matrices.sparsetools`), 1365

`_dummy()` (in module `sympy.integrals.meijerint`), 569

`_dummy_()` (in module `sympy.integrals.meijerint`), 569

`_elements` (`sympy.combinatorics.perm_groups.PermutationGroup` property), 290

`_ensure_complexes_init()` (`sympy.polys.rootoftools.ComplexRootOf` method), 2434

`_ensure_reals_init()` (`sympy.polys.rootoftools.ComplexRootOf` method), 2434

`_eval_cond()` (in module `sympy.integrals.meijerint`), 569

`_eval_evalf()` (`sympy.polys.rootoftools.ComplexRootOf` method), 2434

`_eval_integral()` (`sympy.functions.elementary.integrals.Piecewise` method), 416

`_eval_is_alt_sym_monte_carlo()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 290

`_eval_is_alt_sym_naive()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 291

`_eval_is_imaginary()` (`sympy.polys.rootoftools.ComplexRootOf` method), 2434

`_eval_is_real()` (`sympy.polys.rootoftools.ComplexRootOf` method), 2434

`_exponents()` (in module `sympy.integrals.meijerint`), 569

`_fast_inverse_laplace()` (in module `sympy.integrals.transforms`), 579

`_find_splitting_points()` (in module `sympy.integrals.meijerint`), 570

`_full_transform()` (in module `sympy.integrals.meijerint`), 570

`_full_transform()` (in module `sympy.integrals.transforms`), 580

`_functions()` (in module `sympy.integrals.meijerint`), 570

`_get_coeff_exp()` (in module `sympy.integrals.meijerint`), 570

`_get_complexes()` (`sympy.polys.rootoftools.ComplexRootOf` class method), 2434

`_get_complexes_sqf()` (`sympy.polys.rootoftools.ComplexRootOf` class method), 2434

`_get_interval()` (`sympy.polys.rootoftools.ComplexRootOf` method), 2434

`_get_reals()` (`sympy.polys.rootoftools.ComplexRootOf` class method), 2434

`_get_reals_sqf()` (`sympy.polys.rootoftools.ComplexRootOf` class method), 2434

`_get_roots()` (`sympy.polys.rootoftools.ComplexRootOf` class method), 2435

`_guess_expansion()` (in module `sympy.integrals.meijerint`), 570

`_handle_precomputed_bsqs()` (in module `sympy.combinatorics.util`), 358

`_indexed_root()` (`sympy.polys.rootoftools.ComplexRootOf` class method), 2435

`_inflate_fox_h()` (in module `sympy.integrals.meijerint`), 570

`_inflate_g()` (in module `sympy.integrals.meijerint`), 571

`_is_rootof()` (in module `sympy.integrals.meijerint`), 571

`_is_rootof()` (in module `sympy.integrals.meijerint`), 571

`_int_inversion()` (in module `sympy.integrals.meijerint`), 571

`_is_analytic()` (in module



`sympy.integrals.meijerint`), 571  
`_is_exponential()` (in module `sympy.solvers.solve`), 880  
`_is_logarithmic()` (in module `sympy.solvers.solve`), 883  
`_linear_2eq_order1_type6()` (in module `sympy.solvers.ode.ode`), 806  
`_linear_2eq_order1_type7()` (in module `sympy.solvers.ode.ode`), 806  
`_meijerint_definite_2()` (in module `sympy.integrals.meijerint`), 571  
`_meijerint_definite_3()` (in module `sympy.integrals.meijerint`), 571  
`_meijerint_definite_4()` (in module `sympy.integrals.meijerint`), 571  
`_meijerint_indefinite_1()` (in module `sympy.integrals.meijerint`), 572  
`_modgcd_multivariate_p()` (in module `sympy.polys.modulargcd`), 2648  
`_mul_args()` (in module `sympy.integrals.meijerint`), 572  
`_mul_as_two_parts()` (in module `sympy.integrals.meijerint`), 572  
`_my_principal_branch()` (in module `sympy.integrals.meijerint`), 572  
`_mytype()` (in module `sympy.integrals.meijerint`), 572  
`_naive_list_centralizer()` (in module `sympy.combinatorics.testutil`), 363  
`_new()` (`sympy.polys.rootof`), 2435  
`_nonlinear_2eq_order1_type1()` (in module `sympy.solvers.ode.ode`), 818  
`_nonlinear_2eq_order1_type2()` (in module `sympy.solvers.ode.ode`), 818  
`_nonlinear_2eq_order1_type3()` (in module `sympy.solvers.ode.ode`), 819  
`_nonlinear_2eq_order1_type4()` (in module `sympy.solvers.ode.ode`), 819  
`_nonlinear_2eq_order1_type5()` (in module `sympy.solvers.ode.ode`), 820  
`_nonlinear_3eq_order1_type1()` (in module `sympy.solvers.ode.ode`), 820  
`_nonlinear_3eq_order1_type2()` (in module `sympy.solvers.ode.ode`), 820  
`_nonlinear_3eq_order1_type3()` (in module `sympy.solvers.ode.ode`), 821  
`_nonlinear_3eq_order1_type4()` (in module `sympy.solvers.ode.ode`), 821  
`_nonlinear_3eq_order1_type5()` (in module `sympy.solvers.ode.ode`), 822  
`_orbits_transversals_from_bsgs()` (in module `sympy.combinatorics.util`), 359  
`_p_elements_group()` (`sympy.combinatorics.perm_groups`), 291  
`_postprocess_root()` (`sympy.polys.rootof`), 2435  
`_preprocess_roots()` (`sympy.polys.rootof`), 2435  
`_print()` (`sympy.printing.printer`), 2138  
`_randint()` (in module `sympy.core.random`), 1079  
`_random_pr_init()` (`sympy.combinatorics.perm_groups`), 291  
`_randrange()` (in module `sympy.core.random`), 1079  
`_real_roots()` (`sympy.polys.rootof`), 2435  
`_reals_index()` (`sympy.polys.rootof`), 2435  
`_reals_sorted()` (`sympy.polys.rootof`), 2435  
`_refine_complexes()` (`sympy.polys.rootof`), 2435  
`_remove_gens()` (in module `sympy.combinatorics.util`), 360  
`_reset()` (`sympy.polys.rootof`), 2435  
`_rewritel()` (in module `sympy.integrals.meijerint`), 572  
`_rewrite2()` (in module `sympy.integrals.meijerint`), 572  
`_rewrite_inversion()` (in module `sympy.integrals.meijerint`), 573  
`_rewrite_saxena()` (in module `sympy.integrals.meijerint`), 573  
`_rewrite_saxena_1()` (in module `sympy.integrals.meijerint`), 573  
`_rewrite_single()` (in module `sympy.integrals.meijerint`), 573  
`_roots_trivial()` (`sympy.polys.rootof`), 2435  
`_set_interval()` (`sympy.polys.rootof`), 2435  
`_solve_exponential()` (in module `sympy.solvers.solve`), 881  
`_solve_lin_sys()` (in module `sympy.polys.solvers`), 2669  
`_solve_lin_sys_component()` (in module `sympy.polys.solvers`), 2670  
`_solve_logarithm()` (in module

- `sympy.solvers.solve`), 882
  - `_sort_variable_count()` (`sympy.core.function.Derivative` class method), 1045
  - `_split_mul()` (in module `sympy.integrals.meijerint`), 573
  - `_strip()` (in module `sympy.combinatorics.util`), 361
  - `_strong_gens_from_distr()` (in module `sympy.combinatorics.util`), 362
  - `_syLOW_alt_sym()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 291
  - `_tan1()` (in module `sympy.polys.ring_series`), 2656
  - `_transolve()` (in module `sympy.solvers.solve`), 877
  - `_union_find_merge()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 292
  - `_union_find_rep()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 293
  - `_verify()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 293
  - `_verify_bsgs()` (in module `sympy.combinatorics.testutil`), 364
  - `_verify_centralizer()` (in module `sympy.combinatorics.testutil`), 364
  - `_verify_normal_closure()` (in module `sympy.combinatorics.testutil`), 364
- ## A
- `A` (`sympy.physics.optics.gaussopt.RayTransferMatrix` property), 1864
  - `a` (`sympy.physics.quantum.shor.CMod` property), 1856
  - `alpt_theory()` (`sympy.physics.vector.point.Point` method), 1652
  - `a2idx()` (in module `sympy.matrices.matrices`), 1326
  - `a2pt_theory()` (`sympy.physics.vector.point.Point` method), 1653
  - `a_interval` (`sympy.sets.fancysets.ComplexRegion` property), 1210
  - `abbrev` (`sympy.physics.units.quantities.Quantity` property), 1587
  - `abelian_invariants()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 294
  - `AbelianGroup()` (in module `sympy.combinatorics.named_groups`), 354
  - `above()` (`sympy.printing.pretty.stringpict.stringPict` method), 2186
  - `Abs` (class in `sympy.functions.elementary.complexes`), 385
  - `abs()` (`sympy.polys.domains.domain.Domain` method), 2507
  - `abs()` (`sympy.polys.polyclasses.DMP` method), 2561
  - `abs()` (`sympy.polys.polytools.Poly` method), 2380
  - `absorbing_probabilities()` (`sympy.stats.DiscreteMarkovChain` method), 2945
  - `abundance()` (in module `sympy.ntheory.factor_`), 1508
  - `acc()` (`sympy.physics.vector.point.Point` method), 1654
  - `accepted_latex_functions` (in module `sympy.printing.latex`), 2170
  - `acos` (class in `sympy.functions.elementary.trigonometric`), 396
  - `acosh` (class in `sympy.functions.elementary.hyperbolic`), 406
  - `acot` (class in `sympy.functions.elementary.trigonometric`), 407
  - `acoth` (class in `sympy.functions.elementary.hyperbolic`), 407
  - `acsc` (class in `sympy.functions.elementary.trigonometric`), 399
  - `acsch` (class in `sympy.functions.elementary.hyperbolic`), 408
  - `Add` (class in `sympy.core.add`), 1013
  - `add()` (`sympy.algebras.Quaternion` method), 886
  - `add()` (`sympy.assumptions.assume.AssumptionsContext` method), 195
  - `add()` (`sympy.matrices.matrices.MatrixBase` method), 1291
  - `add()` (`sympy.polys.domains.domain.Domain` method), 2507
  - `add()` (`sympy.polys.matrices.ddm.DDM` method), 2690
  - `add()` (`sympy.polys.matrices.domainmatrix.DomainMatrix` method), 2672
  - `add()` (`sympy.polys.matrices.sdm.SDM` method), 2692
  - `add()` (`sympy.polys.numberfields.modules.Submodule` method), 2727
  - `add()` (`sympy.polys.polyclasses.DMF` method), 2567
  - `add()` (`sympy.polys.polyclasses.DMP` method), 2561
  - `add()` (`sympy.polys.polytools.Poly` method), 2381
  - `add()` (`sympy.polys.rings.PolyRing` method), 2553



[add\\_as\\_roots\(\)](#) (`sympy.liealgebras.root_system.RootSystem` method), 2526  
[add\\_gens\(\)](#) (`sympy.polys.rings.PolyRing` method), 2554  
[add\\_ground\(\)](#) (`sympy.polys.polyclasses.DMP` method), 2561  
[add\\_ground\(\)](#) (`sympy.polys.polytools.Poly` method), 2381  
[add\\_member\(\)](#) (`sympy.physics.continuum_mechanics.algebraic_field` method), 2021  
[add\\_node\(\)](#) (`sympy.physics.continuum_mechanics.algebraic_field` method), 2022  
[add\\_simple\\_roots\(\)](#) (`sympy.liealgebras.root_system.RootSystem` method), 2327  
[adjoint\(\)](#) (`sympy.matrices.common.MatrixCommon` method), 1328  
[adjugate\(\)](#) (`sympy.matrices.matrices.MatrixDeterminant` method), 1227  
[aesara\\_code\(\)](#) (in module `sympy.printing.aesaracode`), 2167  
[aesara\\_function\(\)](#) (in module `sympy.printing.aesaracode`), 2167  
[AesaraPrinter](#) (class in `sympy.printing.aesaracode`), 2166  
[affine\\_rank\(\)](#) (`sympy.geometry.point.Point` static method), 2203  
[airyai](#) (class in `sympy.functions.special.bessel`), 504  
[airyaiprime](#) (class in `sympy.functions.special.bessel`), 508  
[AiryBase](#) (class in `sympy.functions.special.bessel`), 504  
[airybi](#) (class in `sympy.functions.special.bessel`), 506  
[airybiprime](#) (class in `sympy.functions.special.bessel`), 509  
[alg\\_con](#) (`sympy.physics.mechanics.system.SymbolicSystem` property), 1786  
[alg\\_field\\_from\\_poly\(\)](#) (`sympy.polys.domains.domain.Domain` method), 2507  
[algebraic](#), 924  
[algebraic\\_field\(\)](#) (`sympy.polys.domains.AlgebraicField` method), 2543  
[algebraic\\_field\(\)](#) (`sympy.polys.domains.domain.Domain` method), 2508  
[algebraic\\_field\(\)](#) (`sympy.polys.domains.IntegerRing` method), 2543  
[algebraic\\_field\(\)](#) (`sympy.polys.domains.RationalField` method), 2530  
[AlgebraicField](#) (class in `sympy.polys.domains`), 2539  
[AlgebraicHandler](#) (`sympy.assumptions.predicates.sets.AlgebraicHandler` attribute), 232  
[AlgebraicNumber](#) (class in `sympy.core.numbers`), 988  
[AlgebraicPredicate](#) (class in `sympy.assumptions.predicates.sets`), 231  
[AlgIntPowers](#) (class in `sympy.polys.numberfields.utilities`), 2739  
[alignof\(\)](#) (in module `sympy.codegen.cnodes`), 1149  
[all\\_coeffs\(\)](#) (`sympy.polys.polyclasses.DMP` method), 2561  
[all\\_coeffs\(\)](#) (`sympy.polys.polytools.Poly` method), 2381  
[all\\_monoms\(\)](#) (`sympy.polys.polyclasses.DMP` method), 2562  
[all\\_monoms\(\)](#) (`sympy.polys.polytools.Poly` method), 2381  
[all\\_roots\(\)](#) (`sympy.liealgebras.root_system.RootSystem` method), 2327  
[all\\_roots\(\)](#) (`sympy.polys.polytools.Poly` method), 2382  
[all\\_roots\(\)](#) (`sympy.polys.rootoftools.ComplexRootOf` class method), 2435  
[all\\_terms\(\)](#) (`sympy.polys.polyclasses.DMP` method), 2562  
[all\\_terms\(\)](#) (`sympy.polys.polytools.Poly` method), 2382  
[allhints](#) (in module `sympy.solvers.ode`), 767  
[allocated\(\)](#) (in module `sympy.codegen.fnodes`), 1153  
[almosteq\(\)](#) (`sympy.polys.domains.ComplexField` method), 2546  
[almosteq\(\)](#) (`sympy.polys.domains.domain.Domain` method), 2508  
[almosteq\(\)](#) (`sympy.polys.domains.RealField` method), 2545  
[almosteq\(\)](#) (`sympy.polys.rings.PolyElement` method), 2554  
[AlmostLinear](#) (class in `sympy.solvers.ode.single`), 790  
[alternating\(\)](#) (`sympy.combinatorics.generators` method), 287  
[AlternatingGroup\(\)](#) (in module `sympy.combinatorics.named_groups`), 354

`altitudes` (`sympy.geometry.polygon.Triangle` property), 2294

`ambient_dimension` (`sympy.geometry.curve.Curve` property), 2248

`ambient_dimension` (`sympy.geometry.entity.GeometryEntity` property), 2194

`ambient_dimension` (`sympy.geometry.line.LinearEntity` property), 2216

`ambient_dimension` (`sympy.geometry.point.Point` property), 2203

`amplitude` (`sympy.physics.optics.waves.TWave` property), 1888

`an` (`sympy.functions.special.hyper.meijerg` property), 524

`analytic_func()` (`sympy.matrices.matrices.MatrixBase` method), 1291

`ancestors()` (`sympy.polys.numberfields.modules.NumberFieldModule` method), 2720

`And` (class in `sympy.logic.boolalg`), 1166

`anf_coeffs()` (in module `sympy.logic.boolalg`), 1183

`ANFform()` (in module `sympy.logic.boolalg`), 1162

`ang_acc_in()` (`sympy.physics.vector.frame.ReferenceFrame` method), 1629

`ang_vel_in()` (`sympy.physics.mechanics.body.Body` method), 1757

`ang_vel_in()` (`sympy.physics.vector.frame.ReferenceFrame` method), 1629

`angle` (`sympy.physics.optics.gaussopt.GeometricRay` property), 1863

`angle()` (`sympy.algebras.Quaternion` method), 887

`angle_between()` (`sympy.geometry.line.LinearEntity` method), 2216

`angle_between()` (`sympy.geometry.plane.Plane` method), 2304

`angle_between()` (`sympy.physics.vector.vector.Vector` method), 1642

`angles` (`sympy.geometry.polygon.Polygon` property), 2274

`angles` (`sympy.geometry.polygon.RegularPolygon` property), 2285

`angular_momentum()` (in module `sympy.physics.mechanics.functions`), 1752

`angular_momentum()` (`sympy.physics.mechanics.particle.Particle` method), 1743

`angular_momentum()` (`sympy.physics.mechanics.rigidbody.RigidBody` method), 1747

`angular_velocity` (`sympy.physics.optics.waves.TWave` property), 1888

`AnnihilateBoson` (class in `sympy.physics.secondquant`), 1542

`AnnihilateFermion` (class in `sympy.physics.secondquant`), 1543

`annotated()` (in module `sympy.printing.pretty.pretty_symbology`), 2186

`ANP` (class in `sympy.polys.polyclasses`), 2568

`AntiCommutator` (class in `sympy.physics.quantum.anticommutator`), 1791

`antidivisor_count()` (in module `sympy.ntheory.factor_`), 1500

`antidivisors()` (in module `sympy.ntheory.factor_`), 1500

`antihermitian`, 925

`AntiHermitianPredicate` (class in `sympy.assumptions.predicates.sets`), 231

`AntiSymmetricTensor` (class in `sympy.physics.secondquant`), 1545

`aother` (`sympy.functions.special.hyper.meijerg` property), 524

`are` (`sympy.functions.special.hyper.hyper` property), 522

`as` (`sympy.functions.special.hyper.meijerg` property), 524

`asPart` (in module `sympy.polys.partfrac`), 2443

`asPart()` (`sympy.core.expr.Expr` method), 947

`apart_list()` (in module `sympy.polys.partfrac`), 2443

`apoapsis` (`sympy.geometry.ellipse.Ellipse` property), 2254

`apothem` (`sympy.geometry.polygon.RegularPolygon` property), 2286

`appellf1` (class in `sympy.functions.special.hyper`), 525

`append()` (`sympy.plotting.plot.Plot` method), 2822

`applied_loads` (`sympy.physics.continuum_mechanics.beam.Beam` property), 1951

`AppliedPredicate` (class in `sympy.assumptions.assume`), 193

`apply()` (`sympy.combinatorics.permutations.Permutation` method), 264

`apply()` (`sympy.printing.pretty.stringpict.prettyForm` static method), 2188

`apply()` (`sympy.simplify.epathtools.EPath`

method), 688  
 apply\_finite\_diff() (in module sympy.calculus.finite\_diff), 240  
 apply\_force() (sympy.physics.mechanics.body.Body method), 887  
 method), 1758  
 apply\_grover() (in module sympy.physics.quantum.grover), 1848  
 apply\_load() (sympy.physics.continuum\_mechanics.beam.Beam method), 1951  
 apply\_load() (sympy.physics.continuum\_mechanics.beam.Beam3D method), 1983  
 apply\_load() (sympy.physics.continuum\_mechanics.truss.Truss method), 2022  
 apply\_moment\_load() (sympy.physics.continuum\_mechanics.beam.Beam method), 1983  
 apply\_operator() (sympy.physics.secondquantization.SecondQuantization method), 1542  
 apply\_operator() (sympy.physics.secondquantization.SecondQuantization method), 1543  
 apply\_operator() (sympy.physics.secondquantization.SecondQuantization method), 1547  
 apply\_operator() (sympy.physics.secondquantization.SecondQuantization method), 1548  
 apply\_operators() (in module sympy.physics.secondquantization), 1561  
 apply\_support() (sympy.physics.continuum\_mechanics.beam.Beam method), 1952  
 apply\_support() (sympy.physics.continuum\_mechanics.beam.Beam3D method), 1983  
 apply\_torque() (sympy.physics.mechanics.body.Body method), 1760  
 applyfunc() (sympy.matrices.common.MatrixCommon method), 1328  
 applyfunc() (sympy.physics.vector.dyadic.Dyadic method), 1648  
 applyfunc() (sympy.physics.vector.vector.Vector method), 1642  
 approximation() (sympy.core.numbers.NumberSymbol method), 993  
 arbitrary\_point() (sympy.geometry.curve.Curve method), 2248  
 arbitrary\_point() (sympy.geometry.ellipse.Ellipse method), 2254  
 arbitrary\_point() (sympy.geometry.line.LinearEntity method), 2217  
 arbitrary\_point() (sympy.geometry.plane.Plane method), 2304  
 arbitrary\_point() (sympy.geometry.polygon.Polygon method), 2275  
 arc\_coplanar() (sympy.algebras.Quaternion method), 887  
 Arcsin() (in module sympy.stats), 2887  
 are\_collinear() (sympy.geometry.point.Point3D static method), 2212  
 are\_concurrent() (sympy.geometry.line.LinearEntity static method), 2218  
 are\_concurrent() (sympy.geometry.plane.Plane static method), 2305  
 are\_coplanar() (sympy.geometry.point.Point static method), 2203  
 are\_similar() (in module sympy.geometry.util), 2199  
 area() (sympy.geometry.ellipse.Ellipse property), 2255  
 area() (sympy.geometry.polygon.Polygon property), 2275  
 area() (sympy.geometry.polygon.RegularPolygon property), 2286  
 create\_beam() (sympy.physics.continuum\_mechanics.beam.Beam property), 1953  
 create\_beam3d() (sympy.physics.continuum\_mechanics.beam.Beam3D property), 1983  
 arg (class in sympy.functions.elementary.complexes), 386  
 assume() (sympy.solvers.solvers.assume.Assume property), 193  
 assume() (sympy.solvers.solvers.assume.Assume property), 928  
 args (sympy.geometry.polygon.RegularPolygon property), 2286  
 args\_cnc() (sympy.core.expr.Expr method), 947  
 Argument (class in sympy.utilities.codegen), 2047  
 argument (sympy.functions.special.bessel.BesselBase property), 497  
 argument (sympy.functions.special.hyper.hyper property), 522  
 argument (sympy.functions.special.hyper.meijerg property), 524  
 arguments (sympy.assumptions.assume.AppliedPredicate property), 194  
 array() (in module sympy.codegen.fnodes), 1153  
 array\_form (sympy.combinatorics.permutations.Permutation property), 264  
 array\_form (sympy.combinatorics.polyhedron.Polyhedron property), 332  
 ArrayConstructor (class in sympy.codegen.fnodes), 1149  
 ArrowStringDescription (class in sympy.categories.diagram\_drawing),

[2756](#)  
`as_AlgebraicField()` (`sympy.polys.domains.gaussiandomains.GaussianRationalField` method), [2538](#)  
`as_base_exp()` (`sympy.core.function.Function` method), [1051](#)  
`as_base_exp()` (`sympy.core.power.Pow` method), [1007](#)  
`as_base_exp()` (`sympy.functions.elementary.exponential.exp` method), [413](#)  
`as_coeff_Add()` (`sympy.core.add.Add` method), [1015](#)  
`as_coeff_add()` (`sympy.core.add.Add` method), [1015](#)  
`as_coeff_Add()` (`sympy.core.expr.Expr` method), [948](#)  
`as_coeff_add()` (`sympy.core.expr.Expr` method), [948](#)  
`as_coeff_Add()` (`sympy.core.numbers.Number` method), [982](#)  
`as_coeff_Add()` (`sympy.core.numbers.Rational` method), [986](#)  
`as_coeff_exponent()` (`sympy.core.expr.Expr` method), [949](#)  
`as_coeff_Mul()` (`sympy.core.expr.Expr` method), [948](#)  
`as_coeff_mul()` (`sympy.core.expr.Expr` method), [949](#)  
`as_coeff_Mul()` (`sympy.core.mul.Mul` method), [1010](#)  
`as_coeff_Mul()` (`sympy.core.numbers.Number` method), [982](#)  
`as_coeff_Mul()` (`sympy.core.numbers.Rational` method), [986](#)  
`as_coeff_Mul()` (`sympy.matrices.expressions.MatrixExpr` method), [1370](#)  
`as_coefficient()` (`sympy.core.expr.Expr` method), [949](#)  
`as_coefficients_dict()` (`sympy.core.expr.Expr` method), [951](#)  
`as_content_primitive()` (`sympy.core.add.Add` method), [1015](#)  
`as_content_primitive()` (`sympy.core.basic.Basic` method), [929](#)  
`as_content_primitive()` (`sympy.core.expr.Expr` method), [951](#)  
`as_content_primitive()` (`sympy.core.mul.Mul` method), [1010](#)  
`as_content_primitive()` (`sympy.core.numbers.Rational` method), [986](#)  
`as_content_primitive()` (`sympy.core.power.Pow` method), [1007](#)  
`as_content_primitive()` (`sympy.functions.elementary.exponential.exp` method), [413](#)  
`as_content_primitive()` (`sympy.polys.polytools.Poly` method), [2382](#)  
`as_dummy()` (`sympy.core.basic.Basic` method), [929](#)  
`as_explicit()` (`sympy.matrices.expressions.MatrixExpr` method), [1370](#)  
`as_expr()` (`sympy.core.expr.Expr` method), [952](#)  
`as_expr()` (`sympy.core.numbers.AlgebraicNumber` method), [990](#)  
`as_expr()` (`sympy.polys.monomials.Monomial` method), [2429](#)  
`as_expr()` (`sympy.polys.numberfields.modules.PowerBasisE` method), [2732](#)  
`as_expr()` (`sympy.polys.polytools.Poly` method), [2383](#)  
`as_expr_set_pairs()` (`sympy.functions.elementary.piecewise.Piecewise` method), [416](#)  
`as_ferrers()` (`sympy.combinatorics.partitions.IntegerParti` method), [254](#)  
`as_finite_difference()` (`sympy.core.function.Derivative` method), [1046](#)  
`as_immutable()` (`sympy.matrices.dense.DenseMatrix` method), [1362](#)  
`as_independent()` (`sympy.core.expr.Expr` method), [952](#)  
`as_int()` (in module `sympy.utilities.misc`), [2110](#)  
`as_leading_term()` (`sympy.core.expr.Expr` method), [955](#)  
`as_list()` (`sympy.polys.polytools.Poly` method), [2383](#)  
`as_mutable()` (`sympy.matrices.dense.DenseMatrix` method), [1362](#)  
`as_mutable()` (`sympy.matrices.expressions.MatrixExpr` method), [1371](#)  
`as_numer_denom()` (`sympy.core.add.Add` method), [1015](#)  
`as_numer_denom()` (`sympy.core.expr.Expr` method), [955](#)  
`as_ordered_factors()` (`sympy.core.expr.Expr` method), [955](#)  
`as_ordered_factors()` (`sympy.core.mul.Mul` method), [1010](#)



`as_ordered_terms()` (`sympy.core.expr.Expr` method), 955  
`as_poly()` (`sympy.core.expr.Expr` method), 956  
`as_poly()` (`sympy.core.numbers.AlgebraicNumber` method), 991  
`as_poly()` (`sympy.core.relational.Equality` method), 1024  
`as_poly()` (`sympy.polys.polytools.Poly` method), 2383  
`as_powers_dict()` (`sympy.core.expr.Expr` method), 956  
`as_real_imag()` (`sympy.core.add.Add` method), 1016  
`as_real_imag()` (`sympy.core.expr.Expr` method), 956  
`as_real_imag()` (`sympy.functions.elementary.complexes.exp` method), 383  
`as_real_imag()` (`sympy.functions.elementary.complexes.exp` method), 383  
`as_real_imag()` (`sympy.functions.elementary.exp` method), 411  
`as_real_imag()` (`sympy.functions.elementary.exp` method), 413  
`as_real_imag()` (`sympy.functions.elementary.hyperbolic` method), 402  
`as_real_imag()` (`sympy.matrices.common.MatrixCommon` method), 1328  
`as_relational()` (`sympy.sets.fancysets.Range` method), 1208  
`as_relational()` (`sympy.sets.sets.Complement` method), 1201  
`as_relational()` (`sympy.sets.sets.FiniteSet` method), 1197  
`as_relational()` (`sympy.sets.sets.Intersection` method), 1199  
`as_relational()` (`sympy.sets.sets.Interval` method), 1195  
`as_relational()` (`sympy.sets.sets.SymmetricDifference` method), 1201  
`as_relational()` (`sympy.sets.sets.Union` method), 1198  
`as_set()` (`sympy.logic.boolalg.Boolean` method), 1163  
`as_set()` (`sympy.logic.boolalg.BooleanFalse` method), 1166  
`as_set()` (`sympy.logic.boolalg.BooleanTrue` method), 1165  
`as_submodule()` (`sympy.polys.numberfields.primetest` method), 2706  
`as_sum()` (`sympy.integrals.integrals.Integral` method), 601  
`as_terms()` (`sympy.core.expr.Expr` method), 957  
`as_two_terms()` (`sympy.core.add.Add` method), 1016  
`as_two_terms()` (`sympy.core.mul.Mul` method), 1011  
`as_cyclic_permutations()` (`sympy.combinatorics.permutations.Permutation` method), 265  
`asec` (`class` in `sympy.functions.elementary.trigonometric`), 398  
`asech` (`class` in `sympy.functions.elementary.hyperbolic`), 407  
`aseries()` (`sympy.core.expr.Expr` method), 957  
`asin` (`class` in `sympy.functions.elementary.trigonometric`), 395  
`asinh` (`class` in `sympy.functions.elementary.hyperbolic`), 405  
`ask` (`class` in `module sympy.assumptions.ask`), 191  
`as_partfrac_list()` (`in module sympy.polys.partfrac`), 2445  
`AssignmentExpr` (`class` in `sympy.codegen.ast`), 1126  
`AssignmentBase` (`class` in `sympy.codegen.ast`), 1127  
`AssignmentError`, 2183  
`assoc_laguerre` (`class` in `sympy.functions.special.polynomials`), 543  
`assoc_legendre` (`class` in `sympy.functions.special.polynomials`), 540  
`assoc_recurrence_memo()` (`in module sympy.utilities.memoization`), 2110  
`assuming()` (`in module sympy.assumptions.assume`), 197  
`AssumptionKeys` (`class` in `sympy.assumptions.ask`), 191  
`assumptions0` (`sympy.core.basic.Basic` property), 929  
`AssumptionsContext` (`class` in `sympy.assumptions.assume`), 194  
`atan` (`class` in `sympy.functions.elementary.trigonometric`), 397  
`atan2` (`class` in `sympy.functions.elementary.trigonometric`), 400  
`atanh` (`class` in `sympy.functions.elementary.hyperbolic`), 406  
`Atom` (`class` in `sympy.core.basic`), 945  
`AtomExpr` (`class` in `sympy.core.expr`), 976  
`atoms()` (`sympy.combinatorics.permutations.Permutation` method), 265  
`atoms()` (`sympy.core.basic.Basic` method), 930  
`atoms()` (`sympy.matrices.common.MatrixCommon`

method), 1328  
 atoms\_table (in module `sympy.printing.pretty.pretty_symbology`), 2186  
 attempted (sympy.testing.runtests.SymPyTestRunner attribute), 2031  
 attr\_params() (sympy.codegen.ast.Node method), 1135  
 Attribute (class in sympy.codegen.ast), 1127  
 aug\_assign() (in module `sympy.codegen.ast`), 1142  
 AugmentedAssignment (class in `sympy.codegen.ast`), 1127  
 auto\_number() (in module `sympy.parsing.sympy_parser`), 2129  
 auto\_symbol() (in module `sympy.parsing.sympy_parser`), 2128  
 autowrap() (in module `sympy.utilities.autowrap`), 2041  
 auxiliary\_circle() (sympy.geometry.ellipse.Ellipse method), 2255  
 auxiliary\_eqs (sympy.physics.mechanics.kane.KanesMethod property), 1765  
 axial\_force() (sympy.physics.continuum\_mechanics.beam3D.Beam3D method), 1983  
 axial\_stress() (sympy.physics.continuum\_mechanics.beam3D.Beam3D method), 1983  
 axis() (sympy.algebras.Quaternion method), 888  
 AxisOrienter (class in `sympy.vector.orienters`), 1466  
 AZ() (in module `sympy.crypto.crypto`), 2762  
**B**  
 B (in module `sympy.physics.secondquant`), 1546  
 B (sympy.physics.optics.gaussopt.RayTransferMatrix property), 1865  
 b\_interval (sympy.sets.fancysets.ComplexRegion property), 1210  
 banded() (in module `sympy.matrices.sparsetools`), 1366  
 base (sympy.combinatorics.perm\_groups.PermutationGroup property), 295  
 base (sympy.functions.elementary.exponential.exp property), 412  
 base (sympy.tensor.indexed.Indexed property), 1402  
 base\_oneform() (sympy.diffgeom.CoordSystem method), 2803  
 base\_oneforms() (sympy.diffgeom.CoordSystem method), 2803  
 base\_scalar() (sympy.diffgeom.CoordSystem method), 2803  
 base\_scalars() (sympy.diffgeom.CoordSystem method), 2803  
 base\_resolution\_linear() (in module `sympy.solvers.diophantine.diophantine`), 723  
 base\_vector() (sympy.diffgeom.CoordSystem method), 2803  
 base\_vectors() (sympy.diffgeom.CoordSystem method), 2803  
 BaseBackend (class in `sympy.plotting.plot`), 2868  
 BaseCovarDerivativeOp (class in `sympy.diffgeom`), 2813  
 BasePolynomialError (class in `sympy.polys.polyerrors`), 2643  
 BaseScalarField (class in `sympy.diffgeom`), 2807  
 BaseSeries (class in `sympy.plotting.plot`), 2865  
 baseswap() (sympy.combinatorics.perm\_groups.PermutationGroup method), 295  
 BaseVectorField (class in `sympy.diffgeom`), 2808  
 Basic (class in `sympy.core.basic`), 927  
 basic\_orbits (sympy.combinatorics.perm\_groups.PermutationGroup property), 297  
 basic\_root() (sympy.liealgebras.type\_a.TypeA method), 2328  
 basic\_root() (sympy.liealgebras.type\_b.TypeB method), 2330  
 basic\_root() (sympy.liealgebras.type\_c.TypeC method), 2331  
 basic\_root() (sympy.liealgebras.type\_d.TypeD method), 2333  
 basic\_root() (sympy.liealgebras.type\_e.TypeE method), 2334  
 basic\_root() (sympy.liealgebras.type\_f.TypeF method), 2336  
 basic\_stabilizers (sympy.combinatorics.perm\_groups.PermutationGroup property), 297  
 basic\_transversals (sympy.combinatorics.perm\_groups.PermutationGroup property), 298  
 basis() (sympy.liealgebras.type\_a.TypeA method), 2328  
 basis() (sympy.liealgebras.type\_b.TypeB method), 2330  
 basis() (sympy.liealgebras.type\_c.TypeC method), 2331  
 basis() (sympy.liealgebras.type\_d.TypeD method), 2333

**basis()** (`sympy.liealgebras.type_e.TypeE` `sympy.simplify.simplify`), 666  
 method), 2334  
**basis()** (`sympy.liealgebras.type_f.TypeF` `sympy.functions.special.bessel`),  
 method), 2336 498  
**basis()** (`sympy.liealgebras.type_g.TypeG` `beta` (class in `sympy.functions.special.beta_functions`),  
 method), 2337 470  
**basis()** (`sympy.polys.agca.modules.FreeModule` `Beta()` (in module `sympy.stats`), 2889  
 method), 2454 `BetaBinomial()` (in module `sympy.stats`),  
**basis\_element\_pullbacks()** `BetaBinomial()` (in module `sympy.stats`),  
 (`sympy.polys.numberfields.modules.Submodule` `BetaBinomial()` (in module `sympy.stats`),  
 method), 2727 2890  
**basis\_elements()** (`sympy.polys.numberfields.modules.Submodule` `BetaBinomial()` (in module `sympy.stats`), 2891  
 method), 2720 `bidiagonal_decomposition()`  
**BBra** (in module `sympy.physics.secondquant`), `(sympy.matrices.matrices.MatrixEigen`  
 1546 method), 1237  
**Bd** (in module `sympy.physics.secondquant`), `bidiagonalize()` (`sympy.matrices.matrices.MatrixEigen`  
 1546 method), 1238  
**Beam** (class in `sympy.physics.continuum_mechanics.beams`), `binomial` (in module  
 1949 `sympy.crypto.crypto`), 2776  
**Beam3D** (class in `sympy.physics.continuum_mechanics.beams`), `binomial` (in module  
 1981 `sympy.crypto.crypto`), 2779  
**BeamParameter** (class in `sympy.physics.optics.gaussopt`), `bin_to_gray()` (`sympy.combinatorics.graycode`  
 1857 method), 351  
**bell** (class in `sympy.functions.combinatorial.numbers`), `binary_function()` (in module  
 427 `sympy.utilities.autowrap`), 2042  
**below()** (`sympy.printing.pretty.stringpict.stringPict` `binary_partitions()` (in module  
 method), 2186 `sympy.utilities.iterables`), 2071  
**bending\_moment()** (`sympy.physics.continuum_mechanics.beams` `BinaryQuadratic` (class in  
 method), 1953 `sympy.physics.diophantine.diophantine`),  
**bending\_moment()** (`sympy.physics.continuum_mechanics.beams` `binomial` (class in  
 method), 1983 `sympy.functions.combinatorial.factorials`),  
**Benini()** (in module `sympy.stats`), 2888 `Binomial()` (in module `sympy.stats`), 2877  
**bernoulli** (class in `sympy.functions.combinatorial.numbers`), 430  
**Bernoulli** (class in `sympy.solvers.ode.single`), 778  
**Bernoulli()** (in module `sympy.stats`), 2876 `binomial_coefficients()` (in module  
**BernoulliProcess** (class in `sympy.stats`), `sympy.ntheory.multinomial`), 1511  
 2953 `binomial_coefficients_list()` (in module  
**BesselBase** (class in `sympy.functions.special.bessel`), `sympy.ntheory.multinomial`), 1512  
 497 `bisectors()` (`sympy.geometry.line.LinearEntity`  
**besseli** (class in `sympy.functions.special.bessel`), `method`), 2219  
 499 `bisectors()` (`sympy.geometry.polygon.Polygon`  
**besselj** (class in `sympy.functions.special.bessel`), `method`), 2276  
 497 `bisectors()` (`sympy.geometry.polygon.Triangle`  
**besselk** (class in `sympy.functions.special.bessel`), `method`), 2294  
 499 `bitlist_from_subset()`  
**besselsimp()** (in module `sympy.codegen.fnodes`),  
 1385 `BKet` (in module `sympy.physics.secondquant`),  
 1546 `block_collapse()` (in module  
`sympy.matrices.expressions.blockmatrix`),  
 1385

BlockDiagMatrix (class in property), 1040  
sympy.matrices.expressions.blockmatrix), 1385  
bound\_symbols (sympy.core.function.Subs property), 1052  
BlockMatrix (class in boundary (sympy.sets.sets.Set property), 1185  
sympy.matrices.expressions.blockmatrix), 1380  
boundary\_conditions  
bm (sympy.functions.special.hyper.meijerg (sympy.physics.continuum\_mechanics.beam.Beam property), 525  
property), 1954  
bode\_magnitude\_numerical\_data() boundary\_conditions  
(sympy.physics.control.control\_plots (sympy.physics.continuum\_mechanics.beam.Beam3 method), 1938  
property), 1983  
bode\_magnitude\_plot() BoundedPareto() (in module sympy.stats),  
(sympy.physics.control.control\_plots 2892  
method), 1936  
bounds (sympy.geometry.ellipse.Ellipse prop-  
erty), 2255  
bode\_phase\_numerical\_data() bounds (sympy.geometry.entity.GeometryEntity  
(sympy.physics.control.control\_plots property), 2194  
method), 1939  
bounds (sympy.geometry.line.LinearEntity2D  
property), 2237  
bode\_phase\_plot() bounds (sympy.geometry.point.Point2D prop-  
(sympy.physics.control.control\_plots erty), 2209  
method), 1938  
bode\_plot() (sympy.physics.control.control\_plots bounds (sympy.geometry.polygon.Polygon  
method), 1936  
property), 2276  
bodies (sympy.physics.mechanics.jointsmethod JointsMethod property), 1781  
bq (sympy.functions.special.hyper.hyper  
property), 522  
bodies (sympy.physics.mechanics.system.SymbolicSystem property), 1786  
bq (sympy.functions.special.hyper.meijerg  
property), 525  
Body (class in sympy.physics.mechanics.body), 1756  
Bra (class in sympy.physics.quantum.state), 1832  
BodyOrienter (class in sympy.vector.orienters), 1467  
bra (sympy.physics.quantum.operator.OuterProduct  
property), 1812  
bool\_map() (in module sympy.logic.boolalg), 1177  
bra (sympy.physics.secondquant.InnerProduct  
property), 1553  
bool\_maxterm() (in module BraBase (class in  
sympy.logic.boolalg), 1181  
sympy.physics.quantum.state), 1833  
bool\_minterm() (in module bracelets() (in  
sympy.logic.boolalg), 1182  
module  
sympy.utilities.iterables), 2071  
bool\_monomial() (in module BreakToken (class in sympy.codegen.ast),  
sympy.logic.boolalg), 1182  
1127  
Boolean (class in sympy.logic.boolalg), 1163  
brewster\_angle() (in module  
sympy.physics.optics.utils), 1880  
BooleanFalse (class in sympy.logic.boolalg), 1165  
bsgs\_direct\_product() (in module  
sympy.combinatorics.tensor\_can),  
1370  
BooleanKind (in module sympy.core.kind), 1074  
bspline\_basis() (in module  
sympy.functions.special.bsplines),  
511  
BosonicBasis (class in sympy.physics.secondquant), 1546  
bspline\_basis\_set() (in module  
sympy.functions.special.bsplines),  
512  
bother (sympy.functions.special.hyper.meijerg property), 525  
build\_expression\_tree() (in module  
sympy.integrations.gruntz), 623  
bottom\_up() (in module sympy.core.traversal), 1080  
build\_options() (in module  
sympy.polys.polyoptions), 2643  
bound\_symbols (sympy.concrete.expr\_with\_limits.ExprWithLimits property), 606  
bound\_symbols (sympy.core.function.Lambda



## C

- C (sympy.matrices.common.MatrixCommon property), 1327
- C (sympy.physics.optics.gaussopt.RayTransferMatrix property), 1865
- C89CodePrinter (class in sympy.printing.c), 2141
- C99CodePrinter (class in sympy.printing.c), 2141
- calculate\_series() (in module sympy.series.gruntz), 624
- can\_transf\_matrix (sympy.physics.units.dimensions.DimensionSystem property), 1585
- canberra\_distance() (sympy.geometry.point.Point method), 2203
- cancel() (in module sympy.polys.polytools), 2376
- cancel() (sympy.core.expr.Expr method), 958
- cancel() (sympy.polys.polyclasses.DMF method), 2567
- cancel() (sympy.polys.polyclasses.DMP method), 2562
- cancel() (sympy.polys.polytools.Poly method), 2383
- cancel() (sympy.polys.rings.PolyElement method), 2554
- canon\_bp() (in module sympy.tensor.tensor), 1419
- canon\_bp() (sympy.tensor.tensor.TensAdd method), 1416
- canon\_bp() (sympy.tensor.tensor.TensMul method), 1417
- canonical (sympy.core.relational.Relational property), 1020
- canonical\_form() (sympy.stats.DiscreteMarkovChain method), 2946
- canonical\_odes() (in module sympy.solvers.ode.systems), 809
- canonical\_variables (sympy.core.basic.Basic property), 931
- canonicalize() (in module sympy.combinatorics.tensor\_can), 365
- capture() (in module sympy.utilities.iterables), 2071
- cardinality (sympy.combinatorics.permutations.Permutation property), 265
- cardinality (sympy.combinatorics.subsets.Subset property), 340
- cartan\_matrix() (sympy.liealgebras.root\_system.RootSystem method), 2327
- cartan\_matrix() (sympy.liealgebras.type\_a.TypeA method), 2328
- cartan\_matrix() (sympy.liealgebras.type\_b.TypeB method), 2330
- cartan\_matrix() (sympy.liealgebras.type\_c.TypeC method), 2331
- cartan\_matrix() (sympy.liealgebras.type\_d.TypeD method), 2333
- cartan\_matrix() (sympy.liealgebras.type\_e.TypeE method), 2334
- cartan\_matrix() (sympy.liealgebras.type\_f.TypeF method), 2336
- cartan\_matrix() (sympy.liealgebras.type\_g.TypeG method), 2337
- CartanMatrix() (in module sympy.liealgebras.cartan\_matrix), 2341
- CartanType\_generator (class in sympy.liealgebras.cartan\_type), 2340
- CartesianComplexRegion (class in sympy.sets.fancysets), 1212
- casoratian() (in module sympy.matrices.dense), 1322
- cast\_check() (sympy.codegen.ast.Type method), 1139
- cast\_nocheck (sympy.codegen.ast.FloatBaseType attribute), 1131
- cast\_nocheck() (sympy.codegen.ast.FloatType method), 1131
- Catalan (class in sympy.core.numbers), 1002
- catalan (class in sympy.functions.combinatorial.numbers), 431
- Category (class in sympy.categories), 2747
- Cauchy() (in module sympy.stats), 2893
- cauchy\_lower\_bound() (sympy.polys.polyclasses.DMP method), 2562
- cauchy\_upper\_bound() (sympy.polys.polyclasses.DMP method), 2562
- Cbrt (class in sympy.codegen.cfunctions), 1143
- cbrt() (in module sympy.functions.elementary.miscellaneous), 425
- ccode() (in module sympy.printing.c), 2141
- CCodeGen (class in sympy.utilities.codegen), 2047
- ceiling (class in sympy.functions.elementary.integers), 409

- center (*sympy.geometry.ellipse.Ellipse* property), 2255
- center (*sympy.geometry.polygon.RegularPolygon* property), 2287
- center() (*sympy.combinatorics.perm\_groups.PermutationGroup* method), 298
- central\_inertia (*sympy.physics.mechanics.rigidbody.RigidBody* property), 1748
- centralizer() (*sympy.combinatorics.perm\_groups.PermutationGroup* method), 299
- CentralMoment (class in *sympy.stats*), 2977
- centroid (*sympy.geometry.polygon.Polygon* property), 2276
- centroid (*sympy.geometry.polygon.RegularPolygon* property), 2287
- centroid() (in module *sympy.geometry.util*), 2200
- CG (class in *sympy.physics.quantum.cg*), 1792
- cg\_simp() (in module *sympy.physics.quantum.cg*), 1794
- CGate (class in *sympy.physics.quantum.gate*), 1842
- CGateS (class in *sympy.physics.quantum.gate*), 1843
- change\_index() (*sympy.concrete.expr\_with\_intlimits.ExprWithIntLimits* method), 911
- change\_member\_label() (*sympy.physics.continuum\_mechanics.truss.Truss* method), 2023
- change\_node\_label() (*sympy.physics.continuum\_mechanics.truss.Truss* method), 2024
- characteristic() (*sympy.polys.domains.domainelement.DomainElement* method), 2508
- characteristic() (*sympy.polys.domains.finitefield.FiniteField* method), 2524
- charpoly() (*sympy.matrices.matrices.MatrixDeterminant* method), 1228
- charpoly() (*sympy.polys.matrices.ddm.DDM* method), 2690
- charpoly() (*sympy.polys.matrices.domainmatrix.DomainMatrix* method), 2672
- charpoly() (*sympy.polys.matrices.sdm.SDM* method), 2692
- chebyshevt (class in *sympy.functions.special.polynomials*), 536
- chebyshevt\_poly() (in module *sympy.polys.orthopolys*), 2439
- chebyshevt\_root (class in *sympy.functions.special.polynomials*), 539
- chebyshevu (class in *sympy.functions.special.polynomials*), 538
- chebyshevu\_poly() (in module *sympy.polys.orthopolys*), 2439
- chebyshevu\_root (class in *sympy.functions.special.polynomials*), 539
- check\_output() (*sympy.testing.runttests.SymPyOutputChecker* method), 2031
- checkinfsol() (in module *sympy.solvers.ode*), 765
- checkodesol() (in module *sympy.solvers.ode*), 762
- checkpdesol() (in module *sympy.solvers.pde*), 830
- checksol() (in module *sympy.solvers.solvers*), 850
- Chi (class in *sympy.functions.special.error\_functions*), 495
- Chi() (in module *sympy.stats*), 2893
- child (*sympy.physics.mechanics.joint.Joint* property), 1772
- child\_axis (*sympy.physics.mechanics.joint.Joint* property), 1772
- child\_point (*sympy.physics.mechanics.joint.Joint* property), 1772
- Chisquare (class in *sympy.stats*), 2894
- Chisquared() (in module *sympy.stats*), 2895
- cholsky() (*sympy.matrices.dense.DenseMatrix* method), 1362
- cholsky() (*sympy.matrices.matrices.MatrixBase* method), 1291
- cholsky\_solve() (*sympy.matrices.matrices.MatrixBase* method), 1292
- ChiSquared() (class in *sympy.functions.special.error\_functions*), 492
- Circle (class in *sympy.geometry.ellipse*), 2269
- CircuitPlot (class in *sympy.physics.quantum.circuitplot*), 1841
- CircuitPlot (class in *sympy.physics.quantum.circuitplot*), 1840
- circumcenter (*sympy.geometry.polygon.RegularPolygon* property), 2287
- circumcenter (*sympy.geometry.polygon.Triangle* property), 2295
- circumcircle (*sympy.geometry.polygon.RegularPolygon* property), 2288
- circumcircle (*sympy.geometry.polygon.Triangle* property), 2295

circumference (sympy.geometry.ellipse.Circle property), 2270  
 circumference (sympy.geometry.ellipse.Ellipse property), 2256  
 circumradius (sympy.geometry.polygon.RegularPolygon property), 2288  
 circumradius (sympy.geometry.polygon.Triangle property), 2296  
 class\_key() (sympy.core.add.Add class method), 1016  
 class\_key() (sympy.core.basic.Basic class method), 931  
 classify\_diop() (in module sympy.solvers.diophantine.diophantine), 721  
 classify\_ode() (in module sympy.solvers.ode), 760  
 classify\_pde() (in module sympy.solvers.pde), 829  
 clear() (sympy.tensor.tensor.\_TensorManager method), 1422  
 clear\_cache() (sympy.polys.rootoftools.ComplexRootOf class method), 2435  
 clear\_denoms() (sympy.polys.polyclasses.DMPcoefficients (sympy.geometry.line.Line2D property), 2239  
 clear\_denoms() (sympy.polys.polytools.Poly method), 2384  
 clear\_loads() (sympy.physics.mechanics.body.Dof method), 1761  
 clebsch\_gordan() (in module sympy.physics.wigner), 1567  
 clone() (sympy.polys.polyoptions.Options method), 2643  
 closing\_angle() (sympy.geometry.line.Ray2D method), 2240  
 closure (sympy.sets.sets.Set property), 1186  
 CMod (class in sympy.physics.quantum.shor), 1856  
 cmoment() (in module sympy.stats), 2978  
 cmplx (class in sympy.codegen.fnodes), 1154  
 CNOT (in module sympy.physics.quantum.gate), 1843  
 CNotGate (class in sympy.physics.quantum.gate), 1843  
 CodeBlock (class in sympy.codegen.ast), 1128  
 CodeGen (class in sympy.utilities.codegen), 2048  
 codegen() (in module sympy.utilities.codegen), 2053  
 CodePrinter (class in sympy.printing.codeprinter), 2183  
 CodeWrapper (class in sympy.utilities.autowrap), 2040  
 codomain (sympy.categories.CompositeMorphism property), 2745  
 codomain (sympy.categories.Morphism property), 2743  
 coeff() (sympy.core.expr.Expr method), 958  
 coeff() (sympy.polys.rings.PolyElement method), 2555  
 coeff() (sympy.series.sequences.SeqBase method), 632  
 coeff\_bell() (sympy.series.formal.FormalPowerSeries method), 647  
 coeff\_monomial() (sympy.polys.polytools.Poly method), 2384  
 coeff\_mul() (sympy.series.sequences.EmptySequence method), 636  
 coeff\_mul() (sympy.series.sequences.SeqBase method), 632  
 coeff\_mul() (sympy.series.sequences.SeqFormula method), 635  
 coeff\_mul() (sympy.series.sequences.SeqPer method), 636  
 coeff\_search() (in module sympy.polys.numberfields.utilities), 2739  
 coefficients (sympy.geometry.line.Line2D property), 2239  
 coeffs() (sympy.core.numbers.AlgebraicNumber method), 991  
 coeffs() (sympy.polys.polyclasses.DMP method), 2562  
 coeffs() (sympy.polys.polytools.Poly method), 2385  
 coeffs() (sympy.polys.rings.PolyElement method), 2555  
 CoercionFailed (class in sympy.polys.polyerrors), 2644  
 cofactor() (sympy.matrices.matrices.MatrixDeterminant method), 1229  
 cofactor\_matrix() (sympy.matrices.matrices.MatrixDeterminant method), 1229  
 cofactors() (in module sympy.polys.polytools), 2367  
 cofactors() (sympy.core.numbers.Number method), 982  
 cofactors() (sympy.polys.domains.domain.Domain method), 2508  
 cofactors() (sympy.polys.polyclasses.DMP method), 2562  
 cofactors() (sympy.polys.polytools.Poly method), 2385  
 coherent\_state() (in module sympy.physics.qho\_1d), 1540  
 Coin() (in module sympy.stats), 2877  
 col() (sympy.matrices.common.MatrixCommon

