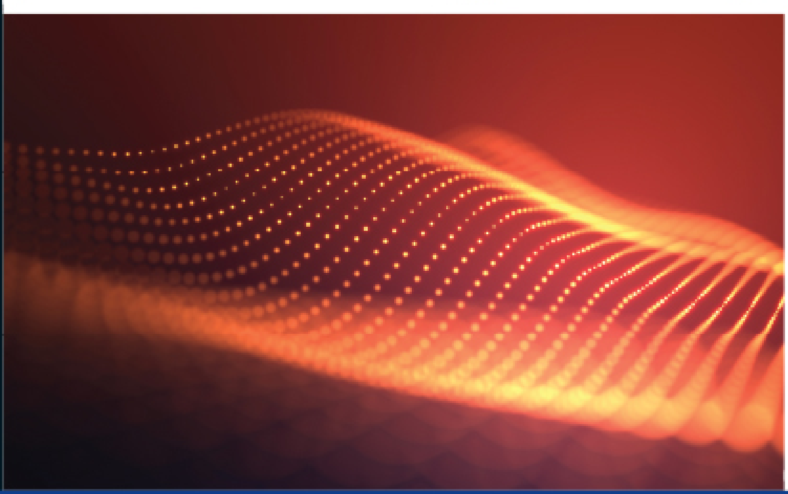
NumPy, SymPy
Matplotlib,
VPython, Sci
Structures,
Numerical ca
Linear equat
Vectors, Fur
Animations,



```
from sympy import *
x=symbols("x")
y=x**3-7*x**2+7*x+15
y_1=diff(y,x,1)
y_2=diff(y,x,2)
y_3=diff(y,x,3)
Y=integrate(y,x)
print("1. Derivation:",y_1)
print("2. Derivation:",y_2)
print("3. Derivation:",y_3)
print(" Integral :",Y)
```

Python for Engineering and Scientific Computing

Veit Steinkamp

Rheinwerk Computing

Veit Steinkamp

Python for Engineering and Scientific Computing

Rheinwerk
Computing

Imprint

This e-book is a publication many contributed to, specifically:

Editor Meagan White
Acquisitions Editor Hareem Shafi
German Edition Editor Christoph Meister
Translation Winema Language Services, Inc.
Copyeditor Yvette Chin
Cover Design Graham Geary
Photo Credit iStockphoto: 137993550/© OGphoto, 136159068/© cemagraphics
Layout Design Vera Brauner
Production E-Book Kelly O'Callaghan
Typesetting E-Book SatzPro, Germany

We hope that you liked this e-book. Please share your feedback with us and read the Service Pages to find out how to contact us.

Library of Congress Cataloging-in-Publication Control Number: 2024930350

ISBN 978-1-4932-2559-0 (print)
ISBN 978-1-4932-2560-6 (e-book)
ISBN 978-1-4932-2561-3 (print and e-book)

Notes on Usage

Contents

Chapter 1

Introduction

This chapter provides a brief overview of the extensibility, application areas, and functionality of the Python programming language.

If you need to perform extensive calculations for your scientific work and also want to present the results in a graphically appealing way, then you should seriously consider using Python. Python is a programming language whose functionality is similar to that of MATLAB when extended with appropriate modules. In addition, Python and all its extension modules are provided free of charge. Using Python, you can, for example, solve systems of equations, create function plots, differentiate, integrate, and also solve differential equations. You can also create *graphical user interfaces (GUIs)*. For almost every problem in engineering and natural sciences, solutions exist that not only cover a wide range of applications, but also excel in their user-friendliness and performance.

The Python programming language was developed in the early 1990s by Dutchman Guido van Rossum at *Centrum voor Wiskunde & Informatica (CWI)* in Amsterdam. Its name has nothing to do with the snake but refers instead to the British comedy group Monty Python.

The particular advantages and features of this programming language include the following:

- Python is an easy-to-learn and powerful programming language.
- It provides efficient data structures.
- It also allows object-oriented programming (OOP).
- It has a clear syntax and dynamic typing.
- Python programs are compiled using an interpreter and are therefore suitable for the rapid development of prototypes.
- Python is available for Linux, macOS, and Windows.
- Python can be extended by modules.

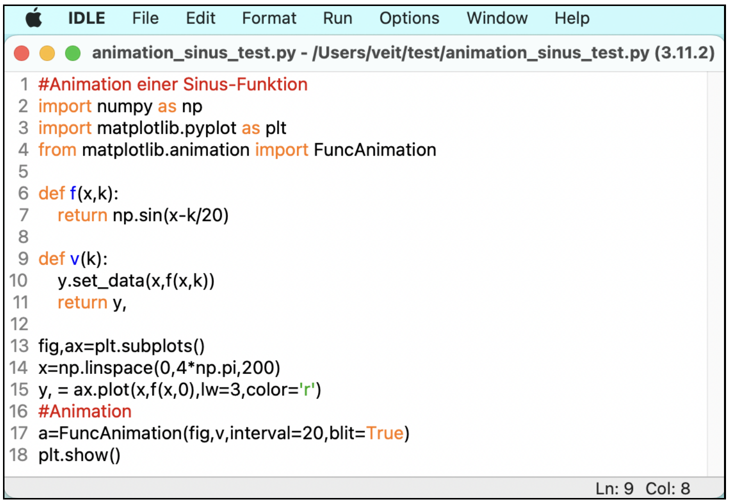
The module concept is the cornerstone and one of Python's outstanding strengths. A *module* is a component of a software system and represents a functionally self-contained unit that provides a specific service. For a definable scientific problem, a module that is tailored precisely to this problem is provided in each case. In this book, I will introduce you to the NumPy, Matplotlib, SymPy, SciPy, and VPython modules.

1.1 Development Environments

A *development environment* is a software program that consists of a text editor, a debugger, and an interpreter. The text editor of a development environment supports a programmer in writing programs, for example, with features like syntax highlighting, automatic indentation of the source code, and so on. The debugger helps programmers find errors, and the interpreter executes the program's statements. Of the many development environments that can be used to develop Python programs, only the Integrated Development and Learning Environment (IDLE), Thonny, and Spyder development environments will be briefly presented here.

1.1.1 IDLE

The abbreviation IDLE stands for "**I**ntegrated **D**evelopment and **L**earning **E**nvironment." Figure 1.1 shows the user interface for IDLE.


IDLE File Edit Format Run Options Window Help
animation_sinus_test.py - /Users/veit/test/animation_sinus_test.py (3.11.2)
Ln: 9 Col: 8

```
#Animation einer Sinus-Funktion
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

def f(x,k):
    return np.sin(x-k/20)

def v(k):
    y.set_data(x,f(x,k))
    return y,

fig,ax=plt.subplots()
x=np.linspace(0,4*np.pi,200)
y, = ax.plot(x,f(x,0),lw=3,color='r')
#Animation
a=FuncAnimation(fig,v,interval=20,blit=True)
plt.show()
```

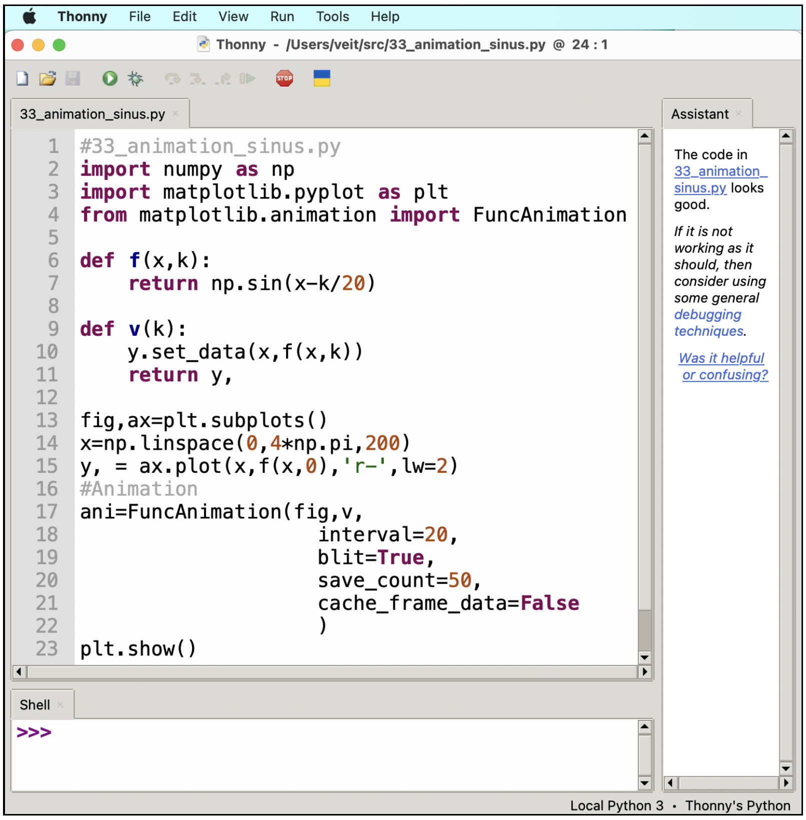
Figure 1.1 The IDLE Development Environment

IDLE is part of the standard Python download. During the installation of Python, IDLE is installed at the same time as the Pip package manager. You can download the latest version of Python for the Linux, macOS, and Windows operating systems at *https://www.python.org/downloads/*. Then, you'll need to install the NumPy, Matplotlib, SymPy, SciPy, and VPython modules individually using the Pip package manager (Section 1.1.4). This step may cause problems if you install a new Python version: The modules can no longer be imported with the new IDLE version, and the programs will no longer run. I will show you a way to fix this problem in Section 1.1.4. If the installation of the Python modules fails, I recommend you use the Thonny development environment.

When you click **Run • Python Shell**, the Python shell will open. Next to the >>> input prompt, you can directly enter Python commands or mathematical expressions, such as 2+3, 3*5, or 7/5. Note that you must complete each entry by pressing the Return key.

1.1.2 Thonny

Compared to the professional solutions, Thonny is a rather simply designed development environment with a comparatively small range of functions. However, it is particularly suitable for programming beginners due to its ease of use. Using Thonny, you can run and test all the sample programs discussed in this book. Figure 1.2 shows the user interface.



Thonny File Edit View Run Tools Help
Thonny - /Users/veit/src/33_animation_sinus.py @ 24 : 1
33_animation_sinus.py
Assistant
Shell
Local Python 3 • Thonny's Python

```
#33_animation_sinus.py
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

def f(x,k):
    return np.sin(x-k/20)

def v(k):
    y.set_data(x,f(x,k))
    return y,

fig,ax=plt.subplots()
x=np.linspace(0,4*np.pi,200)
y, = ax.plot(x,f(x,0),'r-',lw=2)
#Animation
ani=FuncAnimation(fig,v,
                    interval=20,
                    blit=True,
                    save_count=50,
                    cache_frame_data=False
                    )
plt.show()
```

The code in 33_animation_sinus.py looks good.

If it is not working as it should, then consider using some general debugging techniques.

Was it helpful or confusing?

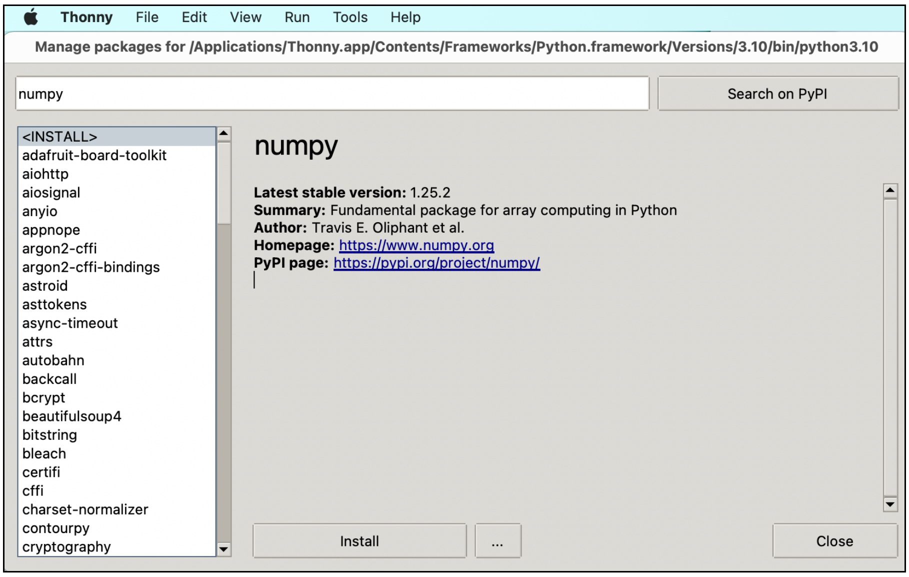
```
>>>
```

Figure 1.2 The Thonny Development Environment

Thonny is available for Linux, macOS, and Windows and can be downloaded at *https://thonny.org*.

The source code of the program must be entered into the text editor (upper left area). Once the program has been started via the F5 function key or by clicking the **Start** button, a window opens where you'll need to enter the file name of the program. The result of numerical calculations is then output in the **Command Line** window at the bottom left of the Python shell. Each function plot of Matplotlib programs will be output in a separate window. In the shell, also referred to as the *Python console*, you can also enter Python commands directly. The **Assistant** in the main window, on the right, supports you in terms of troubleshooting, although you should temper your expectations about its capabilities.

A particularly important feature of Thonny is that you can easily install and update the NumPy, Matplotlib, SymPy, SciPy, and VPython modules. For these tasks, all you need to do is open the **Tools • Manage Packages** dialog box, as shown in Figure 1.3. Then, in the text box in the top-left corner, enter the name of the module you want to install and click **Install** or **Update**.



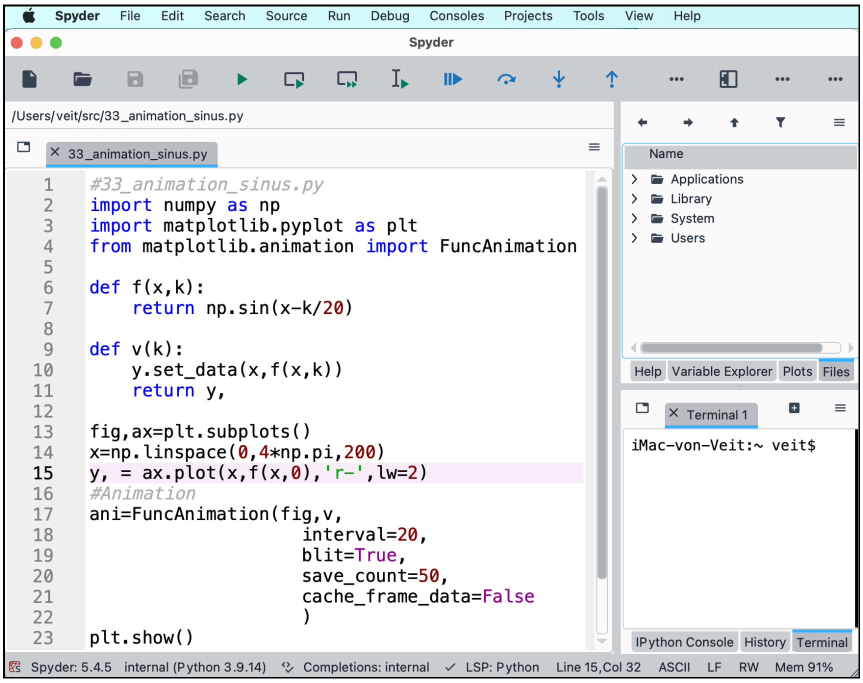
Latest stable version: 1.25.2
Summary: Fundamental package for array computing in Python
Author: Travis E. Oliphant et al.
Homepage: https://www.numpy.org
PyPI page: https://pypi.org/project/numpy/

Figure 1.3 Installing Modules

To remove a module, you must select the corresponding module in the pane on the left. Then, the **Uninstall** command button appears to the right of the **Install** command button. One notable advantage of the package manager in Thonny is that you can also test older versions of all available modules. For this task, simply click the **...** command icon to the right of the **Install** button, which will open a window where you can select the desired version of the module.

1.1.3 Spyder

Spyder is the development environment of the Anaconda distribution of Python. Except for VPython, the modules covered in this book—NumPy, Matplotlib, SymPy, and SciPy—are already built in.


Spyder File Edit Search Source Run Debug Consoles Projects Tools View Help
Spyder
/Users/veit/src/33_animation_sinus.py
33_animation_sinus.py
Name
Applications
Library
System
Users
Help Variable Explorer Plots Files
Terminal 1
IPython Console History Terminal
Spyder: 5.4.5 internal (Python 3.9.14) Completions: internal LSP: Python Line 15,Col 32 ASCII LF RW Mem 91%

```python
#33_animation_sinus.py
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

def f(x,k):
    return np.sin(x-k/20)

def v(k):
    y.set_data(x,f(x,k))
    return y,

fig,ax=plt.subplots()
x=np.linspace(0,4*np.pi,200)
y, = ax.plot(x,f(x,0),'r-',lw=2)
#Animation
ani=FuncAnimation(fig,v,
                          interval=20,
                          blit=True,
                          save_count=50,
                          cache_frame_data=False
                          )
plt.show()
```

```
iMac-von-Veit:~ veit$
```

Figure 1.4 The Spyder Development Environment

Spyder is available as a free download for Linux, macOS, and Windows at *https://www.spyder-ide.org*.

To run an animation using a Matplotlib program, you must select **Automatic** as the backend in the settings under **IPython Console • Graphics**. After starting the program, a separate window will open where the animation will run. Matplotlib programs containing slider controls can also be executed interactively only with this option.

Spyder is an immensely powerful development environment. However, one disadvantage is that the subsequent installation of modules that are not installed by default, such as VPython, can be difficult for beginners. For more information on installing Python modules, see the documentation for Spyder at *https://www.spyder-ide.org*.

1.1.4 Pip

To use development environments other than Thonny or Spyder, you can install Python modules using Pip. Pip is not a development environment but the package manager for Python that installs modules from the *Python Package Index (PyPI)* (*https://pypi.org/*). Pip allows you to download and update modules easily—when you use Python, Pip is a particularly important tool.

If you have installed Python and want to add only the NumPy module, for example, you can enter the following command in a terminal on Windows, Linux, or macOS:

```
pip install numpy
```

The following command enables you to update an existing NumPy installation:

```
pip install --upgrade numpy
```

If you use IDLE (e.g., version 3.9) and install a new version of Python (e.g., 3.11), then the previously installed Python modules will no longer be imported into the updated version. In this case, you should try installing via `pip3.11 install numpy`.

For more information about using Pip, see *https://pypi.org/project/pip*. If the installation or update of the Python modules fails, I recommend using the Thonny development environment instead.

1.2 The Modules of Python

For our first look at the capabilities and features of the module concept in Python, I first want to describe the five modules in a keyword-like manner. Instead of *module*, the terms *library* or *software library* are also commonly used. The capabilities of Python are best illustrated by using short sample programs. Of course, you don't need understand the source code shown in this section yet. After all, understanding is what the other chapters are for.

1.2.1 NumPy

The NumPy module (**num**erical **Py**thon) enables you to perform extensive numerical calculations. For example, you can solve linear systems of equations, even with complex numbers. Listing 1.1 shows a simple vector calculus program.

```
import numpy as np
A=np.array([1, 2, 3])
B=np.array([4, 5, 6])
print("Vector       A:",A)
print("Vector       B:",B)
print("Total       A+B:",A+B)
print("Product     A*B:",A*B)
print("Cross product :",np.cross(A,B))
print("Scalar product:",np.dot(A,B))
```

Listing 1.1 A NumPy Program

Output

```
Vector       A: [1 2 3]
Vector       B: [4 5 6]
```

```
Total     A+B: [5 7 9]
Product   A*B: [ 4 10 18]
Cross product : [-3  6 -3]
Scalar product: 32
```

The NumPy module is described in Chapter 3.

1.2.2 Matplotlib

The Matplotlib module allows you to display mathematical functions, histograms, and many other diagram types as well as to simulate and animate physical processes. The graphical design options are remarkably diverse and rich in detail. Listing 1.2 shows a simple example of the function plot of a polynomial.

```python
import numpy as np
import matplotlib.pyplot as plt
x=np.arange(-2,6,0.01)
y=x**3-7*x**2+7*x+15
plt.plot(x,y)
plt.show()
```

Listing 1.2 Function Plot with Matplotlib

Output

Figure 1.5 shows the output of the function plot.

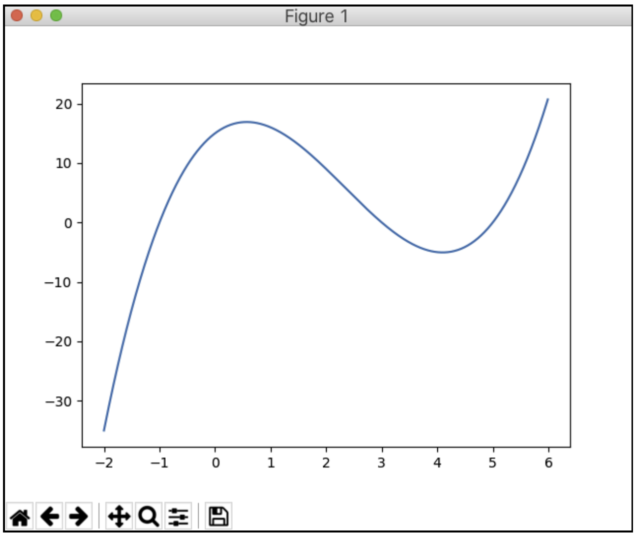


Figure 1.5 A Function Plot Created Using Matplotlib

The Matplotlib module is discussed in detail in Chapter 4.

1.2.3 SymPy

Using SymPy (**sym**bolic **Py**thon), you can calculate integrals or derivatives symbolically or solve differential equations symbolically. A simplification of mathematical terms is also possible (and much more). Listing 1.3 shows a simple example of symbolic differentiation and integration.

```python
from sympy import *
x=symbols("x")
y=x**3-7*x**2+7*x+15
y_1=diff(y,x,1)
y_2=diff(y,x,2)
y_3=diff(y,x,3)
Y=integrate(y,x)
print("1. Derivative:",y_1)
print("2. Derivative:",y_2)
print("3. Derivative:",y_3)
print("   Integral  :",Y)
```

Listing 1.3 Symbolic Differentiation and Integration Using SymPy

Output

```
1. Derivative: 3*x**2 - 14*x + 7
2. Derivative: 2*(3*x - 7)
3. Derivative: 6
   Integral  : x**4/4 - 7*x**3/3 + 7*x**2/2 + 15*x
```

The SymPy module is described in detail in Chapter 5.

1.2.4 SciPy

SciPy (**sci**entific **Py**thon) allows you to numerically differentiate, integrate, and numerically solve systems of differential equations. SciPy is as comprehensive as it is versatile. The capabilities of SciPy can only be partially described in this book. Listing 1.4 shows a simple example of a numerical integration program.

```python
import scipy.integrate as integral
def f(x):
    return x**2
A=integral.quad(f,0,5)
print("Area A=",A[0])
```

Listing 1.4 Numerical Integration Using SciPy

Output

```
Area A= 41.66666666666666
```

The SciPy module is described in Chapter 6.

1.2.5 VPython

Using VPython, you can display fields in a 3D view or even animate their movements in 3D space. As of version 7, the animations are displayed in the standard browser after the program starts. Listing 1.5 shows an example of how you can program the animation of a bouncing ball.

```
from vpython import *
r=1. #radius
h=5. #height
scene.background=color.white
scene.center=vector(0,h,0)
box(pos=vector(0,0,0),size=vector(2*h,r/2,h), color=color.green)
ball = sphere(radius=r, color=color.yellow)
ball.pos=vector(0,2*h,0) #drop height
ball.v = vector(0,0,0) #initial velocity
g=9.81
dt = 0.01
while True:
    rate(100)
    ball.pos = ball.pos + ball.v*dt
    if ball.pos.y < r:
        ball.v.y = -ball.v.y
    else:
        ball.v.y = ball.v.y - g*dt
```

Listing 1.5 Animation of a Bouncing Ball

Output

Figure 1.6 shows a snapshot of the animation. The VPython module is described in Chapter 7. Of course, not all the capabilities of the Python modules we've mentioned can be treated exhaustively in this book. If you miss a particular topic, I recommend referring to the online documentation as a supplemental source of information. A module's maintainers should have a website where you'll find tutorials for each module to get you started, including complete module descriptions.

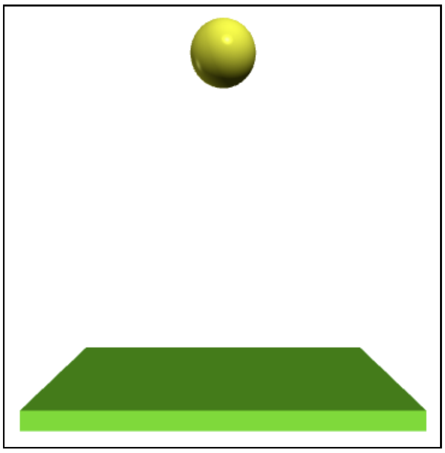


Figure 1.6 A Bouncing Ball Animation Created Using VPython

The chapters after Chapter 7 describe additional possible uses of the modules in greater detail with a focus on the practical application options.

Chapter 8 describes how you can compute alternating current (AC) electrical networks using the symbolic method. In the project assignment, you'll learn how to size an electrical power transmission system.

Chapter 9 focuses primarily on the simulation of a quality control chart. You'll learn how to generate normally distributed random numbers and save them to a file. This data is then read again to calculate its statistical characteristics, such as arithmetic mean and standard deviation.

Chapter 10 describes how to set up truth tables and simplify complex logical circuits using SymPy.

In Chapter 11, you'll learn how to program GUIs using Python. The project task shows you how to simulate simple control circuits.

1.3 The Keywords of Python

Whenever you learn a new programming language, the first thing you must know are the *keywords* defined in that language. Keywords are the reserved words of a programming language. They have a specific meaning in the programming language definition and must therefore not be used as variable names in a program. Python has 35 keywords, which you can view by entering the following commands in the Python console:

```
>>> import keyword
>>> a=keyword.kwlist
>>> print(a)
```

You'll receive the following output:

```
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class',
'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from',
'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass',
'raise', 'return', 'try', 'while', 'with', 'yield']
```

Similarly, the following statement will output the number of keywords:

```
>>> print(len(a))
35
```

You don't need to memorize all the keywords at first. For better overview of Python's keywords, a useful exercise is to first arrange them according to their functionalities. Table 1.1 provides an overview of the most important keywords arranged according to functional criteria.

Conditional Statements	Loops	Classes, Modules, and Functions	Error Handling
`if`	`for`	`class`	`try`
`else`	`in`	`def`	`except`
`elif`	`while`	`global`	`finally`
`not`	`break`	`lambda`	`raise`
`or`	`as`	`nonlocal`	`assert`
`and`	`continue`	`yield`	`with`
`is`		`import`	
`True`		`return`	
`False`		`from`	
`None`			

Table 1.1 The Most Important Keywords in Python

With a few keywords like `if`, `else`, `for`, and `while`, along with the built-in Python function `print()`, you can already write simple Python programs.

1.4 Your Path through This Book

How should you read this book? Basically, you can read the individual chapters independently of each other. If you already know the basic structures of Python, then you can skip Chapter 2. If just starting to learn, you must read that chapter first as a prerequisite for understanding subsequent chapters.

Our approach to presentation and knowledge transfer is based on a uniform principle: One to three examples from electrical engineering, mechanical engineering, or physics are presented for each topic. After a brief description of the task, the complete source code is printed. Directly after the source code, the output (the results of the calculations) takes place. The source code is then discussed and analyzed.

Our analysis of source code also includes an analysis of the results (output). Are the results in line with expectations? Does the program solve the task set for it at all? Often, you won't fully understand the source code of a program until you've taken a closer look at the output. After viewing the output, you can then analyze the source code again.

At the end of each chapter, one or more project assignments are provided, discussed, and fully solved to reinforce and expand on what was learned in that chapter.

Chapter 2
Program Structures

In this chapter, you'll get to know the linear program structure as well as the branching and repetition structures of the imperative programming style of Python. Examples of object-oriented and functional programming describe further ways to program using Python.

A *program* consists of a sequence of statements. A *statement* is a command that tells the interpreter (in this case, the Python interpreter) what actions the CPU is supposed to perform: for example, accept input, process the input, or output the processing results to the screen. These actions, which always run in the same way, are referred to as the *input-process-output (IPO) model* in computer science terminology.

Problems that are to be solved with a computer can be modeled and structured in a variety of ways by programming languages. In applied computer science, the imperative (procedural) programming, object-oriented programming (OOP), and functional programming styles have become established. Python supports all three programming styles, which I will introduce to you in this chapter.

Solutions to problems are also linked to logical conditions: Does the expected case apply or not? Also, depending on the problem, repetitions of the same tasks must be implemented, such as the calculation of value tables for mathematical functions. Like any other procedural programming language, Python supports the linear program structure as well as branching and repetition structures.

2.1 Linear Program Structures

In programs with the *linear flow structure*, calculations are performed in the order of the logic of the problem solution. Thus, calculation C cannot be performed until calculation B has been performed, and calculation B cannot be performed until calculation A has been performed first. So, you absolutely must retain the order of first A, then B, then C, because this order is mandatory for this type of problem. An example from the theory of motion illustrates this fact: If the acceleration of a vehicle is given, the velocity can be calculated from it, and the distance traveled can be calculated from the velocity.

2.1.1 Linear Programs without Function Calls

In real life, most programs are divided into *functions* (also referred to as *subroutines*), which I will explain in Section 2.2. As a rule, a program consists of multiple functions and one main program.

Before we consider this division of a program into several subsections, let's first examine the structure of a simple linear program. For many small, clearly defined problems, this program structure is already sufficient.

A linear program has the following general structure:

```
Statement1
Statement2
Statement3
...
```

The individual statements are translated and executed sequentially line by line by the Python interpreter. Branching and repetition do not occur.

Let's explore some basic concepts of programming such as *statement*, *assignment*, *variable*, *data type*, and *object* using the calculation of a simple power circuit with only one consumer. In such a circuit, the voltage U and the resistance R of the load are given. The program is supposed to calculate the current I, with the following formula:

$$I = \frac{U}{R}$$

In this simple case, the formula for calculating the current provides the development steps for designing the program. The inputs U and R are to the right, and the output is to the left of the equal sign of the formula.

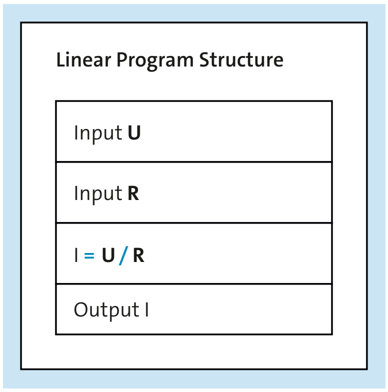


Figure 2.1 Structure Chart for a Linear Program Flow

For the design phase of program development, structure charts are particularly useful because they enable the description of the problem solution independently of the syntax of any particular programming language. In addition, they help you understand the problem and thus to find a suitable solution. Structure charts can be used to clearly illustrate the flow structures of programs. In our example, to calculate the current, you use the formula to create a structure chart with four statements, as shown in Figure 2.1.

You can implement this structure chart directly as Python source code by using the = assignment operator for the simple assignments and using the `print` function for the output. Formulas can also be transferred directly into the source code. Note that you can use the usual mathematical operators (/, *, +, and -). In addition, the quantity searched for must always be positioned to the left of the equal sign (the = assignment operator). For testing, enter the source code from Listing 2.1 into your development environment and start the program.

```
01  #!/usr/bin/env python3
02  #01_linear1.py
03  U=230
04  R=11.8
05  I=U/R
06  b="The current is:"
07  print(b, I, " A")
```

Listing 2.1 Linear Program Structure

Output

```
The current is: 19.491525423728813 A
```

Analysis

The program contains a total of five statements. In applied computer science, a *statement* is a syntactically related section of source code that tells the Python interpreter what action it is supposed to perform. In this sample program, a statement consists of one program line each.

In line 01, which is referred to as the *shebang*, the operating system is told which interpreter should be used for running the program. Due to the `env` specification, you don't need to specify the path of the directory where the Python interpreter is located. To run the program on Linux or macOS directly in the terminal using the `./01_linear1.py` command, you must first set the *executable flag* by using the `chmod +x 01_linear1.py` command.

On Windows, the statement in line 01 gets ignored. In all the following sample programs, the shebang is not specified.

In line 02, the filename of the `#01_linear1.py` program is written as a comment. This specification is useful so that the developer (and you as a learner) don't lose track of the numerous program examples. All lines preceded by a # character are ignored by the Python interpreter. Comments are used to explain statements of the source code in more detail.

Now, the actual program starts with a statement. Line 03 causes the `U` variable to be assigned the value `230`. In computer science, the term *variable* has a completely different meaning than in mathematics. For a simple understanding, think of the `U` variable as a symbolic address within the working memory where the number `230` is stored. At the same time, this *assignment* declares the `U` variable.

The equal sign in the context of Python has the meaning of an assignment, not that of a mathematical equal sign! An *assignment* is a type of statement that gives a variable a new value. The `U` variable automatically receives the *Integer* data type because the number `230` is an integer.

Value Range for Integer

Theoretically, the value range of the integer data type is not limited in the current Python version. For detailed information about the value ranges, refer to the Python documentation at *https://docs.python.org/3/reference/datamodel.html#the-standard-type-hierarchy*.

Thus, integers can consist of any number of digits. The internal designation for the Integer data type is `int`. In Python, variables are declared indirectly; that is, the first time a variable is used, its name and data type are made known to the interpreter during runtime. However, the data type can still change during runtime.

In line 04, the `R` variable of the *float* data type is declared because the *floating point number* `11.8` is assigned to it. This data type is the approximated representation of a real number.

The Float Data Type

In Python, the Float data type has a precision of 64 bits (*double precision*) according to the Institute of Electrical and Electronics Engineers (IEEE) 754 standard. The internal designation for the float data type is `float`, which corresponds to a range of values from about $2.225 \cdot 10^{-308}$ to $1.798 \cdot 10^{308}$. You can use the Python shell to determine the range of values of `float` with the following commands:

```
>>> import sys
>>> print(sys.float_info)
sys.float_info(max=1.7976931348623157e+308,max_exp=1024,
max_10_exp=308,min=2.2250738585072014e-308,min_exp=-1021,
min_10_exp=-307,dig=15,mant_dig=53,epsilon=2.220446049250313e-16,
radix=2, rounds=1)
```

In line 05, the `I` variable is declared, and at the same time, the current is calculated. Notice how you can also declare variables by assigning the result of a formula to them. As in other imperative programming languages, in Python, the division operation is also written using the / operator (*slash*). The result of the division of voltage and resistance is assigned to the I variable.

In line 06, the b variable of the *string* type is declared because a string is assigned to this variable. A string is a sequence of individual characters. The quotation marks tell the interpreter that this is a string variable. You can place any text between the quotation marks. The internal name for the string type in Python is str.

In line 07, the result of the calculation using the print function is output in an unformatted way. You can separate each output by using a comma. If the print function is supposed to output a string, such as the unit of current (A) in this example, the string must be enclosed in quotation marks.

What Are Objects?

Until this point, I've used the terms *variable* and *data type* in the same way as in the traditional imperative programming languages Pascal, C, C++, or Java. Strictly speaking, however, these conceptual borrowings from traditional programming languages do not apply to the Python programming language because, in Python, all variables, data types, data structures, and functions are *objects*. Let's explore exactly I mean with some examples using the Python shell. You'll get information about the type of each variable if you enter the following statement into the Python shell:

```
>>> U=230
>>> R=11.8
>>> I=U/R
>>> b="string"
>>> type(U)
<class 'int'>
>>> type(R)
<class 'float'>
>>> type(I)
<class 'float'>
>>> type(b)
<class 'str'>
```

You can use the built-in type() function in Python to determine the type of an object. The class keyword specifies the respective object type. The U object belongs to the int class, the R and I objects belong to the float class, and the b object belongs to the str class.

Each object is identified by a number. You can use the built-in id() function to determine these numbers (identities):

```
>>> id(U)
4505151824
```

```
>>> id(R)
4506525960
>>> id(I)
4506525888
>>> id(b)
4509028720
```

Each object is given its own integer as its identity, which is guaranteed to be unique and remains constant for the lifetime of the program. The identities themselves represent memory addresses in the working memory (RAM). Even if the type of an object should change during runtime, its identity (memory address) remains the same. Thus, an object has a name (identifier), a value, a type, and an identity, and it belongs to a certain class. If we continue to talk about variables, then what we actually mean is objects. A quote from the Python documentation should clarify this connection again:

> *Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects.... Every object has an identity, a type and a value. An object's identity never changes once it has been created; you may think of it as the object's address in memory. The "is" operator compares the identity of two objects; the id() function returns an integer representing its identity.*

Formatting the Output

The output for the current of Listing 2.1 has too many decimal places. Python provides the option to format floating point numbers, so that the outputs have decimal places that can be used in real life. As an example, just take a look at a series circuit with three resistors. Listing 2.2 shows how you can implement floating point number formatting in Python.

```python
#02_linear2.py
U=230
R1,R2,R3=0.12,0.52,228
Rg=R1+R2+R3
I=U/Rg
P1=R1*I**2
P2=R2*I**2
P3=R3*I**2
print("Current I={0:6.3f} A " .format(I))
print("P1={0:3.2f} W, P2={1:3.2f} W, P3={2:3.2f} W".format(P1,P2,P3))
#print("P1=%3.2f W, P2=%3.2f W, P3=%3.2f W" %(P1,P2,P3))
```

Listing 2.2 Formatting Output

Output

```
Current I= 1.006 A
P1=0.12 W, P2=0.53 W, P3=230.72 W
```

Analysis

The program calculates the total resistance of three resistors (line 04), the current, and the partial powers of the resistors. The assignment in line 03 is new. The `R1` resistor is assigned the value `0.12`; the `R2` resistor, the value `0.52`; and resistor `R3`, the value `228`. The declaration of the three variables is done simultaneously in one program line. The individual identifiers of the variables are separated by commas just like the values of these variables.

In lines 06 to 08, the current `I` is squared using the `**` operator. The Python interpreter considers the mathematical precedence rule. First, the current is squared and then multiplied by the resistances.

The formatting of the outputs is specified in lines 09 and 10. The curly brackets tell the interpreter to output the results in a formatted way. The number before the dot indicates the total number of characters (digits plus separators) of a floating point number. The number after the dot defines the number of decimal places. The output of the current therefore has three decimal places, while those of the partial powers have two decimal places each. The letter `f` stands for `float`. The number before the colon defines the position within the output. The `.format(I)` and `.format(P1,P2,P3)` statements cause the calculated results for the current and the partial powers to be output formatted. Alternatively, the output could have been formatted more simply using the syntax of line 11.

Interactive Input Using the input Function

Up to this point, all inputs were entered by means of static assignments. If you wanted to change the input values, the program would have to be restarted each time. Listing 2.3 shows how this deficiency can be remedied by using the built-in `input` function.

```python
#03_linear3.py
while True:
    print("\n---Input---")
    U=float(input("Voltage: "))
    R=float(input("Resistance: "))
    I=U/R
    P=U*I
    print("\n---Output---")
    print("Current {0:6.2f} A " .format(I))
    print("Power    {0:6.2f} W " .format(P))
```

Listing 2.3 Interactive Input

Output

```
---Input---
Voltage: 230
Resistance: 24

---Output---
Current   9.58 A
Power     2204.17 W
```

Analysis

The program starts with a `while` loop in line 02. This loop construct is introduced at this point because one-time console entries are just as useless as static assignments. Section 2.4.1 discusses the syntax of the `while` loop in greater detail. The `while` keyword is followed by the condition that must be met before the subsequent statements belonging to the *loop body* can be executed. The `while` statement must be terminated with a colon. The editor of a Python development environment automatically indents all subsequent statements that should be run repeatedly (with four spaces by default). This indentation is elementary because it tells the Python interpreter which statements belong to the loop body.

Indentations Are Important

In other programming languages, indentations are solely used to make a program more readable to humans, but actually the entire code could be placed in one line. Not so in Python! Indentations and spacing have syntactic meaning in Python, which means you must ensure that loops and branches are displayed correctly.

This requirement has the great advantage of producing readable code. If you look at a program again after a few months or read code written by colleagues, you'll be grateful for indentations.

Since the condition is always `True`, the loop is an infinite loop. The prompt can be interrupted by pressing the `Ctrl` + `C` shortcut (also often written as ^C) or by an incorrect input (no number or no input). I'll show you how to avoid this pretty inelegant programming style in Section 2.4.

The *escape sequence* `"\n"` in lines 03 and 08 causes a line break in each case. An escape sequence is a character combination that does not represent text but instead is a control statement that tells the computer how it should arrange the screen output. The `n` after the *backslash* (\) stands for **n**ewline.

In line 04, the built-in `input()` function first outputs the text specified in the quotes to the screen and then expects an input which must be terminated via `Return`. Each input is read as a string, converted to the `float` type via the built-in `float()` function and then assigned to the `U` variable.

2.2 Functions

If you place all the statements needed for the calculation of a complex task in a single coherent source code section (the main program), you'll lose track of your own work as the number of program lines increases. The development process itself and subsequent changes to the source code are thus unnecessarily complicated, if not impossible. In this context, the *subroutine technique* provides the option to break down complex problems into subproblems that are easy to master. This structuring option is available in every programming language. However, in the discourse about modern programming languages (C, C++, Java), the term *subroutine* is no longer common; instead, the term *function* is used. In general, in modern programming languages, a function is understood to be a structural element that combines a logically related set of instructions into a holistic unit.

The use of functions provides the following advantages:

- The source code of a program becomes clearer and is thus easier to understand.
- Troubleshooting (debugging) is simplified.
- Programs structured by functions are easier to maintain.
- Once written and tested, functions can be used by other programs.
- A function can be called at different places in the same program.
- A single function can be used for different calculations if the calculation rule for the different tasks has the same structural design. The calculation of kinetic energy, rotational energy, electrical energy, and magnetic energy provides a vivid example of this flexibility, as shown in Listing 2.12.

The general syntax of a function definition is:

```
def functionname(parameter1, parameter2, parameter3):
    statement1
    statement2
    statement3
      ...
    return value
```

The expressions in parentheses are called *parameters*.

Function

A Python function is a subroutine that solves a subproblem. A function definition consists of the function header and the function body. The function header is introduced with the keyword def, followed by the function name func(), which must end with parentheses. A colon marks the end of the function header.

The function body contains the individual statements. It ends with the `return` statement. In a function call such as `a=func()`, the calculated values are stored in the object, `a`.

2.2.1 Built-In Functions

Python provides a total of 68 built-in functions. We've already used some of them, such as `print()`, `input()`, and `type()`. An overview of some selected functions can be found in Table 2.1.

Function	Argument	Description
`abs()`	Integer, Float	Determines the absolute value of the argument.
`bin()`	Integer	Converts the argument to a binary string with the prefix `'0b'`.
`eval()`	String	Evaluates a string as a mathematical expression.
`float()`	Number or String	Converts the argument to a float object.
`hex()`	Integer	Converts the argument to a `hex` value with prefix `'0x'`.
`id()`	Object	Returns the integer value `identity` of the object.
`int()`	Number or String	Converts the argument to an integer object.
`input()`	String	Reads a string from standard input and returns it.
`print()`	Objects	Outputs values.
`range()`	Integer, Integer, Integer	Generates a list of integers.
`round()`	Float, Integer	Rounds a floating point number.
`type()`	Variable	Determines the type of a variable.

Table 2.1 Selection of Built-In Functions in Python

You may wonder why the second column in Table 2.1 does not contain the word "parameter" but "argument" instead. Applied computer science distinguishes between the term *argument* and the term *parameter*. An argument is a value that is passed when the function is called. This value is assigned to the assigned parameters within the function. A parameter is a name that is used within the function.

You can find documentation for all built-in functions at *https://docs.python.org/3.12/library/functions.html*.

2.2.2 Functions without Parameters and without Return Values

The first example shown in Listing 2.4 illustrates how to use functions in a circuit with one load to calculate the current, electrical power, electrical work, and cost of electrical energy:

```python
#04_function1.py
U,R = 230,460
t=8
price=0.3

def current():
    I=U/R
    print("Current: ", I, " A")

def power():
    P=U**2/R
    print("Power : ", P, " W")

def work():
    P=U**2/R
    W=P*t
    print("Work: ", W, " Wh")

def cost():
    I=U/R
    W=U*I*t
    c=W*price/1000.0
    print("Cost: ", c, " Euro")

current()
power()
work()
cost()
```

Listing 2.4 Functions without Parameters and without Return Values

Output

```
Current:  0.5  A
Power:  115.0  W
Work:  920.0  Wh
Cost:  0.276 Euro
```

Analysis

This program consists of four functions. Each function solves a self-contained task. The identifiers for the function names should be formulated in such a way that the task of a function is immediately recognized. Only nouns should be used for function names because an identifier such as `calculate_current()` contains a *pleonasm*, a meaningless duplication, because the actual task of the function is already to perform a calculation. A function name should describe the task of the function as precisely as possible. The first character in a function name must not be a number or a special character.

In lines 02 to 04, the variables necessary for the calculations are defined. The values of these variables are available to all four functions, which is why they are also referred to as *global variables*.

Line 06 contains the function definition for the calculation of the current. All other function definitions follow the same pattern. A function definition is introduced by the `def` keyword. This keyword is followed by a freely selectable function name. The parentheses after the function name are mandatory, even if no parameters are used. Identifiers for function names should consist of lowercase letters, as convention requires. The function definition is terminated with a colon. The function body (lines 07 and 08) consists of the individual statements. All statements of a function must be indented evenly so that the interpreter recognizes which statements belong to the function definition. The function definition is complete when it is followed by a statement that has the same indentation depth as the function header. The variables that are declared within functions are referred to as *local variables*. A local variable cannot be changed outside of the function in which it is declared.

In lines 25 to 28, the individual functions are called by specifying their name. The syntax of a function call is similar to a simple statement without an assignment. When calling the function, don't forget the parentheses. Only the parentheses tell the interpreter that the statement is a function call. The self-explanatory names of the identifiers for the functions significantly improve the readability of the program. At first glance, which calculations are performed can be immediately visible.

2.2.3 Functions with Parameters and a Return

The program shown in Listing 2.5 again performs the same calculations as the example shown in Listing 2.4. With regard to the system, functions with a return can be described as a black box, as shown in Figure 2.2, with inputs and outputs. The parameters represent the inputs, and the `return` statement causes the output of the calculation results to the outputs. The calculations themselves are performed within the black box.

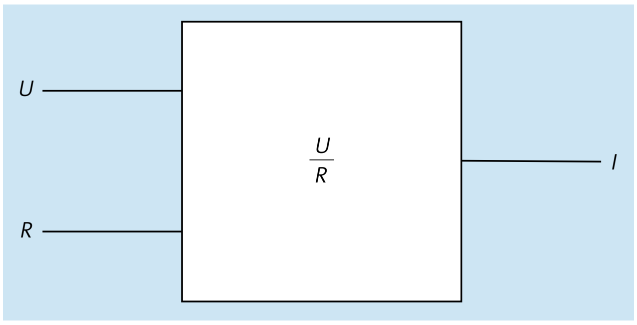


Figure 2.2 The Black Box of a Function with a Return

```python
#05_function2.py
def current(U,R):
    return U/R

def power(U, R):
    return U**2/R

def work(U, R, t):
    P=U**2/R
    W=P*t
    return W

def cost(U, R, t, price):
    I=U/R
    W=U*I*t
    c=W*price/1000.0
    return c

Uq=230    #V
RLoad=23 #ohms
tn=8      #h, hours
price_actual=0.3 #euro
print("Current: ", current(Uq, RLoad), " A")
print("Power  : ", power(Uq, RLoad), " W")
print("Work   : ", work(Uq, RLoad,tn), " Wh")
print("Cost   : ", cost(Uq, RLoad,tn,price_actual), " euros")
```

Listing 2.5 Functions with Return Value

Output

```
Current:  10.0  A
Power  :  2300.0  W
Work   :  18400.0  Wh
Cost   :  5.52 euros
```

Analysis

In line 02, the `current(U,R)` function is defined with the `U` and `R` parameters. The `return` statement in line 03 is followed by the calculation rule for the current. In line 23, this function is called in the built-in `print` function, and the calculated value for the current is output. As a result, there is a function call within a function. The variables in parentheses are referred to as *parameters*. A further distinction can be made between the parameters of the function definition (called *formal parameters*) and the parameters passed in the function call (called *current parameters* or *arguments*). The other function definitions and calls follow the same pattern. The variables for the current parameters are declared in lines 19 to 22.

In lines 23 to 26, the values for voltage `Uq`, resistance `RLoad`, time of use `tn`, and for the present price `preis_actual` are passed as *arguments* to the functions.

Note: Difference between a Parameter and an Argument

Applied computer science distinguishes between the terms *argument* and *parameter*. An argument is a value that is passed when the function is called. This value is assigned to the assigned parameters within the function. A parameter is the name used within the function.

The variables declared in the function bodies are only valid locally (i.e., these variables cannot be accessed from outside). This principle of local validity, also known as *data encapsulation*, ensures that the values of local variables cannot be changed. Thus, an assignment at another position in the program does not cause these values to be overwritten. If this were the case, the calculations of the functions could be manipulated from outside, and the function would return incorrect results.

2.2.4 Functions with Multiple Return Values

Unlike other programming languages, Python also allows the return of multiple values. Let’s look at an example of a solid steel cylinder with a diameter of 1 decimeter and a length of 10 decimeters. With only one function, the volume, mass, moment of inertia, and acceleration torque can be calculated. All four values are supposed to be returned with a `return` statement.

The acceleration torque M_b increases proportionally with the angular acceleration α and the moment of inertia J:

$$M_b = \alpha J$$

The moment of inertia J of a cylinder increases proportionally with its mass m and proportionally with the square of its radius r:

$$J = \frac{1}{2}mr^2$$

The mass is calculated from the volume V and the density ρ of the cylinder:

$$m = \rho V$$

To calculate the volume V, you need the diameter d and the length l of the cylinder:

$$V = 0.785\ d^2 l$$

To implement this task, you must enter the formulas according to the syntax rules in reverse order into the text editor of a Python development environment. Your source code should resemble the code shown in Listing 2.6. Start the program.

```python
#06_function3.py
rho=7.85    #kg/dm^3, density for steel
alpha=1.2  #1/s^2, angular acceleration
g=3         #accuracy

def cylinder(d,l):
    V=round(0.785*d**2*l,g)
    m=round(rho*V,g)
    J=round(0.5*m*(d/2/10)**2,g)
    Mb=round(alpha*J,g)
    return (V,m,J,Mb)
    #return V,m,J,Mb
    #return [V,m,J,Mb]

d1=1  #dm
l1=10 #dm
T=cylinder(d1, l1)
print("Cylinder data: ", T)
print("Volume:              ", T[0]," dm^3")
print("Mass:                ", T[1]," kg")
print("Moment of inertia:  ", T[2]," kgm^2")
print("Acceleration torque:", T[3]," Nm")
```

Listing 2.6 Function with Four Return Values

Output

```
Cylinder data:  (7.85, 61.622, 0.077, 0.092)
Volume:               7.85  dm^3
Mass:                61.622  kg
Moment of inertia:   0.077  kgm^2
Acceleration torque: 0.092  Nm
```

Analysis

In lines 06 to 11, the `cylinder(d,l)` function is defined. The formal parameters are the diameter `d` and the length `l` of the cylinder. First, the volume `V` is calculated, then the mass `m`, then the moment of inertia `J` and finally the acceleration torque `Mb`. The results are rounded to three digits of precision using the built-in `round` function. The `return` statement in line 11 returns the four calculated values as a tuple. A *tuple* is a data structure that consists of an immutable sequence of variables (here `V`, `m`, `J` and `Mb`). The elements of a tuple are enclosed in parentheses and separated by commas (line 11). The parentheses can also be omitted (line 12).

If the return values are enclosed in square brackets (line 13), then the return is a list. Because the elements of a list are changeable, however, you should avoid this option of a return.

In line 17, the `cylinder(d1,l1)` function is called with the current parameters `d1=1` and `l1=10`. The results are assigned to the `T` variable. At this point, it becomes clear that `T` is not a simple variable, but an object containing the memory addresses of the variables `V`, `m`, `J` and `Mb`. In other words: `T` is a reference pointing to the memory addresses of the elements of the tuple `T`.

Line 18 outputs the four values of tuple `T`. Since the output is not unique in this form, in lines 19 to 22 the values are read individually from the tuple using the `[]` operator and then output.

2.2.5 Functions Call Other Functions

Functions can also call other functions. To show you how such function calls can be implemented, I want to use the example of the calculation of dynamic characteristics of a cylinder once again, as shown in Listing 2.7.

```python
#07_function4.py
rho=7.85     #kg/dm^3, density of steel

def volume(d,l):
    return 0.785*d**2*l

def mass(d,l):
    return rho*volume(d,l)
```

```
09
10  def moment_of_inertia(d,l):
11      return 0.5*mass(d,l)*(d/2/10)**2
12
13  def acceleration_torque(d,l,alpha):
14      return alpha*moment_of_inertia(d,l)
15
16  d1=1             #dm
17  l1=10            #dm
18  alpha1=1.2       #1/s^2, angular acceleration
19  V=volume(d1,l1)
20  m=mass(d1,l1)
21  J=moment_of_inertia(d1,l1)
22  Mb=acceleration_torque(d1,l1,alpha1)
23  print("Volume:              ", V, " dm^3")
24  print("Mass:                ", m, " kg")
25  print("moment of inertia:   ", J, " kgm^2")
26  print("Acceleration torque: ", Mb, " Nm")
```

Listing 2.7 Function Call in Other Functions

Output

```
Volume:               7.8500000000000005  dm^3
Mass:                 61.6225  kg
Moment of inertia:    0.07702812500000002  kgm^2
Acceleration torque:  0.09243375000000002  Nm
```

Analysis

The functions are defined, as usual, in lines 04 to 14. In line 08, the first function call of the `volume()` function occurs directly after the `return` statement. The volume is not calculated until the `mass()` function is called in the main program (line 20). The function calls are made in lines 19 to 22. The `mass()` function calls the `volume()` function. The `moment of inertia()` function calls the `mass()` function, and the `acceleration torque()` function calls the `moment of inertia()` function. The return values are assigned to the `V`, `m`, `J`, and `Mb` variables, which are thus available for output in lines 23 to 26.

2.3 Branching Structures

The example on recursion has shown that certain algorithms are not executable without control structures. Thus, in real life, you'll always find programs with branching structures. Applied computer science distinguishes between single selection and multiple selection branches.

2.3.1 Single Selection

A single selection has the following general structure:

```
if condition:
    statement1
    statement2
    statement3
else:
    statement4
    statement5
```

If the `condition` is true in the single selection, then the statement block from `statement1` to `statement3` will be executed; if the `condition` is false, then the statement block from `statement4` to `statement5` will be run. Let's use an example of a quadratic equation to show how a choice between two possible cases is implemented.

The general form of a quadratic equation is as follows:

$$x^2 + px + q = 0$$

The solution of a quadratic equation is calculated in the following way:

$$x_{1,2} = -\frac{p}{2} \pm \sqrt{\frac{p^2}{4} - q}$$

The term under the root is called discriminant D in the technical language of mathematics.

The expression under the root can also take negative values. When this case occurs, the equation can no longer be solved within the real number space. For this reason, the program must catch this case by checking whether $D \geq 0$. For the problem to be solved, the structure chart shown in Figure 2.3 can be created.

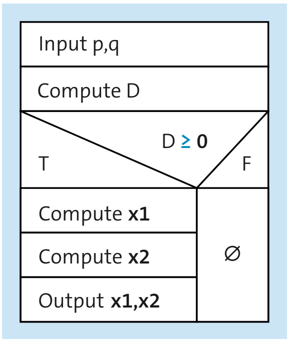


Figure 2.3 Structure Chart for the Single Selection

Listing 2.8 shows the implementation of the structure chart for this simple branching structure.

```
01  #08_branch1.py
02  import math as m
03  p=-8.
04  q=7.
05  D=(p/2)**2 - q
06  if D >= 0:
07      x1 = -p/2 + m.sqrt(D)
08      x2 = -p/2 - m.sqrt(D)
09      print("x1 =",x1,"\nx2 =",x2)
10      print("p =",-(x1+x2),"\nq =",x1*x2)
11  else:
12      print("The equation cannot be solved!")
```

Listing 2.8 Case Query for the Solution of a Quadratic Equation

Output

```
x1 = 7.0
x2 = 1.0
p = -8.0
q = 7.0
```

Analysis

In line 02, the `math` module is imported and assigned to the `m` alias. The values for the `p` and `q` coefficients of the quadratic equation are specified in lines 03 and 04. Line 05 calculates the discriminant `D`. If this value is greater than zero, the `if` branch will be executed, and the values for `x1` and `x2` (lines 07 and 08) will be calculated. The `sqrt()` root function of the `math` module is accessed using the `m` alias and the dot operator, `m.sqrt(D)`. Line 09 outputs the result. Line 10 performs a control calculation according to Vieta's theorem.

If the discriminant is less than zero, the `else` branch will be executed from line 11, and the message that the equation cannot be solved will be output.

2.3.2 Multiple Selection

A multiple selection is always used when there are multiple alternatives to choose from, for example, in a menu that offers different calculations. A multiple selection has the following formal structure:

```
if condition1:
    statement1
    statement2
elif condition2:
    statement3
    statement4
elif condition3:
    statement5
    statement6
```

The `elif` keyword is used to query further conditions. The sample program shown in Listing 2.9 for multiple selection determines the numerical value of a ring from the color coding of a resistor. A carbon film resistor is coded with four color rings. The first two rings represent the digits of an integer. The third ring serves as a multiplier. The fourth ring indicates the tolerance. For simplicity, the complete evaluation of the color rings is omitted in this example. For multiple selection, the structure chart can be created from Figure 2.4.

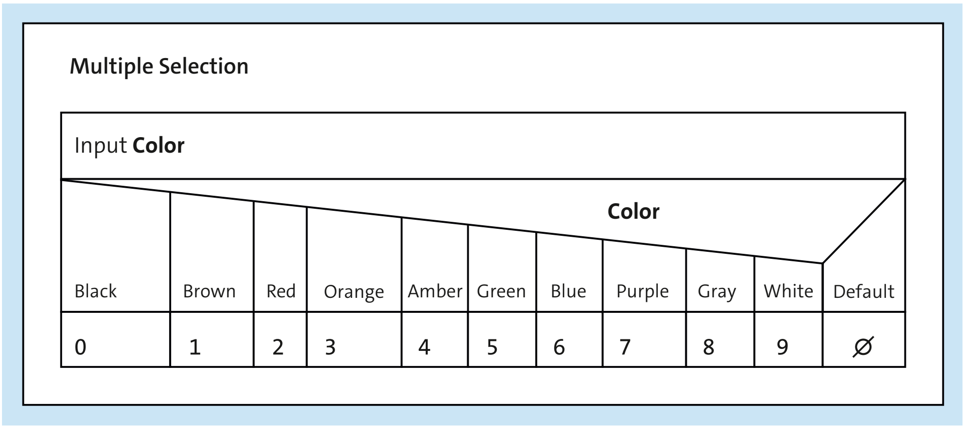


Figure 2.4 Structure Chart for Multiple Selection

The conversion of the structure chart is done in Listing 2.9.

```
#09_multiple_selection1.py
color=["black", "brown", "red", "orange", "yellow",
       "\ngreen","blue","purple","gray","white"]
code="yellow"       #input
if code==color[0]:
    print("The color black is coded as 0.")
elif code==color[1]:
    print("The color brown is coded as 1.")
elif code==color[2]:
    print("The color red is coded as 2.")
elif code==color[3]:
    print("The color orange is coded as 3.")
elif code==color[4]:
    print("The color yellow is coded as 4.")
```

```
15  elif code==color[5]:
16      print("The color green is coded as 5.")
17  elif code==color[6]:
18      print("The color blue is coded as 6.")
19  elif code==color[7]:
20      print("The color purple is coded as 7.")
21  elif code==color[8]:
22      print("The color gray is coded as 8.")
23  elif code==color[9]:
24      print("The color white is coded as 9.")
```

Listing 2.9 Multiple Selection for the Color Coding of Resistors

Output

```
The color yellow is coded as 4.
```

Analysis

In line 02, a list of ten colors is created and assigned to the variable `color`. All properties of the list elements are now stored in the `color` variable (an object!). Each color represents a specific digit. In line 04, the color of the color ring is assigned to the `code` variable. The list elements are accessed using the `[]` operator. The `if` statement in line 05 determines the first alternative. The check whether the respective case applies is performed using the `==` operator. All other cases are queried using the `elif` statement (from line 07). For example, since the color `yellow` stands for value `4` and has the index 4 in the list, the program outputs the value `4`.

Multiple Selection for One Area

In real life, value ranges will need to be queried, for instance, when calculating the energy costs at a specific electricity rate. If the calculations are within a defined range of values, a case distinction must be made. Four ranges for hypothetical electricity rates are calculated in Listing 2.10.

```
#10_multiple_selection2.py
rate1,rate2,rate3=0.3,0.25,0.2 #euros
consumption=5500 #kWh

if 0 < consumption<= 5000:
    print("Amount for rate1:",consumption*rate1, "euros")
elif 5000 < consumption <= 10000:
    print("Amount for rate2:",consumption*rate2, "euros")
elif 10000 < consumption <= 30000:
```

```
10      print("Amount for rate3:",consumption*rate3, "euros")
11  else:
12      print("industry_rate!")
```

Listing 2.10 Multiple Selection for One Area

Output

```
Amount for rate2: 1375.0 euros
```

Analysis

Line 02 establishes three electricity rates. Line 03 determines the actual consumption. The case query for the value ranges is performed using the notation known from mathematics. If the consumption is exactly equal to 5000 kWh or below, then the amount to be paid for `rate1` is calculated and output in line 06 using the `if` statement. Line 07 uses the `elif` statement to query the consumption between 5,000 and 10,000 kWh. The amount to be paid for `rate2` is calculated and output in line 08. The same applies to line 09. If the consumption is not in the specified range, the `else` branch in line 11 will be executed.

2.4 Repetitive Structures

In Python, two constructs exist for implementing repetitive structures: the `while` loop and the `for` loop. Loop constructs are always needed when a statement block must be executed multiple times, for example, when value tables for mathematical functions or when rectangle sums must be calculated for numerical integration.

2.4.1 The while Loop

A `while` loop consists of the loop head and the statement block or loop body that is supposed to be repeated. This loop has the following general structure:

```
while condition:
    statement1
    statement2
    statement3
      ...
```

The loop body can consist of one or more statements. The statements of the loop body are executed as long as the condition is `True`, and its execution is aborted if the condition is no longer true (i.e., if this condition is `False`). The termination condition results either from the calculations performed in the loop body or from a previously defined condition.

The first example shown in Listing 2.11 illustrates how to use a `while` loop to calculate the value table of any mathematical function. The structure chart associated with the program is shown in Figure 2.5. To focus on the essential structural elements of the program, I have omitted the presentation of the function call.

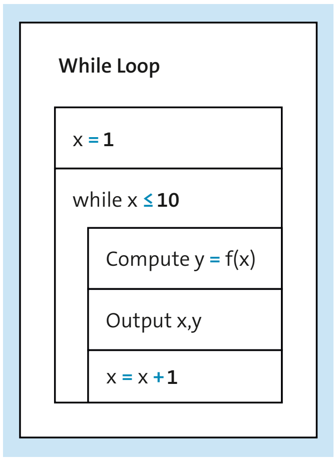


Figure 2.5 Structure Chart for a while Loop

The conversion of the structure chart into a Python program is shown in Listing 2.11.

```
#11_while_loop1.py
def f(x):
    return x**2
x=1
while x<=10:
    y=f(x)
    print(x,y)
    x=x+1   #better x+=1
```

Listing 2.11 while Loop for a Value Table

Output

```
1  1
2  4
3  9
4  16
5  25
6  36
7  49
8  64
9  81
10  100
```

Analysis

In lines 02 and 03, a function named `f(x)` is defined. The `return` statement can be followed by the term of any mathematical function, *f(x)*. Line 04 initializes the independent variable `x` with 1. The `while` statement in line 05 sets the termination condition as `x<=10`. The loop body will be executed as long as `x<=10`. Each termination condition must be terminated using a colon. All statements of the loop body must be indented evenly. In line 08, `x` is *incremented* by the value `1` with each loop pass; in technical jargon, this step is referred to as *incrementing*. This increment and the `x<=10` condition determine that the program executes the loop ten times. Line 06 calls function `f(x)` with the current value of `x`. Line 07 outputs the values for `x` and `y`. In line 08, the value of `x` is increased by 1 with each loop pass. Instead of writing `x=x+1`, the formulation `x+=1` is also common.

while Loop for Program Repetition

The next example shown in Listing 2.12 demonstrates how to run a program that contains a menu for four selection options until the user forces the program termination. The program calculates four kinds of energy: kinetic, rotational, electrical, and magnetic. Because all four formulas for calculating energy have the same structure, as shown in Table 2.2, you also only need to implement one function.

Energy Type	Formula	Energy Storage
Kinetic energy	$W_{\text{kin}} = \frac{1}{2}mv^2$	Mass
Rotational energy	$W_{\text{rot}} = \frac{1}{2}J\omega^2$	Moment of inertia
Electrical energy	$W_{\text{el}} = \frac{1}{2}CU^2$	Capacity
Magnetic energy	$W_{\text{mag}} = \frac{1}{2}LI^2$	Inductance

Table 2.2 Types of Energy

For the storage variables mass, moment of inertia, capacity, and inductance, you generally specify the a variable. The physical quantities such as velocity, angular velocity, voltage, and current are generally denoted by x:

$$f(x) = 0.5ax^2$$

In your development environment, enter the source code shown in Listing 2.12 and then start the program.

```
#12_while_loop2.py
def f(a,x):
    return 0.5*a*x**2

next=True
while next:
    print("Kinetic energy......1")
    print("Rotational energy...2")
    print("Electrical energy...3")
    print("Magnetic energy.....4")
    selection=int(input("Select:"))
    if selection==1:
        m=float(input("Mass m="))
        v=float(input("Velocity v="))
        Wkin=f(m,v)
        print("\nThe kinetic energy is %6.3f Ws\n" %Wkin)
    elif selection==2:
        omega=float(input("Angular velocity \u03C9="))
        J=float(input("Moment of inertia J="))
        Wrot=f(J,omega)
        print("\nThe rotational energy is %6.3f Ws\n" %Wrot)
    elif selection==3:
        C=float(input("Capacity C="))
        U=float(input("Voltage U="))
        Wel=f(C,U)
        print("\nThe electrical energy is %6.3f Ws\n" %Wel)
    elif selection==4:
        L=float(input("Inductance L="))
        I=float(input("Current I="))
        Wmag=f(L,I)
        print("\nThe magnetic energy is %6.3f Ws\n" %Wmag)
    else:
        next =False
```

Listing 2.12 A while Loop with Internal Termination Condition

For example, if you select menu item 2, enter 1.2 s^{-1} for the angular velocity, and 2.4 kg m^2 for the moment of inertia, the program will calculate a value of 1.728 Ws for the rotational energy.

```
Kinetic energy......1
Rotational energy...2
Electrical energy...3
Magnetic energy.....4
Select:2
```

```
Angular velocity ω=1.2
Moment of inertia J=2.4

The rotational energy is 1.728 Ws
```

Analysis

Line 05 initializes the Boolean variable `further` via the `True` value. Line 06 contains the loop header of the `while` loop. The loop body is executed as long as `next` equals `True`, which is the case if the values 1, 2, 3, or 4 were entered for the `Select` variable. For all other values, the `else` branch in line 32 is executed, and the `next` variable is set to `False`.

while Loop with a break Statement

If you need to perform divisions in a loop body, during the course of the calculations, the denominator of a fraction might become zero or be very small. If divided by zero, the result would become infinite, resulting in a memory overflow. This case must be intercepted. For this purpose, Python provides the `break` statement, which causes a loop to abort.

Based on the example of a zero point calculation, I want to show you how a `break` statement works. Using *regula falsi*, zeros can be calculated numerically in a simple way:

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$$

The denominator of the fraction contains the difference between the newly calculated and the previously calculated function value. This value may become zero or take a very small value during the calculations.

Listing 2.13 calculates the zero for the function:

$$f(x) = x - \cos x = 0$$

Using a sketch for the function graphs $f_1(x) = x$ and $f_2(x) = \cos x$, you'll get an intersection point of both function graphs that is approximately at $x = 0.74$. Therefore, $x_1 = 0$ is set for the start value, and $x_2 = 1$, for the end value.

```
#13_while_loop3.py
import math as m
def f(x):
    return x-m.cos(x)

eps=1e-12 #termination condition
x1=0 #start value
x2=1 #end value
n=0
f1=f(x1)
```

```
11  while abs(x2-x1)>eps and n<100:
12      n+=1
13      x0=x1
14      x1=x2
15      f0=f1
16      f1=f(x1)
17      if abs((f1-f0))<eps: break
18      x2=x1-f1*(x1-x0)/(f1-f0)
19      print(n,":", x2)
```

Listing 2.13 A while Loop with a break Statement

Output

```
1 : 0.6850733573260451
2 : 0.736298997613654
3 : 0.7391193619116293
4 : 0.7390851121274639
5 : 0.7390851332150012
6 : 0.7390851332151607
```

The result can be checked in the Python shell with the following command:

```
>>> import math
>>> 0.7390851332151607 - math.cos(0.7390851332151607)
0.0
>>>
```

Analysis

Line 02 imports the math module (`math`), which is needed for the calculation of the `cos` function. The `m` alias saves some typing work. Instead of `math.cos()`, the program calls the cosine function via the `m` alias and the `m.cos()` dot operator. The function definition is performed in lines 03 and 04. In lines 06 to 10, the variables are initialized with their initial values.

Line 11 contains the termination condition of the `while` loop. The loop is supposed to be executed as long as the amount of `eps` is greater than 10^{-12} and `n<100`. In line 13, the value of variable `x1` is assigned to variable `x0`. This assignment causes the last calculated value of `x1` to be temporarily stored in the `(x1-x0)` counter for the calculation of the difference. For the calculation of the denominator, the last value of `f1` is assigned to the `f0` variable in line 15 and thus also temporarily stored. The difference of the denominator `(f1-f0)` can thus be calculated from the current and the previously calculated values.

Line 17 contains the termination condition. The loop is then exited when the amount of the counter becomes less than 10^{-12}. A check whether the case `f1-f0==0` occurs or whether it becomes `f1==f0` would even work in this case. However, I strongly recommend you avoid such an implementation because two floats are very rarely really equal.

In line 18, the calculation of the zero is performed according to the *regula falsi* iteration rule. The result is output with the number of required calculation steps in line 19.

2.4.2 The for Loop

The `for` loop is a counter-driven loop. The number of times the loop body should be executed is already specified in the loop header. This kind of loop is always used when the number of loop passes is known in advance. It has the following general structure:

```
for i in range(start value, end value, increment):
    statements
```

The count variable `i` must always be of type integer. The `range` function sets the start value, the end value, and the increment for the count variable. The head of the `for` loop must end with a colon. The `range(n)` function internally generates a list of integers for a range of values from 0 to *n -1* with an increment of 1. For `range(10)`, the count variable `i` takes the values `i` = `0` to `9` in succession:

```
>>> for i in range(10):
        print(i, end='  ')
0  1  2  3  4  5  6  7  8  9
```

The count variable of a `for` loop can also iterate over a string, list, tuple, or dictionary. These options are discussed in more detail in Section 2.5.

Figure 2.6 shows the structure chart for a `for` loop. The program should calculate the value table of a mathematical function.

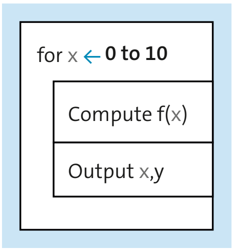


Figure 2.6 Structure Chart for a for Loop

The conversion of the structure chart into a Python source code is shown in Listing 2.14. The program calculates the table of values for a parabola for the range from x = 0 to x = 10. The count variable x is of the integer type. You can also enter any other mathematical functions in the function definition (line 03).

```
01  #14_for_loop1.py
02  def f(x):
03      return x**2
04
05  print(" x\ty")
06  for x in range(11):
07      y=f(x)
08      print("%2i %6.3f" %(x, y))
```

Listing 2.14 Calculating a Value Table Using a for Loop

Output

```
 x     y
 0   0.000
 1   1.000
 2   4.000
 3   9.000
 4  16.000
 5  25.000
 6  36.000
 7  49.000
 8  64.000
 9  81.000
10  100.000
```

Analysis

The program outputs 11 pairs of values for `x` and `y`. The loop header in line 06 contains the count variable `x`, which is automatically declared as `int`, and the `range` function, whose parameters must also be of type `int`. The loop header must always end with a colon. The loop body must be evenly indented. Line 07 calls the $y = f(x)$ function. With each new loop pass, `x` is incremented by 1, and the function value is recalculated until the termination condition is reached. Line 08 outputs the values for x and y in a formatted manner. You can test the loop construct by inserting other start and stop values and other increments into the `range` function. If you type `help(range)` in the Python shell, you'll get detailed information about the `range` class.

Reducing the Increment

Often, values of functions must be calculated whose increment is not 1 but less than 1. This case arises, for example, in numerical integration with rectangle sums. The area A of the rectangles is calculated from the sum of products of the current function value, $f(x_k)$ and a Δx that's chosen as small as possible:

$$A = \sum_{k=0}^{n} f(x_k)\Delta x$$

The program in Listing 2.15 calculates the rectangle sums of the e-function between the limits from 0 to 1. The expected result is A = 1.718281828459045 area units ($e^1 - 1$).

```
01  #15_for_loop2.py
02  import math
03  def f(x):
04      #return x
05      #return -x+1
06      return math.exp(x)
07
08  a=0 #lower limit
09  b=1 #upper limit
10  n=1000
11  delta_x=(b-a)/n
12  r=0
13  x=a
14  for k in range(1,n+1):
15      r=r+f(x)*delta_x
16      x=a+k*delta_x
17  print("%6d %6.3f  %6.15f" %(k, x, r))
```

Listing 2.15 Numerical Integration with Rectangle Sums

Output

```
1000   1.000   1.718422830734965
```

Analysis

The function definition is made in lines 03 and 06. The lines that have been commented out can be used for further test functions. Lines 08 and 09 define the lower and upper integration limits. The n variable in line 10 defines the number of subproducts $f(x_k)\Delta x$. For the intercept Δx on the x-axis (called the *abscissa*), the somewhat unwieldy identifier delta_x was chosen so as to avoid creating a false association with the differential dx. delta_x is calculated in line 11 from the difference between upper and lower limits divided by the number of subproducts n.

The r variable is initialized with 0 (line 12), and the x variable is initialized with the lower limit a (line 13). The for loop in line 14 is run through from k=1 to n+1. Thus, 1,000 subproducts (rectangles r) are added up. The number n of subproducts determines the accuracy of numerical integration. Since rectangle sums are calculated, no improved accuracy can be achieved by increasing n compared to other integration methods (i.e., *trapezoidal*, *Simpson*, or *Romberg*).

The summation of the individual rectangle areas is performed via the summation algorithm in line 15. On the right-hand side of the assignment, the sum of the old `r` value and the rectangle area `f(x)*delta_x` at location `k` is calculated. The `f(x)` function is called anew for each loop pass. The function argument `x` is recalculated at position `k` in line 16 for each loop pass.

Line 17 outputs the number of calculations, the value of the upper limit, and the area. Except for the third digit, the result calculated by the program matches the exact value.

Numerical Solution of First-Order Differential Equations

In engineering and science, you must be able to solve *differential equations*. Based on the example of the *Euler-Cauchy method*, I want to show you how easily you can solve differential equations. This method is described using the following sum algorithm:

$$y_{k+1} = y_k + f(x_k, y_k)\Delta x$$

Listing 2.16 calculates differential equations of the following type: $y' = f(x, y)$. In this concrete case, the solution of the differential equation $y' = xy$ is to be calculated at the point $x = 1$. The exact solution of this differential equation is:

$$y = e^{\frac{x^2}{2}}$$

For $x = 1$, the exact value of the solution is thus $y = 1.6487$.

```python
#16_for_loop3.py
def f(x,y):
    return x*y

x0=0
xn=1
y0=1
n=1000
delta_x=(xn-x0)/n
y=y0
for k in range(n+1):
    x=x0+k*delta_x
    y=y + f(x,y)*delta_x
print("%3i %6.3f  %6.4f" %(k, x, y))
```

Listing 2.16 Solution of a First-Order Differential Equation

Output

```
1000  1.000  1.6493
```

Analysis

The function definition in line 02 expects two parameters when called. With each function call, the product of x and y is returned. The statement in line 09 calculates the delta_x increment from the start and end values as well as the number n. In line 12, the current x value is calculated for the function call in line 13. The algorithm of the Euler-Cauchy method is implemented directly in line 13 in Python syntax. The print function in line 14 outputs the number of calculations and the function value of the solution y at position x=1. The result shows that the Euler-Cauchy method is not suitable for practical purposes because this algorithm still yields an error of 0.0006 even after 1,000 loop passes. Doubling n only halves the error. The *Heun method* or the *Runge–Kutta method* provide more accurate results.

Nested Loops

Loops can also be nested within each other. You can use two nested for loops to create triangular or rectangular number schemes. Pascal's triangle is an example of a triangular number scheme. It can be generated using the following binomial coefficient:

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

The math function comb(n,k) calculates the binomial coefficient. Listing 2.17 demonstrates how you can create Pascal's triangle using this function and two nested for loops.

```python
#17_for_for_loop1.py
from math import *
k=8
for n in range(k):
    for k in range(n+1):
        print(comb(n,k),end=' ')
    print()
```

Listing 2.17 Two Nested for Loops

Output

```
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
```

Analysis

The first `for` loop creates the lines of Pascal's triangle (line 04). In line 05, the counting range is increased by `1` with each new loop pass of the inner loop. Line 06 creates the column entries. The `end=' '` parameter prevents a line break. In line 07, the `print` function forces a new line break.

Usage Example: BubbleSort

What's called the *BubbleSort method* compares adjacent pairs of numbers in a sequence of numbers. These pairs of numbers are compared and swapped until the sequence of numbers has been sorted. While this procedure is rather inefficient, it is particularly well suited to illustrate the operation of two nested `for` loops. Listing 2.18 shows a quite simple implementation of this sorting procedure.

```
01  #18_for_for_loop2.py
02  a=[5,4,3,2,1] #list
03  print(a)
04  for i in range(len(a)-1):
05      for i in range(len(a)-1):
06          if a[i]>a[i+1]: #compare
07              a[i],a[i+1]=a[i+1],a[i] #swap
08      print(a)
```

Listing 2.18 BubbleSort

Output

```
[5, 4, 3, 2, 1]
[4, 3, 2, 1, 5]
[3, 2, 1, 4, 5]
[2, 1, 3, 4, 5]
[1, 2, 3, 4, 5]
```

Analysis

The statement in line 02 creates list `a`. The *list* data structure is discussed in Section 2.5.2. Line 06 compares the predecessor `a[i]` with its direct successor `a[i+1]`. If the predecessor is greater than its successor, the corresponding elements in the list will be swapped (line 07). The swap process is performed using *tuples*, which is a data structure described in Section 2.5.1.

Usage Example: Double Integral

Nested loops are also needed, for example, to calculate a double integral numerically. A typical application is the calculation of the *second area moment*. The second area moment indicates how stiff a beam is based on its cross-sectional area. For the second area moment of a rectangle cross section, the following applies:

$$I_y = \int_{z=-\frac{h}{2}}^{\frac{h}{2}} \left(\int_{y=-\frac{b}{2}}^{\frac{b}{2}} z^2 \,\mathrm{dy} \right) \mathrm{dz} = \int_{-\frac{h}{2}}^{\frac{h}{2}} b \cdot z^2 \,\mathrm{dz} = \frac{b \cdot h^3}{12}$$

You can calculate the double integral numerically by applying the sum algorithm within two nested for loops. Listing 2.19 shows the implementation of such an algorithm.

```
#19_for_for_loop3.py
b=5   #width in cm
h=10  #height in  cm
y1,y2=-b/2,b/2 #limits of the y-axis
z1,z2=-h/2,h/2 #limits of the z-axis
#Function definition
def f(y,z):
    return z**2
#Calculate double integral
dy=dz=1e-2
m=int((z2-z1)/dz) #height
n=int((y2-y1)/dy) #width
sz=0
for i in range(m):    #outside
    z=z1+i*dz
    sy=0
    for j in range(n): #inside
        y=y1+j*dy
        sy=sy+f(y,z)
    sz=sz+sy
Iy=sz*dy*dz
#Output
print("First moment of area for a rectangle cross section")
print("Iy =",Iy, "cm^4")
print("Iy =",b*h**3/12,"cm^4 exactly")
print(m,n)
```

Listing 2.19 Numerical Calculation of a Double Integral

Output

```
First moment of area for a rectangle cross section
Iy = 416.6675 cm^4
Iy = 416.6666666666667 cm^4 exactly
1000 500
```

Analysis

In line 10, you can define the `dy` and `dz` increments. The increments determine the accuracy of the numerical integration. Lines 11 and 12 calculate the number of loop passes for the outer and inner loops. For $m = 1000$ and $n = 500$, we obtain $1000 \times 500 = 50000$ computational steps in the inner loop.