- method), 1329
- col\_del() (sympy.matrices.common.MatrixCommon method), 1329
- col\_insert() (sympy.matrices.common.MatrixCommon method), 1329
- col\_join() (sympy.matrices.common.MatrixCommon method), 1329
- collect() (in module sympy.simplify.radsimp), 672
- collect() (sympy.core.expr.Expr method), 961
- collect\_const() (in module sympy.simplify.radsimp), 676
- collect\_sqrt() (in module sympy.simplify.radsimp), 675
- column() (sympy.polys.numberfields.modules.MatrixModule method), 2730
- columnspace() (sympy.matrices.matrices.MatrixModule method), 1235
- columnspace() (sympy.polys.matrices.domainmodule.MatrixModule method), 2673
- comb\_explicit\_rhs (sympy.physics.mechanics.system.SymbolicSystem property), 1786
- comb\_implicit\_mat (sympy.physics.mechanics.system.SymbolicSystem property), 1787
- comb\_implicit\_rhs (sympy.physics.mechanics.system.SymbolicSystem property), 1787
- combsimp() (in module sympy.simplify.combsimp), 683
- combsimp() (sympy.core.expr.Expr method), 961
- comm\_i2symbol() (sympy.tensor.tensor.\_TensorManager method), 1422
- comm\_symbols2i() (sympy.tensor.tensor.\_TensorManager method), 1422
- CommaOperator (class in sympy.codegen.cnodes), 1148
- Comment (class in sympy.codegen.ast), 1129
- common\_prefix() (in module sympy.utilities.iterables), 2072
- common\_suffix() (in module sympy.utilities.iterables), 2072
- communication\_classes() (sympy.stats.DiscreteMarkovChain method), 2948
- commutative, 924
- commutative\_diagrams (sympy.categories.Category property), 2748
- CommutativePredicate (class in sympy.assumptions.predicates.common), 202
- Commutator (class in sympy.diffgeom), 2809
- Commutator (class in sympy.physics.quantum.commutator), 1795
- Commutator (class in sympy.physics.secondquant), 1546
- commutator() (sympy.combinatorics.perm\_groups.Permutation method), 300
- commutator() (sympy.combinatorics.permutations.Permutation method), 265
- commutes\_with() (sympy.combinatorics.permutations.Permutation method), 266
- commutes\_with() (sympy.tensor.tensor.TensorHead method), 1413
- ComplexElement (sympy.matrices.common.MatrixCommon class method), 1330
- ComplexMatrix (class in sympy.matrices.expressions), 1379
- ComplexDomainModule (sympy.series.gruntz), 623
- compare() (sympy.core.basic.Basic method), 924
- Complement (class in sympy.sets.sets), 1200
- complement() (sympy.sets.sets.Set method), 1200
- complex, 924
- ComplexElement (class in sympy.polys.domains.mpelements), 2546
- ComplexElementsPredicate (class in sympy.assumptions.predicates.matrices), 213
- Complexes (class in sympy.sets.fancysets), 1206
- ComplexField (class in sympy.polys.domains), 2546
- ComplexInfinity (class in sympy.core.numbers), 999
- ComplexPredicate (class in sympy.assumptions.predicates.sets), 229
- ComplexRegion (class in sympy.sets.fancysets), 1209
- ComplexRootOf (class in sympy.polys.rootoftools), 2431
- ComplexSpace (class in sympy.physics.quantum.hilbert), 1802
- ComplexType (class in sympy.codegen.ast), 1129
- components (sympy.categories.CompositeMorphism property), 2745
- components (sympy.vector.dyadic.Dyadic



property), 1458  
 components (sympy.vector.vector.Vector property), 1455  
 components() (in module sympy.integrals.heurisch), 598  
 compose() (in module sympy.polys.polytools), 2370  
 compose() (sympy.categories.Morphism method), 2743  
 compose() (sympy.polys.polyclasses.DMP method), 2562  
 compose() (sympy.polys.polytools.Poly method), 2385  
 compose() (sympy.polys.rings.PolyRing method), 2554  
 compose() (sympy.series.formal.FormalPowerSeries method), 647  
 composite, 924  
 composite() (in module sympy.ntheory.generate), 1485  
 CompositeDomain (class in sympy.polys.domains.compositedomain), 2522  
 CompositeMorphism (class in sympy.categories), 2744  
 compositepi() (in module sympy.ntheory.generate), 1486  
 CompositePredicate (class in sympy.assumptions.predicates.ntheory), 218  
 composition() (sympy.holonomic.holonomic.HolonomicSymbol method), 2318  
 composition\_series() (sympy.combinatorics.perm\_groups.PermutationGroup method), 300  
 CompoundDistribution (class in sympy.stats.compound\_rv), 2959  
 ComputationFailed (class in sympy.polys.polyerrors), 2644  
 compute\_explicit\_form() (sympy.physics.mechanics.system.SymbolicSystem method), 1787  
 compute\_fps() (in module sympy.series.formal), 652  
 compute\_leading\_term() (sympy.core.expr.Expr method), 961  
 compute\_m\_ybar() (in module sympy.solvers.ode.riccati), 805  
 conclusions (sympy.categories.Diagram property), 2750  
 cond (sympy.functions.elementary.piecewise.Expression property), 415  
 condition\_number() (sympy.matrices.matrices.MatrixBase method), 1293  
 ConditionalDomain (class in sympy.stats.rv), 2981  
 ConditionSet (class in sympy.sets.conditionset), 1215  
 conjugacy\_class() (sympy.combinatorics.perm\_groups.PermutationGroup method), 301  
 conjugacy\_classes() (sympy.combinatorics.perm\_groups.PermutationGroup method), 302  
 conjugate (class in sympy.functions.elementary.complexes), 386  
 conjugate (sympy.combinatorics.partitions.IntegerPartition property), 254  
 conjugate() (sympy.core.expr.Expr method), 961  
 conjugate() (sympy.matrices.common.MatrixCommon method), 1330  
 conjugate\_gauss\_beams() (in module sympy.physics.optics.gaussopt), 1866  
 connected\_components() (in module sympy.utilities.iterables), 2072  
 connected\_components() (sympy.matrices.matrices.MatrixBase method), 1293  
 connected\_components\_decomposition() (sympy.matrices.matrices.MatrixBase method), 1294  
 conserve\_mpmath\_dps() (in module sympy.utilities.decorator), 2058  
 const() (sympy.polys.rings.PolyElement method), 2555  
 constant\_renumber() (in module sympy.solvers.ode.ode), 768  
 constant\_symbols() (sympy.physics.mechanics.system.SymbolicSystem method), 1787  
 constantsimp() (in module sympy.solvers.ode), 765  
 construct\_c() (in module sympy.solvers.ode.riccati), 803  
 construct\_d() (in module sympy.solvers.ode.riccati), 803  
 construct\_domain() (in module sympy.polys.constructor), 2427  
 Contains (class in sympy.sets.conditionset), 1216  
 contains() (sympy.combinatorics.perm\_groups.PermutationGroup method), 302  
 contains() (sympy.geometry.line.Line

*method*), 2228  
`contains()` (*sympy.geometry.line.LinearEntity*  
*method*), 2219  
`contains()` (*sympy.geometry.line.Ray*  
*method*), 2231  
`contains()` (*sympy.geometry.line.Segment*  
*method*), 2234  
`contains()` (*sympy.polys.agca.ideals.Ideal*  
*method*), 2462  
`contains()` (*sympy.polys.agca.modules.Module*  
*method*), 2453  
`contains()` (*sympy.polys.polytools.GroebnerBasis*  
*method*), 2423  
`contains()` (*sympy.series.order.Order*  
*method*), 629  
`contains()` (*sympy.sets.sets.Set* *method*),  
1186  
`content()` (*in module sympy.polys.polytools*),  
2369  
`content()` (*sympy.polys.polyclasses.DMP*  
*method*), 2562  
`content()` (*sympy.polys.polytools.Poly*  
*method*), 2386  
`content()` (*sympy.polys.rings.PolyElement*  
*method*), 2556  
`continued_fraction()` (*in module*  
*sympy.ntheory.continued\_fraction*),  
1523  
`continued_fraction_convergents()`  
(*in module*  
*sympy.ntheory.continued\_fraction*),  
1523  
`continued_fraction_iterator()` (*in module*  
*sympy.ntheory.continued\_fraction*),  
1524  
`continued_fraction_periodic()` (*in module*  
*sympy.ntheory.continued\_fraction*),  
1525  
`continued_fraction_reduce()` (*in module*  
*sympy.ntheory.continued\_fraction*),  
1526  
`ContinueToken` (*class in sympy.codegen.ast*),  
1130  
`continuous_domain()` (*in module*  
*sympy.calculus.util*), 244  
`ContinuousDomain` (*class in sympy.stats.crv*),  
2981  
`ContinuousMarkovChain` (*class in*  
*sympy.stats*), 2951  
`ContinuousPSpace` (*class in sympy.stats.crv*),  
2981  
`ContinuousRV()` (*in module sympy.stats*),  
2934  
`contract_metric()` (*sympy.tensor.tensor.TensAdd*  
*method*), 1416  
`contract_metric()`  
(*sympy.tensor.tensor.TensMul*  
*method*), 1417  
`contraction()` (*in module*  
*sympy.physics.secondquant*), 1562  
`control_line()` (*sympy.physics.quantum.circuitplot.Circuit*  
*method*), 1840  
`control_point()` (*sympy.physics.quantum.circuitplot.Circuit*  
*method*), 1841  
`controls` (*sympy.physics.quantum.gate.CGate*  
*property*), 1842  
`controls` (*sympy.physics.quantum.gate.CNotGate*  
*property*), 1843  
`convergence_statement`  
(*sympy.functions.special.hyper.hyper*  
*property*), 522  
`convert()` (*sympy.polys.agca.modules.FreeModule*  
*method*), 2454  
`convert()` (*sympy.polys.agca.modules.Module*  
*method*), 2453  
`convert()` (*sympy.polys.agca.modules.QuotientModule*  
*method*), 2465  
`convert()` (*sympy.polys.agca.modules.SubModule*  
*method*), 2456  
`convert()` (*sympy.polys.domains.domain.Domain*  
*method*), 2508  
`convert()` (*sympy.polys.polyclasses.DMP*  
*method*), 2562  
`convert_from()` (*sympy.polys.domains.domain.Domain*  
*method*), 2508  
`convert_to()` (*in module*  
*sympy.physics.units.util*), 1588  
`convert_to()` (*sympy.physics.units.quantities.Quantity*  
*method*), 1587  
`convert_to()` (*sympy.polys.matrices.domainmatrix.Domain*  
*method*), 2673  
`convert_to()` (*sympy.polys.matrices.sdm.SDM*  
*method*), 2693  
`convert_to_c()` (*sympy.parsing.sym\_expr.SymPyExpression*  
*method*), 2131  
`convert_to_expr()`  
(*sympy.parsing.sym\_expr.SymPyExpression*  
*method*), 2132  
`convert_to_fortran()`  
(*sympy.parsing.sym\_expr.SymPyExpression*  
*method*), 2133  
`convert_to_native_paths()` (*in module*  
*sympy.testing.runtests*), 2031  
`convert_to_python()`  
(*sympy.parsing.sym\_expr.SymPyExpression*  
*method*), 2133  
`convert_xor()` (*in module*

`sympy.parsing.sympy_parser`), 2128  
`convex_hull()` (in module `sympy.geometry.util`), 2198  
`convolution()` (in module `sympy.discrete.convolutions`), 1089  
`convolution_fft()` (in module `sympy.discrete.convolutions`), 1091  
`convolution_fwht()` (in module `sympy.discrete.convolutions`), 1092  
`convolution_ntt()` (in module `sympy.discrete.convolutions`), 1091  
`convolution_subset()` (in module `sympy.discrete.convolutions`), 1093  
`coord_function()` (`sympy.diffgeom.CoordSystem` method), 2803  
`coord_functions()` (`sympy.diffgeom.CoordSystem` method), 2803  
`coord_tuple_transform_to()` (`sympy.diffgeom.CoordSystem` method), 2803  
`coordinates` (`sympy.geometry.point.Point2D` property), 2209  
`coordinates` (`sympy.geometry.point.Point3D` property), 2212  
`coordinates` (`sympy.physics.mechanics.joint.Joint` property), 1772  
`coordinates` (`sympy.physics.mechanics.system.SymbolicSystem` property), 1787  
`CoordinateSym` (class in `sympy.physics.vector.frame`), 1628  
`CoordinateSymbol` (class in `sympy.diffgeom`), 2805  
`coords()` (`sympy.diffgeom.Point` method), 2807  
`CoordSys3D` (class in `sympy.vector.coordsysrect`), 1447  
`CoordSystem` (class in `sympy.diffgeom`), 2801  
`copy()` (`sympy.matrices.matrices.MatrixBase` method), 1295  
`copy()` (`sympy.polys.matrices.sdm.SDM` method), 2693  
`copy()` (`sympy.polys.rings.PolyElement` method), 2556  
`copy()` (`sympy.series.gruntz.SubsSet` method), 625  
`core()` (in module `sympy.ntheory.factor_`), 1504  
`cornacchia()` (in module `sympy.solvers.diophantine.diophantine`), 725  
`corners` (`sympy.combinatorics.polyhedron.Polyhedron` property), 333  
`correlation()` (in module `sympy.stats`), 2975  
`cos` (class in `sympy.functions.elementary.trigonometric`), 390  
`coset_factor()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 303  
`coset_rank()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 304  
`coset_table()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 305  
`coset_transversal()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 305  
`coset_unrank()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 305  
`cosh` (class in `sympy.functions.elementary.hyperbolic`), 403  
`cosine_transform()` (in module `sympy.integrals.transforms`), 583  
`CosineTransform` (class in `sympy.integrals.transforms`), 583  
`coskewness()` (in module `sympy.stats`), 2968  
`cot` (class in `sympy.functions.elementary.trigonometric`), 392  
`coth` (class in `sympy.functions.elementary.hyperbolic`), 404  
`could_extract_minus_sign()` (`sympy.core.expr.Expr` method), 961  
`SymbolicSystem` (`sympy.core.basic.Basic` method), 932  
`count_complex_roots()` (`sympy.polys.polyclasses.DMP` method), 2562  
`count_digits()` (in module `sympy.ntheory.digits`), 1526  
`count_ops()` (in module `sympy.core.function`), 1059  
`count_ops()` (`sympy.core.basic.Basic` method), 932  
`count_ops()` (`sympy.core.expr.Expr` method), 961  
`count_partitions()` (`sympy.utilities.enumerative.MultisetPartitionTraverse` method), 2063  
`count_real_roots()` (`sympy.polys.polyclasses.DMP` method), 2562  
`count_roots()` (in module `sympy.polys.polytools`), 2374  
`count_roots()` (`sympy.polys.polytools.Poly` method), 2386  
`couple()` (in module `sympy.physics.quantum.spin`), 1829  
`CovarDerivativeOp` (class in `sympy.diffgeom`), 2814





- `D` (`sympy.physics.optics.gaussopt.RayTransferMatrix` property), 1865
- `D()` (`sympy.physics.quantum.spin.Rotation` class method), 1826
- `d()` (`sympy.physics.quantum.spin.Rotation` class method), 1827
- `Dagger` (class in `sympy.physics.quantum.dagger`), 1797
- `Dagger` (class in `sympy.physics.secondquant`), 1550
- `Dagum()` (in module `sympy.stats`), 2896
- `DataType` (class in `sympy.utilities.codegen`), 2049
- `dc_gain()` (`sympy.physics.control.lti.TransferFunction` method), 1894
- `dcm()` (`sympy.physics.mechanics.body.Body` method), 1761
- `dcm()` (`sympy.physics.vector.frame.ReferenceFrame` method), 1630
- `DDM` (class in `sympy.polys.matrices.ddm`), 2690
- `debug()` (in module `sympy.utilities.misc`), 2111
- `debug_decorator()` (in module `sympy.utilities.misc`), 2111
- `decimal_dig` (`sympy.codegen.ast.FloatType` property), 1132
- `decipher_affine()` (in module `sympy.crypto.crypto`), 2767
- `decipher_atbash()` (in module `sympy.crypto.crypto`), 2768
- `decipher_bifid()` (in module `sympy.crypto.crypto`), 2775
- `decipher_bifid5()` (in module `sympy.crypto.crypto`), 2778
- `decipher_bifid6()` (in module `sympy.crypto.crypto`), 2779
- `decipher_elgamal()` (in module `sympy.crypto.crypto`), 2794
- `decipher_gm()` (in module `sympy.crypto.crypto`), 2798
- `decipher_hill()` (in module `sympy.crypto.crypto`), 2773
- `decipher_kid_rsa()` (in module `sympy.crypto.crypto`), 2788
- `decipher_railfence()` (in module `sympy.crypto.crypto`), 2799
- `decipher_rot13()` (in module `sympy.crypto.crypto`), 2766
- `decipher_rsa()` (in module `sympy.crypto.crypto`), 2785
- `decipher_shift()` (in module `sympy.crypto.crypto`), 2765
- `decipher_vigenere()` (in module `sympy.crypto.crypto`), 2772
- `Declaration` (class in `sympy.codegen.ast`), 1130
- `decode_morse()` (in module `sympy.crypto.crypto`), 2789
- `decompose()` (in module `sympy.polys.polytools`), 2370
- `decompose()` (`sympy.physics.quantum.gate.CGate` method), 1842
- `decompose()` (`sympy.physics.quantum.gate.SwapGate` method), 1845
- `decompose()` (`sympy.physics.quantum.qft.IQFT` method), 1850
- `decompose()` (`sympy.physics.quantum.qft.QFT` method), 1850
- `decompose()` (`sympy.polys.polyclasses.DMP` method), 2562
- `decompose()` (`sympy.polys.polytools.Poly` method), 2386
- `decompose()` (`sympy.stats.DiscreteMarkovChain` method), 2949
- `deduced()` (`sympy.codegen.ast.Variable` class method), 1141
- `default_sort_key()` (in module `sympy.core.sorting`), 1074
- `deflate()` (`sympy.polys.polyclasses.DMP` method), 2562
- `deflate()` (`sympy.polys.polytools.Poly` method), 2386
- `deflection()` (`sympy.physics.continuum_mechanics.beam.Beam` method), 1954
- `deflection()` (`sympy.physics.continuum_mechanics.beam.Beam` method), 1984
- `degree` (`sympy.combinatorics.perm_groups.PermutationGroup` property), 305
- `degree()` (in module `sympy.polys.polytools`), 2361
- `degree()` (`sympy.polys.polyclasses.DMP` method), 2562
- `degree()` (`sympy.polys.polytools.Poly` method), 2387
- `degree()` (`sympy.polys.rings.PolyElement` method), 2556
- `degree_list()` (in module `sympy.polys.polytools`), 2361
- `degree_list()` (`sympy.polys.polyclasses.DMP` method), 2562
- `degree_list()` (`sympy.polys.polytools.Poly` method), 2387
- `degrees()` (`sympy.polys.rings.PolyElement` method), 2556
- `Del` (class in `sympy.vector.deloperator`), 1460
- `delete_doubles()` (`sympy.liealgebras.weyl_group.WeylGroup` method), 2339

`delta` (`sympy.functions.special.hyper.meijerg` property), 525  
`deltaintegrate()` (in module `sympy.integrals.deltafunctions`), 590  
`den` (`sympy.physics.control.lti.TransferFunction` property), 1895  
`denom()` (`sympy.polys.domains.AlgebraicField` method), 2543  
`denom()` (`sympy.polys.domains.domain.Domain` method), 2509  
`denom()` (`sympy.polys.domains.ExpressionDomain` method), 2549  
`denom()` (`sympy.polys.domains.FractionField` method), 2548  
`denom()` (`sympy.polys.domains.gaussiandomains.GaussianRationalField` method), 2538  
`denom()` (`sympy.polys.domains.GMPYRationalField` method), 2532  
`denom()` (`sympy.polys.domains.RationalField` method), 2530  
`denom()` (`sympy.polys.domains.ring.Ring` method), 2521  
`denom()` (`sympy.polys.polyclasses.DMF` method), 2568  
`DenseMatrix` (class in `sympy.matrices.dense`), 1361  
`density()` (in module `sympy.stats`), 2963  
`deprecated()` (in module `sympy.utilities.decorator`), 2057  
`depth()` (`sympy.polys.agca.ideals.Ideal` method), 2462  
`Derivative` (class in `sympy.core.function`), 1042  
`derive_by_array()` (in module `sympy.tensor.array`), 1392  
`derived_series()` (`sympy.combinatorics.perm_groups.PermGroup` method), 306  
`derived_subgroup()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 306  
`descent()` (in module `sympy.solvers.diophantine.diophantine`), 730  
`descents()` (`sympy.combinatorics.permutations.Permutation` method), 267  
`det()` (`sympy.matrices.matrices.MatrixDeterminant` method), 1230  
`det()` (`sympy.polys.matrices.ddm.DDM` method), 2690  
`det()` (`sympy.polys.matrices.domainmatrix.DomainMatrix` method), 2674  
`det()` (`sympy.polys.matrices.sdm.SDM` method), 2693  
`deviation()` (in module `sympy.physics.optics.utils`), 1881  
`dh_private_key()` (in module `sympy.crypto.crypto`), 2795  
`dh_public_key()` (in module `sympy.crypto.crypto`), 2795  
`dh_shared_key()` (in module `sympy.crypto.crypto`), 2796  
`diag()` (in module `sympy.matrices.dense`), 1319  
`diag()` (`sympy.matrices.common.MatrixCommon` class method), 1331  
`diag()` (`sympy.polys.matrices.domainmatrix.DomainMatrix` class method), 2674  
`diagonal()` (`sympy.matrices.common.MatrixCommon` method), 1331  
`diagonal_solve()` (`sympy.matrices.matrices.MatrixBase` method), 1295  
`diagonalize()` (`sympy.matrices.matrices.MatrixEigen` method), 1238  
`DiagonalPredicate` (class in `sympy.assumptions.predicates.matrices`), 209  
`Diagram` (class in `sympy.categories`), 2749  
`DiagramGrid` (class in `sympy.categories.diagram_drawing`), 2752  
`Dict` (class in `sympy.core.containers`), 1070  
`dict_merge()` (in module `sympy.utilities.iterables`), 2074  
`Die()` (in module `sympy.stats`), 2875  
`DiePSpace` (class in `sympy.stats.frv_types`), 2981  
`diff()` (in module `sympy.core.function`), 1048  
`diff()` (`sympy.holonomic.holonomic.HolonomicFunction` method), 2318  
`diff()` (`sympy.matrices.matrices.MatrixCalculus` method), 1279  
`diff()` (`sympy.physics.vector.vector.Vector` method), 1643  
`diff()` (`sympy.polys.fields.FracElement` method), 2561  
`diff()` (`sympy.polys.polyclasses.DMP` method), 2562  
`diff()` (`sympy.polys.polytools.Poly` method), 2387  
`diff()` (`sympy.polys.rings.PolyElement` method), 2556  
`difference_delta()` (in module `sympy.series.limitseq`), 659  
`Differential` (class in `sympy.diffgeom`), 2810  
`DifferentialOperator` (class in `sympy.holonomic.holonomic`), 2315

DifferentialOperator (class in [method](#)), 2336  
     [sympy.physics.quantum.operator](#)), 1807  
 DifferentialOperatorAlgebra (class in [sympy.holonomic.holonomic](#)), 2316  
 DifferentialOperators() (in module [sympy.holonomic.holonomic](#)), 2316  
 differentiate\_finite() (in module [sympy.calculus.finite\\_diff](#)), 241  
 dig ([sympy.codegen.ast.FloatType](#) property), 1132  
 digamma (class in [sympy.functions.special.gamma\\_functions](#)), 464  
 digit\_2txt (in module [sympy.printing.pretty.pretty\\_symbology](#)), 2185  
 digits() (in module [sympy.ntheory.digits](#)), 1527  
 digits() (in module [sympy.ntheory.factor\\_](#)), 1505  
 dihedral() ([sympy.combinatorics.generators](#) method), 287  
 DihedralGroup() (in module [sympy.combinatorics.named\\_groups](#)), 353  
 dim ([sympy.physics.units.dimensions.DimensionSystem](#) property), 1585  
 dim ([sympy.physics.units.unitsystem.UnitSystem](#) property), 1587  
 dim\_can\_vector() ([sympy.physics.units.dimensions.DimensionSystem](#) method), 1585  
 dim\_handling() (in module [sympy.printing.aesaracode](#)), 2169  
 dim\_vector() ([sympy.physics.units.dimensions.DimensionSystem](#) method), 1585  
 Dimension (class in [sympy.physics.units.dimensions](#)), 1584  
 dimension ([sympy.physics.quantum.hilbert.HilbertSpace](#) property), 1804  
 dimension() (in module [sympy.codegen.fnodes](#)), 1154  
 dimension() ([sympy.liealgebras.type\\_a.TypeA](#) method), 2329  
 dimension() ([sympy.liealgebras.type\\_b.TypeB](#) method), 2330  
 dimension() ([sympy.liealgebras.type\\_c.TypeC](#) method), 2332  
 dimension() ([sympy.liealgebras.type\\_d.TypeD](#) method), 2333  
 dimension() ([sympy.liealgebras.type\\_e.TypeE](#) method), 2335  
 dimension() ([sympy.liealgebras.type\\_f.TypeF](#) method), 2336  
 dimension() ([sympy.liealgebras.type\\_g.TypeG](#) method), 2337  
 DimensionSystem (class in [sympy.physics.units.dimensions](#)), 1584  
 diop\_bf\_DN() (in module [sympy.solvers.diophantine.diophantine](#)), 726  
 diop\_DN() (in module [sympy.solvers.diophantine.diophantine](#)), 724  
 diop\_general\_pythagorean() (in module [sympy.solvers.diophantine.diophantine](#)), 731  
 diop\_general\_sum\_of\_even\_powers() (in module [sympy.solvers.diophantine.diophantine](#)), 732  
 diop\_general\_sum\_of\_squares() (in module [sympy.solvers.diophantine.diophantine](#)), 731  
 diop\_linear() (in module [sympy.solvers.diophantine.diophantine](#)), 722  
 diop\_quadratic() (in module [sympy.solvers.diophantine.diophantine](#)), 723  
 diop\_solve() (in module [sympy.solvers.diophantine.diophantine](#)), 720  
 diop\_ternary\_quadratic() (in module [sympy.solvers.diophantine.diophantine](#)), 729  
 DiophantineEquationType (class in [sympy.solvers.diophantine.diophantine](#)), 745  
 DiophantineSolutionSet (class in [sympy.solvers.diophantine.diophantine](#)), 744  
 DiracDelta (class in [sympy.functions.special.delta\\_functions](#)), 450  
 direct\_product() ([sympy.tensor.tensor.TensorSymmetry](#) class method), 1421  
 direction ([sympy.geometry.line.LinearEntity](#) property), 2219

[direction\\_cosine\(sympy.geometry.line.LinearEntity3D method\)](#), 2234  
[direction\\_cosine\(sympy.geometry.point.Point3D method\)](#), 2213  
[direction\\_ratio\(sympy.geometry.line.LinearEntity3D property\)](#), 2243  
[direction\\_ratio\(sympy.geometry.point.Point3D method\)](#), 2213  
[director\\_circle\(sympy.geometry.ellipse.Ellipse method\)](#), 2256  
[DirectProduct\(\)](#) (in module [sympy.combinatorics.group\\_constructs](#)), 363  
[DirectSumHilbertSpace](#) (class in [sympy.physics.quantum.hilbert](#)), 1803  
[dirichlet\\_eta](#) (class in [sympy.functions.special.zeta\\_functions](#)), 516  
[discard\\_before\(\)](#) ([sympy.polys.numberfields.modules.submodule](#) method), 2727  
[discrete\\_log\(\)](#) (in module [sympy.ntheory.residue\\_ntheory](#)), 1522  
[DiscreteMarkovChain](#) (class in [sympy.stats](#)), 2944  
[DiscreteUniform\(\)](#) (in module [sympy.stats](#)), 2875  
[discriminant\(\)](#) (in module [sympy.polys.polytools](#)), 2366  
[discriminant\(\)](#) ([sympy.polys.domains.AlgebraicField](#) method), 2543  
[discriminant\(\)](#) ([sympy.polys.polyclasses.DMP](#) method), 2562  
[discriminant\(\)](#) ([sympy.polys.polytools.Poly](#) method), 2388  
[DisjointUnion](#) (class in [sympy.sets.sets](#)), 1201  
[dispersion\(\)](#) (in module [sympy.polys.dispersion](#)), 2448  
[dispersion\(\)](#) ([sympy.polys.polytools.Poly](#) method), 2388  
[dispersionset\(\)](#) (in module [sympy.polys.dispersion](#)), 2446  
[dispersionset\(\)](#) ([sympy.polys.polytools.Poly](#) method), 2389  
[distance\(\)](#) ([sympy.geometry.line.Line](#) method), 2229  
[distance\(\)](#) ([sympy.geometry.line.Ray](#) method), 2231  
[distance\(\)](#) ([sympy.geometry.line.Segment](#) method), 2234  
[distance\(\)](#) ([sympy.geometry.plane.Plane](#) method), 2305  
[distance\(\)](#) ([sympy.geometry.point.Point](#) method), 2204  
[distance\(\)](#) ([sympy.geometry.polygon.Polygon](#) method), 2278  
[distribute\\_and\\_over\\_or\(\)](#) (in module [sympy.logic.boolalg](#)), 1178  
[distribute\\_or\\_over\\_and\(\)](#) (in module [sympy.logic.boolalg](#)), 1178  
[distribute\\_xor\\_over\\_and\(\)](#) (in module [sympy.logic.boolalg](#)), 1178  
[div\(\)](#) (in module [sympy.polys.polytools](#)), 2363  
[div\(\)](#) ([sympy.polys.domains.domain.Domain](#) method), 2509  
[div\(\)](#) ([sympy.polys.domains.field.Field](#) method), 2520  
[div\(\)](#) ([sympy.polys.domains.GMPYRationalField](#) method), 2532  
[div\(\)](#) ([sympy.polys.domains.RationalField](#) method), 2530  
[div\(\)](#) ([sympy.polys.domains.ring.Ring](#) method), 2521  
[div\(\)](#) ([sympy.polys.polyclasses.DMP](#) method), 2563  
[div\(\)](#) ([sympy.polys.polytools.Poly](#) method), 2390  
[div\(\)](#) ([sympy.polys.rings.PolyElement](#) method), 2556  
[divergence\(sympy.physics.optics.gaussopt.BeamParameter property\)](#), 1858  
[divergence\(\)](#) (in module [sympy.physics.vector.fieldfunctions](#)), 1671  
[divergence\(\)](#) (in module [sympy.vector](#)), 1472  
[divisible\(\)](#) (in module [sympy.solvers.diophantine.diophantine](#)), 738  
[divisor\\_count\(\)](#) (in module [sympy.ntheory.factor\\_](#)), 1498  
[divisor\\_sigma](#) (class in [sympy.ntheory.factor\\_](#)), 1502  
[divisors\(\)](#) (in module [sympy.ntheory.factor\\_](#)), 1497  
[DMF](#) (class in [sympy.polys.polyclasses](#)), 2567  
[DMP](#) (class in [sympy.polys.polyclasses](#)), 2561  
[dmp\\_abs\(\)](#) (in module [sympy.polys.densearith](#)), 2586  
[dmp\\_add\(\)](#) (in module [sympy.polys.densearith](#)), 2586  
[dmp\\_add\\_ground\(\)](#) (in module [sympy.polys.densearith](#)), 2586



---

|                                                                                          |                                                                                                |
|------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------|
| <code>sympy.polys.densearith</code> ), 2584                                              | <code>sympy.polys.densearith</code> ), 2585                                                    |
| <code>dmp_add_mul()</code> (in module <code>sympy.polys.densearith</code> ), 2587        | <code>dmp_ext_factor()</code> (in module <code>sympy.polys.factortools</code> ), 2639          |
| <code>dmp_add_term()</code> (in module <code>sympy.polys.densearith</code> ), 2583       | <code>dmp_factor_list()</code> (in module <code>sympy.polys.factortools</code> ), 2639         |
| <code>dmp_apply_pairs()</code> (in module <code>sympy.polys.densebasic</code> ), 2583    | <code>dmp_factor_list_include()</code> (in module <code>sympy.polys.factortools</code> ), 2639 |
| <code>dmp_cancel()</code> (in module <code>sympy.polys.euclidtools</code> ), 2635        | <code>dmp_ff_div()</code> (in module <code>sympy.polys.densearith</code> ), 2590               |
| <code>dmp_clear_denoms()</code> (in module <code>sympy.polys.denstools</code> ), 2601    | <code>dmp_ff_prs_gcd()</code> (in module <code>sympy.polys.euclidtools</code> ), 2632          |
| <code>dmp_compose()</code> (in module <code>sympy.polys.denstools</code> ), 2600         | <code>dmp_from_dict()</code> (in module <code>sympy.polys.densebasic</code> ), 2578            |
| <code>dmp_content()</code> (in module <code>sympy.polys.euclidtools</code> ), 2634       | <code>dmp_from_sympy()</code> (in module <code>sympy.polys.densebasic</code> ), 2574           |
| <code>dmp_convert()</code> (in module <code>sympy.polys.densebasic</code> ), 2574        | <code>dmp_gcd()</code> (in module <code>sympy.polys.euclidtools</code> ), 2633                 |
| <code>dmp_copy()</code> (in module <code>sympy.polys.densebasic</code> ), 2573           | <code>dmp_gcdex()</code> (in module <code>sympy.polys.euclidtools</code> ), 2626               |
| <code>dmp_deflate()</code> (in module <code>sympy.polys.densebasic</code> ), 2580        | <code>dmp_ground()</code> (in module <code>sympy.polys.densebasic</code> ), 2577               |
| <code>dmp_degree()</code> (in module <code>sympy.polys.densebasic</code> ), 2571         | <code>dmp_ground_content()</code> (in module <code>sympy.polys.denstools</code> ), 2597        |
| <code>dmp_degree_in()</code> (in module <code>sympy.polys.densebasic</code> ), 2572      | <code>dmp_ground_extract()</code> (in module <code>sympy.polys.denstools</code> ), 2598        |
| <code>dmp_degree_list()</code> (in module <code>sympy.polys.densebasic</code> ), 2572    | <code>dmp_ground_LC()</code> (in module <code>sympy.polys.densebasic</code> ), 2570            |
| <code>dmp_diff()</code> (in module <code>sympy.polys.denstools</code> ), 2593            | <code>dmp_ground_monic()</code> (in module <code>sympy.polys.denstools</code> ), 2596          |
| <code>dmp_diff_eval_in()</code> (in module <code>sympy.polys.denstools</code> ), 2594    | <code>dmp_ground_nth()</code> (in module <code>sympy.polys.densebasic</code> ), 2575           |
| <code>dmp_diff_in()</code> (in module <code>sympy.polys.denstools</code> ), 2593         | <code>dmp_ground_p()</code> (in module <code>sympy.polys.densebasic</code> ), 2576             |
| <code>dmp_discriminant()</code> (in module <code>sympy.polys.euclidtools</code> ), 2631  | <code>dmp_ground_primitive()</code> (in module <code>sympy.polys.denstools</code> ), 2597      |
| <code>dmp_div()</code> (in module <code>sympy.polys.densearith</code> ), 2590            | <code>dmp_ground_TC()</code> (in module <code>sympy.polys.densebasic</code> ), 2571            |
| <code>dmp_eject()</code> (in module <code>sympy.polys.densebasic</code> ), 2582          | <code>dmp_ground_trunc()</code> (in module <code>sympy.polys.denstools</code> ), 2595          |
| <code>dmp_euclidean_prs()</code> (in module <code>sympy.polys.euclidtools</code> ), 2626 | <code>dmp_grounds()</code> (in module <code>sympy.polys.densebasic</code> ), 2577              |
| <code>dmp_eval()</code> (in module <code>sympy.polys.denstools</code> ), 2593            | <code>dmp_half_gcdex()</code> (in module <code>sympy.polys.euclidtools</code> ), 2625          |
| <code>dmp_eval_in()</code> (in module <code>sympy.polys.denstools</code> ), 2594         | <code>dmp_include()</code> (in module <code>sympy.polys.densebasic</code> ), 2581              |
| <code>dmp_eval_tail()</code> (in module <code>sympy.polys.denstools</code> ), 2594       | <code>dmp_inflate()</code> (in module <code>sympy.polys.densebasic</code> ), 2581              |
| <code>dmp_exclude()</code> (in module <code>sympy.polys.densebasic</code> ), 2581        | <code>dmp_inject()</code> (in module <code>sympy.polys.densebasic</code> ), 2581               |
| <code>dmp_expand()</code> (in module <code>sympy.polys.densearith</code> ), 2592         | <code>dmp_inner_gcd()</code> (in module <code>sympy.polys.euclidtools</code> ), 2633           |
| <code>dmp_exquo()</code> (in module <code>sympy.polys.densearith</code> ), 2591          | <code>dmp_inner_subresultants()</code> (in module <code>sympy.polys.euclidtools</code> ), 2628 |
| <code>dmp_exquo_ground()</code> (in module                                               | <code>dmp_integrate()</code> (in module                                                        |

|                                                                                                       |                                                                                                              |
|-------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------|
| <a href="#"><i>sympy.polys.dense tools</i></a> ), 2592                                                | <a href="#"><i>sympy.polys.densearith</i></a> ), 2588                                                        |
| <a href="#">dmp_integrate_in()</a> (in module <a href="#"><i>sympy.polys.dense tools</i></a> ), 2592  | <a href="#">dmp_primitive()</a> (in module <a href="#"><i>sympy.polys.euclidtools</i></a> ), 2634            |
| <a href="#">dmp_invert()</a> (in module <a href="#"><i>sympy.polys.euclidtools</i></a> ), 2626        | <a href="#">dmp_primitive_prs()</a> (in module <a href="#"><i>sympy.polys.euclidtools</i></a> ), 2627        |
| <a href="#">dmp_irreducible_p()</a> (in module <a href="#"><i>sympy.polys.factortools</i></a> ), 2639 | <a href="#">dmp_prs_resultant()</a> (in module <a href="#"><i>sympy.polys.euclidtools</i></a> ), 2629        |
| <a href="#">dmp_ll_norm()</a> (in module <a href="#"><i>sympy.polys.densearith</i></a> ), 2592        | <a href="#">dmp_qq_collins_resultant()</a> (in module <a href="#"><i>sympy.polys.euclidtools</i></a> ), 2630 |
| <a href="#">dmp_LC()</a> (in module <a href="#"><i>sympy.polys.densebasic</i></a> ), 2570             | <a href="#">dmp_qq_heu_gcd()</a> (in module <a href="#"><i>sympy.polys.euclidtools</i></a> ), 2633           |
| <a href="#">dmp_lcm()</a> (in module <a href="#"><i>sympy.polys.euclidtools</i></a> ), 2634           | <a href="#">dmp_quo()</a> (in module <a href="#"><i>sympy.polys.densearith</i></a> ), 2591                   |
| <a href="#">dmp_lift()</a> (in module <a href="#"><i>sympy.polys.dense tools</i></a> ), 2601          | <a href="#">dmp_quo_ground()</a> (in module <a href="#"><i>sympy.polys.densearith</i></a> ), 2585            |
| <a href="#">dmp_list_terms()</a> (in module <a href="#"><i>sympy.polys.densebasic</i></a> ), 2582     | <a href="#">dmp_raise()</a> (in module <a href="#"><i>sympy.polys.densebasic</i></a> ), 2580                 |
| <a href="#">dmp_max_norm()</a> (in module <a href="#"><i>sympy.polys.densearith</i></a> ), 2591       | <a href="#">dmp_rem()</a> (in module <a href="#"><i>sympy.polys.densearith</i></a> ), 2590                   |
| <a href="#">dmp_mul()</a> (in module <a href="#"><i>sympy.polys.densearith</i></a> ), 2587            | <a href="#">dmp_resultant()</a> (in module <a href="#"><i>sympy.polys.euclidtools</i></a> ), 2631            |
| <a href="#">dmp_mul_ground()</a> (in module <a href="#"><i>sympy.polys.densearith</i></a> ), 2585     | <a href="#">dmp_revert()</a> (in module <a href="#"><i>sympy.polys.dense tools</i></a> ), 2601               |
| <a href="#">dmp_mul_term()</a> (in module <a href="#"><i>sympy.polys.densearith</i></a> ), 2584       | <a href="#">dmp_rr_div()</a> (in module <a href="#"><i>sympy.polys.densearith</i></a> ), 2589                |
| <a href="#">dmp_multi_deflate()</a> (in module <a href="#"><i>sympy.polys.densebasic</i></a> ), 2580  | <a href="#">dmp_rr_prs_gcd()</a> (in module <a href="#"><i>sympy.polys.euclidtools</i></a> ), 2631           |
| <a href="#">dmp_neg()</a> (in module <a href="#"><i>sympy.polys.densearith</i></a> ), 2586            | <a href="#">dmp_slice()</a> (in module <a href="#"><i>sympy.polys.densebasic</i></a> ), 2583                 |
| <a href="#">dmp_negative_p()</a> (in module <a href="#"><i>sympy.polys.densebasic</i></a> ), 2577     | <a href="#">dmp_sqr()</a> (in module <a href="#"><i>sympy.polys.densearith</i></a> ), 2588                   |
| <a href="#">dmp_nest()</a> (in module <a href="#"><i>sympy.polys.densebasic</i></a> ), 2579           | <a href="#">dmp_strip()</a> (in module <a href="#"><i>sympy.polys.densebasic</i></a> ), 2572                 |
| <a href="#">dmp_normal()</a> (in module <a href="#"><i>sympy.polys.densebasic</i></a> ), 2574         | <a href="#">dmp_sub()</a> (in module <a href="#"><i>sympy.polys.densearith</i></a> ), 2587                   |
| <a href="#">dmp_nth()</a> (in module <a href="#"><i>sympy.polys.densebasic</i></a> ), 2575            | <a href="#">dmp_sub_ground()</a> (in module <a href="#"><i>sympy.polys.densearith</i></a> ), 2584            |
| <a href="#">dmp_one()</a> (in module <a href="#"><i>sympy.polys.densebasic</i></a> ), 2576            | <a href="#">dmp_sub_mul()</a> (in module <a href="#"><i>sympy.polys.densearith</i></a> ), 2587               |
| <a href="#">dmp_one_p()</a> (in module <a href="#"><i>sympy.polys.densebasic</i></a> ), 2576          | <a href="#">dmp_sub_term()</a> (in module <a href="#"><i>sympy.polys.densearith</i></a> ), 2584              |
| <a href="#">dmp_pdiv()</a> (in module <a href="#"><i>sympy.polys.densearith</i></a> ), 2588           | <a href="#">dmp_subresultants()</a> (in module <a href="#"><i>sympy.polys.euclidtools</i></a> ), 2629        |
| <a href="#">dmp_permute()</a> (in module <a href="#"><i>sympy.polys.densebasic</i></a> ), 2579        | <a href="#">dmp_swap()</a> (in module <a href="#"><i>sympy.polys.densebasic</i></a> ), 2579                  |
| <a href="#">dmp_pexquo()</a> (in module <a href="#"><i>sympy.polys.densearith</i></a> ), 2589         | <a href="#">dmp_TC()</a> (in module <a href="#"><i>sympy.polys.densebasic</i></a> ), 2570                    |
| <a href="#">dmp_positive_p()</a> (in module <a href="#"><i>sympy.polys.densebasic</i></a> ), 2578     | <a href="#">dmp_terms_gcd()</a> (in module <a href="#"><i>sympy.polys.densebasic</i></a> ), 2582             |
| <a href="#">dmp_pow()</a> (in module <a href="#"><i>sympy.polys.densearith</i></a> ), 2588            | <a href="#">dmp_to_dict()</a> (in module <a href="#"><i>sympy.polys.densebasic</i></a> ), 2578               |
| <a href="#">dmp_pquo()</a> (in module <a href="#"><i>sympy.polys.densearith</i></a> ), 2589           | <a href="#">dmp_to_tuple()</a> (in module <a href="#"><i>sympy.polys.densebasic</i></a> ), 2574              |
| <a href="#">dmp_prem()</a> (in module                                                                 | <a href="#">dmp_trial_division()</a> (in module                                                              |

[sympy.polys.factortools](#)), 2635  
[dmp\\_true\\_LT\(\)](#) (in module [sympy.polys.densebasic](#)), 2571  
[dmp\\_trunc\(\)](#) (in module [sympy.polys.densetools](#)), 2595  
[dmp\\_validate\(\)](#) (in module [sympy.polys.densebasic](#)), 2573  
[dmp\\_zero\(\)](#) (in module [sympy.polys.densebasic](#)), 2576  
[dmp\\_zero\\_p\(\)](#) (in module [sympy.polys.densebasic](#)), 2575  
[dmp\\_zeros\(\)](#) (in module [sympy.polys.densebasic](#)), 2577  
[dmp\\_zz\\_collins\\_resultant\(\)](#) (in module [sympy.polys.euclidtools](#)), 2630  
[dmp\\_zz\\_diophantine\(\)](#) (in module [sympy.polys.factortools](#)), 2638  
[dmp\\_zz\\_factor\(\)](#) (in module [sympy.polys.factortools](#)), 2638  
[dmp\\_zz\\_heu\\_gcd\(\)](#) (in module [sympy.polys.euclidtools](#)), 2632  
[dmp\\_zz\\_mignotte\\_bound\(\)](#) (in module [sympy.polys.factortools](#)), 2635  
[dmp\\_zz\\_modular\\_resultant\(\)](#) (in module [sympy.polys.euclidtools](#)), 2630  
[dmp\\_zz\\_wang\(\)](#) (in module [sympy.polys.factortools](#)), 2638  
[dmp\\_zz\\_wang\\_hensel\\_lifting\(\)](#) (in module [sympy.polys.factortools](#)), 2638  
[dmp\\_zz\\_wang\\_lead\\_coeffs\(\)](#) (in module [sympy.polys.factortools](#)), 2638  
[dmp\\_zz\\_wang\\_non\\_divisors\(\)](#) (in module [sympy.polys.factortools](#)), 2638  
[dmp\\_zz\\_wang\\_test\\_points\(\)](#) (in module [sympy.polys.factortools](#)), 2638  
[Do](#) (class in [sympy.codegen.fnodes](#)), 1150  
[do\\_subs\(\)](#) ([sympy.series.gruntz.SubsSet](#) method), 625  
[doctest\(\)](#) (in module [sympy.testing.runtests](#)), 2031  
[doctest\\_depends\\_on\(\)](#) (in module [sympy.utilities.decorator](#)), 2058  
[doit\(\)](#) ([sympy.core.basic.Basic](#) method), 932  
[doit\(\)](#) ([sympy.functions.elementary.piecewise.Piecewise](#) method), 417  
[doit\(\)](#) ([sympy.integrals.integrals.Integral](#) method), 603  
[doit\(\)](#) ([sympy.integrals.transforms.IntegralTransform](#) method), 586  
[doit\(\)](#) ([sympy.physics.control.lti.Feedback](#) method), 1908  
[doit\(\)](#) ([sympy.physics.control.lti.MIMOFeedback](#) method), 1929  
[doit\(\)](#) ([sympy.physics.control.lti.MIMOParallel](#) method), 1926  
[doit\(\)](#) ([sympy.physics.control.lti.MIMOSeries](#) method), 1924  
[doit\(\)](#) ([sympy.physics.control.lti.Parallel](#) method), 1904  
[doit\(\)](#) ([sympy.physics.control.lti.Series](#) method), 1901  
[doit\(\)](#) ([sympy.physics.quantum.anticommutator.AntiComm](#) method), 1792  
[doit\(\)](#) ([sympy.physics.quantum.commutator.Commutator](#) method), 1796  
[doit\(\)](#) ([sympy.physics.secondquant.Commutator](#) method), 1547  
[doit\(\)](#) ([sympy.physics.secondquant.NO](#) method), 1558  
[doit\(\)](#) ([sympy.physics.vector.dyadic.Dyadic](#) method), 1648  
[doit\(\)](#) ([sympy.physics.vector.vector.Vector](#) method), 1644  
[doit\(\)](#) ([sympy.series.limits.Limit](#) method), 621  
[doit\\_numerically\(\)](#) ([sympy.core.function.Derivative](#) method), 1047  
[Domain](#) (class in [sympy.polys.domains.domain](#)), 2504  
[domain](#) ([sympy.categories.CompositeMorphism](#) property), 2746  
[domain](#) ([sympy.categories.Morphism](#) property), 2743  
[domain](#) ([sympy.polys.polytools.Poly](#) property), 2391  
[domain\\_check\(\)](#) (in module [sympy.solvers.solve](#)), 869  
[DomainElement](#) (class in [sympy.polys.domains.domainelement](#)), 2519  
[DomainError](#) (class in [sympy.polys.polyerrors](#)), 2644  
[DomainMatrix](#) (class in [sympy.polys.matrices.domainmatrix](#)), 2671  
[dominant\(\)](#) (in module [sympy.series.limitseq](#)), 660  
[doprint\(\)](#) ([sympy.printing.aesaracode.AesaraPrinter](#) method), 2166  
[doprint\(\)](#) ([sympy.printing.codeprinter.CodePrinter](#) method), 2183  
[doprint\(\)](#) ([sympy.printing.mathml.MathMLPrinterBase](#) method), 2175  
[doprint\(\)](#) ([sympy.printing.printer.Printer](#) method), 2139  
[dot\(\)](#) (in module [sympy.physics.vector.functions](#)),

- 1667
- `dot()` (*sympy.geometry.point.Point* method), 2204
- `dot()` (*sympy.matrices.matrices.MatrixBase* method), 1296
- `dot()` (*sympy.physics.vector.dyadic.Dyadic* method), 1648
- `dot()` (*sympy.physics.vector.vector.Vector* method), 1644
- `dot()` (*sympy.vector.deloperator.Del* method), 1460
- `dot()` (*sympy.vector.dyadic.Dyadic* method), 1458
- `dot()` (*sympy.vector.vector.Vector* method), 1456
- `dot_rot_grad_Ynm()` (in module *sympy.physics.wigner*), 1568
- `dotprint()` (in module *sympy.printing.dot*), 2188
- `double_coset_can_rep()` (in module *sympy.combinatorics.tensor\_can*), 366
- `draw()` (*sympy.categories.diagram\_drawing.XypicDiagramDrawer* method), 2760
- `draw()` (*sympy.physics.continuum\_mechanics.beam.Beam* method), 1955
- `drop()` (*sympy.polys.domains.compositedomain.CompositeDomain* method), 2522
- `drop()` (*sympy.polys.domains.domain.Domain* method), 2510
- `drop()` (*sympy.polys.rings.PolyRing* method), 2554
- `drop_to_ground()` (*sympy.polys.rings.PolyRing* method), 2554
- `dsign` (class in *sympy.codegen.fnodes*), 1154
- `dsolve()` (in module *sympy.solvers.ode*), 755
- `dsolve_system()` (in module *sympy.solvers.ode.systems*), 758
- `dt()` (*sympy.physics.vector.dyadic.Dyadic* method), 1649
- `dt()` (*sympy.physics.vector.vector.Vector* method), 1644
- `dtype` (*sympy.polys.agca.extensions.MonogenicFiniteExtension* attribute), 2476
- `dtype` (*sympy.polys.agca.modules.FreeModule* attribute), 2454
- `dtype` (*sympy.polys.agca.modules.QuotientModule* attribute), 2465
- `dtype` (*sympy.polys.domains.AlgebraicField* attribute), 2543
- `dtype` (*sympy.polys.domains.domain.Domain* attribute), 2510
- `dtype` (*sympy.polys.domains.ExpressionDomain* attribute), 2549
- `dtype` (*sympy.polys.domains.gaussiandomains.GaussianInteger* attribute), 2535
- `dtype` (*sympy.polys.domains.gaussiandomains.GaussianRational* attribute), 2538
- `dual` (*sympy.physics.quantum.state.StateBase* property), 1834
- `dual()` (*sympy.matrices.matrices.MatrixBase* method), 1297
- `dual_class()` (*sympy.physics.quantum.state.StateBase* class method), 1834
- `Dummy` (class in *sympy.core.symbol*), 978
- `dummy_eq()` (*sympy.core.basic.Basic* method), 932
- `DummyWrapper` (class in *sympy.utilities.autowrap*), 2040
- `dump_c()` (*sympy.utilities.autowrap.UfuncifyCodeWrapper* method), 2040
- `dump_c()` (*sympy.utilities.codegen.CCodeGen* method), 2047
- `dump_code()` (*sympy.utilities.codegen.CodeGen* method), 2048
- `dump_f95()` (*sympy.utilities.codegen.FCodeGen* method), 2049
- `dump_h()` (*sympy.utilities.codegen.CCodeGen* method), 2048
- `dump_h()` (*sympy.utilities.codegen.FCodeGen* method), 2050
- `dump_jl()` (*sympy.utilities.codegen.JuliaCodeGen* method), 2051
- `dump_m()` (*sympy.utilities.codegen.OctaveCodeGen* method), 2051
- `dump_pyx()` (*sympy.utilities.autowrap.CythonCodeWrapper* method), 2040
- `dump_rs()` (*sympy.utilities.codegen.RustCodeGen* method), 2052
- `dup_content()` (in module *sympy.polys.denseutils*), 2596
- `dup_cyclotomic_p()` (in module *sympy.polys.factortools*), 2636
- `dup_decompose()` (in module *sympy.polys.denseutils*), 2600
- `dup_extract()` (in module *sympy.polys.denseutils*), 2598
- `dup_gf_factor()` (in module *sympy.polys.factortools*), 2639
- `dup_lshift()` (in module *sympy.polys.densearith*), 2585
- `dup_mirror()` (in module *sympy.polys.denseutils*), 2599
- `dup_monic()` (in module *sympy.polys.denseutils*), 2595
- `dup_primitive()` (in module *sympy.polys.denseutils*), 2597
- `dup_random()` (in module *sympy.polys.denseutils*), 2597



- `sympy.polys.densebasic`), 2583
- `dup_real_imag()` (in module `sympy.polys.densebasic`), 2583
- `dup_reverse()` (in module `sympy.polys.densebasic`), 2573
- `dup_rshift()` (in module `sympy.polys.densearith`), 2586
- `dup_scale()` (in module `sympy.polys.densebasic`), 2573
- `dup_shift()` (in module `sympy.polys.densearith`), 2586
- `dup_sign_variations()` (in module `sympy.polys.densearith`), 2586
- `dup_transform()` (in module `sympy.polys.densearith`), 2586
- `dup_zz_cyclotomic_factor()` (in module `sympy.polys.factortools`), 2637
- `dup_zz_cyclotomic_poly()` (in module `sympy.polys.factortools`), 2637
- `dup_zz_factor()` (in module `sympy.polys.factortools`), 2637
- `dup_zz_factor_sqf()` (in module `sympy.polys.factortools`), 2637
- `dup_zz_hensel_lift()` (in module `sympy.polys.factortools`), 2636
- `dup_zz_hensel_step()` (in module `sympy.polys.factortools`), 2635
- `dup_zz_irreducible_p()` (in module `sympy.polys.factortools`), 2636
- `dup_zz_zassenhaus()` (in module `sympy.polys.factortools`), 2636
- `Dyadic` (class in `sympy.physics.vector.dyadic`), 1648
- `Dyadic` (class in `sympy.vector.dyadic`), 1458
- `dyn_implicit_mat` (`sympy.physics.mechanics.system.SymbolicSystem` property), 1787
- `dyn_implicit_rhs` (`sympy.physics.mechanics.system.SymbolicSystem` property), 1787
- `dynamic_symbols()` (`sympy.physics.mechanics.system.SymbolicSystem` method), 1787
- `dynamic_symbols()` (in module `sympy.physics.vector`), 1666
- `dynkin_diagram()` (`sympy.liealgebras.root_system.RootSystem` method), 2327
- `DynkinDiagram()` (in module `sympy.liealgebras.dynkin_diagram`), 2340
- E**
- `E()` (in module `sympy.stats`), 2961
- `E1()` (in module `sympy.functions.special.error_functions`), 487
- `E_n()` (in module `sympy.physics.qho_1d`), 1539
- `E_nl()` (in module `sympy.physics.hydrogen`), 1533
- `E_nl()` (in module `sympy.physics.sho`), 1541
- `E_nl_dirac()` (in module `sympy.physics.hydrogen`), 1534
- `EC()` (`sympy.polys.polytools.Poly` method), 2379
- `eccentricity` (`sympy.geometry.ellipse.Ellipse` property), 2257
- `echelon_form()` (`sympy.matrices.matrices.MatrixReduction` method), 1232
- `ecm()` (in module `sympy.ntheory.ecm`), 1531
- `edges` (`sympy.combinatorics.polyhedron.Polyhedron` property), 333
- `edges()` (`sympy.combinatorics.prufer.Prufer` static method), 335
- `egyptian_fraction()` (in module `sympy.ntheory.egyptian_fraction`), 1529
- `Ei` (class in `sympy.functions.special.error_functions`), 483
- `eigenvals()` (`sympy.matrices.matrices.MatrixEigen` method), 1239
- `eigenvecs()` (`sympy.matrices.matrices.MatrixEigen` method), 1240
- `Eijk()` (in module `sympy.functions.special.tensor_functions`), 548
- `eject()` (`sympy.polys.polyclasses.DMP` method), 2563
- `eject()` (`sympy.polys.polytools.Poly` method), 2391
- `elastic_modulus` (`sympy.physics.continuum_mechanics.beam.Beam` property), 1956
- `element_from_alg_num()` (`sympy.polys.numberfields.modules.PowerBasis` method), 2725
- `element_from_ANP()` (`sympy.polys.numberfields.modules.PowerBasis` method), 2725
- `element_from_poly()` (`sympy.polys.numberfields.modules.PowerBasis` method), 2725
- `element_from_rational()` (`sympy.polys.numberfields.modules.Module` method), 2720
- `element_order()` (`sympy.liealgebras.weyl_group.WeylGroup` method), 2720