In lines 15 and 18, the current values `z` and `y` are calculated for the z and y coordinates. In line 19, these values are passed as arguments to the function `f(y,z)` and added to the sum `sy` at each new loop pass.

In line 20, the sum is calculated in the z-direction. Line 21 calculates the second moment of area, `Iy`.

The comparison between numerical integration and exact value shows that the accuracy is still acceptable. In Chapter 6 on using SciPy, you'll learn how to use the function `dblquad(f,z1,z2,y1,y2)[0]` to calculate the second moment of area in a much easier way by using only one line of source code.

2.5 Data Structures

Programs are composed of algorithms and data structures. So far, data structures such as *lists* and *tuples* have already appeared in some of the sample programs. Let's now take a closer look at what data structures Python brings to the table.

Data Structures and Data Types

You should not confuse data structures with simple data types, such as `int`, `float`, and `str`. An essential difference between data types and data structures is that data structures have a much more complex structure than the simple data types.

In applied computer science, a data structure is a set of objects that may only be manipulated by means of well-defined operations. In a nutshell, *data structure = objects + operations*. The data is organized in a way that is ideal for the particular data structure in order to access it as efficiently as possible.

Python has the following built-in data structures: *tuples*, *lists*, *dictionaries*, and *sets*.

2.5.1 Tuples

A *tuple* is a sequence of elements that are iterable but cannot be modified. The elements of a tuple do not all need to be of the same type. The immutability of the elements is the decisive characteristic of a tuple. You can define a tuple by enclosing the elements in parentheses separated by commas. The Python shell is helpful again with our first encounter with the tuple data structure through the following commands:

```
>>> t=(2,4,6)
>>> t
(2, 4, 6)
>>> t[1]
4
>>> t[1]=8
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> type(t)
<class 'tuple'>
>>>
```

The individual elements of a tuple can only be accessed in read-only mode. If an attempt is made to assign a value to an element of a tuple, then the Python interpreter throws an error message.

Listing 2.20 shows how tuples are defined and which operations are applicable to tuples.

```python
#20_tuple1.py
t1=(1,2,3)
t2= 4,5,6
t3=t1+t2
t4=3*t2
print("Tuple1 contains the elements",t1)
print("Tuple2 contains the elements",t2)
print("Tuple3 contains the elements",t3)
print("Tuple4 contains the elements",t4)
print("The third object of t3 has the value", t3[2])
print("Are t1 and t2 the same?",t1==t2)
print("t3 belongs to the class",type(t3))
print("t1    has id",(id(t1)))
print("t1[0] has id",(id(t1[0])))
print("t1[1] has id",(id(t1[1])))
print("t1[2] has id",(id(t1[2])))
```

Listing 2.20 Operations on Tuples

Output

```
Tuple1 contains the elements (1, 2, 3)
Tuple2 contains the elements (4, 5, 6)
Tuple3 contains the elements (1, 2, 3, 4, 5, 6)
Tuple4 contains the elements (4, 5, 6, 4, 5, 6, 4, 5, 6)
The third object of t3 has the value 3
Are t1 and t2 the same? False
t3 belongs to the class <class 'tuple'>
t1 has id 4344167376
t1[0] has id 4335508656
t1[1] has id 4335508688
t1[2] has id 4335508720
```

Analysis

Two tuples (i.e., `t1` and `t2`) are defined in lines 02 and 03. You can also omit the parentheses. Line 04 concatenates tuples `t1` and `t2` to form the new tuple, `t3`. In line 05, another new tuple `t4` is created, this one containing three copies of tuple `t2`. Each element of a tuple can be accessed in read-only mode via the `[]` operator (line 10). Tuples can also be checked for equality using the `==` operator (line 11). Not only does a tuple have its own identity (line 13), but also each element of a tuple has its own identity (lines 14 to 16).

Are Tuples Really Immutable?

In the sorting program shown in Listing 2.18, the value of two variables has been swapped:

```
a[i], a[i+1] = a[i+1], a[i]
```

The left-hand value of tuple `a[i]`, `a[i+1]` was assigned the right-hand value `a[i+1]`, `a[i]`. `a[i]` had the value of `a[i+1]` and `a[i+1]` the value of `a[i]` after the swap. The swap operation was apparently performed successfully because the program worked. How can we explain this contradiction, that tuples are supposed to be immutable, but during the exchange process the values of two tuple elements were changed? Listing 2.21 resolves this contradiction.

```
#21_tuple2.py
a,b=10,20
t=(a,b)
print("----before----")
print("Value of a=%i id of a=%i" %(a,id(a)))
print("Value of b=%i id of b=%i" %(b,id(b)))
print("Value of t=",t,"id of t=",id(t))
a,b=b,a
```

```
09 print("----after----")
10 print("Value of a=%i id of a=%i" %(a,id(a)))
11 print("Value of b=%i id of b=%i" %(b,id(b)))
12 print("Value of t=",t,"id of t=",id(t))
```

Listing 2.21 Swapping Two Variables with a Tuple

Output

```
----before----
Value of a=10 id of a=4317511120
Value of b=20 id of b=4317511440
Value of t= (10, 20) id of t= 4326503240
----after----
Value of a=20 id of a=4317511440
Value of b=10 id of b=4317511120
Value of t= (10, 20) id of t= 4326503240
```

Analysis

The left-hand value of the tuple consists of the variables `a` and `b` (line 02). Variable `a` is assigned the value 10, and variable `b`, the value 20. Line 03 defines a tuple with elements `a` and `b`. In lines 05 to 07, the values and the identities of the variables and the tuple are output. In line 08, the swap process takes place. After swapping, variable `a` has the value 20 and variable `b` has the value 10. The values and the IDs of `a` and `b` have changed after the swap. That is, only the memory addresses of `a` and `b` were swapped. In contrast, the values and ID of tuple `t` have not changed.

Tuples in for Loops

In a `for` loop, a count variable can also iterate over a tuple. The following console example shows a possible implementation:

```
>>> for i in ('Iron','Chromium','Nickel'):
        print(i,end='  ')
Iron Chromium Nickel
```

2.5.2 Lists

A *list* is an ordered summary of various objects. The values contained in a list are also referred to as elements. The list itself is also considered an object. The special thing about a list is that its length can be changed at runtime. The Python interpreter recognizes a list definition by the square brackets in which the elements of a list are embedded. The individual elements are separated by commas. Objects of a list can be, for example, floats of measured values of a measuring sequence or any other objects. Lists themselves can also be components of lists. The `range` function enables you to generate lists automatically:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(2,10,2))
[2, 4, 6, 8]
>>> list(range(1,10,2))
[1, 3, 5, 7, 9]
```

Many operations are applicable to lists, such as inserting or removing single or multiple elements, accessing single elements, sorting elements, and so on. The elements of a list can be accessed via an index. Table 2.3 illustrates the structure of a list through a model.

Index	0	1	2	3	4
Value	5.7	6.8	5.9	6.2	5.1

Table 2.3 Model of a List Structure

Each element of a list is assigned an index. This index can be used for read and write access to the individual elements of the list. The count always starts with index 0. Again, you can use the Python shell to explore the `list` data structure, for instance, with the following commands:

```
>>> l=[2,4,6]
>>> l[1]
4
>>> l[1]=8
>>> l
[2, 8, 6]
>>> type(l)
<class 'list'>
>>>
```

By specifying the index, the value of the element at position `i` is output. A new assignment can change this value. The list `l` belongs to the `list` class.

A count variable can also iterate over a list in a `for` loop, as the following console example shows:

```
>>> for i in [1,3,5,7,9]:
        print(i+1,end=';')
2;4;6;8;10;
```

You can perform numerous operations on lists, such as sorting the list elements, determining their length, and appending or removing elements. Table 2.4 contains selected functions for operations on lists.

Function	Description
`n=len(l)`	Returns the number of elements.
`l2=sorted(l1)`	Returns the sorted list `l1`.
`s=sum(l)`	Returns the sum of list `l`.
`mi=min(l)`	Returns the smallest element.
`ma=max(l)`	Returns the largest element.
`l3=zip(l1,l2)`	Connects lists `l1` and `l2` to `l3`.

Table 2.4 Functions for Operations on Lists

In addition to these functions, you can also use methods. Table 2.5 contains important methods for performing operations on lists. The letter `l` stands for list, and the letter `e` stands for the element of a list.

Method	Description
`l1.extend(l2)`	List `l2` is appended to list `l1` at the end.
`l.append(e)`	Element `e` is appended to the end of the list.
`l.remove(e)`	Removes element `e` from the list.
`l.insert(i,e)`	Inserts an element `e` into list `l` at position `i`.
`l.count(e)`	Determines how often element `e` is contained in list `l`.

Table 2.5 Methods for Operations on Lists

Difference between Functions and Methods

Methods are functions that are defined within a class. To use methods, an `obj` object must be created beforehand. Otherwise, no other difference exists between the way a function works and the way a method works.

A function is called via `a=functionname(parameter)`, while a method is called using `a=obj.methodname(parameter)`.

Connecting Two Lists Using the zip Function

You should briefly test the `zip` function in the Python shell because it will be needed later in connection with the *dictionary* data structure. The following console example creates a new list from two lists:

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> zipped = zip(x, y)
>>> list(zipped)
[(1, 4), (2, 5), (3, 6)]
```

The `zip` function creates a new list of three *tuples* from the two lists, `x` and `y`.

Operations on Lists

Listing 2.22 shows how you can implement some selected operations on lists. For this purpose, uniform data of type `float` was deliberately selected as list elements in order to establish a reference to technically relevant topics. The program calculates important statistical parameters of a measuring sequence.

```
#22_list1.py
import statistics as stat
l1=[52.1,48.7,50.1,49.6,51.8]
l2=[50.5,48.5,49.5,51.5,48.8]
l1.extend(l2) #method
sl=sorted(l1) #function
n=len(sl)
minimum=min(l1)
maximum=max(l1)
s=sum(l1)
m=s/n
r=maximum-minimum
z=stat.median(l1)
print("sorted list:\n",sl)
print("Number of elements:",n)
print("Minimum: %6.2f Maximum: %6.2f" %(minimum,maximum))
print("Sum:",s)
print("Mean:",m)
print("Median:",z)
print("Span:",r)
```

Listing 2.22 Operations on a List

Output

```
sorted list:
[48.5, 48.7, 48.8, 49.5, 49.6, 50.1, 50.5, 51.5, 51.8, 52.1]
Number of elements 10
Minimum:  48.50 Maximum:  52.10
Sum: 501.1
Average: 50.11
Median: 49.85
Span: 3.6000000000000014
```

Analysis

The program calculates the arithmetic mean, the median, and the range of a measuring sequence. In lines 03 and 04, two lists with five `float` types each are defined. The values of the two lists are stored in objects `l1` and `l2`. In line 05, list `l2` is appended to list `l1` via the `l1.extend(l2)` *method*.

In line 06, the `sorted(l1)` *function* sorts the extended list `l1` and assigns the result to the `sl` variable. In line 07, the `len(sl)` function determines the length of the sorted list and assigns it to the `n` variable. From the sum of the measured values (line 10), the mean value `m` of the measured values can then be calculated in line 11. The determination of the minimum (line 08) and the maximum (line 09) is performed via the built-in functions `min(sl)` and `max(sl)` respectively. In line 10, the built-in `sum(sl)` function calculates the sum of the measuring sequence. The span `r` is calculated from the difference between maximum and minimum (line 12). In line 13, the `stat.median(l1)` function calculates the median of the measuring sequence. For this purpose, the `statistics` module (line 02) must be imported.

Lines 14 to 20 show the output of the results.

Nested Lists

Nested lists are important for the representation of two-dimensional matrices with NumPy arrays. To convert two nested lists `a` and `b` into NumPy arrays and then add them, consider the following console commands:

```
>>> from numpy import array
>>> a=[[1,2,3],[4,5,6]]
>>> b=[[7,8,9],[10,11,12]]
>>> A=array(a)
>>> B=array(b)
>>> A
array([[1, 2, 3],
       [4, 5, 6]])
>>> B
```

```
array([[ 7,  8,  9],
       [10, 11, 12]])
>>> A+B
array([[ 8, 10, 12],
       [14, 16, 18]])
```

List Comprehension

Python even enables you to run statements within a list, while the list is not generated until runtime. During runtime, its length can be changed (almost) at will. This feature is referred to as *list comprehension*. Listing 2.23 demonstrates the power of the list comprehension feature by calculating the Pythagorean numbers within a selected range.

```
01  #23_list2.py
02  ug=1
03  og=20
04  p=[(a,b,c)
05     for a in range(ug,og)
06     for b in range(a,og)
07     for c in range(b,og)
08     if a**2 + b**2 == c**2]
09  n=len(p)
10  print(p)
11  print("Between %i and %i there are %i Pythagorean triples." %(ug,og,n))
```

Listing 2.23 Dynamic Generation of a List via List Comprehension

Output

```
[(3, 4, 5), (5, 12, 13), (6, 8, 10), (8, 15, 17), (9, 12, 15)]
Between 1 and 20 there are 5 Pythagorean triples.
```

Analysis

Lines 02 and 03 define the upper and lower limits in which the Pythagorean numbers are to be calculated. The list definition starts at line 04 and ends at line 08. The list consists of only one element: a tuple of the triangle sides `(a,b,c)` including the three `for` loops with the `if` query.

In line 05, the `a` variable is iterated from `ug` to `og`. The `b` variable is iterated from `a` to `og` (line 06), and variable `c` is iterated from `b` to `og` (line 07). Line 08 checks that the sum of the squares of *a* and *b* is equal to the square of the hypotenuse *c*. If this is the case, then the Pythagorean triples are generated as list elements and stored in variable `p` (line 04).

Line 09 determines the length of list `p`. In line 10, the output of the Pythagorean numbers occurs in tuples.

2.5.3 Dictionaries

A *dictionary* is a sequence of key-value pairs. Unlike the elements of a list, an element of a dictionary consists of two components: a *key value* and a *data value*. A key value and its data value are separated by a colon. The individual key-value pairs are separated by commas and enclosed in curly brackets. The first entry is the key value, while the second entry is the data value: `{key value:data value}`. Table 2.6 illustrates the dictionary data structure. The left-hand column contains the key values, while the right-hand column contains the data values.

Key Value	Data Value
unique	eindeutig
Anweisung	statement
Zuweisung	assignment
Schleife	loop
Klammern	parentheses

Table 2.6 Table for an English-German Dictionary

The following console example shows how a dictionary is implemented and how an element can be accessed:

```
>>> d={"unique":"eindeutig","statement":"Anweisung"}
>>> d["unique"]
'eindeutig'
>>> type(d)
<class 'dict'>
>>>
```

Access to the value of a dictionary is enabled by using the key value with the `[]` operator. Table 2.7 lists the most important methods for operations on dictionaries.

Method	Description
`d.keys()`	Returns the key values of the dictionary `d`.
`d.values()`	Returns the data values of the dictionary `d`.
`d.items()`	Returns a list of tuples. Each tuple contains a key-value pair from the dictionary `d`.

Table 2.7 Important Methods for Operations on Dictionaries

Method	Description
`del d[k]`	Deletes the key-value pair with the key value `k` from the dictionary `d`.
`k in d`	Checks if `k` is a key value of the dictionary `d`.

Table 2.7 Important Methods for Operations on Dictionaries (Cont.)

The first sample program shown in Listing 2.24 shows how lists can be converted to dictionaries and how basic operations on dictionaries must be implemented.

```
#24_dictionary1.py
l1=["Al","Mg"]
l2=[2.71,1.738]
l12=zip(l1,l2)
m1=dict(l12)
new={"Ti":4.5}
m1.update(new)
m2={"Fe":7.85,"V":6.12,"Mn":7.43,"Cr":7.2}
print("Key values of light metals:",m1.keys())
print("Densities of light metals:",m1.values())
print("Key values of heavy metals:",m2.keys())
print("Densities of heavy metals:",m2.values())
print("Light metals %s %i entries" %(m1,len(m1)))
print("Heavy metals %s %i entries" %(m2,len(m2)))
m2.update(m1)
print("Metals %s %i entries" %(m2,len(m2)))
del m2["V"]
print("Metals %s %i entries" %(m2,len(m2)))
print("Density of chromium: %s kg/dm^3" %(m2["Cr"]))
```

Listing 2.24 Operations on Dictionaries

Output

```
Key values of light metals: dict_keys(['Al', 'Mg', 'Ti'])
Densities of light metals: dict_values([2.71, 1.738, 4.5])
Key values of heavy metals: dict_keys(['Fe', 'V', 'Mn', 'Cr'])
Densities of heavy metals: dict_values([7.85, 6.12, 7.43, 7.2])
Light metals {'Al': 2.71, 'Mg': 1.738, 'Ti': 4.5} 3 entries
Heavy metals {'Fe': 7.85, 'V': 6.12, 'Mn': 7.43, 'Cr': 7.2}
4 entries
Metals {'Fe': 7.85, 'V': 6.12, 'Mn': 7.43, 'Cr': 7.2, 'Al': 2.71,
'Mg': 1.738, 'Ti': 4.5} 7 entries
Metals {'Fe': 7.85, 'Mn': 7.43, 'Cr': 7.2, 'Al': 2.71, 'Mg': 1.738,
'Ti': 4.5} 6 entries
Density of chromium: 7.2 kg/dm^3
```

Analysis

Line 02 creates a list (i.e., list `l1`), which contains two light metals. Line 03 generates a list (i.e., list `l2`) containing the corresponding densities. In line 04, the `zip()` method is used to join both lists into a new list: `l12`. The `dict()` method converts list `l12` to a dictionary in line 05. In line 06, a new dictionary is created with only one key-value pair, `{"Ti":4.5}`. In line 07, this element is inserted into dictionary `m1`. In line 08, a new dictionary is created containing heavy metals `m2`. In lines 09 and 10, the key values are output using the `m1.keys()` method, while the data values are output via the `m1.values()` method of the `m1` light metals. The same statements apply to heavy metals `m2` (lines 11 and 12). Lines 13 and 14 output the key-value pairs of the light metals and heavy metals. The `m2.update(m1)` method in line 15 merges the two dictionaries `m1` and `m2` into a new dictionary named `m2`. The dictionary, which now contains the light and heavy metals, is output in line 16. The `del()` method deletes vanadium from dictionary `m2` in line 17. The output in line 18 confirms the deletion. Line 19 illustrates once again how the `[]` operator is used to access the key of a dictionary.

A Dictionary

Listing 2.25 shows how an English–German and German–English dictionary is created from two lists.

```
01  #25_dictionary2.py
02  e=["unique","statement","assignment","loop","parentheses"]
03  d=["eindeutig","Anweisung","Zuweisung","Schleife","Klammern"]
04  e2d=dict(zip(e,d))
05  d2e=dict(zip(d,e))
06  print("statement:", e2d["statement"])
07  print("Schleife:", d2e["Schleife"])
```

Listing 2.25 A Simple Dictionary

Output

```
statement: Anweisung
Schleife: loop
```

Analysis

Line 02 contains a list of English words. Line 03 contains a list of German words. In line 04, a dictionary for an English–German translation is created. In line 05, a dictionary for the German–English translation is created. In lines 06 and 07, the translations are output. The special trick behind this dictionary is that only one list is created for one language at a time. If we were to proceed intuitively, we would probably implement two dictionaries, each with two entries (key value and data value).

2.5.4 Sets

A *set* is an unordered collection of elements that can be iterated and modified. A set never contains any duplicate elements. Sets are defined like a dictionary using curly brackets. Empty curly brackets create an empty `dict`, not an empty `set`. The `set` Python class implements the notion of sets known from mathematics. Consequently, the three set operations—union (`&` operator), intersection (`-` operator), and difference quantity (`|` operator)—are possible. For these set operations, the following methods are also available: `s1.union(s2)`, `s1.intersection(s2)`, and `s1.difference(s2)`, as described in Table 2.8. Compared to the `list` data structure, the main advantage of a set is that it has a highly optimized method for checking whether a particular object is included in the set.

Let's get a first impression of sets using the Python shell:

```
>>> s={1,2,3}
>>> s
{1, 2, 3}
>>> s.add(23)
>>> s
{1, 2, 3, 23}
>>> type(s)
<class 'set'>
```

A set is created by enclosing the comma-separated numbers in curly brackets. The `add` method adds the element `23` to the set `s`. A type check reveals that `s` is an object of the `set` class. Table 2.8 contains the most important methods for operations on sets.

Methods	Description
`s.add(e)`	Inserts the element `e` as a new element into the set `s`.
`s.clear()`	Removes all elements from the set `s`.
`s.copy()`	Copies the set `s`.
`s.discard(e)`	The element `e` is removed from the set `s`.
`s1.difference(s2)`	Calculates the difference of the two quantities `s1` and `s2`.
`s1.intersection(s2)`	Returns the intersection of `s1` and `s2`.
`s1.union(s2)`	Forms the union of `s1` and `s2`.

Table 2.8 Methods for Operations on Sets

Listing 2.26 forms the intersection, the difference, and the union of two quantities.

```
01 #26_sets.py
02 set1={"A","B","C","D"}
03 set2={"C","D","E","F"}
04 intersection= set1 & set2
05 difference=set1 - set2
06 union=set1 | set2
07 print("Set1:",set1)
08 print("Set2:",set2)
09 print("Intersection:",intersection)
10 print("Difference:",difference)
11 print("Union:",union)
12 print("Is B contained in set1:", set1.issuperset("B"))
```

Listing 2.26 Operations on Sets

Output

```
Set1: {'B', 'D', 'A', 'C'}
Set2: {'F', 'E', 'D', 'C'}
Intersection: {'D', 'C'}
Difference: {'B', 'A'}
Union: {'B', 'F', 'D', 'C', 'E', 'A'}
Is B contained in set1? True
```

Analysis

Two sets are defined in lines 02 and 03. The set operations are performed in lines 04 to 06. Line 12 checks if element `B` is contained in `set1`.

2.6 Functional Program Style

The first functional programming language, Lisp, was developed as early as 1958 at the Massachusetts Institute of Technology (MIT) by John McCarthy. After Fortran, it is considered the second oldest programming language. The design of the language is not based on the architecture of a computer, as is the case, for example, with Fortran, but instead on how a mathematically trained programmer thinks.

The functional programming style can be characterized in the following way:

- Programs are composed of functions.
- Functions are not represented as a sequence of statements but as nested function calls instead.
- Functions can also be passed back to other functions as arguments.

- Functions can also be defined without explicit naming. This type of function is also referred to as a `lambda` function.
- For elementary mathematical operations, the prefix notation is used.

To illustrate the functional programming style in an example, I want to use the calculation of the moment of inertia and acceleration torque of a solid cylinder again. This time, however, we won't start with a Python program, but with a "real" functional programming language. Listing 2.27 is written in the Racket programming language, which is a Lisp derivative. The fact that Python does not support a true functional programming style becomes clear when you compare these two programming languages. In addition, a common approach to learning a programming language is to highlight the differences among them.

```
#lang racket
;cylinder.rkt
(define (volume d l)
         (* 0.785 d d l))

(define (mass d l)
         (* 7.85 (volume d l)))

(define (momentofinertia d l)
         (* 0.5 (mass d l) 0.25e-3 d d))

(define (accelerationtorque d l alpha)
         (* alpha  0.5 (mass d l) 0.25e-3 d d))

(display "Volume: ")(writeln (volume 1 10))
(display "Mass: ")(writeln (mass 1 10))
(display "Moment of inertia:")(writeln (momentofinertia 1 10))
(display "Acceleration torque: ")
(writeln(accelerationtorque 1 10 1.2))
```

Listing 2.27 Functional Program Using Racket

Output

```
Volume: 7.8500000000000005
Mass:   61.6225
Moment of inertia: 0.0770281
Acceleration torque:   0.0924337
```

Analysis

The function definitions are made using the define keyword. What is noticeable in this context is the high number of parentheses. Suitable development environments, such as DrRacket, are available if you ever want to develop professional applications in Racket.

The function arguments are not enclosed in parentheses, only the functions themselves. Formatting the source code is not mandatory. You could also place all the functions on one line. To improve clarity, the definition part and the calculation part should be separated from each other. The prefix notation, also referred to as the *Polish notation*, is also unusual. Polish mathematician Jan Łukasiewicz used prefix notation in the 1920s to describe mathematical propositional logic in a more compact way. To those accustomed to school mathematics, Polish notation may seem difficult to understand. But enthusiasts of functional programming languages defend it as particularly simple and elegant because not only is it clearer, but also much easier to handle than infix notation.

The outputs are realized either using the display or writeln keywords. The units have been deliberately omitted in our example.

Functional programs are implemented in Python using the lambda operator. The lambda calculus was originally introduced in the 1930s by Alonzo Church and Stephen Cole Kleene for the description of function definitions. John McCarthy used this concept in the late 1950s to define the functions of the functional programming language Lisp. The general syntax of a lambda function looks as follows:

```
lambda param1, param2, param3: calculation rule
```

The Python version programmed in a functional programming style is implemented in Listing 2.28. Compared to the previous Python variants, this version turns out to be particularly compact as only one line is needed for each function.

```
#28_functional.py
rho=7.85 #kg/dm^3
volume=lambda d,l: 0.785*d**2*l
mass=lambda d,l: rho*volume(d,l)
moment_of_inertia=lambda d,l: 0.5*mass(d,l)*(d/2/10)**2
acceleration_torque=lambda d,l,omega:omega*moment_of_inertia(d,l)
#Output d and l in dm
print("Volume:",volume(1,10), "dm^3")
print("Mass:",mass(1,10),"kg")
print("Moment of inertia:",moment_of_inertia(1,10),"kgm^2")
print("Acceleration torque:",acceleration_torque(1,10,1.2),"Nm")
```

Listing 2.28 Functional Program Using Python

Output

```
Volume: 7.8500000000000005 dm^3
Mass: 61.6225 kg
Moment of inertia: 0.07702812500000002 kgm^2
Acceleration torque: 0.09243375000000002 Nm
```

Analysis

In lines 03 to 06, the functions are defined using the `lambda` operator. Because these functions are not given a name, they are also called *anonymous functions*. Directly after the `lambda` operator and separated by commas are the formal parameters. The colon is followed by the calculation rules. Anonymous functions are treated like normal expressions. For this reason, they can also be assigned to variables.

The output is shown in lines 08 to 11. Inside the `print` functions, the variables are treated like normal function calls with current parameter passes.

2.7 Object-Oriented Program Style

In the 1960s, as the size of software projects increased, so did their complexity. During the test phases, unforeseen effects occurred, and the programs did not provide the desired results. One answer to the question of how to reduce complexity was the invention of *object-oriented programming (OOP) languages*.

Simula is considered to be the first OOP language. It was developed by Ole-Johan Dahl and Kristen Nygaard in the 1960s at the Norsk Regnesentral (Norwegian Computing Center) at the University of Oslo to simulate physical processes on a computer.

Alan Kay took up the basic ideas of the Simula programming language and developed the OOP language Smalltalk in the 1970s, which was equipped with an extensive class library in the course of its development process. His definition of OOP contains six criteria:

1. *Everything is an object.*
2. *Objects communicate by sending and receiving messages (in terms of objects),*
3. *Objects have their own memory (in terms of objects),*
4. *Every object is an instance of a class (which must be an object),*
5. *The class holds the shared behavior for its instances (in the form of objects in a program list),*
6. *To eval a program list, control is passed to the first object and the remainder is treated as its message.*

("The Early History of Smalltalk," 1993, abridged)

Python has adopted criteria 1 to 5. The basic idea is simple: All data and operations are combined into one unit, the object. This principle is referred to as *data encapsulation*. In the terminology of current OOP, instead of the terms *data* and *operations*, the terms *attributes* and *methods* are used, where a method is just another name for the already familiar term *function*. The syntax of methods and functions is completely consistent. In a nutshell:

Object = Attributes + Methods

The concept behind OOP has three goals in particular:

- **The principle of reusability**
  Once defined, classes are supposed to be reusable in other software projects.
- **The principle of data encapsulation**
  The fact that extensive programs with many thousands of program lines (statements) are divided into clearly arranged classes is supposed to reduce the complexity. As a result, software projects become more manageable. Each programmer can freely choose their own variables for their classes without mutual interference (side effects) during program execution.
- **Increased maintainability**
  For example, if a more effective algorithm with a better runtime has been found for certain methods of a class, it can be re-implemented in its class as a method with the same name as its predecessor without having to change the main program.

2.7.1 Objects and Classes

The class definition for a solid cylinder clearly explains the concept of OOP. Cylinders play an important role in drive technology (e.g., as drive rollers or rope winches, for the production of films). They occur in the real world in an infinite variety. For IT-based modeling, the diversity of real circumstances must be abstracted from, while only the properties relevant for the calculations are selected. If the rotational frequency of a cylinder must be controlled, then its moment of inertia must be known for the calculation of the acceleration torque. The moment of inertia is determined by the density, diameter, and length of the cylinder. If you then combine these three attributes with the methods for calculating the moment of inertia to form a self-contained unit, then we speak of a *class*. A class is a custom abstract data type (ADT) that contains all properties (attributes) and all arithmetic operations (methods) that can be applied to the objects of this class.

Class

A class combines data (properties) and methods (functions). Classes are the smallest units of an object-oriented program. A class definition describes how objects are constructed and which operations can be performed on them.

According to the *unified modeling language (UML)* notation, classes are represented as *class diagrams*.

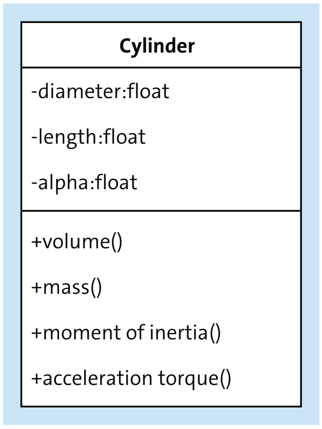


Figure 2.7 Class Diagram for the Cylinder Class

Figure 2.7 shows the class diagram for the `Cylinder` class. A class diagram consists of a rectangle divided into three horizontal areas. The upper rectangle contains the name of the class. The properties are listed in the middle area. The negative sign means that the variables must not be changed from the outside. Formulated in the technical language of OOP, these variables are defined as `private`. This concept is called data encapsulation.

Data Encapsulation

Data encapsulation is the prevention of uncontrolled access to the properties (the data) of a class.

The lower area contains the methods of the class. The positive sign identifies the methods as *public*, which means that they are accessible from the outside, that is, from outside the class definition.

Methods

The Python functions defined within a class are called methods.

An object-oriented program always consists of a definition part, which contain class definitions, and an execution part. In the execution part, the objects are created by an assignment:

```
objName = class(parameterlist)
```

On the left-hand side of the assignment operator is a freely selectable identifier. On the right-hand side of the assignment operator is the name of the class with the list of parameters enclosed in parentheses.

The methods of a class are accessed using a dot operator in the following way:

```
objName.method()
```

Object

An *object* is a symbolically addressed memory area in which all data and methods of the class definition are stored. An object is an instance of a class. Every object has a name, and via this name, the methods of a class can be accessed. Any number of objects can be created from one class.

When an object is created, what's referred to as a *constructor* is called:

```
def __init__(self, paramterlist):
```

The two underscores mark Python internal special functions. In this case, this special function is about the initialization of the properties (hence `init`).

Constructor

A constructor is a special method that is called when an object is created. This special method takes care of the initialization of the properties.

Listing 2.29 shows how the `Cylinder` class is implemented. The volume, mass, moment of inertia, and acceleration torque of a solid cylinder are calculated.

```
#29_oop.py
class Cylinder:
    rho=7.85
    def __init__(self,diameter,length,alpha):
        self.__d=diameter #private
        self.__l=length   #private
        self.__a=alpha    #private

    def volume(self):
        return 0.785*self.__d**2*self.__l

    def mass(self):
        return self.rho*self.volume()

    def moment_of_inertia(self):
        return 0.5*self.mass()*(self.__d/2/10)**2

```

```
18      def accelerationtorque(self):
19          return self.__a*self.moment_of_inertia()
20  #Main program d and l in dm
21  z=Cylinder(1,10,1.2)
22  #Cylinder.rho=2.3
23  #z.__d=100
24  print("Volume:",z.volume(),"dm^3")
25  print("Mass:  ",z.mass(),"kg")
26  print("Moment of inertia:  ",z.moment_of_inertia(),"kgm^2")
27  print("Acceleration torque:",z.accelerationtorque(),"Nm")
```

Listing 2.29 OOP Program Using Python

Output

```
Volume: 7.8500000000000005 dm^3
Mass:  61.6225 kg
Moment of inertia:   0.07702812500000002 kgm^2
Acceleration torque: 0.09243375000000002 Nm
```

Analysis

The class definition, shown in lines 02 to 19, is preceded by the `class` keyword. A class name should always start with an uppercase letter, as convention demands. The header of a class definition is terminated with a colon.

In line 03, the *class variable* `rho` is defined. The namespace of these variables spans the entire class, which means that all methods can use them in their calculations. A class variable with the notation `cylinder.rho=2.3` (line 22) allows for external write access.

In lines 04 to 07, the `__init__()` method is defined. This method is introduced and concluded with two underscores and followed by the parameter list enclosed in parentheses with the `self` parameter and the parameters of the `diameter`, `length`, and `alpha` attributes. The `self` parameter is not a keyword, and thus, its name can be freely chosen. However, the convention is to use this identifier. In the function body of the `__init__` function, the assignments follow the pattern `self.__d = diameter` (line 05). All variables that are prefixed with a `self` are called *instance variables*. As a result, each newly created object (line 21) gets its own namespace. The two underscores in front of an instance variable have the effect that these variables are declared as `private`; that is, they cannot be modified from the outside (through the principle of data encapsulation). If, for example, you remove the comment in line 23, the intended result won't change. If, on the other hand, the underscores of the instance variables are removed, the value for the diameter can still be changed in line 23. Just try it!

The methods of the `Cylinder` class are defined from line 09 onwards. What is new compared to the usual function definition is that only `self` is passed as a parameter. In addition, the instance variables and methods with the self parameter as prefix are connected by the dot operator. This notation causes a separate namespace to be formed for each method whenever a new object is created.

In line 21, the z object is created by calling the Cylinder(1,10,1.2) method. A method that bears the name of the class is called a *constructor*. Unlike C++ and Java, Python's language concept does not include an explicit constructor. When an object is created, the implicit constructor is started first, and the init method is called immediately afterwards. The constructor forms a clearly defined interface to the "outside world" with its parameters.

Instance variables should only be addressed through such an interface. The z object is a copy of the Cylinder class. This object can be used to access the methods of the Cylinder class via the dot operator (lines 24 to 27). Many other objects of the Cylinder class can be re-created with different current parameters. All instance variables and methods are then each assigned their own namespace. The access of an object to a method of the Cylinder class can also be interpreted in this way: The object z sends the message "Calculate the acceleration torque" to the accelerationtorque() method of the Cylinder class, and the method, thus addressed, returns the response "Here is the result."

2.7.2 Inheritance

Inheritance is another important concept in OOP. The basic idea is again the reusability of source code.

This concept can be confusing at first because the derived classes not only take over the properties of the base class, but they also extend them. More vividly, you can think of inheritance as a *takeover* or an *extension*. A derived class inherits attributes and methods from one or more base classes.

Inheritance

A base class makes its properties and methods available to other classes (the *derived classes*).

Listing 2.30 shows the mechanism of inheritance through an example—calculating the volume of a cuboid.

```
#30_inheritance.py
class Area:

    def __init__(self,width,length):
        self.w=width
        self.l=length
```

```
07
08      def area(self):
09          return self.w*self.l
10
11  class Volume(Area):
12
13      def __init__(self,width,length,height):
14          Area.__init__(self,width,length)
15          #super().__init__(width,length)
16          self.h=height
17
18      def volume(self):
19          return Area.area(self)*self.h
20          #return super().area()*self.h
21
22  A=Area(1,2)
23  V=Volume(1,2,3)
24  print("Area:  ",A.area()," m^2")
25  print("Volume:",V.volume()," m^3")
```

Listing 2.30 Inheritance

Output

```
Area:   2  m^2
Volume: 6  m^3
```

Analysis

The base class `Area` (line 02 to 09) contains the `area()` method for the calculation of a rectangular area. Starting from line 11, the derived class `Volume` takes over the attributes `width` and `length` as well as the `area()` method from the base class. The fact that a class inherits from another class is communicated to the Python interpreter by passing the name of the base class as a parameter to the derived class (line 11). The `__init__` method in line 13 requires the width, length, and height of a cuboid as parameters. The `__init__` method of the base class is accessed either through the name of the base class `Area` (line 14) or through the built-in function `super()` in line 15. In line 19 or line 20, the volume is calculated according to the well-known formula “base area multiplied by height.” Line 22 creates an object `A` for the base area, and line 23 creates an object `V` for the volume calculation. These two objects can be used to access the methods of the `Area` and `Volume` classes using the dot operator in the `print()` function (lines 24 and 25).

2.8 Project Task: Dimensions of a Shaft

For a shaft subjected to deflection only, as shown in Figure 2.8, the minimum diameter, static deflection, and critical rotational frequency need to be calculated. The load caused by twisting (torsion) can be neglected.

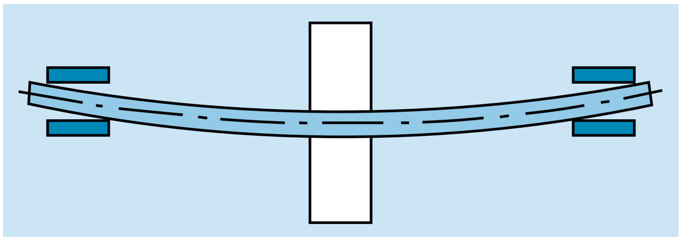


Figure 2.8 Deflected Shaft

Given is the mass of the load $m_a = 1$ kg, the modulus of elasticity $E = 216 \cdot 10^3$ N/mm^2, the length of the shaft $l = 120$ mm, and the maximum permissible bending stress $\sigma_B = 100$ N/mm^2.

The minimum diameter d, the static deflection f, and the critical rotational speed n_k are to be determined.

For the calculation of the deflection and the bending stiffness, we need the second moment of area I_a:

$$I_a = \frac{\pi d^4}{64}$$

If the second moment of area is divided by half the shaft diameter, we obtain the axial section modulus W. This value is needed for the calculation of the shaft diameter.

$$W = \frac{\pi d^3}{32}$$

Using the bending main equation, the bending stress σ_B of the shaft can then be used to calculate the minimum diameter of the shaft.

$$\sigma_B = \frac{M_b}{W}$$

$$d \geq \sqrt[3]{\frac{32 M_b}{\pi \sigma_B}}$$

If the force acts exactly in the center of the shaft, the maximum bending moment is calculated with the following formula:

$$M_b = \frac{Fl}{4}$$

For the deflection in the center of the shaft, the following formula applies:

$$f = \frac{Fl^3}{48EI_a}$$

Based on the deflection, the bending stiffness $R_b = F/f$ can be determined:

$$R_b = \frac{48EI_a}{l^3}$$

The critical rotational speed n_k is calculated from the root of the quotient of the bending stiffness R_b and the mass m:

$$n_k = \frac{1}{2\pi}\sqrt{\frac{R_b}{m}}$$

All formulas can be transferred directly into Python source code, as shown in Listing 2.31.

```
#31_project_shaft.py
from math import sqrt,pi
g=9.81      #Acceleration due to gravity
rho=7.85    #kg/dm^3
E=216e3     #N/mm^2
l=120       #mm
sigma=100   #N/mm^2
m=1         #kg
#Calculations
F=m*g
Mb=F*l/4
d=pow((32*Mb)/(pi*sigma),1/3)
d=round(d+0.5)
Ia=pi*d**4/64.0
f=F*l**3/(48*E*Ia)
Rb=48*E*Ia/l**3     #F/f
nk=sqrt(1e3*Rb/m)/(2*pi)
#Outputs
print("Diameter in mm:",round(d,2))
print("Deflection in mm:",round(f,3))
print("Critical rotational speed 1/min:",int(60*nk))
```

Listing 2.31 Dimensioning a Shaft

Output

```
Diameter in mm: 4
Deflection in mm: 0.13
Critical rotational speed 1/min: 2622
```

Analysis

The inputs are implemented as assignments in lines 03 to 08. If several different parameters are to be tested, these assignments can be replaced by `input` functions.

The individual calculations are performed in lines 10 to 17. In line 12, the cube root is calculated using the `pow` function. The rounding function `round(d+0.5)` in line 13 ensures that the next larger integer diameter is determined.