`method`), 2339  
`elementary_col_op()` (`sympy.matrices.matrices.MatrixReductions` `method`), 1233  
`elementary_row_op()` (`sympy.matrices.matrices.MatrixReductions` `method`), 1233  
`elements` (`sympy.combinatorics.perm_groups.PermutationGroup` `property`), 307  
`elgamal_private_key()` (`in module sympy.crypto.crypto`), 2793  
`elgamal_public_key()` (`in module sympy.crypto.crypto`), 2792  
`eliminate_implications()` (`in module sympy.logic.boolalg`), 1179  
`Ellipse` (`class in sympy.geometry.ellipse`), 2253  
`elliptic_e` (`class in sympy.functions.special.elliptic_integrals`), 527  
`elliptic_f` (`class in sympy.functions.special.elliptic_integrals`), 527  
`elliptic_k` (`class in sympy.functions.special.elliptic_integrals`), 526  
`elliptic_pi` (`class in sympy.functions.special.elliptic_integrals`), 528  
`EM()` (`sympy.polys.polytools.Poly` `method`), 2379  
`emptyPrinter()` (`sympy.printing.repr.ReprPrinter` `method`), 2177  
`EmptySequence` (`class in sympy.series.sequences`), 636  
`EmptySet` (`class in sympy.sets.sets`), 1202  
`enable_automatic_int_sympification()` (`in module sympy.interactive.session`), 2116  
`enable_automatic_symbols()` (`in module sympy.interactive.session`), 2116  
`encipher_affine()` (`in module sympy.crypto.crypto`), 2766  
`encipher_atbash()` (`in module sympy.crypto.crypto`), 2767  
`encipher_bifid()` (`in module sympy.crypto.crypto`), 2774  
`encipher_bifid5()` (`in module sympy.crypto.crypto`), 2776  
`encipher_bifid6()` (`in module sympy.crypto.crypto`), 2778  
`encipher_elgamal()` (`in module sympy.crypto.crypto`), 2793  
`encipher_gm()` (`in module sympy.crypto.crypto`), 2798  
`encipher_hill()` (`in module sympy.crypto.crypto`), 2772  
`encipher_kid_rsa()` (`in module sympy.crypto.crypto`), 2788  
`encipher_railfence()` (`in module sympy.crypto.crypto`), 2798  
`encipher_rot13()` (`in module sympy.crypto.crypto`), 2765  
`encipher_rsa()` (`in module sympy.crypto.crypto`), 2784  
`encipher_shift()` (`in module sympy.crypto.crypto`), 2764  
`encipher_substitution()` (`in module sympy.crypto.crypto`), 2768  
`encipher_vigenere()` (`in module sympy.crypto.crypto`), 2769  
`encloses()` (`sympy.geometry.entity.GeometryEntity` `method`), 2194  
`encloses_point()` (`sympy.geometry.ellipse.Ellipse` `method`), 2257  
`encloses_point()` (`sympy.geometry.polygon.Polygon` `method`), 2278  
`encloses_point()` (`sympy.geometry.polygon.RegularPolygon` `method`), 2288  
`encode_morse()` (`in module sympy.crypto.crypto`), 2788  
`end` (`sympy.sets.sets.Interval` `property`), 1195  
`endomorphism_ring()` (`sympy.polys.numberfields.modules.Module` `method`), 2720  
`EndomorphismRing` (`class in sympy.polys.numberfields.modules`), 2734  
`entropy()` (`in module sympy.stats`), 2963  
`enum_all()` (`sympy.utilities.enumerative.MultisetPartitionT` `method`), 2064  
`enum_large()` (`sympy.utilities.enumerative.MultisetPartitionT` `method`), 2064  
`enum_range()` (`sympy.utilities.enumerative.MultisetPartitionT` `method`), 2065  
`enum_small()` (`sympy.utilities.enumerative.MultisetPartitionT` `method`), 2065  
`enumerate_states()` (`in module sympy.physics.quantum.represent`), 1815  
`EPath` (`class in sympy.simplify.epathtools`), 688  
`epath()` (`in module sympy.simplify.epathtools`), 689  
`eps` (`sympy.codegen.ast.FloatType` `property`), 1132  
`Eq` (`in module sympy.core.relational`), 1022  
`eq()` (`sympy.polys.agca.modules.QuotientModuleElement`

method), 2467

eqs\_to\_matrix() (in module sympy.polys.solvers), 2667

Equality (class in sympy.core.relational), 1023

equals() (sympy.combinatorics.perm\_groups.PermGroup method), 307

equals() (sympy.core.expr.Expr method), 961

equals() (sympy.core.relational.Relational method), 1020

equals() (sympy.geometry.line.Line method), 2229

equals() (sympy.geometry.line.Ray method), 2232

equals() (sympy.geometry.line.Segment method), 2235

equals() (sympy.geometry.plane.Plane method), 2305

equals() (sympy.geometry.point.Point method), 2204

equals() (sympy.logic.boolalg.Boolean method), 1163

equals() (sympy.matrices.expressions.MatrixExpr method), 1371

equation() (sympy.geometry.ellipse.Circle method), 2271

equation() (sympy.geometry.ellipse.Ellipse method), 2257

equation() (sympy.geometry.line.Line2D method), 2239

equation() (sympy.geometry.line.Line3D method), 2244

equation() (sympy.geometry.plane.Plane method), 2306

equiv() (sympy.polys.numberfields.modules.ModuleElement method), 2730

Equivalent (class in sympy.logic.boolalg), 1171

equivalent() (in module sympy.solvers.diophantine.diophantine), 739

erf (class in sympy.functions.special.error\_functions), 471

erf2 (class in sympy.functions.special.error\_functions), 476

erf2inv (class in sympy.functions.special.error\_functions), 479

erfc (class in sympy.functions.special.error\_functions), 473

erfcinv (class in sympy.functions.special.error\_functions), 478

erfi (class in sympy.functions.special.error\_functions), 474

erfinv (class in sympy.functions.special.error\_functions), 477

Erlang() (in module sympy.stats), 2897

Erftotalsym (sympy.polys.polytools.Poly method), 2379

eta (sympy.functions.special.hyper.hyper property), 522

euler (class in sympy.functions.combinatorial.numbers), 433

euler\_equations() (in module sympy.calculus.euler), 233

euler\_maclaurin() (sympy.concrete.summations.Sum method), 902

EulerGamma (class in sympy.core.numbers), 1001

eulerline (sympy.geometry.polygon.Triangle property), 2296

eval() (sympy.assumptions.assume.Predicate method), 196

eval() (sympy.functions.elementary.piecewise.Piecewise class method), 417

eval() (sympy.functions.special.delta\_functions.DiracDelta class method), 451

eval() (sympy.functions.special.delta\_functions.Heaviside class method), 455

eval() (sympy.functions.special.singularity\_functions.Singularity class method), 458

eval() (sympy.functions.special.tensor\_functions.KroneckerDelta class method), 550

eval() (sympy.physics.quantum.hilbert.DirectSumHilbertSpace class method), 1803

eval() (sympy.physics.quantum.hilbert.TensorProductHilbertSpace class method), 1806

eval() (sympy.physics.secondquant.Commutator class method), 1547

eval() (sympy.physics.secondquant.Dagger class method), 1550

eval() (sympy.physics.secondquant.KroneckerDelta class method), 1554

evals() (sympy.polys.polyclasses.DMP method), 2563

evals(), (sympy.polys.polytools.Poly method), 2391

eval\_approx() (sympy.polys.rootoftools.ComplexRootOf method), 2435

eval\_controls() (sympy.physics.quantum.gate.CGate method), 1842

eval\_expr() (in module sympy.parsing.sympy\_parser), 2125

eval\_levicivita() (in module sympy.functions.special.tensor\_functions), 478

548  
 eval\_rational() (sympy.polys.rootoftools.ComplexRootOf method), 2436  
 eval\_zeta\_function() (sympy.concrete.summations.Sum method), 903  
 evalf() (sympy.core.evalf.EvalfMixin method), 1065  
 evalf() (sympy.holonomic.holonomic.HolonomicFunction method), 2320  
 evalf() (sympy.matrices.common.MatrixCommon method), 1333  
 evalf() (sympy.polys.domains.domain.Domain method), 2511  
 EvalfMixin (class in sympy.core.evalf), 1065  
 evaluate\_deltas() (in module sympy.physics.secondquant), 1562  
 evaluate\_pauli\_product() (in module sympy.physics.paulialgebra), 1539  
 EvaluationFailed (class in sympy.polys.polyerrors), 2643  
 even, 924  
 EvenPredicate (class in sympy.assumptions.predicates.ntheory), 216  
 evolve() (sympy.geometry.ellipse.Ellipse method), 2258  
 ExactQuotientFailed (class in sympy.polys.polyerrors), 2643  
 excenters (sympy.geometry.polygon.Triangle property), 2296  
 exclude() (sympy.polys.polyclasses.DMP method), 2563  
 exclude() (sympy.polys.polytools.Poly method), 2392  
 Exclusive (class in sympy.logic.boolalg), 1172  
 ExGaussian() (in module sympy.stats), 2898  
 exp (class in sympy.functions.elementary.exponential), 411  
 exp() (sympy.algebras.Quaternion method), 888  
 exp() (sympy.matrices.matrices.MatrixBase method), 1297  
 Exp1 (class in sympy.core.numbers), 999  
 exp2 (class in sympy.codegen.cfunctions), 1144  
 exp\_polar (class in sympy.functions.elementary.exponential), 414  
 exp\_re() (in module sympy.series.formal), 656  
 expand() (in module sympy.core.function), 1053  
 expand() (sympy.core.expr.Expr method), 1061  
 expand() (sympy.matrices.common.MatrixCommon method), 1333  
 expand() (sympy.physics.control.lti.TransferFunction method), 1895  
 expand() (sympy.physics.control.lti.TransferFunctionMatrix method), 1920  
 expand\_complex() (in module sympy.core.function), 1062  
 expand\_func() (in module sympy.core.function), 1061  
 expand\_log() (in module sympy.core.function), 1061  
 expand\_mul() (in module sympy.core.function), 1061  
 expand\_multinomial() (in module sympy.core.function), 1062  
 expand\_power\_base() (in module sympy.core.function), 1063  
 expand\_power\_exp() (in module sympy.core.function), 1063  
 expand\_trig() (in module sympy.core.function), 1062  
 Expectation (class in sympy.stats), 2961  
 expectation() (sympy.stats.BernoulliProcess method), 2954  
 ExpectationMatrix (class in sympy.stats), 2978  
 expint (class in sympy.functions.special.error\_functions), 485  
 expml (class in sympy.codegen.cfunctions), 1145  
 Exponential() (in module sympy.stats), 2900  
 Expr (class in sympy.core.expr), 947  
 expr (sympy.core.function.Lambda property), 1040  
 expr (sympy.core.function.Subs property), 1052  
 expr (sympy.functions.elementary.piecewise.ExprCondPair property), 415  
 expr (sympy.physics.quantum.operator.DifferentialOperator property), 1807  
 expr (sympy.physics.quantum.state.Wavefunction property), 1838  
 expr\_free\_symbols (sympy.core.expr.Expr property), 962  
 expr\_to\_holonomic() (in module sympy.holonomic.holonomic), 2324  
 ExprCondPair (class in sympy.functions.elementary.piecewise), 415  
 express() (in module sympy.physics.vector.functions),



1668  
 express() (in module *sympy.vector*), 1470  
 express() (*sympy.physics.vector.dyadic.Dyadic* method), 1649  
 express() (*sympy.physics.vector.vector.Vector* method), 1644  
 ExpressionDomain (class in *sympy.polys.domains*), 2549  
 ExpressionDomain.Expression (class in *sympy.polys.domains*), 2549  
 ExpressionDomain.Expression (class in *sympy.polys.domains.expressiondomain*), 2551  
 ExprWithIntLimits (class in *sympy.concrete.expr\_with\_intlimits*), 911  
 ExprWithLimits (class in *sympy.concrete.expr\_with\_limits*), 606  
 exquo() (in module *sympy.polys.polytools*), 2364  
 exquo() (*sympy.polys.domains.domain.Domain* method), 2511  
 exquo() (*sympy.polys.domains.field.Field* method), 2520  
 exquo() (*sympy.polys.domains.GMPYRationalField* method), 2532  
 exquo() (*sympy.polys.domains.RationalField* method), 2530  
 exquo() (*sympy.polys.domains.ring.Ring* method), 2521  
 exquo() (*sympy.polys.polyclasses.DMF* method), 2568  
 exquo() (*sympy.polys.polyclasses.DMP* method), 2563  
 exquo() (*sympy.polys.polytools.Poly* method), 2392  
 exquo\_ground() (*sympy.polys.polyclasses.DMP* method), 2563  
 exquo\_ground() (*sympy.polys.polytools.Poly* method), 2393  
 exradii (*sympy.geometry.polygon.Triangle* property), 2297  
 ext (*sympy.polys.domains.AlgebraicField* attribute), 2543  
 extend() (*sympy.ntheory.generate.Sieve* method), 1476  
 extend() (*sympy.physics.units.unitsystem.Unitsystem* method), 1587  
 extend() (*sympy.plotting.plot.Plot* method), 2823  
 extend\_to\_no() (*sympy.ntheory.generate.Sieve* method), 1477  
 extended\_negative, 925  
 extended\_nonnegative, 925  
 extended\_nonpositive, 925  
 extended\_nonzero, 925  
 extended\_positive, 925  
 extended\_real, 924  
 ExtendedRealPredicate (class in *sympy.assumptions.predicates.sets*), 228  
 ExtensionElement (class in *sympy.polys.agca.extensions*), 2476  
 Extent (class in *sympy.codegen.fnodes*), 1150  
 exterior\_angle (*sympy.geometry.polygon.RegularPolygon* property), 2289  
 extract() (*sympy.matrices.common.MatrixCommon* method), 1333  
 extract\_additively() (*sympy.core.expr.Expr* method), 962  
 extract\_branch\_factor() (*sympy.core.expr.Expr* method), 963  
 extract\_fundamental\_discriminant() (in module *sympy.polys.numberfields.utilities*), 2738  
 extract\_leading\_order() (*sympy.core.add.Add* method), 1016  
 extract\_multiplicatively() (*sympy.core.expr.Expr* method), 963  
 extract\_type\_tens() (in module *sympy.physics.hep.gamma\_matrices*), 1590  
 ExtraneousFactors (class in *sympy.polys.polyerrors*), 2643  
 eye() (in module *sympy.matrices.dense*), 1319  
 eye() (*sympy.matrices.common.MatrixCommon* class method), 1334  
 eye() (*sympy.polys.matrices.domainmatrix.DomainMatrix* class method), 2674  
 eye() (*sympy.polys.matrices.sdm.SDM* class method), 2694

## F

F (in module *sympy.physics.secondquant*), 1550  
 FCodeWrapper (class in *sympy.utilities.autowrap*), 2040  
 faces (*sympy.combinatorics.polyhedron.Polyhedron* property), 333  
 factor() (in module *sympy.polys.polytools*), 2373

`factor()` (`sympy.core.expr.Expr` method), 2147  
`factor_list()` (in module `sympy.polys.polytools`), 2372  
`factor_list()` (`sympy.polys.polyclasses.DMP` method), 2563  
`factor_list()` (`sympy.polys.polytools.Poly` method), 2393  
`factor_list_include()` (`sympy.polys.polyclasses.DMP` method), 2563  
`factor_list_include()` (`sympy.polys.polytools.Poly` method), 2393  
`factor_terms()` (in module `sympy.core.exprtools`), 1072  
`Factorable` (class in `sympy.solvers.ode.single`), 770  
`factorial` (class in `sympy.functions.combinatorial.factorials`), 434  
`factorial()` (`sympy.polys.domains.FractionField` method), 2548  
`factorial()` (`sympy.polys.domains.GMPYIntegerField` method), 2528  
`factorial()` (`sympy.polys.domains.GMPYRationalField` method), 2532  
`factorial()` (`sympy.polys.domains.IntegerRing` method), 2526  
`factorial()` (`sympy.polys.domains.PolynomialRing` method), 2547  
`factorial2` (class in `sympy.functions.combinatorial.factorials`), 436  
`factorial_moment()` (in module `sympy.stats`), 2973  
`factorial_notation()` (in module `sympy.parsing.sympy_parser`), 2129  
`factoring_visitor()` (in module `sympy.utilities.enumerative`), 2061  
`factorint()` (in module `sympy.ntheory.factor_`), 1493  
`factorrat()` (in module `sympy.ntheory.factor_`), 1496  
`factors()` (`sympy.core.numbers.Rational` method), 987  
`failed` (`sympy.testing.runtests.SymPyTestResults` attribute), 2031  
`FallingFactorial` (class in `sympy.functions.combinatorial.factorials`), 436  
`FBra` (in module `sympy.physics.secondquant`), 1550  
`fcode()` (in module `sympy.printing.fortran`), 2049  
`FCodeGen` (class in `sympy.utilities.codegen`), 2049  
`FCodePrinter` (class in `sympy.printing.fortran`), 2149  
`Fd` (in module `sympy.physics.secondquant`), 1550  
`fdiff()` (`sympy.codegen.cfunctions.Cbrt` method), 1144  
`fdiff()` (`sympy.codegen.cfunctions.exp2` method), 1145  
`fdiff()` (`sympy.codegen.cfunctions.expm1` method), 1145  
`fdiff()` (`sympy.codegen.cfunctions.fma` method), 1146  
`fdiff()` (`sympy.codegen.cfunctions.hypot` method), 1146  
`fdiff()` (`sympy.codegen.cfunctions.log10` method), 1147  
`fdiff()` (`sympy.codegen.cfunctions.log1p` method), 1147  
`fdiff()` (`sympy.codegen.cfunctions.log2` method), 1148  
`fdiff()` (`sympy.codegen.cfunctions.Sqrt` method), 1144  
`fdiff()` (`sympy.core.function.Function` method), 1051  
`fdiff()` (`sympy.functions.elementary.complexes.Abs` method), 385  
`fdiff()` (`sympy.functions.elementary.exponential.exp` method), 412  
`fdiff()` (`sympy.functions.elementary.exponential.LambertW` method), 413  
`fdiff()` (`sympy.functions.elementary.exponential.log` method), 414  
`fdiff()` (`sympy.functions.elementary.hyperbolic.csch` method), 405  
`fdiff()` (`sympy.functions.elementary.hyperbolic.sinh` method), 402  
`fdiff()` (`sympy.functions.special.delta_functions.DiracDelta` method), 452  
`fdiff()` (`sympy.functions.special.delta_functions.Heaviside` method), 455  
`fdiff()` (`sympy.functions.special.singularity_functions.SingularityFunction` method), 458  
`FDistribution()` (in module `sympy.stats`), 2901  
`Feedback` (class in `sympy.physics.control.lti`), 1906  
`fft()` (in module `sympy.discrete.transforms`), 1083  
`fglm()` (`sympy.polys.polytools.GroebnerBasis` method), 2423  
`fibonacci` (class in `sympy.functions.combinatorial.factorials`), 436

- [sympy.functions.combinatorial.numbers](#),  
[437](#)  
[Field](#) (class in [sympy.polys.domains.field](#)),  
[2520](#)  
[field\(\)](#) (in module [sympy.polys.fields](#)), [2560](#)  
[field\\_element\(\)](#) ([sympy.core.numbers.AlgebraicNumber](#)  
method), [991](#)  
[field\\_isomorphism\(\)](#) (in module  
[sympy.polys.numberfields.subfield](#)),  
[2711](#)  
[filldedent\(\)](#) (in module  
[sympy.utilities.misc](#)), [2111](#)  
[filter\\_symbols\(\)](#) (in module  
[sympy.utilities.iterables](#)), [2074](#)  
[find\(\)](#) ([sympy.core.basic.Basic](#) method), [933](#)  
[find\\_DN\(\)](#) (in module  
[sympy.solvers.diophantine.diophantine](#)),  
[728](#)  
[find\\_dynamicsymbols\(\)](#) (in module  
[sympy.physics.mechanics](#)), [1789](#)  
[find\\_executable\(\)](#) (in module  
[sympy.utilities.misc](#)), [2111](#)  
[find\\_linear\\_recurrence\(\)](#)  
([sympy.series.sequences.SeqBase](#)  
method), [633](#)  
[find\\_min\\_poly\(\)](#) (in module  
[sympy.polys.numberfields.modules](#)),  
[2736](#)  
[finite](#), [925](#)  
[finite\\_diff\\_weights\(\)](#) (in module  
[sympy.calculus.finite\\_diff](#)), [242](#)  
[FiniteDomain](#) (class in [sympy.stats.frv](#)), [2981](#)  
[FiniteField](#) (class in [sympy.polys.domains](#)),  
[2522](#)  
[FiniteFormalPowerSeries](#) (class in  
[sympy.series.formal](#)), [654](#)  
[FinitePredicate](#) (class in  
[sympy.assumptions.predicates.calculus](#)),  
[202](#)  
[FinitePSpace](#) (class in [sympy.stats.frv](#)), [2981](#)  
[FiniteRV\(\)](#) (in module [sympy.stats](#)), [2879](#)  
[FiniteSet](#) (class in [sympy.sets.sets](#)), [1197](#)  
[first\\_moment\\_of\\_area\(\)](#)  
([sympy.geometry.polygon.Polygon](#)  
method), [2279](#)  
[FirstExact](#) (class in  
[sympy.solvers.ode.single](#)), [771](#)  
[FirstLinear](#) (class in  
[sympy.solvers.ode.single](#)), [776](#)  
[FisherZ\(\)](#) (in module [sympy.stats](#)), [2902](#)  
[fixed\\_row\\_vector\(\)](#)  
([sympy.stats.DiscreteMarkovChain](#)  
method), [2950](#)  
[FixedBosonicBasis](#) (class in
- [sympy.physics.secondquant](#)), [1550](#)  
[FKet](#) (in module [sympy.physics.secondquant](#)),  
[1550](#)  
[FlagError](#) (class in [sympy.polys.polyerrors](#)),  
[2644](#)  
[FlatNumber](#) (class in  
[sympy.physics.optics.gaussopt](#)),  
[1861](#)  
[FlatRefraction](#) (class in  
[sympy.physics.optics.gaussopt](#)),  
[1861](#)  
[flatten\(\)](#) (in module  
[sympy.utilities.iterables](#)), [2074](#)  
[flatten\(\)](#) ([sympy.categories.CompositeMorphism](#)  
method), [2746](#)  
[flatten\(\)](#) ([sympy.core.add.Add](#) class  
method), [1017](#)  
[flatten\(\)](#) ([sympy.core.mul.Mul](#) class  
method), [1011](#)  
[Float](#) (class in [sympy.core.numbers](#)), [982](#)  
[FloatBaseType](#) (class in [sympy.codegen.ast](#)),  
[1131](#)  
[FloatType](#) (class in [sympy.codegen.ast](#)), [1131](#)  
[floor](#) (class in [sympy.functions.elementary.integers](#)),  
[409](#)  
[fma](#) (class in [sympy.codegen.cfunctions](#)),  
[1145](#)  
[foci](#) ([sympy.geometry.ellipse.Ellipse](#) prop-  
erty), [2259](#)  
[FockSpace](#) (class in  
[sympy.physics.quantum.hilbert](#)),  
[1803](#)  
[FockState](#) (class in  
[sympy.physics.secondquant](#)), [1551](#)  
[FockStateBosonBra](#) (class in  
[sympy.physics.secondquant](#)), [1551](#)  
[FockStateBosonKet](#) (class in  
[sympy.physics.secondquant](#)), [1552](#)  
[FockStateBra](#) (class in  
[sympy.physics.secondquant](#)), [1552](#)  
[FockStateFermionBra](#) (class in  
[sympy.physics.secondquant](#)), [1552](#)  
[FockStateFermionKet](#) (class in  
[sympy.physics.secondquant](#)), [1552](#)  
[FockStateKet](#) (class in  
[sympy.physics.secondquant](#)), [1553](#)  
[focus\\_distance](#) ([sympy.geometry.ellipse.Ellipse](#)  
property), [2259](#)  
[For](#) (class in [sympy.codegen.ast](#)), [1132](#)  
[forcing](#) ([sympy.physics.mechanics.jointsmethod.JointsMeth](#)  
property), [1781](#)  
[forcing](#) ([sympy.physics.mechanics.kane.KanesMethod](#)  
property), [1765](#)  
[forcing](#) ([sympy.physics.mechanics.lagrange.LagrangesMet](#)

[property](#)), 1769  
[forcing\\_full\(sympy.physics.mechanics.joints.methods.MethodsMethod\)](#), 606  
[property](#)), 1781  
[forcing\\_full\(sympy.physics.mechanics.kane.KaneMethods\)](#) (sympy.core.basic.Basic [property](#)), 933  
[property](#)), 1765  
[forcing\\_full\(sympy.physics.mechanics.lagrange.LagrangeMethodsMethod\)](#), 2249  
[property](#)), 1769  
[form\\_eoms\(\)](#) (sympy.physics.mechanics.joints.methods.MethodsMethod), 1335  
[method](#)), 1782  
[form\\_lagranges\\_equations\(\)](#) (sympy.physics.mechanics.lagrange.LagrangeMethodsMethod), 1769  
[FormalPowerSeries](#) (class in sympy.series.formal), 647  
[FormalPowerSeriesCompose](#) (class in sympy.series.formal), 653  
[FormalPowerSeriesInverse](#) (class in sympy.series.formal), 653  
[FormalPowerSeriesProduct](#) (class in sympy.series.formal), 653  
[FortranReturn](#) (class in sympy.codegen.fnodes), 1150  
[fourier\\_series\(\)](#) (in module sympy.series.fourier), 643  
[fourier\\_series\(\)](#) (sympy.core.expr.Expr [method](#)), 964  
[fourier\\_transform\(\)](#) (in module sympy.integrals.transforms), 579  
[FourierSeries](#) (class in sympy.series.fourier), 640  
[FourierTransform](#) (class in sympy.integrals.transforms), 580  
[fps\(\)](#) (in module sympy.series.formal), 650  
[fps\(\)](#) (sympy.core.expr.Expr [method](#)), 964  
[frac](#) (class in sympy.functions.elementary.integers), 410  
[frac](#) (in module sympy.printing.pretty.pretty\_symbolology), 2185  
[frac\\_field\(\)](#) (sympy.polys.domains.domain.Domain [method](#)), 2512  
[frac\\_unify\(\)](#) (sympy.polys.polyclasses.DMF [method](#)), 2568  
[FracElement](#) (class in sympy.polys.fields), 2561  
[FracField](#) (class in sympy.polys.fields), 2561  
[fraction\(\)](#) (in module sympy.simplify.radsimp), 677  
[FractionField](#) (class in sympy.polys.domains), 2548  
[Frechet\(\)](#) (in module sympy.stats), 2903  
[free\\_dynamicsymbols\(\)](#) (sympy.physics.vector.vector.Vector [method](#)), 1645  
[free\\_module\(\)](#) (sympy.polys.domains.ring.Ring [method](#)), 2521  
[free\\_symbols](#) (sympy.concrete.expr\_with\_limits.ExprWithLimits [property](#)), 606  
[free\\_symbols](#) (sympy.core.basic.Basic [property](#)), 933  
[free\\_symbols](#) (sympy.geometry.curve.Curve [property](#)), 2249  
[free\\_symbols](#) (sympy.integrals.integrals.Integral [property](#)), 603  
[free\\_symbols](#) (sympy.matrices.common.MatrixCommon [property](#)), 1335  
[free\\_symbols](#) (sympy.physics.quantum.operator.DifferentialEquation [property](#)), 1808  
[free\\_symbols](#) (sympy.physics.units.quantities.Quantity [property](#)), 1588  
[free\\_symbols](#) (sympy.polys.polytools.Poly [property](#)), 2394  
[free\\_symbols](#) (sympy.polys.polytools.PurePoly [property](#)), 2423  
[free\\_symbols](#) (sympy.series.sequences.SeqBase [property](#)), 633  
[free\\_symbols\(\)](#) (sympy.physics.vector.vector.Vector [method](#)), 1645  
[free\\_symbols\\_in\\_domain](#) (sympy.polys.polytools.Poly [property](#)), 2394  
[FreeModule](#) (class in sympy.polys.agca.modules), 2454  
[FreeModuleElement](#) (class in sympy.polys.agca.modules), 2456  
[FreeSpace](#) (class in sympy.physics.optics.gaussopt), 1862  
[frequency](#) (sympy.physics.optics.waves.TWave [property](#)), 1888  
[fresnel\\_coefficients\(\)](#) (in module sympy.physics.optics.utils), 1882  
[fresnelc](#) (class in sympy.functions.special.error\_functions), 482  
[FresnelIntegral](#) (class in sympy.functions.special.error\_functions), 480  
[fresnels](#) (class in sympy.functions.special.error\_functions), 480  
[from\\_AlgebraicField\(\)](#) (sympy.polys.domains.AlgebraicField [method](#)), 2543  
[from\\_AlgebraicField\(\)](#) (sympy.polys.domains.domain.Domain [method](#)), 2512  
[from\\_AlgebraicField\(\)](#) (sympy.polys.domains.FractionField [method](#)), 2512



`method`), 2548  
`from_AlgebraicField()` (`sympy.polys.domains.gaussiandomains.GaussianIntegerRing` `method`), 2533  
`from_AlgebraicField()` (`sympy.polys.domains.IntegerRing` `method`), 2526  
`from_AlgebraicField()` (`sympy.polys.domains.PolynomialRing` `method`), 2547  
`from_AlgebraicField()` (`sympy.polys.domains.RationalField` `method`), 2531  
`from_axis_angle()` (`sympy.algebras.Quaternion` `class method`), 889  
`from_ComplexField()` (`sympy.polys.domains.domain.Domain` `method`), 2512  
`from_ComplexField()` (`sympy.polys.domains.FractionField` `method`), 2548  
`from_ComplexField()` (`sympy.polys.domains.PolynomialRing` `method`), 2547  
`from_ddm()` (`sympy.polys.matrices.sdm.SDM` `class method`), 2694  
`from_dict()` (`sympy.polys.polyclasses.DMP` `class method`), 2563  
`from_dict()` (`sympy.polys.polytools.Poly` `class method`), 2394  
`from_dict_sympy()` (`sympy.polys.matrices.domainmatrix.DomainMatrix` `class method`), 2675  
`from_expr()` (`sympy.codegen.ast.Type` `class method`), 1139  
`from_expr()` (`sympy.polys.polytools.Poly` `class method`), 2394  
`from_ExpressionDomain()` (`sympy.polys.domains.domain.Domain` `method`), 2512  
`from_ExpressionDomain()` (`sympy.polys.domains.ExpressionDomain` `method`), 2550  
`from_ExpressionRawDomain()` (`sympy.polys.domains.domain.Domain` `method`), 2513  
`from_FF()` (`sympy.polys.domains.domain.Domain` `method`), 2513  
`from_FF()` (`sympy.polys.domains.FiniteField` `method`), 2524  
`from_FF()` (`sympy.polys.domains.IntegerRing` `method`), 2526  
`from_FF_gmpy()` (`sympy.polys.domains.domain.Domain` `method`), 2513  
`from_FF_gmpy()` (`sympy.polys.domains.FiniteField` `method`), 2524  
`from_FF_gmpy()` (`sympy.polys.domains.GMPYIntegerRing` `method`), 2528  
`from_FF_gmpy()` (`sympy.polys.domains.IntegerRing` `method`), 2526  
`from_FF_python()` (`sympy.polys.domains.domain.Domain` `method`), 2513  
`from_FF_python()` (`sympy.polys.domains.FiniteField` `method`), 2524  
`from_FF_python()` (`sympy.polys.domains.GMPYIntegerRing` `method`), 2528  
`from_FF_python()` (`sympy.polys.domains.IntegerRing` `method`), 2526  
`from_FractionField()` (`sympy.polys.domains.domain.Domain` `method`), 2513  
`from_FractionField()` (`sympy.polys.domains.ExpressionDomain` `method`), 2550  
`from_FractionField()` (`sympy.polys.domains.FractionField` `method`), 2548  
`from_FractionField()` (`sympy.polys.domains.PolynomialRing` `method`), 2547  
`from_GaussianIntegerRing()` (`sympy.polys.domains.AlgebraicField` `method`), 2543  
`from_GaussianIntegerRing()` (`sympy.polys.domains.ExpressionDomain` `method`), 2550  
`from_GaussianIntegerRing()` (`sympy.polys.domains.FractionField` `method`), 2548  
`from_GaussianIntegerRing()` (`sympy.polys.domains.gaussiandomains.GaussianIntegerRing` `method`), 2536  
`from_GaussianIntegerRing()` (`sympy.polys.domains.gaussiandomains.GaussianRationalField` `method`), 2538  
`from_GaussianIntegerRing()` (`sympy.polys.domains.PolynomialRing` `method`), 2547  
`from_GaussianRationalField()` (`sympy.polys.domains.AlgebraicField` `method`), 2543  
`from_GaussianRationalField()` (`sympy.polys.domains.ExpressionDomain` `method`), 2550  
`from_GaussianRationalField()` (`sympy.polys.domains.FractionField` `method`), 2548

```

from_GaussianRationalField() (sympy.polys.domains.domain.Domain
(sympy.polys.domains.gaussiandomains.GaussianRationalField
method), 2536
from_GaussianRationalField() (sympy.polys.domains.ExpressionDomain
(sympy.polys.domains.gaussiandomains.GaussianRationalField
method), 2538
from_GaussianRationalField() (sympy.polys.domains.FractionField
(sympy.polys.domains.GMPYRationalField
method), 2532
from_GaussianRationalField() (sympy.polys.domains.PolynomialRing
(sympy.polys.domains.PolynomialRing
method), 2547
from_GaussianRationalField() (sympy.polys.domains.PolynomialRing
(sympy.polys.domains.PolynomialRing
method), 2547
from_GaussianRationalField() (sympy.polys.domains.RationalField
(sympy.polys.domains.RationalField
method), 2531
from_GlobalPolynomialRing() (sympy.polys.domains.domain.Domain
(sympy.polys.domains.domain.Domain
method), 2513
from_GlobalPolynomialRing() (sympy.polys.domains.PolynomialRing
(sympy.polys.domains.PolynomialRing
method), 2547
from_hyper() (in module
sympy.holonomic.holonomic), 2323
from_index_summation() (sympy.matrices.expressions.MatrixExpr
static method), 1371
from_int_list() (sympy.polys.numberfields.modules.ModuleElement
class method), 2731
from_inversion_vector() (sympy.combinatorics.permutations.Permutation
class method), 268
from_list() (sympy.polys.matrices.domainmatrix.DomainMatrix
class method), 2676
from_list() (sympy.polys.matrices.sdm.SDM
class method), 2694
from_list() (sympy.polys.polyclasses.DMP
class method), 2563
from_list() (sympy.polys.polytools.Poly
class method), 2394
from_list_sympy() (sympy.polys.matrices.domainmatrix.DomainMatrix
class method), 2676
from_Matrix() (sympy.physics.control.lti.TransferFunctionModel
class method), 1920
from_Matrix() (sympy.polys.matrices.domainmatrix.DomainMatrix
class method), 2675
from_meijerg() (in module
sympy.holonomic.holonomic), 2323
from_MonogenicFiniteExtension() (sympy.polys.domains.domain.Domain
(sympy.polys.domains.domain.Domain
method), 2513
from_poly() (sympy.polys.polytools.Poly
class method), 2394
from_PolynomialRing() (sympy.polys.domains.domain.Domain
(sympy.polys.domains.domain.Domain
method), 2513
from_PolynomialRing() (sympy.polys.domains.ExpressionDomain
(sympy.polys.domains.ExpressionDomain
method), 2513
from_PolynomialRing() (sympy.polys.domains.FractionField
(sympy.polys.domains.FractionField
method), 2548
from_PolynomialRing() (sympy.polys.domains.PolynomialRing
(sympy.polys.domains.PolynomialRing
method), 2547
from_QQ() (sympy.polys.domains.AlgebraicField
(sympy.polys.domains.AlgebraicField
method), 2543
from_QQ() (sympy.polys.domains.ExpressionDomain
(sympy.polys.domains.ExpressionDomain
method), 2550
from_QQ() (sympy.polys.domains.FiniteField
(sympy.polys.domains.FiniteField
method), 2524
from_QQ() (sympy.polys.domains.FractionField
(sympy.polys.domains.FractionField
method), 2548
from_QQ() (sympy.polys.domains.gaussiandomains.GaussianRationalField
(sympy.polys.domains.gaussiandomains.GaussianRationalField
method), 2533
from_QQ() (sympy.polys.domains.GMPYIntegerRing
(sympy.polys.domains.GMPYIntegerRing
method), 2528
from_QQ() (sympy.polys.domains.IntegerRing
(sympy.polys.domains.IntegerRing
method), 2527
from_QQ() (sympy.polys.domains.PolynomialRing
(sympy.polys.domains.PolynomialRing
method), 2547
from_QQ() (sympy.polys.domains.RationalField
(sympy.polys.domains.RationalField
method), 2531
from_QQ_gmpy() (sympy.polys.domains.AlgebraicField
(sympy.polys.domains.AlgebraicField
method), 2543
from_QQ_gmpy() (sympy.polys.domains.domain.Domain
(sympy.polys.domains.domain.Domain
method), 2513
from_QQ_gmpy() (sympy.polys.domains.ExpressionDomain
(sympy.polys.domains.ExpressionDomain
method), 2550
from_QQ_gmpy() (sympy.polys.domains.FiniteField
(sympy.polys.domains.FiniteField
method), 2524
from_QQ_gmpy() (sympy.polys.domains.FractionField
(sympy.polys.domains.FractionField
method), 2549
from_QQ_gmpy() (sympy.polys.domains.gaussiandomains.GaussianRationalField
(sympy.polys.domains.gaussiandomains.GaussianRationalField
method), 2533
from_QQ_gmpy() (sympy.polys.domains.GMPYIntegerRing
(sympy.polys.domains.GMPYIntegerRing
method), 2528
from_QQ_gmpy() (sympy.polys.domains.GMPYRationalField
(sympy.polys.domains.GMPYRationalField
method), 2532
from_QQ_gmpy() (sympy.polys.domains.IntegerRing
(sympy.polys.domains.IntegerRing
method), 2527
from_QQ_gmpy() (sympy.polys.domains.PolynomialRing
(sympy.polys.domains.PolynomialRing
method), 2547
from_QQ_gmpy() (sympy.polys.domains.RationalField
(sympy.polys.domains.RationalField
method), 2531
from_QQ_python() (sympy.polys.domains.AlgebraicField
(sympy.polys.domains.AlgebraicField
method), 2543
from_QQ_python() (sympy.polys.domains.domain.Domain
(sympy.polys.domains.domain.Domain
method), 2513

```

`method`), 2513  
`from_QQ_python()` (`sympy.polys.domains.ExpressionField` `method`), 2550  
`from_QQ_python()` (`sympy.polys.domains.FiniteField` `method`), 2524  
`from_QQ_python()` (`sympy.polys.domains.FractionField` `method`), 2549  
`from_QQ_python()` (`sympy.polys.domains.gaussian_domains.FiniteField` `method`), 2533  
`from_QQ_python()` (`sympy.polys.domains.GMPYIntegerRing` `method`), 2528  
`from_QQ_python()` (`sympy.polys.domains.GMPYRationalField` `method`), 2532  
`from_QQ_python()` (`sympy.polys.domains.IntegerRing` `method`), 2527  
`from_QQ_python()` (`sympy.polys.domains.PolynomialRing` `method`), 2547  
`from_QQ_python()` (`sympy.polys.domains.RationalField` `method`), 2531  
`from_rational_expression()` (`sympy.physics.control.lti.TransferFunction` `class method`), 1895  
`from_real()` (`sympy.sets.fancysets.ComplexRegion` `class method`), 1211  
`from_RealField()` (`sympy.polys.domains.AlgebraicField` `method`), 2543  
`from_RealField()` (`sympy.polys.domains.domain.Domain` `method`), 2513  
`from_RealField()` (`sympy.polys.domains.ExpressionDomain` `method`), 2550  
`from_RealField()` (`sympy.polys.domains.FiniteField` `method`), 2524  
`from_RealField()` (`sympy.polys.domains.FractionField` `method`), 2549  
`from_RealField()` (`sympy.polys.domains.GMPYIntegerRing` `method`), 2528  
`from_RealField()` (`sympy.polys.domains.GMPYRationalField` `method`), 2532  
`from_RealField()` (`sympy.polys.domains.IntegerRing` `method`), 2527  
`from_RealField()` (`sympy.polys.domains.PolynomialRing` `method`), 2547  
`from_RealField()` (`sympy.polys.domains.RationalField` `method`), 2531  
`from_rep()` (`sympy.polys.matrices.domainmatrix.DomainMatrix` `class method`), 2677  
`from_rgs()` (`sympy.combinatorics.partitions.Partition` `class method`), 252  
`from_rotation_matrix()` (`sympy.algebras.Quaternion` `class method`), 889  
`from_sequence()` (`sympy.combinatorics.permutations.Permutation` `class method`), 268  
`from_sympy()` (`sympy.polys.domains.AlgebraicField` `method`), 2544  
`from_sympy()` (`sympy.polys.domains.ComplexField` `method`), 2546  
`from_sympy()` (`sympy.polys.domains.domain.Domain` `method`), 2513  
`from_sympy()` (`sympy.polys.domains.ExpressionDomain` `method`), 2550  
`from_sympy()` (`sympy.polys.domains.FiniteField` `method`), 2524  
`from_sympy()` (`sympy.polys.domains.FractionField` `method`), 2549  
`from_sympy()` (`sympy.polys.domains.gaussian_domains.GaussianField` `method`), 2533  
`from_sympy()` (`sympy.polys.domains.GMPYIntegerRing` `method`), 2529  
`from_sympy()` (`sympy.polys.domains.GMPYRationalField` `method`), 2532  
`from_sympy()` (`sympy.polys.domains.IntegerRing` `method`), 2527  
`from_sympy()` (`sympy.polys.domains.PolynomialRing` `method`), 2547  
`from_sympy()` (`sympy.polys.domains.RationalField` `method`), 2531  
`from_sympy()` (`sympy.polys.domains.RealField` `method`), 2545  
`from_sympy_list()` (`sympy.polys.polyclasses.DMP` `class method`), 2563  
`from_ZZ()` (`sympy.polys.domains.AlgebraicField` `method`), 2543  
`from_ZZ()` (`sympy.polys.domains.ExpressionDomain` `method`), 2550  
`from_ZZ()` (`sympy.polys.domains.FiniteField` `method`), 2524  
`from_ZZ()` (`sympy.polys.domains.FractionField` `method`), 2549  
`from_ZZ()` (`sympy.polys.domains.gaussian_domains.GaussianField` `method`), 2533  
`from_ZZ()` (`sympy.polys.domains.IntegerRing` `method`), 2527  
`from_ZZ()` (`sympy.polys.domains.PolynomialRing` `method`), 2547  
`from_ZZ()` (`sympy.polys.domains.RationalField` `method`), 2531  
`from_ZZ_gmpy()` (`sympy.polys.domains.domain.Domain` `method`), 2513  
`from_ZZ_gmpy()` (`sympy.polys.domains.ExpressionDomain` `method`), 2550  
`from_ZZ_gmpy()` (`sympy.polys.domains.FiniteField` `method`), 2524  
`from_ZZ_gmpy()` (`sympy.polys.domains.FractionField` `method`), 2549





- GammaInverse() (in module *sympy.stats*), 2905
- GammaProcess (class in *sympy.stats*), 2956
- gammasimp() (*sympy.core.expr.Expr* method), 964
- Gate (class in *sympy.physics.quantum.gate*), 1843
- gate (*sympy.physics.quantum.gate.CGate* property), 1842
- gate (*sympy.physics.quantum.gate.CNotGate* property), 1843
- gate\_simp() (in module *sympy.physics.quantum.gate*), 1846
- gate\_sort() (in module *sympy.physics.quantum.gate*), 1846
- gateinputcount() (in module *sympy.logic.boolalg*), 1175
- gaunt() (in module *sympy.physics.wigner*), 1568
- gauss\_chebyshev\_t() (in module *sympy.integrals.quadrature*), 613
- gauss\_chebyshev\_u() (in module *sympy.integrals.quadrature*), 614
- gauss\_gen\_laguerre() (in module *sympy.integrals.quadrature*), 612
- gauss\_hermite() (in module *sympy.integrals.quadrature*), 611
- gauss\_jacobi() (in module *sympy.integrals.quadrature*), 615
- gauss\_jordan\_solve() (*sympy.matrices.matrices.MatrixBase* method), 1297
- gauss\_laguerre() (in module *sympy.integrals.quadrature*), 610
- gauss\_legendre() (in module *sympy.integrals.quadrature*), 609
- gauss\_lobatto() (in module *sympy.integrals.quadrature*), 616
- gaussian\_conj() (in module *sympy.physics.optics.gaussopt*), 1867
- gaussian\_reduce() (in module *sympy.solvers.diophantine.diophantine*), 742
- GaussianDomain (class in *sympy.polys.domains.gaussiandomains*), 2533
- GaussianElement (class in *sympy.polys.domains.gaussiandomains*), 2533
- GaussianInteger (class in *sympy.polys.domains.gaussiandomains*), 2536
- GaussianIntegerRing (class in *sympy.polys.domains.gaussiandomains*), 2534
- GaussianRational (class in *sympy.polys.domains.gaussiandomains*), 2539
- GaussianRationalField (class in *sympy.polys.domains.gaussiandomains*), 2536
- gcd() (in module *sympy.polys.polytools*), 2368
- gcd() (*sympy.core.numbers.Number* method), 982
- gcd() (*sympy.polys.domains.ComplexField* method), 2546
- gcd() (*sympy.polys.domains.domain.Domain* method), 2514
- gcd() (*sympy.polys.domains.field.Field* method), 2520
- gcd() (*sympy.polys.domains.gaussiandomains.GaussianInteger* method), 2536
- gcd() (*sympy.polys.domains.GMPYIntegerRing* method), 2529
- gcd() (*sympy.polys.domains.IntegerRing* method), 2527
- gcd() (*sympy.polys.domains.PolynomialRing* method), 2547
- gcd() (*sympy.polys.domains.RealField* method), 2545
- gcd() (*sympy.polys.monomials.Monomial* method), 2429
- gcd() (*sympy.polys.polyclasses.DMP* method), 2563
- gcd() (*sympy.polys.polytools.Poly* method), 2394
- gcd\_list() (in module *sympy.polys.polytools*), 2368
- gcd\_terms() (in module *sympy.core.exprtools*), 1071
- gcdex() (in module *sympy.polys.polytools*), 2365
- gcdex() (*sympy.polys.domains.domain.Domain* method), 2514
- gcdex() (*sympy.polys.domains.GMPYIntegerRing* method), 2529
- gcdex() (*sympy.polys.domains.IntegerRing* method), 2527
- gcdex() (*sympy.polys.domains.PolynomialRing* method), 2547
- gcdex() (*sympy.polys.polyclasses.DMP* method), 2563
- gcdex() (*sympy.polys.polytools.Poly* method), 2395
- Ge (in module *sympy.core.relational*), 1023
- gegenbauer (class in *sympy.polys.domains.gaussiandomains*), 2534

[sympy.functions.special.polynomials](#)),  
[535](#)  
[gegenbauer\\_poly\(\)](#) (in module  
[sympy.polys.orthopolys](#)), [2440](#)  
[gen](#) ([sympy.polys.polytools.Poly](#) property),  
[2395](#)  
[gen](#) ([sympy.series.sequences.SeqBase](#) prop-  
[erty](#)), [634](#)  
[GeneralizedMultivariateLogGamma\(\)](#) (in  
[module sympy.stats](#)), [2938](#)  
[GeneralizedMultivariateLogGammaOmega\(\)](#)  
(in module [sympy.stats](#)), [2939](#)  
[GeneralPythagorean](#) (class in  
[sympy.solvers.diophantine.diophantine](#)),  
[749](#)  
[GeneralSumOfEvenPowers](#) (class in  
[sympy.solvers.diophantine.diophantine](#)),  
[750](#)  
[GeneralSumOfSquares](#) (class in  
[sympy.solvers.diophantine.diophantine](#)),  
[749](#)  
[generate\(\)](#) ([sympy.combinatorics.perm\\_groups](#)  
[method](#)), [307](#)  
[generate\\_bell\(\)](#) (in module  
[sympy.utilities.iterables](#)), [2074](#)  
[generate\\_derangements\(\)](#) (in module  
[sympy.utilities.iterables](#)), [2076](#)  
[generate\\_dimino\(\)](#)  
([sympy.combinatorics.perm\\_groups](#)  
[method](#)), [308](#)  
[generate\\_gray\(\)](#) ([sympy.combinatorics.graycode](#)  
[method](#)), [348](#)  
[generate\\_involutions\(\)](#) (in module  
[sympy.utilities.iterables](#)), [2076](#)  
[generate\\_oriented\\_forest\(\)](#) (in module  
[sympy.utilities.iterables](#)), [2076](#)  
[generate\\_schreier\\_sims\(\)](#)  
([sympy.combinatorics.perm\\_groups](#)  
[method](#)), [309](#)  
[generator](#) ([sympy.polys.numberfields.modules](#)  
[property](#)), [2732](#)  
[generator\\_product\(\)](#)  
([sympy.combinatorics.perm\\_groups](#)  
[method](#)), [309](#)  
[generators](#) ([sympy.combinatorics.perm\\_groups](#)  
[property](#)), [309](#)  
[generators\(\)](#) ([sympy.liealgebras.weyl\\_group](#)  
[method](#)), [2339](#)  
[GeneratorsError](#) (class in  
[sympy.polys.polyerrors](#)), [2644](#)  
[GeneratorsNeeded](#) (class in  
[sympy.polys.polyerrors](#)), [2644](#)  
[genocchi](#) (class in  
[sympy.functions.combinatorial.numbers](#)),  
[442](#)  
[Geometric\(\)](#) (in module [sympy.stats](#)), [2880](#)  
[geometric\\_conj\\_ab\(\)](#) (in module  
[sympy.physics.optics.gaussopt](#)),  
[1867](#)  
[geometric\\_conj\\_af\(\)](#) (in module  
[sympy.physics.optics.gaussopt](#)),  
[1868](#)  
[geometric\\_conj\\_bf\(\)](#) (in module  
[sympy.physics.optics.gaussopt](#)),  
[1868](#)  
[GeometricRay](#) (class in  
[sympy.physics.optics.gaussopt](#)),  
[1862](#)  
[GeometryEntity](#) (class in  
[sympy.geometry.entity](#)), [2194](#)  
[get\(\)](#) ([sympy.core.containers.Dict](#) method),  
[1070](#)  
[get\\_adjacency\\_distance\(\)](#)  
([sympy.combinatorics.permutations](#)  
[method](#)), [268](#)  
[get\\_adjacency\\_matrix\(\)](#)  
([sympy.combinatorics.permutations](#)  
[method](#)), [269](#)  
[get\\_basis\(\)](#) (in module  
[sympy.physics.quantum.represent](#)),  
[1816](#)  
[get\\_class\(\)](#) (in module  
[sympy.utilities.source](#)), [2116](#)  
[get\\_comm\(\)](#) ([sympy.tensor.tensor.\\_TensorManager](#)  
[method](#)), [1422](#)  
[get\\_contraction\\_structure\(\)](#) (in module  
[sympy.tensor.index\\_methods](#)), [1406](#)  
[get\\_data\(\)](#) ([sympy.plotting.plot.Line2D](#)  
[method](#)), [2866](#)  
[get\\_default\\_datatype\(\)](#) (in module  
[sympy.utilities.codegen](#)), [2055](#)  
[get\\_diag\\_blocks\(\)](#)  
([sympy.matrices.common.MatrixCommon](#)  
[method](#)), [1335](#)  
[get\\_diag\\_blocks\(\)](#)  
([sympy.matrices.expressions.blockmatrix](#)  
[method](#)), [1385](#)  
[get\\_domain\(\)](#) ([sympy.polys.polytools.Poly](#)  
[method](#)), [2395](#)  
[get\\_exact\(\)](#) ([sympy.polys.domains.ComplexField](#)  
[method](#)), [2546](#)  
[get\\_exact\(\)](#) ([sympy.polys.domains.domain.Domain](#)  
[method](#)), [2514](#)  
[get\\_exact\(\)](#) ([sympy.polys.domains.RealField](#)  
[method](#)), [2545](#)  
[get\\_field\(\)](#) ([sympy.polys.domains.domain.Domain](#)  
[method](#)), [2514](#)  
[get\\_field\(\)](#) ([sympy.polys.domains.ExpressionDomain](#)

`method`), 2550  
`get_field()` (`sympy.polys.domains.field.Field` `method`), 2520  
`get_field()` (`sympy.polys.domains.FiniteField` `method`), 2525  
`get_field()` (`sympy.polys.domains.gaussiandomains.GaussianField` `method`), 2536  
`get_field()` (`sympy.polys.domains.gaussiandomains.GaussianField` `method`), 2538  
`get_field()` (`sympy.polys.domains.IntegerRing` `method`), 2527  
`get_field()` (`sympy.polys.domains.PolynomialRing` `method`), 2548  
`get_free_indices()` (`sympy.tensor.tensor.TensMul` `method`), 1418  
`get_gen_sol_from_part_sol()` (in module `sympy.solvers.ode.riccati`), 805  
`get_indices()` (in module `sympy.tensor.index_methods`), 1408  
`get_indices()` (`sympy.tensor.tensor.TensMul` `method`), 1418  
`get_interface()` (`sympy.utilities.codegen.FCodeGen` `method`), 2050  
`get_matrix()` (`sympy.tensor.tensor.TensExpr` `method`), 1414  
`get_mod_func()` (in module `sympy.utilities.source`), 2116  
`get_modulus()` (`sympy.polys.polytools.Poly` `method`), 2395  
`get_motion_params()` (in module `sympy.physics.vector.functions`), 1659  
`get_num_denom()` (in module `sympy.polys.numberfields.utilities`), 2738  
`get_period()` (`sympy.functions.special.hyper.meijerg` `method`), 525  
`get_permuted()` (`sympy.physics.secondquant.PermutationOperator` `method`), 1560  
`get_points()` (`sympy.plotting.plot.LineOver1DRangeSeries` `method`), 2866  
`get_points()` (`sympy.plotting.plot.Parametric2DLineSeries` `method`), 2866  
`get_positional_distance()` (`sympy.combinatorics.permutations.Permutation` `method`), 270  
`get_precedence_distance()` (`sympy.combinatorics.permutations.Permutation` `method`), 270  
`get_precedence_matrix()` (`sympy.combinatorics.permutations.Permutation` `method`), 270  
`get_prototype()` (`sympy.utilities.codegen.CCodeGen` `method`), 2048  
`get_prototype()` (`sympy.utilities.codegen.RustCodeGen` `method`), 2053  
`get_resource()` (in module `sympy.utilities.pkgdata`), 2115  
`get_ring()` (`sympy.polys.domains.AlgebraicField` `method`), 2544  
`get_ring()` (`sympy.polys.domains.ComplexField` `method`), 2546  
`get_ring()` (`sympy.polys.domains.domain.Domain` `method`), 2514  
`get_ring()` (`sympy.polys.domains.ExpressionDomain` `method`), 2550  
`get_ring()` (`sympy.polys.domains.field.Field` `method`), 2520  
`get_ring()` (`sympy.polys.domains.FractionField` `method`), 2549  
`get_ring()` (`sympy.polys.domains.gaussiandomains.GaussianField` `method`), 2536  
`get_ring()` (`sympy.polys.domains.gaussiandomains.GaussianField` `method`), 2538  
`get_ring()` (`sympy.polys.domains.GMPYRationalField` `method`), 2532  
`get_ring()` (`sympy.polys.domains.RationalField` `method`), 2531  
`get_ring()` (`sympy.polys.domains.RealField` `method`), 2546  
`get_ring()` (`sympy.polys.domains.ring.Ring` `method`), 2521  
`get_segments()` (`sympy.plotting.plot.MatplotlibBackend` `static method`), 2868  
`get_subNO()` (`sympy.physics.secondquant.NO` `method`), 1558  
`get_subset_from_bitstring()` (`sympy.combinatorics.graycode` `method`), 351  
`get_symmetric_group_sgs()` (in module `sympy.combinatorics.tensor_can`), 1510  
`get_sympy_dir()` (in module `sympy.testing.runtests`), 2032  
`get_target_matrix()` (`sympy.physics.quantum.gate.Gate` `method`), 1843  
`get_target_matrix()` (`sympy.physics.quantum.gate.UGate` `method`), 1845  
`get_units_non_prefixed()` (`sympy.physics.units.unitsystem.UnitSystem` `method`), 1587  
`getn()` (`sympy.core.expr.Expr` `method`), 964  
`getn()` (`sympy.core.expr.Expr` `method`), 964  
`gf_add()` (in module `sympy.polys.galoistools`), 2607

|                                 |                                                         |                              |                                                         |
|---------------------------------|---------------------------------------------------------|------------------------------|---------------------------------------------------------|
| <code>gf_add_ground()</code>    | (in module <code>sympy.polys.galoistools</code> ), 2606 | <code>gf_lshift()</code>     | (in module <code>sympy.polys.galoistools</code> ), 2610 |
| <code>gf_add_mul()</code>       | (in module <code>sympy.polys.galoistools</code> ), 2608 | <code>gf_monic()</code>      | (in module <code>sympy.polys.galoistools</code> ), 2613 |
| <code>gf_berlekamp()</code>     | (in module <code>sympy.polys.galoistools</code> ), 2618 | <code>gf_mul()</code>        | (in module <code>sympy.polys.galoistools</code> ), 2608 |
| <code>gf_cofactors()</code>     | (in module <code>sympy.polys.galoistools</code> ), 2612 | <code>gf_mul_ground()</code> | (in module <code>sympy.polys.galoistools</code> ), 2607 |
| <code>gf_compose()</code>       | (in module <code>sympy.polys.galoistools</code> ), 2614 | <code>gf_multi_eval()</code> | (in module <code>sympy.polys.galoistools</code> ), 2614 |
| <code>gf_compose_mod()</code>   | (in module <code>sympy.polys.galoistools</code> ), 2614 | <code>gf_neg()</code>        | (in module <code>sympy.polys.galoistools</code> ), 2606 |
| <code>gf_crt()</code>           | (in module <code>sympy.polys.galoistools</code> ), 2602 | <code>gf_normal()</code>     | (in module <code>sympy.polys.galoistools</code> ), 2605 |
| <code>gf_crt1()</code>          | (in module <code>sympy.polys.galoistools</code> ), 2602 | <code>gf_pow()</code>        | (in module <code>sympy.polys.galoistools</code> ), 2611 |
| <code>gf_crt2()</code>          | (in module <code>sympy.polys.galoistools</code> ), 2603 | <code>gf_pow_mod()</code>    | (in module <code>sympy.polys.galoistools</code> ), 2611 |
| <code>gf_csolve()</code>        | (in module <code>sympy.polys.galoistools</code> ), 2620 | <code>gf_Qbasis()</code>     | (in module <code>sympy.polys.galoistools</code> ), 2617 |
| <code>gf_degree()</code>        | (in module <code>sympy.polys.galoistools</code> ), 2603 | <code>gf_Qmatrix()</code>    | (in module <code>sympy.polys.galoistools</code> ), 2617 |
| <code>gf_diff()</code>          | (in module <code>sympy.polys.galoistools</code> ), 2613 | <code>gf_quo()</code>        | (in module <code>sympy.polys.galoistools</code> ), 2610 |
| <code>gf_div()</code>           | (in module <code>sympy.polys.galoistools</code> ), 2609 | <code>gf_quo_ground()</code> | (in module <code>sympy.polys.galoistools</code> ), 2607 |
| <code>gf_eval()</code>          | (in module <code>sympy.polys.galoistools</code> ), 2613 | <code>gf_random()</code>     | (in module <code>sympy.polys.galoistools</code> ), 2615 |
| <code>gf_expand()</code>        | (in module <code>sympy.polys.galoistools</code> ), 2609 | <code>gf_rem()</code>        | (in module <code>sympy.polys.galoistools</code> ), 2609 |
| <code>gf_exquo()</code>         | (in module <code>sympy.polys.galoistools</code> ), 2610 | <code>gf_rshift()</code>     | (in module <code>sympy.polys.galoistools</code> ), 2611 |
| <code>gf_factor()</code>        | (in module <code>sympy.polys.galoistools</code> ), 2619 | <code>gf_shoup()</code>      | (in module <code>sympy.polys.galoistools</code> ), 2618 |
| <code>gf_factor_sqf()</code>    | (in module <code>sympy.polys.galoistools</code> ), 2618 | <code>gf_sqf_list()</code>   | (in module <code>sympy.polys.galoistools</code> ), 2616 |
| <code>gf_from_dict()</code>     | (in module <code>sympy.polys.galoistools</code> ), 2605 | <code>gf_sqf_p()</code>      | (in module <code>sympy.polys.galoistools</code> ), 2616 |
| <code>gf_from_int_poly()</code> | (in module <code>sympy.polys.galoistools</code> ), 2605 | <code>gf_sqf_part()</code>   | (in module <code>sympy.polys.galoistools</code> ), 2616 |
| <code>gf_gcd()</code>           | (in module <code>sympy.polys.galoistools</code> ), 2612 | <code>gf_sqr()</code>        | (in module <code>sympy.polys.galoistools</code> ), 2608 |
| <code>gf_gcdex()</code>         | (in module <code>sympy.polys.galoistools</code> ), 2612 | <code>gf_strip()</code>      | (in module <code>sympy.polys.galoistools</code> ), 2604 |
| <code>gf_int()</code>           | (in module <code>sympy.polys.galoistools</code> ), 2603 | <code>gf_sub()</code>        | (in module <code>sympy.polys.galoistools</code> ), 2607 |
| <code>gf_irreducible()</code>   | (in module <code>sympy.polys.galoistools</code> ), 2615 | <code>gf_sub_ground()</code> | (in module <code>sympy.polys.galoistools</code> ), 2606 |
| <code>gf_irreducible_p()</code> | (in module <code>sympy.polys.galoistools</code> ), 2615 | <code>gf_sub_mul()</code>    | (in module <code>sympy.polys.galoistools</code> ), 2608 |
| <code>gf_LC()</code>            | (in module <code>sympy.polys.galoistools</code> ), 2604 | <code>gf_TC()</code>         | (in module <code>sympy.polys.galoistools</code> ), 2604 |
| <code>gf_lcm()</code>           | (in module <code>sympy.polys.galoistools</code> ), 2612 | <code>gf_to_dict()</code>    | (in module <code>sympy.polys.galoistools</code> ), 2605 |



- [gf\\_to\\_int\\_poly\(\)](#) (in module [sympy.polys.galoistools](#)), 2606  
[gf\\_trace\\_map\(\)](#) (in module [sympy.polys.galoistools](#)), 2614  
[gf\\_trunc\(\)](#) (in module [sympy.polys.galoistools](#)), 2604  
[gf\\_value\(\)](#) (in module [sympy.polys.galoistools](#)), 2620  
[gf\\_zassenhaus\(\)](#) (in module [sympy.polys.galoistools](#)), 2618  
[gff\(\)](#) (in module [sympy.polys.polytools](#)), 2371  
[gff\\_list\(\)](#) (in module [sympy.polys.polytools](#)), 2371  
[gff\\_list\(\)](#) ([sympy.polys.polyclasses.DMP](#) method), 2563  
[gff\\_list\(\)](#) ([sympy.polys.polytools.Poly](#) method), 2396  
[given\(\)](#) (in module [sympy.stats](#)), 2964  
[gm\\_private\\_key\(\)](#) (in module [sympy.crypto.crypto](#)), 2797  
[gm\\_public\\_key\(\)](#) (in module [sympy.crypto.crypto](#)), 2797  
[GMPYFiniteField](#) (class in [sympy.polys.domains](#)), 2525  
[GMPYIntegerRing](#) (class in [sympy.polys.domains](#)), 2528  
[GMPYRationalField](#) (class in [sympy.polys.domains](#)), 2531  
[GoldenRatio](#) (class in [sympy.core.numbers](#)), 1003  
[Gompertz\(\)](#) (in module [sympy.stats](#)), 2906  
[gosper\\_normal\(\)](#) (in module [sympy.concrete.gosper](#)), 916  
[gosper\\_sum\(\)](#) (in module [sympy.concrete.gosper](#)), 917  
[gosper\\_term\(\)](#) (in module [sympy.concrete.gosper](#)), 917  
[goto](#) (class in [sympy.codegen.cnodes](#)), 1149  
[GoTo](#) (class in [sympy.codegen.fnodes](#)), 1151  
[gouy](#) ([sympy.physics.optics.gaussopt.BeamParameter](#) property), 1858  
[GradedLexOrder](#) (class in [sympy.polys.orderings](#)), 2431  
[gradient\(\)](#) (in module [sympy.physics.vector.fieldfunctions](#)), 1671  
[gradient\(\)](#) (in module [sympy.vector](#)), 1472  
[gradient\(\)](#) ([sympy.vector.deloperator.Del](#) method), 1461  
[GramSchmidt\(\)](#) (in module [sympy.matrices.dense](#)), 1321  
[gray\\_to\\_bin\(\)](#) ([sympy.combinatorics.graycode](#) method), 350  
[GrayCode](#) (class in [sympy.combinatorics.graycode](#)), 347  
[graycode\\_subsets\(\)](#) ([sympy.combinatorics.graycode](#) method), 351  
[GreaterThan](#) (class in [sympy.core.relational](#)), 1024  
[greek\\_letters](#) (in module [sympy.printing.pretty.pretty\\_symbology](#)), 2184  
[groebner\(\)](#) (in module [sympy.polys.groebnertools](#)), 2639  
[groebner\(\)](#) (in module [sympy.polys.polytools](#)), 2377  
[GroebnerBasis](#) (class in [sympy.polys.polytools](#)), 2423  
[ground\\_roots\(\)](#) (in module [sympy.polys.polytools](#)), 2375  
[ground\\_roots\(\)](#) ([sympy.polys.polytools.Poly](#) method), 2396  
[group\(\)](#) (in module [sympy.utilities.iterables](#)), 2077  
[group\\_name\(\)](#) ([sympy.liealgebras.weyl\\_group.WeylGroup](#) method), 2339  
[group\\_order\(\)](#) ([sympy.liealgebras.weyl\\_group.WeylGroup](#) method), 2339  
[grover\\_iteration\(\)](#) (in module [sympy.physics.quantum.grover](#)), 1849  
[gruntz\(\)](#) (in module [sympy.series.gruntz](#)), 623  
[Gt](#) (in module [sympy.core.relational](#)), 1023  
[Gumbel\(\)](#) (in module [sympy.stats](#)), 2907
- ## H
- [H](#) (in module [sympy.physics.quantum.gate](#)), 1844  
[H](#) ([sympy.matrices.common.MatrixCommon](#) property), 1327  
[HadamardGate](#) (class in [sympy.physics.quantum.gate](#)), 1844  
[HadamardPower](#) (class in [sympy.matrices.expressions](#)), 1374  
[HadamardProduct](#) (class in [sympy.matrices.expressions](#)), 1373  
[Half](#) (class in [sympy.core.numbers](#)), 996  
[half\\_gcdex\(\)](#) (in module [sympy.polys.polytools](#)), 2364  
[half\\_gcdex\(\)](#) ([sympy.polys.domains.domain.Domain](#) method), 2514  
[half\\_gcdex\(\)](#) ([sympy.polys.polyclasses.DMP](#) method), 2563

`half_gcdex()` (sympy.polys.polytools.Poly method), 2396  
`half_per()` (sympy.polys.polyclasses.DMF method), 2568  
`half_wave_retarder()` (in module sympy.physics.optics.polarization), 1872  
`handler` (sympy.assumptions.assume.Predicate attribute), 196  
`handler` (sympy.assumptions.predicates.calculus.FinitePredicate attribute), 203  
`handler` (sympy.assumptions.predicates.calculus.InfinitePredicate attribute), 203  
`handler` (sympy.assumptions.predicates.common.CohenPredicate attribute), 202  
`handler` (sympy.assumptions.predicates.common.HolierPredicate attribute), 202  
`handler` (sympy.assumptions.predicates.matrices.CoblePredicate attribute), 213  
`handler` (sympy.assumptions.predicates.matrices.DiligoPredicate attribute), 210  
`handler` (sympy.assumptions.predicates.matrices.FullerPredicate attribute), 210  
`handler` (sympy.assumptions.predicates.matrices.HolzerPredicate attribute), 212  
`handler` (sympy.assumptions.predicates.matrices.InvertiblePredicate attribute), 205  
`handler` (sympy.assumptions.predicates.matrices.KolvenPredicate attribute), 209  
`handler` (sympy.assumptions.predicates.matrices.KolvenByPredicate attribute), 214  
`handler` (sympy.assumptions.predicates.matrices.KotlerPredicate attribute), 206  
`handler` (sympy.assumptions.predicates.matrices.KotlerByPredicate attribute), 208  
`handler` (sympy.assumptions.predicates.matrices.KotlerByPredicate attribute), 212  
`handler` (sympy.assumptions.predicates.matrices.KotlerByPredicate attribute), 214  
`handler` (sympy.assumptions.predicates.matrices.KotlerByPredicate attribute), 211  
`handler` (sympy.assumptions.predicates.matrices.SymmetricPredicate attribute), 204  
`handler` (sympy.assumptions.predicates.matrices.TriangularPredicate attribute), 215  
`handler` (sympy.assumptions.predicates.matrices.KotlerByPredicate attribute), 207  
`handler` (sympy.assumptions.predicates.matrices.KotlerByPredicate attribute), 216  
`handler` (sympy.assumptions.predicates.matrices.KotlerByPredicate attribute), 208  
`handler` (sympy.assumptions.predicates.ntheory.CompositePredicate attribute), 218  
`handler` (sympy.assumptions.predicates.ntheory.EvenPredicate attribute), 216  
`handler` (sympy.assumptions.predicates.ntheory.OddPredicate attribute), 217  
`handler` (sympy.assumptions.predicates.ntheory.PrimePredicate attribute), 218  
`handler` (sympy.assumptions.predicates.order.NegativePredicate attribute), 220  
`handler` (sympy.assumptions.predicates.order.NonNegativePredicate attribute), 224  
`handler` (sympy.assumptions.predicates.order.NonPositivePredicate attribute), 223  
`handler` (sympy.assumptions.predicates.order.NonZeroPredicate attribute), 222  
`handler` (sympy.assumptions.predicates.order.PositivePredicate attribute), 219  
`handler` (sympy.assumptions.predicates.order.ZeroPredicate attribute), 221  
`handler` (sympy.assumptions.predicates.sets.AlgebraicPredicate attribute), 232  
`handler` (sympy.assumptions.predicates.sets.AntihermitianPredicate attribute), 231  
`handler` (sympy.assumptions.predicates.sets.ComplexPredicate attribute), 230  
`handler` (sympy.assumptions.predicates.sets.ExtendedRealPredicate attribute), 228  
`handler` (sympy.assumptions.predicates.sets.HermitianPredicate attribute), 229  
`handler` (sympy.assumptions.predicates.sets.ImaginaryPredicate attribute), 231  
`handler` (sympy.assumptions.predicates.sets.IntegerPredicate attribute), 225  
`handler` (sympy.assumptions.predicates.sets.IrrationalPredicate attribute), 226  
`handler` (sympy.assumptions.predicates.sets.RationalPredicate attribute), 225  
`handler` (sympy.assumptions.predicates.sets.RealPredicate attribute), 228  
`handler` (sympy.assumptions.predicates.sets.TranscendentalPredicate attribute), 232  
`hankel1` (class in sympy.functions.special.bessel), 500  
`hankel2` (class in sympy.functions.special.bessel), 500  
`hankel_transform()` (in module sympy.integrals.transforms), 584  
`hankel_transform_predicate` (class in sympy.integrals.transforms), 585  
`hankel_transform_predicate` (class in sympy.integrals.transforms), 585  
`hankel_transform_predicate` (class in sympy.integrals.transforms), 585  
`hankel_transform_predicate` (class in sympy.integrals.transforms), 585  
`has()` (sympy.core.basic.Basic method), 933  
`has_even_predicate` (sympy.matrices.common.MatrixCommon attribute), 218

`method`), 1335  
`has_assoc_Field` (`sympy.polys.domains.domain.Domain` attribute), 2514  
`has_assoc_Ring` (`sympy.polys.domains.domain.Domain` attribute), 2514  
`has_dups()` (in module `sympy.utilities.iterables`), 2077  
`has_empty_sequence` (`sympy.concrete.expr_with_intlimits.ExprWithIntLimits` property), 912  
`has_finite_limits` (`sympy.concrete.expr_with_limits.ExprWithLimits` property), 607  
`has_free()` (`sympy.core.basic.Basic` method), 934  
`has_integer_powers()` (`sympy.physics.units.dimensions.Dimension` method), 1584  
`has_only_gens()` (`sympy.polys.polytools.Poly` method), 2396  
`has_q_annihilators` (`sympy.physics.secondquant.NO` property), 1559  
`has_q_creators` (`sympy.physics.secondquant.NO` property), 1559  
`has_reversed_limits` (`sympy.concrete.expr_with_limits.ExprWithLimits` property), 607  
`has_variety()` (in module `sympy.utilities.iterables`), 2077  
`HBar` (class in `sympy.physics.quantum.constants`), 1796  
`Heaviside` (class in `sympy.functions.special.delta_functions`), 454  
`height` (`sympy.categories.diagram_drawing.DiagramGroup` property), 2755  
`height` (`sympy.physics.optics.gaussopt.GeometricRay` property), 1863  
`height()` (`sympy.polys.agca.ideals.Ideal` method), 2462  
`height()` (`sympy.printing.pretty.stringpict.stringPict` method), 2187  
`hermite` (class in `sympy.functions.special.polynomials`), 541  
`Hermite()` (in module `sympy.stats`), 2881  
`hermite_normal_form()` (in module `sympy.matrices.normalforms`), 1386  
`hermite_normal_form()` (in module `sympy.polys.matrices.normalforms`), 2700  
`hermite_poly()` (in module `sympy.polys.orthopolys`), 2440  
`hermitian`, 925  
`HermitianOperator` (class in `sympy.physics.quantum.operator`), 1809  
`HermitianPredicate` (class in `sympy.assumptions.predicates.sets`), 228  
`hessian()` (in module `sympy.matrices.dense`), 1120  
`heuristic()` (in module `sympy.integrals.heuristic`), 596  
`HeuristicGCDFailed` (class in `sympy.polys.polyerrors`), 2643  
`highest_root()` (`sympy.liealgebras.type_a.TypeA` method), 2329  
`HilbertSpace` (class in `sympy.physics.quantum.hilbert`), 1804  
`hobj()` (in module `sympy.printing.pretty.pretty_symbology`), 2185  
`HolonomicFunction` (class in `sympy.holonomic.holonomic`), 2314  
`Holzer()` (in module `sympy.solvers.diophantine.diophantine`), 742  
`Hom()` (`sympy.categories.Diagram` method), 2750  
`homogeneous_order()` (in module `sympy.solvers.ode`), 763  
`homogeneous_order()` (`sympy.polys.polyclasses.DMP` method), 2564  
`homogeneous_order()` (`sympy.polys.polytools.Poly` method), 2397  
`HomogeneousCoeffBest` (class in `sympy.solvers.ode.single`), 772  
`HomogeneousCoeffSubsDepDivIndep` (class in `sympy.solvers.ode.single`), 773  
`HomogeneousCoeffSubsIndepDivDep` (class in `sympy.solvers.ode.single`), 774  
`HomogeneousGeneralQuadratic` (class in `sympy.solvers.diophantine.diophantine`), 748  
`HomogeneousTernaryQuadratic` (class in `sympy.solvers.diophantine.diophantine`), 748  
`HomogeneousTernaryQuadraticNormal` (class in `sympy.solvers.diophantine.diophantine`), 747  
`homogenize()` (`sympy.polys.polyclasses.DMP` method), 2564  
`homogenize()` (`sympy.polys.polytools.Poly`

method), 2397  
 homomorphism() (in module `sympy.polys.agca.homomorphisms`), 2469  
 HomomorphismFailed (class in `sympy.polys.polyerrors`), 2643  
 horner() (in module `sympy.polys.polyfuncs`), 2425  
 hradius (`sympy.geometry.ellipse.Ellipse` property), 2259  
 hstack() (`sympy.matrices.common.MatrixCommon` class method), 1336  
 hstack() (`sympy.polys.matrices.ddm.DDM` method), 2690  
 hstack() (`sympy.polys.matrices.domainmatrix.DomainMatrix` method), 2678  
 hstack() (`sympy.polys.matrices.sdm.SDM` method), 2695  
 hyper (class in `sympy.functions.special.hyper`), 520  
 hyper\_algorithm() (in module `sympy.series.formal`), 658  
 hyper\_re() (in module `sympy.series.formal`), 656  
 HyperbolicFunction (class in `sympy.functions.elementary.hyperbolic`), 402  
 hyperexpand() (in module `sympy.simplify.hyperexpand`), 687  
 hyperfocal\_distance() (in module `sympy.physics.optics.utils`), 1883  
 Hypergeometric() (in module `sympy.stats`), 2879  
 hypersimilar() (in module `sympy.simplify.simplify`), 667  
 hypersimp() (in module `sympy.simplify.simplify`), 667  
 hypot (class in `sympy.codegen.cfunctions`), 1146  
 |  
 ibin() (in module `sympy.utilities.iterables`), 2078  
 Ideal (class in `sympy.polys.agca.ideals`), 2461  
 ideal() (`sympy.polys.domains.ring.Ring` method), 2521  
 Identity (class in `sympy.matrices.expressions`), 1378  
 identity (`sympy.combinatorics.perm_groups.PermGroup` property), 309  
 identity\_hom() (`sympy.polys.agca.modules.FreeModule` method), 2454  
 identity\_hom() (`sympy.polys.agca.modules.Module` method), 2453  
 identity\_hom() (`sympy.polys.agca.modules.QuotientModule` method), 2465  
 identity\_hom() (`sympy.polys.agca.modules.SubModule` method), 2457  
 IdentityFunction (class in `sympy.functions.elementary.miscellaneous`), 420  
 IdentityGate (class in `sympy.physics.quantum.gate`), 1844  
 IdentityMorphism (class in `sympy.categories`), 2746  
 IdentityOperator (class in `sympy.physics.quantum.operator`), 1809  
 idiff() (in module `sympy.geometry.util`), 2201  
 Idx (class in `sympy.tensor.indexed`), 1400  
 ifft() (in module `sympy.discrete.transforms`), 1084  
 ifwht() (in module `sympy.discrete.transforms`), 1087  
 igcd() (in module `sympy.core.numbers`), 994  
 ignore\_warnings() (in module `sympy.utilities.exceptions`), 2067  
 ilcm() (in module `sympy.core.numbers`), 994  
 ild\_moment (`sympy.physics.continuum_mechanics.beam.Beam` property), 1956  
 ild\_reactions (`sympy.physics.continuum_mechanics.beam.Beam` property), 1956  
 ild\_shear (`sympy.physics.continuum_mechanics.beam.Beam` property), 1956  
 im (class in `sympy.functions.elementary.complexes`), 383  
 image() (`sympy.polys.agca.homomorphisms.ModuleHomomorphism` method), 2470  
 ImageSet (class in `sympy.sets.fancysets`), 1206  
 imageset() (in module `sympy.sets.sets`), 1193  
 imaginary, 924  
 ImaginaryPredicate (class in `sympy.assumptions.predicates.sets`), 230  
 ImaginaryUnit (class in `sympy.core.numbers`), 1000  
 ImmutableDenseMatrix (class in `sympy.matrices.immutable`), 1369  
 ImmutableDenseNDimArray (class in `sympy.tensor.array`), 1392  
 ImmutableMatrix (in module `sympy.matrices.immutable`), 1369  
 ImmutableSparseMatrix (class in



- `sympy.matrices.immutable`), 1364
- `ImmutableSparseNDimArray` (class in `sympy.tensor.array`), 1392
- `implemented_function()` (in module `sympy.utilities.lambdify`), 2099
- `implicit_application()` (in module `sympy.parsing.sympy_parser`), 2127
- `implicit_multiplication()` (in module `sympy.parsing.sympy_parser`), 2127
- `implicit_multiplication_application()` (in module `sympy.parsing.sympy_parser`), 2128
- `ImplicitRegion` (class in `sympy.vector.implicitregion`), 1462
- `ImplicitSeries` (class in `sympy.plotting.plot_implicit`), 2867
- `ImpliedDoLoop` (class in `sympy.codegen.fnodes`), 1151
- `Implies` (class in `sympy.logic.boolalg`), 1170
- `impulse_response_numerical_data()` (`sympy.physics.control.control_plots` method), 1942
- `impulse_response_plot()` (`sympy.physics.control.control_plots` method), 1940
- `imul_num()` (`sympy.polys.rings.PolyElement` method), 2557
- `in_terms_of_generators()` (`sympy.polys.agca.modules.SubModule` method), 2457
- `incenter` (`sympy.geometry.polygon.Triangle` property), 2298
- `incircle` (`sympy.geometry.polygon.RegularPolygon` property), 2289
- `incircle` (`sympy.geometry.polygon.Triangle` property), 2298
- `inclusion_hom()` (`sympy.polys.agca.modules.SubModule` method), 2457
- `indent_code()` (`sympy.printing.c.C89CodePrinter` method), 2141
- `indent_code()` (`sympy.printing.fortran.FCodePrinter` method), 2149
- `indent_code()` (`sympy.printing.jscode.JavascriptCodePrinter` method), 2155
- `indent_code()` (`sympy.printing.julia.JuliaCodePrinter` method), 2158
- `indent_code()` (`sympy.printing.octave.OctaveCodePrinter` method), 2161
- `indent_code()` (`sympy.printing.rcode.RCodePrinter` method), 2144
- `indent_code()` (`sympy.printing.rust.RustCodePrinter` method), 2164
- `index()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 309
- `index()` (`sympy.combinatorics.permutations.Permutation` method), 271
- `index()` (`sympy.concrete.expr_with_intlimits.ExprWithIntLimits` method), 913
- `index()` (`sympy.core.containers.Tuple` method), 1069
- `index()` (`sympy.physics.secondquant.FixedBosonicBasis` method), 1551
- `index()` (`sympy.physics.secondquant.VarBosonicBasis` method), 1561
- `index()` (`sympy.polys.rings.PolyRing` method), 2554
- `index_vector()` (`sympy.algebras.Quaternion` method), 890
- `Indexed` (class in `sympy.tensor.indexed`), 1402
- `IndexedBase` (class in `sympy.tensor.indexed`), 1404
- `indices` (`sympy.tensor.indexed.Indexed` property), 1402
- `indices_contain_equal_information` (`sympy.functions.special.tensor_functions.KroneckerDelta` property), 550
- `indices_contain_equal_information` (`sympy.physics.secondquant.KroneckerDelta` property), 1554
- `inertia()` (in module `sympy.physics.mechanics.functions`), 1750
- `inertia_of_point_mass()` (in module `sympy.physics.mechanics.functions`), 1751
- `infinite` (`sympy.sets.sets.Set` property), 1187
- `infinite`, 925
- `infinite` (`sympy.series.formal.FormalPowerSeries` property), 648
- `InModulePredicate` (class in `sympy.assumptions.predicates.calculus`), 203
- `infinitesimals()` (in module `sympy.solvers.ode`), 764
- `Infinity` (class in `sympy.core.numbers`), 998
- `InhomogeneousGeneralQuadratic` (class in `sympy.solvers.diophantine.diophantine`), 748
- `InhomogeneousTernaryQuadratic` (class in `sympy.solvers.diophantine.diophantine`), 747
- `init_ipython_session()` (in module `sympy.interactive.session`), 2116
- `init_printing()` (in module `sympy.interactive.printing`), 2119
- `init_python_session()` (in module `sympy.interactive.session`), 2116

[init\\_session\(\)](#) (in module [sympy.interactive.session](#)), 2117  
[init\\_vprinting\(\)](#) (in module [sympy.physics.vector.printing](#)), 1662  
[initial](#) ([sympy.series.sequences.RecursiveSequence](#) property), 639  
[inject\(\)](#) ([sympy.polys.domains.compositedomain.CompositeDomain](#) method), 2522  
[inject\(\)](#) ([sympy.polys.domains.domain.Domain](#) method), 2514  
[inject\(\)](#) ([sympy.polys.domains.gaussiandomain.GaussianDomain](#) method), 2533  
[inject\(\)](#) ([sympy.polys.domains.simplesdomain.SimpleDomain](#) method), 2522  
[inject\(\)](#) ([sympy.polys.polyclasses.DMP](#) method), 2564  
[inject\(\)](#) ([sympy.polys.polytools.Poly](#) method), 2397  
[inner\\_endomorphism\(\)](#) ([sympy.polys.numberfields.modules.Endomorphism](#) method), 2734  
[InnerEndomorphism](#) (class in [sympy.polys.numberfields.modules](#)), 2734  
[InnerProduct](#) (class in [sympy.physics.quantum.innerproduct](#)), 1798  
[InnerProduct](#) (class in [sympy.physics.secondquant](#)), 1553  
[inradius](#) ([sympy.geometry.polygon.RegularPolygon](#) property), 2290  
[inradius](#) ([sympy.geometry.polygon.Triangle](#) property), 2298  
[int\\_to\\_Integer\(\)](#) (in module [sympy.interactive.session](#)), 2118  
[IntBaseType](#) (class in [sympy.codegen.ast](#)), 1134  
[intcurve\\_diffgeu\(\)](#) (in module [sympy.diffgeom](#)), 2816  
[intcurve\\_series\(\)](#) (in module [sympy.diffgeom](#)), 2814  
[integer](#), 924  
[Integer](#) (class in [sympy.core.numbers](#)), 987  
[integer\\_log\(\)](#) (in module [sympy.core.power](#)), 1008  
[integer\\_nthroot\(\)](#) (in module [sympy.core.power](#)), 1008  
[integer\\_to\\_term\(\)](#) (in module [sympy.logic.boolalg](#)), 1180  
[IntegerElementsPredicate](#) (class in [sympy.assumptions.predicates.matrices](#)), 211  
[IntegerPartition](#) (class in [sympy.combinatorics.partitions](#)),  
[IntegerPredicate](#) (class in [sympy.assumptions.predicates.sets](#)), 224  
[IntegerRing](#) (class in [sympy.polys.domains](#)), 2526  
[IntegerSetDomain](#) (class in [sympy.sets.fancysets](#)), 1204  
[Integral](#) (class in [sympy.integrals.integrals](#)), 601  
[IntegralDomain](#) (class in [sympy.integrals.integrals](#)), 601  
[IntegralDomain.is\\_commutative](#) (in module [sympy.integrals.integrals](#)), 601  
[IntegralDomain.is\\_integral\\_domain](#) (in module [sympy.integrals.integrals](#)), 601  
[integral\\_steps\(\)](#) (in module [sympy.integrals.manualintegrate](#)), 596  
[IntegralTransform](#) (class in [sympy.integrals.transforms](#)), 586  
[IntegralTransformError](#), 587  
[integrand\(\)](#) ([sympy.functions.special.hyper.meijerg](#) method), 525  
[integrate\(\)](#) (in module [sympy.integrals.integrals](#)), 598  
[integrate\(\)](#) ([sympy.algebras.Quaternion](#) method), 890  
[integrate\(\)](#) ([sympy.core.expr.Expr](#) method), 964  
[integrate\(\)](#) ([sympy.core.relational.Equality](#) method), 1024  
[integrate\(\)](#) ([sympy.holonomic.holonomic.HolonomicFunction](#) method), 2318  
[integrate\(\)](#) ([sympy.matrices.matrices.MatrixCalculus](#) method), 1279  
[integrate\(\)](#) ([sympy.polys.polyclasses.DMP](#) method), 2564  
[integrate\(\)](#) ([sympy.polys.polytools.Poly](#) method), 2398  
[integrate\(\)](#) ([sympy.series.formal.FormalPowerSeries](#) method), 648  
[integrate\\_result\(\)](#) (in module [sympy.physics.quantum.represent](#)), 1816  
[interior](#) ([sympy.sets.sets.Set](#) property), 1187  
[interior\\_angle](#) ([sympy.geometry.polygon.RegularPolygon](#) property), 2290  
[internal\\_forces](#) ([sympy.physics.continuum\\_mechanics.truss](#) property), 2024  
[interpolate\(\)](#) (in module [sympy.polys.polyfuncs](#)), 2426  
[interpolating\\_poly\(\)](#) (in module [sympy.polys.specialpolys](#)), 2438  
[interpolating\\_spline\(\)](#) (in module

`sympy.functions.special.bsplines)`, 513  
`intersect()` (`sympy.polys.agca.ideals.Ideal` method), 2462  
`intersect()` (`sympy.polys.agca.modules.SubModule` method), 2457  
`intersect()` (`sympy.sets.sets.Set` method), 1187  
`intersecting_product()` (in module `sympy.discrete.convolution`), 1094  
`Intersection` (class in `sympy.sets.sets`), 1198  
`intersection()` (in module `sympy.geometry.util`), 2197  
`intersection()` (`sympy.geometry.ellipse.Circle` method), 2271  
`intersection()` (`sympy.geometry.ellipse.Ellipse` method), 2260  
`intersection()` (`sympy.geometry.entity.GeometricEntity` method), 2194  
`intersection()` (`sympy.geometry.line.LinearEntity` method), 2220  
`intersection()` (`sympy.geometry.plane.Plane` method), 2306  
`intersection()` (`sympy.geometry.point.Point` method), 2205  
`intersection()` (`sympy.geometry.point.Point3D` method), 2213  
`intersection()` (`sympy.geometry.polygon.Polygon` method), 2280  
`intersection()` (`sympy.sets.sets.Set` method), 1188  
`Interval` (class in `sympy.sets.sets`), 1194  
`interval` (`sympy.series.sequences.RecursiveSeq` property), 639  
`interval` (`sympy.series.sequences.SeqBase` property), 634  
`intervals()` (in module `sympy.polys.polytools`), 2374  
`intervals()` (`sympy.polys.polyclasses.DMP` method), 2564  
`intervals()` (`sympy.polys.polytools.Poly` method), 2398  
`IntQubit` (class in `sympy.physics.quantum.qubit`), 1850  
`IntQubitBra` (class in `sympy.physics.quantum.qubit`), 1852  
`intt()` (in module `sympy.discrete.transforms`), 1085  
`inv()` (`sympy.matrices.matrices.MatrixBase` method), 1299  
`inv()` (`sympy.polys.matrices.ddm.DDM` method), 2690  
`inv()` (`sympy.polys.matrices.domainmatrix.DomainMatrix` method), 2678  
`inv()` (`sympy.polys.matrices.sdm.SDM` method), 2695  
`inv_can_transf_matrix` (`sympy.physics.units.dimensions.DimensionSystem` property), 1585  
`inv_mod()` (`sympy.matrices.matrices.MatrixBase` method), 1301  
`Inverse` (class in `sympy.matrices.expressions`), 1374  
`inverse()` (`sympy.algebras.Quaternion` method), 891  
`inverse()` (`sympy.functions.elementary.exponential.log` method), 414  
`inverse()` (`sympy.functions.elementary.hyperbolic.acosh` method), 406  
`inverse()` (`sympy.functions.elementary.hyperbolic.acoth` method), 407  
`inverse()` (`sympy.functions.elementary.hyperbolic.acsch` method), 408  
`inverse()` (`sympy.functions.elementary.hyperbolic.asech` method), 408  
`inverse()` (`sympy.functions.elementary.hyperbolic.asinh` method), 405  
`inverse()` (`sympy.functions.elementary.hyperbolic.atanh` method), 406  
`inverse()` (`sympy.functions.elementary.hyperbolic.coth` method), 404  
`inverse()` (`sympy.functions.elementary.hyperbolic.sinh` method), 402  
`inverse()` (`sympy.functions.elementary.hyperbolic.tanh` method), 403  
`inverse()` (`sympy.functions.elementary.trigonometric.acos` method), 396  
`inverse()` (`sympy.functions.elementary.trigonometric.acot` method), 398  
`inverse()` (`sympy.functions.elementary.trigonometric.acsc` method), 400  
`inverse()` (`sympy.functions.elementary.trigonometric.asec` method), 399  
`inverse()` (`sympy.functions.elementary.trigonometric.asin` method), 396  
`inverse()` (`sympy.functions.elementary.trigonometric.atan` method), 397  
`inverse()` (`sympy.functions.elementary.trigonometric.cot` method), 393  
`inverse()` (`sympy.functions.elementary.trigonometric.tan` method), 392  
`inverse()` (`sympy.functions.special.error_functions.erf` method), 473  
`inverse()` (`sympy.functions.special.error_functions.erfc` method), 474  
`inverse()` (`sympy.functions.special.error_functions.erfcinv` method), 479  
`inverse()` (`sympy.functions.special.error_functions.erfinv` method), 479

`method`), 478  
`inverse()` (`sympy.polys.agca.extensions.ExtensionElement` method), 2514  
`method`), 2476  
`inverse()` (`sympy.series.formal.FormalPowerSeries` method), 2521  
`method`), 649  
`inverse_ADJ()` (`sympy.matrices.matrices.MatrixBase` method), 2568  
`method`), 1301  
`inverse_BLOCK()` (`sympy.matrices.matrices.MatrixBase` method), 2564  
`method`), 1301  
`inverse_CH()` (`sympy.matrices.matrices.MatrixBase` method), 2399  
`method`), 1301  
`inverse_cosine_transform()` (in module `sympy.integrals.transforms`), 583  
`inverse_fourier_transform()` (in module `sympy.integrals.transforms`), 580  
`inverse_GE()` (`sympy.matrices.matrices.MatrixBase` method), 1301  
`inverse_hankel_transform()` (in module `sympy.integrals.transforms`), 585  
`inverse_laplace_transform()` (in module `sympy.integrals.transforms`), 579  
`inverse_LDL()` (`sympy.matrices.matrices.MatrixBase` method), 1301  
`inverse_LU()` (`sympy.matrices.matrices.MatrixBase` method), 1302  
`inverse_mellin_transform()` (in module `sympy.integrals.transforms`), 577  
`inverse_mobius_transform()` (in module `sympy.discrete.transforms`), 1088  
`inverse_QR()` (`sympy.matrices.matrices.MatrixBase` method), 1302  
`inverse_sine_transform()` (in module `sympy.integrals.transforms`), 582  
`InverseCosineTransform` (class in `sympy.integrals.transforms`), 584  
`InverseFourierTransform` (class in `sympy.integrals.transforms`), 581  
`InverseHankelTransform` (class in `sympy.integrals.transforms`), 586  
`InverseLaplaceTransform` (class in `sympy.integrals.transforms`), 579  
`InverseMellinTransform` (class in `sympy.integrals.transforms`), 577  
`InverseSineTransform` (class in `sympy.integrals.transforms`), 583  
`inversion_vector()` (`sympy.combinatorics.permutations.Permutation` method), 271  
`inversions()` (`sympy.combinatorics.permutations.Permutation` method), 272  
`invert()` (in module `sympy.polys.polytools`), 2365  
`invert()` (`sympy.core.expr.Expr` method), 964  
`invert()` (`sympy.polys.domains.domain.Domain` method), 2514  
`invert()` (`sympy.polys.domains.ring.Ring` method), 2521  
`invert()` (`sympy.polys.polyclasses.DMF` method), 2568  
`invert()` (`sympy.polys.polyclasses.DMP` method), 2564  
`invert()` (`sympy.polys.polytools.Poly` method), 2399  
`invert_complex()` (in module `sympy.solvers.solve`), 868  
`invert_real()` (in module `sympy.solvers.solve`), 868  
`InvertiblePredicate` (class in `sympy.assumptions.predicates.matrices`), 204  
`iproduct()` (in module `sympy.utilities.iterables`), 2078  
`IQFT` (class in `sympy.physics.quantum.qft`), 1850  
`Irrational` (class in `sympy.polys.polyclasses`), 924  
`IrrationalPredicate` (class in `sympy.assumptions.predicates.sets`), 226  
`irregular()` (`sympy.matrices.matrices.MatrixBase` class method), 1302  
`is_abelian` (`sympy.combinatorics.perm_groups.PermutationGroup` property), 310  
`is_abelian_number()` (in module `sympy.combinatorics.group_numbers`), 355  
`is_above_fermi` (`sympy.functions.special.tensor_functions.SpecialFunctions` property), 551  
`is_above_fermi` (`sympy.physics.secondquant.KroneckerDelta` property), 1555  
`is_absolutely_convergent()` (`sympy.concrete.summations.Sum` method), 903  
`is_abundant()` (in module `sympy.ntheory.factor_`), 1508  
`is_algebraic_expr()` (`sympy.core.expr.Expr` method), 964  
`is_aliased` (`sympy.core.numbers.AlgebraicNumber` property), 991  
`is_alt_sym()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 310  
`is_alternating` (`sympy.combinatorics.perm_groups.PermutationGroup` property), 311  
`is_amicable()` (in module `sympy.ntheory.factor_`), 1509  
`is_anf()` (in module `sympy.logic.boolalg`), 1174  
`is_anti_symmetric()`



`(sympy.matrices.common.MatrixCommon.is_cyclotomic(sympy.polys.polyclasses.DMP method), 1336`  
`property), 2564`  
`is_below_fermi(sympy.functions.special.tensor_functions.kronecker_delta(sympy.polys.polytools.Poly property), 551`  
`property), 2399`  
`is_below_fermi(sympy.physics.secondquant.KroneckerDelta() (in module sympy.calculus.singularities), 234`  
`property), 1555`  
`is_biproper(sympy.physics.control.lti.Parallel.is_deficient() (in module sympy.ntheory.factor_), 1508`  
`property), 1905`  
`is_biproper(sympy.physics.control.lti.Series.is_diagonal() (sympy.matrices.common.MatrixCommon method), 1337`  
`property), 1901`  
`is_biproper(sympy.physics.control.lti.TransferFunction.is_diagonalizable() (sympy.matrices.matrices.MatrixEigen method), 1241`  
`property), 1896`  
`is_closed(sympy.sets.sets.Set property), 1188`  
`is_cnf() (in module sympy.logic.boolalg), 1174`  
`is_collinear() (sympy.geometry.point.Point method), 2205`  
`is_commutative(sympy.physics.quantum.state.Wavefunction (in module sympy.logic.boolalg), 1175`  
`property), 1838`  
`is_comparable(sympy.core.basic.Basic.is_echelon(sympy.matrices.matrices.MatrixReductions property), 935`  
`property), 1233`  
`is_compat() (sympy.polys.numberfields.modules.ModuleElement(sympy.combinatorics.perm_groups.Permutation method), 2731`  
`method), 312`  
`is_compat_col() (sympy.polys.numberfields.modules.ModuleElement(sympy.combinatorics.permutations.Permutation method), 2720`  
`property), 272`  
`is_concyclic() (sympy.geometry.point.Point method), 2205`  
`is_eulerian() (sympy.geometry.polygon.Polygon method), 2299`  
`is_conservative() (in module sympy.physics.vector.fieldfunctions), 1674`  
`is_euler_pseudoprime() (in module sympy.ntheory.primetest), 1513`  
`is_conservative() (in module sympy.vector), 1473`  
`is_extra_strong_lucas_prp() (in module sympy.ntheory.primetest), 1516`  
`is_consistent(sympy.physics.units.dimensions.DimensionSystem), 1586`  
`is_Field(sympy.polys.domains.domain.Domain attribute), 2514`  
`is_consistent(sympy.physics.units.unitsystem.UnitSystem), 1587`  
`is_full_module() (sympy.polys.agca.modules.SubModule method), 2458`  
`is_constant() (sympy.core.expr.Expr method), 965`  
`is_full_module() (sympy.polys.agca.modules.SubQuotient method), 2467`  
`is_convergent() (sympy.concrete.products.Product method), 908`  
`is_gaussian_prime() (in module sympy.ntheory.primetest), 1518`  
`is_convergent() (sympy.concrete.summations.Sum method), 903`  
`is_groebner() (in module sympy.polys.groebnertools), 2640`  
`is_convex() (in module sympy.calculus.util), 246`  
`is_ground(sympy.polys.polyclasses.ANP property), 2568`  
`is_convex() (sympy.geometry.polygon.Polygon method), 2280`  
`is_ground(sympy.polys.polyclasses.DMP property), 2564`  
`is_coplanar() (sympy.geometry.plane.Plane method), 2307`  
`is_ground(sympy.polys.polytools.Poly property), 2399`  
`is_cyclic(sympy.combinatorics.perm_groups.PermutationGroup property), 311`  
`is_hermitian(sympy.matrices.common.MatrixCommon property), 1338`  
`is_cyclic_number() (in module sympy.combinatorics.group_numbers), 356`  
`is_homogeneous(sympy.polys.polyclasses.DMP property), 2564`

`is_homogeneous` (`sympy.polys.polytools.Poly` property), 2400

`is_identity` (`sympy.combinatorics.permutations.Permutation` property), 273

`is_identity` (`sympy.core.function.Lambda` property), 1040

`is_increasing()` (in module `sympy.calculus.singularities`), 235

`is_indefinite` (`sympy.matrices.matrices.MatrixEigen` property), 1242

`is_inert` (`sympy.polys.numberfields.primes.PrimeIdeal` property), 2707

`is_injective()` (`sympy.polys.agca.homomorphisms.Homomorphism` method), 2471

`is_int()` (in module `sympy.polys.numberfields.utilities`), 2737

`is_irreducible` (`sympy.polys.polyclasses.DMP` property), 2564

`is_irreducible` (`sympy.polys.polytools.Poly` property), 2400

`is_isomorphism()` (`sympy.polys.agca.homomorphisms.Homomorphism` method), 2471

`is_isosceles()` (`sympy.geometry.polygon.Triangle` method), 2299

`is_iterable` (`sympy.sets.sets.ProductSet` property), 1200

`is_left_unbounded` (`sympy.sets.sets.Interval` property), 1196

`is_linear` (`sympy.polys.polyclasses.DMP` property), 2564

`is_linear` (`sympy.polys.polytools.Poly` property), 2400

`is_lower` (`sympy.matrices.common.MatrixCommon` property), 1338

`is_lower` (`sympy.polys.matrices.domainmatrix.DomainMatrix` property), 2679

`is_lower()` (`sympy.polys.matrices.ddm.DDM` method), 2690

`is_lower()` (`sympy.polys.matrices.sdm.SDM` method), 2695

`is_lower_hessenberg` (`sympy.matrices.common.MatrixCommon` property), 1339

`is_lucas_prp()` (in module `sympy.ntheory.primetest`), 1515

`is_maximal()` (`sympy.polys.agca.ideals.Ideal` method), 2463

`is_meromorphic()` (`sympy.core.expr.Expr` method), 966

`is_mersenne_prime()` (in module `sympy.ntheory.factor_`), 1507

`is_minimal()` (in module `sympy.polys.groebnertools`), 2640

`is_monic` (`sympy.polys.polyclasses.DMP` property), 2564

`is_monic` (`sympy.polys.polytools.Poly` property), 2401

`is_monomial` (`sympy.polys.polyclasses.DMP` property), 2564

`is_monomial` (`sympy.polys.polytools.Poly` property), 2401

`is_monotonic()` (in module `sympy.calculus.singularities`), 236

`is_multivariate` (`sympy.polys.polytools.Poly` property), 2401

`is_nilpotent` (`sympy.polys.domains.AlgebraicField` method), 2544

`is_negative()` (`sympy.polys.domains.ComplexField` method), 2546

`is_negative()` (`sympy.polys.domains.domain.Domain` method), 2515

`is_negative()` (`sympy.polys.domains.ExpressionDomain` method), 2550

`is_negative()` (`sympy.polys.domains.FractionField` method), 2549

`is_negative()` (`sympy.polys.domains.gaussiandomains.GaussianDomain` method), 2533

`is_negative()` (`sympy.polys.domains.PolynomialRing` method), 2548

`is_negative_definite` (`sympy.matrices.matrices.MatrixEigen` property), 1249

`is_negative_definite` (`sympy.matrices.matrices.MatrixEigen` property), 1256

`is_nilpotent` (`sympy.combinatorics.perm_groups.PermutationGroup` property), 312

`is_nilpotent()` (`sympy.matrices.matrices.MatrixBase` method), 1302

`is_nilpotent_number()` (in module `sympy.combinatorics.group_numbers`), 355

`is_nnf()` (in module `sympy.logic.boolalg`), 1175

`is_nonnegative()` (`sympy.polys.domains.AlgebraicField` method), 2544

`is_nonnegative()` (`sympy.polys.domains.ComplexField` method), 2546

`is_nonnegative()` (`sympy.polys.domains.domain.Domain` method), 2515

`is_nonnegative()` (`sympy.polys.domains.ExpressionDomain` method), 2550

`is_nonnegative()` (`sympy.polys.domains.FractionField` method), 2549

`is_nonnegative()` (`sympy.polys.domains.gaussiandomains.GaussianDomain` method), 2533

`is_nonnegative()` (`sympy.polys.domains.PolynomialRing` method), 2548

`method`), 2548  
`is_nonpositive()` (`sympy.polys.domains.AlgebraicField` `property`), 1543  
`method`), 2544  
`is_nonpositive()` (`sympy.polys.domains.ComplexField` `property`), 1543  
`method`), 2546  
`is_nonpositive()` (`sympy.polys.domains.domain.Domain` `property`), 1548  
`method`), 2515  
`is_nonpositive()` (`sympy.polys.domains.ExpressionDomain` `property`), 1543  
`method`), 2550  
`is_nonpositive()` (`sympy.polys.domains.FractionField` `property`), 1543  
`method`), 2549  
`is_nonpositive()` (`sympy.polys.domains.gaussiandomains.GaussianDomain` `property`), 1543  
`method`), 2533  
`is_nonpositive()` (`sympy.polys.domains.PolynomialRing` `property`), 1548  
`method`), 2548  
`is_nonzero` (`sympy.geometry.point.Point` `property`), 2206  
`is_normal()` (`sympy.combinatorics.perm_groups.PermutationGroup` `property`), 313  
`method`), 313  
`is_normalized` (`sympy.physics.quantum.state.WaveFunction` `property`), 1838  
`is_nthpow_residue()` (`sympy.ntheory.residue_ntheory` `property`), 1521  
`is_number` (`sympy.concrete.expr_with_limits.ExprWithLimits` `property`), 608  
`is_number` (`sympy.core.expr.Expr` `property`), 967  
`is_number` (`sympy.functions.special.hyper.meijerg` `property`), 525  
`is_odd` (`sympy.combinatorics.permutations.Permutation` `property`), 274  
`is_one` (`sympy.polys.polyclasses.ANP` `property`), 2569  
`is_one` (`sympy.polys.polyclasses.DMF` `property`), 2568  
`is_one` (`sympy.polys.polyclasses.DMP` `property`), 2564  
`is_one` (`sympy.polys.polytools.Poly` `property`), 2402  
`is_one()` (`sympy.polys.domains.domain.Domain` `method`), 2515  
`is_only_above_fermi` (`sympy.functions.special.tensor_functions.KroneckerDelta` `property`), 552  
`is_only_above_fermi` (`sympy.physics.secondquant.KroneckerDelta` `property`), 1555  
`is_only_below_fermi` (`sympy.functions.special.tensor_functions.KroneckerDelta` `property`), 552  
`is_only_below_fermi` (`sympy.physics.secondquant.KroneckerDelta` `property`), 1556  
`is_only_q_annihilator` (`sympy.physics.secondquant.AnnihilateFermion` `property`), 1543  
`is_only_q_annihilator` (`sympy.physics.secondquant.CreateFermion` `property`), 1543  
`is_only_q_creator` (`sympy.physics.secondquant.AnnihilateFermion` `property`), 1543  
`is_only_q_creator` (`sympy.physics.secondquant.CreateFermion` `property`), 1543  
`is_open` (`sympy.sets.sets.Set` `property`), 1189  
`is_palindromic()` (`sympy.ntheory.digits` `property`), 1528  
`is_perfect()` (`sympy.combinatorics.perm_groups.PermutationGroup` `property`), 313  
`is_perfect()` (`sympy.ntheory.factor_` `property`), 1507  
`is_perpendicular()` (`sympy.geometry.line.LinearEntity` `method`), 2221  
`is_perpendicular()` (`sympy.geometry.plane.Plane` `method`), 2307  
`is_PID` (`sympy.polys.domains.domain.Domain` `attribute`), 2515  
`is_polycyclic` (`sympy.combinatorics.perm_groups.PermutationGroup` `property`), 314  
`is_polynomial()` (`sympy.core.expr.Expr` `method`), 968  
`is_positive()` (`sympy.polys.domains.AlgebraicField` `method`), 2544  
`is_positive()` (`sympy.polys.domains.ComplexField` `method`), 2546  
`is_positive()` (`sympy.polys.domains.domain.Domain` `method`), 2515  
`is_positive()` (`sympy.polys.domains.ExpressionDomain` `method`), 2550  
`is_positive()` (`sympy.polys.domains.FractionField` `method`), 2549  
`is_positive()` (`sympy.polys.domains.gaussiandomains.GaussianDomain` `method`), 2533  
`is_positive()` (`sympy.polys.domains.PolynomialRing` `method`), 2548  
`is_positive_definite` (`sympy.matrices.matrices.MatrixEigen` `property`), 1263

[is\\_positive\\_seminegative](#) ([sympy.matrices.matrices.MatrixEigen](#) property), 1270  
[is\\_prefixed](#) ([sympy.physics.units.quantities.Quantity](#) property), 1588  
[is\\_primary](#) ([sympy.polys.agca.ideals.Ideal](#) method), 2463  
[is\\_prime](#) ([sympy.polys.agca.ideals.Ideal](#) method), 2463  
[is\\_primitive](#) ([sympy.polys.polyclasses.DMP](#) property), 2564  
[is\\_primitive](#) ([sympy.polys.polytools.Poly](#) property), 2402  
[is\\_primitive](#) ([sympy.combinatorics.perm\\_groups.PermGroup](#) method), 314  
[is\\_primitive\\_element](#) ([sympy.core.numbers.AlgebraicNumber](#) property), 991  
[is\\_primitive\\_root](#) (in module [sympy.ntheory.residue\\_ntheory](#)), 1518  
[is\\_principal](#) ([sympy.polys.agca.ideals.Ideal](#) method), 2463  
[is\\_proper](#) ([sympy.physics.control.lti.Parallel](#) property), 1905  
[is\\_proper](#) ([sympy.physics.control.lti.Series](#) property), 1902  
[is\\_proper](#) ([sympy.physics.control.lti.TransferFunction](#) property), 1897  
[is\\_proper\\_subset](#) ([sympy.sets.sets.Set](#) method), 1189  
[is\\_proper\\_superset](#) ([sympy.sets.sets.Set](#) method), 1189  
[is\\_pure](#) ([sympy.algebras.Quaternion](#) method), 891  
[is\\_q\\_annihilator](#) ([sympy.physics.secondquant.AnnihilationOperator](#) property), 1544  
[is\\_q\\_annihilator](#) ([sympy.physics.secondquant.CreateOperator](#) property), 1549  
[is\\_q\\_creator](#) ([sympy.physics.secondquant.AnnihilationOperator](#) property), 1544  
[is\\_q\\_creator](#) ([sympy.physics.secondquant.CreateOperator](#) property), 1549  
[is\\_quad\\_residue](#) (in module [sympy.ntheory.residue\\_ntheory](#)), 1521  
[is\\_quadratic](#) ([sympy.polys.polyclasses.DMP](#) property), 2564  
[is\\_quadratic](#) ([sympy.polys.polytools.Poly](#) property), 2402  
[is\\_radical](#) ([sympy.polys.agca.ideals.Ideal](#) method), 2463  
[is\\_rat](#) (in module [sympy.polys.numberfields.utilities](#)), 2737  
[is\\_rational](#) ([sympy.polys.numberfields.modules.PowerBasis](#) property), 2732  
[is\\_rational\\_function](#) ([sympy.core.expr.Expr](#) method), 969  
[is\\_reduced](#) (in module [sympy.polys.groebnertools](#)), 2640  
[is\\_right](#) ([sympy.geometry.polygon.Triangle](#) method), 2299  
[is\\_right\\_unbounded](#) ([sympy.sets.sets.Interval](#) property), 1196  
[is\\_singular](#) ([sympy.polys.domains.domain.Domain](#) attribute), 2515  
[is\\_scalar\\_multiple](#) ([sympy.geometry.point.Point](#) method), 2206  
[is\\_scalene](#) ([sympy.geometry.polygon.Triangle](#) method), 2300  
[is\\_sequence](#) (in module [sympy.utilities.iterables](#)), 2079  
[is\\_similar](#) ([sympy.geometry.entity.GeometryEntity](#) method), 2195  
[is\\_similar](#) ([sympy.geometry.line.LinearEntity](#) method), 2222  
[is\\_similar](#) ([sympy.geometry.polygon.Triangle](#) method), 2300  
[is\\_simple](#) ([sympy.functions.special.delta\\_functions.DiracDelta](#) method), 453  
[is\\_Singleton](#) ([sympy.combinatorics.permutations.Permutation](#) property), 273  
[is\\_singular](#) ([sympy.core.function.Function](#) class method), 1051  
[is\\_singular](#) ([sympy.holonomic.holonomic.DifferentialOperator](#) method), 2315  
[is\\_solenoidal](#) (in module [sympy.physics.vector.fieldfunctions](#)), 1674  
[is\\_stable](#) ([sympy.physics.vector.Vector](#) method), 1473  
[is\\_solvable](#) ([sympy.combinatorics.perm\\_groups.PermutationGroup](#) property), 315  
[is\\_sqf](#) ([sympy.polys.polyclasses.DMP](#) property), 2564  
[is\\_sqf](#) ([sympy.polys.polytools.Poly](#) property), 2402  
[is\\_square](#) ([sympy.matrices.common.MatrixCommon](#) property), 1340  
[is\\_square](#) (in module [sympy.ntheory.primetest](#)), 1514  
[is\\_stable](#) ([sympy.physics.control.lti.TransferFunction](#) method), 1897  
[is\\_strictly\\_decreasing](#) (in module