Outputs are shown in lines 19 through 21. The program calculates a diameter of 4 mm for the given sizes. At this diameter and the load with the mass of 1 kg, the shaft deflects by 0.13 mm. Dangerous resonance effects occur at the critical rotational speed of 2622 1/min. Due to the increased deflection, the shaft may break at this speed.

2.9 Tasks

1. Write a Python program that calculates the air resistance of a car (or bicycle) in complete calm according to this formula:

   $$F = \frac{1}{2}\rho c_w A v^2$$

   Furthermore, the program should calculate the drive power and the work done for a given travel time.
2. Formulate the following mathematical expressions as Python source code:

   $$a^2 + b^2 = c^2$$

   $$m = \frac{m_0}{\sqrt{1 - \frac{v^2}{c^2}}}$$

   $$y = \ln \cosh x$$

   $$y = \frac{1}{a}\arctan\frac{x}{a}$$
3. The central difference quotient is to be calculated for an arbitrarily differentiable function. Write a Python program that calculates the slope and angle of the secant.
4. Write a Python program for calculating a root using the Heron algorithm (Babylonian root extraction).
5. Write a Python program that calculates the greatest common divisor of two numbers.
6. The integration by rectangle sums can be done either by lower or upper sums. Optimize this procedure by selecting the center of the interval for a rectangle.

   $$\left[f\left(\frac{x_k + x_{k+1}}{2}\right)\right]$$

7. For the function $z = f(x,y) = 5 - x - y$, the volume is to be calculated numerically with a double integral. In the direction of the x-axis, you should integrate from 0 to 2, and in the direction of the y-axis, from 0 to 1. Write a program with two nested `for` loops that meets these requirements.
8. Write a Python program that, when given a trigonometric function, outputs the root function of that function as a string. Use the dictionary data structure for the solution.
9. Write a program as an object-oriented version for the calculation of the air resistance of a passenger car (formula from task 1).
10. Implement the computation of the acceleration torque of a cylinder as classes with inheritance (M_b inherits from J, J inherits from m, and m inherits from V).

Chapter 3
Numerical Calculations Using NumPy

In this chapter, you'll learn how to perform operations on vectors and matrices and solve systems of linear equations using NumPy.

The acronym NumPy stands for **num**eric **Py**thon. As this name suggests, this module provides functions for numerical calculations. Besides the number of functions provided, the short runtime of the NumPy functions is particularly noteworthy. You should always import the NumPy module using the `import numpy as np` import statement. Assigning the `np` alias has become the accepted convention. NumPy forms the basis for almost all scientific calculations and is therefore often used in combination with the Matplotlib and SciPy modules.

3.1 NumPy Functions

The most commonly used NumPy functions are `arange()` and `linspace()`. Both functions create *one-dimensional* arrays of length *n*. During runtime, *n* can no longer be modified. If you choose the `arange()` function, the distances between the array elements will be defined; on the other hand, if you choose the `linspace()` function, the number of array elements will be defined. One of the two functions occurs in every Matplotlib program for generating the independent variables in value tables. The NumPy function `array()` creates a *two-dimensional* array when given a nested list as an argument. Furthermore, NumPy also provides trigonometric, hyperbolic, and logarithmic functions as well as important statistical functions.

3.1.1 Creating One-Dimensional Arrays Using arange() and linspace()

The NumPy functions `arange()` and `linspace()` can be used to create one-dimensional arrays of a given length. The data type of the elements of an array must be uniform. The general syntax for `arange()` is as follows:

```
np.arange(start,stop,step,dtype=None)
```

You don't need to specify the data type, which is determined automatically by NumPy. As a rule, numbers of the float type are processed. Three different float data types are possible:

- **Float16**: Half precision with 10-bit mantissa and 5-bit exponent
- **Float32**: Single precision with 23-bit mantissa and 8-bit exponent
- **Float64**: Double precision with 52-bit mantissa and 11-bit exponent

The `linspace()` function specifies the number of elements (`num`) instead of the increment (`step`). The default value is 50. The general syntax for `linspace()` is:

```
linspace(start,stop,num=50,endpoint=True,retstep=False, dtype=None, axis=0)
```

Listing 3.1 compares both functions with each other.

```
01  #01_1dim_array.py
02  import numpy as np
03  x1=list(range(10))
04  x2=np.arange(10)
05  x3=np.arange(1,10,0.5)
06  x4=np.linspace(1,10,10)
07  x5=np.linspace(1,10,10,endpoint=False)
08  print("Python list:",type(x1) ,"\n",x1)
09  print("arange() Increment 1:",type(x2),"\n",x2)
10  print("arange() Increment 0.5:",type(x3),"\n",x3)
11  print("linspace() Increment 1:",type(x4),"\n",x4)
12  print("linspace() Increment 0.9:",type(x5),"\n",x5)
```

Listing 3.1 Arrays with arange() and linspace()

Output

```
Python list: <class 'list'>
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
arange() Increment 1: <class 'numpy.ndarray'>
[0 1 2 3 4 5 6 7 8 9]
arange() Increment 0.5: <class 'numpy.ndarray'>
[1. 1.5 2. 2.5 3. 3.5 4. 4.5 5. 5.5 6. 6.5 7. 7.5 8. 8.5 9. 9.5]
linspace() Increment 1: <class 'numpy.ndarray'>
[ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
linspace() Increment 0.9: <class 'numpy.ndarray'>
[1. 1.9 2.8 3.7 4.6 5.5 6.4 7.3 8.2 9.1]
```

Analysis

Line 03 generates the list `x1` from Python function `range(10)`. Line 08 outputs the numbers from 0 to 9 for `x1`. The preset increment is 1.

Line 04 creates an array `x2` with NumPy function `arange()`. Line 09 also outputs the numbers from 0 to 9 for `x2`. The increment of 1 is also preset.

Line 05 creates a NumPy array x3 with increment 0.5. The final value is not output (line 10).

In lines 06 and 07, two arrays are created via NumPy function linspace(). If the endpoint=False property is set, then the last element won't be output, and the increment is 0.9 (line 12). The default is endpoint=True.

The NumPy functions arange() and linspace() are of type numpy.ndarray. The nd prefix stands for multi-dimensional arrays (*n-dimensional*).

Runtime of arange() and linspace()

A particular advantage of NumPy functions is said to be their *short* runtimes. Listing 3.2 calculates and compares the runtimes of a Python list with the runtimes of the NumPy functions arange() and linspace(). All three functions—version1(n), version2(n), and version3(n)—add 1 million numbers from two arrays element by element. The runtime is determined using the time() function from Python module time.

```
#02_runtime_comparison.py
import time as t
import numpy as np
#Python list
def version1(n):
    t1=t.time()
    x1=list(range(n)) #generate list
    x2=list(range(n))
    sum=[]
    for i in range(n):
        sum.append(x1[i]+x2[i])
    return t.time() - t1
#NumPy arange()
def version2(n):
    t1=t.time()
    x1=np.arange(n)
    x2=np.arange(n)
    sum=x1+x2
    return t.time() - t1
#NumPy linspace()
def version3(n):
    t1=t.time()
    x1=np.linspace(0,n,n)
    x2=np.linspace(0,n,n)
    sum=x1+x2
    return t.time() - t1

nt=1000000
```

```
runtime1=version1(nt)
runtime2=version2(nt)
runtime3=version3(nt)
factor1=runtime1/runtime2
factor2=runtime1/runtime3
#Output
print("Runtime for Python range()...:",runtime1)
print("Runtime for NumPy  arange()..:",runtime2)
print("Runtime for NumPy linspace():",runtime3)
print("arange()   is%4d times faster than range()" %factor1)
print("linspace() is%4d times faster than range()" %factor2)
```

Listing 3.2 Runtime Comparison

Output

```
Runtime for Python range()...: 0.1445789337158203
Runtime for NumPy  arange()..: 0.0028200149536132812
Runtime for NumPy  linspace(): 0.0020291805267333984
arange()   is 51 times faster than range()
linspace() is 71 times faster than range()
```

Analysis

The NumPy function `arange()` is about 51 times faster, and the NumPy function `linspace()` is about 71 times faster than the Python list generated by the Python function, `range()`. The time measurements are only rough estimates. With each new program start and with different hardware, the results will turn out differently.

In conclusion, for the numerical analysis of large data sets, you should use NumPy arrays.

3.1.2 Creating Two-Dimensional Arrays Using array()

Up to this point, only one-dimensional arrays were created using the NumPy functions `arange()` and `linspace()`. In real life, for example, for the calculation of electrical networks or for the solution of linear systems of equations, two-dimensional arrays are also required. Two-dimensional arrays are created from nested lists using the NumPy function, `array()`. You should use only the `array()` function for calculations involving matrices because the `matrix()` function will be removed from the NumPy module in the future.

Matrices

Matrices are represented by NumPy arrays.

The following console example demonstrates the difference between a one-dimensional array and a two-dimensional array:

```
>>> import numpy as np
>>> a=np.array([1,2,3])
>>> a
array([1, 2, 3])
>>> b=np.array([[1,2,3],[4,5,6]])
>>> b
array([[1, 2, 3],
       [4, 5, 6]])
>>> a.ndim
1
>>> b.ndim
2
>>> type(b)
<class 'numpy.ndarray'>
```

The `array` function is a *member of* the `ndarray` class, as are the NumPy functions `arange()` and `linspace()`. To test the operations on arrays, it is convenient to automate the creation of two-dimensional arrays. You can use the NumPy method `obj.reshape()` to convert a one-dimensional array into a two-dimensional array. Listing 3.3 creates an $m \times n$ matrix from a one-dimensional array. The program also determines the *type* (i.e., the shape) of the array with the `shape` property and shows how a matrix is transposed.

```
#03_2dim_array.py
import numpy as np
m=3 #lines
n=4 #columns
a=np.arange(m*n).reshape(m,n)
b=a.reshape(n*m,)
print("Type of the array",a.shape,"\n",a)
print("Linearize\n",b)
print("Transpose\n",a.T)
```

Listing 3.3 Generating a Two-Dimensional Array

Output

```
Type of the array (3, 4)
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
Linearize
 [ 0  1  2  3  4  5  6  7  8  9 10 11]
Transpose
```

```
[[ 0  4  8]
 [ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]]
```

Analysis

In lines 03 and 04, you can change the number of lines and columns of the array.

In line 05, the NumPy method `reshape(m,n)` converts the one-dimensional array into a two-dimensional array.

In line 06, the `a.reshape(n*m,)` method linearizes the two-dimensional array `a`. The `a.reshape(m,n)` statement is called a method here because `reshape()` requires an object to execute. The object notation `a.reshape(m,n)` can be translated into everyday language using the phrase "create an array object with `m` lines and `n` columns from array object `a`."

In line 07, the `shape` property determines the type of array `a`.

In line 09, array `a` is transposed via `a.T`. You can also transpose an array using the `np.transpose(a)` statement.

3.1.3 Slicing

Slicing allows you to read selected portions of elements from a two-dimensional array. With the general syntax `a[start:stop:step,start:stop:step]`, a subrange of an array `a` defined by the parameters `start,stop,step` is read from the *m*-th line and the *n*-th column. The default value of `step` is 1. Using `a[m,:]` you can read the *m*-th line, and using `a[:,n]`, you can read the *n*-th column of the array `a`. Listing 3.4 shows in a 4×4 matrix how slicing works for reading columns. The matrix is created using the NumPy method `reshape()`.

```python
#04_slicing.py
import numpy as np
m=4 #lines
n=4 #columns
a=np.arange(m*n).reshape(m,n)
#Output
print(a)
print("First column\n",a[:,0])
print("Second column\n",a[:,1])
print("First line\n", a[0,:])
print("Second line\n", a[1,:])
print("a[1:3,0:2]\n",   a[1:3,0:2])
```

Listing 3.4 Slicing

Output

```
[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
First column
 [ 0  4  8 12]
Second column
 [ 1  5  9 13]
First line
 [0 1 2 3]
Second line
 [4 5 6 7]
a[1:3,0:2]
 [[4 5]
 [8 9]]
```

Analysis

In lines 03 and 04, the number of lines and columns for the matrix created in 05 is defined. In line 05, the NumPy method `reshape(m,n)` creates a 4×4 matrix from a sequence of 16 integers.

In lines 08 to 11, individual columns and lines are read. Note that the count starts at index 0.

In line 12, a range of the matrix is read.

3.1.4 Mathematical NumPy Functions

NumPy provides the same mathematical functions that are known from the Python module `math`. But only mathematical functions from the NumPy module may be used when passing arguments from a NumPy array to these functions, as shown in Listing 3.5.

```python
#05_numpy_functions.py
import numpy as np
#import math
x=np.arange(-3,4,1)
#y1=math.sin(x)
y1=np.sin(x)
y2=np.exp(x)
y3=np.sinh(x)
y4=np.cosh(x)
y5=np.hypot(3,4)#diagonal
```

```
11 y1,y2,y3,y4=np.round((y1,y2,y3,y4),decimals=3)
12 #Output
13 print("x values:\n",x)
14 print("sin function:\n",y1)
15 print("e-function:\n",y2)
16 print("sinh function:\n",y3)
17 print("cosh function:\n",y4)
18 print("Hypotenuse:",y5)
```

Listing 3.5 Selected Mathematical NumPy Functions

Output

```
x values
 [-3 -2 -1 0 1 2 3]
sin function:
 [-0.141 -0.909 -0.841 0. 0.841 0.909 0.141]
e-function:
 [0.05 0.135 0.368 1. 2.718 7.389 20.086]
sinh function
 [-10.018 -3.627 -1.175 0. 1.175 3.627 10.018]
cosh function
 [10.068 3.762 1.543 1. 1.543 3.762 10.068]
Hypotenuse: 5.0
```

Analysis

The program calculates value tables for a `sin`, an `e`, a `sinh`, and a `cosh function`. The value range is between -3 and +3 (line 04). The increment is 1. The fact that the upper limit for the *x* values breaks off at +3, although 4 was specified as the upper limit in the source code, may be confusing at first. The NumPy documentation provides the explanation: For both integer and non-integer increments, the interval end of the value range is not included. Thus, $x < 4$ is always valid. In exceptional cases, rounding effects may cause the end of the interval to be included.

NumPy functions can be accessed via the dot operator with the `np` alias. If the comments in lines 03 and 05 are removed, the following error message appears after the program start:

```
y1=math.sin(x)
TypeError: only size-1 arrays can be converted to Python scalars
```

This message means that value tables for the mathematical functions from the `math` module may only be created with loop constructs. For each new calculation of a function value for `math.sin(x)`, the loop must be run again. If, on the other hand, value tables are created using the NumPy functions `arange()` or `linspace()` and the predefined mathematical functions from the NumPy module, then a `for` or `while` loop is no longer needed. All mathematical NumPy functions return an `ndarray`. Each discrete value of the variables (i.e., `y1` to `y4`) can thus be accessed via the index operator. The expenditures on lines 14 to 17 demonstrate this result: For each `x` argument, the corresponding function value is output.

The statement in line 11 is interesting. At this point, the NumPy function `round()` rounds the outputs for all four function values to three digits by passing it a tuple of four elements. The `round()` function returns a tuple with four elements as well.

3.1.5 Statistical NumPy Functions

NumPy also provides functions for generating uniformly and normally distributed random numbers. Using statistical NumPy functions, you can calculate the arithmetic mean, median, variance, and standard deviation from these numbers. These statistical functions are also provided by the Python `statistics` module by default. However, NumPy's statistical functions are much more powerful. For this reason, you should prefer using NumPy functions when statistically analyzing large volumes of data. Listing 3.6 shows how you can use these functions.

```python
#06_numpy_statistics.py
import numpy as np
lines=5
columns=10
np.random.seed(1)
x=np.random.normal(8,4,size=(lines,columns))
mw=np.mean(x)
md=np.median(x)
v=np.var(x)
staw=np.std(x)
minimum=np.amin(x)
maximum=np.amax(x)
min_index=np.where(x==np.amin(x))
max_index=np.where(x==np.amax(x))
#min_index=np.argmin(x)
#max_index=np.argmax(x)
#Output
print("Random numbers\n",np.round(x,decimals=2),"\n")
print("Smallest number..........:",minimum)
print("Largest number...........:",maximum)
print("Index of the smallest number:",min_index)
print("Index of the largest number..:",max_index)
print("Mean....................:",mw)
```

```
24 print("Median.................:",md)
25 print("Variance...............:",v)
26 print("Standard deviation......:",staw)
27 print("Type of  x:",type(x))
28 print("Type of mw:",type(mw))
```

Listing 3.6 Statistical NumPy Functions

Output

```
Random numbers
 [[14.5   5.55  5.89  3.71 11.46 -1.21 14.98  4.96  9.28  7.  ]
 [13.85 -0.24  6.71  6.46 12.54  3.6   7.31  4.49  8.17 10.33]
 [ 3.6  12.58 11.61 10.01 11.6   5.27  7.51  4.26  6.93 10.12]
 [ 5.23  6.41  5.25  4.62  5.32  7.95  3.53  8.94 14.64 10.97]
 [ 7.23  4.45  5.01 14.77  8.2   5.45  8.76 16.4   8.48 10.47]]

Smallest number..........: -1.2061547875211307
Largest number...........: 16.40102054591537
Index of the smallest number: (array([0]), array([5]))
Index of the largest number..: (array([4]), array([7]))
Mean....................: 7.8979406079693995
Median..................: 7.271472480175898
Variance................: 15.041654042036352
Standard deviation......: 3.8783571318325434
Type of  x: <class 'numpy.ndarray'>
Type of mw: <class 'numpy.float64'>
```

Analysis

In line 06, the NumPy function `random.normal(8,4,size=(lines,columns))` generates 50 normally distributed random numbers as a matrix with five lines and ten columns. This function expects the center of the distribution as a first argument, a rough specification for the spread of the random numbers to be generated as a second argument and a tuple for the number of lines and columns as the third argument.

To ensure that the same random numbers are generated each time the program is restarted, line 05 contains the `random.seed()` function. If new random numbers should also be generated at each new program start, this function must be commented out or deleted.

Lines 07 to 10 calculate the desired statistical measures: the mean `mw`, median `md`, variance `v`, and standard deviation `staw`.

An interesting task is to find the array index for the smallest and the largest random number in lines 13 and 14. In line 13, the `where(x==np.amin(x))` function determines the position in the array with the smallest random number: `[0.5]` (output in line 21). The same applies to the determination of the index of the largest random number. The program outputs the index `[4.7]` for this number in line 22. A check against the random numbers output in line 18 confirms the results. A simpler way to determine the location in the array where the smallest or largest random number is located is to use the functions in lines 15 and 16, which have been commented out.

All calculated statistical measures are of type Float64 (line 28). So, you have double precision with 52-bit mantissa and 11-bit exponent.

3.2 Vectors

Vectors (Latin *vector*; English *carrier, driver*) are physical quantities that, in contrast to scalar quantities, are characterized by a direction in addition to a magnitude. Examples of directed magnitudes include velocities, forces, or field strengths. In physics and mathematics, vectors are graphically represented as arrows, as shown in Figure 3.1.

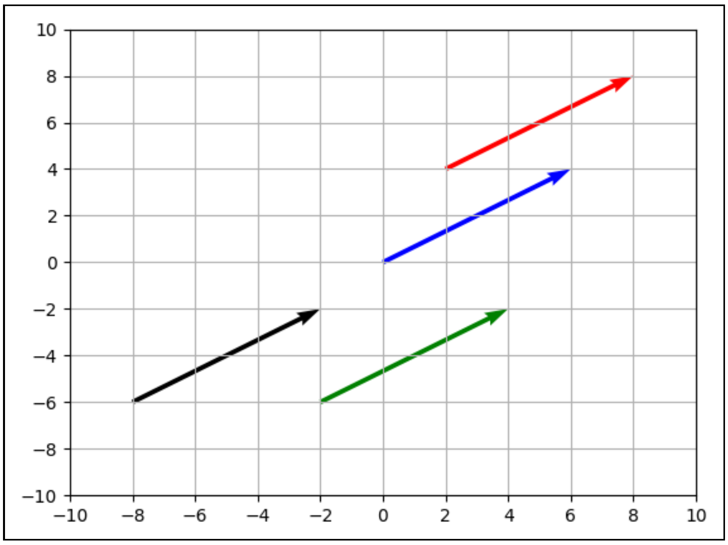


Figure 3.1 Vector Shift

As shown in Figure 3.1, vectors can be shifted arbitrarily in the plane, provided that their magnitudes and directions do not change. The same statement is true in three-dimensional space. The vectors shown have the same x and y components of $x = 6$ and $y = 4$. In mathematics, the formulation (6,4) is common. The angle is about 33.7° in each case.

3.2.1 Addition of Vectors

Vectors are added component by component, an operation shown in Figure 3.2.

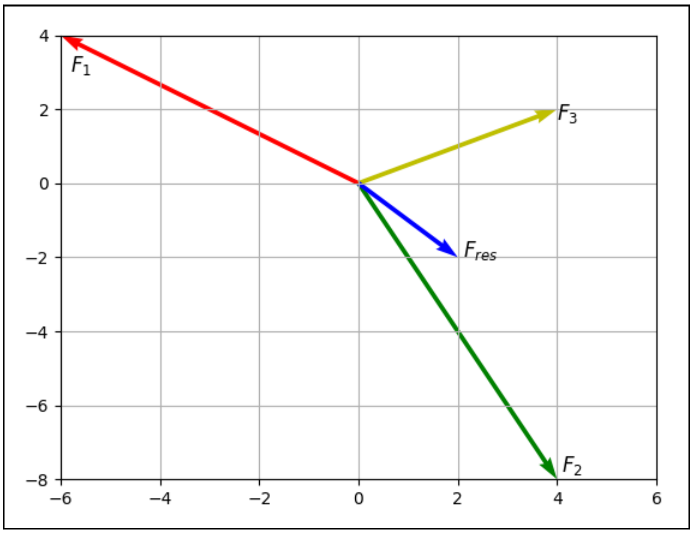


Figure 3.2 Addition of Three Vectors

If you add up vector $\boldsymbol{F_1}$ = (–6,4), vector $\boldsymbol{F_2}$ = (4,–8), and vector $\boldsymbol{F_3}$ = (4,2), you get the resulting vector $\boldsymbol{F}_{\text{res}}$ = (2,–2). In the language of mathematics:

$$\vec{F}_{res} = \begin{pmatrix} F_{x1} \\ F_{y1} \end{pmatrix} + \begin{pmatrix} F_{x2} \\ F_{y2} \end{pmatrix} + \begin{pmatrix} F_{x3} \\ F_{y3} \end{pmatrix} = \begin{pmatrix} -6 \\ 4 \end{pmatrix} + \begin{pmatrix} 4 \\ -8 \end{pmatrix} + \begin{pmatrix} 4 \\ 2 \end{pmatrix} = \begin{pmatrix} 2 \\ -2 \end{pmatrix}$$

Vectors can be created from tuples or lists using the `array` function. Listing 3.7 shows the implementation of a vector addition via tuples.

```python
#07_vectoraddition.py
import numpy as np
F1=-6,4
F2=4,-8
F3=4,2
F1=np.array(F1)
F2=np.array(F2)
F3=np.array(F3)
Fres=F1+F2+F3
F_1=np.sqrt(F1[0]**2+F1[1]**2)
F_2=np.sqrt(F2[0]**2+F2[1]**2)
F_3=np.sqrt(F3[0]**2+F3[1]**2)
F_res=np.sqrt(Fres[0]**2+Fres[1]**2)
angle=np.arctan(Fres[0]/Fres[1])
angle=np.degrees(angle)
#Output
print("Coordinates of F1:",F1)
print("Coordinates of F2:",F2)
```

```
19 print("Coordinates of F3:",F3)
20 print("Magnitude of F1  :",F_1)
21 print("Magnitude of F2  :",F_2)
22 print("Magnitude of F3  :",F_3)
23 print("Resulting force  :",Fres)
24 print("Magnitude of Fres:",F_res)
25 print("Angle of Fres    :",angle,"°")
```

Listing 3.7 Addition of Three Vectors

Output

```
Coordinates of F1: [-6 4]
Coordinates of F2: [ 4 -8]
Coordinates of F3: [4 2]
Magnitude of F1  : 7.211102550927978
Magnitude of F2  : 8.94427190999916
Magnitude of F3  : 4.47213595499958
Resulting force  : [ 2 -2]
Magnitude of Fres: 2.8284271247461903
Angle of Fres    : -45.0 °
```

Analysis

In lines 03 to 05, the x-y components of the three forces are passed as tuples to variables `F1` to `F3`. The statements in lines 06 to 08 each create a one-dimensional NumPy array from the force components.

In line 09, the vector addition takes place. The forces are added element by element. The internal processes remain hidden from the user. Internally, the program calculates `Fres[0]=F1[0]+F2[0]+F3[0]` and `Fres[1]=F1[1]+F2[1]+F3[1]`. Line 23 outputs the result. The program calculates the magnitudes of the three forces using the Pythagorean theorem (lines 10 to 12). Line 14 calculates the angle of the resulting force using NumPy function `arctan(Fres[0]/Fres[1])`. The `degrees(angle)` NumPy function in line 15 ensures that the angle is converted to degrees.

The output of the program in lines 17 to 25 can be easily checked using Figure 3.2. The units have been deliberately omitted.

3.2.2 Scalar Product

In mechanical engineering, work is defined as the product of the force F multiplied by the displacement s multiplied by the cosine of the angle α between the two magnitudes:

$$W = F \cdot s \cdot \cos\alpha$$

From this definition, the coordinate form of the scalar product can be derived using the cosine theorem:

$$W = F_x s_x + F_y s_y + F_z s_z = \begin{pmatrix} F_x \\ F_y \\ F_z \end{pmatrix} \cdot \begin{pmatrix} s_x \\ s_y \\ s_z \end{pmatrix}$$

The *scalar product* is calculated as the sum of the products of force and displacement components.

In abbreviated notation, the following applies to the definition of the scalar product:

$$W = \vec{F} \cdot \vec{s}$$

The magnitude of the force is calculated from the square root of the scalar product of the force vector with itself:

$$F = \sqrt{\vec{F} \cdot \vec{F}}$$

And the magnitude of the displacement is calculated from the square root of the scalar product of the displacement vector with itself:

$$s = \sqrt{\vec{s} \cdot \vec{s}}$$

For the angle between force vector and path vector, the following applies:

$$\alpha = \arccos\left(\frac{\vec{F} \cdot \vec{s}}{Fs}\right)$$

Now, let's consider an example referring to $\vec{F} = (2,7,-3)$ N and $\vec{s} = (-2,3,4)$ m to show how the scalar product is calculated for three-dimensional vectors. The following applies:

$$W = \begin{pmatrix} F_x \\ F_y \\ F_z \end{pmatrix} \cdot \begin{pmatrix} s_x \\ s_y \\ s_z \end{pmatrix} = \begin{pmatrix} 2 \\ 7 \\ -3 \end{pmatrix} \mathrm{N} \cdot \begin{pmatrix} -2 \\ 3 \\ 4 \end{pmatrix} \mathrm{m} = 5\ \mathrm{Nm}$$

For the specified components of the force and path vectors, a work of 5 Nm is performed. The NumPy function `dot(F,s)` calculates the scalar product. Listing 3.8 shows how the mechanical work is calculated from the scalar product of the force and path vectors.

```
#08_scalarproduct.py
import numpy as np
F=2,7,-3
s=-2,3,4
F_B=np.sqrt(np.dot(F,F))
s_B=np.sqrt(np.dot(s,s))
cos_Fs=np.dot(F,s)/(F_B*s_B)
angle=np.degrees(np.arccos(cos_Fs))
W=np.dot(F,s)
#Output
```

```
11  print("Magnitude of force:   ",F_B, "N")
12  print("Magnitude of path:    ",s_B,"m")
13  print("Angle between F and s:",angle,"°")
14  print("Work:",W,"Nm")
```

Listing 3.8 Scalar Product

Output

```
Magnitude of force:   7.874007874011811 N
Magnitude of path:    5.385164807134504 m
Angle between F and s:83.22811782220313 °
Work: 5 Nm
```

Analysis

In lines 03 and 04, three force and three path components are passed as tuples to the F and s variables, respectively. The first element of a tuple contains the x-component, the second the y-component, and the third the z-component of the force (F) and path (s) vectors.

Lines 05 and 06 calculate the magnitudes of the vectors with the scalar product of the NumPy function dot(F,F) and dot(s,s), respectively. The angle between the force vector and the path vector is also calculated using the dot function (line 07). Line 09 calculates the mechanical work W with the scalar product W=np.dot(F,s). The program internally calculates the mechanical work by the element-wise multiplication as required by the definition of the scalar product: W=F[0]s[0]+F[1]s[1]+F[2]s[2].

Line 14 outputs the mechanical work W performed on a mass point shift in space. The result of 5 Nm matches the previously determined value.

3.2.3 Cross Product

The magnitude of torque M is defined as the product of the force F multiplied by the lever arm l multiplied by the sine of angle α between the two magnitudes:

$$M = F \cdot l \cdot \sin\alpha$$

From this definition, the coordinate form of the *cross product* can be derived:

$$\vec{M} = \begin{pmatrix} F_y l_z - F_z l_y \\ F_z l_x - F_x l_z \\ F_x l_y - F_y l_x \end{pmatrix} = \begin{pmatrix} F_x \\ F_y \\ F_z \end{pmatrix} \times \begin{pmatrix} l_x \\ l_y \\ l_z \end{pmatrix}$$

The abbreviated variant applies to the definition of the cross product:

$$\vec{M} = \vec{F} \times \vec{l}$$

For $\vec{F} = (2,7,-3)$ and $\vec{l} = (-2,3,4)$, the following torque vector results:

$$\vec{M}=\begin{pmatrix}F_y l_z - F_z l_y\\ F_z l_x - F_x l_z\\ F_x l_y - F_y l_x\end{pmatrix}=\begin{pmatrix}7\cdot 4 & -(-3)\cdot 3\\ -3\cdot -2 & -2\cdot 4\\ 2\cdot 3 & -7\cdot -2\end{pmatrix}\mathrm{Nm}=\begin{pmatrix}37\\ -2\\ 20\end{pmatrix}\mathrm{Nm}$$

Listing 3.9 calculates the torque from the force and the lever vector in three-dimensional space with the NumPy function `cross(F,l)`.

```
01  #09_crossproduct.py
02  import numpy as np
03  F=2,7,-3
04  l=-2,3,4
05  F_B=np.sqrt(np.dot(F,F))
06  l_B=np.sqrt(np.dot(l,l))
07  cos_Fl=np.dot(F,l)/(F_B*l_B)
08  angle=np.degrees(np.arccos(cos_Fl))
09  M=np.cross(F,l)
10  M_B=np.sqrt(np.dot(M,M))
11  #Output
12  print("Magnitude of force    :",F_B,"N")
13  print("Magnitude of lever arm:",l_B,"m")
14  print("Angle between F and l : ",angle,"°")
15  print("Torque M              :",M,"Nm")
16  print("Magnitude of torque   :",M_B,"Nm")
```

Listing 3.9 Cross Product

Output

```
Magnitude of force    : 7.874007874011811 N
Magnitude of lever arm: 5.385164807134504 m
Angle between F and l : 83.22811782220313 °
Torque M              : [37 -2 20] Nm
Magnitude of torque   : 42.1070065428546 Nm
```

Analysis

The force vector `F` and the vector of the lever arm `l` are again defined as tuples in lines 03 and 04.

In line 09, the program calculates the cross product using the NumPy function, `M=np.cross(F,l)`. The result is again a vector `[37 -2 20] Nm` (output in line 15). The magnitude of `42.1 Nm` of the torque corresponds to the area of the parallelogram spanned by the force vector `F` and the vector of the lever arm `l`.

3.2.4 Triple Product

The *triple product* calculates the volume of a parallelepiped from the cross product and the scalar product:

$$V = \vec{c} \cdot (\vec{a} \times \vec{b})$$

Listing 3.10 calculates the volume of a cuboid using the triple product `dot(c, np.cross(a,b))`.

```
01  #10_tripleproduct.py
02  import numpy as np
03  a=2,0,0
04  b=0,3,0
05  c=0,0,4
06  a_B=np.sqrt(np.dot(a,a))
07  b_B=np.sqrt(np.dot(b,b))
08  c_B=np.sqrt(np.dot(c,c))
09  V=np.dot(c,np.cross(a,b))
10  #Output
11  print("Magnitude of a:",a_B)
12  print("Magnitude of b:",b_B)
13  print("Magnitude of c:",c_B)
14  print("Triple product:",V)
```

Listing 3.10 Triple Product

Output

```
Magnitude of a: 2.0
Magnitude of b: 3.0
Magnitude of c: 4.0
Triple product: 24
```

Analysis

The components of the three vectors `a`, `b`, and `c` were chosen to form a cuboid with the following sides: `a=2`, `b=3` and `c=4`.

Line 09 calculates the triple product of NumPy functions `dot()` and `cross()`. The `dot` function is passed the variable `c` for the height of the box and the `cross(a,b)` function for the calculation of the base area as arguments.

The output in line 14 returns the correct result of 24 space units.

3.2.5 Dyadic Product

In the *dyadic product*, also called the *outer product*, the row vectors are multiplied by the column vector:

$$\begin{pmatrix}1 & 2 & 3\end{pmatrix} \otimes \begin{pmatrix}4\\5\\6\end{pmatrix} = \begin{pmatrix}4 & 5 & 6\\8 & 10 & 12\\12 & 15 & 18\end{pmatrix}$$

Listing 3.11 calculates the dyadic product for the given matrices.

```
01  #11_outer.py
02  import numpy as np
03  A=np.array([[1,2,3]])
04  B=np.array([[4],[5],[6]])
05  C=np.outer(A,B)
06  print("Matrix A")
07  print(A)
08  print("Matrix B")
09  print(B)
10  print("Dyadic product")
11  print(C)
```

Listing 3.11 Dyadic Product

Output

```
Matrix A
[[1 2 3]]
Matrix B
[[4]
 [5]
 [6]]
Dyadic product
[[ 4  5  6]
 [ 8 10 12]
 [12 15 18]]
```

Analysis

In line 03, a row vector `A` is defined and in line 04, a column vector `B` is defined. The dyadic product is calculated by NumPy function `outer(A,B)` in line 05. The result matches the manually calculated value.

3.3 Matrix Multiplication

Matrix multiplication is needed, for example, in the calculation of electrical networks. If several two-port networks are connected in series (catenary circuit), then the catenary shape (A parameter) can be used to calculate the required input voltage and current for a given output voltage and current by matrix multiplication.

Two matrices are multiplied with each other by multiplying the rows of the first matrix by the columns of the second matrix, element by element, and adding the individual products (according to *Falk's scheme*):

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}.b_{11} + a_{12} \cdot b_{21} & a_{11}.b_{12} + a_{12} \cdot b_{22} \\ a_{21}.b_{11} + a_{22} \cdot b_{21} & a_{21}.b_{12} + a_{22} \cdot b_{22} \end{pmatrix}$$

A simple example will demonstrate the matrix multiplication using a schema (see Table 3.1). The following two matrices are to be multiplied:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \cdot \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

The first matrix is entered in the first and second columns and the third and fourth rows of a table. The second matrix is entered in the third and fourth columns and in the first and second rows of the table.

		5	6
		7	8
1	2	$1 \cdot 5 + 2 \cdot 7 =$ **19**	$1 \cdot 6 + 2 \cdot 8 =$ **22**
3	4	$3 \cdot 5 + 4 \cdot 7 =$ **43**	$3 \cdot 6 + 4 \cdot 8 =$ **50**

Table 3.1 Schema for Matrix Multiplication

The first row of the first matrix is multiplied element by element by the first column of the second matrix. The two products are added up. The second row of the first matrix is multiplied by the first column of the second matrix. The two products are added up again. The second column is calculated according to the same schema.

NumPy provides the `array([[a11,a12],[a21,a22]])` function for generating the matrices. You can adjust the number of rows and columns as needed.

The easiest way to perform matrix multiplication is to use the infix operator `@`. Alternatives are `matmul(A,B)` or `multi_dot([A,B,C,...])`.

Listing 3.12 shows how you can perform matrix multiplication using the numbers from our earlier example.

```
#12_mulmatrix1.py
import numpy as np
```

```
03  A=np.array ([[1, 2],
                [3, 4]])
05  B=np.array ([[5, 6],
                [7, 8]])
07  C=A@B
08  D=B@A
09  #Output
10  print(type(A))
11  print("Matrix A\n",A)
12  print("Matrix B\n",B)
13  print("Product A*B\n",C)
14  print("Product B*A\n",D)
```

Listing 3.12 Matrix Multiplication

Output

```
<class 'numpy.ndarray'>
Matrix A
 [[1 2]
 [3 4]]
Matrix B
 [[5 6]
 [7 8]]
Product A*B
 [[19 22]
 [43 50]]
Product B*A
 [[23 34]
 [31 46]]
```

Analysis

Lines 03 to 06 define matrices with two rows and two columns each. The values of the individual coefficients are stored in variables A and B.

Line 07 performs the matrix multiplication C=A@B, while line 08 performs the multiplication with an interchanged order of factors D=B@A.

The product for C is correctly output line 13 and matches the value that was manually calculated in Table 3.1. The result from line 14, on the other hand, deviates from this. This result is also correct, as you can easily check by recalculation. (You thus learn from this that the commutative law does not apply to a matrix product.)

Usage Example: Analysis of a π-Substitute Circuit

Let's say a Python program needs to calculate the matrix of catenary parameters and the required input variables U_1 and I_1 for given output variables U_2 and I_2 for a π-substitute circuit, as shown in Figure 3.3.

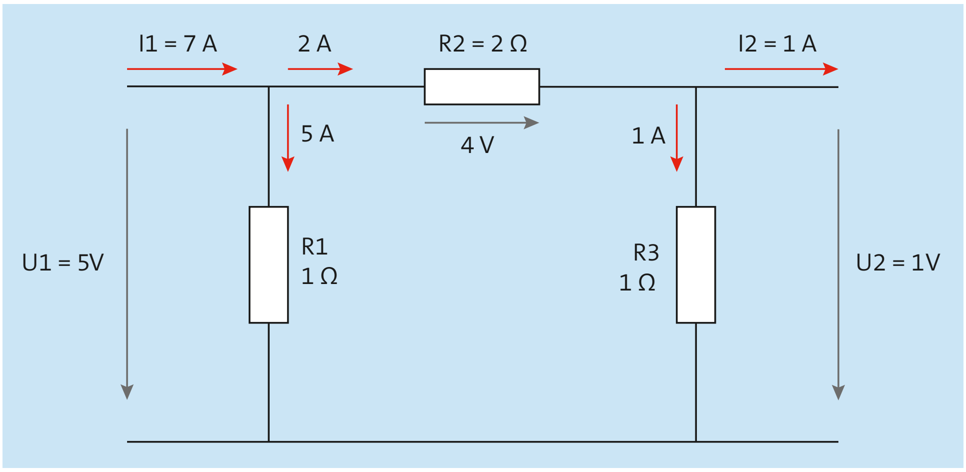


Figure 3.3 π-Substitute Circuit

Any passive two-port network can be described in general terms by a linear system of equations with a matrix of four parameters and the column vectors from voltages or currents.

Catenary Shape with A Parameters

For this problem, the catenary shape must be chosen. On the left-hand side of the equation system, a column vector contains the input variables we are looking for. On the right-hand side, we have a catenary matrix with the four coefficients $\boldsymbol{A}_{11}$ to $\boldsymbol{A}_{22}$. The catenary matrix is multiplied by the column vector of the output variables. If the coefficients of the catenary matrix and the column vector of the output variables are known, the input variables U_1 and I_1 can be calculated:

$$\begin{pmatrix} U_1 \\ I_1 \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} U_2 \\ I_2 \end{pmatrix}$$

For the transverse resistors R_1 and R_3, the following A parameters can be determined from the circuit shown in Figure 3.3:

$$A_{1q} = \begin{pmatrix} 1 & 0 \\ \frac{1}{R_1} & 1 \end{pmatrix}, A_{2q} = \begin{pmatrix} 1 & 0 \\ \frac{1}{R_3} & 1 \end{pmatrix}$$

For the series resistance R_2, the following matrix results:

$$A_l = \begin{pmatrix} 1 & R_2 \\ 0 & 1 \end{pmatrix}$$

To obtain the system matrix of the entire circuit shown in Figure 3.3, you need to multiply all three partial matrices with each other.