`sympy.calculus.singularities`), 236  
`is_strictly_increasing()` (in module `sympy.calculus.singularities`), 237  
`is_strictly_proper` (`sympy.physics.control.lti.Parallel` property), 1905  
`is_strictly_proper` (`sympy.physics.control.lti.Series` property), 1902  
`is_strictly_proper` (`sympy.physics.control.lti.TransferFunction` property), 1897  
`is_strong_lucas_prp()` (in module `sympy.ntheory.primetest`), 1515  
`is_strongly_diagonally_dominant` (`sympy.matrices.common.MatrixCommon` property), 1340  
`is_subdiagram()` (`sympy.categories.Diagram` method), 2751  
`is_subgroup()` (`sympy.combinatorics.perm_groups.PermGroup` method), 315  
`is_submodule()` (`sympy.polys.agca.modules.FreeModule` method), 2455  
`is_submodule()` (`sympy.polys.agca.modules.Module` method), 2453  
`is_submodule()` (`sympy.polys.agca.modules.QuotientModule` method), 2465  
`is_submodule()` (`sympy.polys.agca.modules.SubModule` method), 2458  
`is_subset()` (`sympy.sets.sets.Set` method), 1189  
`is_superset()` (`sympy.sets.sets.Set` method), 1190  
`is_surjective()` (`sympy.polys.agca.homomorphisms.ModuleHomomorphism` method), 2472  
`is_symbolic()` (`sympy.matrices.common.MatrixCommon` method), 1341  
`is_symmetric` (`sympy.combinatorics.perm_groups.PermGroup` property), 316  
`is_symmetric()` (`sympy.matrices.common.MatrixCommon` method), 1341  
`is_tangent()` (`sympy.geometry.ellipse.Ellipse` method), 2261  
`is_transitive()` (`sympy.combinatorics.perm_groups.PermGroup` method), 316  
`is_trivial` (`sympy.combinatorics.perm_groups.PermGroup` property), 317  
`is_unit()` (`sympy.polys.domains.field.Field` method), 2520  
`is_unit()` (`sympy.polys.domains.PolynomialRing` method), 2548  
`is_univariate` (`sympy.polys.polytools.Poly` property), 2403  
`is_upper` (`sympy.matrices.common.MatrixCommon` property), 1342  
`is_upper` (`sympy.polys.matrices.domainmatrix.DomainMatrix` property), 2679  
`is_upper()` (`sympy.polys.matrices.ddm.DDM` method), 2690  
`is_upper()` (`sympy.polys.matrices.sdm.SDM` method), 2695  
`is_upper_hessenberg` (`sympy.matrices.common.MatrixCommon` property), 1343  
`is_weakly_diagonally_dominant` (`sympy.matrices.common.MatrixCommon` property), 1343  
`is_whole_ring()` (`sympy.polys.agca.ideals.Ideal` method), 2463  
`is_zero` (`sympy.geometry.point.Point` property), 2206  
`is_zero` (`sympy.polys.polyclasses.ANP` property), 2569  
`is_zero` (`sympy.polys.polyclasses.DMF` property), 2568  
`is_zero` (`sympy.polys.polyclasses.DMP` property), 2564  
`is_zero` (`sympy.polys.polytools.Poly` property), 2403  
`is_zero` (`sympy.polys.agca.homomorphisms.ModuleHomomorphism` method), 2472  
`is_zero` (`sympy.polys.agca.ideals.Ideal` method), 2463  
`is_zero()` (`sympy.polys.agca.modules.FreeModule` method), 2455  
`is_zero()` (`sympy.polys.agca.modules.Module` method), 2453  
`is_zero` (`sympy.polys.agca.modules.QuotientModule` method), 2466  
`is_zero` (`sympy.polys.agca.modules.SubModule` method), 2458  
`is_zero_dimensional` (`sympy.polys.domains.domain.Domain` method), 2515  
`is_zero_dimensional` (`sympy.polys.polytools.GroebnerBasis` property), 2424  
`is_zero_dimensional()` (in module `sympy.polys.polytools`), 2378  
`is_zero_matrix` (`sympy.matrices.common.MatrixCommon` property), 1344  
`is_zero_matrix()` (`sympy.polys.matrices.ddm.DDM` method), 2690  
`is_zero_matrix()` (`sympy.polys.matrices.sdm.SDM` method), 2695  
`is_zero_quaternion()` (`sympy.algebras.Quaternion` method), 891  
`is_disjoint()` (`sympy.sets.sets.Set` method),

1190  
isign (class in `sympy.codegen.fnodes`), 1154  
isolate() (in module `sympy.polys.numberfields.utilities`), 2741  
IsomorphismFailed (class in `sympy.polys.polyerrors`), 2643  
isprime() (in module `sympy.ntheory.primetest`), 1517  
issubset() (`sympy.sets.sets.Set` method), 1190  
issuperset() (`sympy.sets.sets.Set` method), 1190  
IsTruePredicate (class in `sympy.assumptions.predicates.common`), 201  
ITE (class in `sympy.logic.boolalg`), 1171  
items() (`sympy.core.containers.Dict` method), 1070  
iter\_q\_annihilators() (`sympy.physics.secondquant.NO` method), 1559  
iter\_q\_creators() (`sympy.physics.secondquant.NO` method), 1560  
iterable() (in module `sympy.utilities.iterables`), 2080  
iterate\_binary() (`sympy.combinatorics.subsets.Subset` method), 340  
iterate\_graycode() (`sympy.combinatorics.subsets.Subset` method), 341  
itercoeffs() (`sympy.polys.rings.PolyElement` method), 2557  
itermonomials() (in module `sympy.polys.monomials`), 2429  
itermonoms() (`sympy.polys.rings.PolyElement` method), 2557  
iterterms() (`sympy.polys.rings.PolyElement` method), 2558

**J**  
J20p (class in `sympy.physics.quantum.spin`), 1820  
jacobi (class in `sympy.functions.special.polynomials`), 533  
jacobi\_normalized() (in module `sympy.functions.special.polynomials`), 534  
jacobi\_poly() (in module `sympy.polys.orthopolys`), 2440  
jacobi\_symbol() (in module `sympy.ntheory.residue_ntheory`), 1522  
jacobian() (`sympy.diffgeom.CoordSystem` method), 2803  
jacobian() (`sympy.matrices.matrices.MatrixCalculus` method), 1279  
jacobian\_determinant() (`sympy.diffgeom.CoordSystem` method), 2804  
jacobian\_matrix() (`sympy.diffgeom.CoordSystem` method), 2804  
JavascriptCodePrinter (class in `sympy.printing.jscode`), 2155  
jn (class in `sympy.functions.special.bessel`), 501  
jn\_zeros() (in module `sympy.functions.special.bessel`), 503  
join() (`sympy.physics.continuum_mechanics.beam.Beam` method), 1956  
Joint (class in `sympy.physics.mechanics.joint`), 1770  
JointRV() (in module `sympy.stats`), 2935  
JointsMethod (class in `sympy.physics.mechanics.jointsmethod`), 1780  
jones\_2\_stokes() (in module `sympy.physics.optics.polarization`), 1873  
jones\_vector() (in module `sympy.physics.optics.polarization`), 1873  
jordan\_block() (`sympy.matrices.common.MatrixCommon` class method), 1345  
jordan\_cell() (in module `sympy.matrices.dense`), 1320  
jordan\_form() (`sympy.matrices.matrices.MatrixEigen` method), 1277  
josephus() (`sympy.combinatorics.permutations.Permutation` class method), 274  
jscode() (in module `sympy.printing.jscode`), 2155  
julia\_code() (in module `sympy.printing.julia`), 2158  
JuliaCodeGen (class in `sympy.utilities.codegen`), 2050  
JuliaCodePrinter (class in `sympy.printing.julia`), 2158  
JxBra (class in `sympy.physics.quantum.spin`), 1820  
JxBraCoupled (class in `sympy.physics.quantum.spin`), 1820  
JxKet (class in `sympy.physics.quantum.spin`), 1820  
JxKetCoupled (class in

- [sympy.physics.quantum.spin](#)), 1820  
[JyBra](#) (class in [sympy.physics.quantum.spin](#)), 1820  
[JyBraCoupled](#) (class in [sympy.physics.quantum.spin](#)), 1820  
[JyKet](#) (class in [sympy.physics.quantum.spin](#)), 1821  
[JyKetCoupled](#) (class in [sympy.physics.quantum.spin](#)), 1821  
[JzBra](#) (class in [sympy.physics.quantum.spin](#)), 1821  
[JzBraCoupled](#) (class in [sympy.physics.quantum.spin](#)), 1821  
[JzKet](#) (class in [sympy.physics.quantum.spin](#)), 1821  
[JzKetCoupled](#) (class in [sympy.physics.quantum.spin](#)), 1823  
[JzOp](#) (class in [sympy.physics.quantum.spin](#)), 1825
- K**
- [kahane\\_simplify\(\)](#) (in module [sympy.physics.hep.gamma\\_matrices](#)), 1590  
[kane\\_equations\(\)](#) ([sympy.physics.mechanics.kane.KanesMethod](#) method), 1765  
[KanesMethod](#) (class in [sympy.physics.mechanics.kane](#)), 1764  
[kbins\(\)](#) (in module [sympy.utilities.iterables](#)), 2081  
[kdes](#) ([sympy.physics.mechanics.joint.Joint](#) property), 1772  
[kdes](#) ([sympy.physics.mechanics.jointsmethod.JointsMethod](#) property), 1782  
[kernel\(\)](#) ([sympy.polys.agca.homomorphisms.ModuleHomomorphism](#) method), 2473  
[kernel\(\)](#) ([sympy.polys.numberfields.modules.ModuleHomomorphism](#) method), 2733  
[Ket](#) (class in [sympy.physics.quantum.state](#)), 1833  
[ket](#) ([sympy.physics.quantum.operator.OuterProduct](#) property), 1812  
[ket](#) ([sympy.physics.secondquant.InnerProduct](#) property), 1553  
[KetBase](#) (class in [sympy.physics.quantum.state](#)), 1834  
[key2bounds\(\)](#) ([sympy.matrices.matrices.MatrixBase](#) method), 1302  
[key2ij\(\)](#) ([sympy.matrices.matrices.MatrixBase](#) method), 1303  
[keys\(\)](#) ([sympy.core.containers.Dict](#) method), 1071  
[kid\\_rsa\\_private\\_key\(\)](#) (in module [sympy.crypto.crypto](#)), 2787  
[kid\\_rsa\\_public\\_key\(\)](#) (in module [sympy.crypto.crypto](#)), 2787  
[killable\\_index](#) ([sympy.functions.special.tensor\\_functions](#) property), 552  
[killable\\_index](#) ([sympy.physics.secondquant.KroneckerDelta](#) property), 1556  
[kin\\_explicit\\_rhs](#) ([sympy.physics.mechanics.system.SymbolicEquation](#) property), 1787  
[kind](#) (class in [sympy.codegen.fnodes](#)), 1154  
[Kind](#) (class in [sympy.core.kind](#)), 1073  
[kind](#) ([sympy.core.containers.Tuple](#) property), 1069  
[kind](#) ([sympy.sets.sets.Set](#) property), 1190  
[kindiffdict\(\)](#) ([sympy.physics.mechanics.kane.KanesMethod](#) method), 1766  
[kinematic\\_equations\(\)](#) (in module [sympy.physics.vector.functions](#)), 1660  
[kinetic\\_energy\(\)](#) (in module [sympy.physics.mechanics.functions](#)), 1753  
[kinetic\\_energy\(\)](#) ([sympy.physics.mechanics.body.Body](#) method), 1762  
[kinetic\\_energy\(\)](#) ([sympy.physics.mechanics.particle.Particle](#) method), 1744  
[kinetic\\_energy\(\)](#) ([sympy.physics.mechanics.rigidbody.RigidBody](#) method), 1748  
[known\\_fcns\\_src1](#) (in module [sympy.printing.julia](#)), 2157  
[known\\_fcns\\_src1](#) (in module [sympy.printing.octave](#)), 2160  
[known\\_fcns\\_src2](#) (in module [sympy.printing.julia](#)), 2158  
[known\\_fcns\\_src2](#) (in module [sympy.printing.octave](#)), 2161  
[known\\_functions](#) (in module [sympy.printing.jscode](#)), 2155  
[known\\_functions](#) (in module [sympy.printing.mathematica](#)), 2152  
[known\\_functions](#) (in module [sympy.printing.rcode](#)), 2144  
[known\\_functions](#) (in module [sympy.printing.rust](#)), 2164  
[known\\_functions\\_C89](#) (in module [sympy.printing.c](#)), 2140  
[known\\_functions\\_C99](#) (in module [sympy.printing.c](#)), 2140  
[KroneckerDelta](#) (class in [sympy.functions.special.tensor\\_functions](#)), 549  
[KroneckerDelta](#) (class in [sympy.physics.secondquant](#)), 1553

`kroneckersimp()` (in module `sympy.simplify.simplify`), 666

`ksubsets()` (`sympy.combinatorics.subsets` method), 347

`Kumaraswamy()` (in module `sympy.stats`), 2908

`kurtosis()` (in module `sympy.stats`), 2974

`kwargs()` (`sympy.codegen.ast.Token` method), 1137

**L**

`l1_norm()` (`sympy.polys.polyclasses.DMP` method), 2564

`l1_norm()` (`sympy.polys.polytools.Poly` method), 2403

`L2` (class in `sympy.physics.quantum.hilbert`), 1804

`l2_norm_squared()` (`sympy.polys.polyclasses.DMP` method), 2564

`Label` (class in `sympy.codegen.cnodes`), 1148

`label` (`sympy.physics.quantum.state.TimeDepState` property), 1836

`label` (`sympy.tensor.indexed.Idx` property), 1401

`label` (`sympy.tensor.indexed.IndexedBase` property), 1405

`labeller()` (in module `sympy.physics.quantum.circuitplot`), 1841

`LagrangesMethod` (class in `sympy.physics.mechanics.lagrange`), 1767

`Lagrangian()` (in module `sympy.physics.mechanics.functions`), 1755

`laguerre` (class in `sympy.functions.special.polynomials`), 542

`laguerre_poly()` (in module `sympy.polys.orthopolys`), 2441

`Lambda` (class in `sympy.core.function`), 1039

`lambda_notation()` (in module `sympy.parsing.sympy_parser`), 2128

`LambdaPrinter` (class in `sympy.printing.lambdarepr`), 2170

`lambdarepr()` (in module `sympy.printing.lambdarepr`), 2170

`lambdastr()` (in module `sympy.utilities.lambdify`), 2100

`lambdify()` (in module `sympy.utilities.lambdify`), 2100

`LambertW` (class in `sympy.functions.elementary.exponential`), 412

`Laplace()` (in module `sympy.stats`), 2909

`laplace_transform()` (in module `sympy.integrals.transforms`), 577

`LaplaceTransform` (class in `sympy.integrals.transforms`), 579

`latex()` (in module `sympy.printing.latex`), 2170

`LaTeXParsingError` (class in `sympy.parsing.latex`), 2130

`LatexPrinter` (class in `sympy.printing.latex`), 2170

`lbound()` (in module `sympy.codegen.fnodes`), 1154

`LC()` (in module `sympy.polys.polytools`), 2361

`LC()` (`sympy.polys.polyclasses.ANP` method), 2568

`LC()` (`sympy.polys.polyclasses.DMP` method), 2561

`LC()` (`sympy.polys.polytools.Poly` method), 2379

`lcm()` (in module `sympy.calculus.util`), 247

`lcm()` (in module `sympy.polys.polytools`), 2368

`lcm()` (`sympy.core.numbers.Number` method), 982

`lcm()` (`sympy.polys.domains.ComplexField` method), 2546

`lcm()` (`sympy.polys.domains.domain.Domain` method), 2515

`lcm()` (`sympy.polys.domains.field.Field` method), 2520

`lcm()` (`sympy.polys.domains.gaussiandomains.GaussianInteger` method), 2536

`lcm()` (`sympy.polys.domains.GMPYIntegerRing` method), 2529

`lcm()` (`sympy.polys.domains.IntegerRing` method), 2527

`lcm()` (`sympy.polys.domains.PolynomialRing` method), 2548

`lcm()` (`sympy.polys.domains.RealField` method), 2546

`lcm()` (`sympy.polys.monomials.Monomial` method), 2429

`lcm()` (`sympy.polys.polyclasses.DMP` method), 2565

`lcm()` (`sympy.polys.polytools.Poly` method), 2404

`lcm_list()` (in module `sympy.polys.polytools`), 2368

`ldescent()` (in module `sympy.solvers.diophantine.diophantine`), 741

`LDLdecomposition()` (`sympy.matrices.dense.DenseMatrix` method), 2546



`method`), 1361  
`LDLdecomposition()` (`sympy.matrices.matrices.MatrixBase` `method`), 1281  
`LDLsolve()` (`sympy.matrices.matrices.MatrixBase` `method`), 1282  
`LDUdecomposition()` (`sympy.matrices.expressions.blockmatrix.BlockMatrix` `method`), 1381  
`Le` (in module `sympy.core.relational`), 1022  
`leading_expv()` (`sympy.polys.rings.PolyElement` `method`), 2558  
`leading_monom()` (`sympy.polys.rings.PolyElement` `method`), 2558  
`leading_term()` (`sympy.polys.rings.PolyElement` `method`), 2558  
`leadterm()` (`sympy.core.expr.Expr` `method`), 970  
`least_rotation()` (in module `sympy.utilities.iterables`), 2082  
`left()` (`sympy.printing.pretty.stringpict.stringPict` `method`), 2187  
`left_eigenvects()` (`sympy.matrices.matrices.MatrixEigen` `method`), 1278  
`left_open` (`sympy.sets.sets.Interval` `property`), 1196  
`leftslash()` (`sympy.printing.pretty.stringpict.stringPict` `method`), 2187  
`legendre` (class in `sympy.functions.special.polynomials`), 540  
`legendre_poly()` (in module `sympy.polys.orthopolys`), 2440  
`legendre_symbol()` (in module `sympy.ntheory.residue_ntheory`), 1521  
`length` (`sympy.geometry.curve.Curve` `property`), 2250  
`length` (`sympy.geometry.line.LinearEntity` `property`), 2222  
`length` (`sympy.geometry.line.Segment` `property`), 2235  
`length` (`sympy.geometry.point.Point` `property`), 2206  
`length` (`sympy.geometry.polygon.RegularPolygon` `property`), 2290  
`length` (`sympy.physics.continuum_mechanics.beam.Beam` `property`), 1958  
`length` (`sympy.series.sequences.SeqBase` `property`), 634  
`length()` (`sympy.combinatorics.permutations.Permutation` `method`), 275  
`length()` (`sympy.polys.polytools.Poly` `method`), 2404  
`lens_formula()` (in module `sympy.physics.optics.utils`), 1883  
`lens_makers_formula()` (in module `sympy.physics.optics.utils`), 1884  
`lerchphi` (class in `sympy.functions.special.zeta_functions`), 177  
`LessThan` (class in `sympy.core.relational`), 1028  
`LeviCivita` (class in `sympy.functions.special.tensor_functions`), 548  
`Levy()` (in module `sympy.stats`), 2910  
`lexOrder` (class in `sympy.polys.orderings`), 2431  
`lfsr_autocorrelation()` (in module `sympy.crypto.crypto`), 2790  
`lfsr_connection_polynomial()` (in module `sympy.crypto.crypto`), 2791  
`lfsr_sequence()` (in module `sympy.crypto.crypto`), 2789  
`lhs` (`sympy.core.relational.Relational` `property`), 1020  
`Li` (class in `sympy.functions.special.error_functions`), 489  
`li` (class in `sympy.functions.special.error_functions`), 487  
`lie_algebra()` (`sympy.liealgebras.type_a.TypeA` `method`), 2329  
`lie_algebra()` (`sympy.liealgebras.type_b.TypeB` `method`), 2330  
`lie_algebra()` (`sympy.liealgebras.type_c.TypeC` `method`), 2332  
`lie_algebra()` (`sympy.liealgebras.type_d.TypeD` `method`), 2333  
`lie_heuristic_abaco1_product()` (in module `sympy.solvers.ode.lie_group`), 798  
`lie_heuristic_abaco1_simple()` (in module `sympy.solvers.ode.lie_group`), 798  
`lie_heuristic_abaco2_similar()` (in module `sympy.solvers.ode.lie_group`), 800  
`lie_heuristic_abaco2_unique_general()` (in module `sympy.solvers.ode.lie_group`), 802  
`lie_heuristic_abaco2_unique_unknown()` (in module `sympy.solvers.ode.lie_group`), 801  
`lie_heuristic_bivariate()` (in module `sympy.solvers.ode.lie_group`), 799  
`lie_heuristic_chi()` (in module `sympy.solvers.ode.lie_group`), 799  
`lie_heuristic_function_sum()` (in module `sympy.solvers.ode.lie_group`), 800

`lie_heuristic_linear()` (in module `sympy.solvers.ode.lie_group`), 802

`LieDerivative` (class in `sympy.diffgeom`), 2812

`LieGroup` (class in `sympy.solvers.ode.single`), 793

`lift()` (`sympy.polys.polyclasses.DMP` method), 2565

`lift()` (`sympy.polys.polytools.Poly` method), 2404

`Limit` (class in `sympy.series.limits`), 621

`limit()` (in module `sympy.series.limits`), 620

`limit()` (`sympy.core.expr.Expr` method), 970

`limit()` (`sympy.matrices.matrices.MatrixCalculus` method), 1280

`limit_denominator()` (`sympy.core.numbers.Rational` method), 987

`limit_seq()` (in module `sympy.series.limitseq`), 660

`limitinf()` (in module `sympy.series.gruntz`), 624

`limiting_distribution` (`sympy.stats.DiscreteMarkovChain` property), 2950

`limits` (`sympy.concrete.expr_with_limits.ExprWithLimits` property), 608

`limits` (`sympy.geometry.curve.Curve` property), 2250

`limits` (`sympy.physics.quantum.state.Wavefunction` property), 1838

`Line` (class in `sympy.geometry.line`), 2227

`Line2D` (class in `sympy.geometry.line`), 2238

`Line2DBaseSeries` (class in `sympy.plotting.plot`), 2865

`Line3D` (class in `sympy.geometry.line`), 2243

`Line3DBaseSeries` (class in `sympy.plotting.plot`), 2867

`line_integrate()` (in module `sympy.integrals.integrals`), 601

`Linear` (class in `sympy.solvers.diophantine.diophantine`), 746

`linear_eq_to_matrix()` (in module `sympy.solvers.solve`), 870

`linear_momentum()` (in module `sympy.physics.mechanics.functions`), 1751

`linear_momentum()` (`sympy.physics.mechanics.particle.Particle` method), 1745

`linear_momentum()` (`sympy.physics.mechanics.rigidbody.RigidBody` method), 1748

`linear_ode_to_matrix()` (in module `sympy.solvers.ode.systems`), 807

`linear_polarizer()` (in module `sympy.physics.optics.polarization`), 1874

`LinearCoefficients` (class in `sympy.solvers.ode.single`), 791

`LinearEntity` (class in `sympy.geometry.line`), 2216

`LinearEntity2D` (class in `sympy.geometry.line`), 2236

`LinearEntity3D` (class in `sympy.geometry.line`), 2242

`linearize()` (`sympy.physics.mechanics.kane.KanesMethod` method), 1766

`linearize()` (`sympy.physics.mechanics.lagrange.Lagrange` method), 1769

`linearize()` (`sympy.physics.mechanics.linearize.Linearizer` method), 1788

`Linearizer` (class in `sympy.physics.mechanics.linearize`), 1787

`LineOver1DRangeSeries` (class in `sympy.plotting.plot`), 2866

`linodesolve()` (in module `sympy.solvers.ode.systems`), 814

`linodesolve_type()` (in module `sympy.solvers.ode.systems`), 810

`linsolve()` (in module `sympy.solvers.solve`), 872

`linsolve` (class in `sympy.solvers.ode.single`), 780

`list()` (`sympy.combinatorics.permutations.Cycle` method), 285

`list()` (`sympy.combinatorics.permutations.Permutation` method), 275

`list2numpy()` (in module `sympy.matrices.dense`), 1323

`list_can_dims` (`sympy.physics.units.dimensions.Dimension` property), 1586

`list_visitor()` (in module `sympy.utilities.enumerative`), 2062

`listcoeffs()` (`sympy.polys.rings.PolyElement` method), 2558

`listmonoms()` (`sympy.polys.rings.PolyElement` method), 2558

`listterms()` (`sympy.polys.rings.PolyElement` method), 2558

`literal_dp` (class in `sympy.codegen.fnodes`), 1155

`literal_sp` (class in `sympy.codegen.fnodes`), 1155

`LM` (in module `sympy.polys.polytools`), 2361

`LM()` (`sympy.polys.polytools.Poly` method), 2380

[load \(sympy.physics.continuum\\_mechanics.beam.Beam property\), 1958](#)  
[load\\_vector \(sympy.physics.continuum\\_mechanics.beam.Beam3D property\), 1984](#)  
[loads \(sympy.physics.continuum\\_mechanics.truss.Truss property\), 2025](#)  
[loads \(sympy.physics.mechanics.jointsmethod.JointsMethod property\), 1782](#)  
[loads \(sympy.physics.mechanics.system.SymbolicSystem property\), 1787](#)  
[locate\\_new\(\) \(sympy.vector.coordsysrect.CoordSys3D method\), 1449](#)  
[locatenew\(\) \(sympy.physics.vector.point.Point method\), 1654](#)  
[log \(class in sympy.functions.elementary.exponential\), 413](#)  
[log\(\) \(sympy.matrices.matrices.MatrixBase method\), 1303](#)  
[log\(\) \(sympy.polys.domains.domain.Domain method\), 2516](#)  
[log\(\) \(sympy.polys.domains.IntegerRing method\), 2527](#)  
[log10 \(class in sympy.codegen.cfunctions\), 1146](#)  
[log1p \(class in sympy.codegen.cfunctions\), 1147](#)  
[log2 \(class in sympy.codegen.cfunctions\), 1147](#)  
[Logarithmic\(\) \(in module sympy.stats\), 2883](#)  
[logcombine\(\) \(in module sympy.simplify.simplify\), 669](#)  
[loggamma \(class in sympy.functions.special.gamma\\_functions\), 460](#)  
[Logistic\(\) \(in module sympy.stats\), 2910](#)  
[LogLogistic\(\) \(in module sympy.stats\), 2911](#)  
[LogNormal\(\) \(in module sympy.stats\), 2912](#)  
[Lomax\(\) \(in module sympy.stats\), 2913](#)  
[Lopen\(\) \(sympy.sets.sets.Interval class method\), 1195](#)  
[lower \(sympy.physics.secondquant.AntiSymmetricTensor property\), 1545](#)  
[lower \(sympy.tensor.indexed.Idx property\), 1401](#)  
[lower\\_central\\_series\(\) \(sympy.combinatorics.perm\\_groups.PermutationGroup method\), 317](#)  
[lower\\_triangular\(\) \(sympy.matrices.common.MatrixCommon method\), 1347](#)  
[lower\\_triangular\\_solve\(\) \(sympy.matrices.dense.DenseMatrix method\), 1363](#)  
[lower\\_triangular\\_solve\(\) \(sympy.matrices.matrices.MatrixBase method\), 1304](#)  
[lower\\_gamma \(class in sympy.functions.special.gamma\\_functions\), 460](#)  
[LowerTriangularPredicate \(class in sympy.assumptions.predicates.matrices\), 208](#)  
[LsSystem\(\) \(sympy.core.expr.Expr method\), 970](#)  
[LT\(\) \(in module sympy.core.relational\), 1022](#)  
[LT\(\) \(in module sympy.polys.polytools\), 2362](#)  
[LT\(\) \(sympy.polys.polytools.Poly method\), 2380](#)  
[LT\(\) \(sympy.polys.polytools.Poly method\), 2404](#)  
[lu\(\) \(sympy.polys.matrices.ddm.DDM method\), 2690](#)  
[lu\(\) \(sympy.polys.matrices.domainmatrix.DomainMatrix method\), 2679](#)  
[lu\(\) \(sympy.polys.matrices.sdm.SDM method\), 2696](#)  
[lu\\_solve\(\) \(sympy.polys.matrices.ddm.DDM method\), 2690](#)  
[lu\\_solve\(\) \(sympy.polys.matrices.domainmatrix.DomainMatrix method\), 2679](#)  
[lu\\_solve\(\) \(sympy.polys.matrices.sdm.SDM method\), 2696](#)  
[lucas \(class in sympy.functions.combinatorial.numbers\), 441](#)  
[LUdecomposition\(\) \(sympy.matrices.expressions.blockmatrix.BlockMatrix method\), 1382](#)  
[LUdecomposition\(\) \(sympy.matrices.matrices.MatrixBase method\), 1282](#)  
[LUdecomposition\\_Simple\(\) \(sympy.matrices.matrices.MatrixBase method\), 1284](#)  
[LUdecompositionFF\(\) \(sympy.matrices.matrices.MatrixBase method\), 1283](#)  
[LUsolve\(\) \(sympy.matrices.matrices.MatrixBase method\), 1287](#)  
**M**  
[magnitude\(\) \(sympy.physics.vector.vector.Vector method\), 1645](#)  
[magnitude\(\) \(sympy.vector.vector.Vector method\), 1456](#)  
[major \(sympy.geometry.ellipse.Ellipse property\), 2261](#)  
[make\\_mod\\_elt\(\) \(in module sympy.polys.numberfields.modules\), 1401](#)

[2733](#)  
[make\\_perm\(\)](#) (`sympy.combinatorics.perm_groups.PermutationGroup` `method`), [318](#)  
[make\\_routine\(\)](#) (`in module sympy.utilities.codegen`), [2055](#)  
[Manifold](#) (`class in sympy.diffgeom`), [2799](#)  
[manualintegrate\(\)](#) (`in module sympy.integrals.manualintegrate`), [595](#)  
[map\(\)](#) (`sympy.polys.domains.domain.Domain` `method`), [2516](#)  
[maple\\_code\(\)](#) (`in module sympy.printing.maple`), [2154](#)  
[MapleCodePrinter](#) (`class in sympy.printing.maple`), [2154](#)  
[marcumq](#) (`class in sympy.functions.special.bessel`), [503](#)  
[marginal\\_distribution\(\)](#) (`in module sympy.stats`), [2935](#)  
[mass](#) (`sympy.physics.mechanics.particle.Particle` `property`), [1745](#)  
[mass\\_matrix](#) (`sympy.physics.mechanics.jointsmethod.JointSMMethod` `property`), [1782](#)  
[mass\\_matrix](#) (`sympy.physics.mechanics.kane.KanesMethod` `property`), [1766](#)  
[mass\\_matrix](#) (`sympy.physics.mechanics.lagrange.LagrangianMethod` `property`), [1769](#)  
[mass\\_matrix\\_full](#) (`sympy.physics.mechanics.jointsmethod.JointSMMethod` `property`), [1782](#)  
[mass\\_matrix\\_full](#) (`sympy.physics.mechanics.kane.KanesMethod` `property`), [1766](#)  
[mass\\_matrix\\_full](#) (`sympy.physics.mechanics.lagrange.LagrangianMethod` `property`), [1769](#)  
[masscenter\\_vel\(\)](#) (`sympy.physics.mechanics.body.Body` `method`), [1762](#)  
[MatAdd](#) (`class in sympy.matrices.expressions`), [1372](#)  
[match\(\)](#) (`sympy.core.basic.Basic` `method`), [935](#)  
[match\(\)](#) (`sympy.polys.polytools.Poly` `method`), [2405](#)  
[matches\(\)](#) (`sympy.core.basic.Basic` `method`), [936](#)  
[matches\(\)](#) (`sympy.solvers.diophantine.diophantine.DiophantineEquationType` `method`), [745](#)  
[mathematica\\_code\(\)](#) (`in module sympy.printing.mathematica`), [2154](#)  
[MathieuBase](#) (`class in sympy.functions.special.mathieu_functions`), [529](#)  
[mathieuc](#) (`class in sympy.functions.special.mathieu_functions`), [530](#)  
[mathieucprime](#) (`class in sympy.functions.special.mathieu_functions`), [531](#)  
[mathieus](#) (`class in sympy.functions.special.mathieu_functions`), [529](#)  
[mathieusprime](#) (`class in sympy.functions.special.mathieu_functions`), [531](#)  
[mathml\(\)](#) (`in module sympy.printing.mathml`), [2176](#)  
[mathml\\_tag\(\)](#) (`sympy.printing.mathml.MathMLContentPrinter` `method`), [2175](#)  
[mathml\\_tag\(\)](#) (`sympy.printing.mathml.MathMLPresentationPrinter` `method`), [2175](#)  
[MathMLContentPrinter](#) (`class in sympy.printing.mathml`), [2175](#)  
[MathMLPresentationPrinter](#) (`class in sympy.printing.mathml`), [2175](#)  
[MathMLPrinterBase](#) (`class in sympy.printing.mathml`), [2175](#)  
[MatMul](#) (`class in sympy.matrices.expressions`), [1372](#)  
[matmul\(\)](#) (`sympy.polys.matrices.ddm.DDM` `method`), [2690](#)  
[matmul\(\)](#) (`sympy.polys.matrices.domainmatrix.DomainMatrix` `method`), [2680](#)  
[matmul\(\)](#) (`sympy.polys.matrices.sdm.SDM` `method`), [2680](#)  
[MatplotlibBackend](#) (`class in sympy.plotting.plot`), [2868](#)  
[MatPow](#) (`class in sympy.matrices.expressions`), [1372](#)  
[Matrix](#) (`in module sympy.matrices.dense`), [361](#)  
[matrix\(\)](#) (`sympy.polys.numberfields.modules.ModuleHomomorphism` `method`), [2734](#)  
[matrix2numpy\(\)](#) (`in module sympy.matrices.dense`), [1323](#)  
[matrix\\_exp\(\)](#) (`in module sympy.solvers.ode.systems`), [813](#)  
[matrix\\_exp\\_jordan\\_form\(\)](#) (`in module sympy.solvers.ode.systems`), [812](#)  
[matrix\\_fgln\(\)](#) (`in module sympy.solvers.ode.systems`), [2640](#)  
[matrix\\_form\(\)](#) (`sympy.liealgebras.weyl_group.WeylGroup` `method`), [2340](#)  
[matrix\\_multiply\\_elementwise\(\)](#) (`in module sympy.matrices.dense`), [1319](#)  
[matrix\\_rep\(\)](#) (`in module sympy.physics.secondquant`), [1564](#)  
[matrix\\_to\\_density\(\)](#) (`in module sympy.physics.quantum.qubit`), [1853](#)  
[matrix\\_to\\_qubit\(\)](#) (`in module sympy.physics.quantum.qubit`), [1853](#)



|                                                                                                              |                                                                                                           |
|--------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------|
| <code>sympy.physics.quantum.qubit</code> ), 1853                                                             | <code>property</code> ), 1132                                                                             |
| <code>matrix_to_vector()</code> (in module <code>sympy.vector</code> ), 1470                                 | <code>max_norm()</code> ( <code>sympy.polys.polyclasses.DMP</code> method), 2565                          |
| <code>MatrixBase</code> (class in module <code>sympy.matrices.matrices</code> ), 1280                        | <code>max_norm()</code> ( <code>sympy.polys.polytools.Poly</code> method), 2405                           |
| <code>MatrixCalculus</code> (class in module <code>sympy.matrices.matrices</code> ), 1279                    | <code>max_shear_force()</code> ( <code>sympy.physics.continuum_mechanics.beam.Beam</code> method), 1959   |
| <code>MatrixCommon</code> (class in module <code>sympy.matrices.common</code> ), 1327                        | <code>max_shear_force()</code> ( <code>sympy.physics.continuum_mechanics.beam.Beam3D</code> method), 1986 |
| <code>MatrixDeterminant</code> (class in module <code>sympy.matrices.matrices</code> ), 1227                 | <code>maximal_order()</code> ( <code>sympy.polys.domains.AlgebraicField</code> method), 2545              |
| <code>MatrixEigen</code> (class in module <code>sympy.matrices.matrices</code> ), 1237                       | <code>maximum()</code> (in module <code>sympy.calculus.util</code> ), 247                                 |
| <code>MatrixError</code> (class in module <code>sympy.matrices.matrices</code> ), 1318                       | <code>Maxwell()</code> (in module <code>sympy.stats</code> ), 2914                                        |
| <code>MatrixExpr</code> (class in module <code>sympy.matrices.expressions</code> ), 1370                     | <code>MCodePrinter</code> (class in module <code>sympy.printing.mathematica</code> ), 2154                |
| <code>MatrixGamma()</code> (in module <code>sympy.stats</code> ), 2957                                       | <code>mdft()</code> (in module <code>sympy.physics.matrices</code> ), 1537                                |
| <code>MatrixKind</code> (class in module <code>sympy.matrices.common</code> ), 1360                          | <code>measure</code> ( <code>sympy.sets.sets.Set</code> property), 1191                                   |
| <code>MatrixNormal()</code> (in module <code>sympy.stats</code> ), 2958                                      | <code>measure_all()</code> (in module <code>sympy.physics.quantum.qubit</code> ), 1853                    |
| <code>MatrixPermute</code> (class in module <code>sympy.matrices.expressions</code> ), 1378                  | <code>measure_all_oneshot()</code> (in module <code>sympy.physics.quantum.qubit</code> ), 1854            |
| <code>MatrixReductions</code> (class in module <code>sympy.matrices.matrices</code> ), 1232                  | <code>measure_partial()</code> (in module <code>sympy.physics.quantum.qubit</code> ), 1854                |
| <code>MatrixSet</code> (class in module <code>sympy.matrices.expressions</code> ), 1379                      | <code>measure_partial_oneshot()</code> (in module <code>sympy.physics.quantum.qubit</code> ), 1855        |
| <code>MatrixSolve</code> (class in module <code>sympy.codegen.matrix_nodes</code> ), 1121                    | <code>medial</code> ( <code>sympy.geometry.polygon.Triangle</code> property), 2301                        |
| <code>MatrixSubspaces</code> (class in module <code>sympy.matrices.matrices</code> ), 1235                   | <code>median()</code> (in module <code>sympy.stats</code> ), 2969                                         |
| <code>MatrixSymbol</code> (class in module <code>sympy.matrices.expressions</code> ), 1372                   | <code>medians</code> ( <code>sympy.geometry.polygon.Triangle</code> property), 2301                       |
| <code>Max</code> (class in module <code>sympy.functions.elementary.miscellaneous</code> ), 421               | <code>Medium</code> (class in module <code>sympy.physics.optics.medium</code> ), 1870                     |
| <code>max</code> ( <code>sympy.codegen.ast.FloatType</code> property), 1132                                  | <code>meets()</code> ( <code>sympy.series.gruntz.SubsSet</code> method), 625                              |
| <code>max()</code> ( <code>sympy.combinatorics.permutations.PermutationGroup</code> method), 275             | <code>MeijerG</code> (class in module <code>sympy.functions.special.hyper</code> ), 522                   |
| <code>max_bending_moment()</code> ( <code>sympy.physics.continuum_mechanics.beam.Beam3D</code> method), 1984 | <code>MeijerGDefinite()</code> (in module <code>sympy.integrals.meijerint</code> ), 574                   |
| <code>max_bmoment()</code> ( <code>sympy.physics.continuum_mechanics.beam.Beam3D</code> method), 1959        | <code>meijerGDefinite()</code> (in module <code>sympy.integrals.meijerint</code> ), 574                   |
| <code>max_bmoment()</code> ( <code>sympy.physics.continuum_mechanics.beam.Beam3D</code> method), 1984        | <code>meijerGDefinite3D()</code> (in module <code>sympy.integrals.meijerint</code> ), 574                 |
| <code>max_deflection()</code> ( <code>sympy.physics.continuum_mechanics.beam.Beam3D</code> method), 1959     | <code>meijerGDefinite3D()</code> (in module <code>sympy.integrals.meijerint</code> ), 574                 |
| <code>max_deflection()</code> ( <code>sympy.physics.continuum_mechanics.beam.Beam3D</code> method), 1985     | <code>MeijerGTransform</code> (class in module <code>sympy.integrals.transforms</code> ), 576             |
| <code>max_div</code> ( <code>sympy.combinatorics.perm_groups.PermutationGroup</code> property), 318          | <code>MeijerGTransform3D</code> (class in module <code>sympy.integrals.transforms</code> ), 576           |
| <code>max_exponent</code> ( <code>sympy.codegen.ast.FloatType</code> property), 1132                         | <code>member_label</code> ( <code>sympy.physics.continuum_mechanics.truss.Truss</code> property), 2025    |

- [property](#)), 2025
- [memoize\\_property\(\)](#) (in module [sympy.utilities.decorator](#)), 2059
- [mensor\(\)](#) ([sympy.algebras.Quaternion](#) method), 892
- [merge](#) (class in [sympy.codegen.fnodes](#)), 1155
- [merge\\_solution\(\)](#) (in module [sympy.solvers.diophantine.diophantine](#)), 737
- [mersenne\\_prime\\_exponent\(\)](#) (in module [sympy.ntheory.factor\\_](#)), 1506
- [method](#) ([sympy.physics.mechanics.jointsmethod.JointsMethod](#) property), 1782
- [metric\\_to\\_Christoffel\\_1st\(\)](#) (in module [sympy.diffgeom](#)), 2818
- [metric\\_to\\_Christoffel\\_2nd\(\)](#) (in module [sympy.diffgeom](#)), 2819
- [metric\\_to\\_Ricci\\_components\(\)](#) (in module [sympy.diffgeom](#)), 2819
- [metric\\_to\\_Riemann\\_components\(\)](#) (in module [sympy.diffgeom](#)), 2819
- [mgamma\(\)](#) (in module [sympy.physics.matrices](#)), 1537
- [midpoint](#) ([sympy.geometry.line.Segment](#) property), 2235
- [midpoint\(\)](#) ([sympy.geometry.point.Point](#) method), 2207
- [mignotte\\_sep\\_bound\\_squared\(\)](#) ([sympy.polys.polyclasses.DMP](#) method), 2565
- [MIMOFeedback](#) (class in [sympy.physics.control.lti](#)), 1927
- [MIMOParallel](#) (class in [sympy.physics.control.lti](#)), 1925
- [MIMOSeries](#) (class in [sympy.physics.control.lti](#)), 1922
- [Min](#) (class in [sympy.functions.elementary.miscellaneous](#)), 420
- [min\(\)](#) ([sympy.combinatorics.permutations.Permutation](#) method), 276
- [min\\_exponent](#) ([sympy.codegen.ast.FloatType](#) property), 1132
- [min\\_qubits](#) ([sympy.physics.quantum.gate.CGate](#) property), 1842
- [min\\_qubits](#) ([sympy.physics.quantum.gate.CNotGate](#) property), 1843
- [min\\_qubits](#) ([sympy.physics.quantum.gate.Gate](#) property), 1844
- [minimal\\_block\(\)](#) ([sympy.combinatorics.perm\\_groups.perm\\_group](#) method), 319
- [minimal\\_blocks\(\)](#) ([sympy.combinatorics.perm\\_groups.perm\\_group](#) method), 319
- [minimal\\_polynomial\(\)](#) (in module [sympy.polys.numberfields.minpoly](#)), 2710
- [minimum\(\)](#) (in module [sympy.calculus.util](#)), 248
- [minlex\(\)](#) (in module [sympy.utilities.iterables](#)), 2083
- [minor](#) ([sympy.geometry.ellipse.Ellipse](#) property), 2262
- [minor\(\)](#) ([sympy.matrices.matrices.MatrixDeterminant](#) method), 1231
- [minor\\_submatrix\(\)](#) ([sympy.matrices.matrices.MatrixDeterminant](#) method), 1231
- [minpoly\(\)](#) (in module [sympy.polys.numberfields.minpoly](#)), 2711
- [minpoly\\_of\\_element\(\)](#) ([sympy.core.numbers.AlgebraicNumber](#) method), 991
- [mirror\\_formula\(\)](#) (in module [sympy.physics.optics.utils](#)), 1884
- [mobius](#) (class in [sympy.ntheory](#)), 1528
- [mobius\\_transform\(\)](#) (in module [sympy.discrete.transforms](#)), 1087
- [mobiusrange\(\)](#) ([sympy.ntheory.generate.Sieve](#) method), 1477
- [Mod](#) (class in [sympy.core.mod](#)), 1018
- [mod](#) ([sympy.polys.domains.AlgebraicField](#) attribute), 2545
- [mod\\_inverse\(\)](#) (in module [sympy.core.numbers](#)), 1004
- [in modgcd\\_bivariate\(\)](#) (in module [sympy.polys.modulargcd](#)), 2645
- [in modgcd\\_multivariate\(\)](#) (in module [sympy.polys.modulargcd](#)), 2647
- [in modgcd\\_univariate\(\)](#) (in module [sympy.polys.modulargcd](#)), 2644
- [modules](#), 884
- [sympy.abc](#), 884
- [sympy.algebras](#), 885
- [sympy.assumptions](#), 186
- [sympy.assumptions.ask](#), 191
- [sympy.assumptions.assume](#), 193
- [sympy.assumptions.predicates](#), 200
- [sympy.assumptions.refine](#), 197
- [sympy.calculus](#), 233
- [sympy.calculus.euler](#), 233
- [sympy.calculus.finite\\_diff](#), 240
- [sympy.calculus.singularities](#), 234
- [sympy.combinatorics](#), 244
- [sympy.combinatorics.perm\\_groups](#), 2742
- [sympy.combinatorics.perm\\_groups.perm\\_group](#), 2752
- [sympy.codegen.algorithms](#), 1157
- [sympy.codegen.approximations](#), 1122

[sympy.codegen.ast](#), 1124  
[sympy.codegen.cfunctions](#), 1143  
[sympy.codegen.cnodes](#), 1148  
[sympy.codegen.cutils](#), 1159  
[sympy.codegen.cxxnodes](#), 1149  
[sympy.codegen.fnodes](#), 1149  
[sympy.codegen.futils](#), 1159  
[sympy.codegen.matrix\\_nodes](#), 1121  
[sympy.codegen.pyutils](#), 1159  
[sympy.codegen.rewriting](#), 1118  
[sympy.combinatorics.generators](#), 287  
[sympy.combinatorics.graycode](#), 347  
[sympy.combinatorics.group\\_constructs](#), 363  
[sympy.combinatorics.group\\_numbers](#), 355  
[sympy.combinatorics.named\\_groups](#), 352  
[sympy.combinatorics.partitions](#), 251  
[sympy.combinatorics.perm\\_groups](#), 288  
[sympy.combinatorics.permutations](#), 257  
[sympy.combinatorics.polyhedron](#), 332  
[sympy.combinatorics.prufer](#), 335  
[sympy.combinatorics.subsets](#), 340  
[sympy.combinatorics.tensor\\_can](#), 365  
[sympy.combinatorics.testutil](#), 363  
[sympy.combinatorics.util](#), 356  
[sympy.core.add](#), 1013  
[sympy.core.assumptions](#), 923  
[sympy.core.basic](#), 927  
[sympy.core.cache](#), 927  
[sympy.core.containers](#), 1068  
[sympy.core.core](#), 945  
[sympy.core.evalf](#), 1065  
[sympy.core.expr](#), 947  
[sympy.core.exprtools](#), 1071  
[sympy.core.function](#), 1039  
[sympy.core.kind](#), 1073  
[sympy.core.mod](#), 1018  
[sympy.core.mul](#), 1009  
[sympy.core.multidimensional](#), 1039  
[sympy.core.numbers](#), 981  
[sympy.core.power](#), 1005  
[sympy.core.random](#), 1077  
[sympy.core.relational](#), 1018  
[sympy.core.singleton](#), 945  
[sympy.core.symbol](#), 976  
[sympy.core.sympify](#), 918  
[sympy.core.traversal](#), 1080  
[sympy.crypto.crypto](#), 2762  
[sympy.diffgeom](#), 2799  
[sympy.discrete](#), 1083  
[sympy.discrete.convolutions](#), 1089  
[sympy.discrete.transforms](#), 1083  
[sympy.functions](#), 381  
[sympy.functions.special.bessel](#), 497  
[sympy.functions.special.beta\\_functions](#), 470  
[sympy.functions.special.elliptic\\_integrals](#), 526  
[sympy.functions.special.error\\_functions](#), 471  
[sympy.functions.special.gamma\\_functions](#), 459  
[sympy.functions.special.mathieu\\_functions](#), 529  
[sympy.functions.special.polynomials](#), 532  
[sympy.functions.special.singularity\\_functions](#), 456  
[sympy.functions.special.zeta\\_functions](#), 514  
[sympy.geometry.curve](#), 2247  
[sympy.geometry.ellipse](#), 2253  
[sympy.geometry.entity](#), 2194  
[sympy.geometry.line](#), 2216  
[sympy.geometry.plane](#), 2303  
[sympy.geometry.point](#), 2202  
[sympy.geometry.polygon](#), 2273  
[sympy.geometry.util](#), 2197  
[sympy.holonomic](#), 2312  
[sympy.integrals](#), 575  
[sympy.integrals.intpoly](#), 617  
[sympy.integrals.meijerint](#), 568  
[sympy.integrals.meijerint\\_doc](#), 564  
[sympy.integrals.transforms](#), 576  
[sympy.interactive](#), 2116  
[sympy.interactive.printing](#), 2119  
[sympy.interactive.session](#), 2116  
[sympy.lialgebras](#), 2326  
[sympy.logic](#), 1160  
[sympy.logic.inference](#), 1184  
[sympy.matrices](#), 1217  
[sympy.matrices.common](#), 1327  
[sympy.matrices.expressions](#), 1369  
[sympy.matrices.expressions.blockmatrix](#), 1380  
[sympy.matrices.immutable](#), 1369  
[sympy.matrices.matrices](#), 1218  
[sympy.matrices.sparse](#), 1364  
[sympy.matrices.sparsetools](#), 1365  
[sympy.ntheory.bbp\\_pi](#), 1531  
[sympy.ntheory.continued\\_fraction](#), 1523  
[sympy.ntheory.digits](#), 1526  
[sympy.ntheory.ecm](#), 1531

`sympy.ntheory.egyptian_fraction`, 1529  
`sympy.ntheory.factor_`, 1486  
`sympy.ntheory.generate`, 1476  
`sympy.ntheory.modular`, 1509  
`sympy.ntheory.multinomial`, 1511  
`sympy.ntheory.partitions_`, 1513  
`sympy.ntheory.primetest`, 1513  
`sympy.ntheory.qs`, 1532  
`sympy.ntheory.residue_ntheory`, 1518  
`sympy.parsing`, 2122  
`sympy.parsing.sym_expr`, 2130  
`sympy.physics`, 1533  
`sympy.physics.continuum_mechanics.beam`, 1949  
`sympy.physics.continuum_mechanics.truss`, 2021  
`sympy.physics.control`, 1890  
`sympy.physics.control.lti`, 1891  
`sympy.physics.hep.gamma_matrices`, 1589  
`sympy.physics.hydrogen`, 1533  
`sympy.physics.matrices`, 1537  
`sympy.physics.mechanics`, 1675  
`sympy.physics.mechanics.body`, 1756  
`sympy.physics.mechanics.joint`, 1770  
`sympy.physics.mechanics.jointsmethod`, 1780  
`sympy.physics.mechanics.kane`, 1764  
`sympy.physics.mechanics.lagrange`, 1767  
`sympy.physics.mechanics.linearize`, 1787  
`sympy.physics.mechanics.particle`, 1743  
`sympy.physics.mechanics.rigidbody`, 1746  
`sympy.physics.mechanics.system`, 1783  
`sympy.physics.optics.gaussopt`, 1857  
`sympy.physics.optics.medium`, 1870  
`sympy.physics.optics.polarization`, 1871  
`sympy.physics.optics.utils`, 1880  
`sympy.physics.optics.waves`, 1887  
`sympy.physics.paulialgebra`, 1539  
`sympy.physics.qho_ld`, 1539  
`sympy.physics.quantum.anticommutator`, 1791  
`sympy.physics.quantum.cartesian`, 1801  
`sympy.physics.quantum.cg`, 1792  
`sympy.physics.quantum.circuitplot`, 1840  
`sympy.physics.quantum.commutator`, 1795  
`sympy.physics.quantum.constants`, 1796  
`sympy.physics.quantum.dagger`, 1797  
`sympy.physics.quantum.gate`, 1842  
`sympy.physics.quantum.grover`, 1847  
`sympy.physics.quantum.hilbert`, 1802  
`sympy.physics.quantum.innerproduct`, 1798  
`sympy.physics.quantum.operator`, 1807  
`sympy.physics.quantum.operatorset`, 1812  
`sympy.physics.quantum.piab`, 1857  
`sympy.physics.quantum.qapply`, 1814  
`sympy.physics.quantum.qft`, 1850  
`sympy.physics.quantum.qubit`, 1850  
`sympy.physics.quantum.represent`, 1815  
`sympy.physics.quantum.shor`, 1856  
`sympy.physics.quantum.spin`, 1820  
`sympy.physics.quantum.state`, 1832  
`sympy.physics.quantum.tensorproduct`, 1799  
`sympy.physics.secondquant`, 1542  
`sympy.physics.sho`, 1541  
`sympy.physics.units.dimensions`, 1584  
`sympy.physics.units.prefixes`, 1586  
`sympy.physics.units.quantities`, 1587  
`sympy.physics.units.unitsystem`, 1587  
`sympy.physics.units.util`, 1588  
`sympy.physics.vector`, 1592  
`sympy.physics.vector.functions`, 1659  
`sympy.physics.vector.point`, 1652  
`sympy.physics.wigner`, 1566  
`sympy.plotting.plot`, 2820  
`sympy.plotting.pygletplot`, 2869  
`sympy.polys`, 2360  
`sympy.polys.numberfields.modules`, 2716  
`sympy.polys.numberfields.subfield`, 2711  
`sympy.polys.polyconfig`, 2643  
`sympy.polys.polyoptions`, 2642  
`sympy.polys.solvers`, 2666  
`sympy.printing.aesaracode`, 2166  
`sympy.printing.c`, 2140  
`sympy.printing.codeprinter`, 2183  
`sympy.printing.conventions`, 2183



- `sympy.printing.cxx`, 2144
- `sympy.printing.fortran`, 2147
- `sympy.printing.gtk`, 2170
- `sympy.printing.jscode`, 2155
- `sympy.printing.julia`, 2157
- `sympy.printing.lambdarepr`, 2170
- `sympy.printing.latex`, 2170
- `sympy.printing.maple`, 2154
- `sympy.printing.mathematica`, 2152
- `sympy.printing.mathml`, 2175
- `sympy.printing.octave`, 2160
- `sympy.printing.precedence`, 2184
- `sympy.printing.pretty`, 2139
- `sympy.printing.pretty.pretty`, 2139
- `sympy.printing.pretty.pretty_symbology`, 2184
- `sympy.printing.pretty.stringpict`, 2186
- `sympy.printing.preview`, 2181
- `sympy.printing.printer`, 2135
- `sympy.printing.pycode`, 2176
- `sympy.printing.python`, 2177
- `sympy.printing.rcode`, 2144
- `sympy.printing.repr`, 2177
- `sympy.printing.rust`, 2164
- `sympy.printing.str`, 2178
- `sympy.printing.tree`, 2178
- `sympy.sets.conditionset`, 1215
- `sympy.sets.fancysets`, 1203
- `sympy.sets.powerset`, 1213
- `sympy.sets.sets`, 1185
- `sympy.simplify.combsimp`, 683
- `sympy.simplify.cse_main`, 685
- `sympy.simplify.epathtools`, 688
- `sympy.simplify.hyperexpand`, 687
- `sympy.simplify.hyperexpand_doc`, 699
- `sympy.simplify.powsimp`, 680
- `sympy.simplify.radsimp`, 670
- `sympy.simplify.ratsimp`, 678
- `sympy.simplify.simplify`, 154
- `sympy.simplify.sqrtdenest`, 684
- `sympy.simplify.trigsimp`, 679
- `sympy.solvers`, 836
- `sympy.solvers.inequalities`, 750
- `sympy.solvers.ode`, 755
- `sympy.solvers.ode.ode`, 822
- `sympy.solvers.pde`, 835
- `sympy.solvers.recurr`, 853
- `sympy.solvers.solveset`, 858
- `sympy.stats`, 2873
- `sympy.stats.crv`, 2981
- `sympy.stats.crv_types`, 2981
- `sympy.stats.frv`, 2981
- `sympy.stats.frv_types`, 2981
- `sympy.stats.rv`, 2980
- `sympy.tensor`, 1387
- `sympy.tensor.array`, 1387
- `sympy.tensor.array.expressions`, 1395
- `sympy.tensor.index_methods`, 1406
- `sympy.tensor.indexed`, 1398
- `sympy.tensor.tensor`, 1409
- `sympy.tensor.toperators`, 1423
- `sympy.testing`, 2027
- `sympy.testing.pytest`, 2027
- `sympy.testing.randtest`, 2030
- `sympy.testing.runtests`, 2030
- `sympy.utilities`, 2038
- `sympy.utilities.autowrap`, 2039
- `sympy.utilities.codegen`, 2046
- `sympy.utilities.decorator`, 2057
- `sympy.utilities.enumerative`, 2060
- `sympy.utilities.exceptions`, 2066
- `sympy.utilities.iterables`, 2071
- `sympy.utilities.lambdify`, 2099
- `sympy.utilities.memoization`, 2110
- `sympy.utilities.misc`, 2110
- `sympy.utilities.pkgdata`, 2115
- `sympy.utilities.source`, 2116
- `sympy.utilities.timeutils`, 2116
- `sympy.vector`, 1425
- Module (*class in sympy.codegen.fnodes*), 1151
- Module (*class in sympy.polys.agca.modules*), 2453
- Module (*class in sympy.polys.numberfields.modules*), 2719
- `module_quotient()` (*sympy.polys.agca.modules.SubModule method*), 2459
- ModuleElement (*class in sympy.polys.numberfields.modules*), 2729
- ModuleEndomorphism (*class in sympy.polys.numberfields.modules*), 2734
- ModuleHomomorphism (*class in sympy.polys.agca.homomorphisms*), 2470
- ModuleHomomorphism (*class in sympy.polys.numberfields.modules*), 2733
- Moment (*class in sympy.stats*), 2976
- `moment()` (*in module sympy.stats*), 2977
- `moment_load_vector` (*sympy.physics.continuum\_mechanics.beam.Beam3 property*), 1986
- `momentum` (*sympy.physics.quantum.cartesian.PxBra property*), 1801