Listing 3.13 performs the matrix multiplication from the three partial matrices for the π-substitution circuit. You can of course change the values of the resistors for further testing.

```python
#13_mulmatrix2.py
import numpy as np
R1=1
R2=2
R3=1
U2=1
I2=1
A1q=np.array([[1, 0],
              [1/R1, 1]])
Al=np.array([[1, R2],
             [0, 1]])
A2q=np.array([[1, 0],
              [1/R3, 1]])
A=A1q@Al@A2q
b=np.array([[U2],[I2]])
E=A@b
U1,I1=E[0,0],E[1,0]
print("Chain shape A\n",A)
print("Input variables\n",E)
print("Input voltage U1=%3.2f V" %U1)
print("Input current I1=%3.2f A" %I1)
```

Listing 3.13 Matrix Multiplication with A Catenary Parameters

Output

```
Chain shape A
 [[3. 2.]
 [4. 3.]]
Input variables
 [[5.]
 [7.]]
Input voltage U1=5.00 V
Input current I1=7.00 A
```

Analysis

The values for the output voltage U_2; the output current I_2; and the three resistors R_1, R_2, and R_3 were taken from the specifications of the circuit shown in Figure 3.3.

In lines 08 to 13, the three partial matrices `A1q`, `Al`, and `A2q` are defined for the transverse resistances R_1 and R_3 and the series resistance R_2. Line 14 performs the matrix multiplication `A=A1q@Al@A2q`. Pay attention to the correct sequence of factors. As shown earlier in Listing 3.12, the commutative law does not apply to matrix multiplication! Changing the order of the partial matrices would also represent a different circuit structure.

Line 15 creates the column vector `b=np.array([[U2],[I2]])` for the output variables. In line 16, system matrix `A` is multiplied by column vector `b`. The result of the matrix multiplication is assigned to column vector `E`.

The input voltage must be 5 V so that a voltage of $U_2 = 1$ V is present at the output of the π-substitute circuit. A current of $I_1 = 7$ A must flow at the input of the circuit so that a current of $I_2 = 1$ A flows at the output. You can check the results using the circuit shown in Figure 3.3.

3.3.1 Chain Shape with B Parameters

The output variables U_2 and I_2 of a two-port network are calculated using the B catenary parameters. The following two-port equations are then obtained for a π-substitute circuit:

$$\begin{pmatrix} U_2 \\ I_2 \end{pmatrix} = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \cdot \begin{pmatrix} U_1 \\ I_1 \end{pmatrix}$$

For the transverse resistors R_1 and R_3, the B parameters can be determined from the circuit shown in as follows:

$$B_{1q} = \begin{pmatrix} 1 & 0 \\ -\dfrac{1}{R_1} & 1 \end{pmatrix}, \qquad B_{2q} = \begin{pmatrix} 1 & 0 \\ -\dfrac{1}{R_3} & 1 \end{pmatrix}$$

For the series resistance R_2, the following matrix is obtained:

$$B_l = \begin{pmatrix} 1 & -R_2 \\ 0 & 1 \end{pmatrix}$$

In general, the B parameters can be determined from the inverse matrix of A. The following applies:

$$B = A^{-1}$$

Listing 3.14 calculates the output voltage U_2 and output current I_2 of a π-substitute circuit with the B catenary parameters.

```
#14_mulmatrix3.py
import numpy as np
R1=1
R2=2
R3=1
U1=5
I1=7
```

```
B1q=np.array([[1, 0],
              [-1/R1, 1]])
B2l=np.array([[1, -R2],
              [0, 1]])
B3q=np.array([[1, 0],
              [-1/R3, 1]])
B=B1q@B2l@B3q
b=np.array([[U1],[I1]])
E=B@b
U2,I2=E[0,0],E[1,0]
print("Chain shape B\n",B)
print("Output variables\n",E)
print("Output voltage U2=%3.2fV" %U2)
print("Output current I2=%3.2fA" %I2)
```

Listing 3.14 Matrix Multiplication with B Catenary Parameters

Output

```
Chain shape B
 [[ 3. -2.]
 [-4.  3.]]
Output variables
 [[1.]
 [1.]]
Output voltage U2=1.00V
Output current I2=1.00A
```

Analysis

Basically, the program is structured in the same way as shown in Listing 3.13, except that the parameters in the secondary diagonal have a negative sign.

The result for the output voltage U_2 and the output current I_2 matches the values determined using Kirchhoff's circuit laws in the circuit shown in .

3.3.2 Usage Example: Calculating the Energy of a Rotating Rigid Body in Space

The next example shows the multiplication of the row vector of an angular velocity with an inertia tensor I (3×3 matrix) and the column vector of an angular velocity.

For the rotational energy, the following applies:

$$E_{\text{rot}} = \frac{1}{2}\vec{\omega}^T \cdot I \cdot \vec{\omega}$$

The superscript *T* means that the vector of angular velocity must be *transposed*, that is, the column vector is converted into a row vector. In component notation, you obtain the following:

$$E_{\text{rot}} = \frac{1}{2}(\omega_x \quad \omega_y \quad \omega_z) \cdot \begin{pmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{yx} & I_{yy} & -I_{yz} \\ -I_{zx} & -I_{zy} & I_{zz} \end{pmatrix} \cdot \begin{pmatrix} \omega_x \\ \omega_y \\ \omega_z \end{pmatrix}$$

For the inertia tensor of a point mass m, the following applies:

$$I = m \cdot \begin{pmatrix} y^2 + z^2 & -xy & -xz \\ -yx & x^2 + z^2 & -yz \\ -zx & -zy & x^2 + y^2 \end{pmatrix}$$

The product of the mass m and the matrix with the location coordinates is referred to as the inertia tensor. If you perform the matrix multiplication, you'll get the rotational energy, which is stored in the rotating body.

For a case where mass m with radius $x = r$ rotates around the z-axis in the x-y-plane, the following applies in a simplified way:

$$I = m \cdot \begin{pmatrix} 0 & 0 & 0 \\ 0 & r^2 & 0 \\ 0 & 0 & r^2 \end{pmatrix}$$

Listing 3.15 calculates the rotational energy of a point mass of mass $m = 6$ kg rotating in space around the z-axis with an angular velocity of $\vec{\omega} = (0,0,1)s^{-1}$.

```
#15_mulmatrix4.py
import numpy as np
x=1 #distance in m
y=0
z=0
wx=0
wy=0
wz=1 #angular velocity
m=6  #mass in kg
w_Z=np.array([wx,wy,wz])
I=m*np.array([[y**2+z**2, -x*y, -x*z],
              [-x*y, x**2+z**2, -y*z],
              [-x*z, -y*z, x**2+y**2]])
w_S=np.array([[wx],
              [wy],
              [wz]])
#Calculation of the rotational energy
Erot=0.5*w_Z@I@w_S
#Erot=0.5*w_S.T@I@w_S
Er=Erot[0]
```

```
21  #Output
22  print("Rotational energy: %3.2f joules" %Er)
```

Listing 3.15 Matrix Multiplication with Three Vectors

Output

```
Rotational energy: 3.00 joules
```

Analysis

The rotational energy is calculated according to the rule: “row vector multiplied by 3×3 matrix multiplied by column vector.” Following this sequence is mandatory because the commutative law does not apply with matrices! Line 10 contains the row vector of angular velocity, lines 11 to 13 contain the 3×3 matrix of the inertia tensor, and lines 14 to 16 contain the column vector of the angular velocity.

The statement in line 18 performs the matrix multiplication and stores the result in the `Erot` variable. Alternatively, you can comment out lines 10 and 18 and remove the comment in line 19. In this line, the column vector from line 14 is transposed into a row vector using the `T` property.

3.4 Linear Systems of Equations

Electrical engineering uses linear systems of equations to calculate mesh currents and node voltages in networks. Structural analysis also uses linear systems of equations for the calculation of member forces in trusses. Thus, solving systems of linear equations with n unknowns is an important and indispensable tool in engineering. The NumPy function `solve()` enables you to solve systems of equations with real and complex coefficients as easily and without much effort as, for example, using the MATLAB program.

3.4.1 Systems of Equations with Real Coefficients

A linear system of equations can generally be represented as a matrix product of coefficient matrix (system matrix) a_{11} to a_{mn} and solution vector x as an equation. On the right-hand side of the equation system, we have the inhomogeneity vector b.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

The following abbreviated notation is commonly used:

$$A \cdot x = b$$

To determine solution vector x, the inverse matrix A^{-1} must be formed and multiplied by the inhomogeneity vector b:

$$x = A^{-1} \cdot b$$

Based a simple example, let's walk you through the solution of a simple system of equations with three unknowns:

$$\begin{pmatrix} 1 & 1 & 1 \\ 2 & -2 & 3 \\ 3 & -4 & 2 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 6 \\ 7 \\ 1 \end{pmatrix}$$

Listing 3.16 solves a linear system of equations for three unknowns using NumPy function `solve(A,b)`.

```python
#16_equation_system.py
import numpy as np
from numpy.linalg import solve
#coefficient matrix
A = np.array([[1,  1, 1],
              [2, -2, 3],
              [3, -4, 2]])
#inhomogeneity vector
b = np.array([6, 7, 1])
#solution
solution=solve(A,b)
#Output
print("Solution of a linear system of equations")
print("Coefficient matrix\n",A)
print("Inhomogeneity vector\n",b)
print("Solution:\n",solution)
```

Listing 3.16 Solution of a Linear System of Equations

Output

```
Solution of a linear system of equations
Coefficient matrix
 [[ 1  1  1]
 [ 2 -2  3]
 [ 3 -4  2]]
Inhomogeneity vector
 [6 7 1]
Solution:
 [1. 2. 3.]
```

Analysis

Line 03 imports the `linalg` submodule with the `solve` function.

In lines 05 to 07, the coefficient matrix `A` of the equation system is generated as a two-dimensional NumPy array.

In line 09, the inhomogeneity vector `array([6,7,1])` is assigned to variable `b`.

In line 11, NumPy function `solve(A,b)` calculates the solution of the linear system of equations. The solution vector is stored in variable `solution`.

The solution vector contains floats although the coefficient matrix and the inhomogeneity vector consist of integers. If you use

```
print(type(A[0,0]))
print(type(b[0]))
print(type(solution[0]))
```

to output the data types, you'll get the following output:

```
<class 'numpy.int64'>
<class 'numpy.int64'>
<class 'numpy.float64'>
```

When a mathematical operation on arrays produces floats, then all the integers in the array are converted to floats. If you declare a single arbitrary integer of an array as a float (e.g., `2.` instead of `2`), then all other elements of the array are automatically converted to floats.

3.4.2 Systems of Equations with Complex Coefficients

In an alternating current (AC) network, a complex resistor consists of either an inductive or capacitive impedance:

$$Z_L = R + \mathrm{j}\omega L$$

$$Z_C = R - \mathrm{j}\frac{1}{\omega C}$$

For a network with four meshes, the general rule is:

$$\begin{pmatrix} Z_{11} & Z_{12} & Z_{13} & Z_{14} \\ Z_{21} & Z_{22} & Z_{23} & Z_{24} \\ Z_{31} & Z_{32} & Z_{33} & Z_{34} \\ Z_{41} & Z_{42} & Z_{43} & Z_{44} \end{pmatrix} \cdot \begin{pmatrix} I_1 \\ I_2 \\ I_3 \\ I_4 \end{pmatrix} = \begin{pmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{pmatrix}$$

Using the network shown in Figure 3.4 as an example, try reading a system of equations directly from the circuit using mesh analysis.

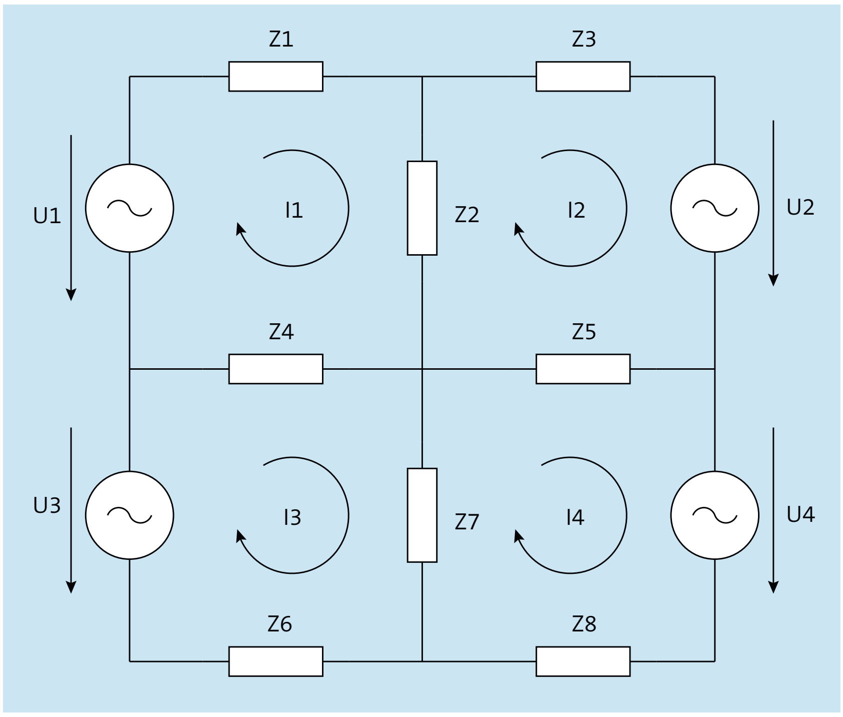


Figure 3.4 AC Network with Four Meshes

The coefficient matrix is entered in Table 3.2. This table consists of four rows and five columns. The fifth column is for the vector of source voltages.

	I_1	I_2	I_3	I_4	U
1	$Z_1 + Z_2 + Z_4$	$-Z_2$	$-Z_4$	0	U_1
2	$-Z_2$	$Z_2 + Z_3 + Z_5$	0	$-Z_5$	$-U_2$
3	$-Z_4$	0	$Z_4 + Z_6 + Z_7$	$-Z_7$	U_3
4	0	$-Z_5$	$-Z_7$	$Z_5 + Z_7 + Z_8$	$-U_4$

Table 3.2 System of Equations according to the Mesh Analysis Method

The sums of the impedances from the individual meshes are shown along the main diagonal. The secondary diagonals track the common impedances of two meshes. If two meshes have no common impedances, a 0 is entered in the table. All coefficients of the secondary diagonals have a negative sign and are reflected on the main diagonal of the impedance matrix. As shown in Listing 3.17, the coefficient matrix of rows 1 to 4 and columns 1 to 4 from Table 3.2 is transferred directly into a NumPy array.

```
01  #17_mesh4c.py
02  import numpy as np
03  import numpy.linalg
04  U1=230
05  U2=-230
06  U3=230
07  U4=-230
08  Z1=1+2j
09  Z2=2-4j
10  Z3=3+4j
11  Z4=2+5j
12  Z5=1+5j
13  Z6=2+5j
14  Z7=4-5j
15  Z8=1+5j
16  Z=np.array([[Z1+Z2+Z4,-Z2,-Z4, 0],
17              [-Z2,Z2+Z3+Z5, 0,-Z5],
18              [-Z4, 0,Z4+Z6+Z7,-Z7],
19              [0,-Z5,-Z7,Z5+Z7+Z8]])
20  U=np.array([U1,-U2,U3,-U4])
21  current=np.linalg.solve(Z,U) #numpy.ndarray
22  for k, I in enumerate(current,start=1):
23      print("I%d = (%0.2f, %0.2fj)A" %(k,I.real,I.imag))
```

Listing 3.17 Network with Complex Resistors

Output

```
I1 = (33.16, -52.04j)A
I2 = (19.63, -49.35j)A
I3 = (20.09, -41.98j)A
I4 = (18.09, -51.66j)A
```

Analysis

Lines 04 to 15 contain the values for the voltages and impedances of the network. The coefficient matrix `Z` is defined in lines 16 to 19. The rows and columns of the matrix are arranged in a NumPy `array([[],...,[]])` according to Table 3.2. For the calculation of the solution vector `I`, the inhomogeneity vector `U` must still be defined in line 20. The solution is calculated using NumPy function `linalg.solve(Z,U)` in line 21. The solution vector `current` contains the four mesh currents `I[0]`, `I[1]`, `I[2]`, and `I[3]`.

The Python function `enumerate(current)` allows for the output of the individual mesh currents within a `for` loop (line 23). Each individual mesh current `I` is marked with the index `k`. With each iteration, the `enumerate(current)` function returns a tuple containing the index `k` and the corresponding element `I` of the `current` array.

3.5 Project Task: Lightning Protection System

Let's say, for a cuboid-shaped building, the currents in the lightning conductor and down conductor need to be calculated. The building has a length of 10 m, a width of 5 m, and a height of 3 m. The lightning and down conductors made of steel with a conductor cross-section of A = 50 mm^2 are installed on the edges of the building. For the top view, this results in a network with four nodes and eight resistors, as shown in Figure 3.5. The time course of a lightning current can be approximately described by a triangle with a rise time of about 10 µs and a fall time of about 1 ms. The maximums of the currents are approximately between 10,000 A and 300,000 A.

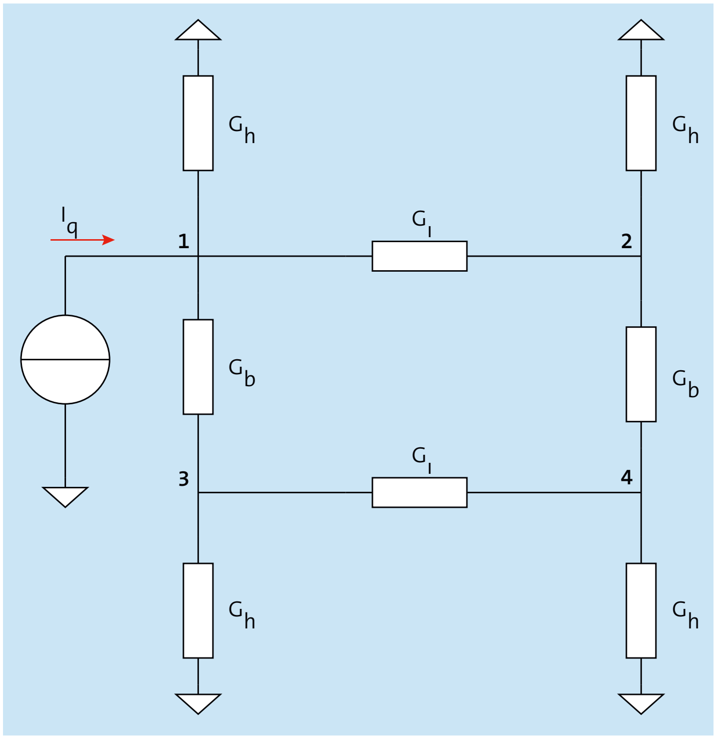


Figure 3.5 Substitute Circuit for a Lightning Protection System

The voltage drops between the nodes are calculated using the node potential method. You can read the system of equations directly from the circuit and represent it as a matrix:

$$\begin{pmatrix} G_b + G_h + G_l & -G_l & -G_b & 0 \\ -G_l & G_b + G_h + G_l & 0 & -G_b \\ -G_b & 0 & G_b + G_h + G_l & -G_l \\ 0 & -G_b & -G_l & G_b + G_h + G_l \end{pmatrix} \cdot \begin{pmatrix} U_{10} \\ U_{20} \\ U_{30} \\ U_{40} \end{pmatrix} = \begin{pmatrix} I_q \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

The following applies to the conductance of the lightning and down conductors:

$$G = \frac{\gamma A}{l}$$

We can use Ohm's law to calculate the currents in the lightning and down conductors from the potential differences of the node voltages and the conductances of the lightning and down conductors.

Listing 3.18 solves the system of equations for the four unknown nodal voltages using NumPy function `U=linalg.solve(G,I)`.

```
#18_project_lightning_protection.py
import numpy as np
Iq=1e5 #current of the lightning in A
g=10   #conductance for steel S*m/mm^2
A=50   #conductor cross section in mm^2
l=10   #length in m
b=5    #width in m
h=3    #height in m
Gh=g*A/h   #conductance for height in S
Gl=g*A/l   #conductance for length in S
Gb=g*A/b   #conductance for width in S
G=np.array([[Gb+Gh+Gl, -Gl, -Gb, 0],
            [-Gl, Gb+Gh+Gl, 0,-Gb],
            [-Gb, 0, Gb+Gh+Gl,-Gl],
            [ 0,-Gb,-Gl, Gb+Gh+Gl]])
I=np.array([Iq,0,0,0])
U=np.linalg.solve(G,I)
I10=U[0]*Gh
I20=U[1]*Gh
I30=U[2]*Gh
I40=U[3]*Gh
I12=(U[0]-U[1])*Gl
I13=(U[0]-U[2])*Gb
I34=(U[2]-U[3])*Gl
I24=(U[1]-U[3])*Gb
print("--Voltage drops of down conductors--")
print("Voltage U10: %3.2f V" %U[0])
print("Voltage U20: %3.2f V" %U[1])
print("Voltage U30: %3.2f V" %U[2])
print("Voltage U40: %3.2f V" %U[3])
print("--Currents in down conductors--")
print("Current I10: %3.2f A" %I10)
print("Current I20: %3.2f A" %I20)
print("Current I30: %3.2f A" %I30)
```

```
35  print("Current I40: %3.2f A" %I40)
36  print("--Currents in lightning conductors--")
37  print("Current I12: %3.2f A" %I12)
38  print("Current I13: %3.2f A" %I13)
39  print("Current I34: %3.2f A" %I34)
40  print("Current I24: %3.2f A" %I24)
```

Listing 3.18 Distribution of Currents in the Lightning and Down Conductors

Output

```
--Voltage drops of down conductors--
Voltage U10: 365.50 V
Voltage U20: 70.86 V
Voltage U30: 122.00 V
Voltage U40: 41.64 V
--Currents in down conductors--
Current I10: 60917.21 A
Current I20: 11810.06 A
Current I30: 20332.79 A
Current I40: 6939.94 A
--Currents in lightning conductors--
Current I12: 14732.14 A
Current I13: 24350.65 A
Current I34: 4017.86 A
Current I24: 2922.08 A
```

Analysis

Line 03 specifies the peak lightning current value of 100,000 A. The cross-section of the lightning and down conductors is usually 50 mm^2 (line 05). Lines 06 to 08 define the length, width, and height of the building in meters.

In lines 09 to 11, the conductances of the lightning and down conductors are calculated. The coefficient matrix of the conductances is written in lines 12 to 15. In line 16, the inhomogeneity vector specifies that lightning strikes node 1. The solution vector for the voltage drops is calculated in line 17 using NumPy function `linalg.solve(G,I)` and assigned to variable `U`. The calculation of the currents in the down conductors is performed in lines 18 to 21. Lines 22 to 25 calculate the currents in the lightning conductors from the potential differences.

The outputs in lines 26 to 40 show that very high currents can flow with a maximum current density of about 1,218 A/mm^2. These high current densities are still acceptable because the current only flows for a few milliseconds.

3.6 Tasks

1. Calculate the volume of a *parallelepiped* using a determinant and the `dot(cross(a, b),c)` function.
2. A catenary circuit is composed of three voltage dividers (longitudinal link R_1, cross link R_2). All resistors have the same value of 1Ω. The output voltage is $U_2 = 1\,\text{V}$. Using the catenary parameter method, calculate the input voltage U_1 and the input current I_1.
3. Calculate the dyadic product for:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix} \otimes \begin{pmatrix} 5 \\ 6 \\ 7 \\ 8 \end{pmatrix}$$

4. The expanded coefficient matrix of a linear system of equations is given as the following:

$$\left(\begin{array}{ccc|c} 3 & 2 & 1 & 10 \\ 1 & 2 & 3 & 14 \\ 2 & 1 & 4 & 16 \end{array}\right)$$

   Solve this system of equations using NumPy function `solve()`. The coefficient matrix and the inhomogeneity vector should be determined by means of slicing from the extended coefficient matrix.
5. A linear system of equations with a large number of unknowns (50 to 1000) is to be solved using NumPy function `solve()`. Generate the coefficient matrix and the inhomogeneity vector using NumPy function `random.normal()`. Test the limits of `solve()` by gradually increasing the number of unknowns.
6. Calculate all the mesh currents for the catenary circuit from Task 2 using the mesh analysis method. The input voltage is 13 V.
7. Calculate all node voltages for the catenary circuit from Task 2 using the node potential method. The input current has a value of 8 A.

Chapter 4
Function Plots and Animations Using Matplotlib

In this chapter, you'll learn how to use the Matplotlib module to display and animate mathematical functions, vectors, and geometric figures in different variations.

Matplotlib is a program library for plotting mathematical functions and geometric figures. With just a few statements, you can easily create meaningful diagrams for scientific papers and publications.

The `matplotlib` module is usually imported together with the `pyplot` submodule. The `pyplot` submodule serves as an interface, specifically an application programming interface (API), to the `matplotlib` module. Matplotlib contains a collection of functions similar to the functionality of MATLAB. Matplotlib methods create drawing areas for diagrams (plots), draw lines or points in a predefined drawing area, specify line styles, and provide numerous options for labels and the scaling of coordinate axes. Several mathematical functions can be represented in one plot or in different *subplots*. Extensive design options for labels (legends) of the individual function plots support the reader in finding their way around.

The module can include using the `import matplotlib.pyplot as plt` statement. The `plt` alias is commonly accepted as a convention.

4.1 2D Function Plots

Matplotlib provides the option to display 2D function plots in either Cartesian or polar coordinates. I will show you how to implement this type of function plot and how you can vary their representation style using real-life examples from mathematics, electrical engineering, and physics.

4.1.1 Basic Structure of a Function Plot

For the representation of function plots, the Matplotlib method `plot(x_coordinate,y_coordinate)` holds a central place. This method, however, does not immediately display mathematical functions on the screen as they are first stored in a two-dimensional array. This process runs in the background and is not visible to the user.

Creating a Function Plot with a for Loop

Listing 4.1 shows the implementation of a function plot with a `for` loop and two lists. The individual function values of the x and y coordinates of a parabola are stored in two lists and then displayed.

```
01  #01_plot_loop.py
02  import matplotlib.pyplot as plt
03  lx,ly = [],[]
04  for x in range(11):
05      y=x**2
06      lx.append(x)
07      ly.append(y)
08  plt.plot(lx,ly)
09  plt.show()
```

Listing 4.1 Function Plot with Loop

Output

Figure 4.1 shows the output of the parabola whose function plot was programmed with a loop.

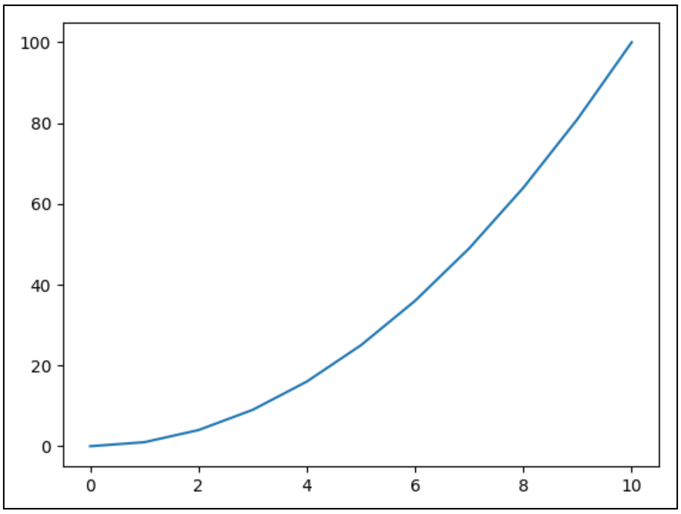


Figure 4.1 Function Plot of a Parabola

Indications and Procedures

In line 02, the `matplotlib` module is imported with the `pyplot` submodule. The `plt` alias allows you to access all `matplotlib` methods used in the program.

Line 03 creates two empty lists. Inside the `for` loop (lines 04 to 07), the parabola is defined in line 05. The x and y values of the parabola are stored in lines 06 and 07 in lists `lx` and `ly`. In line 08, the values of the x-y coordinates are prepared for the `plot` using the `plot(lx,ly)` method. In line 09, the `show()` method displays the parabola on the screen.

The approach shown in Listing 4.1 is cumbersome because you must implement a loop. In some programming languages, such as Java, C, C++, and Delphi, a loop construct is necessary for the creation of function plots. In Python, you can do without a loop construct if you import the `numpy` module in addition to the `matplotlib.pyplot` module.

The Object-Oriented Variant of a Function Plot

To better understand the object-oriented variant of Matplotlib, the terms *figure* and *axes* must be clarified up front. Figure 4.2 shows these terms in context.

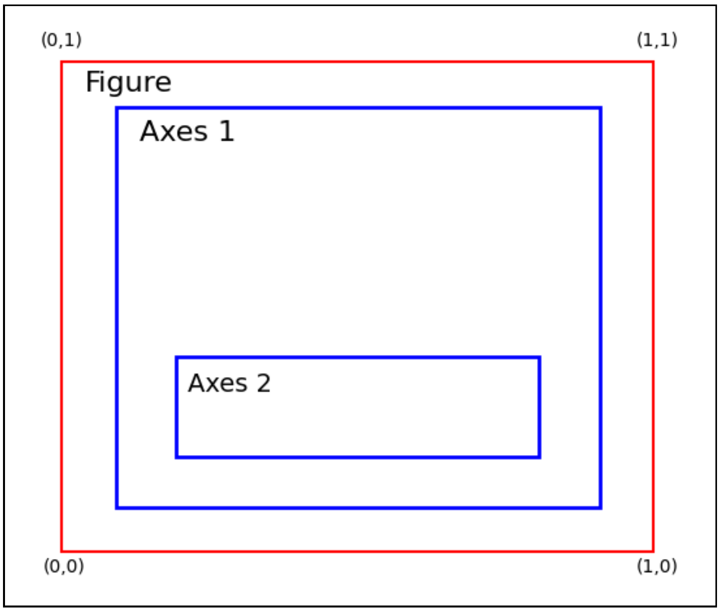


Figure 4.2 The Terms "Figure" and "Axes" Illustrated

Figure

In the Matplotlib documentation, a *figure* is a container for the top level of all plot elements. A figure object defines the entire drawing area.

One or more *axes* objects can be embedded in a *figure*.

A figure object (`fig`) can be created using the `fig = plt.figure()` statement.

The coordinate data of a figure object cannot be changed; as shown in Figure 4.2, the coordinates are fixed. You can use the `print(fig.get_figwidth())` and `print(fig.get_figheight())` statements to output the width and height of the figure object. The default values are 6.4 inches for the width and 4.8 inches for the height. If you multiply these values by the default resolution of 100 dpi, you get a drawing area of 640×480 pixels.

The `fig.set_figwidth(12)` and `fig.set_figheight(10)` statements enable you to increase the width and height of the drawing area. These values enlarge the drawing area to 1200×1000 pixels.

Axes

The most important container element is an *axis*. The documentation defines axes as a coordinate system in which one or more function plots can be displayed. Several axes objects (subplots) can be embedded in one figure object.

The following statement allows you to change the coordinates of the axes objects:

```
fig.subplots_adjust(left=0.15,bottom=0.15,right=0.7,top=0.8)
```

For the arguments, you can only use values between 0 and 1. Usually, you don't need to care about the coordinate data of an axis because the set default values are pretty convenient.

The frame lines (axes) of an axes object are referred to as *spines* in the Matplotlib documentation. You can use the methods of the axes class to label coordinate axes, insert legends and any text into a plot, and give a plot a meaningful title. LaTeX notation even enables you to represent complicated mathematical expressions, such as formulas, Greek letters, and operators.

A coordinate system can be created using the `ax=fig.add_subplot()` or `ax=fig.subplots()` statements.

You can also create a figure and an axes object using a single statement: `fig,ax = fig.subplots()`. The `subplots()` method returns the `fig,ax` tuple. The `fig` and `ax` identifiers are from the Matplotlib documentation and have now become accepted as a convention.

The following console program enables you to display the default coordinate data of the axes object:

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots()
>>> print(ax.get_position())
Bbox(x0=0.125, y0=0.10999999999999999, x1=0.9, y1=0.88)
```

The following statement allows you to hide the upper and right spines:

```
ax.spines[['top', 'right']].set_visible(False)
```

Table 4.1 contains an overview of the most important figure methods.

Method	Description
`add_axes([l,b,w,h])`	Creates a coordinate system: left border, bottom left border, width, height. The values must be between 0 and 1.
`add_subplot(r,c,n)`	Creates subplots with *r* rows and *c* columns. The number *n* indicates the number of subplots. If you enter no parameters at all, only one plot will be created.
`savefig("name.png")`	Saves a graphic in PNG format. The file extension defines the file format. The following formats are also possible: EPS (`eps`), JPEG (`jpeg`), JPG (`jpg`), PDF (`pdf`), PGF (`pgf`), PNG (`png`), PS (`ps`), RAW (`raw`), RGBA (`rgba`), SVG (`svg`), SVGGZ (`svgz`), TIF (`tif`), TIFF (`tiff`), and WEBP (`webp`).
`show()`	Causes the plot to be displayed on the screen.
`subplots_adjust(par)`	Sets the position of the drawing area. The following parameters (`par`) are possible: `left`, `right`, `bottom`, and `top`.
`subplots(r,c)`	Creates subplots with *r* rows and *c* columns.
`tight_layout()`	Creates spacing between subplots.

Table 4.1 Figure Methods (Selection)

Table 4.2 contains a selection of important axes methods.

Method	Description
`annotate(parameter)`	Labels a point of a function plot to be highlighted.
`axis([x1,x2,y1,y2])`	Sets the range of values for a function plot within an axes object.
`grid()`	Draws grid lines on a plot.
`legend()`	Places a legend on the drawing area.
`plot(x,f(x), ...)`	Internally generates a value table for a function plot. The `show` method displays the function plot on the screen.
`set_title('Text')`	Inserts a heading in a plot.
`set_xlabel('Text')`	Labels the x-axis.
`set_ylabel("Text")`	Labels the y-axis.
`set_xlim(x1,x2)`	Sets the display range for the x-axis.

Table 4.2 Important Axes Methods (Selection)

Method	Description
`set_ylim(y1,y2)`	Sets the display range for the y-axis.
`set_xticks([0,1,2, ...])`	Sets the scaling of the x-axis. If the list does not contain any entries, the display of the scaling will be suppressed.
`set_yticks([0,1,2, ...])`	Sets the scaling of the y-axis.
`set_text('y=%3.2f'%x)`	Outputs the value of the `x` variable as formatted text.
`text(x,y,'Text')`	Places a text at the x and y position on the plot.

Table 4.2 Important Axes Methods (Selection) (Cont.)

Representing a Mathematical Function

The representation of a mathematical function is performed in seven steps:

1. Importing the `numpy` module
2. Importing the `matplotlib.pyplot` module
3. Creating an array via `np.linspace(start,stop,number)` or `np.arange(start,stop,dx)` for the range of values of the independent variables
4. Defining one or multiple mathematical functions
5. Creating a `fig` and an `ax` object
6. Preparing the function chart for plotting using the `ax.plot(x,y)` method
7. Displaying the function plot on the screen using the `fig.show()` method

A minimal, object-oriented version of a function plot thus consists of only seven program lines.

Listing 4.2 shows a program in the object-oriented style for the function plot of the parabola defined earlier in Listing 4.1.

```
#02_plotnumpy.py
import numpy as np
import matplotlib.pyplot as plt
x=np.linspace(0,10,10)
y=x**2
fig, ax = plt.subplots(figsize=(6.4,4.8),dpi=100)#640x480 pixels
ax.plot(x,y)
fig.show()
```

Listing 4.2 Parabola with NumPy Array

The program produces the same output as shown earlier in Figure 4.1.

Indications and Procedures

Line 02 imports the NumPy module for creating an array. The one-dimensional array for the value range of the independent variable x is created in line 04 using the NumPy function, linspace(). This function automatically determines the value range of the x-axis. Line 05 contains the mathematical function definition for a parabola. Other mathematical functions can also be placed here. The number of functions is limited only by the representability. A user doesn't need to worry about the scaling of the axes and their labeling with numbers since these tasks are performed automatically by the plot method.

In line 06, the subplots(figsize=(6.4,4.8)) method creates the fig and ax objects. The figsize(width, height) parameter determines the size of the graphic. The first number sets the width, and the second number sets the height of the window (figure). The numbers given are the default values. You can also omit this parameter if the default values are sufficient for your requirements. The fig object provides access to the methods responsible for the entire drawing area. You can access the methods for axis design via the ax object.

In line 07, the plot(x,y) method calculates a table of values for the coordinate data of the mathematical function. This process remains hidden from the user.

The fig.show() method (line 08) displays the function plot on the screen. However, the Matplotlib documentation and the technical literature use the notation plt.show(). I will adopt this convention for all the following program examples.

You can test the program using plt.show(fig), which makes it more clear that the entire window will be displayed on the screen. But a common practice is to avoid using this parameter.

Programming graphics applications in object-oriented style is not mandatory, as the following console program shows:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> x=np.linspace(0,10,10)
>>> plt.plot(x,x**2)
[<matplotlib.lines.Line2D object at 0x13195e980>]
>>> plt.show()
```

Thus, you could get by with only five program lines for the representation of a simple function plot. However, to provide richer functionality for presenting graphics, the object-oriented style is preferred in Matplotlib programs.

The fig and ax objects do not have to be implemented as shown in line 06 by using fig, ax=plt.subplots() within one program line. The fig and ax objects can also be created using the figure() and fig.add_subplot() methods, as in the following examples:

```
fig=plt.figure(parameter) #designing the user interface
ax=fig.add_subplot(parameter) #designing the coordinate system
#ax=fig.subplots(parameter) #for multiple subplots
```

In the following examples, I want to show in greater detail which parameters you can use. Given the large number of options, only a small selection can be presented in this chapter. For more details, see the Matplotlib documentation.

Suggestions for the Program Test

To better understand all the details of the program, you must change the parameters in the NumPy function `linspace(par1,par2,par3)` in line 04 and observe closely the effects on the display of the function plot. Alternatively, you can test the program using the NumPy function `arange(0,10,0.1)`. If you change the range of values for the x-axis, you'll notice that the scaling of the y-axis adjusts automatically.

In line 05, you can also test the program using another function, such as `np.sin(x)`. Notice how the scaling of the y-axis is automatically done.

In line 06, you can try out the different options for object creation. In particular, you should vary the `figsize` and `dpi` parameters to observe their corresponding effects on the plot's size. The values entered are default values.

To hide the display of the upper and right spines of the axes object, you must insert the following statement between lines 06 and 08:

```
ax.spines[['top', 'right']].set_visible(False)
```

Below line 07, you can insert the `fig.savefig("parabel.png")` statement. The `savefig ("name.fileextension",dpi=number)` method saves the function plot in a selectable file format. The file extension defines the file type. Besides the PNG file format, PDF (`pdf`), EPS (`eps`), JPG (`jpg`), and SVG (`svg`) formats are also possible. The `dpi` parameter sets the resolution. The default setting is 100. Thus, the graphic has a size of 640×480 pixels. You can double the size of the graphic by using `dpi=200`.

4.1.2 Gridlines

To better read function values, a useful step is to include *gridlines* in the function plot. The `grid()` method provides numerous design options for this feature, as shown in Listing 4.3.

```
01  #03_grid.py
02  import numpy as np
03  import matplotlib.pyplot as plt
04  t=np.linspace(0,20,500)
05  u=325*np.sin(2*np.pi*50*t/1000)
06  fig, ax = plt.subplots()
```

```
07  ax.plot(t,u,linewidth=2)
08  ax.grid(color='black',linestyle='solid',lw=0.5)
09  #ax.grid(color='black',ls='dashed',lw=0.5)
10  #ax.grid(color='black',ls='dotted',lw=0.5)
11  #ax.grid(color='black',ls='dashdot',lw=0.5)
12  #ax.grid(True)
13  plt.show()
```

Listing 4.3 Gridlines

Output

An example function plot with an integrated grid is shown in Figure 4.3.

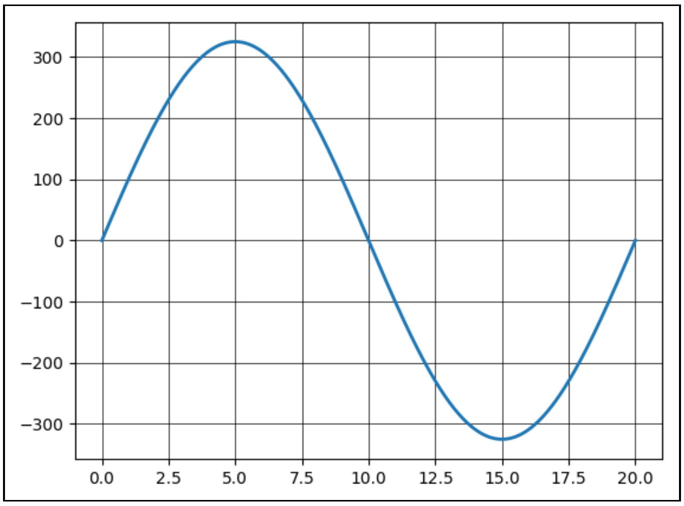


Figure 4.3 Gridlines

Analysis

The program represents the course *u(t)* of a 50 Hz alternating voltage. The `linewidth=2` parameter in line 07 sets the line width of the function plot. The abbreviated notation `lw=2` has the same effect.