`momentum(sympy.physics.quantum.cartesian.PxKet property)`, 1802

`monic()` (in module `sympy.polys.polytools`), 2369

`monic()` (`sympy.polys.polyclasses.DMP method`), 2565

`monic()` (`sympy.polys.polytools.Poly method`), 2405

`monic()` (`sympy.polys.rings.PolyElement method`), 2559

`MonogenicFiniteExtension` (class in `sympy.polys.agca.extensions`), 2475

`Monomial` (class in `sympy.polys.monomials`), 2429

`monomial_basis()` (`sympy.polys.rings.PolyRing method`), 2554

`monomial_count()` (in module `sympy.polys.monomials`), 2430

`MonomialOrder` (class in `sympy.polys.orderings`), 2431

`monoms()` (`sympy.polys.polyclasses.DMP method`), 2565

`monoms()` (`sympy.polys.polytools.Poly method`), 2405

`monoms()` (`sympy.polys.rings.PolyElement method`), 2559

`monotonicity_helper()` (in module `sympy.calculus.singularities`), 238

`Morphism` (class in `sympy.categories`), 2742

`morphisms` (`sympy.categories.diagram_drawing.Diagrammorphism property`), 2755

`Moyal()` (in module `sympy.stats`), 2915

`MpmathPrinter` (class in `sympy.printing.pycode`), 2176

`mr()` (in module `sympy.ntheory.primetest`), 1514

`mr()` (in module `sympy.series.gruntz`), 624

`mr_leadterm()` (in module `sympy.series.gruntz`), 623

`mr_max1()` (in module `sympy.series.gruntz`), 624

`mr_max3()` (in module `sympy.series.gruntz`), 624

`msigma()` (in module `sympy.physics.matrices`), 1538

`msubs()` (in module `sympy.physics.mechanics`), 1789

`mueller_matrix()` (in module `sympy.physics.optics.polarization`), 1875

`Mul` (class in `sympy.core.mul`), 1009

`mul()` (`sympy.algebras.Quaternion method`), 892

`mul()` (`sympy.polys.domains.domain.Domain method`), 2516

`mul()` (`sympy.polys.matrices.domainmatrix.DomainMatrix method`), 2681

`mul()` (`sympy.polys.matrices.sdm.SDM method`), 2696

`mul()` (`sympy.polys.numberfields.modules.Submodule method`), 2727

`mul()` (`sympy.polys.polyclasses.DMF method`), 2568

`mul()` (`sympy.polys.polyclasses.DMP method`), 2565

`mul()` (`sympy.polys.polytools.Poly method`), 2406

`mul()` (`sympy.polys.rings.PolyRing method`), 2554

`mul_ground()` (`sympy.polys.polyclasses.DMP method`), 2565

`mul_ground()` (`sympy.polys.polytools.Poly method`), 2406

`mul_inv()` (`sympy.combinatorics.permutations.Permutation method`), 276

`mul_xin()` (in module `sympy.polys.ring_series`), 2665

`mult_tab()` (`sympy.polys.numberfields.modules.Module method`), 2721

`MultiFactorial` (class in `sympy.functions.combinatorial.factorials`), 443

`multigamma` (class in `sympy.functions.special.gamma_functions`), 469

`Multinomial()` (in module `sympy.stats`), 2940

`multinomial_coefficients()` (in module `sympy.ntheory.multinomial`), 1512

`multinomial_coefficients_iterator()` (in module `sympy.ntheory.multinomial`), 1512

`multiplicity()` (in module `sympy.ntheory.factor_`), 1488

`multiplicity()` (`sympy.vector.implicitregion.ImplicitRegion method`), 1463

`multiply()` (`sympy.matrices.common.MatrixCommon method`), 1347

`multiply_elementwise()` (`sympy.matrices.common.MatrixCommon method`), 1347

`multiply_ideal()` (`sympy.polys.agca.modules.FreeModule method`), 2455

`multiply_ideal()` (`sympy.polys.agca.modules.Module method`), 2453

`multiply_ideal()` (`sympy.polys.agca.modules.SubModule method`), 2459

`multiset()` (in module `sympy.utilities.iterables`), 2083

- [multiset\\_combinations\(\)](#) (in module [sympy.utilities.iterables](#)), 2083  
[multiset\\_derangements\(\)](#) (in module [sympy.utilities.iterables](#)), 2084  
[multiset\\_partitions\(\)](#) (in module [sympy.utilities.iterables](#)), 2084  
[multiset\\_partitions\\_taocp\(\)](#) (in module [sympy.utilities.enumerative](#)), 2060  
[multiset\\_permutations\(\)](#) (in module [sympy.utilities.iterables](#)), 2086  
[MultisetPartitionTraverser](#) (class in [sympy.utilities.enumerative](#)), 2062  
[MultivariateBeta\(\)](#) (in module [sympy.stats](#)), 2940  
[MultivariateEwens\(\)](#) (in module [sympy.stats](#)), 2941  
[MultivariateLaplace\(\)](#) (in module [sympy.stats](#)), 2937  
[MultivariateNormal\(\)](#) (in module [sympy.stats](#)), 2936  
[MultivariatePolynomialError](#) (class in [sympy.polys.polyerrors](#)), 2644  
[MultivariateT\(\)](#) (in module [sympy.stats](#)), 2942  
[MutableDenseMatrix](#) (class in [sympy.matrices.dense](#)), 1363  
[MutableDenseNDimArray](#) (class in [sympy.tensor.array](#)), 1392  
[MutableSparseMatrix](#) (class in [sympy.matrices.sparse](#)), 1364  
[MutableSparseNDimArray](#) (class in [sympy.tensor.array](#)), 1392  
[Mx](#) (class in [sympy.physics.quantum.circuitplot](#)), 1841  
[Mz](#) (class in [sympy.physics.quantum.circuitplot](#)), 1841  
**N**  
[n](#) ([sympy.combinatorics.graycode.GrayCode](#) property), 348  
[n](#) ([sympy.physics.optics.waves.TWave](#) property), 1889  
[N](#) ([sympy.physics.quantum.shor.CMod](#) property), 1856  
[n](#) ([sympy.polys.numberfields.modules.Module](#) property), 2721  
[n](#) ([sympy.polys.numberfields.modules.ModuleElement](#) property), 2731  
[n](#) ([sympy.series.sequences.RecursiveSeq](#) property), 639  
[N\(\)](#) (in module [sympy.core.evalf](#)), 1068  
[n\(\)](#) ([sympy.core.evalf.EvalfMixin](#) method), 1066  
[n\(\)](#) ([sympy.matrices.common.MatrixCommon](#) method), 1348  
[n\(\)](#) ([sympy.polys.domains.domain.Domain](#) method), 2516  
[n\\_order\(\)](#) (in module [sympy.ntheory.residue\\_ntheory](#)), 1518  
[Nakagami\(\)](#) (in module [sympy.stats](#)), 2916  
[name](#) ([sympy.categories.Category](#) property), 2748  
[name](#) ([sympy.categories.NamedMorphism](#) property), 2744  
[NamedMorphism](#) (class in [sympy.categories](#)), 2744  
[NaN](#) (class in [sympy.core.numbers](#)), 997  
[Nand](#) (class in [sympy.logic.boolalg](#)), 1169  
[nargs](#) ([sympy.core.function.FunctionClass](#) property), 1049  
[native\\_coeffs\(\)](#) ([sympy.core.numbers.AlgebraicNumber](#) method), 992  
[Naturals](#) (class in [sympy.sets.fancysets](#)), 1203  
[Naturals0](#) (class in [sympy.sets.fancysets](#)), 1204  
[nC\(\)](#) (in module [sympy.functions.combinatorial.numbers](#)), 446  
[Ne](#) (in module [sympy.core.relational](#)), 1022  
[nearest\\_common\\_ancestor\(\)](#) ([sympy.polys.numberfields.modules.Module](#) method), 2721  
[necklaces\(\)](#) (in module [sympy.utilities.iterables](#)), 2086  
[neg\(\)](#) ([sympy.polys.domains.domain.Domain](#) method), 2516  
[neg\(\)](#) ([sympy.polys.matrices.ddm.DDM](#) method), 2691  
[neg\(\)](#) ([sympy.polys.matrices.domainmatrix.DomainMatrix](#) method), 2681  
[neg\(\)](#) ([sympy.polys.matrices.sdm.SDM](#) method), 2697  
[neg\(\)](#) ([sympy.polys.polyclasses.DMF](#) method), 2568  
[neg\(\)](#) ([sympy.polys.polyclasses.DMP](#) method), 2565  
[neg\(\)](#) ([sympy.polys.polytools.Poly](#) method), 2406  
[negative](#) ([sympy.core.relational.Relational](#) property), 1020  
[negative](#), 925  
[NegativeBinomial\(\)](#) (in module [sympy.stats](#)), 2884  
[NegativeInfinity](#) (class in [sympy.core.numbers](#)), 998  
[NegativeMultinomial\(\)](#) (in module

`sympy.stats`), 2942  
NegativeOne (class in `sympy.core.numbers`), 996  
NegativePredicate (class in `sympy.assumptions.predicates.order`), 220  
new() (`sympy.polys.domains.gaussiandomains.GaussianElement` class method), 2534  
new() (`sympy.polys.matrices.sdm.SDM` class method), 2697  
new() (`sympy.polys.polytools.Poly` class method), 2407  
new() (`sympy.polys.rootoftools.RootSum` class method), 2436  
newtons\_method() (in module `sympy.codegen.algorithms`), 1157  
newtons\_method\_function() (in module `sympy.codegen.algorithms`), 1158  
next() (`sympy.combinatorics.graycode.GrayCode` method), 348  
next() (`sympy.combinatorics.prufer.Prufer` method), 336  
next() (`sympy.printing.pretty.stringpict.stringPict` static method), 2187  
next\_binary() (`sympy.combinatorics.subsets.Subset` method), 341  
next\_gray() (`sympy.combinatorics.subsets.Subset` method), 341  
next\_lex() (`sympy.combinatorics.partitions.IntegerPartition` method), 254  
next\_lex() (`sympy.combinatorics.permutations.Permutation` method), 276  
next\_lexicographic() (`sympy.combinatorics.subsets.Subset` method), 342  
next\_nonlex() (`sympy.combinatorics.permutations.Permutation` method), 276  
next\_trotterjohnson() (`sympy.combinatorics.permutations.Permutation` method), 277  
nextprime() (in module `sympy.ntheory.generate`), 1480  
nfloat() (in module `sympy.core.function`), 1064  
nine\_point\_circle (`sympy.geometry.polygon.Triangle` property), 2302  
NO (class in `sympy.physics.secondquant`), 1557  
no\_attrs\_in\_subclass (class in `sympy.utilities.decorator`), 2059  
no\_symmetry() (`sympy.tensor.tensor.TensorSymmetry` class method), 1421  
nocache\_fail() (in module `sympy.testing.pytest`), 2027  
Node (class in `sympy.codegen.ast`), 1134  
node\_labels (`sympy.physics.continuum_mechanics.truss.Truss` property), 2025  
node\_positions (`sympy.physics.continuum_mechanics.truss.Truss` property), 2025  
Nodes (`sympy.combinatorics.prufer.Prufer` property), 336  
nodes (`sympy.physics.continuum_mechanics.truss.Truss` property), 2025  
NonElementaryIntegral (class in `sympy.integrals.risch`), 594  
NoneToken (class in `sympy.codegen.ast`), 1135  
nonlinsolve() (in module `sympy.solvers.solve`), 875  
nonnegative, 925  
NonNegativePredicate (class in `sympy.assumptions.predicates.order`), 223  
nonpositive, 925  
NonPositivePredicate (class in `sympy.assumptions.predicates.order`), 222  
NonSquareMatrixError (class in `sympy.matrices.matrices`), 1318  
nonzero, 924  
NonZeroPredicate (class in `sympy.assumptions.predicates.order`), 221  
Nor (class in `sympy.logic.boolalg`), 1169  
Norm (`sympy.physics.quantum.state.Wavefunction` property), 1839  
norm() (`sympy.algebras.Quaternion` method), 892  
norm() (`sympy.matrices.matrices.MatrixBase` method), 1304  
norm() (`sympy.polys.numberfields.modules.PowerBasisElement` method), 2732  
norm() (`sympy.polys.polyclasses.DMP` method), 2565  
norm() (`sympy.polys.polytools.Poly` method), 2407  
Normal() (in module `sympy.stats`), 2917  
normal() (`sympy.core.expr.Expr` method), 971  
normal\_closure() (`sympy.combinatorics.perm_groups.PermGroup` method), 320  
normal\_lines() (`sympy.geometry.ellipse.Ellipse` method), 2263  
normal\_vector (`sympy.geometry.plane.Plane` property), 2308  
NormalGamma() (in module `sympy.stats`), 2943  
normalize() (`sympy.algebras.Quaternion` method), 893



`normalize()` (`sympy.physics.quantum.state.Wavefunction` method), 1839  
`normalize()` (`sympy.physics.vector.vector.Vector` method), 1645  
`normalize()` (`sympy.polys.domains.gaussianintegerdomain` method), 2536  
`normalize()` (`sympy.vector.vector.Vector` method), 1456  
`normalize_theta_set()` (in module `sympy.sets.fancysets`), 1212  
`normalized()` (in module `sympy.physics.quantum.gate`), 1846  
`normalized()` (`sympy.matrices.matrices.MatrixBase` method), 1305  
`NormalPredicate` (class in `sympy.assumptions.predicates.matrices`), 214  
`NormalPSpace` (class in `sympy.stats.crv_types`), 2981  
`Not` (class in `sympy.logic.boolalg`), 1167  
`not_empty_in()` (in module `sympy.calculus.util`), 249  
`not_point()` (`sympy.physics.quantum.circuitplot.CircuitPlot` method), 1841  
`NotAlgebraic` (class in `sympy.polys.polyerrors`), 2644  
`NotInvertible` (class in `sympy.polys.polyerrors`), 2644  
`NotIterable` (class in `sympy.utilities.iterables`), 2071  
`NotReversible` (class in `sympy.polys.polyerrors`), 2644  
`nP()` (in module `sympy.functions.combinatorial.numbers`), 447  
`npartitions()` (in module `sympy.ntheory.partitions_`), 1513  
`nqubits` (`sympy.physics.quantum.gate.CGate` property), 1842  
`nqubits` (`sympy.physics.quantum.gate.Gate` property), 1844  
`nroots()` (in module `sympy.polys.polytools`), 2375  
`nroots()` (`sympy.polys.polytools.Poly` method), 2407  
`nseries()` (`sympy.core.expr.Expr` method), 971  
`nsimplify()` (in module `sympy.simplify.simplify`), 667  
`nsimplify()` (`sympy.core.expr.Expr` method), 972  
`nsolve()` (in module `sympy.solvers.solvers`), 848  
`nT()` (in module `sympy.functions.combinatorial.numbers`), 448  
`nth()` (`sympy.polys.polyclasses.DMP` method), 2565  
`nth()` (`sympy.polys.polytools.Poly` method), 2408  
`nth_order_coefficient` (in module `sympy.polys.polytools`), 2375  
`nth_power_roots_poly()` (`sympy.polys.polytools.Poly` method), 2408  
`NthAlgebraic` (class in `sympy.solvers.ode.single`), 788  
`NthLinearConstantCoeffHomogeneous` (class in `sympy.solvers.ode.single`), 782  
`NthLinearConstantCoeffUndeterminedCoefficients` (class in `sympy.solvers.ode.single`), 783  
`NthLinearConstantCoeffVariationOfParameters` (class in `sympy.solvers.ode.single`), 784  
`NthLinearEulerEqHomogeneous` (class in `sympy.solvers.ode.single`), 785  
`NthLinearEulerEqNonhomogeneousUndeterminedCoefficients` (class in `sympy.solvers.ode.single`), 788  
`NthLinearEulerEqNonhomogeneousVariationOfParameters` (class in `sympy.solvers.ode.single`), 787  
`NthOrderReducible` (class in `sympy.solvers.ode.single`), 789  
`nthroot()` (in module `sympy.simplify.simplify`), 665  
`nthroot_mod()` (in module `sympy.ntheory.residue_ntheory`), 1520  
`ntt()` (in module `sympy.discrete.transforms`), 1085  
`nu` (`sympy.functions.special.hyper.meijerg` property), 525  
`nullspace()` (`sympy.matrices.matrices.MatrixSubspaces` method), 1236  
`nullspace()` (`sympy.polys.matrices.domainmatrix.DomainMatrix` method), 2682  
`nullspace()` (`sympy.polys.matrices.sdm.SDM` method), 2697  
`num` (`sympy.physics.control.lti.TransferFunction` property), 1898  
`num_inputs` (`sympy.physics.control.lti.MIMOParallel` property), 1927  
`num_inputs` (`sympy.physics.control.lti.MIMOSeries` property), 1924  
`num_inputs` (`sympy.physics.control.lti.TransferFunctionMatrix` property), 1920  
`num_inputs` (`sympy.physics.control.lti.MIMOParallel` property), 1927

num\_outputs (sympy.physics.control.lti.MIMOSode\_2nd\_power\_series\_regular() (in module sympy.solvers.ode.ode), 797  
 property), 1924  
 num\_outputs (sympy.physics.control.lti.TransferFunctionMatrix (in module sympy.solvers.deutils), 852  
 property), 1921  
 Number (class in sympy.core.numbers), 981  
 number\_field (sympy.polys.numberfields.modules.Module (in module sympy.solvers.ode.ode), 769  
 property), 2721  
 numbered\_symbols() (in module sympy.solvers.ode.ode), 767  
 sympy.utilities.iterables), 2087  
 NumberKind (in module sympy.core.kind), 1074  
 NumberSymbol (class in sympy.core.numbers), 993  
 numer() (sympy.polys.domains.AlgebraicField method), 2545  
 numer() (sympy.polys.domains.domain.Domain method), 2516  
 numer() (sympy.polys.domains.ExpressionDomain method), 2551  
 numer() (sympy.polys.domains.FractionField method), 2549  
 numer() (sympy.polys.domains.gaussiandomains.GaussianField method), 2539  
 numer() (sympy.polys.domains.GMPYRationalField method), 2532  
 numer() (sympy.polys.domains.RationalField method), 2531  
 numer() (sympy.polys.domains.ring.Ring method), 2521  
 numer() (sympy.polys.polyclasses.DMF method), 2568  
 numerator() (sympy.polys.numberfields.modules.Module (in module sympy.solvers.ode.ode), 769  
 method), 2732

**O**  
 Object (class in sympy.categories), 2742  
 objects (sympy.categories.Category property), 2748  
 objects (sympy.categories.Diagram property), 2751  
 octave\_code() (in module sympy.printing.octave), 2161  
 OctaveCodeGen (class in sympy.utilities.codegen), 2051  
 OctaveCodePrinter (class in sympy.printing.octave), 2161  
 odd, 924  
 OddPredicate (class in sympy.assumptions.predicates.ntheory), 216  
 ode\_1st\_power\_series() (in module sympy.solvers.ode.ode), 795  
 ode\_2nd\_power\_series\_ordinary() (in module sympy.solvers.ode.ode), 796  
 ode\_2nd\_power\_series\_regular() (in module sympy.solvers.ode.ode), 797  
 ode\_solve() (in module sympy.solvers.ode.ode), 797  
 ode\_sol\_simplicity() (in module sympy.solvers.ode.ode), 769  
 odesimp() (in module sympy.solvers.ode.ode), 767  
 of\_type() (sympy.polys.domains.domain.Domain method), 2516  
 offset (sympy.tensor.indexed.IndexedBase property), 1405  
 old\_frac\_field() (sympy.polys.domains.domain.Domain method), 2516  
 old\_poly\_ring() (sympy.polys.domains.domain.Domain method), 2516  
 One (class in sympy.core.numbers), 995  
 one (sympy.polys.domains.domain.Domain attribute), 2516  
 one (sympy.polys.polytools.Poly property), 2408  
 one (sympy.polys.numberfields.modules.Module method), 2722  
 one\_qubit\_box() (sympy.physics.quantum.circuitplot.CircuitDiagramme method), 1841  
 OneQubitGate (class in sympy.physics.quantum.gate), 1844  
 ones() (in module sympy.matrices.dense), 1319  
 ones() (sympy.matrices.common.MatrixCommon class method), 1348  
 ones() (sympy.matrices.dense.DenseMatrix class method), 2682  
 open() (sympy.sets.sets.Interval class method), 1196  
 OperationNotSupported (class in sympy.polys.polyerrors), 2643  
 Operator (class in sympy.physics.quantum.operator), 1809  
 operators (sympy.physics.quantum.state.StateBase property), 1834  
 operators\_to\_state() (in module sympy.physics.quantum.operatorset), 1812  
 opt\_cse() (in module sympy.simplify.cse\_main), 686  
 Optimization (class in sympy.codegen.rewriting), 1119  
 optimize() (in module sympy.codegen.rewriting), 1120  
 OptionError (class in sympy.polys.polyerrors), 2644  
 Options (class in sympy.polys.polyoptions), 2644

2642  
 Or (class in `sympy.logic.boolalg`), 1167  
 OracleGate (class in `sympy.physics.quantum.grover`), 1847  
 orbit() (`sympy.combinatorics.perm_groups.PermutationGroup` method), 321  
 orbit\_rep() (`sympy.combinatorics.perm_groups.PermutationGroup` method), 321  
 orbit\_transversal() (`sympy.combinatorics.perm_groups.PermutationGroup` method), 322  
 orbits() (`sympy.combinatorics.perm_groups.PermutationGroup` method), 322  
 Order (class in `sympy.series.order`), 627  
 order (`sympy.functions.special.bessel.BesselBaseFunction` property), 497  
 order() (`sympy.combinatorics.perm_groups.PermutationGroup` method), 322  
 order() (`sympy.combinatorics.permutations.Permutation` method), 277  
 ordered() (in module `sympy.core.sorting`), 1076  
 ordered\_partitions() (in module `sympy.utilities.iterables`), 2087  
 ordinal() (in module `sympy.utilities.misc`), 2112  
 orient() (`sympy.physics.vector.frame.ReferenceFrame` method), 1631  
 orient\_axis() (`sympy.physics.vector.frame.ReferenceFrame` method), 1632  
 orient\_body\_fixed() (`sympy.physics.vector.frame.ReferenceFrame` method), 1633  
 orient\_explicit() (`sympy.physics.vector.frame.ReferenceFrame` method), 1634  
 orient\_new() (`sympy.vector.coordsysrect.CoordSys3D` method), 1449  
 orient\_new\_axis() (`sympy.vector.coordsysrect.CoordSys3D` method), 1450  
 orient\_new\_body() (`sympy.vector.coordsysrect.CoordSys3D` method), 1451  
 orient\_new\_quaternion() (`sympy.vector.coordsysrect.CoordSys3D` method), 1452  
 orient\_new\_space() (`sympy.vector.coordsysrect.CoordSys3D` method), 1453  
 orient\_quaternion() (`sympy.physics.vector.frame.ReferenceFrame` method), 1635  
 orient\_space\_fixed() (`sympy.physics.vector.frame.ReferenceFrame` method), 1636  
 Orienter (class in `sympy.vector.orienters`), 1466  
 orientation() (`sympy.physics.vector.frame.ReferenceFrame` method), 1638  
 or\_else() (`sympy.physics.domains.AlgebraicField` attribute), 2545  
 origin (`sympy.geometry.point.Point` property), 2307  
 orthocenter (`sympy.geometry.polygon.Triangle` property), 2302  
 orthogonal() (`sympy.algebras.Quaternion` method), 893  
 orthogonal\_direction (`sympy.geometry.point.Point` property), 2307  
 OrthogonalBra (class in `sympy.physics.quantum.state`), 1834  
 orthogonalize() (`sympy.matrices.matrices.MatrixSubspace` class method), 1236  
 OrthogonalKet (class in `sympy.physics.quantum.state`), 1834  
 OrthogonalPredicate (class in `sympy.assumptions.predicates.matrices`), 205  
 OrthogonalState (class in `sympy.physics.quantum.state`), 1834  
 OuterProduct (class in `sympy.physics.quantum.operator`), 1811  
 OutputArgument (class in `sympy.utilities.codegen`), 2052  
 over\_power\_basis() (`sympy.polys.numberfields.modules.ModuleElement` method), 2731

## P

P() (in module `sympy.stats`), 2960  
 p1 (`sympy.geometry.line.LinearEntity` property), 2222  
 p1 (`sympy.geometry.plane.Plane` property), 2308  
 p2 (`sympy.geometry.line.LinearEntity` property), 2223

`padded_key()` (in module `sympy.printing.latex.LatexPrinter` method), 2170

`Parallel` (class in `sympy.physics.control.lti`), 1903

`parallel()` (`sympy.algebras.Quaternion` method), 893

`parallel_axis()` (`sympy.physics.mechanics.particle.Particle` method), 1745

`parallel_axis()` (`sympy.physics.mechanics.rigidbody.RigidBody` method), 1749

`parallel_line()` (`sympy.geometry.line.LinearEntity` method), 2223

`parallel_plane()` (`sympy.geometry.plane.Plane` method), 2308

`parallel_poly_from_expr()` (in module `sympy.polys.polytools`), 2360

`parameter` (`sympy.geometry.curve.Curve` property), 2250

`parameter_value()` (`sympy.geometry.entity.GeometryEntity` method), 2195

`parameter_value()` (`sympy.geometry.plane.Plane` method), 2309

`Parametric2DLineSeries` (class in `sympy.plotting.plot`), 2866

`Parametric3DLineSeries` (class in `sympy.plotting.plot`), 2867

`ParametricIntegral` (class in `sympy.vector.integrals`), 1465

`ParametricRegion` (class in `sympy.vector.parametricregion`), 1461

`ParametricSurfaceSeries` (class in `sympy.plotting.plot`), 2867

`parametrize_ternary_quadratic()` (in module `sympy.solvers.diophantine.diophantine`), 739

`parens()` (`sympy.printing.pretty.stringpict.stringPict` method), 2187

`parent` (`sympy.physics.mechanics.joint.Joint` property), 1772

`parent` (`sympy.polys.numberfields.modules.Module` property), 2722

`parent()` (`sympy.polys.domains.domainelement.DomainElement` method), 2519

`parent()` (`sympy.polys.domains.gaussiandomain.GaussianDomain` method), 2534

`parent_axis` (`sympy.physics.mechanics.joint.Joint` property), 1772

`parent_point` (`sympy.physics.mechanics.joint.Joint` property), 1772

`parenthesize_super()`

`Pareto()` (in module `sympy.stats`), 2919

`pargs` (`sympy.functions.special.delta_functions.Heaviside` property), 456

`parity()` (`sympy.combinatorics.permutations.Permutation` method), 278

`parse_expr()` (in module `sympy.parsing.sympy_parser`), 2122

`parse_latex()` (in module `sympy.parsing.latex`), 2129

`parse_mathematica()` (in module `sympy.parsing.mathematica`), 2125

`parse_maxima()` (in module `sympy.parsing.maxima`), 2125

`partial_velocity()` (in module `sympy.physics.vector.functions`), 1661

`partial_velocity()` (`sympy.physics.vector.frame.ReferenceFrame` method), 1639

`partial_velocity()` (`sympy.physics.vector.point.Point` method), 1655

`PartialDerivative` (class in `sympy.tensor.toperators`), 1423

`Particle` (class in `sympy.physics.mechanics.particle`), 1743

`Partition` (class in `sympy.combinatorics.partitions`), 251

`partition` (class in `sympy.functions.combinatorial.numbers`), 442

`partition` (`sympy.combinatorics.partitions.Partition` property), 252

`partition()` (in module `sympy.solvers.diophantine.diophantine`), 734

`partitions()` (in module `sympy.utilities.iterables`), 2089

`pat_matrix()` (in module `sympy.physics.matrices`), 1538

`Patch` (class in `sympy.diffgeom`), 2800

`pde_1st_linear_constant_coeff()` (in module `sympy.solvers.pde`), 832

`pde_1st_linear_constant_coeff_homogeneous()` (in module `sympy.solvers.pde`), 831

`pde_1st_linear_variable_coeff()` (in module `sympy.solvers.pde`), 834

`pde_separate()` (in module `sympy.solvers.pde`), 826

`pde_separate_add()` (in module



- `sympy.solvers.pde`), 827
- `pde_separate_mul()` (in module `sympy.solvers.pde`), 827
- `pdiv()` (in module `sympy.polys.polytools`), 2362
- `pdiv()` (`sympy.polys.polyclasses.DMP` method), 2565
- `pdiv()` (`sympy.polys.polytools.Poly` method), 2409
- `pdsolve()` (in module `sympy.solvers.pde`), 828
- `per()` (`sympy.matrices.matrices.MatrixDeterminant` method), 1232
- `per()` (`sympy.polys.polyclasses.DMF` method), 2568
- `per()` (`sympy.polys.polyclasses.DMP` method), 2565
- `per()` (`sympy.polys.polytools.Poly` method), 2409
- `perfect_power()` (in module `sympy.ntheory.factor_`), 1488
- `periapsis` (`sympy.geometry.ellipse.Ellipse` property), 2263
- `perimeter` (`sympy.geometry.polygon.Polygon` property), 2281
- `period_find()` (in module `sympy.physics.quantum.shor`), 1856
- `periodic_argument` (class in `sympy.functions.elementary.complexes`), 388
- `periodicity()` (in module `sympy.calculus.util`), 249
- `perm2tensor()` (`sympy.tensor.tensor.TensMul` method), 1419
- `Permutation` (class in `sympy.combinatorics.permutations`), 257
- `PermutationGroup` (class in `sympy.combinatorics.perm_groups`), 288
- `PermutationMatrix` (class in `sympy.matrices.expressions`), 1377
- `PermutationOperator` (class in `sympy.physics.secondquant`), 1560
- `permute()` (`sympy.matrices.common.MatrixCommon` method), 1348
- `permute()` (`sympy.polys.polyclasses.DMP` method), 2565
- `permute_cols()` (`sympy.matrices.common.MatrixCommon` method), 1349
- `permute_rows()` (`sympy.matrices.common.MatrixCommon` method), 1350
- `permute_signs()` (in module `sympy.utilities.iterables`), 2090
- `permutedims()` (in module `sympy.tensor.array`), 1392
- `perpendicular_bisector()` (`sympy.geometry.line.Segment` method), 2235
- `perpendicular_line()` (`sympy.geometry.line.LinearEntity` method), 2224
- `perpendicular_line()` (`sympy.geometry.line.LinearEntity2D` method), 2237
- `perpendicular_line()` (`sympy.geometry.plane.Plane` method), 2309
- `perpendicular_plane()` (`sympy.geometry.plane.Plane` method), 2310
- `perpendicular_segment()` (`sympy.geometry.line.LinearEntity` method), 2224
- `pexquo()` (in module `sympy.polys.polytools`), 2363
- `pexquo()` (`sympy.polys.polyclasses.DMP` method), 2565
- `pexquo()` (`sympy.polys.polytools.Poly` method), 2409
- `pgroup` (`sympy.combinatorics.polyhedron.Polyhedron` property), 333
- `Phase` (in module `sympy.physics.quantum.gate`), 1844
- `phase` (`sympy.physics.optics.waves.TWave` property), 1889
- `phase_retarder()` (in module `sympy.physics.optics.polarization`), 1875
- `PhaseGate` (class in `sympy.physics.quantum.gate`), 1844
- `Pi` (class in `sympy.core.numbers`), 1000
- `pi_hex_digits()` (in module `sympy.ntheory.bbp_pi`), 1531
- `PIABra` (class in `sympy.physics.quantum.piab`), 1857
- `PIABHamiltonian` (class in `sympy.physics.quantum.piab`), 1857
- `PIABKet` (class in `sympy.physics.quantum.piab`), 1857
- `Piecewise` (class in `sympy.functions.elementary.piecewise`), 415
- `piecewise_exclusive()` (in module `sympy.functions.elementary.piecewise`), 418
- `piecewise_fold()` (in module `sympy.functions.elementary.piecewise`), 418

- 419
- `piecewise_integrate()` (`sympy.functions.elementary.piecewise.Piecewise` method), 417
- `PinJoint` (class in `sympy.physics.mechanics.joint`), 1773
- `pinv()` (`sympy.matrices.matrices.MatrixBase` method), 1305
- `pinv_solve()` (`sympy.matrices.matrices.MatrixBase` method), 1306
- `Plane` (class in `sympy.geometry.plane`), 2303
- `Plot` (class in `sympy.plotting.plot`), 2821
- `plot()` (in module `sympy.plotting.plot`), 2825
- `plot3d()` (in module `sympy.plotting.plot`), 2838
- `plot3d_parametric_line()` (in module `sympy.plotting.plot`), 2839
- `plot3d_parametric_surface()` (in module `sympy.plotting.plot`), 2846
- `plot_bending_moment()` (`sympy.physics.continuum_mechanics.beam.Beam` method), 1959
- `plot_bending_moment()` (`sympy.physics.continuum_mechanics.beam.Beam3D` method), 1986
- `plot_deflection()` (`sympy.physics.continuum_mechanics.beam.Beam` method), 1959
- `plot_deflection()` (`sympy.physics.continuum_mechanics.beam.Beam3D` method), 1987
- `plot_gate()` (`sympy.physics.quantum.gate.CGate` method), 1842
- `plot_ild_moment()` (`sympy.physics.continuum_mechanics.beam.Beam` method), 1961
- `plot_ild_reactions()` (`sympy.physics.continuum_mechanics.beam.Beam` method), 1963
- `plot_ild_shear()` (`sympy.physics.continuum_mechanics.beam.Beam` method), 1965
- `plot_implicit()` (in module `sympy.plotting.plot_implicit`), 2847
- `plot_interval()` (`sympy.geometry.curve.Curve` method), 2251
- `plot_interval()` (`sympy.geometry.ellipse.Ellipse` method), 2263
- `plot_interval()` (`sympy.geometry.line.Line` method), 2229
- `plot_interval()` (`sympy.geometry.line.Ray` method), 2232
- `plot_interval()` (`sympy.geometry.line.Segment` method), 2236
- `plot_interval()` (`sympy.geometry.polygon.Polygon` method), 2281
- `plot_loading_results()` (`sympy.physics.continuum_mechanics.beam.Beam` method), 1967
- `plot_loading_results()` (`sympy.physics.continuum_mechanics.beam.Beam3D` method), 1989
- `plot_parametric()` (in module `sympy.plotting.plot`), 2828
- `plot_shear_force()` (`sympy.physics.continuum_mechanics.beam.Beam` method), 1969
- `plot_shear_force()` (`sympy.physics.continuum_mechanics.beam.Beam3D` method), 1990
- `plot_shear_stress()` (`sympy.physics.continuum_mechanics.beam.Beam` method), 1970
- `plot_shear_stress()` (`sympy.physics.continuum_mechanics.beam.Beam3D` method), 1992
- `plot_slope()` (`sympy.physics.continuum_mechanics.beam.Beam` method), 1972
- `plot_slope3D()` (`sympy.physics.continuum_mechanics.beam.Beam3D` method), 1994
- `PlotGrid` (class in `sympy.plotting.plot`), 2855
- `Point` (class in `sympy.diffgeom`), 2806
- `Point` (class in `sympy.geometry.point`), 2202
- `Point` (class in `sympy.physics.vector.point`), 1653
- `point` (`sympy.core.function.Subs` property), 1053
- `point` (`sympy.physics.mechanics.particle.Particle` property), 1746
- `point` (`sympy.diffgeom.CoordSystem` method), 2805
- `Point2D` (class in `sympy.geometry.point`), 2208
- `Point3D` (class in `sympy.geometry.point`), 2212
- `point_cflexure()` (`sympy.physics.continuum_mechanics.beam.Beam` method), 1973
- `point_to_coords()` (`sympy.diffgeom.CoordSystem` method), 2805
- `Printer` (class in `sympy.codegen.ast`), 1135
- `points` (`sympy.geometry.line.LinearEntity` property), 2225
- `pointwise_stabilizer()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 323
- `Poisson()` (in module `sympy.stats`), 2882
- `PoissonProcess` (class in `sympy.stats`), 2954

polar\_lift (class in [332](#)  
[sympy.functions.elementary.complexes](#))  
 387  
 polar\_moment() (sympy.physics.continuum\_mechanics.Beam.Beam3D  
 method), [1996](#)  
 polar\_second\_moment\_of\_area()  
 (sympy.geometry.ellipse.Ellipse  
 method), [2264](#)  
 polar\_second\_moment\_of\_area()  
 (sympy.geometry.polygon.Polygon  
 method), [2281](#)  
 PolarComplexRegion (class in  
 sympy.sets.fancysets), [1212](#)  
 polarizing\_beam\_splitter() (in module  
 sympy.physics.optics.polarization),  
[1876](#)  
 pole\_zero\_numerical\_data()  
 (sympy.physics.control.control\_plots  
 method), [1935](#)  
 pole\_zero\_plot() (sympy.physics.control.control\_plots  
 method), [1933](#)  
 PoleError (class in sympy.core.function),  
[1059](#)  
 poles() (sympy.physics.control.lti.TransferFunction  
 method), [1898](#)  
 PolificationFailed (class in  
 sympy.polys.polyerrors), [2644](#)  
 pollard\_pml() (in module  
 sympy.ntheory.factor\_), [1491](#)  
 pollard\_rho() (in module  
 sympy.ntheory.factor\_), [1490](#)  
 Poly (class in sympy.polys.polytools), [2378](#)  
 poly() (in module sympy.polys.polytools),  
[2360](#)  
 poly() (sympy.polys.numberfields.modules.PowerBasisElement  
 method), [2732](#)  
 poly\_from\_expr() (in module  
 sympy.polys.polytools), [2360](#)  
 poly\_ring() (sympy.polys.domains.domain.Domain  
 method), [2516](#)  
 poly\_unify() (sympy.polys.polyclasses.DMF  
 method), [2568](#)  
 polycyclic\_group()  
 (sympy.combinatorics.perm\_groups.PermGroup  
 method), [324](#)  
 PolyElement (class in sympy.polys.rings),  
[2554](#)  
 polygamma (class in  
 sympy.functions.special.gamma\_functions),  
[462](#)  
 Polygon (class in sympy.geometry.polygon),  
[2273](#)  
 Polyhedron (class in  
 sympy.combinatorics.polyhedron),

polylog (class in  
 sympy.functions.special.zeta\_functions),  
 polynomial() (sympy.series.formal.FormalPowerSeries  
 method), [650](#)  
 PolynomialError (class in  
 sympy.polys.polyerrors), [2644](#)  
 PolynomialRing (class in  
 sympy.polys.domains), [2547](#)  
 PolyRing (class in sympy.polys.rings), [2553](#)  
 polytope\_integrate() (in module  
 sympy.integrals.intpoly), [619](#)  
 pos() (sympy.polys.domains.domain.Domain  
 method), [2516](#)  
 pos\_from() (sympy.physics.vector.point.Point  
 method), [1655](#)  
 POSform() (in module sympy.logic.boolalg),  
[1161](#)  
 posify() (in module sympy.simplify.simplify),  
[668](#)  
 position (sympy.physics.quantum.cartesian.XBra  
 property), [1802](#)  
 position (sympy.physics.quantum.cartesian.XKet  
 property), [1802](#)  
 position\_wrt() (sympy.vector.coordsysrect.CoordSys3D  
 method), [1454](#)  
 position\_x (sympy.physics.quantum.cartesian.PositionState  
 property), [1801](#)  
 position\_y (sympy.physics.quantum.cartesian.PositionState  
 property), [1801](#)  
 position\_z (sympy.physics.quantum.cartesian.PositionState  
 property), [1801](#)  
 PositionBra3D (class in  
 sympy.physics.quantum.cartesian),  
[1801](#)  
 PositionKet3D (class in  
 sympy.physics.quantum.cartesian),  
[1801](#)  
 PositionState3D (class in  
 sympy.physics.quantum.cartesian),  
[1801](#)  
 positive, [925](#)  
 positive\_group() (sympy.liealgebras.type\_a.TypeA  
 method), [2329](#)  
 positive\_roots() (sympy.liealgebras.type\_b.TypeB  
 method), [2330](#)  
 positive\_roots() (sympy.liealgebras.type\_c.TypeC  
 method), [2332](#)  
 positive\_roots() (sympy.liealgebras.type\_d.TypeD  
 method), [2333](#)  
 positive\_roots() (sympy.liealgebras.type\_e.TypeE  
 method), [2335](#)  
 positive\_roots() (sympy.liealgebras.type\_f.TypeF

`method`), 2336  
`positive_roots()` (`sympy.liealgebras.type_g.TypeG` `method`), 2338  
`PositiveDefinitePredicate` (class in `sympy.polys.numberfields.modules`), 2732  
`PositivePredicate` (class in `sympy.assumptions.predicates.matrices`), 207  
`PostDecrement` (class in `sympy.assumptions.predicates.order`), 219  
`PostIncrement` (class in `sympy.codegen.cnodes`), 1148  
`postfixes()` (in module `sympy.utilities.iterables`), 2090  
`PostIncrement` (class in `sympy.codegen.cnodes`), 1148  
`postorder_traversal()` (in module `sympy.core.traversal`), 1080  
`potential_energy` (`sympy.physics.mechanics.particle.Particle` property), 1746  
`potential_energy` (`sympy.physics.mechanics.rigidbody.RigidBody` property), 1749  
`potential_energy()` (in module `sympy.physics.mechanics.functions`), 1754  
`Pow` (class in `sympy.core.power`), 1005  
`pow()` (`sympy.algebras.Quaternion` `method`), 894  
`pow()` (`sympy.matrices.common.MatrixCommon` `method`), 1350  
`pow()` (`sympy.polys.domains.domain.Domain` `method`), 2516  
`pow()` (`sympy.polys.matrices.domainmatrix.DomainMatrix` `method`), 2683  
`pow()` (`sympy.polys.polyclasses.ANP` `method`), 2569  
`pow()` (`sympy.polys.polyclasses.DMF` `method`), 2568  
`pow()` (`sympy.polys.polyclasses.DMP` `method`), 2566  
`pow()` (`sympy.polys.polytools.Poly` `method`), 2409  
`pow_cos_sin()` (`sympy.algebras.Quaternion` `method`), 894  
`pow_xin()` (in module `sympy.polys.ring_series`), 2666  
`powdenest()` (in module `sympy.simplify.powsimp`), 682  
`power_basis_ancestor()` (`sympy.polys.numberfields.modules.Module` `method`), 2722  
`power_representation()` (in module `sympy.solvers.diophantine.diophantine`), 733  
`PowerBasis` (class in `sympy.polys.numberfields.modules`), 2725  
`PowerBasisElement` (class in `sympy.polys.numberfields.modules`), 2732  
`PowerFunction()` (in module `sympy.stats`), 2919  
`PowerSet` (class in `sympy.sets.powerset`), 1213  
`powerset()` (`sympy.sets.sets.Set` `method`), 1192  
`powsimp()` (in module `sympy.simplify.powsimp`), 680  
`powsimp()` (`sympy.core.expr.Expr` `method`), 972  
`pprint_nodes()` (in module `sympy.printing.tree`), 2178  
`Particle` (in module `sympy.solvers.diophantine.diophantine`), 2362  
`pquo()` (in module `sympy.polys.polytools`), 2362  
`pquo()` (`sympy.polys.polyclasses.DMP` `method`), 2566  
`pquo()` (`sympy.polys.polytools.Poly` `method`), 2410  
`PRECEDENCE` (in module `sympy.printing.precedence`), 2184  
`precedence()` (in module `sympy.printing.precedence`), 2184  
`PRECEDENCE_FUNCTIONS` (in module `sympy.printing.precedence`), 2184  
`PRECEDENCE_VALUES` (in module `sympy.printing.precedence`), 2184  
`PrecisionExhausted` (class in `sympy.core.evalf`), 1067  
`PreDecrement` (class in `sympy.codegen.cnodes`), 1148  
`Predicate` (class in `sympy.assumptions.assume`), 195  
`preferred_index` (`sympy.functions.special.tensor_functions` property), 553  
`preferred_index` (`sympy.physics.secondquant.KroneckerDelta` property), 1557  
`Prefix` (class in `sympy.physics.units.prefixes`), 1586  
`prefixes()` (in module `sympy.utilities.iterables`), 2090  
`PreIncrement` (class in `sympy.codegen.cnodes`), 1149  
`prem()` (in module `sympy.polys.polytools`), 2362  
`prem()` (`sympy.polys.polyclasses.DMP` `method`), 2566



```

prem() (sympy.polys.polytools.Poly method), 2410
preorder_traversal() (in module sympy.core.traversal), 1081
presentation() (sympy.combinatorics.perm_groups.PermGroup method), 324
pretty() (in module sympy.printing.pretty.pretty), 2139
pretty_atom() (in module sympy.printing.pretty.pretty_symbology), 2186
pretty_print() (in module sympy.printing.pretty.pretty), 2139
pretty_symbol() (in module sympy.printing.pretty.pretty_symbology), 2186
pretty_try_use_unicode() (in module sympy.printing.pretty.pretty_symbology), 2184
pretty_use_unicode() (in module sympy.printing.pretty.pretty_symbology), 2184
prettyForm (class in sympy.printing.pretty.stringpict), 2188
PrettyPrinter (class in sympy.printing.pretty.pretty), 2139
prev() (sympy.combinatorics.prufer.Prufer method), 336
prev_binary() (sympy.combinatorics.subsets.Subset method), 342
prev_gray() (sympy.combinatorics.subsets.Subset method), 342
prev_lex() (sympy.combinatorics.partitions.IntegerPartition method), 255
prev_lexicographic() (sympy.combinatorics.subsets.Subset method), 343
preview() (in module sympy.printing.preview), 2181
preview_diagram() (in module sympy.categories.diagram_drawing), 2761
prevprime() (in module sympy.ntheory.generate), 1481
prime, 924
prime() (in module sympy.ntheory.generate), 1479
prime_as_sum_of_two_squares() (in module sympy.solvers.diophantine.diophantine), 743
prime_decomp() (in module sympy.polys.numberfields.primes), 2705
prime_valuation() (in module sympy.polys.numberfields.primes), 2709
primefactors() (in module sympy.polys.numberfields.primes), 2706
PrimeIdeal (class in sympy.polys.numberfields.primes), 2706
primenu (class in sympy.ntheory.factor_), 1505
primeomega (class in sympy.ntheory.factor_), 1506
primepi() (in module sympy.ntheory.generate), 1479
PrimePredicate (class in sympy.assumptions.predicates.ntheory), 217
primerange() (in module sympy.ntheory.generate), 1481
primerange() (sympy.ntheory.generate.Sieve method), 1477
primes_above() (sympy.polys.domains.AlgebraicField method), 2545
primitive() (in module sympy.polys.polytools), 2369
primitive() (sympy.core.add.Add method), 1017
primitive() (sympy.core.expr.Expr method), 972
primitive() (sympy.polys.polyclasses.DMP method), 2566
primitive() (sympy.polys.polytools.Poly method), 2411
primitive() (sympy.polys.rings.PolyElement method), 2559
primitive_element() (in module sympy.polys.numberfields.subfield), 2712
primitive_element() (sympy.core.numbers.AlgebraicNumber method), 992
primitive_root() (in module sympy.ntheory.residue_ntheory), 1519
primorial() (in module sympy.ntheory.generate), 1483
principal_branch (class in sympy.functions.elementary.complexes), 389
principal_value() (sympy.integrals.integrals.Integral method), 604

```

[Print \(class in sympy.codegen.ast\)](#), 1135  
[print\\_ccode\(\)](#) (in module [sympy.printing.c](#)), 2143  
[print\\_dim\\_base\(\)](#) ([sympy.physics.units.dimensions.Dimension](#) method), 1586  
[print\\_fcode\(\)](#) (in module [sympy.printing.fortran](#)), 2149  
[print\\_gtk\(\)](#) (in module [sympy.printing.gtk](#)), 2170  
[print\\_latex\(\)](#) (in module [sympy.printing.latex](#)), 2175  
[print\\_maple\\_code\(\)](#) (in module [sympy.printing.maple](#)), 2155  
[print\\_mathml\(\)](#) (in module [sympy.printing.mathml](#)), 2176  
[print\\_node\(\)](#) (in module [sympy.printing.tree](#)), 2179  
[print\\_nonzero\(\)](#) ([sympy.matrices.matrices.MatrixBase](#) method), 1307  
[print\\_rcode\(\)](#) (in module [sympy.printing.rcode](#)), 2146  
[print\\_tree\(\)](#) (in module [sympy.printing.tree](#)), 2179  
[Printer](#) (class in [sympy.printing.printer](#)), 2138  
[printmethod](#) ([sympy.printing.aesarcode.AesarCodePrinter](#) attribute), 2166  
[printmethod](#) ([sympy.printing.c.C89CodePrinter](#) attribute), 2141  
[printmethod](#) ([sympy.printing.c.C99CodePrinter](#) attribute), 2141  
[printmethod](#) ([sympy.printing.codeprinter.CodePrinter](#) attribute), 2183  
[printmethod](#) ([sympy.printing.cxx.CXX11CodePrinter](#) attribute), 2144  
[printmethod](#) ([sympy.printing.cxx.CXX98CodePrinter](#) attribute), 2144  
[printmethod](#) ([sympy.printing.fortran.FCodePrinter](#) attribute), 2149  
[printmethod](#) ([sympy.printing.jscode.JavascriptCodePrinter](#) attribute), 2155  
[printmethod](#) ([sympy.printing.julia.JuliaCodePrinter](#) attribute), 2158  
[printmethod](#) ([sympy.printing.lambdarepr.LambdaPrinter](#) attribute), 2170  
[printmethod](#) ([sympy.printing.latex.LatexPrinter](#) attribute), 2170  
[printmethod](#) ([sympy.printing.maple.MapleCodePrinter](#) attribute), 2154  
[printmethod](#) ([sympy.printing.mathematica.MCCodePrinter](#) attribute), 2154  
[printmethod](#) ([sympy.printing.mathml.MathMLCodePrinter](#) attribute), 2175  
[printmethod](#) ([sympy.printing.mathml.MathMLPresentationPrinter](#) attribute), 2175  
[printmethod](#) ([sympy.printing.octave.OctaveCodePrinter](#) attribute), 2161  
[printmethod](#) ([sympy.printing.pretty.pretty.PrettyPrinter](#) attribute), 2139  
[printmethod](#) ([sympy.printing.printer.Printer](#) attribute), 2138  
[printmethod](#) ([sympy.printing.rcode.RCodePrinter](#) attribute), 2144  
[printmethod](#) ([sympy.printing.repr.ReprPrinter](#) attribute), 2177  
[printmethod](#) ([sympy.printing.rust.RustCodePrinter](#) attribute), 2164  
[printmethod](#) ([sympy.printing.str.StrPrinter](#) attribute), 2178  
[PrismaticJoint](#) (class in [sympy.physics.mechanics.joint](#)), 777  
[prob\(\)](#) ([sympy.physics.quantum.state.Wavefunction](#) method), 1839  
[Probability](#) (class in [sympy.stats](#)), 2960  
[probability\(\)](#) ([sympy.stats.BernoulliProcess](#) method), 2954  
[process\\_series\(\)](#) ([sympy.plotting.plot.MatplotlibBackend](#) method), 2868  
[product\(\)](#) (in module [sympy.core.mul](#)), 1013  
[Product](#) (class in [sympy.concrete.products](#)), 905  
[product\(\)](#) (in module [sympy.concrete.products](#)), 916  
[product\(\)](#) ([sympy.polys.agca.ideals.Ideal](#) method), 2463  
[product\(\)](#) ([sympy.series.formal.FormalPowerSeries](#) method), 650  
[ProductDomain](#) (class in [sympy.stats.rv](#)), 2981  
[ProductPSpace](#) (class in [sympy.stats.rv](#)), 2981  
[ProductSet](#) (class in [sympy.sets.sets](#)), 1199  
[Projector](#) (class in [sympy.codegen.fnodes](#)), 1152  
[project\(\)](#) ([sympy.geometry.point.Point](#) static method), 2207  
[project\(\)](#) ([sympy.matrices.matrices.MatrixBase](#) method), 1308  
[projection\(\)](#) ([sympy.geometry.line.LinearEntity](#) method), 2225  
[projection\(\)](#) ([sympy.geometry.plane.Plane](#) method), 2310  
[projection\(\)](#) ([sympy.vector.vector.Vector](#) method), 1457  
[ProjectiveLine](#) (class in [sympy.geometry.plane.Plane](#) module), 2311

[proper\\_divisor\\_count\(\)](#) (in module [sympy.ntheory.factor\\_](#)), 1498  
[proper\\_divisors\(\)](#) (in module [sympy.ntheory.factor\\_](#)), 1498  
[Prufer](#) (class in [sympy.combinatorics.prufer](#)), 335  
[prufer\\_rank\(\)](#) ([sympy.combinatorics.prufer.Prufer](#) method), 337  
[prufer\\_repr](#) ([sympy.combinatorics.prufer.Prufer](#) property), 337  
[psets](#) ([sympy.sets.fancysets.ComplexRegion](#) property), 1211  
[psi\\_n\(\)](#) (in module [sympy.physics.qho\\_1d](#)), 1540  
[Psi\\_nlm\(\)](#) (in module [sympy.physics.hydrogen](#)), 1535  
[PSpace](#) (class in [sympy.stats.rv](#)), 2980  
[pspace\(\)](#) (in module [sympy.stats.rv](#)), 2982  
[public\(\)](#) (in module [sympy.utilities.decorator](#)), 2059  
[PurePoly](#) (class in [sympy.polys.polytools](#)), 2423  
[PxBra](#) (class in [sympy.physics.quantum.cartesian](#)), 1801  
[PxKet](#) (class in [sympy.physics.quantum.cartesian](#)), 1802  
[PxOp](#) (class in [sympy.physics.quantum.cartesian](#)), 1802  
[pycode\(\)](#) (in module [sympy.printing.pycode](#)), 2176  
[PyTestReporter](#) (class in [sympy.testing.runtests](#)), 2030  
[Python Enhancement Proposals](#) PEP 335, 3045, 3046  
[PythonFiniteField](#) (class in [sympy.polys.domains](#)), 2525  
[PythonIntegerRing](#) (class in [sympy.polys.domains](#)), 2528  
[PythonMPQ](#) (class in [sympy.external.pythonmpq](#)), 2532  
[PythonRationalField](#) (class in [sympy.polys.domains](#)), 2531  
**Q**  
[q](#) ([sympy.physics.mechanics.jointsmethod.Jointsmethod](#) property), 1783  
[q](#) ([sympy.physics.optics.gaussopt.BeamParameter](#) property), 1859  
[qapply\(\)](#) (in module [sympy.physics.quantum.qapply](#)), 1814  
[QFT](#) (class in [sympy.physics.quantum.qft](#)), 1850  
[QQ\\_col](#) ([sympy.polys.numberfields.modules.ModuleElement](#) property), 2730  
[QQ\\_matrix](#) ([sympy.polys.numberfields.modules.Submodule](#) property), 2726  
[QRdecomposition\(\)](#) ([sympy.matrices.matrices.MatrixBase](#) method), 1287  
[QRsolve\(\)](#) ([sympy.matrices.matrices.MatrixBase](#) method), 1291  
[qs\(\)](#) (in module [sympy.ntheory.qs](#)), 1532  
[quadrant\(\)](#) ([sympy.polys.domains.gaussiandomains.Gaussian](#) method), 2534  
[quadratic\\_residues\(\)](#) (in module [sympy.ntheory.residue\\_ntheory](#)), 1520  
[QuadraticU\(\)](#) (in module [sympy.stats](#)), 2921  
[quantile\(\)](#) (in module [sympy.stats](#)), 2970  
[Quantity](#) (class in [sympy.physics.units.quantities](#)), 1587  
[quarter\\_wave\\_retarder\(\)](#) (in module [sympy.physics.optics.polarization](#)), 1877  
[Quaternion](#) (class in [sympy.algebras](#)), 885  
[QuaternionOrienter](#) (class in [sympy.vector.orienters](#)), 1469  
[Qubit](#) (class in [sympy.physics.quantum.qubit](#)), 1852  
[qubit\\_to\\_matrix\(\)](#) (in module [sympy.physics.quantum.qubit](#)), 1855  
[QubitBra](#) (class in [sympy.physics.quantum.qubit](#)), 1853  
[quo\(\)](#) (in module [sympy.polys.polytools](#)), 2364  
[quo\(\)](#) ([sympy.polys.domains.domain.Domain](#) method), 2516  
[quo\(\)](#) ([sympy.polys.domains.field.Field](#) method), 2521  
[quo\(\)](#) ([sympy.polys.domains.GMPYRationalField](#) method), 2532  
[quo\(\)](#) ([sympy.polys.domains.RationalField](#) method), 2531  
[quo\(\)](#) ([sympy.polys.domains.ring.Ring](#) method), 2521  
[quo\(\)](#) ([sympy.polys.polyclasses.DMF](#) method), 2568  
[quo\(\)](#) ([sympy.polys.polyclasses.DMP](#) method), 2566  
[quo\(\)](#) ([sympy.polys.polytools.Poly](#) method), 2411  
[quo\\_ground\(\)](#) ([sympy.polys.polyclasses.DMP](#) method), 2566  
[quo\\_ground\(\)](#) ([sympy.polys.polytools.Poly](#) method), 2411

QuotedString (class in `sympy.codegen.ast`), 1136  
 quotient() (`sympy.polys.agca.ideals.Ideal` method), 2463  
 quotient\_codomain() (`sympy.polys.agca.homomorphisms.ModuleHomomorphism` method), 2473  
 quotient\_domain() (`sympy.polys.agca.homomorphisms.ModuleHomomorphism` method), 2473  
 quotient\_hom() (`sympy.polys.agca.modules.QuotientModule` method), 2466  
 quotient\_hom() (`sympy.polys.agca.modules.SubQuotientModule` method), 2467  
 quotient\_module() (`sympy.polys.agca.modules.FreeModule` method), 2456  
 quotient\_module() (`sympy.polys.agca.modules.Module` method), 2453  
 quotient\_module() (`sympy.polys.agca.modules.SubModule` method), 2459  
 quotient\_ring() (`sympy.polys.domains.ring.Ring` method), 2522  
 QuotientModule (class in `sympy.polys.agca.modules`), 2465  
 QuotientModuleElement (class in `sympy.polys.agca.modules`), 2467  
 QuotientRing (class in `sympy.polys.domains.quotientring`), 2551  
 R  
 R\_nl() (in module `sympy.physics.hydrogen`), 1536  
 R\_nl() (in module `sympy.physics.sho`), 1541  
 racah() (in module `sympy.physics.wigner`), 1570  
 rad\_rationalize() (in module `sympy.simplify.radsimp`), 672  
 Rademacher() (in module `sympy.stats`), 2880  
 radical() (`sympy.polys.agca.ideals.Ideal` method), 2464  
 radius (`sympy.geometry.ellipse.Circle` property), 2271  
 radius (`sympy.geometry.polygon.RegularPolygon` property), 2291  
 radius (`sympy.physics.optics.gaussopt.BeamParameter` property), 1859  
 radius\_of\_convergence (`sympy.functions.special.hyper.hyper` property), 522  
 radsimp() (in module `sympy.simplify.radsimp`), 670  
 radsimp() (`sympy.core.expr.Expr` method), 972  
 raise\_on\_deprecated() (in module `sympy.testing.runtests`), 2032  
 RaisedCosine() (in module `sympy.stats`), 2922  
 raise\_test() (in module `sympy.testing.pytest`), 2027  
 raise\_test() (in module `sympy.testing.pytest`), 2027  
 random\_numerical\_data() (`sympy.physics.control.control_plots` method), 1947  
 ramp\_response\_plot() (`sympy.physics.control.control_plots` method), 1946  
 randMatrix() (in module `sympy.matrices.dense`), 1322  
 random() (`sympy.combinatorics.perm_groups.PermutationGroup` method), 324  
 random() (`sympy.combinatorics.permutations.Permutation` class method), 278  
 random\_bitstring() (`sympy.combinatorics.graycode` method), 350  
 random\_circuit() (in module `sympy.physics.quantum.gate`), 1847  
 random\_complex\_number() (in module `sympy.core.random`), 1078  
 random\_derangement() (in module `sympy.utilities.iterables`), 2090  
 random\_integer\_partition() (in module `sympy.combinatorics.partitions`), 255  
 random\_point() (`sympy.geometry.ellipse.Ellipse` method), 2264  
 random\_point() (`sympy.geometry.line.LinearEntity` method), 2226  
 random\_point() (`sympy.geometry.plane.Plane` method), 2311  
 random\_poly() (in module `sympy.polys.specialpolys`), 2439  
 random\_pr() (`sympy.combinatorics.perm_groups.PermutationGroup` method), 324  
 random\_stab() (`sympy.combinatorics.perm_groups.PermutationGroup` method), 324  
 random\_symbols() (in module `sympy.stats.rv`), 2982  
 RandomDomain (class in `sympy.stats.rv`), 2980  
 RandomSymbol (class in `sympy.stats.rv`), 2981  
 randprime() (in module `sympy.ntheory.generate`), 1483  
 Range (class in `sympy.sets.fancysets`), 1207  
 ranges (`sympy.tensor.indexed.Indexed` property), 1403



[rank \(sympy.combinatorics.graycode.GrayCode property\), 349](#)  
[rank \(sympy.combinatorics.partitions.Partition property\), 253](#)  
[rank \(sympy.combinatorics.prufer.Prufer property\), 337](#)  
[rank \(sympy.tensor.indexed.Indexed property\), 1403](#)  
[rank\(\) \(sympy.combinatorics.permutations.Permutation method\), 279](#)  
[rank\(\) \(sympy.liealgebras.cartan\\_type.StandardCartanType method\), 2340](#)  
[rank\(\) \(sympy.matrices.matrices.MatrixReductionTool method\), 1233](#)  
[rank\\_binary \(sympy.combinatorics.subsets.Subset property\), 343](#)  
[rank\\_decomposition\(\) \(sympy.matrices.matrices.MatrixBase method\), 1308](#)  
[rank\\_gray \(sympy.combinatorics.subsets.Subset property\), 343](#)  
[rank\\_lexicographic \(sympy.combinatorics.subsets.Subset property\), 344](#)  
[rank\\_nonlex\(\) \(sympy.combinatorics.permutations.Permutation method\), 279](#)  
[rank\\_trotterjohnson\(\) \(sympy.combinatorics.permutations.Permutation method\), 279](#)  
[rat\\_clear\\_denoms\(\) \(sympy.polys.polytools.Poly method\), 2411](#)  
[ratint\(\) \(in module sympy.integrals.rationaltools\), 587](#)  
[ratint\\_logpart\(\) \(in module sympy.integrals.rationaltools\), 588](#)  
[ratint\\_ratpart\(\) \(in module sympy.integrals.rationaltools\), 588](#)  
[rational, 924](#)  
[Rational \(class in sympy.core.numbers\), 985](#)  
[rational\\_algorithm\(\) \(in module sympy.series.formal\), 654](#)  
[rational\\_independent\(\) \(in module sympy.series.formal\), 654](#)  
[rational\\_laurent\\_series\(\) \(in module sympy.solvers.ode.riccati\), 804](#)  
[rational\\_parametrization\(\) \(sympy.vector.implicitregion.ImplicitRegion method\), 1463](#)  
[RationalField \(class in sympy.polys.domains\), 2530](#)  
[rationalize\(\) \(in module sympy.parsing.sympy\\_parser\), 2128](#)  
[RationalPredicate \(class in sympy.assumptions.predicates.sets\), 225](#)  
[RationalRiccati \(class in sympy.solvers.ode.single\), 777](#)  
[Rationals \(class in sympy.sets.fancysets\), 1203](#)  
[ratsimp\(\) \(in module sympy.simplify.ratsimp\), 678](#)  
[ratsimp\(\) \(sympy.core.expr.Expr method\), 972](#)  
[ratsimpmodprime\(\) \(in module sympy.simplify.ratsimp\), 678](#)  
[raw\\_lines\(\) \(in module sympy.utilities.misc\), 2112](#)  
[Ray \(class in sympy.geometry.line\), 2230](#)  
[Ray2D \(class in sympy.geometry.line\), 2240](#)  
[Ray3D \(class in sympy.geometry.line\), 2244](#)  
[Rayleigh\(\) \(in module sympy.stats\), 2923](#)  
[rayleigh2waist\(\) \(in module sympy.physics.optics.gaussopt\), 1869](#)  
[RayTransferMatrix \(class in sympy.physics.optics.gaussopt\), 1863](#)  
[real \(class in sympy.functions.elementary.complexes\), 382](#)  
[real\\_root\(\) \(in module sympy.functions.elementary.miscellaneous\), 426](#)  
[real\\_roots\(\) \(in module sympy.polys.polytools\), 2375](#)  
[real\\_roots\(\) \(sympy.polys.polytools.Poly method\), 2412](#)  
[real\\_roots\(\) \(sympy.polys.rootoftools.ComplexRootOf class method\), 2436](#)  
[RealElement \(class in sympy.polys.domains.mpelements\), 2546](#)  
[RealElementsPredicate \(class in sympy.assumptions.predicates.matrices\), 212](#)

[RealField \(class in sympy.polys.domains\), 2545](#)  
[RealNumber \(in module sympy.core.numbers\), 993](#)  
[RealPredicate \(class in sympy.assumptions.predicates.sets\), 226](#)  
[Reals \(class in sympy.sets.fancysets\), 1205](#)  
[Reciprocal\(\) \(in module sympy.stats\), 2923](#)  
[reconstruct\(\) \(in module sympy.solvers.diophantine.diophantine\), 744](#)  
[recurrence \(sympy.series.sequences.RecursiveSeq property\), 640](#)  
[recurrence\\_memo\(\) \(in module sympy.utilities.memoization\), 2110](#)  
[RecursiveSeq \(class in sympy.series.sequences\), 638](#)  
[red\\_groebner\(\) \(in module sympy.polys.groebnertools\), 2640](#)  
[reduce\(\) \(sympy.polys.polytools.GroebnerBasis method\), 2424](#)  
[reduce\(\) \(sympy.series.sequences.SeqAdd static method\), 637](#)  
[reduce\(\) \(sympy.series.sequences.SeqMul static method\), 638](#)  
[reduce\(\) \(sympy.sets.sets.Complement static method\), 1201](#)  
[reduce\\_abs\\_inequalities\(\) \(in module sympy.solvers.inequalities\), 753](#)  
[reduce\\_abs\\_inequality\(\) \(in module sympy.solvers.inequalities\), 752](#)  
[reduce\\_alg\\_num\(\) \(sympy.polys.numberfields.primes.PrimRing method\), 2707](#)  
[reduce\\_ANP\(\) \(sympy.polys.numberfields.primes.PrimRing method\), 2707](#)  
[reduce\\_element\(\) \(sympy.polys.agca.ideals.Ideal method\), 2464](#)  
[reduce\\_element\(\) \(sympy.polys.agca.modules.SubModule method\), 2460](#)  
[reduce\\_element\(\) \(sympy.polys.numberfields.modules.Submodule method\), 2728](#)  
[reduce\\_element\(\) \(sympy.polys.numberfields.primes.PrimRing method\), 2708](#)  
[reduce\\_inequalities\(\) \(in module sympy.solvers.inequalities\), 753](#)  
[reduce\\_rational\\_inequalities\(\) \(in module sympy.solvers.inequalities\), 752](#)  
[reduced\(\) \(in module sympy.polys.polytools\), 2376](#)  
[reduced\(\) \(sympy.polys.numberfields.modules.Module method\), 2731](#)  
[reduced\(\) \(sympy.polys.numberfields.modules.Submodule class method\), 2729](#)  
[reduced\\_mod\\_p\(\) \(sympy.polys.numberfields.modules.Module method\), 2731](#)  
[reduced\\_totient \(class in sympy.ntheory.factor\\_\), 1501](#)  
[ReferenceFrame \(class in sympy.physics.vector.frame\), 1629](#)  
[refine\(\) \(in module sympy.assumptions.refine\), 197](#)  
[refine\(\) \(sympy.core.basic.Basic method\), 937](#)  
[refine\(\) \(sympy.matrices.common.MatrixCommon method\), 1350](#)  
[refine\\_abs\(\) \(in module sympy.assumptions.refine\), 198](#)  
[refine\\_arg\(\) \(in module sympy.assumptions.refine\), 198](#)  
[refine\\_atan2\(\) \(in module sympy.assumptions.refine\), 199](#)  
[refine\\_im\(\) \(in module sympy.assumptions.refine\), 199](#)  
[refine\\_matrixelement\(\) \(in module sympy.assumptions.refine\), 199](#)  
[refine\\_Pow\(\) \(in module sympy.assumptions.refine\), 198](#)  
[refine\\_re\(\) \(in module sympy.assumptions.refine\), 200](#)  
[refine\\_root\(\) \(in module sympy.polys.polytools\), 2374](#)  
[refine\\_root\(\) \(sympy.polys.polyclasses.DMP method\), 2566](#)  
[refine\\_root\(\) \(sympy.polys.polytools.Poly method\), 2412](#)  
[refine\\_single\(\) \(in module sympy.assumptions.refine\), 200](#)  
[RefinementFailed \(class in sympy.polys.polyerrors\), 2643](#)  
[reflect\(\) \(sympy.geometry.ellipse.Circle method\), 2272](#)  
[reflect\(\) \(sympy.geometry.ellipse.Ellipse method\), 2265](#)  
[reflect\(\) \(sympy.geometry.entity.GeometryEntity method\), 2195](#)  
[reflect\(\) \(sympy.geometry.polygon.RegularPolygon method\), 2291](#)  
[reflective\\_filter\(\) \(in module sympy.physics.optics.polarization\), 1878](#)  
[refraction\\_angle\(\) \(in module sympy.physics.optics.utils\), 1885](#)  
[refractive\\_index \(sympy.physics.optics.medium.Medium property\), 1871](#)  
[register\(\) \(sympy.assumptions.assume.Predicate class method\), 196](#)  
[register\\_handler\(\) \(in module](#)

`sympy.assumptions.ask`), 193  
`register_many()` (`sympy.assumptions.assume.Predicate` method), 2413  
`class method`), 196  
`regular_point()` (`sympy.vector.implicitregion.ImplicitRegion` method), 1464  
`RegularPolygon` (class in `sympy.geometry.polygon`), 2284  
`Rel` (in module `sympy.core.relational`), 1022  
`Relational` (class in `sympy.core.relational`), 1018  
`rem()` (in module `sympy.polys.polytools`), 2363  
`rem()` (`sympy.polys.domains.domain.Domain` method), 2517  
`rem()` (`sympy.polys.domains.field.Field` method), 2521  
`rem()` (`sympy.polys.domains.GMPYRationalField` method), 2532  
`rem()` (`sympy.polys.domains.RationalField` method), 2531  
`rem()` (`sympy.polys.domains.ring.Ring` method), 2522  
`rem()` (`sympy.polys.polyclasses.DMP` method), 2566  
`rem()` (`sympy.polys.polytools.Poly` method), 2412  
`remove_handler()` (in module `sympy.assumptions.ask`), 193  
`remove_load()` (`sympy.physics.continuum_mechanics.beam.Beam` method), 1975  
`remove_load()` (`sympy.physics.continuum_mechanics.mesh.Mesh` method), 2025  
`remove_load()` (`sympy.physics.mechanics.body.Body` method), 1763  
`remove_member()` (`sympy.physics.continuum_mechanics.mesh.Mesh` method), 2026  
`remove_node()` (`sympy.physics.continuum_mechanics.mesh.Mesh` method), 2026  
`remove_redundant_sols()` (in module `sympy.solvers.ode.riccati`), 805  
`remove_support()` (`sympy.physics.continuum_mechanics.mesh.Mesh` method), 2026  
`remove0()` (`sympy.core.expr.Expr` method), 972  
`render()` (`sympy.printing.pretty.stringpict.stringpic` method), 2187  
`render_as_module()` (in module `sympy.codegen.futils`), 1159  
`render_as_module()` (in module `sympy.codegen.pyutils`), 1159  
`render_as_source_file()` (in module `sympy.codegen.cutils`), 1159  
`reorder()` (`sympy.concrete.expr_with_intlimits.ExprWithIntLimits` method), 914  
`reorder()` (`sympy.polys.polytools.Poly` method), 2413  
`reorder_limit()` (`sympy.concrete.expr_with_intlimits.ExprWithIntLimits` method), 914  
`rep_expectation()` (in module `sympy.physics.quantum.represent`), 1817  
`rep_innerproduct()` (in module `sympy.physics.quantum.represent`), 1817  
`repeated_decimals()` (in module `sympy.parsing.sympy_parser`), 2129  
`replace()` (in module `sympy.utilities.misc`), 2113  
`replace()` (`sympy.core.basic.Basic` method), 937  
`replace()` (`sympy.matrices.common.MatrixCommon` method), 1350  
`replace()` (`sympy.polys.polytools.Poly` method), 2413  
`replace_in_Add()` (`sympy.codegen.rewriting.FuncMinusOn` method), 1119  
`replace_with_arrays()` (`sympy.tensor.tensor.TensExpr` method), 1414  
`ReplaceOptim` (class in `sympy.codegen.rewriting`), 1119  
`Reporter` (class in `sympy.testing.runtests`), 2070  
`repr()` (`sympy.polys.numberfields.primes.PrimeIdeal` method), 2708  
`represent()` (in module `sympy.physics.quantum.represent`), 1818  
`represent()` (`sympy.polys.numberfields.modules.Endomorphism` method), 2735  
`represent()` (`sympy.polys.numberfields.modules.Module` method), 2722  
`represent()` (`sympy.polys.numberfields.modules.PowerBase` method), 2725  
`represent()` (`sympy.polys.numberfields.modules.Submodule` method), 2729  
`reprify()` (`sympy.printing.repr.ReprPrinter` method), 2177  
`ReprPrinter` (class in `sympy.printing.repr`), 2177  
`reset()` (`sympy.combinatorics.polyhedron.Polyhedron` method), 333  
`reshape()` (in module `sympy.codegen.fnodes`), 1155  
`reshape()` (in module `sympy.utilities.iterables`), 2091  
`reshape()` (`sympy.matrices.common.MatrixCommon` method), 1351

`residue()` (in module `sympy.series.residues`), 631  
`resize()` (`sympy.combinatorics.permutations.P66` generalized() (in module `sympy.combinatorics.permutations`), 280  
`restrict_codomain()` (`sympy.polys.agca.homomorphisms.ModuleHomomorphism` (in module `sympy.combinatorics.permutations`), 2474  
`restrict_domain()` (`sympy.polys.agca.homomorphisms.ModuleHomomorphism` (in module `sympy.combinatorics.permutations`), 2474  
`Result` (class in `sympy.utilities.codegen`), 2052  
`result_variables` (`sympy.utilities.codegen.Routine` property), 2052  
`resultant()` (in module `sympy.polys.polytools`), 2366  
`resultant()` (`sympy.polys.polyclasses.DMP` method), 2566  
`resultant()` (`sympy.polys.polytools.Poly` method), 2413  
`retract()` (`sympy.polys.polytools.Poly` method), 2414  
`Return` (class in `sympy.codegen.ast`), 1136  
`return_expr()` (`sympy.parsing.sym_expr.SymPyFixpointExpression` method), 2133  
`reverse_order()` (`sympy.concrete.products.Product` method), 909  
`reverse_order()` (`sympy.concrete.summations.Sum` method), 904  
`reversed` (`sympy.core.relational.Relational` property), 1021  
`reversed` (`sympy.sets.fancysets.Range` property), 1208  
`ReversedGradedLexOrder` (class in `sympy.polys.orderings`), 2431  
`reversedsign` (`sympy.core.relational.Relational` property), 1021  
`revert()` (`sympy.polys.domains.domain.Domain` method), 2517  
`revert()` (`sympy.polys.domains.field.Field` method), 2521  
`revert()` (`sympy.polys.domains.ring.Ring` method), 2522  
`revert()` (`sympy.polys.polyclasses.DMP` method), 2566  
`revert()` (`sympy.polys.polytools.Poly` method), 2414  
`rewrite()` (in module `sympy.series.gruntz`), 623  
`rewrite()` (`sympy.core.basic.Basic` method), 940  
`RGS` (`sympy.combinatorics.partitions.Partition` property), 251  
`RGS_enum()` (in module `sympy.combinatorics.partitions`), 256  
`ResGeneralized()` (in module `sympy.combinatorics.partitions`), 255  
`RGS_Homomorphism` (in module `sympy.combinatorics.partitions`), 257  
`RGS_Homomorphism` (in module `sympy.combinatorics.partitions`), 256  
`rhs` (`sympy.core.relational.Relational` property), 1021  
`rhs()` (`sympy.physics.mechanics.jointsmethod.JointsMethod` method), 1783  
`rhs()` (`sympy.physics.mechanics.kane.KanesMethod` method), 1766  
`rhs()` (`sympy.physics.mechanics.lagrange.LagrangesMethod` method), 1769  
`riccati_inverse_normal()` (in module `sympy.solvers.ode.riccati`), 803  
`riccati_normal()` (in module `sympy.solvers.ode.riccati`), 803  
`riccati_special_reduced()` (in module `sympy.solvers.ode.riccati`), 803  
`RiccatiSpecial` (class in `sympy.solvers.ode.single`), 781  
`Richardson()` (in module `sympy.series.acceleration`), 630  
`riemann()` (`sympy.tensor.tensor.TensorSymmetry` class method), 1421  
`riemann_cyclic()` (in module `sympy.tensor.tensor`), 1420  
`riemann_cyclic_replace()` (in module `sympy.tensor.tensor`), 1419  
`right()` (`sympy.printing.pretty.stringpict.stringPict` method), 2187  
`right_open` (`sympy.sets.sets.Interval` property), 1196  
`RigidBody` (class in `sympy.physics.mechanics.rigidbody`), 1746  
`Ring` (class in `sympy.polys.domains.ring`), 2521  
`ring()` (in module `sympy.polys.rings`), 2551  
`risch_integrate()` (in module `sympy.integrals.risch`), 592  
`RisingFactorial` (class in `sympy.functions.combinatorial.factorials`), 443  
`Rk` (in module `sympy.physics.quantum.qft`), 1850  
`RkGate` (class in `sympy.physics.quantum.qft`), 1850



`rmul()` (`sympy.combinatorics.permutations.Permutation` static method), 280  
`rmul_with_af()` (`sympy.combinatorics.permutations.Permutation` class method), 281  
`rmultiply()` (`sympy.matrices.common.MatrixCommon` method), 1351  
`root` (in module `sympy.printing.pretty.pretty_symbolology`), 2185  
`root()` (in module `sympy.functions.elementary.miscellaneous`), 422  
`root()` (`sympy.polys.polytools.Poly` method), 2415  
`root()` (`sympy.printing.pretty.stringpict.stringPict` method), 2187  
`root_space()` (`sympy.liealgebras.root_system.RootSystem` method), 2328  
`RootOf` (class in `sympy.polys.rootoftools`), 2431  
`rootof()` (in module `sympy.polys.rootoftools`), 2431  
`roots()` (in module `sympy.polys.polyroots`), 2436  
`roots()` (`sympy.liealgebras.type_a.TypeA` method), 2329  
`roots()` (`sympy.liealgebras.type_b.TypeB` method), 2331  
`roots()` (`sympy.liealgebras.type_c.TypeC` method), 2332  
`roots()` (`sympy.liealgebras.type_d.TypeD` method), 2334  
`roots()` (`sympy.liealgebras.type_e.TypeE` method), 2335  
`roots()` (`sympy.liealgebras.type_f.TypeF` method), 2337  
`roots()` (`sympy.liealgebras.type_g.TypeG` method), 2338  
`RootSum` (class in `sympy.polys.rootoftools`), 2436  
`RootSystem` (class in `sympy.liealgebras.root_system`), 2326  
`Ropen()` (`sympy.sets.sets.Interval` class method), 1195  
`rot90()` (`sympy.matrices.common.MatrixCommon` method), 1351  
`rot_axis1()` (in module `sympy.matrices.dense`), 1324  
`rot_axis2()` (in module `sympy.matrices.dense`), 1325  
`rot_axis3()` (in module `sympy.matrices.dense`), 1326  
`rotate()` (`sympy.combinatorics.polyhedron.Polyhedron` method), 334  
`rotation` (`sympy.geometry.curve.Curve` method), 2251  
`rotation` (`sympy.geometry.ellipse.Ellipse` method), 2266  
`rotation` (`sympy.geometry.entity.GeometryEntity` method), 2196  
`rotate()` (`sympy.geometry.point.Point2D` method), 2209  
`rotate()` (`sympy.geometry.polygon.RegularPolygon` method), 2291  
`rotate_left()` (in module `sympy.utilities.iterables`), 2092  
`rotate_point()` (`sympy.algebras.Quaternion` static method), 895  
`rotate_right()` (in module `sympy.utilities.iterables`), 2092  
`Rotation` (class in `sympy.physics.quantum.spin`), 1825  
`rotation` (`sympy.geometry.polygon.RegularPolygon` property), 2292  
`rotation_matrix()` (`sympy.vector.coordsysrect.CoordSys3D` method), 1454  
`rotation_matrix()` (`sympy.vector.orienters.AxisOrienter` method), 1467  
`rotation_matrix()` (`sympy.vector.orienters.Orienter` method), 1466  
`rotations()` (in module `sympy.utilities.iterables`), 2092  
`round()` (`sympy.core.expr.Expr` method), 973  
`round_two()` (in module `sympy.polys.numberfields.basis`), 2703  
`RoundFunction` (class in `sympy.functions.elementary.integers`), 410  
`roundrobin()` (in module `sympy.utilities.iterables`), 2092  
`Routine` (class in `sympy.utilities.codegen`), 2052  
`routine()` (`sympy.utilities.codegen.CodeGen` method), 2049  
`routine()` (`sympy.utilities.codegen.JuliaCodeGen` method), 2051  
`routine()` (`sympy.utilities.codegen.OctaveCodeGen` method), 2052  
`routine()` (`sympy.utilities.codegen.RustCodeGen` method), 2053  
`row()` (`sympy.matrices.common.MatrixCommon` method), 1352  
`row_def()` (`sympy.matrices.common.MatrixCommon` method), 1352

`row_insert()` (`sympy.matrices.common.MatrixCommon`) (in module `sympy.matrices.common`), 1352  
`row_join()` (`sympy.matrices.common.MatrixCommon`) (in module `sympy.matrices.common`), 1353  
`rowspace()` (`sympy.matrices.matrices.MatrixSubspaces`) (in module `sympy.matrices.matrices`), 1237  
`rowspace()` (`sympy.polys.matrices.domainmatrix.DomainMatrix`) (in module `sympy.polys.matrices`), 2683  
`rref()` (`sympy.matrices.matrices.MatrixReductions`) (in module `sympy.matrices.matrices`), 1234  
`rref()` (`sympy.polys.matrices.ddm.DDM`) (in module `sympy.polys.matrices.ddm`), 2691  
`rref()` (`sympy.polys.matrices.domainmatrix.DomainMatrix`) (in module `sympy.polys.matrices`), 2684  
`rref()` (`sympy.polys.matrices.sdm.SDM`) (in module `sympy.polys.matrices.sdm`), 2698  
`rs_asin()` (in module `sympy.polys.ring_series`), 2655  
`rs_atan()` (in module `sympy.polys.ring_series`), 2655  
`rs_atanh()` (in module `sympy.polys.ring_series`), 2657  
`rs_compose_add()` (in module `sympy.polys.ring_series`), 2663  
`rs_cos()` (in module `sympy.polys.ring_series`), 2657  
`rs_cos_sin()` (in module `sympy.polys.ring_series`), 2657  
`rs_cosh()` (in module `sympy.polys.ring_series`), 2658  
`rs_cot()` (in module `sympy.polys.ring_series`), 2656  
`rs_diff()` (in module `sympy.polys.ring_series`), 2662  
`rs_exp()` (in module `sympy.polys.ring_series`), 2654  
`rs_fun()` (in module `sympy.polys.ring_series`), 2665  
`rs_hadamard_exp()` (in module `sympy.polys.ring_series`), 2658  
`rs_integrate()` (in module `sympy.polys.ring_series`), 2663  
`rs_is_puiseux()` (in module `sympy.polys.ring_series`), 2664  
`rs_LambertW()` (in module `sympy.polys.ring_series`), 2654  
`rs_log()` (in module `sympy.polys.ring_series`), 2654  
`rs_mul()` (in module `sympy.polys.ring_series`), 2659  
`rs_newton()` (in module `sympy.polys.ring_series`), 2663  
`rs_nth_root()` (in module `sympy.polys.ring_series`), 2661  
`rs_puiseux()` (in module `sympy.polys.ring_series`), 2659  
`rs_puiseux2()` (in module `sympy.polys.ring_series`), 2664  
`rs_series_inversion()` (in module `sympy.polys.ring_series`), 2660  
`rs_series_reversion()` (in module `sympy.polys.ring_series`), 2660  
`rs_sinh()` (in module `sympy.polys.ring_series`), 2657  
`rs_square()` (in module `sympy.polys.ring_series`), 2659  
`rs_subs()` (in module `sympy.polys.ring_series`), 2662  
`rs_swap()` (in module `sympy.stats.rv`), 2982  
`rs_tan()` (in module `sympy.polys.ring_series`), 2655  
`rs_tanh()` (in module `sympy.polys.ring_series`), 2658  
`rs_trunc()` (in module `sympy.polys.ring_series`), 2661  
`rsa_private_key()` (in module `sympy.crypto.crypto`), 2782  
`rsa_public_key()` (in module `sympy.crypto.crypto`), 2780  
`rsolve()` (in module `sympy.solvers.recurr`), 853  
`rsolve_hyper()` (in module `sympy.solvers.recurr`), 855  
`rsolve_hypergeometric()` (in module `sympy.series.formal`), 657  
`rsolve_poly()` (in module `sympy.solvers.recurr`), 853  
`rsolve_ratio()` (in module `sympy.solvers.recurr`), 854  
`run()` (`sympy.testing.runtests.SymPyDocTestRunner`) (in module `sympy.testing.runtests`), 2031  
`run_all_tests()` (in module `sympy.testing.runtests`), 2032  
`run_in_subprocess_with_hash_randomization()` (in module `sympy.testing.runtests`), 2033  
`runs()` (in module `sympy.utilities.iterables`), 2092  
`runs()` (`sympy.combinatorics.permutations.Permutation`) (in module `sympy.combinatorics.permutations`), 281  
`rust_code()` (in module `sympy.printing.rust`), 2164

- RustCodeGen (class `sympy.utilities.codegen`), 2052
- RustCodePrinter (class `sympy.printing.rust`), 2164
- ## S
- S (in module `sympy.physics.quantum.gate`), 1845
- same\_root() (`sympy.polys.polytools.Poly` method), 2415
- sample() (in module `sympy.stats`), 2971
- sample() (`sympy.stats.DiscreteMarkovChain` method), 2950
- sample\_iter() (in module `sympy.stats`), 2972
- sampling\_density() (in module `sympy.stats.rv`), 2976
- sampling\_E() (in module `sympy.stats.rv`), 2976
- sampling\_P() (in module `sympy.stats.rv`), 2976
- satisfiable() (in module `sympy.logic.inference`), 1184
- saturate() (`sympy.polys.agca.ideals.Ideal` method), 2464
- scalar\_map() (`sympy.vector.coordsysrect.CoordSys3D` method), 1455
- scalar\_part() (`sympy.algebras.Quaternion` method), 895
- scalar\_potential() (in module `sympy.physics.vector.fieldfunctions`), 1672
- scalar\_potential() (in module `sympy.vector`), 1474
- scalar\_potential\_difference() (in module `sympy.physics.vector.fieldfunctions`), 1673
- scalar\_potential\_difference() (in module `sympy.vector`), 1474
- scale() (`sympy.geometry.curve.Curve` method), 2252
- scale() (`sympy.geometry.ellipse.Circle` method), 2272
- scale() (`sympy.geometry.ellipse.Ellipse` method), 2266
- scale() (`sympy.geometry.entity.GeometryEntity` method), 2196
- scale() (`sympy.geometry.point.Point2D` method), 2210
- scale() (`sympy.geometry.point.Point3D` method), 2214
- scale() (`sympy.geometry.polygon.RegularPolygon` method), 2292
- scale() (`sympy.series.fourier.FourierSeries` method), 640
- scale\_factor (`sympy.physics.units.quantities.Quantity` property), 1588
- scalex() (`sympy.series.fourier.FourierSeries` method), 640
- scc() (`sympy.polys.matrices.ddm.DDM` method), 2691
- scc() (`sympy.polys.matrices.domainmatrix.DomainMatrix` method), 2684
- scc() (`sympy.polys.matrices.sdm.SDM` method), 2698
- schreier\_sims() (`sympy.combinatorics.perm_groups.PermGroup` method), 324
- schreier\_sims\_incremental() (`sympy.combinatorics.perm_groups.PermutationGroup` method), 325
- schreier\_sims\_random() (`sympy.combinatorics.perm_groups.PermutationGroup` method), 326
- schreier\_vector() (`sympy.combinatorics.perm_groups.PermutationGroup` method), 327
- schur() (`sympy.matrices.expressions.blockmatrix.BlockMatrix` method), 1383
- Scope (class in `sympy.codegen.ast`), 1136
- SDM (class in `sympy.polys.matrices.sdm`), 2692
- sdm\_add() (in module `sympy.polys.distributedmodules`), 2622
- sdm\_deg() (in module `sympy.polys.distributedmodules`), 2624
- sdm\_ecart() (in module `sympy.polys.distributedmodules`), 2641
- sdm\_from\_dict() (in module `sympy.polys.distributedmodules`), 2622
- sdm\_from\_vector() (in module `sympy.polys.distributedmodules`), 2624
- sdm\_groebner() (in module `sympy.polys.distributedmodules`), 2641
- sdm\_LC() (in module `sympy.polys.distributedmodules`), 2622
- sdm\_LM() (in module `sympy.polys.distributedmodules`), 2623
- sdm\_LT() (in module `sympy.polys.distributedmodules`), 2623
- sdm\_monomial\_deg() (in module `sympy.polys.distributedmodules`), 2623

- 2621  
 sdm\_monomial\_divides() (in module  
 sympy.polys.distributedmodules),  
 2621  
 sdm\_monomial\_mul() (in module  
 sympy.polys.distributedmodules),  
 2620  
 sdm\_mul\_term() (in module  
 sympy.polys.distributedmodules),  
 2623  
 sdm\_nf\_mora() (in module  
 sympy.polys.distributedmodules),  
 2641  
 sdm\_spoly() (in module  
 sympy.polys.distributedmodules),  
 2640  
 sdm\_to\_dict() (in module  
 sympy.polys.distributedmodules),  
 2622  
 sdm\_to\_vector() (in module  
 sympy.polys.distributedmodules),  
 2624  
 sdm\_zero() (in module  
 sympy.polys.distributedmodules),  
 2624  
 search() (sympy.ntheory.generate.Sieve  
 method), 1478  
 search\_function(sympy.physics.quantum.grover.OracleGrover), 1848  
 sec (class in sympy.functions.elementary.trigonometric), 393  
 sech (class in sympy.functions.elementary.hyperbolic), 404  
 second\_moment(sympy.physics.continuum\_mechanics.Beam3D), 1975  
 second\_moment(sympy.physics.continuum\_mechanics.Beam3D), 1997  
 second\_moment\_of\_area()  
 (sympy.geometry.ellipse.Ellipse  
 method), 2266  
 second\_moment\_of\_area()  
 (sympy.geometry.polygon.Polygon  
 method), 2282  
 SecondHypergeometric (class  
 sympy.solvers.ode.single), 794  
 SecondLinearAiry (class  
 sympy.solvers.ode.single), 777  
 SecondLinearBessel (class  
 sympy.solvers.ode.single), 778  
 section\_modulus()  
 (sympy.geometry.ellipse.Ellipse  
 method), 2267  
 section\_modulus()  
 (sympy.geometry.polygon.Polygon  
 method), 2283  
 Segment (class in sympy.geometry.line), 2233  
 Segment2D (class in sympy.geometry.line),  
 2242  
 Segment3D (class in sympy.geometry.line),  
 2247  
 select() (sympy.simplify.epathtools.EPath  
 method), 689  
 selections(sympy.combinatorics.graycode.GrayCode  
 property), 349  
 semilatus\_rectum(sympy.geometry.ellipse.Ellipse  
 property), 2268  
 sensitivity(sympy.physics.control.lti.Feedback  
 property), 1908  
 sensitivity(sympy.physics.control.lti.MIMOFeedback  
 property), 1930  
 Separable (class in  
 sympy.solvers.ode.single), 789  
 SeparableReduced (class in  
 sympy.solvers.ode.single), 792  
 separate() (sympy.core.expr.Expr method),  
 973  
 separate() (sympy.physics.vector.vector.Vector  
 method), 1646  
 separate() (sympy.vector.vector.Vector  
 method), 1457  
 separatevars() (in module  
 sympy.simplify.simplify), 664  
 SeqAdd (class in sympy.series.sequences), 636  
 SeqBase (class in sympy.series.sequences),  
 632  
 SeqFormula (class in  
 sympy.series.sequences), 634  
 SeqMul (class in sympy.series.sequences), 637  
 SeqPer (class in sympy.series.sequences), 635  
 SeqSum (class in sympy.series.sequences), 632  
 Series (class in sympy.physics.control.lti),  
 1899  
 series() (in module sympy.series.series),  
 626  
 series() (sympy.core.expr.Expr method),  
 973  
 in series() (sympy.holonomic.holonomic.HolonomicFunction  
 method), 2320  
 in series() (sympy.liealgebras.cartan\_type.Standard\_Cartan  
 method), 2340  
 in SeriesApprox (class in  
 sympy.codegen.approximations),  
 1122  
 Set (class in sympy.sets.sets), 1185  
 set\_acc() (sympy.physics.vector.point.Point  
 method), 1656  
 set\_ang\_acc() (sympy.physics.vector.frame.ReferenceFrame  
 method), 1656



method), 1640  
 set\_ang\_vel() (sympy.physics.vector.frame.ReferenceFrame method), 1640  
 set\_comm() (sympy.tensor.tensor.TensorManager method), 1422  
 set\_comms() (sympy.tensor.tensor.TensorManager method), 1423  
 set\_domain() (sympy.polys.polytools.Poly method), 2416  
 set\_global\_relative\_scale\_factor() (sympy.physics.units.quantities.Quantity method), 1588  
 set\_global\_settings() (sympy.printing.printer.Printer class method), 2139  
 set\_modulus() (sympy.polys.polytools.Poly method), 2416  
 set\_pos() (sympy.physics.vector.point.Point method), 1656  
 set\_vel() (sympy.physics.vector.point.Point method), 1656  
 seterr() (in module sympy.core.numbers), 995  
 SetKind (class in sympy.sets.conditionset), 1217  
 sets (sympy.sets.fancysets.ComplexRegion property), 1211  
 setup() (in module sympy.polys.polyconfig), 2643  
 sfield() (in module sympy.polys.fields), 2560  
 shanks() (in module sympy.series.acceleration), 630  
 shape (sympy.matrices.common.MatrixCommon property), 1353  
 shape (sympy.physics.control.lti.MIMOParallel property), 1927  
 shape (sympy.physics.control.lti.MIMOSeries property), 1924  
 shape (sympy.physics.control.lti.TransferFunctionBlock property), 1921  
 shape (sympy.tensor.indexed.Indexed property), 1403  
 shape (sympy.tensor.indexed.IndexedBase property), 1405  
 shape() (in module sympy.codegen.fnodes), 1155  
 ShapeError (class in sympy.matrices.matrices), 1318  
 shear\_force() (sympy.physics.continuum\_mechanics.beam.Beam method), 1976  
 shear\_force() (sympy.physics.continuum\_mechanics.beam.Beam method), 1997  
 shear\_modulus (sympy.physics.continuum\_mechanics.beam.Beam property), 1997  
 shear\_stress() (sympy.physics.continuum\_mechanics.beam.Beam method), 1976  
 shear\_stress() (sympy.physics.continuum\_mechanics.beam.Beam method), 1997  
 Shi (class in sympy.functions.special.error\_functions), 494  
 shift() (sympy.polys.polyclasses.DMP method), 2566  
 shift() (sympy.polys.polytools.Poly method), 2416  
 shift() (sympy.series.fourier.FourierSeries method), 641  
 ShiftedGompertz() (in module sympy.stats), 2925  
 shiftx() (sympy.series.fourier.FourierSeries method), 641  
 shor() (in module sympy.physics.quantum.shor), 1856  
 Si (class in sympy.functions.special.error\_functions), 491  
 sides (sympy.geometry.polygon.Polygon property), 2283  
 Sieve (class in sympy.ntheory.generate), 1476  
 sift() (in module sympy.utilities.iterables), 2093  
 sigma\_approximation() (sympy.series.fourier.FourierSeries method), 642  
 sign (class in sympy.functions.elementary.complexes), 384  
 sign (sympy.physics.control.lti.Feedback property), 1909  
 sign (sympy.physics.control.lti.MIMOFeedback property), 1931  
 sign() (in module sympy.series.gruntz), 624  
 signature (sympy.core.function.Lambda property), 1040  
 signmatrix() (sympy.combinatorics.permutations.Permutation method), 282  
 signed\_permutations() (in module sympy.utilities.iterables), 2094  
 SignedIntType (class in sympy.codegen.ast), 1136  
 simple\_root() (sympy.liealgebras.type\_a.TypeA method), 2329  
 simple\_root() (sympy.liealgebras.type\_b.TypeB method), 2331  
 simple\_root() (sympy.liealgebras.type\_c.TypeC method), 2332  
 simple\_root() (sympy.liealgebras.type\_d.TypeD method), 2334  
 simple\_root() (sympy.liealgebras.type\_e.TypeE method), 2335

`method`), 2335  
`simple_root()` (`sympy.liealgebras.type_f.TypeF` `method`), 2337  
`simple_root()` (`sympy.liealgebras.type_g.TypeG` `method`), 2338  
`simple_roots()` (`sympy.liealgebras.root_systems.RootSystem` `method`), 2328  
`simpleDE()` (in module `sympy.series.formal`), 656  
`SimpleDomain` (class in `sympy.polys.domains.simplesdomain`), 2522  
`simplify()` (in module `sympy.simplify.simplify`), 661  
`simplify()` (`sympy.core.basic.Basic` `method`), 941  
`simplify()` (`sympy.matrices.common.MatrixCommon` `method`), 1353  
`simplify()` (`sympy.matrices.dense.MutableDenseMatrix` `method`), 1363  
`simplify()` (`sympy.physics.vector.dyadic.Dyadic` `method`), 1650  
`simplify()` (`sympy.physics.vector.vector.Vector` `method`), 1646  
`simplify_gpgp()` (in module `sympy.physics.hep.gamma_matrices`), 1592  
`simplify_index_permutations()` (in module `sympy.physics.secondquant`), 1564  
`simplify_logic()` (in module `sympy.logic.boolalg`), 1176  
`sin` (class in `sympy.functions.elementary.trigonometric`), 389  
`sinc` (class in `sympy.functions.elementary.trigonometric`), 394  
`sine_transform()` (in module `sympy.integrals.transforms`), 581  
`SineTransform` (class in `sympy.integrals.transforms`), 582  
`SingleDomain` (class in `sympy.stats.rv`), 2981  
`SinglePSpace` (class in `sympy.stats.rv`), 2981  
`Singleton` (class in `sympy.core.singleton`), 946  
`SingletonRegistry` (class in `sympy.core.singleton`), 945  
`singular_points()` (`sympy.vector.implicitregion.ImplicitRegion` `method`), 1465  
`singular_value_decomposition()` (`sympy.matrices.matrices.MatrixBase` `method`), 1309  
`singular_values()` (`sympy.matrices.matrices.MatrixEigen` `method`), 1278  
`singularities()` (in module `sympy.calculus.singularities`), 239  
`SingularityFunction` (class in `sympy.functions.special.singularity_functions`), 456  
`sin_integrate()` (in module `sympy.integrals.singularityfunctions`), 591  
`SingularPredicate` (class in `sympy.assumptions.predicates.matrices`), 213  
`sinh` (class in `sympy.functions.elementary.hyperbolic`), 402  
`size` (`sympy.combinatorics.permutations.Permutation` `property`), 282  
`size` (`sympy.combinatorics.polyhedron.Polyhedron` `property`), 335  
`size` (`sympy.combinatorics.prufer.Prufer` `property`), 338  
`size` (`sympy.combinatorics.subsets.Subset` `property`), 344  
`size()` (in module `sympy.codegen.fnodes`), 1155  
`sizeof()` (in module `sympy.codegen.cnodes`), 1149  
`Skellam()` (in module `sympy.stats`), 2885  
`skewness()` (in module `sympy.stats`), 2975  
`SKIP()` (in module `sympy.testing.pytest`), 2027  
`skip()` (`sympy.combinatorics.graycode.GrayCode` `method`), 349  
`slice()` (`sympy.polys.polyclasses.DMP` `method`), 2566  
`slice()` (`sympy.polys.polytools.Poly` `method`), 2416  
`slope` (`sympy.geometry.line.LinearEntity2D` `property`), 2237  
`slope()` (`sympy.physics.continuum_mechanics.beam.Beam` `method`), 1976  
`slope()` (`sympy.physics.continuum_mechanics.beam.Beam3D` `method`), 1997  
`smallest_angle_between()` (`sympy.geometry.line.LinearEntity` `method`), 2227  
`smith_normal_form()` (in module `sympy.matrices.normalforms`), 1386  
`smith_normal_form()` (in module `sympy.polys.matrices.normalforms`), 2700  
`smoothness()` (in module `sympy.ntheory.factor_`), 1486  
`smoothness_p()` (in module `sympy.ntheory.factor_`), 1487  
`solve()` (in module `sympy.solvers.solvers`),

- 836
- `solve()` (`sympy.matrices.matrices.MatrixBase` method), 1312
- `solve_aux_eq()` (in module `sympy.solvers.ode.riccati`), 805
- `solve_congruence()` (in module `sympy.ntheory.modular`), 1511
- `solve_de()` (in module `sympy.series.formal`), 658
- `solve_for_ild_moment()` (`sympy.physics.continuum_mechanics.beam.Beam` method), 1977
- `solve_for_ild_reactions()` (`sympy.physics.continuum_mechanics.beam.Beam` method), 1978
- `solve_for_ild_shear()` (`sympy.physics.continuum_mechanics.beam.Beam` method), 1979
- `solve_for_reaction_loads()` (`sympy.physics.continuum_mechanics.beam.Beam` method), 1980
- `solve_for_reaction_loads()` (`sympy.physics.continuum_mechanics.beam.Beam` method), 1997
- `solve_least_squares()` (`sympy.matrices.matrices.MatrixBase` method), 1313
- `solve_lin_sys()` (in module `sympy.polys.solvers`), 2666
- `solve_linear()` (in module `sympy.solvers.solvers`), 844
- `solve_linear_system()` (in module `sympy.solvers.solvers`), 846
- `solve_linear_system_LU()` (in module `sympy.solvers.solvers`), 847
- `solve_multipliers()` (`sympy.physics.mechanics.lagrange.LagrangianMechanics` method), 1770
- `solve_poly_inequalities()` (in module `sympy.solvers.inequalities`), 751
- `solve_poly_inequality()` (in module `sympy.solvers.inequalities`), 751
- `solve_poly_system()` (in module `sympy.solvers.polysys`), 856
- `solve_rational_inequalities()` (in module `sympy.solvers.inequalities`), 750
- `solve_riccati()` (in module `sympy.solvers.ode.riccati`), 805
- `solve_triangulated()` (in module `sympy.solvers.polysys`), 857
- `solve_undetermined_coeffs()` (in module `sympy.solvers.solvers`), 847
- `solve_univariate_inequality()` (in module `sympy.solvers.inequalities`), 753
- `solveset()` (in module `sympy.solvers.solveset`), 866
- `solveset_complex()` (in module `sympy.solvers.solveset`), 868
- `solveset_real()` (in module `sympy.solvers.solveset`), 868
- `solvify()` (in module `sympy.solvers.solveset`), 869
- `SOPform()` (in module `sympy.logic.boolalg`), 1160
- `sort_key()` (`sympy.combinatorics.partitions.Partition` method), 253
- `sort_key()` (`sympy.core.basic.Basic` method), 941
- `sorted_components()` (`sympy.tensor.tensor.TensMul` method), 1419
- `source` (`sympy.geometry.line.Ray` property), 2232
- `source_de()` (in module `sympy.utilities.source`), 2116
- `SpaceOrienter` (class in `sympy.vector.orienters`), 1468
- `spaces` (`sympy.physics.quantum.hilbert.DirectSumHilbertSpace` property), 1803
- `spaces` (`sympy.physics.quantum.hilbert.TensorProductHilbertSpace` property), 1806
- `SparseMatrix` (in module `sympy.matrices.sparse`), 1364
- `speed` (`sympy.physics.optics.medium.Medium` property), 1871
- `speed` (`sympy.physics.optics.waves.TWave` property), 1889
- `speeds` (`sympy.physics.mechanics.joint.Joint` property), 1772
- `speeds` (`sympy.physics.mechanics.system.SymbolicSystem` property), 1787
- `spherical_bessel_fn()` (in module `sympy.polys.orthopolys`), 2441
- `spin()` (`sympy.geometry.polygon.RegularPolygon` method), 2292
- `split()` (`sympy.tensor.tensor.TensMul` method), 1419
- `split_list()` (in module `sympy.testing.runtests`), 2033
- `split_super_sub()` (in module `sympy.printing.conventions`), 2183
- `split_symbols()` (in module `sympy.parsing.sympy_parser`), 2126
- `split_symbols_custom()` (in module `sympy.parsing.sympy_parser`), 2126
- `spoly()` (in module `sympy.polys.groebnertools`), 2640
- `sqf()` (in module `sympy.polys.polytools`),

[2372](#)  
`sqf_list()` (in module `sympy.polys.polytools`), [2372](#)  
`sqf_list()` (`sympy.polys.polyclasses.DMP` method), [2566](#)  
`sqf_list()` (`sympy.polys.polytools.Poly` method), [2416](#)  
`sqf_list_include()` (`sympy.polys.polyclasses.DMP` method), [2566](#)  
`sqf_list_include()` (`sympy.polys.polytools.Poly` method), [2417](#)  
`sqf_norm()` (in module `sympy.polys.polytools`), [2371](#)  
`sqf_norm()` (`sympy.polys.polyclasses.DMP` method), [2566](#)  
`sqf_norm()` (`sympy.polys.polytools.Poly` method), [2417](#)  
`sqf_normal()` (in module `sympy.solvers.diophantine.diophantine`), [743](#)  
`sqf_part()` (in module `sympy.polys.polytools`), [2372](#)  
`sqf_part()` (`sympy.polys.polyclasses.DMP` method), [2566](#)  
`sqf_part()` (`sympy.polys.polytools.Poly` method), [2418](#)  
`sqr()` (`sympy.polys.polyclasses.DMP` method), [2566](#)  
`sqr()` (`sympy.polys.polytools.Poly` method), [2418](#)  
`Sqrt` (class in `sympy.codegen.cfunctions`), [1144](#)  
`sqrt()` (in module `sympy.functions.elementary.miscellaneous`), [424](#)  
`sqrt()` (`sympy.polys.domains.domain.Domain` method), [2517](#)  
`sqrt()` (`sympy.polys.domains.GMPYIntegerRing` method), [2529](#)  
`sqrt()` (`sympy.polys.domains.IntegerRing` method), [2528](#)  
`sqrt_mod()` (in module `sympy.ntheory.residue_ntheory`), [1519](#)  
`sqrt_mod_iter()` (in module `sympy.ntheory.residue_ntheory`), [1520](#)  
`sqrtdenest()` (in module `sympy.simplify.sqrtdenest`), [684](#)  
`square()` (`sympy.polys.rings.PolyElement` method), [2559](#)  
`square_factor()` (in module `sympy.solvers.diophantine.diophantine`), [730](#)  
`SquarePredicate` (class in `sympy.assumptions.predicates.matrices`), [211](#)  
`srepr()` (in module `sympy.printing.repr`), [2178](#)  
`sring()` (in module `sympy.polys.rings`), [2553](#)  
`sstr()` (in module `sympy.printing.str`), [2178](#)  
`sstrrepr()` (in module `sympy.printing.str`), [2178](#)  
`stabilizer()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), [328](#)  
`stack()` (`sympy.printing.pretty.stringpict.stringPict` static method), [2187](#)  
`Standard_Cartan` (class in `sympy.liealgebras.cartan_type`), [2340](#)  
`standard_transformations` (in module `sympy.parsing.sympy_parser`), [2126](#)  
`start` (`sympy.series.sequences.RecursiveSequence` property), [640](#)  
`start` (`sympy.series.sequences.SeqBase` property), [634](#)  
`start` (`sympy.sets.sets.Interval` property), [1196](#)  
`starts_with_unity()` (`sympy.polys.numberfields.modules.Module` method), [2723](#)  
`State` (class in `sympy.physics.quantum.state`), [1834](#)  
`state()` (`sympy.physics.secondquant.FixedBosonicBasis` method), [1551](#)  
`state()` (`sympy.physics.secondquant.VarBosonicBasis` method), [1561](#)  
`state_to_operators()` (in module `sympy.physics.quantum.operatorset`), [1813](#)  
`StateBase` (class in `sympy.physics.quantum.state`), [1834](#)  
`states` (`sympy.physics.mechanics.system.SymbolicSystem` property), [1787](#)  
`stationary_distribution()` (`sympy.stats.DiscreteMarkovChain` method), [2950](#)  
`stationary_points()` (in module `sympy.calculus.util`), [250](#)  
`std()` (in module `sympy.stats`), [2970](#)  
`step_response_numerical_data()` (`sympy.physics.control.control_plots` method), [1944](#)  
`step_response_plot()` (`sympy.physics.control.control_plots` method), [1943](#)



`stieltjes` (class in `sturm()` (`sympy.polys.polytools.Poly` method),  
`sympy.functions.special.zeta_functions`), 2418  
519  
`stirling()` (in module `sympy.printing.pretty.pretty_symbology`), 2185  
`sympy.functions.combinatorial.numbers` `sub()` (`sympy.polys.domains.domain.Domain`  
444 method), 2517  
`stokes_vector()` (in module `sub()` (`sympy.polys.matrices.ddm.DDM`  
`sympy.physics.optics.polarization`), method), 2691  
1878  
`stop` (`sympy.series.sequences.RecursiveSeq` `sub()` (`sympy.polys.matrices.domainmatrix.DomainMatrix`  
property), 640 method), 2686  
`stop` (`sympy.series.sequences.SeqBase` prop- `sub()` (`sympy.polys.matrices.sdm.SDM`  
erty), 634 method), 2698  
`Stream` (class in `sympy.codegen.ast`), 1136 `sub()` (`sympy.polys.polyclasses.DMF`  
method), 2568  
`strict` (`sympy.core.relational.Relational` `sub()` (`sympy.polys.polyclasses.DMP`  
property), 1021 method), 2566  
`StrictGreaterThan` (class in `sub()` (`sympy.polys.polytools.Poly` method),  
`sympy.core.relational`), 1032 2419  
`StrictLessThan` (class in `sub_ground()` (`sympy.polys.polyclasses.DMP`  
`sympy.core.relational`), 1035 method), 2567  
`strides` (`sympy.tensor.indexed.IndexedBase` `sub_ground()` (`sympy.polys.polytools.Poly`  
property), 1406 method), 2419  
`String` (class in `sympy.codegen.ast`), 1137 `subdiagram_from_objects()`  
`stringify_expr()` (in module (`sympy.categories.Diagram` method),  
`sympy.parsing.sympy_parser`), 2125 2752  
`stringPict` (class in `subfactorial` (class in  
`sympy.printing.pretty.stringpict`), `sympy.functions.combinatorial.factorials`),  
2186 435  
`strip_zero()` (`sympy.polys.rings.PolyElement` `subgroup()` (`sympy.combinatorics.perm_groups.Permutation`  
method), 2559 method), 329  
`strlines()` (in module `sympy.utilities.misc`), `subgroup_search()`  
2114 (`sympy.combinatorics.perm_groups.PermutationGroup` method), 329  
`strong_gens` (`sympy.combinatorics.perm_groups.PermutationGroup` property), 328  
`strong_presentation()` `SubModule` (class in  
(`sympy.combinatorics.perm_groups.PermutationGroup` method), 329 `sympy.polys.agca.modules`), 2456  
`strongly_connected_components()` (in mod- `SubModuleGroup` (class in  
ule `sympy.utilities.iterables`), 2094 `sympy.polys.numberfields.modules`),  
2726  
`strongly_connected_components()` `submodule()` (`sympy.polys.agca.modules.Module`  
(`sympy.matrices.matrices.MatrixBase` method), 2453  
method), 1314 `submodule()` (`sympy.polys.agca.modules.QuotientModule`  
method), 2466  
`strongly_connected_components_decomposition()` `submodule()` (`sympy.polys.agca.modules.SubModule`  
(`sympy.matrices.matrices.MatrixBase` method), 2460  
method), 1315 `submodule_from_gens()`  
(`sympy.polys.numberfields.modules.Module`  
method), 2723  
`StrPrinter` (class in `sympy.printing.str`), 2178  
`struct` (class in `sympy.codegen.cnodes`), 1149  
`StudentT()` (in module `sympy.stats`), 2924  
`sturm()` (in module `sympy.polys.polytools`), 2370  
`sturm()` (`sympy.polys.polyclasses.DMP` `subresultants()` (in module  
method), 2566 `sympy.polys.polytools`), 2365