Lines 08 to 12 show the different styles of possible gridlines. By removing the comments, you can test their appearance. The simplest way to display gridlines in function plots is shown by line 12, which has been commented out. The `linestyle` property can also be replaced by the abbreviated notation `ls`.

The `fig` object is not needed in this program. But if you leave it out, an error message will be displayed after the program starts. You can also use this object to save the function plot as a file by using `fig.savefig('name.fileformat')`.

For the syntax in line 06, one alternative is to create the `fig` and `ax` objects separately using `fig=plt.figure()` or `ax=fig.add_subplot()`.

4.1.3 Labels

For the labels of function plots, Matplotlib provides the `legend()` and `annotate()` methods. You can use the `set()` method to label the x and y axes. In addition, this method can also include the title of a plot. The `set_title()`, `set_xlabel`, and `set_ylabel()` methods allow you to implement these specifications separately.

Legends and Labeling the Axes

If special features of a mathematical function are to be identified, a useful task is to explain them in more detail in the function plot by using *legends*. For this purpose, Matplotlib provides the `legend(location)` method. The `location` parameter specifies the position of the legend. In addition, the x-axis and y-axis should also be labeled to clearly show the relationship between independent and dependent variables. For this purpose, Matplotlib provides the `set_xlabel()` and `set_ylabel()` methods. The separate labeling of the axes can be accommodated along with the plot title in only one program line by using the `set(xlabel='x',ylabel='y',title='Title')` method. Through the example of a 50 Hz alternating voltage, Listing 4.4 shows the implementation of a legend and the axis labels.

```
#04_labels_legend.py
import numpy as np
import matplotlib.pyplot as plt
t=np.arange(0,20,0.001)
Ueff=[230,230]
u=325*np.sin(2*np.pi*50*t*1e-3)
fig, ax = plt.subplots()
ax.plot(t,u,'b',lw=2,label='Instantaneous value: u(t)')
ax.plot([0,20],Ueff,'r--',label='RMS value: 230V')
ax.plot(5,325,'ro',label='Peak value:325V')
ax.set(xlabel='t in ms',ylabel='u(t) in V',title='50 Hz AC voltage')
#ax.legend(loc='upper right')
#ax.legend(loc='lower left')
ax.legend(loc='best')
ax.grid(color='g',ls='dashed',lw='0.5')
plt.show()
```

Listing 4.4 Legend

Output

Figure 4.4 shows the integration of a legend.

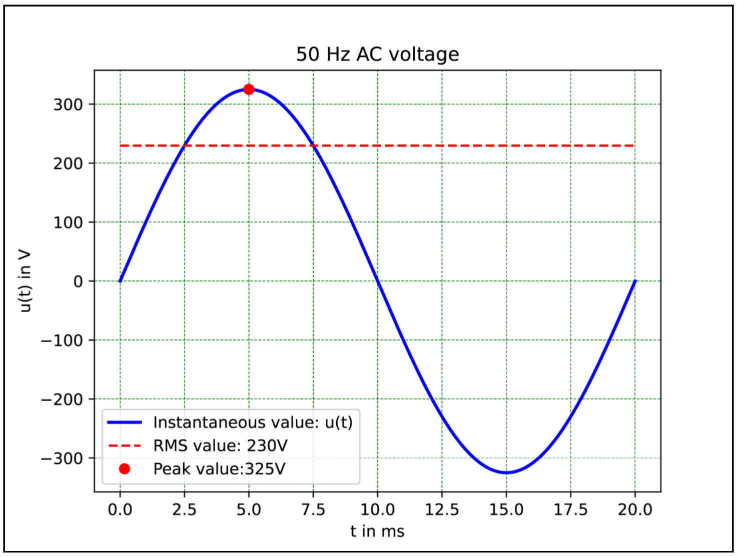


Figure 4.4 Legend

Analysis

The `Ueff=[230,230]` statement in line 05 specifies the y-coordinates of a constant voltage. The `plot()` method is passed the line colors (`b` stands for `blue`) as the third parameter (line 08). The fifth parameter `label='Instantaneous value: u(t)'` contains the label of the legend.

The third parameter (`'ro'`) in line 10 causes a red dot to be drawn at the point `t=5ms` and `u=325V`. The letter `r` stands for the color `red`. The `o` means that a point is to be drawn.

The `set()` method in line 11 creates the labels for the x-axis and the y-axis as well as the title of the plot. The `ax.legend(loc='best')` method searches for the optimal location to place the legend (line 14). The commented-out lines 12 and 13 show alternatives.

You can also spread the labels across three program lines, as in the following example:

```
ax.set_title('50 Hz AC voltage')
ax.set_xlabel('t in ms')
ax.set_ylabel('u(t) in V')
```

This variant provides more options for the positioning and color design of the axis labels.

Labeling Using the annotate() Method

The sample program shown in Listing 4.5 illustrates how you can display and mark several mathematical functions in a plot using the `annotate()` method. The program displays three resistance characteristic curves and one power hyperbola.

```
#05_labels_functions.py
import numpy as np
import matplotlib.pyplot as plt
R1,R2,R3=2,4,8
I=0.2
P=1
I=np.linspace(0.1, 1, 100)
U1=R1*I
U2=R2*I
U3=R3*I
U=P/I
fig, ax = plt.subplots()
ax.plot(I,U1,I,U2,I,U3,lw=2,color='blue')
ax.plot(I,U,lw=2,color='green')
ax.set(xlabel='I in A',ylabel='U in V',title='Power hyperbola U=P/I')
ax.annotate(r'$R_1$',xy=(1,2),xytext=(+2,-3),textcoords='offset points')
ax.annotate(r'$R_2$',xy=(1,4),xytext=(+2,-3),textcoords='offset points')
ax.annotate(r'$R_3$',xy=(1,8),xytext=(+2,-3),textcoords='offset points')
ax.grid(True)
plt.show()
```

Listing 4.5 Labeling Functions

Output

Figure 4.5 shows how multiple functions can be represented and labeled in the function plot.

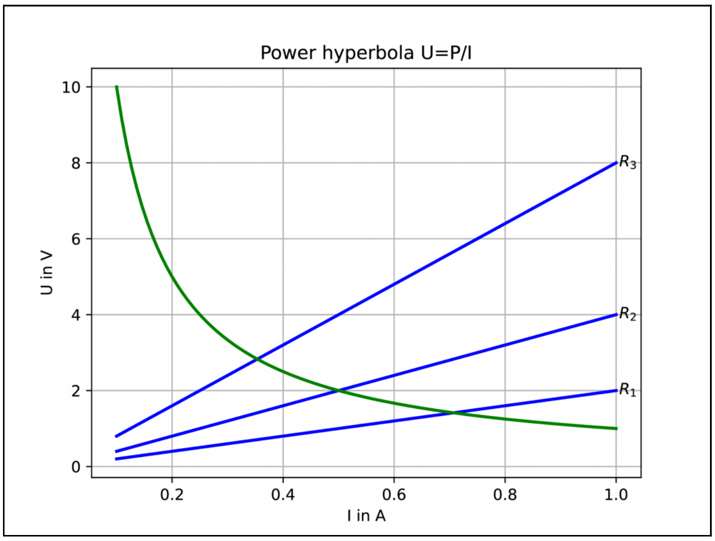


Figure 4.5 Marking Functions

Analysis

In lines 08 to 11, the functions for the resistance characteristic curves and the power hyperbola are defined. Lines 13 and 14 generate the function plots for the resistance characteristic curves and the power hyperbola.

The statements in lines 16 to 18 create the labels using the `annotate(param1,param2, param3,param4)` method. The `$R_1$` label of the resistance characteristic curve is passed as the first parameter. The underscore in the LaTeX notation causes the index to be subscript. The second parameter sets the x-y coordinates of the label. The third parameter determines the offset. The label is moved two points to the right in the x-direction and three points down in the y-direction. The fourth parameter specifies that the displacement of the label should be in terms of points.

Within the drawing area, you can also place a commenting text or even formulas by using the `text()` method, such as in the following example:

```
ax.text(0.2,9,r'voltage $U=\frac{P}{I}$',fontsize=12)
```

In this example, the first number determines the x-coordinate, and the second number determines the y-coordinate where the text should be positioned.

4.1.4 Line Styles

To better identify individual functions when plotting multiple functions in a function plot, each individual mathematical function can be assigned a special line style. Listing 4.6 shows four different line styles for the basic oscillation and three harmonics of a rectangular function.

```python
#06_linestyle.py
import numpy as np
import matplotlib.pyplot as plt
x=np.linspace(0,6.3,500)
y1=np.sin(x)
y3=np.sin(3*x)/3
y5=np.sin(5*x)/5
y7=np.sin(7*x)/7
y=y1+y3+y5+y7
fig, ax = plt.subplots()
ax.plot(x,y1,color='b',lw=2,linestyle='-') #blue
ax.plot(x,y3,color='r',lw=2,linestyle='--')#red
ax.plot(x,y5,color='m',lw=2,linestyle=':') #magenta
ax.plot(x,y7,color='g',lw=2,linestyle='-.')#green
ax.plot(x,y,color='black',lw=3)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.grid(True)
plt.show()
```

Listing 4.6 Line Styles

Output

The line styles that can be used to distinguish between different functions are shown in Figure 4.6.

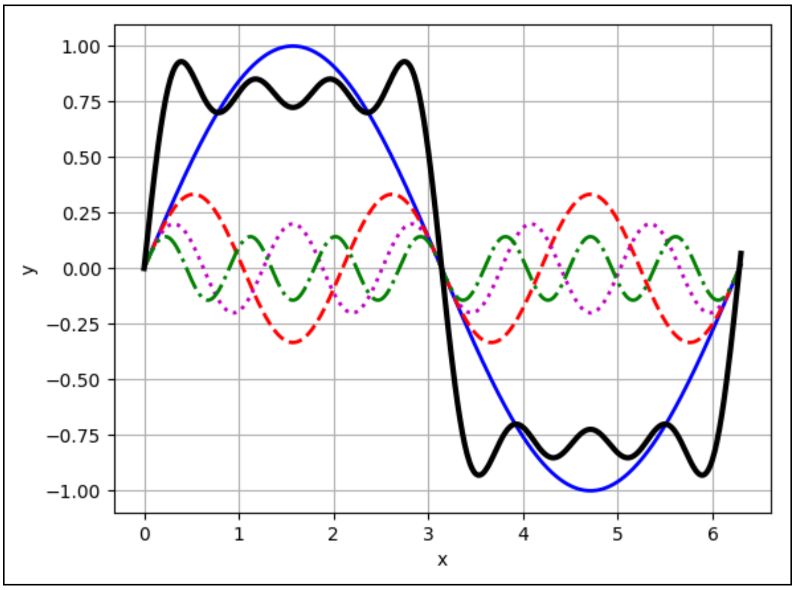


Figure 4.6 Different Line Styles

Analysis

Lines 11 to 14 define the line styles. For a solid line, a dash is assigned to the `linestyle='-'` property. A double hyphen (`--`) draws a dashed line. A colon (`:`) draws a dotted line. A dash and a dot (`-.`) draw a dash-dot line. If no property is specified for the line style (line 15), then a solid line will be drawn by default. The `linestyle` property can also be abbreviated as `ls`.

You can also change the line styles and line colors using the abbreviations `b-`, `r--`, `m:`, and `g-`.

4.1.5 Designing Axes

Previously, Matplotlib automatically performed axis scaling and specified the outer frame of the function plot. However, in many cases, changing the axis scaling or the shape of the coordinates in the form of a cross might be desirable.

Changing the Axis Scaling

Listing 4.7 shows how to change axis scaling. This data originates from a tensile test. The relationship between the length of the test bar and the tensile force applied is shown.

```python
#07_axis_scaling.py
import matplotlib.pyplot as plt
l=[0,0.02,0.1,0.2,1.15,2.2,3.25,4.3,5.4,6.4]
F=[5.7,7.5,7.2,7.3,8.9,10.4,11.3,12,11.4,9.3]
fig, ax = plt.subplots()
ax.plot(l, F,'ro-')
ax.set_xticks([0,1,2.2,3.25,4.3,5.4,6.4])
ax.set_yticks([0,5.7,7.5,8.9,10.4,12,9.3])
#ax.axis([-0.5,7,5,13])
ax.set_xlabel("l in mm")
ax.set_ylabel("F in kN")
plt.show()
```

Listing 4.7 Individual Axis Scaling

Output

The output of the individual axis scaling from Listing 4.7 is shown in Figure 4.7.

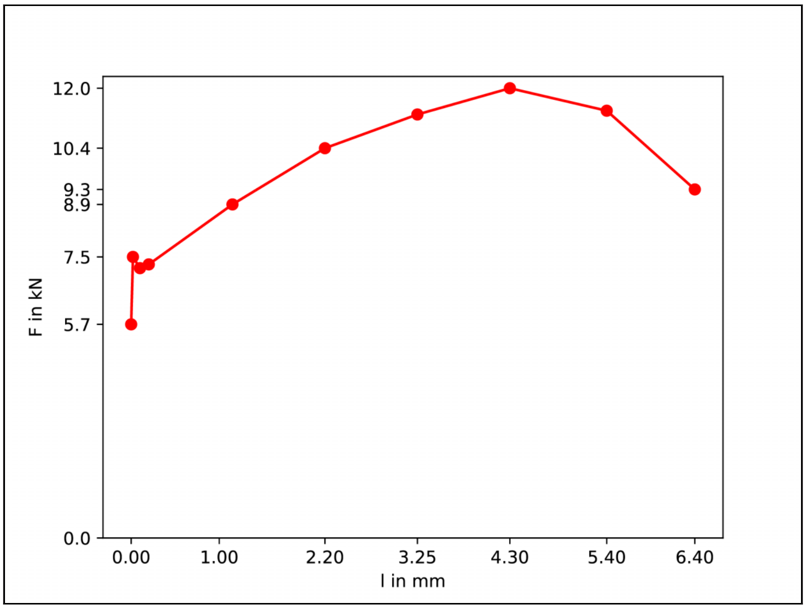


Figure 4.7 Individual Axis Scaling

Analysis

Lines 03 and 04 contain the data for the bar length and the tensile force obtained from a tensile test. Line 06 specifies that the tensile force is displayed as a red line with red markings of the measuring points over the bar length.

The `set_xticks([])` and `set_yticks([])` methods in lines 07 and 08 set the scaling for prominent function values. The alternative in the commented-out line 09 causes the function to be displayed on the x-axis for the value range from -0.5 to 7 and on the y-axis for the value range from 5 to 13.

Creating an Axis Cross

Listing 4.8 shows how you can create an axis cross using the `spines` method. The following statements hide the `top` and `right` spines of the axes object via `set_visible(False)`:

```
spines['top'].set_visible(False)
spines['right'].set_visible(False)
```

The following statements move the left and right spines to the coordinate origin:

```
spines['left'].set_position(("data", 0))
spines[,'bottom']].set_position(("data", 0))
```

A parabola $y = x^2 - 4$ shifted on the y-axis serves as a demonstration object:

```
#08_axis_style.py
import numpy as np
import matplotlib.pyplot as plt
#Function definition
def f(x):
    #y=np.sin(np.pi*x)
    y=x**2-4
    return y
#Graphics area
fig, ax = plt.subplots()
ax.spines[['top', 'right']].set_visible(False)
ax.spines[['left', 'bottom']].set_position(("data", 0))
x = np.linspace(-5, 5, 100)
ax.plot(x,f(x),'r-',lw=2)
ax.set_xlabel('x',loc='right')
ax.set_ylabel('f(x)',loc='top',rotation=0)
plt.show()
```

Listing 4.8 Axis Cross

Output

The result of the conversion to an axis cross is shown in Figure 4.8.

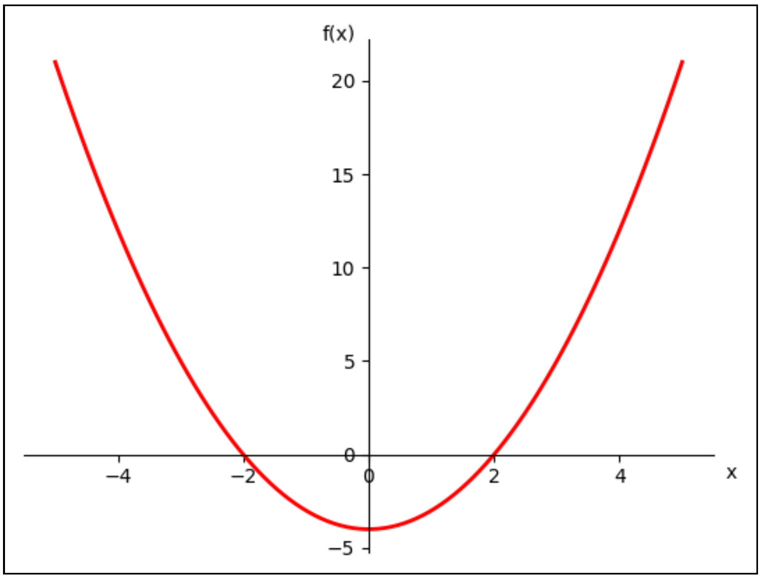


Figure 4.8 Axis Cross

Analysis

The statement in line 11 will hide the upper and right spines.

In line 12, the left and bottom axes are moved to the coordinate origin.

To embellish the x-axis and the y-axis with arrows, you must insert the following lines into the graphics area:

```
ax.plot(1,0,'>k',transform=ax.get_yaxis_transform(), clip_on=False)
ax.plot(0,1,'^k',transform=ax.get_xaxis_transform(), clip_on=False)
```

Logarithmic Scale Division

A logarithmic scale division is useful when the range of values of the data to be represented spans many orders of magnitude. The logarithmic representation makes correlations in the range of small values more visible. For the representation of the transmission behavior of low-pass filters, a logarithmic scale division of the frequency axis is usually always selected. The transfer behavior (frequency response) of a Butterworth low-pass filter of the *n*th degree is described by the following formula:

$$A = \frac{1}{\sqrt{1+\Omega^{2n}}}$$

Ω is the frequency normalized to 1 Hz and *n* is the degree of the filter. Up to the cutoff frequency of 1 Hz, the transfer factor remains almost constant, especially when the degree of the filter is increased.

Listing 4.9 shows how the transfer behavior $A = f(\Omega)$ of a Butterworth low-pass filter of the first to third degree with logarithmic scale division is represented. Due to the `semilogx()` method, the x-axis gets a logarithmic scale division.

```
#09_log_axis.py
import numpy as np
import matplotlib.pyplot as plt
omega=np.linspace(0.1,100,1000)
fig, ax = plt.subplots()
for n in range(1,4):
    A=1./(np.sqrt(1.+omega**(2*n)))
    ax.semilogx(omega,A)
ax.set_xlabel('Frequency in Hz')
ax.set_ylabel('A')
ax.grid(True)
plt.show()
```

Listing 4.9 Logarithmic Scale Division

Output

The logarithmic scale generated using the `semilogx()` method is shown in Figure 4.9.

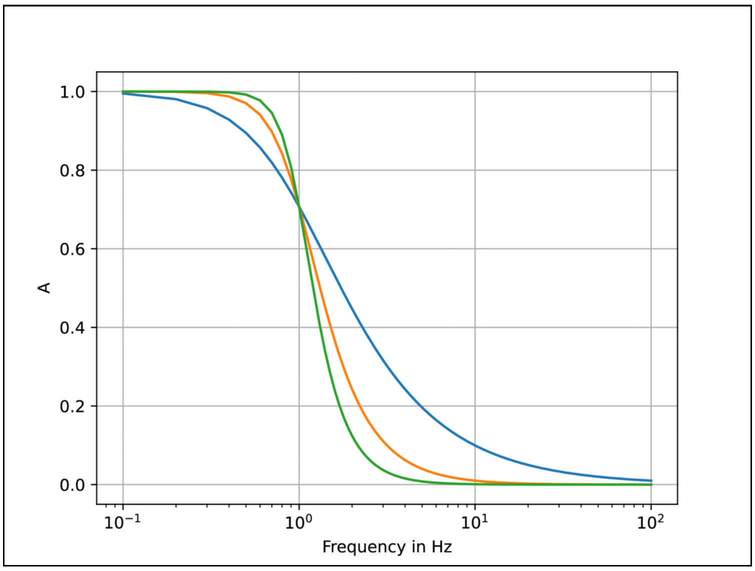


Figure 4.9 Logarithmic Scale Division

Analysis

The transmission behavior shown in this example clearly shows that the transmission factor hardly changes up to the cutoff frequency of 1 Hz. The `semilogx(omega,A)` method in line 08 causes the x-axis to be scaled logarithmically in the range from 0.1 Hz to 100 Hz. Between the frequencies of 0.1 to 1 Hz, 1 Hz to 10 Hz and 10 Hz to 100 Hz, the sections on the x-axis have the same length.

If you add the `ax.set_xscale('log')` method below line 05 and change the `ax.semilogx (omega,A)` statement to `ax.plot (omega,A)`, the x-axis will also get a logarithmic scale division.

Polar Coordinates

In mathematics, a polar coordinate system (also called a circular coordinate system) is a two-dimensional coordinate system in which each point is defined by the distance from the center and an angle. Lines are represented using the `plot(a1,a2,r1,r2)` method. The abbreviations `a` and `r` stand for angle and radius, respectively. Listing 4.10 demonstrates how a square and a line are represented in a polar coordinate system.

```python
#10_polar_coordinates.py
import numpy as np
import matplotlib.pyplot as plt

def theta_rad(angle1,angle2):
    theta=[angle1,angle2]
    return np.radians(theta)

fig, ax = plt.subplots(subplot_kw={'projection': 'polar'})
ax.plot(theta_rad(0,90),[1,1],'r',lw=3)
ax.plot(theta_rad(90,180),[1,1],'g',lw=3)
ax.plot(theta_rad(180,270),[1,1],'m',lw=3)
ax.plot(theta_rad(270,0),[1,1],'b',lw=3)
ax.plot(theta_rad(0,45),[0,0.6],'black',lw=3)
ax.grid(True)
plt.show()
```

Listing 4.10 Polar Coordinates

Output

Figure 4.10 shows lines in a diagram with polar coordinates.

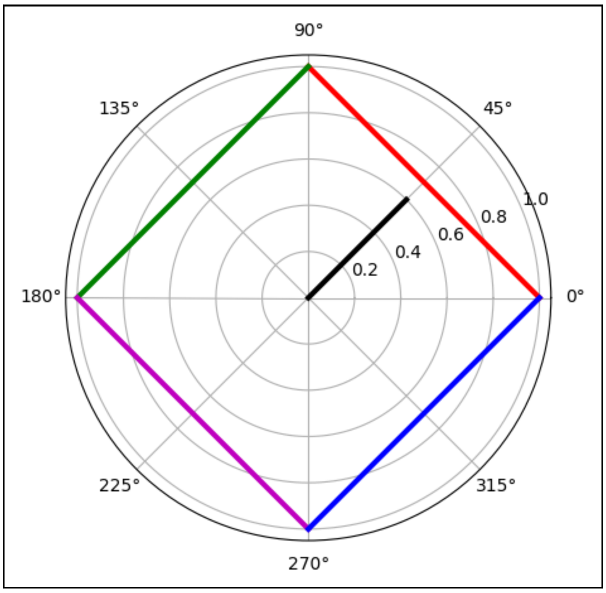


Figure 4.10 Polar Coordinates

Analysis

In line 09, the `subplot_kw={'projection': 'polar'}` parameter in the `subplots()` method specifies that the default Cartesian coordinate system is converted to a polar coordinate system. The labels for the angles and radiuses are generated automatically. Using lines 10 to 13, the `plot` method draws the four lines of a square.

In line 14, the following method draws a black line (radius) with polar coordinates 45° and 0.6:

```
ax.plot(theta_rad(0,45),[0,0.6],'black',lw=3)
```

4.1.6 Coloring Areas

A convenient task, in some cases, is to highlight a surface by coloring, which occurs when a parabola opens in the direction of the positive y-axis and a parabola opens in the direction of the negative y-axis intersect at two points. Likewise, the coloring of the area under a function graph and its intersections with the x-axis could be equally useful. You can perform this task using the `fill_between()` method.

Coloring Areas between Two Function Graphs

By coloring areas between two function graphs, you can illustrate the calculation of an area integral. Listing 4.11 shows how to color the areas between the intersections of two parabolas defined by the following equations:

$y_1 = (x-3)^2$

$y_2 = -(x-2)^2 + 8$

```
01  #11_color_parabola.py
02  import numpy as np
03  import matplotlib.pyplot as plt
04  x = np.linspace(0,5,100)
05  y1 = (x-3)**2
06  y2 = -(x-2)**2+8
07  fig, ax=plt.subplots()
08  ax.plot(x, y1, x, y2, color='black')
09  ax.fill_between(x,y1,y2,where=y2>=y1,facecolor='b',alpha=0.2)
10  ax.set_xlabel('x')
11  ax.set_ylabel('y')
12  plt.show()
```

Listing 4.11 Coloring Areas between Function Graphs

Output

The result of the coloring between the function graphs from Listing 4.11 is shown in Figure 4.11.

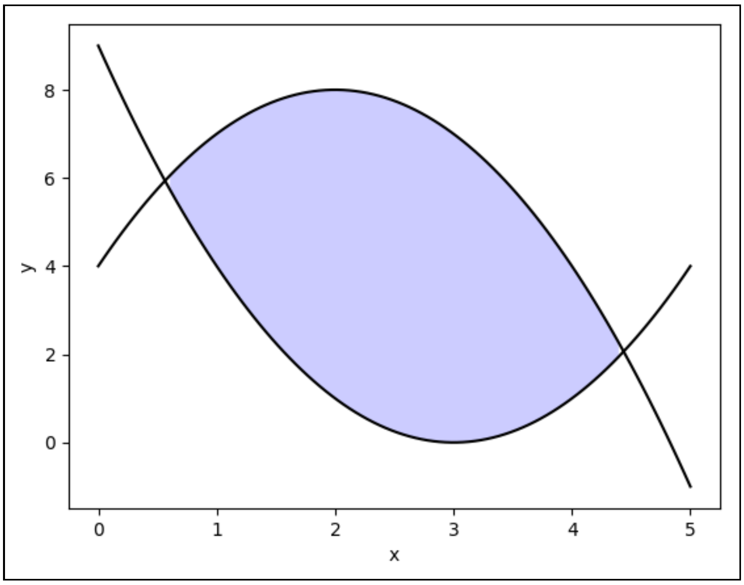


Figure 4.11 Coloring Areas between Function Graphs

Analysis

In line 09, the following method is used to color the area between the intersections of the two parabolas:

```
ax.fill_between(x,y1,y2,where=y2>=y1,facecolor='b',alpha=0.2)
```

The range of coloring is defined by the `where=y2>=y1` condition. Thus, only for the range of values in which `y2` is greater than or equal to `y1`, the coloring should be done. The `alpha=0.2` parameter sets the transparency (translucency) of the coloring. The smaller the value, the higher the transparency.

Coloring Areas above and below the x-Axis of a Sine Function

Listing 4.12 shows the coloring between the function graph of the power curve $p(t)$ and the x-axis using the example of the AC power.

```
01  #12_color_power.py
02  import numpy as np
03  import matplotlib.pyplot as plt
04  f=50
05  URms=230
06  R=10
07  Xc=10
08  XL=0
09  Z=np.sqrt(R**2 + (XL-Xc)**2)
10  phi=np.arctan((XL-Xc)/R)
11  I=URms/Z
12  t=np.linspace(0,20,500)
13  u=np.sqrt(2)*URms*np.sin(2*np.pi*f*t*1e-3)
14  i=np.sqrt(2)*I*np.sin(2*np.pi*f*t*1e-3-phi)
15  p=u*i
16  fig, ax=plt.subplots()
17  ax.plot(t, p, color='black')
18  ax.fill_between(t,0,p,where=p >=0,facecolor='r',alpha=0.2,
label='positive portion')
19  ax.fill_between(t,0,p,where=p <=0,facecolor='g',alpha=0.2,
label='negative portion')
20  ax.set(xlabel='t in ms',ylabel='p(t) in Watt',title= 'AC power')
21  ax.legend(loc='best')
22  plt.show()
```

Listing 4.12 Coloring between the Function Graph and the x-Axis

Output

How helpful this coloring can be for visualizing and quickly grasping the content using the `matplotlib` module is shown in Figure 4.12.

Analysis

The statement in line 18 colors the area of the power curve for the positive range `where=p>=0` in red, while line 19 colors the negative range `where=p<=0` of the power curve in green. The legend at the bottom left of the diagram identifies the two areas.

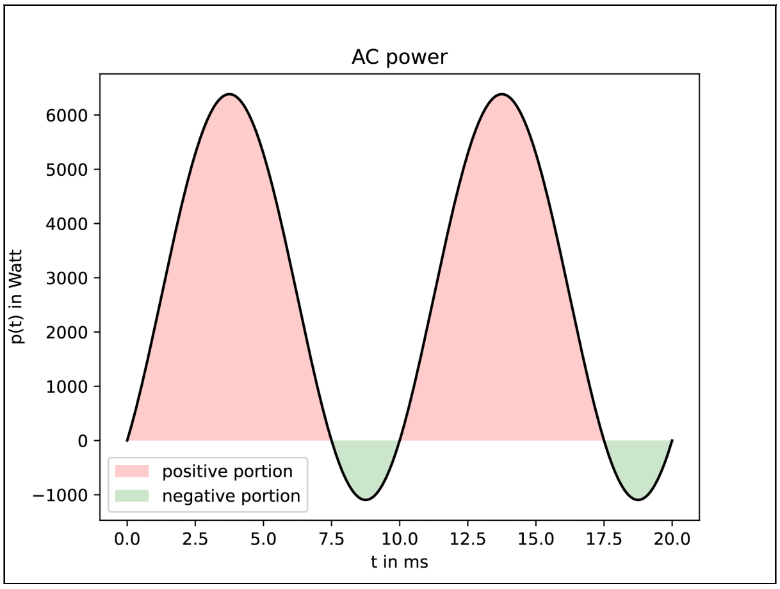


Figure 4.12 Coloring between Function Graph and x-Axis

4.1.7 Subplots

Matplotlib provides the following method:

```
fig,ax = plt.subplots(rows,columns)
```

This method enables you to display multiple mathematical functions with different value ranges in *subplots*.

This method returns the `fig,ax` tuple. The `fig` object determines the dimensions of the entire drawing area, and the `ax` object lets you access the properties of the axes.

Row and Column Layout

Listing 4.13 represents four function plots: a linear function with positive slope, a linear function with negative slope, a parabola, and a hyperbola. The four subplots are created internally using the `ax= subplots(2,2)` statement. The `ax` object is of the `<class 'numpy.ndarray'>` type and can be treated like a 2×2 matrix. The indexes determine the position of the subplot on the screen.

```
#13_subplot_functions.py
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0, 10, 100)
y1=x
y2=5-x
y3=x**2
```

```
y4=1/(0.2*x+1)
fig, ax = plt.subplots(2, 2)
#1st row, 1st column
ax[0,0].set(ylabel='y',title='Linear function')
ax[0,0].plot(x,y1,'b',lw=2)#blue
ax[0,0].grid(True)
#1st row, 2nd column
ax[0,1].set(title='negative slope')
ax[0,1].plot(x,y2,'r',lw=2)#red
ax[0,1].grid(True)
#2nd row, 1st column
ax[1,0].set(xlabel='x',ylabel='y',title='Parabola')
ax[1,0].plot(x,y3,'g',lw=2)#green
ax[1,0].grid(True)
#2nd row, 2nd column
ax[1,1].set(xlabel='x',title='Hyperbola')
ax[1,1].plot(x,y4,'k',lw=2)#black
ax[1,1].grid(True)
fig.tight_layout()
plt.show()
```

Listing 4.13 Function Representation in Four Subplots

Output

How the four functions are represented in four subplots is shown in Figure 4.13.

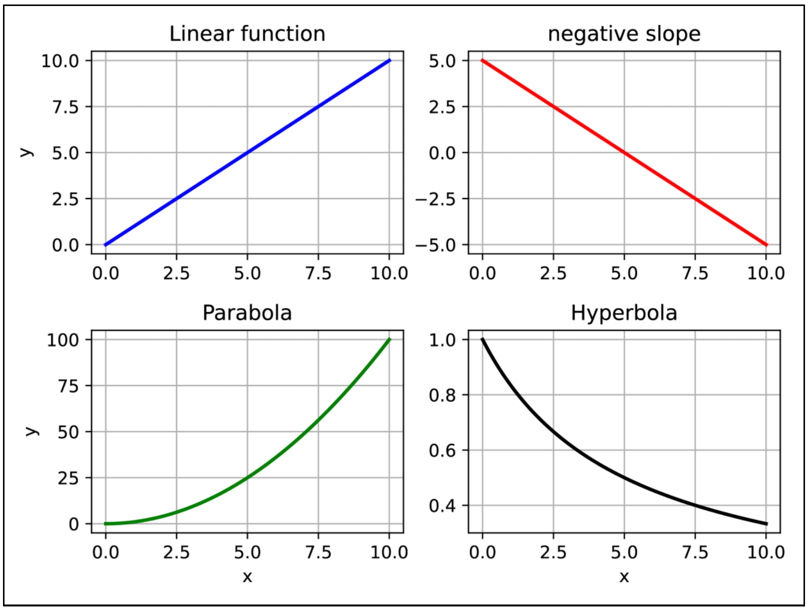


Figure 4.13 Function Representation in Four Subplots

Analysis

In lines 05 to 08, four mathematical functions are defined. The first two parameters of the `subplots(2,2)` method in line 09 specify that the diagram consists of subplots with two rows and two columns. The first number determines the number of rows, and the second number determines the number of columns.

The arrangement of the subplots is determined by the indexing. For example, the index pair `[0,0]` specifies that the first subplot is placed in the first row and column. The second subplot in the first row and second column is indexed with `[0,1]` and so on.

The `tight_layout()` method in line 26 is tasked with creating enough space between the four function plots. You can change the spacing using the `pad`, `w_pad`, and `h_pad` parameters. The `pad` property determines the distance between the edges of the drawing area and the edges of the subplots as a fraction of the font size. The default value of `pad` is 1.08. The `w_pad` and `h_pad` properties set the vertical and horizontal spacing between subplots.

Usage Example: AC Power

The next example shown in Listing 4.14 represents the time history of voltage, current, and power in a 50 Hz AC circuit. The subplots are arranged below each other in three lines.

```
01  #14_subplot_power.py
02  import numpy as np
03  import matplotlib.pyplot as plt
04  f=50
05  URms=230
06  R=0.001
07  Xc=10
08  XL=0
09  Z= np.sqrt(R**2+(XL-Xc)**2)
10  phi=np.arctan((XL-Xc)/R)
11  I=URms/Z
12  t=np.linspace(0.0, 20, 1000)
13  u=np.sqrt(2)*URms*np.sin(2*np.pi*f*t*1e-3)
14  i=np.sqrt(2)*I*np.sin(2*np.pi*f*t*1e-3-phi)
15  p=u*i
16  fig, ax = plt.subplots(3,1)
17  #voltage
18  ax[0].plot(t, u,'b',lw=2)
19  ax[0].set_ylabel('u(t)')
20  ax[0].grid(True)
21  #current
22  ax[1].plot(t,i,'r',lw=2)
```

```
23  ax[1].set_ylabel('i(t)')
24  ax[1].grid(True)
25  #power
26  ax[2].plot(t,p,'g',lw=2)
27  ax[2].set(xlabel='Time in ms',ylabel='p(t)')
28  ax[2].grid(True)
29  fig.tight_layout()
30  plt.show()
```

Listing 4.14 Function Representation in Three Rows

Output

How the functions of the example are represented in three subplots in three rows below each other is shown in Figure 4.14.

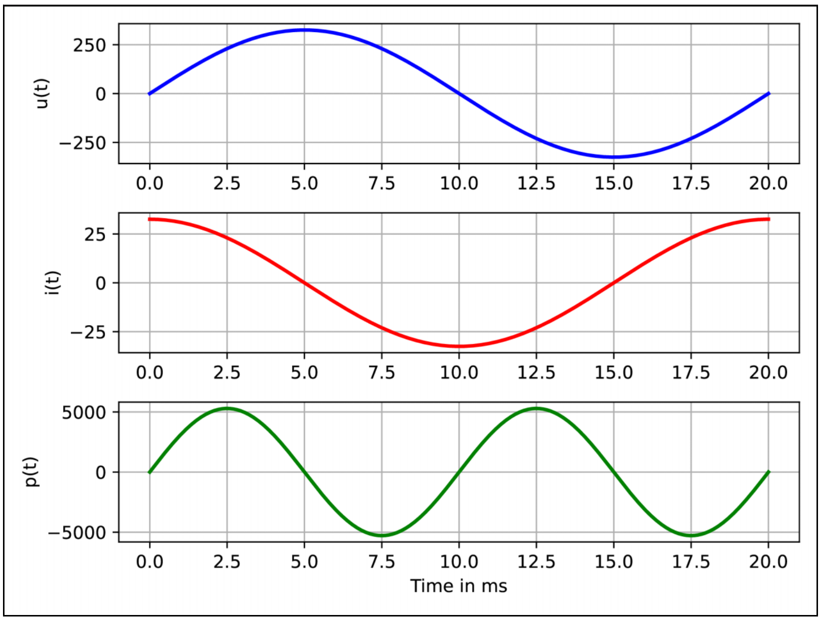


Figure 4.14 Function Representation in Three Rows

Analysis

In line 16, the `fig` and `ax` objects are created using the `subplots(3,1)` method. The two parameters specify that the subplots are arranged in three rows and one column. The index of `ax` determines the order of the subplots to be displayed. First, the course of the voltage is shown with the line color blue, then the course of the current with the line color red, and finally in the third line the course of the power with the line color green. You can arrange the code for the subplots in a different order, and the described representation will not change.

Inserting an Axes Object Into Another Axes Object

Sometimes, highlighting a particular detail of a function plot is useful. The following statement allows you to embed a subplot into an already existing plot:

```
ax2=fig.add_axes([left,bottom,width,height])
```

The numerical values for the parameters must be between 0 and 1.

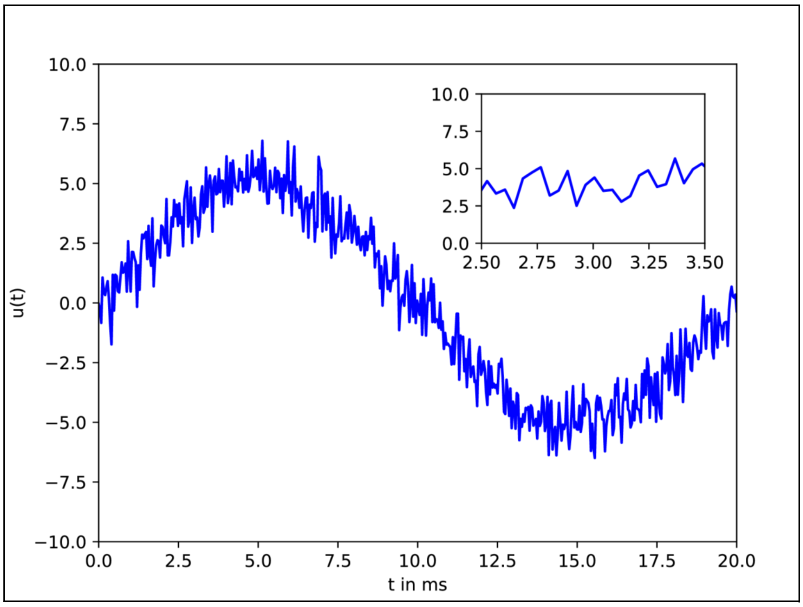


Figure 4.15 Embedding a Subplot

Listing 4.15 shows how a small section of a noisy sinusoidal signal is clarified in an embedded plot (subplot). This section is displayed enlarged (zoomed) to make the details of the signal more visible.

```
#15_axes_axes.py
import numpy as np
import matplotlib.pyplot as plt
f=50 #Frequency in Ht
tmax=20 #Time in ms
t = np.linspace(0, tmax, 500)
ut=5*np.sin(2*np.pi*f*t*1e-3) + 0.8*np.random.randn(t.size)
fig=plt.figure()
#left, bottom, width, height
ax1=fig.add_axes([0.12,0.1,0.8,0.8]) #outside
ax2=fig.add_axes([0.6,0.6,0.28,0.25])#inside
#x1,x2,y1,y2
ax1.axis([0,tmax,-10,10])
```

```
14 ax2.axis([2.5,3.5,0,10])
15 #Graphics output
16 ax1.plot(t,ut,"b-")
17 ax1.set_xlabel('t in ms')
18 ax1.set_ylabel('u(t)')
19 ax2.plot(t,ut,"b-")
20 plt.show()
```

Listing 4.15 Inserting an Axes Object into Another Axes Object

Output

The output is shown in Figure 4.15.

Analysis

In line 08, the `figure()` method creates the `fig` object. Using this object, you can access the `add_axes()` method.

In line 11, the `add_axes([0.6,0.6,0.28,0.25])` method creates the `ax2` object. For the subplot, the default values of 640×480 pixels and a resolution of 100 dpi apply: The left-hand distance is 0.6×640 pixels = 384 pixels, and the bottom-left corner has a distance of 0.6×480 pixels = 288 pixels from the bottom edge. This corresponds to 60% each of the total width and height of the plot area. The width has a value of 0.28×640 pixels = 179 pixels, and the height has a value of 0.25×480 pixels = 120 pixels. This represents 28% of the total width and 25% of the total height of the plot area.

In lines 13 and 14, the `axis()` method sets the range of values for the x and y axes. In line 14, you can influence the zoom effect by changing the `x1`, `x2`, `y1`, and `y2` arguments.

The `ax1` object in line 10 can also be created more easily via the `ax1=fig.add_subplot()` statement. In this case, the default values apply. You can query them using print`(ax1.get_position())`, which will return the following output:

```
Bbox(x0=0.125, y0=0.10999999999999999, x1=0.9, y1=0.88)
```

Combining Polar and Cartesian Coordinates

In the third example on subplots (see Listing 4.16), we want to combine a polar coordinate system with a Cartesian coordinate system. This time, the Cartesian coordinate system is created using the `spines()` method. In fact, although the word *spine* has the connotation of a *backbone* or *spinal column*, the Matplotlib documentation understands *spines* to be lines that delimit the drawing area. These lines can be made invisible via `spines['location'].set_visible(False)`. By using `spines[location].set_position(('data',0))`, you can move lines to the origin of the coordinate system. For `location`, only the `top`, `bottom`, `left`, and `right` parameters are allowed.

```
#16_subplot_polar_sinus.py
import numpy as np
import matplotlib.pyplot as plt

def theta_rad(angle1,angle2):
    theta=[angle1,angle2]
    return np.radians(theta)

angle=45
x=np.linspace(0, 360, 500)
y=np.sin(np.pi*x/180)
r=[np.cos(np.radians(angle)),1]
fig=plt.figure(figsize=(8,4))
#Polar coordinates
ax1=fig.add_subplot(1,2,1,projection='polar')
ax1.set_rticks([])
ax1.plot(theta_rad(0,angle),[0,1],'b',lw=2)
ax1.plot(theta_rad(0,angle),r,'b',lw=2)
ax1.plot(theta_rad(0,angle),[0,1],'ro')
ax1.grid(True)
#Cartesian coordinates
ax2=fig.add_subplot(1,2,2)
ax2.spines[['top', 'right']].set_visible(False)
ax2.spines[['bottom', 'left']].set_position(('data',0))
ax2.plot(x, y,'b',linewidth=2)
ax2.plot(angle,np.sin(np.radians(angle)),'ro')
ax2.plot(0,np.sin(np.radians(0)),'ro')
wg=[]
for w in range(0,361,45):
    wg.append(w)
ax2.set_xticks(wg[1:])
ax2.set_xlabel('x in °',loc='right')
ax2.set_ylabel('f(x)',loc='top',rotation=0)
plt.show()
```

Listing 4.16 Combining Polar and Cartesian Coordinates

Output

The output of the combination of polar and Cartesian coordinates in two subplots side by side is shown in Figure 4.16.

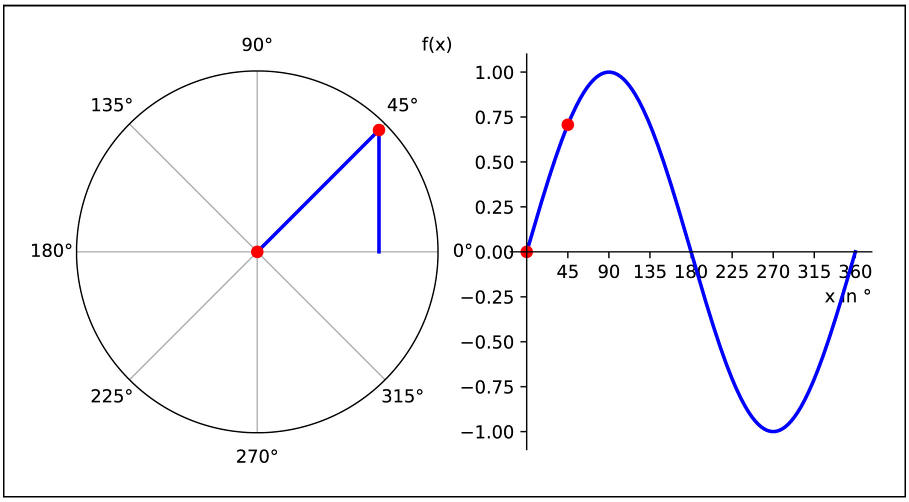


Figure 4.16 Polar and Cartesian Coordinates

Analysis

The most important implementation details are already known from our earlier examples. In line 15, the `add_subplot(1,2,1,projection='polar')` method creates the `ax1` object. The two subplots are displayed in one row and two columns. The fourth parameter (`projection='polar'`) specifies that the first subplot is displayed with polar coordinates.

In line 22, the `add_subplot(1,2,2)` method creates the `ax2` object. This object accesses the `spines` method in lines 23 and 24.

The statement in line 23 causes the upper and right-hand frame lines (*spines*) of the drawing frame not to be displayed. In line 24, `set_position(('data',0))` causes the bottom line of the drawing frame to be moved to the origin of the coordinate system. You can test the program by inserting a value other than 0 for the position in line 24, and you'll see how the x-axis shifts upward or downward.

The `set_xticks()` method sets the individual label of the x-axis (line 31). The degree measure used in this context, which is not popular with mathematicians, can also be represented as a radian using LaTeX notation, as in the following example:

```
ax2.set_xticks([45,90,135,180,225,270,315,360],
          [r'$\frac{1}{4}\pi$',r'$\frac{1}{2}\pi$',
           r'$\frac{3}{4}\pi$',r'$\pi$',r'$\frac{5}{4}\pi$',
           r'$\frac{3}{2}\pi$',r'$\frac{7}{4}\pi$',r'$2\pi$'])
```

4.1.8 Parameter Representation

In mathematics, a parameter representation is the representation of a mathematical function in which the (x,y) points of a curve are traversed as a function of a variable (i.e., the parameter). The *oblique throw* and the *lemniscate* were chosen as examples of parameter representation.

Oblique Throw

In an oblique throw, the time t is the parameter on which the x and y components of the trajectory (throwing parabola) depend. The place-time law is determined by the initial velocity v_0 and by the cosine and sine of the throwing angle α :

$$x = v_0 \cdot t \cdot \cos\alpha$$

$$y = v_0 \cdot t \cdot \sin\alpha - \frac{g}{2}t^2$$

The throwing time is calculated with the following formula:

$$t_{\max} = \frac{2v_0 \cdot \sin\alpha}{g}$$

Listing 4.17 shows how you can implement parameter equations as Python source code. The course of the trajectory of an oblique throw for a throwing angle of 45° with an initial velocity of 20 m/s is shown.

```
#17_parameter_throw.py
import numpy as np
import matplotlib.pyplot as plt
g=9.81    #Gravitational acceleration in m/s^2
v0=20     #Initial velocity in m/s
alpha=45 #Throwing angle in °
alpha=np.radians(alpha)
tmax=2*v0*np.sin(alpha)/g
t=np.linspace(0,tmax,100)
#Parameter equations
x=v0*np.cos(alpha)*t
y=v0*np.sin(alpha)*t-0.5*g*t**2
#Representation
fig, ax=plt.subplots()
ax.plot(x,y,linewidth=2)
ax.set(xlabel='x in m',ylabel='y in m')
ax.grid(True)
plt.show()
```

Listing 4.17 Oblique Throw

Output

The output of the trajectory of the oblique throw is shown in Figure 4.17.

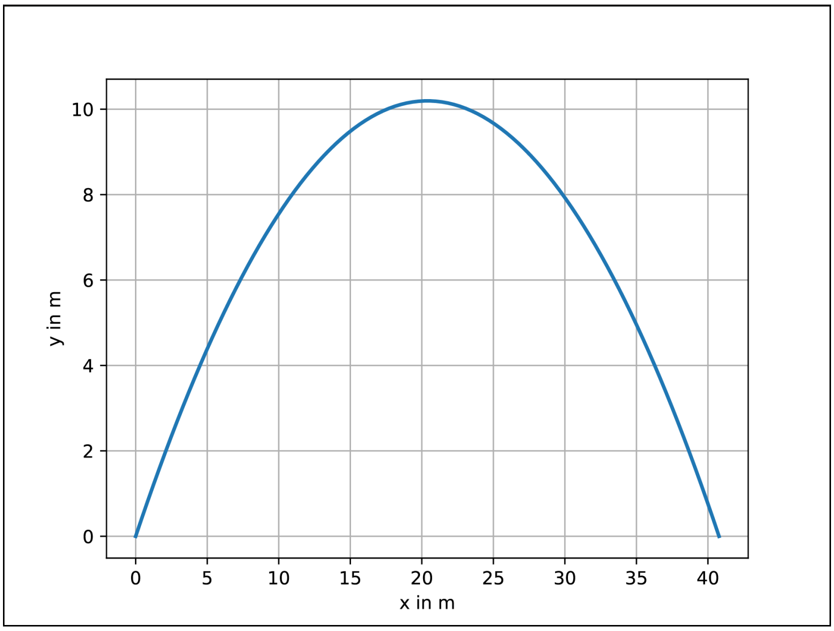


Figure 4.17 Trajectory of an Oblique Throw

Analysis

Line 08 calculates the throwing time `tmax`. This time is needed to set the array for the parameter `t` in line 09. In lines 11 and 12, the x and y components of the trajectory are calculated and stored as an array with 100 values each in variables `x` and `y`. The preparation for the representation of the function plot is again done using the `plot(x,y,...)` method.

Lemniscate

The *lemniscate* by Jakob Bernoulli (1654–1705) is a plane curve in the shape of a lying eight. It describes the motion curve in the Watt parallelogram (James Watt, 1736–1819). The following parameter equations can be used to plot the lemniscate as a function plot:

$$x = \frac{a\sqrt{2}\cos t}{\sin^2 t + 1}$$

$$y = \frac{a\sqrt{2}\cos t \cdot \sin t}{\sin^2 t + 1}$$

Listing 4.18 shows the source code for the graphical representation of a lemniscate.

```
#18_parameter_lemniscate.py
import numpy as np
import matplotlib.pyplot as plt
```

```
04  t=np.linspace(-np.pi,np.pi,200)
05  a=1
06  #Parameter equations
07  x=a*np.sqrt(2)*np.cos(t)/(np.sin(t)**2+1)
08  y=a*np.sqrt(2)*np.cos(t)*np.sin(t)/(np.sin(t)**2+1)
09  #Representation
10  fig, ax=plt.subplots()
11  ax.plot(x,y,linewidth=2)
12  ax.set(xlabel='x',ylabel='y')
13  ax.grid(True)
14  plt.show()
```

Listing 4.18 Lemniscate

Output

Figure 4.18 shows the graphical representation of the lemniscate.

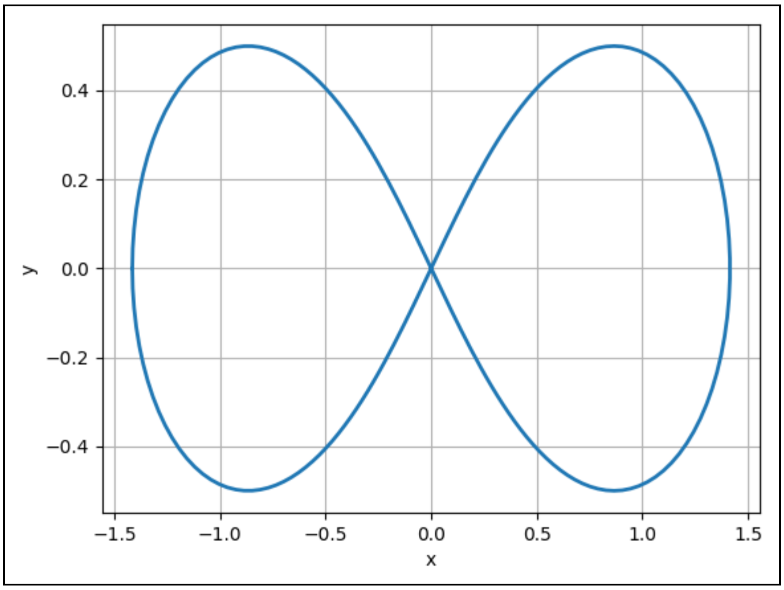


Figure 4.18 Lemniscate

Analysis

The t parameter in line 04 does not have the meaning of a time in this case but of a range of values. In the t variable, 200 values from −π to +π are stored. Lines 07 and 08 each calculate the 200 values for the x and y components for the t parameter, which are then cached in line 11 via `ax.plot(x,y,...)` for display using `plt.show()` in line 14. The a variable in line 05 determines the extent of the lemniscate on the x-axis.

4.1.9 Changing Function Parameters Interactively

The matplotlib module also provides the option to write interactive programs using the widgets submodule. The usual controls are available for this purpose, such as a command button (Button); selection controls (CheckButton, RadioButton); sliders (Slider); and text boxes (TextBox). You can import the controls using the following statement:

```
from matplotlib.widgets import Slider, Button, ...
```

An overview of the controls provided by matplotlib.widgets can be found at *https://matplotlib.org/stable/api/widgets_api.html*.

However, the options to write “real” interactive programs using the widgets submodule are rather limited. If you want to write more complex interactive programs with graphical user interfaces (GUIs), you should use tkinter or PyQt5, for example.

The implementation of an interactive Matplotlib program is done in six steps:

1. Importing matplotlib.widgets with the classes for the controls.
2. Defining a mathematical function whose parameters are to be changed.
3. Positioning the controls.
4. Creating objects for controls.
5. Defining functions for event processing.
6. Querying events using the built-in on_changed() and on_clicked() methods.

The example shown in Listing 4.19 is taken from the Matplotlib documentation. This example shows how the amplitude and frequency of a sine function can be changed interactively during runtime using two *sliders*.

```
#19_slider_sinus.py
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.widgets import Slider,Button
fig,ax=plt.subplots()
fig.subplots_adjust(left=0.2,bottom=0.25)
t=np.linspace(0.0,1.0,200)
a0=5
f0=5
s=a0*np.sin(2*np.pi*f0*t)
kurve, = ax.plot(t,s,lw=2,color='blue')
ax.axis([0, 1, -10, 10])
#Position objects for controls
#left margin, bottom margin, length, height
xyAmp =  fig.add_axes([0.25, 0.15, 0.65, 0.03])
```

```
xyFreq = fig.add_axes([0.25, 0.1, 0.65, 0.03])
xyReset= fig.add_axes([0.8,0.025,0.1,0.04])
#Create objects for controls
sldAmp=Slider(xyAmp,'Amplitude',1,10,valinit=a0,valstep=0.1)
sldFreq=Slider(xyFreq,'Frequency',1,10,valinit=f0,valstep=0.1)
cmdReset=Button(xyReset,'Reset')

def update(val):
    A = sldAmp.val
    f = sldFreq.val
    kurve.set_data(t,A*np.sin(2*np.pi*f*t))

def reset(event):
    sldFreq.reset()
    sldAmp.reset()
#Event processing
sldAmp.on_changed(update)
sldFreq.on_changed(update)
cmdReset.on_clicked(reset)
plt.show()
```

Listing 4.19 Changing Function Parameters Interactively

Output

Figure 4.19 shows the output of the interactively changeable function parameters.

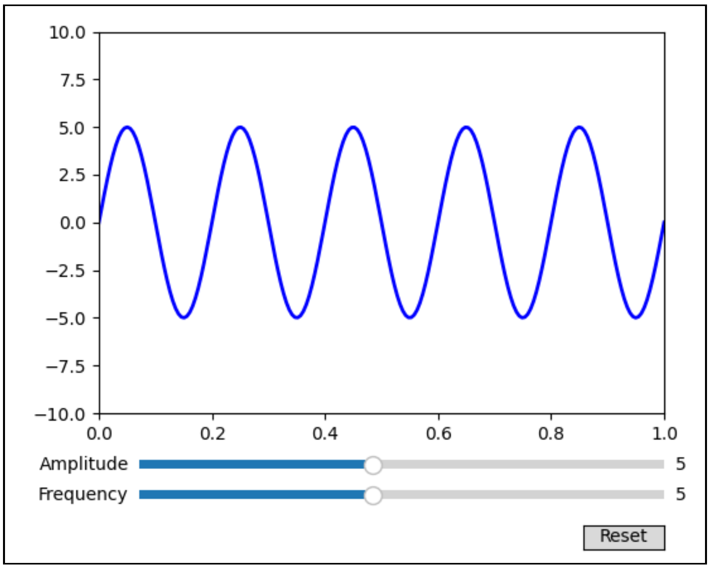


Figure 4.19 Changing Function Parameters Interactively

Analysis

In line 04, the `widgets` submodule with the `Slider` and `Button` classes is imported. Line 10 defines a sine function with the amplitude `a0` and frequency `f0` parameters. The function coordinates with the initialization values are assigned to the `curve` variable in line 11. When the program is started, a sine function with amplitude 5 and frequency 5 will be displayed.

The coordinates for the `sldAmp` slider, which is supposed to change the amplitude, are assigned to the `xyAmp` variable in line 15. For the coordinates of the `sldFreq` slider, which is supposed to change the frequency, the same applies in line 16.

The objects for the `sldAmp` and `sldFreq` sliders as well as for the `cmdReset` command button are created in lines 19 to 21. The `Slider()` constructor of the `Slider` class expects the x-y coordinates of the control as the first parameter, the second parameter determines the label, the third and the fourth parameters determine the adjustment range, the fifth parameter `valinit=5` determines the initialization value, and the sixth and final parameter determines the increment of the value change.

In lines 23 to 26, the `update(val)` function is defined. This function is responsible for assigning the values set by the sliders to the `A` and `f` variables. The `val` variable is accessed via the `sldAmp` and `sldFreq` slider objects. In line 26, the values for amplitude and frequency are updated using the `set_data()` method and prepared for display on the screen.

In lines 28 to 30, the reset function is defined. The `sldAmp` and `sldFreq` objects are reset by the built-in `reset()` method. If this function is called in line 34 by a mouse click on the **Reset** command button, then the sine function will be displayed again with its initialization values.

In lines 32 to 34, the event query is done via the built-in `on_changed(update)` method. This method is accessed with the name of the object. The `sld` and `cmd` prefixes, which do not exist in the original, were assigned to better identify the controls in the source code.

This program is a negative example of a programming style you should avoid! The arrangement of inputs (lines 08 and 09), graphic elements (line 05, lines 11 to 21) and function definitions (lines 23 to 30) does not correspond to the desirable arrangement of the program parts, which would adhere to this sequence:

1. Inputs
2. Function definitions
3. Graphic area

Proposed change: Change the order of the instructions to meet these criteria and test the program.

Usage Example: Phase Angle Control

The output voltage of a bridge rectifier circuit is changed by using a phase angle control. The voltage waveform $u = f(\alpha)$ is supposed to be simulated as a function of the control angle α. Figure 4.20 shows the output voltage waveform of such a control.

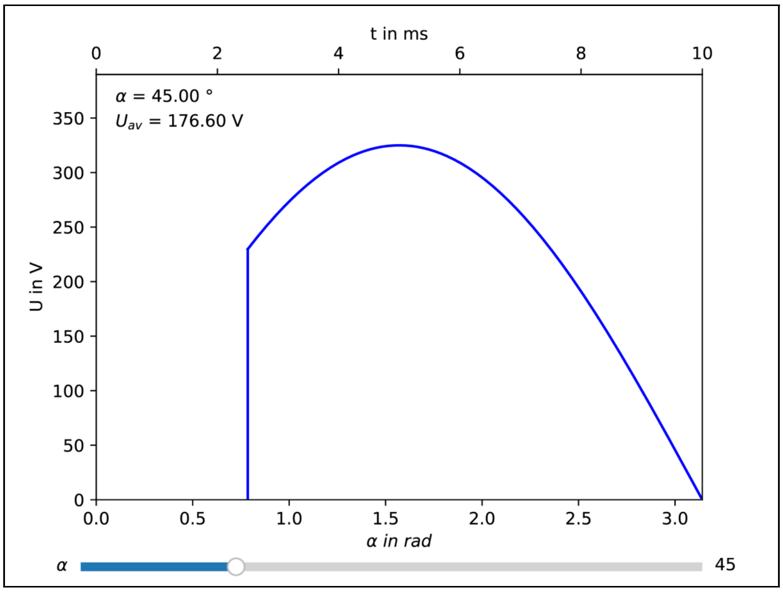


Figure 4.20 Phase Angle Control

The slider adjusts the control angle from 0 to 180°. The current value of the arithmetic mean U_{AV} of the output voltage and the control angle α should be displayed in the GUI of the program.

The following holds true for the arithmetic mean value of the output voltage:

$$U_{AV}(\alpha) = \frac{\hat{u}}{\pi}\int_{\alpha}^{\pi} \sin(x)\mathrm{d}x = \frac{\hat{u}}{\pi}(1 + \cos\alpha)$$

Listing 4.20 allows you to simulate phase angle control.

```
#20_sld_phase_angle_control.py
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.widgets import Slider
Us=325      #peak value in V
a0=np.pi/4 #initial value 45°
xmax=np.pi
#u(x), x is an angle
```

```
09  def u(x):
10      return Us*np.sin(x)
11  #query slider
12  def update(val):
13      alpha = sldAlpha.val #control angle in °
14      a=np.radians(alpha)  #control angle in rad
15      x = np.arange(a,xmax,0.01)
16      y.set_data(x,u(x))
17      line.set_data([a,a],[u(0),u(a)])
18      Uav=Us*(1.0 + np.cos(a))/np.pi
19      txtAngle.set_text(r'$\alpha$ = %.2f °' %alpha)
20      txtUav.set_text(r'$U_{av}$ = %.2f V' %Uav)
21  #Graphics area
22  fig, ax = plt.subplots()
23  txtAngle=ax.text(0.1,1.12*Us,r'$\alpha$ = %.2f °' %45)
24  txtUav=ax.text(0.1,1.05*Us,r'$U_{av}$ = 176.60 V')
25  fig.subplots_adjust(left=0.12,bottom=0.15)
26  ax.set_xlim(0,xmax)
27  ax.set_ylim(0,1.2*Us)
28  x0 = np.arange(a0,xmax,0.01) #for initial values
29  line, = ax.plot([a0,a0],[u(0),u(a0)],'b-')
30  y, = ax.plot(x0,u(x0),'b-')
31  xyAlpha = fig.add_axes([0.1, 0.02, 0.8, 0.03])
32  sldAlpha=Slider(xyAlpha,r'$\alpha$',0,180,valinit=np.degrees(a0),
valstep=1)
33  sldAlpha.on_changed(update)
34  ax.set(xlabel=r'$\alpha \  in\  rad$',ylabel='U in V')
35  secax = ax.secondary_xaxis('top',functions=(lambda x:10*x/np.pi,
lambda x:np.pi*x))
36  secax.set_xlabel('t in ms')
37  plt.show()
```

Listing 4.20 Phase Angle Control

Analysis

In line 15, the NumPy function `arange(a,xmax,0.01)` updates the range of values for control angle `a` on the x-axis. In line 16, this value is adopted by the matplotlib method, `set_data(x,u(x))`. In the range from 0 to `a`, the sine function does not display. To make the phase angle clearly visible, the `set_data([a,a],[u(0),u(a)])` method in line 17 creates a vertical line. The statement in line 18 calculates the arithmetic mean value `Uav` of the output voltage.

What is new in this case is the dynamic output of control angle `alpha` in line 19 on the drawing area (`ax` object). On line 19, the `set_text()` method outputs the updated slider setting of `alpha` at the x,y position of the axes object `ax` specified in line 23. The same rule applies to the output of the output voltage `Uav` in lines 20 and 24. A dynamic text output always consists of two parts:

1. Placement of the static text object with `text(x-position,y-position,'text')` in the drawing area
2. The dynamic output using the `set_text('y=%.2f'%x)` method

The statement in line 35 causes the corresponding time segments in ms to be assigned to control angle `a`. The new scale is displayed on the top x-axis (`'top'` property). The specifications apply to 50 Hz alternating current.

4.1.10 Contour Plots

A contour line is a plane cut through a three-dimensional graph of function *f*(x, y) parallel to the (x,y) plane. This line therefore describes the course of a three-dimensional graph in the plane. The points with the same values are connected to form a curve. In cartography, a contour plot represents the depths of valleys and the heights of mountains as contour lines. To display contour plots, you'll need the `meshgrid()` NumPy function and the `contour()` Matplotlib method.

Demonstrating the meshgrid Function

To plot contour lines via Pyplot, the `meshgrid(x,y)` NumPy function must be used to capture all relevant points in the (x,y) plane. Listing 4.21 shows how this function works.

```
#21_meshgrid_demo.py
import numpy as np
import matplotlib.pyplot as plt
x=y=np.linspace(1,6,6)
x,y=np.meshgrid(x,y)
fig, ax=plt.subplots()
plt.plot(x,y,marker='x',color='red',ls='none')
plt.show()
```

Listing 4.21 Mesh Grid

Output

How the mesh grid presents itself in the user interface (UI) is shown in Figure 4.21.

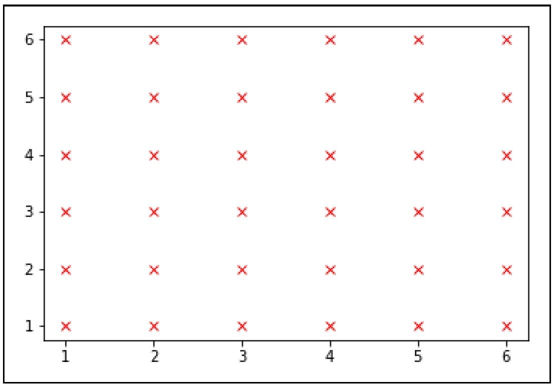


Figure 4.21 Mesh Grid for Magnetic Field Lines

Analysis

Line 04 creates an array for each of the variables x and y with the numbers from 1 to 6. In line 05, the `meshgrid(x,y)` NumPy function generates a matrix with six rows and six columns from it. In line 07, the `plot(x,y,marker='x',color='red',ls='none')` method creates a square matrix as a graph with 36 red crosses.

Now, we can insert the following statements below line 06:

```
ax.set_xticks([])
ax.set_yticks([])
ax.set_frame_on(False)
```

In this case, the x-axis and y-axis will not be labeled, and the borders (*frame*) will not be displayed. You can then use this graph to illustrate magnetic field lines.

Contour Plot for the Field Strength Curve

The next sample program shows in a contour plot how the magnetic field of a straight current-carrying conductor runs. The magnetic field strength H increases proportionally to the current strength I and decreases inversely proportionally to the distance r of the conductor:

$$H = \frac{I}{2\pi \cdot r}$$

To ensure that all points in the (x,y) plane are captured, the radiuses for each x-y coordinate must be calculated:

$$r = \sqrt{x^2 + y^2}$$

A contour plot is created using the `contour()` matplotlib method, as shown in Listing 4.22:

```
01  #22_contour_plot_circles.py
02  import numpy as np
```

```
03  import matplotlib.pyplot as plt
04  I=62.8 #current
05  rmax=10
06  n=100
07  lev=[1,2,4,8,16]
08  x=y=np.linspace(-rmax,rmax,n)
09  x,y=np.meshgrid(x,y)
10  H=I/(2*np.pi*np.hypot(x,y))
11  fig,ax=plt.subplots()
12  cp=ax.contour(x,y,H,levels=lev,colors='red')
13  ax.clabel(cp,inline=True)
14  ax.set(xlabel='x', ylabel='y')
15  ax.set_aspect('equal')
16  plt.show()
```

Listing 4.22 Contour Plot of a Magnetic Field

Output

Figure 4.22 shows how the contour plot for a magnetic field is graphically represented in the output.

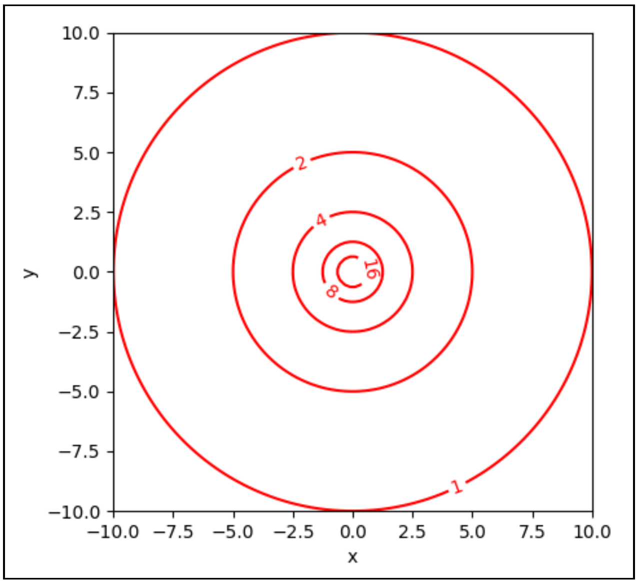


Figure 4.22 Contour Plot for a Magnetic Field

Analysis

In line 08, the `linspace()` NumPy function creates an array of 100 values in the value range ±10 for the x and y coordinates. From this array, the `np.meshdrid(x,y)` function creates a matrix of 100 rows and 100 columns in line 09.

Line 10 calculates the magnetic field strength `H`. The `hypot(x,y)` NumPy function determines the distance (hypotenuse) of the field lines from the center point for each point at location (x|y) according to the Pythagorean theorem. Since the matrix generated by `np.meshgrid(100,100)` contains 100 rows and 100 columns, a total of 10,000 values is stored in variable `H`.

In line 12, following Matplotlib method calculates the circles (contour lines) of the magnetic field lines:

```
contour(x,y,H,levels=lev,colors='red')
```

The first two parameters contain all x-y coordinate data. The data of the `H` variable for the contour plot is in the third place of the parameter list. The fourth parameter (`levels`) is assigned the `lev=[1,2,4,8,16]` list from line 07. Spreading out the spacing prevents the field lines from being too dense around the center point. All contour line data is stored in the `cp` variable so that it can be provided as the first parameter to the `clabel(cp,inline=True)` Matplotlib method in line 13. The `inline=True` parameter causes the values of the field strengths to be displayed in the graph. To keep the outer dimensions of the contour graphic the same length on screen output, the `set_aspect('equal')` method is added to the source code in line 15.

You can test the `set_aspect('equal')` method by commenting out line 16.

4.2 3D Function Plots

Up to this point, mathematical or physical relationships were visualized in the plane using the `plot(x,y)` method. However, in reality, for example, electromagnetic waves propagate in three-dimensional space, and physical bodies can only exist with a spatial extension. Thus, an option would have to be provided to add a third dimension to the x-y plane. Matplotlib provides the option to create 3D plots using the `plt.figure().add_subplot(projection='3d')` statement. Mathematical functions of the form $f(x,y,z)$ are projected onto a 2D plane. Using the mouse pointer (keep the left mouse button pressed), the screen outputs can be rotated to any position you want. A helical line and a circular ring were selected as examples.

4.2.1 Helical Line

An electron shot into a homogeneous magnetic field with velocity v_0 rotates with angular velocity ω on the path of a helical line of radius R. The spatial representation of a helix is described by the following three parameter equations:

$$x = R \cdot \cos \omega t$$
$$y = R \cdot \sin \omega t$$
$$z = v_0 \cdot t$$

The first two parameter equations describe a circular path. The third equation defines the slope of the helix in the z-direction.

Listing 4.23 shows how a helical line can be plotted using the `plot(x,y,z)` function:

```
#23_3d_helix.py
import numpy as np
import matplotlib.pyplot as plt
R=6
v0=5
omega=3
t=np.linspace(0,2*np.pi,500)
x=R*np.cos(omega*t)
y=R*np.sin(omega*t)
z=v0*t
ax = plt.figure(figsize=(6,6)).add_subplot(projection='3d')
ax.plot(x,y,z,lw=2)
ax.set(xlabel='x',ylabel='y',zlabel='z',title='Electron in the magnetic
field')
plt.show()
```

Listing 4.23 Helical Line

Output

The graphical implementation of the helical in the coordinate system is shown in Figure 4.23.

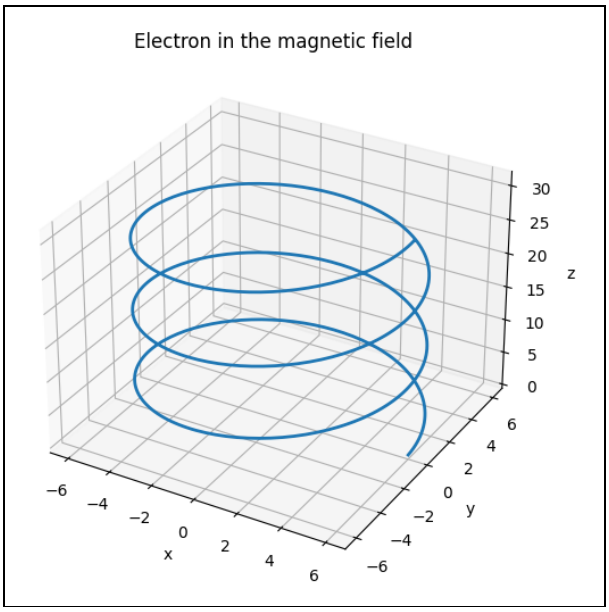


Figure 4.23 Helical Line

Analysis

In line 07, 500 values in the range from 0 to 2π are stored in the `t` variable. With these values, the `np.cos()`, `np.sin()`, and `v0*t` functions in lines 08 to 10 calculate the values for the x, y, and z coordinates. The `R` variable determines the radius of the helical line.

The `projection='3d'` parameter in line 11 ensures that, in the further course of the program, the functions and properties necessary for the 3D plot can be accessed using the `ax` object. The `plot(x,y,z, ...)` method is used to prepare the 3D plot for output in line 12 and displayed on the screen via `show()` in line 14.

4.2.2 Circular Ring

The second example shows how you can use Matplotlib to plot a circular ring (torus) in a 3D coordinate system. A torus is described by the following system of equations:

$x = (R + r \cdot \cos p) \cdot \cos t$

$y = (R + r \cdot \cos p) \cdot \sin t$

$z = r \cdot sin\, p$

R is the mean diameter, and r is the diameter of the circular cross-section of a circular ring.

Since the *surface* of a body is to be represented, the `plot_surface()` method must be used instead of `plot()`. Listing 4.24 shows how to use this function.

```python
#24_3d_torus.py
import numpy as np
import matplotlib.pyplot as plt
n=100
R=2 #mean radius
r=1 #cross-sectionradius
p=np.linspace(0,2*np.pi,n)
t=np.linspace(0,2*np.pi,n)
p,t=np.meshgrid(p,t)
#Parameter equations
x=(R+r*np.cos(p))*np.cos(t)
y=(R+r*np.cos(p))*np.sin(t)
z=r*np.sin(p)
#Draw circular ring
ax = plt.figure().add_subplot(projection='3d')
ax.plot_surface(x,y,z,rstride=5,cstride=5,color='y',edgecolors='r')
ax.set(xlabel='x',ylabel='y',zlabel='z',title='Torus')
ax.set_zlim(-3,3)
plt.show()
```

Listing 4.24 Circular Ring

Output

The circular ring generated using the `plot_surface()` method is shown in Figure 4.24.

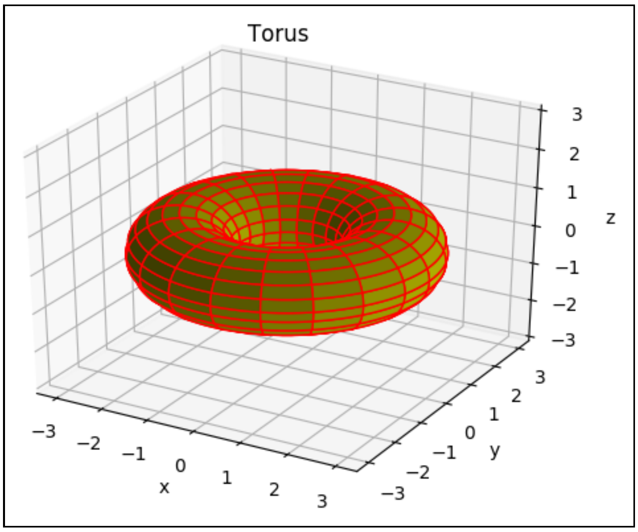


Figure 4.24 Circular Ring

Analysis

Lines 11 to 13 contain the three parameter equations. The method in line 16 is new:

```
plot_surface(x,y,z,rstride=5,cstride=5,color='y',
edgecolors='r')
```

The `rstride=5` parameter sets the increment of the horizontal lines, while `cstride=5` sets the increment of the vertical lines.

4.2.3 Combining a 3D Plot with a Contour Plot

A 3D function plot can also be combined with a contour plot. Listing 4.25 shows how the 3D plot of a paraboloid, which is described by this equation:

$z = 100 - x^2 - y^2$

This task can be implemented with the corresponding level lines.

```
#25_3d_mountain.py
import numpy as np
import matplotlib.pyplot as plt
width=10
h=100
x=y=np.linspace(-width,width,100)
x,y=np.meshgrid(x,y)
```

```
#Equation for paraboloid
z=h-x**2-y**2
#Show paraboloid
fig=plt.figure(figsize=(4.2,8))
ax1=fig.add_subplot(2,1,1,projection='3d')
ax1.plot_surface(x,y,z,rstride=5,cstride=5,color='g',edgecolors='y')
ax1.set_zlim(-h,h)
ax1.set(xlabel='x',ylabel='y',zlabel='z',title='Paraboloid')
#Level lines
ax2=fig.add_subplot(2,1,2)
hl=ax2.contour(x,y,z,levels=10,colors='b')
ax2.clabel(hl,inline=True)
ax2.set_xlim(-width,width)
ax2.set_ylim(-width,width)
ax2.set(xlabel='x',ylabel='y',title='Level lines')
ax2.set_aspect('equal')
plt.show()
```

Listing 4.25 Combination of "plot_surface" and "contour"

Output

The output of the paraboloid and level lines of the contour plot according to Listing 4.25 in Figure 1.25.

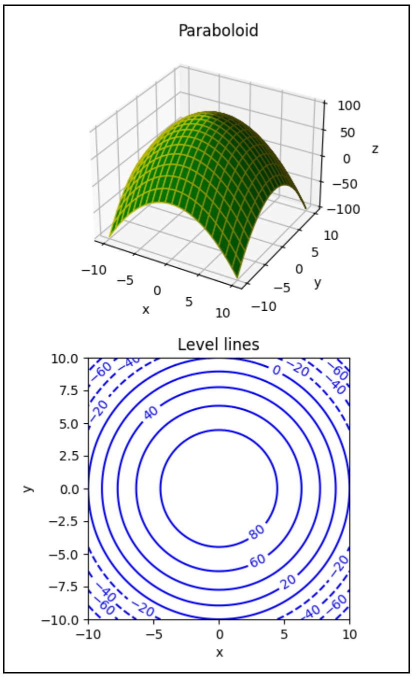


Figure 4.25 Mountain with Contour Lines

Analysis

Almost all programming elements are known from the previous examples. In line 12, the `add_subplot(2,1,1,projection='3d')` method creates the subplot for the 3D plot. In line 17, the `add_subplot(2,1,2)` method creates the subplot for the contour plot.

The `set_aspect('equal')` method in line 23 scales the x-axis and y-axis of the second subplot with equal drawing units.

You can also test the `figure()` method using the `figsize=plt.figaspect(2)` parameter (line 11). Then, the graphic will be displayed twice as high as it is wide.

4.3 Vectors

Vectors describe directed quantities in physics, such as forces or field strengths. For the representation of vectors, the Matplotlib module provides the `quiver([X,Y],U,V,[C],**kwargs)` method.

The `[X,Y]` list sets the initial coordinates of the vector. The `U` parameter determines the x-component, while the `V` parameter determines the y-component of the vector arrow. If you omit the `[X,Y]` parameter, then the start of the arrow is automatically placed at the origin of the coordinate system, and the arrowhead points to the specified u-v coordinate. The `C` parameter allows you to set the color of the vector arrow. The optional parameter `**kwargs` provides supplementary properties.

More information on this topic is available at *https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.quiver.html*.