`subresultants()` (`sympy.polys.polyclasses.DMP` [method](#)), [2567](#)  
`subresultants()` (`sympy.polys.polytools.Poly` [method](#)), [2419](#)  
`Subroutine` (class in `sympy.codegen.fnodes`), [1152](#)  
`SubroutineCall` (class in `sympy.codegen.fnodes`), [1152](#)  
`Subs` (class in `sympy.core.function`), [1051](#)  
`subs()` (`sympy.core.basic.Basic` [method](#)), [941](#)  
`subs()` (`sympy.matrices.common.MatrixCommon` [method](#)), [1354](#)  
`subs()` (`sympy.physics.vector.dyadic.Dyadic` [method](#)), [1650](#)  
`subs()` (`sympy.physics.vector.vector.Vector` [method](#)), [1646](#)  
`Subset` (class in `sympy.combinatorics.subsets`), [340](#)  
`subset` (`sympy.combinatorics.subsets.Subset` [property](#)), [344](#)  
`subset()` (`sympy.polys.agca.ideals.Ideal` [method](#)), [2464](#)  
`subset()` (`sympy.polys.agca.modules.Module` [method](#)), [2453](#)  
`subset_from_bitlist()` (`sympy.combinatorics.subsets.Subset` [class method](#)), [345](#)  
`subset_indices()` (`sympy.combinatorics.subsets.Subset` [class method](#)), [345](#)  
`subsets()` (in module `sympy.utilities.iterables`), [2096](#)  
`SubsSet` (class in `sympy.series.gruntz`), [624](#)  
`substitute_dummies()` (in module `sympy.physics.secondquant`), [1564](#)  
`Sum` (class in `sympy.concrete.summations`), [900](#)  
`sum_of_four_squares()` (in module `sympy.solvers.diophantine.diophantine`), [735](#)  
`sum_of_powers()` (in module `sympy.solvers.diophantine.diophantine`), [736](#)  
`sum_of_squares()` (in module `sympy.solvers.diophantine.diophantine`), [737](#)  
`sum_of_three_squares()` (in module `sympy.solvers.diophantine.diophantine`), [734](#)  
`SumApprox` (class in `sympy.codegen.approximations`), [1123](#)  
`summation()` (in module `sympy.concrete.summations`), [915](#)  
`sup` (in module `sympy.printing.pretty.pretty_symbolic`), [2185](#)  
`sup` (`sympy.sets.sets.Set` [property](#)), [1192](#)  
`superposition_basis()` (in module `sympy.physics.quantum.grover`), [1849](#)  
`superset` (`sympy.combinatorics.subsets.Subset` [property](#)), [345](#)  
`superset_size` (`sympy.combinatorics.subsets.Subset` [property](#)), [346](#)  
`supplement_a_subspace()` (in module `sympy.polys.numberfields.utilities`), [2740](#)  
`support()` (`sympy.combinatorics.permutations.Permutation` [method](#)), [282](#)  
`supports` (`sympy.physics.continuum_mechanics.truss.Truss` [property](#)), [2027](#)  
`SurfaceBaseSeries` (class in `sympy.plotting.plot`), [2867](#)  
`SurfaceOver2DRangeSeries` (class in `sympy.plotting.plot`), [2867](#)  
`SWAP` (in module `sympy.physics.quantum.gate`), [1845](#)  
`swap_point()` (`sympy.physics.quantum.circuitplot.CircuitPlot` [method](#)), [1841](#)  
`SwapGate` (class in `sympy.physics.quantum.gate`), [1845](#)  
`swinnerton_dyer_poly()` (in module `sympy.polys.specialpolys`), [2438](#)  
`syLOW_subgroup()` (`sympy.combinatorics.perm_groups.PermGroup` [method](#)), [330](#)  
`symarray()` (in module `sympy.matrices.dense`), [1323](#)  
`symb_2txt` (in module `sympy.printing.pretty.pretty_symbolic`), [2185](#)  
`Symbol` (class in `sympy.core.symbol`), [976](#)  
`symbol` (`sympy.physics.secondquant.AntiSymmetricTensor` [property](#)), [1545](#)  
`SymbolicSystem` (class in `sympy.physics.mechanics.system`), [1783](#)  
`symbols()` (in module `sympy.core.symbol`), [978](#)  
`symmetric()` (`sympy.combinatorics.generators` [method](#)), [287](#)  
`symmetric_difference()` (`sympy.sets.sets.Set` [method](#)), [1192](#)  
`symmetric_poly()` (in module `sympy.polys.specialpolys`), [2438](#)  
`symmetric_residue()` (in module `sympy.ntheory.modular`), [1509](#)  
`SymmetricDifference` (class in `sympy.sets.sets`), [1201](#)  
`SymmetricGroup()` (in module `sympy.combinatorics.perm_groups`), [330](#)

[sympy.combinatorics.named\\_groups](#)),  
[352](#)  
[SymmetricPredicate](#) (class in [sympy.assumptions.predicates.matrices](#))  
[203](#)  
[symmetrize\(\)](#) (in [sympy.polys.polyfuncs](#)), [2425](#)  
[sympify\(\)](#) (in [module sympy.core.sympify](#)),  
[918](#)  
[sympy.abc](#)  
[module, 884](#)  
[sympy.algebras](#)  
[module, 885](#)  
[sympy.assumptions](#)  
[module, 186](#)  
[sympy.assumptions.ask](#)  
[module, 191](#)  
[sympy.assumptions.assume](#)  
[module, 193](#)  
[sympy.assumptions.predicates](#)  
[module, 200](#)  
[sympy.assumptions.refine](#)  
[module, 197](#)  
[sympy.calculus](#)  
[module, 233](#)  
[sympy.calculus.euler](#)  
[module, 233](#)  
[sympy.calculus.finite\\_diff](#)  
[module, 240](#)  
[sympy.calculus.singularities](#)  
[module, 234](#)  
[sympy.calculus.util](#)  
[module, 244](#)  
[sympy.categories](#)  
[module, 2742](#)  
[sympy.categories.diagram\\_drawing](#)  
[module, 2752](#)  
[sympy.codegen.algorithms](#)  
[module, 1157](#)  
[sympy.codegen.approximations](#)  
[module, 1122](#)  
[sympy.codegen.ast](#)  
[module, 1124](#)  
[sympy.codegen.cfunctions](#)  
[module, 1143](#)  
[sympy.codegen.cnodes](#)  
[module, 1148](#)  
[sympy.codegen.cutils](#)  
[module, 1159](#)  
[sympy.codegen.cxxnodes](#)  
[module, 1149](#)  
[sympy.codegen.fnodes](#)  
[module, 1149](#)  
[sympy.codegen.futils](#)  
[module, 1159](#)  
[sympy.codegen.matrix\\_nodes](#)  
[module, 1121](#)  
[sympy.codegen.pyutils](#)  
[module, 1159](#)  
[sympy.codegen.rewriting](#)  
[module, 1118](#)  
[sympy.combinatorics.generators](#)  
[module, 287](#)  
[sympy.combinatorics.graycode](#)  
[module, 347](#)  
[sympy.combinatorics.group\\_constructs](#)  
[module, 363](#)  
[sympy.combinatorics.group\\_numbers](#)  
[module, 355](#)  
[sympy.combinatorics.named\\_groups](#)  
[module, 352](#)  
[sympy.combinatorics.partitions](#)  
[module, 251](#)  
[sympy.combinatorics.perm\\_groups](#)  
[module, 288](#)  
[sympy.combinatorics.permutations](#)  
[module, 257](#)  
[sympy.combinatorics.polyhedron](#)  
[module, 332](#)  
[sympy.combinatorics.prufer](#)  
[module, 335](#)  
[sympy.combinatorics.subsets](#)  
[module, 340](#)  
[sympy.combinatorics.tensor\\_can](#)  
[module, 365](#)  
[sympy.combinatorics.testutil](#)  
[module, 363](#)  
[sympy.combinatorics.util](#)  
[module, 356](#)  
[sympy.core.add](#)  
[module, 1013](#)  
[sympy.core.assumptions](#)  
[module, 923](#)  
[sympy.core.basic](#)  
[module, 927](#)  
[sympy.core.cache](#)  
[module, 927](#)  
[sympy.core.containers](#)  
[module, 1068](#)  
[sympy.core.core](#)  
[module, 945](#)  
[sympy.core.evalf](#)  
[module, 1065](#)  
[sympy.core.expr](#)  
[module, 947](#)  
[sympy.core.exprtools](#)  
[module, 1071](#)  
[sympy.core.function](#)

- module, [1039](#)
- sympy.core.kind
  - module, [1073](#)
- sympy.core.mod
  - module, [1018](#)
- sympy.core.mul
  - module, [1009](#)
- sympy.core.multidimensional
  - module, [1039](#)
- sympy.core.numbers
  - module, [981](#)
- sympy.core.power
  - module, [1005](#)
- sympy.core.random
  - module, [1077](#)
- sympy.core.relational
  - module, [1018](#)
- sympy.core.singleton
  - module, [945](#)
- sympy.core.symbol
  - module, [976](#)
- sympy.core.sympify
  - module, [918](#)
- sympy.core.traversal
  - module, [1080](#)
- sympy.crypto.crypto
  - module, [2762](#)
- sympy.diffgeom
  - module, [2799](#)
- sympy.discrete
  - module, [1083](#)
- sympy.discrete.convolution
  - module, [1089](#)
- sympy.discrete.transforms
  - module, [1083](#)
- sympy.functions
  - module, [381](#)
- sympy.functions.special.bessel
  - module, [497](#)
- sympy.functions.special.beta\_functions
  - module, [470](#)
- sympy.functions.special.elliptic\_integrals
  - module, [526](#)
- sympy.functions.special.error\_functions
  - module, [471](#)
- sympy.functions.special.gamma\_functions
  - module, [459](#)
- sympy.functions.special.mathieu\_functions
  - module, [529](#)
- sympy.functions.special.polynomials
  - module, [532](#)
- sympy.functions.special.singularity\_functions
  - module, [456](#)
- sympy.functions.special.zeta\_functions
  - module, [514](#)
- sympy.geometry.curve
  - module, [2247](#)
- sympy.geometry.ellipse
  - module, [2253](#)
- sympy.geometry.entity
  - module, [2194](#)
- sympy.geometry.line
  - module, [2216](#)
- sympy.geometry.plane
  - module, [2303](#)
- sympy.geometry.point
  - module, [2202](#)
- sympy.geometry.polygon
  - module, [2273](#)
- sympy.geometry.util
  - module, [2197](#)
- sympy.holonomic
  - module, [2312](#)
- sympy.integrals
  - module, [575](#)
- sympy.integrals.intpoly
  - module, [617](#)
- sympy.integrals.meijerint
  - module, [568](#)
- sympy.integrals.meijerint\_doc
  - module, [564](#)
- sympy.integrals.transforms
  - module, [576](#)
- sympy.interactive
  - module, [2116](#)
- sympy.interactive.printing
  - module, [2119](#)
- sympy.interactive.session
  - module, [2116](#)
- sympy.liealgebras
  - module, [2326](#)
- sympy.logic
  - module, [1160](#)
- sympy.logic.inference
  - module, [1184](#)
- sympy.matrices
  - module, [1217](#)
- sympy.matrices.common
  - module, [1327](#)
- sympy.matrices.expressions
  - module, [1369](#)
- sympy.matrices.expressions.blockmatrix
  - module, [1380](#)
- sympy.matrices.immutable
  - module, [1369](#)
- sympy.matrices.matrices
  - module, [1218](#)
- sympy.matrices.sparse
  - module, [1218](#)



module, 1364  
 sympy.matrices.sparsetools  
   module, 1365  
 sympy.ntheory.bbp\_pi  
   module, 1531  
 sympy.ntheory.continued\_fraction  
   module, 1523  
 sympy.ntheory.digits  
   module, 1526  
 sympy.ntheory.ecm  
   module, 1531  
 sympy.ntheory.egyptian\_fraction  
   module, 1529  
 sympy.ntheory.factor\_  
   module, 1486  
 sympy.ntheory.generate  
   module, 1476  
 sympy.ntheory.modular  
   module, 1509  
 sympy.ntheory.multinomial  
   module, 1511  
 sympy.ntheory.partitions\_  
   module, 1513  
 sympy.ntheory.primetest  
   module, 1513  
 sympy.ntheory.qs  
   module, 1532  
 sympy.ntheory.residue\_ntheory  
   module, 1518  
 sympy.parsing  
   module, 2122  
 sympy.parsing.sym\_expr  
   module, 2130  
 sympy.physics  
   module, 1533  
 sympy.physics.continuum\_mechanics.beam  
   module, 1949  
 sympy.physics.continuum\_mechanics.truss  
   module, 2021  
 sympy.physics.control  
   module, 1890  
 sympy.physics.control.lti  
   module, 1891  
 sympy.physics.hep.gamma\_matrices  
   module, 1589  
 sympy.physics.hydrogen  
   module, 1533  
 sympy.physics.matrices  
   module, 1537  
 sympy.physics.mechanics  
   module, 1675  
 sympy.physics.mechanics.body  
   module, 1756  
 sympy.physics.mechanics.joint  
   module, 1770  
 sympy.physics.mechanics.jointsmethod  
   module, 1780  
 sympy.physics.mechanics.kane  
   module, 1764  
 sympy.physics.mechanics.lagrange  
   module, 1767  
 sympy.physics.mechanics.linearize  
   module, 1787  
 sympy.physics.mechanics.particle  
   module, 1743  
 sympy.physics.mechanics.rigidbody  
   module, 1746  
 sympy.physics.mechanics.system  
   module, 1783  
 sympy.physics.optics.gaussopt  
   module, 1857  
 sympy.physics.optics.medium  
   module, 1870  
 sympy.physics.optics.polarization  
   module, 1871  
 sympy.physics.optics.utils  
   module, 1880  
 sympy.physics.optics.waves  
   module, 1887  
 sympy.physics.paulialgebra  
   module, 1539  
 sympy.physics.qho\_ld  
   module, 1539  
 sympy.physics.quantum.anticommutator  
   module, 1791  
 sympy.physics.quantum.cartesian  
   module, 1801  
 sympy.physics.quantum.cg  
   module, 1792  
 sympy.physics.quantum.circuitplot  
   module, 1840  
 sympy.physics.quantum.commutator  
   module, 1795  
 sympy.physics.quantum.constants  
   module, 1796  
 sympy.physics.quantum.dagger  
   module, 1797  
 sympy.physics.quantum.gate  
   module, 1842  
 sympy.physics.quantum.grover  
   module, 1847  
 sympy.physics.quantum.hilbert  
   module, 1802  
 sympy.physics.quantum.innerproduct  
   module, 1798  
 sympy.physics.quantum.operator  
   module, 1807  
 sympy.physics.quantum.operatorset

- module, [1812](#)
- `sympy.physics.quantum.piab`
  - module, [1857](#)
- `sympy.physics.quantum.qapply`
  - module, [1814](#)
- `sympy.physics.quantum.qft`
  - module, [1850](#)
- `sympy.physics.quantum.qubit`
  - module, [1850](#)
- `sympy.physics.quantum.represent`
  - module, [1815](#)
- `sympy.physics.quantum.shor`
  - module, [1856](#)
- `sympy.physics.quantum.spin`
  - module, [1820](#)
- `sympy.physics.quantum.state`
  - module, [1832](#)
- `sympy.physics.quantum.tensorproduct`
  - module, [1799](#)
- `sympy.physics.secondquant`
  - module, [1542](#)
- `sympy.physics.sho`
  - module, [1541](#)
- `sympy.physics.units.dimensions`
  - module, [1584](#)
- `sympy.physics.units.prefixes`
  - module, [1586](#)
- `sympy.physics.units.quantities`
  - module, [1587](#)
- `sympy.physics.units.unitsystem`
  - module, [1587](#)
- `sympy.physics.units.util`
  - module, [1588](#)
- `sympy.physics.vector`
  - module, [1592](#)
- `sympy.physics.vector.functions`
  - module, [1659](#)
- `sympy.physics.vector.point`
  - module, [1652](#)
- `sympy.physics.wigner`
  - module, [1566](#)
- `sympy.plotting.plot`
  - module, [2820](#)
- `sympy.plotting.pygletplot`
  - module, [2869](#)
- `sympy.polys`
  - module, [2360](#)
- `sympy.polys.numberfields.modules`
  - module, [2716](#)
- `sympy.polys.numberfields.subfield`
  - module, [2711](#)
- `sympy.polys.polyconfig`
  - module, [2643](#)
- `sympy.polys.polyoptions`
  - module, [2642](#)
- `sympy.polys.solvers`
  - module, [2666](#)
- `sympy.printing.aesaracode`
  - module, [2166](#)
- `sympy.printing.c`
  - module, [2140](#)
- `sympy.printing.codeprinter`
  - module, [2183](#)
- `sympy.printing.conventions`
  - module, [2183](#)
- `sympy.printing.cxx`
  - module, [2144](#)
- `sympy.printing.fortran`
  - module, [2147](#)
- `sympy.printing.gtk`
  - module, [2170](#)
- `sympy.printing.jrcode`
  - module, [2155](#)
- `sympy.printing.julia`
  - module, [2157](#)
- `sympy.printing.lambdarepr`
  - module, [2170](#)
- `sympy.printing.latex`
  - module, [2170](#)
- `sympy.printing.maple`
  - module, [2154](#)
- `sympy.printing.mathematica`
  - module, [2152](#)
- `sympy.printing.mathml`
  - module, [2175](#)
- `sympy.printing.octave`
  - module, [2160](#)
- `sympy.printing.precedence`
  - module, [2184](#)
- `sympy.printing.pretty`
  - module, [2139](#)
- `sympy.printing.pretty.pretty`
  - module, [2139](#)
- `sympy.printing.pretty.pretty_symbology`
  - module, [2184](#)
- `sympy.printing.pretty.stringpict`
  - module, [2186](#)
- `sympy.printing.preview`
  - module, [2181](#)
- `sympy.printing.printer`
  - module, [2135](#)
- `sympy.printing.pycode`
  - module, [2176](#)
- `sympy.printing.python`
  - module, [2177](#)
- `sympy.printing.rcode`
  - module, [2144](#)
- `sympy.printing.repr`

---

- module, 2177
- sympy.printing.rust
  - module, 2164
- sympy.printing.str
  - module, 2178
- sympy.printing.tree
  - module, 2178
- sympy.sets.conditionset
  - module, 1215
- sympy.sets.fancysets
  - module, 1203
- sympy.sets.powerset
  - module, 1213
- sympy.sets.sets
  - module, 1185
- sympy.simplify.combsimp
  - module, 683
- sympy.simplify.cse\_main
  - module, 685
- sympy.simplify.epathtools
  - module, 688
- sympy.simplify.hyperexpand
  - module, 687
- sympy.simplify.hyperexpand\_doc
  - module, 699
- sympy.simplify.powsimp
  - module, 680
- sympy.simplify.radsimp
  - module, 670
- sympy.simplify.ratsimp
  - module, 678
- sympy.simplify.simplify
  - module, 154
- sympy.simplify.sqrtdenest
  - module, 684
- sympy.simplify.trigsimp
  - module, 679
- sympy.solvers
  - module, 836
- sympy.solvers.inequalities
  - module, 750
- sympy.solvers.ode
  - module, 755
- sympy.solvers.ode.ode
  - module, 822
- sympy.solvers.pde
  - module, 835
- sympy.solvers.recurr
  - module, 853
- sympy.solvers.solve
  - module, 858
- sympy.stats
  - module, 2873
- sympy.stats.crv
  - module, 2981
- sympy.stats.crv\_types
  - module, 2981
- sympy.stats.Die()
  - (in *sympy.stats.crv\_types*), 2982
- sympy.stats.frv
  - module, 2981
- sympy.stats.frv\_types
  - module, 2981
- sympy.stats.Normal()
  - (in *sympy.stats.crv\_types*), 2982
- sympy.stats.rv
  - module, 2980
- sympy.tensor
  - module, 1387
- sympy.tensor.array
  - module, 1387
- sympy.tensor.array.expressions
  - module, 1395
- sympy.tensor.index\_methods
  - module, 1406
- sympy.tensor.indexed
  - module, 1398
- sympy.tensor.tensor
  - module, 1409
- sympy.tensor.toperators
  - module, 1423
- sympy.testing
  - module, 2027
- sympy.testing.pytest
  - module, 2027
- sympy.testing.randtest
  - module, 2030
- sympy.testing.runtests
  - module, 2030
- sympy.utilities
  - module, 2038
- sympy.utilities.autowrap
  - module, 2039
- sympy.utilities.codegen
  - module, 2046
- sympy.utilities.decorator
  - module, 2057
- sympy.utilities.enumerative
  - module, 2060
- sympy.utilities.exceptions
  - module, 2066
- sympy.utilities.iterables
  - module, 2071
- sympy.utilities.lambdify
  - module, 2099
- sympy.utilities.memoization
  - module, 2110
- sympy.utilities.misc

module, 2110  
sympy.utilities.pkgdata  
  module, 2115  
sympy.utilities.source  
  module, 2116  
sympy.utilities.timeutils  
  module, 2116  
sympy.vector  
  module, 1425  
sympy\_deprecation\_warning() (in module  
  sympy.utilities.exceptions), 2067  
sympy\_eqs\_to\_ring() (in module  
  sympy.polys.solvers), 2668  
SymPyDeprecationWarning, 2066  
SymPyDocTestFinder (class in  
  sympy.testing.runtests), 2030  
SymPyDocTestRunner (class in  
  sympy.testing.runtests), 2030  
SymPyExpression (class in  
  sympy.parsing.sym\_expr), 2130  
SymPyOutputChecker (class in  
  sympy.testing.runtests), 2031  
sympytestfile() (in module  
  sympy.testing.runtests), 2034  
SymPyTestResults (class in  
  sympy.testing.runtests), 2031  
sys1 (sympy.physics.control.lti.Feedback  
  property), 1909  
sys1 (sympy.physics.control.lti.MIMOFeedback  
  property), 1931  
sys2 (sympy.physics.control.lti.Feedback  
  property), 1909  
sys2 (sympy.physics.control.lti.MIMOFeedback  
  property), 1932  
syzygy\_module() (sympy.polys.agca.modules.SubModule  
  method), 2460

**T**  
T (in module sympy.physics.quantum.gate),  
  1845  
T (sympy.matrices.common.MatrixCommon  
  property), 1327  
T (sympy.matrices.expressions.MatrixExpr  
  property), 1370  
t (sympy.physics.quantum.shor.CMod prop-  
  erty), 1856  
T (sympy.polys.numberfields.modules.PowerBasisElement  
  property), 2732  
table() (sympy.matrices.matrices.MatrixBase  
  method), 1317  
tail\_degree() (sympy.polys.rings.PolyElement  
  method), 2559  
tail\_degrees() (sympy.polys.rings.PolyElement  
  method), 2559  
take() (in module sympy.utilities.iterables),  
  2096  
tan (class in sympy.functions.elementary.trigonometric),  
  391  
tangent\_lines() (sympy.geometry.ellipse.Ellipse  
  method), 2268  
tanh (class in sympy.functions.elementary.hyperbolic),  
  403  
targets (sympy.physics.quantum.gate.CGate  
  property), 1843  
targets (sympy.physics.quantum.gate.CNotGate  
  property), 1843  
targets (sympy.physics.quantum.gate.Gate  
  property), 1844  
targets (sympy.physics.quantum.gate.UGate  
  property), 1846  
targets (sympy.physics.quantum.grover.OracleGate  
  property), 1848  
taxicab\_distance() (sympy.geometry.point.Point  
  method), 2208  
taylor\_term() (sympy.core.expr.Expr  
  method), 975  
taylor\_term() (sympy.functions.elementary.exponential.ex  
  static method), 412  
taylor\_term() (sympy.functions.elementary.exponential.log  
  static method), 414  
taylor\_term() (sympy.functions.elementary.hyperbolic.csch  
  static method), 405  
taylor\_term() (sympy.functions.elementary.hyperbolic.sinh  
  static method), 402  
TC() (sympy.polys.polyclasses.ANP method),  
  2568  
TC() (sympy.polys.polyclasses.DMP method),  
  2561  
TC() (sympy.polys.polytools.Poly method),  
  2380  
TensAdd (class in sympy.tensor.tensor), 1415  
TensExpr (class in sympy.tensor.tensor), 1413  
TensMul (class in sympy.tensor.tensor), 1416  
tensor\_heads() (in module  
  sympy.tensor.tensor), 1413  
tensor\_product\_simp() (in module  
  sympy.physics.quantum.tensorproduct),  
  1800  
tensorcontraction() (in module  
  sympy.tensor.array), 1393  
tensordiagonal() (in module  
  sympy.tensor.array), 1394  
TensorHead (class in sympy.tensor.tensor),  
  1411  
TensorIndex (class in sympy.tensor.tensor),  
  1410  
TensorIndexType (class in

- `sympy.tensor.tensor`), 1409
- TensorPowerHilbertSpace (class in `sympy.physics.quantum.hilbert`), 1805
- TensorProduct (class in `sympy.diffgeom`), 2811
- TensorProduct (class in `sympy.physics.quantum.tensorproduct`), 1799
- `tensorproduct()` (in module `sympy.tensor.array`), 1394
- TensorProductHilbertSpace (class in `sympy.physics.quantum.hilbert`), 1806
- TensorSymmetry (class in `sympy.tensor.tensor`), 1420
- `tensorsymmetry()` (in module `sympy.tensor.tensor`), 1421
- TensorType (class in `sympy.tensor.tensor`), 1421
- `term_to_integer()` (in module `sympy.logic.boolalg`), 1181
- `terminal_width()` (`sympy.printing.pretty.stringpict.as_string` method), 2187
- `terms()` (`sympy.polys.polyclasses.DMP` method), 2567
- `terms()` (`sympy.polys.polytools.Poly` method), 2420
- `terms()` (`sympy.polys.rings.PolyElement` method), 2560
- `terms_gcd()` (in module `sympy.polys.polytools`), 2366
- `terms_gcd()` (`sympy.polys.polyclasses.DMP` method), 2567
- `terms_gcd()` (`sympy.polys.polytools.Poly` method), 2420
- `termwise()` (`sympy.polys.polytools.Poly` method), 2420
- `test()` (in module `sympy.testing.runtests`), 2035
- `test_derivative_numerically()` (in module `sympy.core.random`), 1078
- `test_factor()` (`sympy.polys.numberfields.primes.PrimeIdeal` method), 2709
- TextBackend (class in `sympy.plotting.plot`), 2869
- `textplot()` (in module `sympy.plotting.textplot`), 2872
- TGate (class in `sympy.physics.quantum.gate`), 1845
- ThinLens (class in `sympy.physics.optics.gaussopt`), 1865
- `threaded()` (in module `sympy.utilities.decorator`), 2060
- `threaded_factory()` (in module `sympy.utilities.decorator`), 2060
- `time` (`sympy.physics.quantum.state.TimeDepState` property), 1836
- `time_derivative()` (in module `sympy.physics.vector.functions`), 1669
- `time_period` (`sympy.physics.optics.waves.TWave` property), 1889
- `timed()` (in module `sympy.utilities.timeutils`), 2116
- TimeDepBra (class in `sympy.physics.quantum.state`), 1834
- TimeDepKet (class in `sympy.physics.quantum.state`), 1835
- TimeDepState (class in `sympy.physics.quantum.state`), 1836
- tiny (`sympy.codegen.ast.FloatType` property), 1132
- `to_alg_num()` (`sympy.polys.domains.AlgebraicField` method), 2545
- `to_alg_num()` (`sympy.polys.numberfields.modules.PowerBasisElement` method), 2732
- `to_algebraic_integer()` (`sympy.core.numbers.AlgebraicNumber` method), 992
- `to_ancestor()` (`sympy.polys.numberfields.modules.Module` method), 2731
- `to_anf()` (in module `sympy.logic.boolalg`), 1172
- `to_ANP()` (`sympy.polys.numberfields.modules.PowerBasisElement` method), 2732
- `to_axis_angle()` (`sympy.algebras.Quaternion` method), 896
- `to_cnf()` (in module `sympy.logic.boolalg`), 1173
- `to_ddm()` (`sympy.polys.matrices.sdm.SDM` method), 2698
- `to_dense()` (`sympy.polys.matrices.domainmatrix.DomainMatrix` method), 2687
- `to_dict()` (`sympy.polys.polyclasses.ANP` method), 2569
- `to_dict()` (`sympy.polys.polyclasses.DMP` method), 2567
- `to_dnf()` (in module `sympy.logic.boolalg`), 1173
- `to_exact()` (`sympy.polys.polyclasses.DMP` method), 2567
- `to_exact()` (`sympy.polys.polytools.Poly` method), 2421
- `to_expr()` (`sympy.holonomic.holonomic.HolonomicFunction` method), 2323
- `to_expr()` (`sympy.physics.control.lti.Parallel`



`method`), 1906  
`to_expr()` (`sympy.physics.control.lti.Series` `method`), 1902  
`to_expr()` (`sympy.physics.control.lti.TransferFunction` `method`), 1898  
`to_field()` (`sympy.polys.matrices.domainmatrix.DomainMatrix` `method`), 2687  
`to_field()` (`sympy.polys.polyclasses.DMP` `method`), 2567  
`to_field()` (`sympy.polys.polytools.Poly` `method`), 2421  
`to_hyper()` (`sympy.holonomic.holonomic.HolonomicFunction` `method`), 2321  
`to_int_repr()` (in module `sympy.logic.boolalg`), 1183  
`to_linearizer()` (`sympy.physics.mechanics.kane.KaneMethod` `method`), 1767  
`to_linearizer()` (`sympy.physics.mechanics.lagrange.LagrangeMethod` `method`), 1770  
`to_list()` (`sympy.polys.matrices.sdm.SDM` `method`), 2699  
`to_list()` (`sympy.polys.polyclasses.ANP` `method`), 2569  
`to_list()` (`sympy.polys.polyclasses.DMP` `method`), 2567  
`to_matrix()` (`sympy.physics.vector.dyadic.Dyadic` `method`), 1650  
`to_matrix()` (`sympy.physics.vector.vector.Vector` `method`), 1646  
`to_Matrix()` (`sympy.polys.matrices.domainmatrix.DomainMatrix` `method`), 2686  
`to_matrix()` (`sympy.vector.dyadic.Dyadic` `method`), 1459  
`to_matrix()` (`sympy.vector.vector.Vector` `method`), 1457  
`to_meijerg()` (`sympy.holonomic.holonomic.HolonomicFunction` `method`), 2322  
`to_nnf()` (in module `sympy.logic.boolalg`), 1173  
`to_number_field()` (in module `sympy.polys.numberfields.subfield`), 2714  
`to_parent()` (`sympy.polys.numberfields.modules.SympyElement` `method`), 2731  
`to_primitive_element()` (`sympy.core.numbers.AlgebraicNumber` `static method`), 992  
`to_prufer()` (`sympy.combinatorics.prufer.Prufer` `static method`), 338  
`to_rational()` (`sympy.polys.domains.RealField` `method`), 2546  
`to_ring()` (`sympy.polys.polyclasses.DMP` `method`), 2567  
`to_ring()` (`sympy.polys.polytools.Poly` `method`), 2421  
`to_root()` (`sympy.core.numbers.AlgebraicNumber` `method`), 993  
`to_rotation_matrix()` (`sympy.algebras.Quaternion` `method`), 896  
`to_sequence()` (`sympy.holonomic.holonomic.HolonomicFunction` `method`), 2319  
`to_sparse()` (`sympy.polys.matrices.domainmatrix.DomainMatrix` `method`), 2687  
`to_sympy()` (`sympy.polys.domains.AlgebraicField` `method`), 2545  
`to_sympy()` (`sympy.polys.domains.ComplexField` `method`), 2546  
`to_sympy()` (`sympy.polys.domains.domain.Domain` `method`), 2517  
`to_sympy()` (`sympy.polys.domains.ExpressionDomain` `method`), 2545  
`to_sympy()` (`sympy.polys.domains.FiniteField` `method`), 2525  
`to_sympy()` (`sympy.polys.domains.FractionField` `method`), 2549  
`to_sympy()` (`sympy.polys.domains.gaussiandomains.GaussianDomain` `method`), 2533  
`to_sympy()` (`sympy.polys.domains.GMPYIntegerRing` `method`), 2529  
`to_sympy()` (`sympy.polys.domains.GMPYRationalField` `method`), 2532  
`to_sympy()` (`sympy.polys.domains.IntegerRing` `method`), 2528  
`to_sympy()` (`sympy.polys.domains.PolynomialRing` `method`), 2548  
`to_sympy()` (`sympy.polys.domains.RationalField` `method`), 2531  
`to_sympy()` (`sympy.polys.domains.RealField` `method`), 2546  
`to_sympy_dict()` (`sympy.polys.polyclasses.ANP` `method`), 2569  
`to_sympy_dict()` (`sympy.polys.polyclasses.DMP` `method`), 2567  
`to_sympy_list()` (`sympy.polys.polyclasses.ANP` `method`), 2569  
`to_sympy_list()` (`sympy.polys.polyclasses.DMP` `method`), 2567  
`to_tree()` (`sympy.combinatorics.prufer.Prufer` `static method`), 338  
`to_tuple()` (`sympy.polys.polyclasses.ANP` `method`), 2569  
`to_tuple()` (`sympy.polys.polyclasses.DMP` `method`), 2567  
`todod()` (`sympy.matrices.common.MatrixCommon` `method`), 1354  
`todok()` (`sympy.matrices.common.MatrixCommon` `method`), 1354

- together() (in module `sympy.polys.rationaltools`), 2442  
 together() (`sympy.core.expr.Expr` method), 975  
 Token (class in `sympy.codegen.ast`), 1137  
 tolist() (`sympy.matrices.common.MatrixCommon` method), 1355  
 topological\_sort() (in module `sympy.utilities.iterables`), 2096  
 topological\_sort() (`sympy.codegen.ast.CodeBlock` class method), 1129  
 torsional\_moment() (`sympy.physics.continuum_mechanics.beams.Beam3D` method), 1998  
 total\_degree() (`sympy.polys.polyclasses.DMP` method), 2567  
 total\_degree() (`sympy.polys.polytools.Poly` method), 2421  
 totient (class in `sympy.ntheory.factor_`), 1501  
 totientrange() (`sympy.ntheory.generate.Sieve` method), 1478  
 tp (`sympy.polys.domains.domain.Domain` property), 2519  
 TR0() (in module `sympy.simplify.fu`), 704  
 TR1() (in module `sympy.simplify.fu`), 704  
 TR10() (in module `sympy.simplify.fu`), 707  
 TR10i() (in module `sympy.simplify.fu`), 707  
 TR11() (in module `sympy.simplify.fu`), 708  
 TR11i() (in module `sympy.simplify.fu`), 711  
 TR12() (in module `sympy.simplify.fu`), 708  
 TR12i() (in module `sympy.simplify.fu`), 709  
 TR13() (in module `sympy.simplify.fu`), 709  
 TR14() (in module `sympy.simplify.fu`), 711  
 TR15() (in module `sympy.simplify.fu`), 711  
 TR16() (in module `sympy.simplify.fu`), 711  
 TR2() (in module `sympy.simplify.fu`), 704  
 TR22() (in module `sympy.simplify.fu`), 712  
 TR2i() (in module `sympy.simplify.fu`), 704  
 TR3() (in module `sympy.simplify.fu`), 705  
 TR4() (in module `sympy.simplify.fu`), 705  
 TR5() (in module `sympy.simplify.fu`), 705  
 TR6() (in module `sympy.simplify.fu`), 706  
 TR7() (in module `sympy.simplify.fu`), 706  
 TR8() (in module `sympy.simplify.fu`), 706  
 TR9() (in module `sympy.simplify.fu`), 707  
 Trace (class in `sympy.matrices.expressions`), 1375  
 trace() (`sympy.matrices.common.MatrixCommon` method), 1355  
 trailing() (in module `sympy.ntheory.factor_`), 1488  
 transcendental, 924  
 TranscendentalPredicate (class in `sympy.assumptions.predicates.sets`), 232  
 TransferFunction (class in `sympy.physics.control.lti`), 1891  
 TransferFunctionMatrix (class in `sympy.physics.control.lti`), 1910  
 transform() (`sympy.diffgeom.CoordSystem` method), 2805  
 transform() (`sympy.geometry.point.Point2D` method), 2210  
 transform() (`sympy.geometry.point.Point3D` method), 2214  
 transform3D() (`sympy.integrals.integrals.Integral` method), 604  
 transform() (`sympy.polys.polyclasses.DMP` method), 2567  
 transform() (`sympy.polys.polytools.Poly` method), 2422  
 transform\_variable (`sympy.integrals.transforms.IntegralTransform` property), 587  
 transformation() (`sympy.diffgeom.CoordSystem` method), 2805  
 transformation\_to\_DN() (in module `sympy.solvers.diophantine.diophantine`), 727  
 transformation\_to\_normal() (in module `sympy.solvers.diophantine.diophantine`), 728  
 transition\_probabilities (`sympy.stats.DiscreteMarkovChain` property), 2951  
 transitivity\_degree (`sympy.combinatorics.perm_groups.PermutationGroup` property), 331  
 translate() (in module `sympy.utilities.misc`), 2114  
 translate() (`sympy.geometry.curve.Curve` method), 2252  
 translate() (`sympy.geometry.entity.GeometryEntity` method), 2197  
 translate() (`sympy.geometry.point.Point2D` method), 2210  
 translate() (`sympy.geometry.point.Point3D` method), 2214  
 transmissive\_filter() (in module `sympy.physics.optics.polarization`), 1879  
 transpose (class in `sympy.matrices.expressions`), 1375  
 transpose() (`sympy.matrices.common.MatrixCommon` method), 1355  
 transpose() (`sympy.matrices.expressions.blockmatrix.BlockMatrix` method), 1375

method), 1384  
 transpose() (sympy.physics.control.lti.TransferFunctionMatrix method), 1922  
 transpose() (sympy.polys.matrices.domainmatrix.DomainMatrix method), 2688  
 transpose() (sympy.polys.matrices.sdm.SDM method), 2699  
 transpositions() (sympy.combinatorics.permutations.Permutation method), 283  
 transverse\_magnification() (in module sympy.physics.optics.utils), 1886  
 Trapezoidal() (in module sympy.stats), 2926  
 tree() (in module sympy.printing.tree), 2179  
 tree\_cse() (in module sympy.simplify.cse\_main), 687  
 tree\_repr (sympy.combinatorics.prufer.Prufer property), 339  
 Triangle (class in sympy.geometry.polygon), 2293  
 Triangular() (in module sympy.stats), 2927  
 TriangularPredicate (class in sympy.assumptions.predicates.matrices), 214  
 tribonacci (class in sympy.functions.combinatorial.numbers), 438  
 TribonacciConstant (class in sympy.core.numbers), 1003  
 trigamma (class in sympy.functions.special.gamma\_functions), 465  
 trigintegrate() (in module sympy.integrals.trigonometry), 589  
 trigsimp() (in module sympy.simplify.trigsimp), 679  
 trigsimp() (sympy.core.expr.Expr method), 975  
 TRmorrie() (in module sympy.simplify.fu), 709  
 TRpower() (in module sympy.simplify.fu), 712  
 trunc() (in module sympy.polys.polytools), 2369  
 trunc() (sympy.polys.polyclasses.DMP method), 2567  
 trunc() (sympy.polys.polytools.Poly method), 2422  
 truncate() (sympy.series.formal.FormalPowerSeries method), 650  
 truncate() (sympy.series.fourier.FourierSeries method), 643  
 Truss (class in sympy.physics.continuum\_mechanics.truss.Truss), 2021  
 truth\_table() (in module sympy.logic.boolalg), 1179  
 Tuple (class in sympy.core.containers), 1068  
 TupleMatrix (class in sympy.core.containers.Tuple method), 1069  
 TupleKind (class in sympy.core.containers), 1069  
 Twave (class in sympy.physics.optics.waves), 1887  
 twoqubit\_permutation() (sympy.physics.quantum.circuitplot.CircuitDiagramme method), 1841  
 twoform\_to\_matrix() (in module sympy.diffgeom), 2818  
 TwoQubitGate (class in sympy.physics.quantum.gate), 1845  
 Type (class in sympy.codegen.ast), 1138  
 TypeA (class in sympy.liealgebras.type\_a), 2328  
 TypeB (class in sympy.liealgebras.type\_b), 2330  
 TypeC (class in sympy.liealgebras.type\_c), 2331  
 TypeD (class in sympy.liealgebras.type\_d), 2333  
 TypeE (class in sympy.liealgebras.type\_e), 2334  
 TypeF (class in sympy.liealgebras.type\_f), 2336  
 TypeG (class in sympy.liealgebras.type\_g), 2337  
 U (class in sympy.physics.mechanics.jointsmethod.JointsMethod property), 1783  
 U() (in module sympy.printing.pretty.pretty\_symbology), 2184  
 udivisor\_count() (in module sympy.ntheory.factor\_), 1499  
 udivisor\_sigma (class in sympy.ntheory.factor\_), 1503  
 udivisors() (in module sympy.ntheory.factor\_), 1499  
 UDLdecomposition() (sympy.matrices.expressions.blockmatrix.BlockMatrix method), 1382  
 ufuncify() (in module sympy.utilities.autowrap), 2043  
 UfuncifyCodeWrapper (class in sympy.utilities.autowrap), 2040  
 UGate (class in sympy.physics.quantum.gate), 1845  
 uncouple() (in module sympy.physics.quantum.spin), 1830  
 UndefinedKind (in module sympy.core.kind), 1074



UndefinedPredicate (class in `sympy.assumptions.assume`), 196  
 Unequality (class in `sympy.core.relational`), 1031  
 UnevaluatedExpr (class in `sympy.core.expr`), 975  
 unflatten() (in module `sympy.utilities.iterables`), 2098  
 UnificationFailed (class in `sympy.polys.polyerrors`), 2644  
 Uniform() (in module `sympy.stats`), 2928  
 UniformSum() (in module `sympy.stats`), 2929  
 unify() (`sympy.polys.domains.domain.Domain` method), 2519  
 unify() (`sympy.polys.matrices.domainmatrix.DomainMatrix` method), 2688  
 unify() (`sympy.polys.numberfields.modules.ModuleElement` method), 2731  
 unify() (`sympy.polys.polyclasses.ANP` method), 2569  
 unify() (`sympy.polys.polyclasses.DMP` method), 2567  
 unify() (`sympy.polys.polytools.Poly` method), 2422  
 union (class in `sympy.codegen.cnodes`), 1149  
 Union (class in `sympy.sets.sets`), 1197  
 union() (`sympy.polys.agca.ideals.Ideal` method), 2464  
 union() (`sympy.polys.agca.modules.SubModule` method), 2461  
 union() (`sympy.series.gruntz.SubSet` method), 625  
 union() (`sympy.sets.sets.Set` method), 1193  
 uniq() (in module `sympy.utilities.iterables`), 2098  
 unit (`sympy.geometry.point.Point` property), 2208  
 unit (`sympy.polys.polytools.Poly` property), 2423  
 UnitaryOperator (class in `sympy.physics.quantum.operator`), 1812  
 UnitaryPredicate (class in `sympy.assumptions.predicates.matrices`), 206  
 UnitSystem (class in `sympy.physics.units.unitsystem`), 1587  
 UnitTriangularPredicate (class in `sympy.assumptions.predicates.matrices`), 215  
 Univariate (class in `sympy.solvers.diophantine.diophantine`), 746  
 UnivariatePolynomialError (class in `sympy.polys.polyerrors`), 2644  
 UniversalSet (class in `sympy.sets.sets`), 1202  
 unrad() (in module `sympy.solvers.solvers`), 851  
 unrank() (`sympy.combinatorics.graycode.GrayCode` class method), 350  
 unrank() (`sympy.combinatorics.prufer.Prufer` class method), 339  
 unrank\_binary() (`sympy.combinatorics.subsets.Subset` class method), 346  
 unrank\_gray() (`sympy.combinatorics.subsets.Subset` class method), 346  
 unrank\_lex() (`sympy.combinatorics.permutations.Permutation` class method), 283  
 unrank\_nonlex() (`sympy.combinatorics.permutations.Permutation` class method), 284  
 unrank\_trotterjohnson() (`sympy.combinatorics.permutations.Permutation` class method), 284  
 UnsignedIntType (class in `sympy.codegen.ast`), 1140  
 update() (`sympy.physics.quantum.circuitplot.CircuitPlot` method), 1841  
 upper (`sympy.physics.secondquant.AntiSymmetricTensor` property), 1545  
 upper (`sympy.tensor.indexed.Idx` property), 1401  
 upper\_hessenberg\_decomposition() (`sympy.matrices.matrices.MatrixBase` method), 1317  
 upper\_triangular() (`sympy.matrices.common.MatrixCommon` method), 1356  
 upper\_triangular\_solve() (`sympy.matrices.dense.DenseMatrix` method), 1363  
 upper\_triangular\_solve() (`sympy.matrices.matrices.MatrixBase` method), 1318  
 uppgamma (class in `sympy.functions.special.gamma_functions`), 466  
 UpperTriangularPredicate (class in `sympy.assumptions.predicates.matrices`), 208  
 use (class in `sympy.codegen.fnodes`), 1156  
 use() (in module `sympy.core.traversal`), 1082  
 use\_rename (class in `sympy.codegen.fnodes`), 1156  
 using (class in `sympy.codegen.cxxnodes`), 1149

## V

- `v1pt_theory()` (sympy.physics.vector.point.Point method), 1657
- `v2pt_theory()` (sympy.physics.vector.point.Point method), 1658
- `valuation()` (sympy.polys.numberfields.primes.PrimeIdeal method), 2709
- `values()` (sympy.core.containers.Dict method), 1071
- `values()` (sympy.matrices.common.MatrixCommon method), 1357
- `var` (sympy.physics.control.lti.Feedback property), 1910
- `var` (sympy.physics.control.lti.MIMOFeedback property), 1932
- `var` (sympy.physics.control.lti.MIMOParallel property), 1927
- `var` (sympy.physics.control.lti.MIMOSeries property), 1924
- `var` (sympy.physics.control.lti.Parallel property), 1906
- `var` (sympy.physics.control.lti.Series property), 1903
- `var` (sympy.physics.control.lti.TransferFunction property), 1899
- `var` (sympy.physics.control.lti.TransferFunctionMatrix property), 1922
- `var()` (in module sympy.core.symbol), 980
- `VarBosonicBasis` (class in sympy.physics.secondquant), 1560
- `Variable` (class in sympy.codegen.ast), 1140
- `variable` (sympy.physics.continuum\_mechanics.beam.Beam property), 1981
- `variable_map()` (sympy.physics.vector.frame.ReferenceFrame method), 1641
- `variables` (sympy.concrete.expr\_with\_limits.ExprWithLimits property), 608
- `variables` (sympy.core.function.Lambda property), 1040
- `variables` (sympy.core.function.Subs property), 1053
- `variables` (sympy.physics.quantum.operator.DifferentialOperator property), 1808
- `variables` (sympy.physics.quantum.state.Wavefunction property), 1840
- `variables` (sympy.series.sequences.SeqBase property), 634
- `variables` (sympy.utilities.codegen.Routine property), 2052
- `Variance` (class in sympy.stats), 2966
- `variance()` (in module sympy.stats), 2965
- `VarianceMatrix` (class in sympy.stats), 2979
- `variations()` (in module sympy.utilities.iterables), 2098
- `vec()` (sympy.matrices.common.MatrixCommon method), 1357
- `vech()` (sympy.matrices.common.MatrixCommon method), 1357
- `Vector` (class in sympy.physics.vector.vector), 1641
- `Vector` (class in sympy.vector.vector), 1455
- `vector_coplanar()` (sympy.algebras.Quaternion class method), 897
- `vector_integrate()` (in module sympy.vector.integrals), 1475
- `vector_part()` (sympy.algebras.Quaternion method), 898
- `vectorize` (class in sympy.core.multidimensional), 1039
- `vectors_in_basis()` (in module sympy.diffgeom), 2817
- `vel()` (sympy.physics.vector.point.Point method), 1658
- `verify_numerically()` (in module sympy.core.random), 1078
- `vertices` (sympy.combinatorics.polyhedron.Polyhedron property), 335
- `vertices` (sympy.geometry.polygon.Polygon property), 2284
- `vertices` (sympy.geometry.polygon.RegularPolygon property), 2293
- `vertices` (sympy.geometry.polygon.Triangle property), 2303
- `VF()` (in module sympy.printing.pretty.pretty\_symbology), 2185
- `vfield()` (in module sympy.polys.fields), 2560
- `viete()` (in module sympy.polys.polyfuncs), 2427
- `vlatex()` (in module sympy.physics.vector.printing), 1665
- `vobj()` (in module sympy.printing.pretty.pretty\_symbology), 2185
- `vonMises()` (in module sympy.stats), 2930
- `vpprint()` (in module sympy.physics.vector.printing), 1665
- `vprint()` (in module sympy.physics.vector.printing), 1664
- `vradius` (sympy.geometry.ellipse.Circle property), 2272
- `vradius` (sympy.geometry.ellipse.Ellipse property), 2269
- `vring()` (in module sympy.polys.rings), 2552
- `vstack()` (sympy.matrices.common.MatrixCommon class method), 1358
- `vstack()` (sympy.polys.matrices.ddm.DDM

- method), 2691
- vstack() (sympy.polys.matrices.domainmatrix.DomainMatrix), 1794
- method), 2689
- vstack() (sympy.polys.matrices.sdm.SDM), 1794
- method), 2699
- ## W
- w (sympy.physics.optics.gaussopt.BeamParameter property), 1859
- w\_0 (sympy.physics.optics.gaussopt.BeamParameter property), 1859
- waist2rayleigh() (in module sympy.physics.optics.gaussopt), 1869
- waist\_approximation\_limit (sympy.physics.optics.gaussopt.BeamParameter property), 1860
- Wald() (in module sympy.stats), 2931
- walk() (in module sympy.core.traversal), 1082
- warns() (in module sympy.testing.pytest), 2028
- warns\_deprecated\_sympy() (in module sympy.testing.pytest), 2029
- Wavefunction (class in sympy.physics.quantum.state), 1836
- wavelength (sympy.physics.optics.waves.TWave property), 1889
- wavenumber (sympy.physics.optics.waves.TWave property), 1890
- weak (sympy.core.relational.Relational property), 1022
- WedgeProduct (class in sympy.diffgeom), 2812
- Weibull() (in module sympy.stats), 2932
- WeylGroup (class in sympy.liealgebras.weyl\_group), 2338
- WGate (class in sympy.physics.quantum.grover), 1848
- where() (in module sympy.stats), 2965
- While (class in sympy.codegen.ast), 1142
- whole\_submodule() (sympy.polys.numberfields.modules.Module method), 2725
- wicks() (in module sympy.physics.secondquant), 1566
- width (sympy.categories.diagram\_drawing.Diagram property), 2755
- width() (sympy.printing.pretty.stringpict.StringPict method), 2188
- WienerProcess (class in sympy.stats), 2955
- Wigner3j (class in sympy.physics.quantum.cg), 1793
- Wigner6j (class in sympy.physics.quantum.cg), 1794
- Wigner9j (class in sympy.physics.quantum.cg), 1794
- wigner\_3j() (in module sympy.physics.wigner), 1571
- wigner\_6j() (in module sympy.physics.wigner), 1572
- wigner\_9j() (in module sympy.physics.wigner), 1573
- wigner\_d() (in module sympy.physics.wigner), 1574
- wigner\_d\_small() (in module sympy.physics.wigner), 1575
- WignerD (class in sympy.physics.quantum.spin), 1828
- WignerSemicircle() (in module sympy.stats), 2933
- Wild (class in sympy.core.symbol), 976
- WildFunction (class in sympy.core.function), 1040
- wilkinson() (sympy.matrices.common.MatrixCommon class method), 1358
- Wishart() (in module sympy.stats), 2958
- write() (sympy.testing.runtests.PyTestReporter method), 2030
- write() (sympy.utilities.codegen.CodeGen method), 2049
- wronskian() (in module sympy.matrices.dense), 1321
- ## X
- X (in module sympy.physics.quantum.gate), 1846
- x (sympy.geometry.point.Point2D property), 2211
- x (sympy.geometry.point.Point3D property), 2215
- x (sympy.physics.mechanics.body.Body property), 1763
- x (sympy.physics.vector.frame.ReferenceFrame property), 1641
- XGate (class in sympy.physics.quantum.cartesian), 1802
- xdirection (sympy.geometry.line.Ray2D property), 2241
- xdirection (sympy.geometry.line.Ray3D property), 2245
- XField() (in module sympy.polys.fields), 2560
- XGate (class in sympy.physics.quantum.gate), 1846
- XKet (class in sympy.physics.quantum.cartesian), 1802

Xnor (class in `sympy.logic.boolalg`), 1170  
xobj() (in module `sympy.printing.pretty.pretty_symbology`), 2185  
XOp (class in `sympy.physics.quantum.cartesian`), 1802  
Xor (class in `sympy.logic.boolalg`), 1168  
xreplace() (`sympy.core.basic.Basic` method), 943  
xreplace() (`sympy.matrices.common.MatrixCommon` method), 1359  
xreplace() (`sympy.physics.vector.dyadic.Dyadic` method), 1651  
xreplace() (`sympy.physics.vector.vector.Vector` method), 1647  
xring() (in module `sympy.polys.rings`), 2552  
xstr() (in module `sympy.printing.pretty.pretty_symbology`), 2184  
xsym() (in module `sympy.printing.pretty.pretty_symbology`), 2185  
xthreaded() (in module `sympy.utilities.decorator`), 2060  
xypic\_draw\_diagram() (in module `sympy.categories.diagram_drawing`), 2760  
XypicDiagramDrawer (class in `sympy.categories.diagram_drawing`), 2758  
  
Y  
Y (in module `sympy.physics.quantum.gate`), 1846  
y (`sympy.geometry.point.Point2D` property), 2211  
y (`sympy.geometry.point.Point3D` property), 2215  
y (`sympy.physics.mechanics.body.Body` property), 1763  
y (`sympy.physics.vector.frame.ReferenceFrame` property), 1641  
y (`sympy.series.sequences.RecursiveSequence` property), 640  
ydirection (`sympy.geometry.line.Ray2D` property), 2241  
ydirection (`sympy.geometry.line.Ray3D` property), 2246  
YGate (class in `sympy.physics.quantum.gate`), 1846  
yn (class in `sympy.functions.special.bessel`), 502  
yn (`sympy.series.sequences.RecursiveSequence` property), 640  
Ynm (class in `sympy.functions.special.spherical_harmonics`), 545  
Ynm\_c() (in module `sympy.functions.special.spherical_harmonics`), 547  
YOp (class in `sympy.physics.quantum.cartesian`), 1802  
YuleSimon() (in module `sympy.stats`), 2886  
  
Z  
Z (in module `sympy.physics.quantum.gate`), 1846  
z (`sympy.geometry.point.Point3D` property), 2215  
z (`sympy.physics.mechanics.body.Body` property), 1763  
z (`sympy.physics.vector.frame.ReferenceFrame` property), 1641  
zdirection (`sympy.geometry.line.Ray3D` property), 2246  
zero, 924  
Zero (class in `sympy.core.numbers`), 995  
zero (`sympy.polys.domains.domain.Domain` attribute), 2519  
zero (`sympy.polys.polytools.Poly` property), 2423  
zero() (`sympy.polys.numberfields.modules.Module` method), 2725  
ZeroMatrix (class in `sympy.matrices.expressions`), 1379  
ZeroPredicate (class in `sympy.assumptions.predicates.order`), 221  
zeros() (in module `sympy.matrices.dense`), 1319  
zeros() (`sympy.matrices.common.MatrixCommon` class method), 1359  
zeros() (`sympy.physics.control.lti.TransferFunction` method), 1899  
zeros() (`sympy.polys.matrices.domainmatrix.DomainMatrix` class method), 2689  
zeros() (`sympy.polys.matrices.sdm.SDM` class method), 2700  
zeta (class in `sympy.functions.special.zeta_functions`), 514  
Zeta() (in module `sympy.stats`), 2886  
ZGate (class in `sympy.physics.quantum.gate`), 1846  
Znm (class in `sympy.functions.special.spherical_harmonics`), 547  
ZOp (class in `sympy.physics.quantum.cartesian`), 1802