4.3.1 Vector Addition

Listing 4.26 shows how you can add three force vectors using matplotlib method `quiver()`.

```python
#26_vector_add.py
import matplotlib.pyplot as plt
xmin, xmax=-8, 5
ymin, ymax=-6,6
F1x,F1y=-4,4
F2x,F2y=-4,-4
F3x,F3y=2,0
Fresx=F1x+F2x+F3x
Fresy=F1y+F2y+F3y
fig, ax=plt.subplots()
#vectors
ax.quiver(F1x,F1y,angles='xy',scale_units='xy',scale=1,color='m')
ax.quiver(F2x,F2y,angles='xy',scale_units='xy',scale=1,color='g')
```

```
14 ax.quiver(F3x,F3y,angles='xy',scale_units='xy',scale=1,color='b')
15 ax.quiver(Fresx,Fresy,angles='xy',
16            scale_units='xy',scale=1,color='r',label="$F_{res}$")
17 ax.axis([xmin,xmax,ymin,ymax])
18 ax.set(xlabel="$F_{x}$",title="Vector addition ")
19 ax.set_ylabel("$F_{y}$",rotation=True)
20 ax.legend(loc='best')
21 plt.show()
```

Listing 4.26 Addition of Three Force Vectors

Output

The graphical output of the vector addition is shown in Figure 1.26.

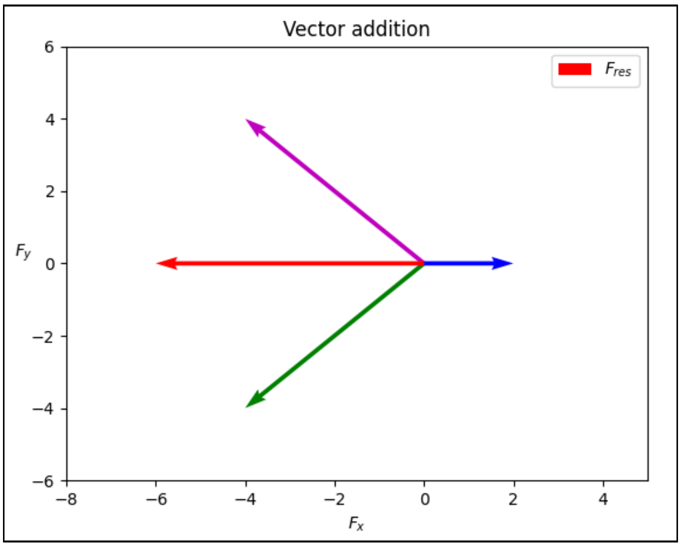


Figure 4.26 Addition of Three Force Vectors

Analysis

In lines 05 to 07, the x-y components for three force vectors are given. Line 08 calculates the total of the x-components, and line 09 calculates the total of the y-components.

The following method in lines 12 to 15 defines force vectors:

```
quiver(F1x,F1y,angles='xy',scale_units='xy',scale=1)
```

The various parameters have the following meanings:

- `F1x,F1y`: The endpoints of vector arrows.
- `angles='xy'`: The arrows point from (x,y) to ($x + u$, $y + v$). Since x and y were not used in the program (i.e., they are zero), the arrows point from the coordinate origin to (u,v).

- `scale_units='xy'`: The units of the given axis scalings are taken over.
- `scale=1`: The scaling factor is 1, which means that the given axis scaling is not changed. A smaller number increases the length of the vector arrow.

4.3.2 Vector Field

Listing 4.27 enables you to represent a vector field. The program draws 90 parallel vectors in the x-direction.

```python
#27_vector_field.py
import numpy as np
import matplotlib.pyplot as plt
n=10
x1,x2=0, 10
u=2 #length
v=0 #direction
xk=yk=np.linspace(x1,x2+u,n)
x,y=np.meshgrid(xk,yk)
fig, ax=plt.subplots()
#Define vectors
ax.quiver(x,y,u,v,units='xy',scale=2,color='blue')
#Range of the x-axis
ax.set_xlim(x1-u/2,x2+u)
plt.show()
```

Listing 4.27 Homogeneous Vector Field

Output

The graphical output of the vector field is shown in Figure 4.27.

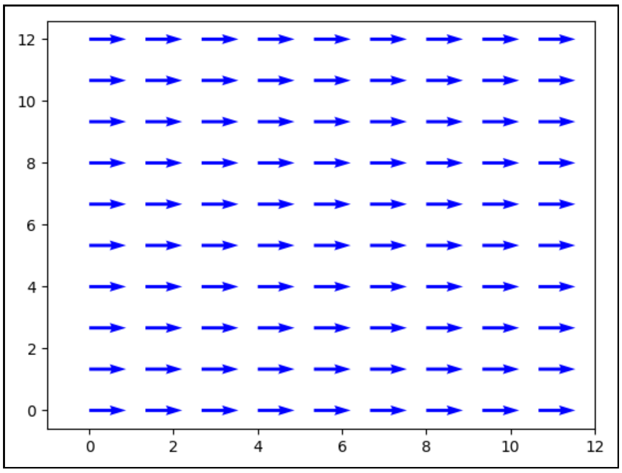


Figure 4.27 Homogeneous Vector Field

Analysis

In line 09, the `meshgrid()` NumPy function generates the matrix for the x-y coordinates of the vectors (arrow beginnings). In line 12, the following method defines the vector field:

```
quiver(x,y,u,v,units='xy',scale=2,color='blue')
```

The `x,y` parameters define the coordinates of the arrow beginnings of the individual vectors. The length of a vector is set to two length units in line 06. The `scale=2` scaling factor shortens the length of a vector by a factor of 0.5. You could have obtained the same result with `u=1` and `scale=1`.

4.4 Displaying Figures, Lines, and Arrows

The `matplotlib` module can plot geometric figures such as rectangles, circles, and triangles. These figures can then illustrate mathematical, technical, and physical relationships. I'll now demonstrate the creative options of `matplotlib` through three examples by illustrating the Pythagorean theorem: a gear representation, a pointer diagram, and a current-carrying conductor in a homogeneous magnetic field.

4.4.1 Rectangles

Rectangle objects `r` are created using the following method:

```
r=patches.Rectangle((x1,y1),b,h,fill,edgecolor,angle)
```

These objects are then inserted into the drawing area via `add_patch(r)`.

The `x1,y1` tuple defines the lower-left corner of the rectangle. The `b` and `h` parameters determine its width and height. The `fill` parameter is set to `True` by default and provides the option to fill the rectangle with a specific color: `facecolor=color`. The `edgecolor` parameter can be used to change the edge color, and the `angle` parameter allows the rectangle to be rotated by a specified angle in degrees.

Listing 4.28 shows how the `Rectangle()` function can be used to display three rectangles with predefined dimensions in a drawing area.

```
#28_pythagoras.py
import numpy as np
import matplotlib as mlt
import matplotlib.pyplot as plt
x1,x2=-3,8
y1,y2=-1,11
```

```
a,b=3,4
alpha=np.degrees(np.arctan(b/a))
beta=90-np.degrees(np.arctan(a/b))
c=np.hypot(a,b)
fig,ax=plt.subplots()
ax.axis([x1,x2,y1,y2])
#(x1,y1),width,height
ra=mlt.patches.Rectangle((0,c),a,a,fill=False,lw=2,edgecolor='b',
angle=alpha)
rb=mlt.patches.Rectangle((c,c),b,b,fill=False,lw=2,edgecolor='b',
angle=beta)
rc=mlt.patches.Rectangle((0,0),c,c,fill=False,lw=2,edgecolor='b')
ax.add_patch(ra)
ax.add_patch(rb)
ax.add_patch(rc)
ax.set_aspect('equal')
ax.set_xticks([])
ax.set_yticks([])
ax.set_frame_on(False)
plt.show()
```

Listing 4.28 Representation of Three Rectangles

Output

Figure 4.28 shows how the program from Listing 4.28 illustrates the Pythagorean theorem.

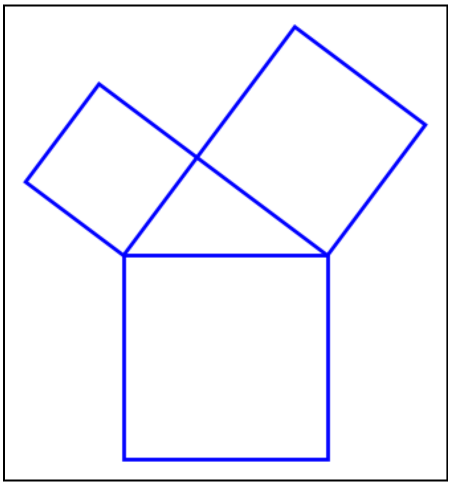


Figure 4.28 Illustration of the Pythagorean Theorem

Analysis

Lines 05 and 06, together with the `axis([x1,x2,y1,y2])` method in line 12, specify the dimensions of the drawing area. In lines 14 to 16, the three rectangle objects `ra`, `rb`, and `rc` are generated using the following method:

```
Rectangle((x1,y1),b,h,fill=False,lw=2,edgecolor='b',angle=...)
```

In this example, the `fill` parameter is set to `False`. The edges of the rectangles are drawn in blue due to `edgecolor='b'`. The rectangle in the upper left is rotated by the angle `alpha` in the mathematically positive direction (i.e., counterclockwise). The rectangle in the upper right is rotated by the angle `beta`. The calculations of the angles are performed using the `arctan()` NumPy function in lines 08 and 09.

The `add_patch()` method in lines 17 to 19 places the rectangles in the drawing area.

4.4.2 Circles and Lines

Circle objects (`circle`) can be created using the following method:

```
kreis=patches.Circle((x,y),radius,fill,lw,edgecolor)
```

These objects can then be embedded into the drawing area via the `add_patch(circle)` function. In this case, the `(x,y)` tuple stands for the coordinates of the center of a circle. The third parameter determines the radius.

Lines can be created using the `plot([x1,x2],[y1,y2])` method you already know. The known properties can be used to change the line styles and widths.

Based on the transmission with three gears example, Listing 4.29 demonstrates how circles and lines can be displayed using the `Circle()` and `plot()` methods. The simplified representation of gears as circles with the mean diameters is used whenever the structures and gear ratios of transmissions must be illustrated.

```
#29_transmission.py
import matplotlib as mlt
import matplotlib.pyplot as plt
x1,x2=-12,22
y1,y2=-17,12
fig,ax=plt.subplots()
ax.axis([x1,x2,y1,y2])
#(x,y),radius
c1=mlt.patches.Circle((-5,5),5,fill=False,lw=2,edgecolor='b')
c2=mlt.patches.Circle((-5,-5),5,fill=False,lw=2,edgecolor='b')
c3=mlt.patches.Circle((10,-5),10,fill=False,lw=2,edgecolor='b')
ax.add_patch(c1)
ax.add_patch(c2)
ax.add_patch(c3)
```

```
15  #x1,x2,y1,y2
16  ax.plot([-5,-5],[-5, 5],lw=1,color='black',ls='-.')
17  ax.plot([-5,10],[-5,-5],lw=1,color='black',ls='-.')
18  ax.plot([-5,10],[5,-5],lw=1,color='black',ls='-.')
19  ax.set_aspect('equal')
20  ax.set_xticks([])
21  ax.set_yticks([])
22  ax.set_frame_on(False)
23  plt.show()
```

Listing 4.29 Circles with a Triangle

Output

As a result of Listing 4.29, three circles and a triangle are output in the drawing area as a technology schema of a transmission, as shown in Figure 1.29.

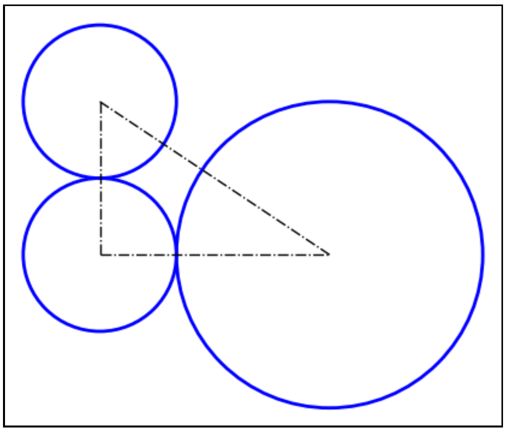


Figure 4.29 Technology Schema of a Transmission

Analysis

The lines of the triangle illustrate the distances between the individual gears. The three circle objects (c1, c2, and c3) are created in lines 09 to 11 using the following method:

```
mlt.patches.Circle((x,y),radius,fill=False,lw=2,edgecolor='b')
```

These lines are embedded in the drawing area in lines 12 to 14 by using the add_patch() method.

To change the diameters of the circles for further program tests, then a useful approach is to comment out the statements in lines 20 to 22, while inserting the ax.grid() statement below line 22. In this way, you can better check coordinate changes.

The circle.center=(x,y) statement allows you to move the circle object to the desired x,y position. This statement is used in Listing 4.35 for the animation of circular objects.

4.4.3 Arrows

Pointer diagrams are needed in AC theory to illustrate the phase shift between voltage and current. Listing 4.30 demonstrates how the `arrow()` method can generate a pointer diagram of arrows for a series circuit consisting of an ohmic resistor and an inductor.

```
#30_pointer_diagram.py
import matplotlib.pyplot as plt
x1,x2=0,12
y1,y2=0,8
lb=2   #line width
pb=0.5 #arrow width
pl=1   #arrow length
U_R=10 #ohmic voltage drop
U_L=5  #inductive voltage drop
I=12   #current
fig,ax=plt.subplots()
ax.axis([x1,x2,y1,y2])
#Arrows: x,y,x+dx,y+dy
ax.arrow(0,1.8,I,0,color='r',lw=lb,length_includes_head=True,
         head_width=pb,head_length=pl)
ax.arrow(0,2,U_R,0,color='b',lw=lb,length_includes_head=True,
         head_width=pb,head_length=pl)
ax.arrow(U_R,2,0,U_L,color='b',lw=lb,length_includes_head=True,
         head_width=pb,head_length=pl)
ax.arrow(0,2,U_R,U_L,color='b',lw=lb,length_includes_head=True,
         head_width=pb,head_length=pl)
#Labels
ax.annotate("$I$",xy=(5,1),xytext=(5,1),fontsize=12)
ax.annotate("$U_g$",xy=(5,5),xytext=(5,5.5),fontsize=12)
ax.annotate("$U_L$",xy=(10.5,4),xytext=(10.5,4),fontsize=12)
ax.annotate("$U_R$",xy=(5,3),xytext=(5,2.5),fontsize=12)
ax.set_xticks([])
ax.set_yticks([])
ax.set_frame_on(False)
ax.set_aspect('equal')
plt.show()
```

Listing 4.30 Pointer Diagram

Output

Figure 4.30 shows see the output pointer diagram.

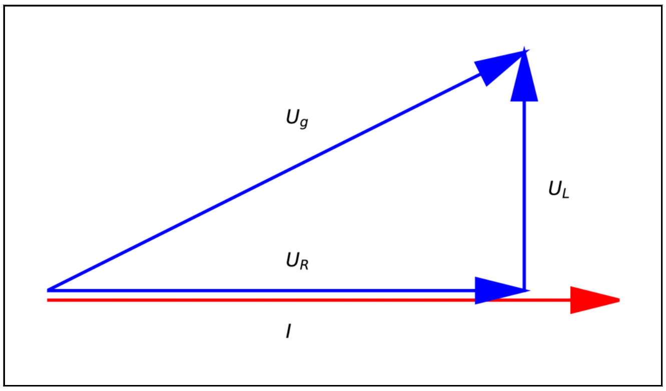


Figure 4.30 Pointer Diagram for R-L Series Circuit

Analysis

Arrows can be represented using the following method:

```
arrow(x,y,x+dx,y+dy,color,lw,length_includes_head=True, head_width,head_length)
```

The `arrow()` method is accessed via the `ax` object created in line 11. The first two parameters (`x` and `y`) define the coordinates of the start of the arrow. The `dx` and `dy` parameters determine the direction and length of the arrow. The `head_width` and `head_length` properties set the width and length of the arrowhead. An especially important point is that the `length_includes_head` property must be set to `True` so that a closed pointer triangle is displayed when rendering.

4.4.4 Polygons

A polygon is a plane geometric figure formed by a traverse line. A polygon can be created using the `Polygon(xy)` constructor of the following class:

```
class matplotlib.patches.Polygon(xy, closed=True, **kwargs)
```

The first parameter (xy) is an array containing the coordinates of the vertices of a polygon. Listing 4.31 enables you to draw polygons with any number of corners.

```
#31_polygon.py
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon
r=10
n=6
R=1.1*r
fig,ax=plt.subplots()
ax.axis([-R,R,-R,R])
```

```
10 for k in range(n):
11     w=2*np.pi/n
12     x1,y1=r*np.cos(k*w),r*np.sin(k*w)
13     x2,y2=r*np.cos((k+1)*w),r*np.sin((k+1)*w)
14     ax.plot([0,x2],[0,y2],lw=1,color='b')
15     p=Polygon([[x1,y1],[x2,y2]],fill=False,lw=2)
16     ax.add_patch(p)
17 ax.set_aspect('equal')
18 ax.grid(True)
19 plt.show()
```

Listing 4.31 Drawing Polygons

Output

Figure 4.31 shows an example of a polygon drawn using the `Polygon` method.

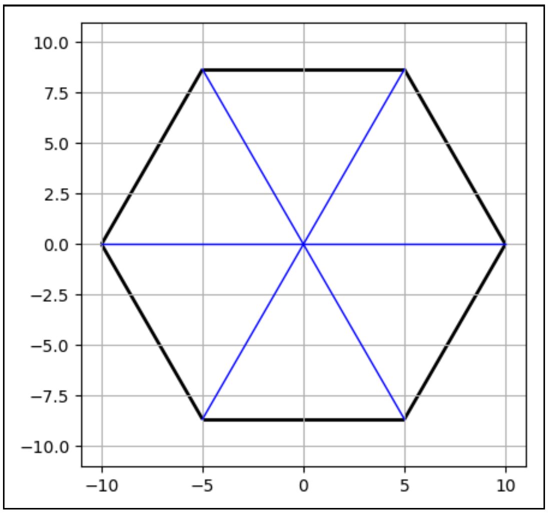


Figure 4.31 Hexagon

Analysis

Line 04 imports the `patches` submodule with the `Polygon` class. In line 05, you can define the radius `r` of the perimeter, and in line 06, the number of corners `n` of the polygon.

The most important program actions take place within the `for` loop (line 10 to 16). Line 11 calculates the angle `w` of a circle sector (circle section). In lines 12 and 13, the coordinates of the corner points are calculated. The `plot()` method in line 14 marks the boundaries of the circle sectors. In line 15, the `Polygon([[x1,y1],[x2,y2]], ...)` constructor of the `Polygon` class creates the `p` object. The `add_patch(p)` method in line 16 adds the `p` object to the drawing area.

You could also have represented a regular polygon much more easily, without a loop construct, using the following statement:

```
p=RegularPolygon((x,y),n,radius=10,fill=False)
ax.add_patch(p)
```

The `(x,y)` tuple specifies the center of the polynomial. The `n` and `radius` parameters define the number of corners and the radius of the polygon. However, the effort of the algorithm for the calculation of the x-y coordinates, which appears complicated at first sight, is justified because it is absolutely necessary in many applications, for example, for the representation of pointers in the complex plane.

4.4.5 Usage Example: A Metal Rod in a Magnetic Field

Matplotlib also allows you to create drawings that illustrate physical relationships. Listing 4.32 demonstrates how to plot a homogeneous magnetic field with a current-carrying conductor resting on two bus bars.

```
#32_mag_field.py
import numpy as np
import matplotlib as mlt
import matplotlib.pyplot as plt
x1,x2=0,12
y1,y2=0,7
x=np.linspace(1,9,9)
y=np.linspace(1,6,6)
x,y=np.meshgrid(x,y)
fig,ax=plt.subplots()
ax.axis([x1,x2,y1,y2])
rod=mlt.patches.Rectangle((1.4,0.25),0.2,6.5,color='black')#width,height
circle=mlt.patches.Circle((10,3.5),0.8,fill=False,lw=2,edgecolor='black')
ax.add_patch(circle)
ax.add_patch(rod)
ax.plot([1,10],[6.5,6.5],lw=2,color='black') #upper line
ax.plot([1,10],[0.5,0.5],lw=2,color='black') #bottom line
ax.plot([10,10],[0.5,6.5],lw=2,color='black') #right line
ax.plot(x,y,marker='x',color='red',ls='none') #magnetic field lines
ax.arrow(1.6,3.5,1,0,color='k',lw=2,head_width=0.15)#x,y,x+dx,y+dy
ax.arrow(11,6,0,-4.5,color='b',lw=2,head_width=0.16,head_length=0.5)
ax.annotate("v",xy=(3,3),xytext=(3,3.4),fontsize=12) #labels
ax.annotate("$U_q$",xy=(11.2,3),xytext=(11.3,3.2),fontsize=12)
ax.set_xticks([])#no axis labels
ax.set_yticks([])#no axis labels
ax.set_frame_on(False)
ax.set_aspect('equal')
plt.show()
```

Listing 4.32 Current-Carrying Conductor in a Magnetic Field

Output

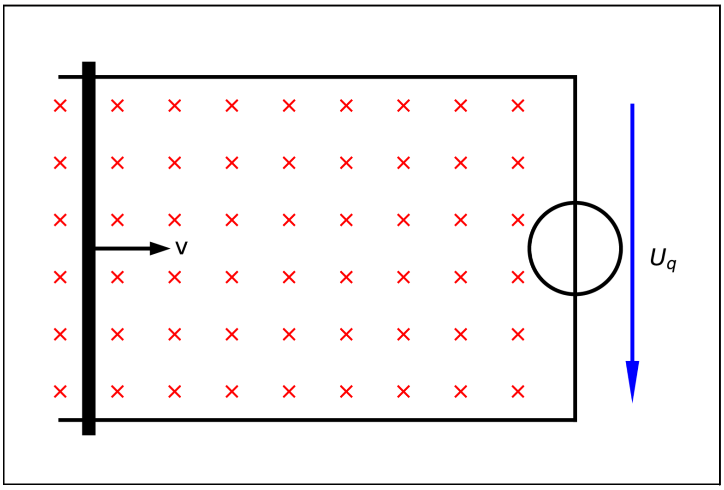


Figure 4.32 Current-Carrying Conductor in a Magnetic Field

Analysis

Figure 4.32 shows the top view of a homogeneous magnetic field with two parallel bus bars, a rod of conducting material, and a voltage source. The magnetic field lines are represented by red crosses, which should mean that the magnetic field lines are at right angles to the drawing plane and point in the direction of the drawing plane according to convention.

Almost all display elements are known from earlier examples.

4.5 Animations

Computer animation is a process in which a moving image is created from a sequence of frames. An algorithm continuously changes the positions of the individual images. In the process, each frame must be deleted before it is then moved to a new position and displayed there. If the algorithm generates 24 new images in 1 second, for example, then the viewer is given the illusion of an almost fluid movement. Computer animations can be used to illustrate physical phenomena that are beyond human perception by slowing down fast processes or speeding up slow processes.

The `from matplotlib.animation import FuncAnimation` statement imports the `FuncAnimation` method.

The following method creates the `ani` object:

```
ani=FuncAnimation(fig, func, frames=None, init_func=None, fargs=None,
save_count=None, cache_frame_data=True)
```

Not all possible parameters are specified. Although this object is not needed in the animation program, it must be created; otherwise, the animation will not be executed, and only a static image will appear on the screen.

Note

You must store an animation in a variable, which means you should always create an explicit object. If you don't, an implicitly created animation object will be subjected to an automatic *garbage collection* process, and the animation will be stopped.

You can create an implicit object by not assigning a variable to a method. The following console example creates an implicit object:

```
>>> import matplotlib.pyplot as plt
>>> from matplotlib.animation import FuncAnimation
>>> fig=plt.Figure()
>>> def func():pass
>>> FuncAnimation(fig,func) #no variable present
<matplotlib.animation.FuncAnimation object at 0x135e65c30>
```

The warning generated by this console program was not included in this case.

The second parameter (`func`) stands for the name of a custom Python function that is to be animated. This function is called without specifying a parameter. The other parameters will be discussed during the analysis of each program.

The examples selected for this purpose include the time-based shifting of a sine wave on the x-axis, the oblique throw, and the motion of a planet in an elliptical orbit.

4.5.1 A Simple Animation: Shifting a Sine Function

Listing 4.33 shows how to use the `FuncAnimation()` method to shift a sine wave in the direction of the x-axis.

```
01  #33_animation_sine.py
02  import numpy as np
03  import matplotlib.pyplot as plt
04  from matplotlib.animation import FuncAnimation
05
06  def f(x,k):
07      return np.sin(x-k/20)
08
```

```
09  def v(k):
10      y.set_data(x,f(x,k))
11      return y,
12
13  fig,ax=plt.subplots()
14  x=np.linspace(0,4*np.pi,200)
15  y, = ax.plot(x,f(x,0),'r-',lw=3)
16  #Animation
17  ani=FuncAnimation(fig,v,
18                      interval=20,
19                      #frames=200,
20                      blit=True,
21                      # save_count=50,
22                      # cache_frame_data=False
23                      )
24  plt.show()
```

Listing 4.33 Shifted Sine Wave in Positive x-Direction

Output

Figure 4.33 shows a snapshot of the animation.

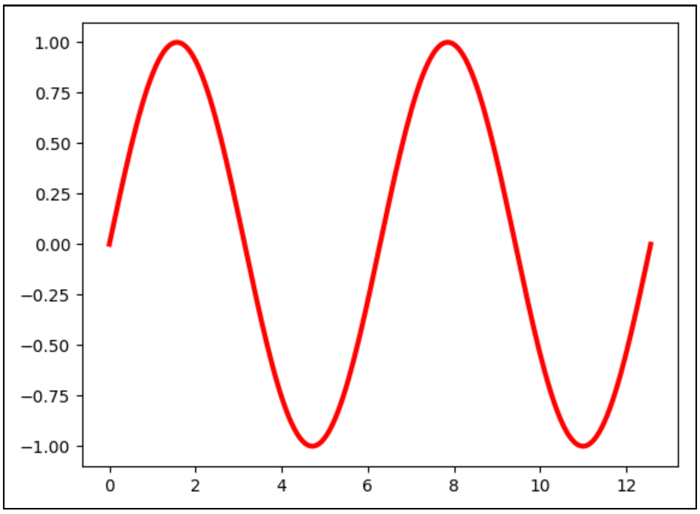


Figure 4.33 Snapshot of a Sine Wave

Analysis

This program draws a sine curve that is moving on the screen in the x-direction. The direction of the movement can be changed by the sign in line 07. If the k parameter is negative, the curve moves from left to right. If the k parameter is positive, it moves from right to left.

The program consists of three parts: the `v(k)` function in line 09, the initialization part in line 15, and the animation in line 17.

To perform an animation, the `matplotlib.animation` module must be imported (line 4).

In lines 06 and 07, the sine function `sin(x-k/20)` with variables `x` and `k` is defined. The `x` variable changes the angle, while the `k` variable causes the movement of the sine function on the x-axis.

In line 09, the most important function for the animation is defined, namely, `v(k)`. The `y.set_data(x,f(x,k))` method in line 10 changes the value `k` for the shift on the x-axis at each function call of `v(k)` in line 17. The `y` object returned in line 11 must be terminated with a comma because the return value must be a tuple. If you omit the comma, an error message will display.

Line 13 creates the `fig` and `ax` objects. The `fig` object is needed for the animation in line 17. The `ax` object is used to access the `plot` method.

In line 14, the `linspace(0,4*np.pi,200)` NumPy function stores 200 values in the `x` variable for the range from 0 to 4π. When the `y` object is initialized in line 15, the 200 values for the `x` angles and for `k=0` are stored in this object. Thus, the `y` object contains a static image for two sine waves. The `y` object must be followed by a comma again, otherwise the animation will not be executed.

In line 17, the following method performs the animation:

```
ani=FuncAnimation(fig,v,interval=20,blit=True)
```

Notice that an explicit `ani` object must be created (line 17), although it is not used in the program. The identifier of this object is freely selectable. If you do not create an explicit object, then the animation will not be executed. This object has the task of controlling an internal counter (*timer*) that accesses the explicitly created animation object `ani`. If this is missing, then the implicit animation object will be collected by the automatic memory management functionality (*garbage collection*) as data garbage, and the animation will be stopped. A static image will appear on the monitor.

The first parameter (`fig`) sets the properties of the drawing area in which the animation will take place.

As the second parameter, the `FuncAnimation()` method expects the custom animation function `v(k)`, which must be called without the `k` argument.

The `interval` parameter determines the delay (in milliseconds) with which the individual images are to be generated. The standard value is 200 ms. The larger this value, the greater the pauses between the generation of new images. The animation no longer runs as "smoothly" and shows clear signs of "bucking."

The `frames=200` parameter sets the number of *frames* to be drawn. This parameter is not needed in this animation and will be explained in more detail in later examples.

The `blit` parameter specifies whether blitting should be used to optimize dynamic drawing. The default value is `False`. *Blitting* means the fast copying and moving of the object to be moved. If `blit=True`, only the areas of the image that have changed will be redrawn. The animations should run more or less "smoothly" due to blitting. If you comment out this parameter, however, you'll notice that hardly any change is perceptible. You can learn more about blitting at *https://matplotlib.org/stable/tutorials/advanced/blitting.html*.

The `save_count` parameter sets the number of *frames* to be stored in the cache. This parameter is used only if no value is assigned to the `frames` parameter.

The `cache_frame_data` parameter prevents a memory overflow from occurring in the cache. The default value is `True`. If the `frames` parameter is not assigned a value, you should set `cache_frame_data=False` as of Matplotlib version 3.7. Otherwise, the Python interpreter will issue the following warning:

```
UserWarning: frames=None which we can infer the length of, did not pass an
explicit *save_count* and passed cache_frame_data=True. To avoid a possibly
unbounded cache, frame data caching has been disabled. To suppress this warning
either pass `cache_frame_data=False` or `save_count=MAX_FRAMES`.
ani=FuncAnimation(fig,v,
```

You should test this program extensively by changing the individual parameters and closely observing the effects on the animation.

4.5.2 Animated Oblique Throw

The animated oblique throw shows the motion sequence of a ball thrown with a certain initial velocity at a certain throwing angle. The parameter equations can be taken from Listing 4.17. For the determination of the drawing area, the throw distance and the climb height must be calculated.

$$x_{\max} = \frac{v_0^2 \cdot \sin 2\alpha}{g}$$

$$y_{\max} = \frac{v_0^2 \cdot \sin^2 2\alpha}{2g}$$

Listing 4.34 enables you to animate the motion sequence of the oblique throw.

```
#34_animation_throw.py
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
g=9.81
v0=10
throwing_angle=45
```

```
alpha=np.radians(throwing_angle)
tmax=2*v0*np.sin(alpha)/g
xmax=v0**2*np.sin(2*alpha)/g
ymax=v0**2*np.sin(alpha)**2/(2*g)
#Calculate trajectory
def throw(t):
    x = v0*np.cos(alpha)*t
    y = v0*np.sin(alpha)*t-0.5*g*t**2
    ball.set_data([x],[y])
    return ball,
#generate objects
fig,ax=plt.subplots()
ax.axis([0,xmax+0.5,0,ymax+0.5])
ball, = ax.plot([],[],'ro')
t=np.linspace(0,tmax,100)
ani=FuncAnimation(fig,throw,frames=t,interval=20,blit=True)
ax.set(xlabel="x in m",ylabel="y in m",title="Oblique throw")
plt.show()
```

Listing 4.34 Animated Oblique Throw

Output

Figure 4.34 shows a snapshot of the animation of the oblique throw.

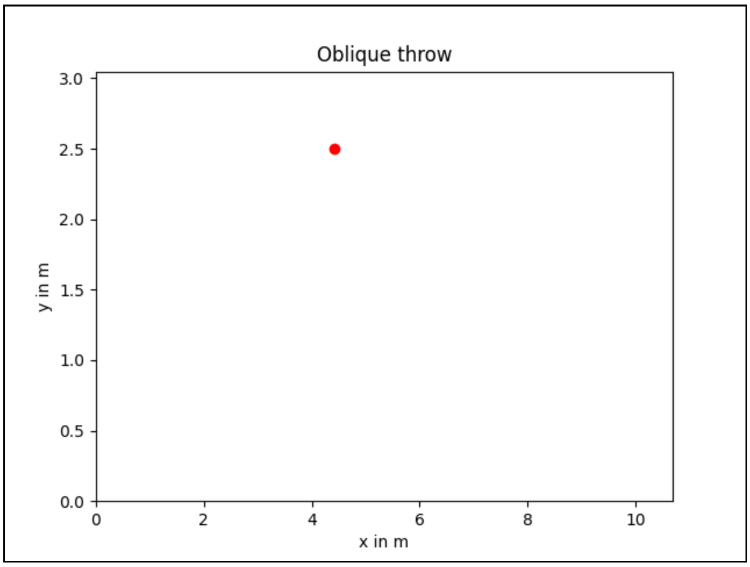


Figure 4.34 Snapshot of the Animation of the Oblique Throw

Analysis

Of course, the animated throwing process cannot be shown in this book. For testing purposes, you can run the program with different initial velocities in line 06 and throwing angles in line 07.

The custom `throw(t)` function in lines 13 to 17 calculates the new positions of the x and y coordinates for each new function call by the `FuncAnimation()` method and stores them in the `ball` object. As of Matplotlib version 3.7, the `x` and `y` arguments must be enclosed in square brackets (line 16) because the `set_data()` method only accepts sequences as arguments. If you omit the brackets, the following warning appears: `MatplotlibDeprecationWarning: Setting data with a non sequence type is deprecated since 3.7 and will be remove two minor releases later.`

Line 21 initializes the `ball` object with an empty list for the `plot` method. The `ro` parameter causes the ball to be displayed as a red dot. If you insert the `markersize='15'` parameter, the diameter of the ball will increase.

The `FuncAnimation()` method in line 23 executes the animation. The `frames` parameter sets the number of frames to be displayed per second if the parameter is assigned an integer. In this animation, `frames` is assigned a sequence `t` from `0` to `tmax` out of 100 values, which guarantees an almost smooth display. If you were to assign an integer to the `frames` parameter, the ball would flash at a fixed position on the trajectory and not move.

4.5.3 Animated Planetary Orbit

Planets move in an elliptical orbit. The shape of an ellipse is described by the two axes a and b. The x-y components are described by the following parameter equations:

$x = a \cdot \cos t$

$y = b \cdot \sin t$

Listing 4.35 enables you to animate the motion sequence of a planet around a star.

```
01  #35_animation_elipse.py
02  import numpy as np
03  import matplotlib as mlt
04  import matplotlib.pyplot as plt
05  from matplotlib.animation import FuncAnimation
06  #Data
07  r1,r2=0.5,0.25
08  a,b=8,4 #Ellipse axes
09  width=10
10  #Initialization
11  def init():
12      planet.center=(1,2)
```

```
    ax.add_patch(planet)
    return planet,
#Trajectory calculation
def trajectory(t):
    x,y=a*np.cos(np.radians(t)),b*np.sin(np.radians(t))
    planet.center=(x,y)
    return planet,
#Graphics area
fig,ax=plt.subplots()
ax.axis([-width,width,-width,width])
planet= mlt.patches.Circle((0,0),radius=r2, color='blue')
star= mlt.patches.Circle((2.5,0),radius=r1, color='red')
ax.add_artist(star)
ani=FuncAnimation(fig,trajectory,
      init_func=init,frames=360,interval=20,blit=True)
ax.set_aspect('equal')
ax.set(xlabel='x',ylabel='y',title='elliptical orbit')
plt.show()
```

Listing 4.35 Animated Planetary Orbit

Output

Figure 4.35 shows a snapshot of the animation of a planetary orbit.

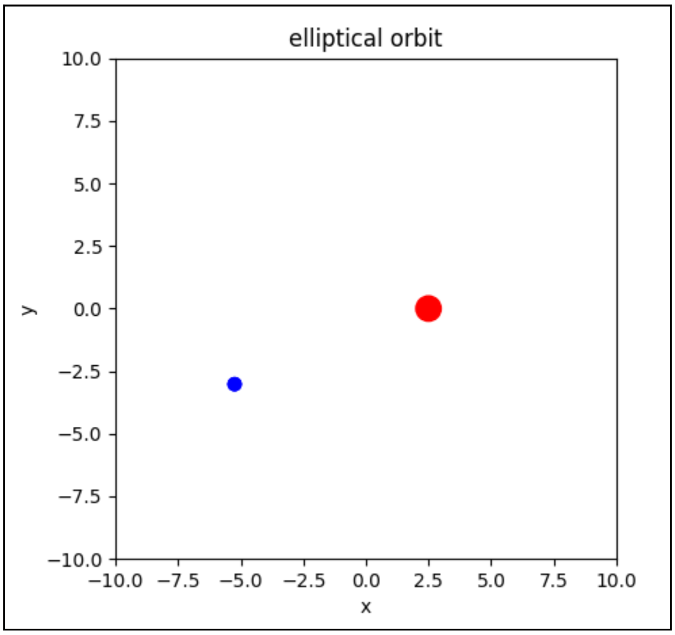


Figure 4.35 Snapshot of the Animation of a Planetary Orbit

Analysis

Basically, the program has the same structure as the animation of the throwing parabola. Line 08 defines the `a` and `b` axes of the elliptical orbit.

In lines 11 to 14, the `init()` function initializes the circular object `planet` created in line 23 for the $x = 1$ und $y = 2$ values (line 12) using the `planet.center=(1,2)` statement. These values are chosen arbitrarily. When the program starts, this placement cannot be perceived.

The custom `trajectory(t)` function in lines 16 to 19 contains the parameter equations of the ellipse. The coordinate data is calculated in line 17. The `planet.center=(x,y)` statement takes the coordinates of the orbit and stores them in the `planet` object.

The `FuncAnimation()` method in line 26 calls the `trajectory` function and the `init` function without parameters and creates the animation. In line 24, the `star` object is created, and in line 25, this object is added to the center of the drawing area using the `add_artist(stern)` method. Determining the number of `frames` is important. For example, you can test the program with `frames=300`. Then, after 300°, you'll observe the jumping movement of the planet.

4.6 Project Task: Stirling Cycle

For a Stirling engine, the cycle can be represented in a *p*-*V* state diagram. First, the cycle is visualized statically using Matplotlib. The individual states are identified by dots and numbers. Based on this information, a simulation program with slider controls for the temperature and volume change of the cyclic process will be developed. This program should calculate the amount of volume work output and the efficiency of the Stirling engine. The results should be displayed simultaneously with each slider change in the user interface of the program.

The Stirling engine is a hot-air engine, invented in 1816 by the Scottish clergyman Robert Stirling (1790–1878). Figure 4.36 shows a cylinder with a piston that moves freely up and down. The model is highly simplified and is only intended to illustrate the basic operation of the Stirling engine. We won't go into technical details in this example. Heat energy is supplied to the gas of the cylinder from a heat reservoir. Subsequently, the thermal energy is extracted from the gas again by shifting the cold reservoir to the left.

The cylinder is alternately heated and cooled. When heated, the gas in the cylinder expands, and the piston moves upward. Thus, some mechanical work (volume work) is performed. As the cylinder cools, the volume of the gas decreases, and the piston moves back down. As a rule, air is used as the gas. If the air is not compressed too much, it behaves like an ideal gas, and general gas laws can be applied.

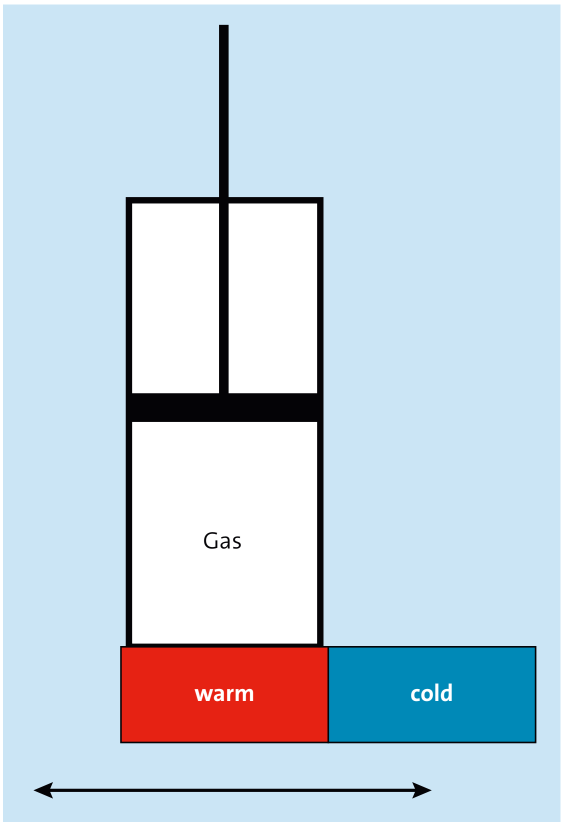


Figure 4.36 Basic Operation of a Stirling Engine

The pressure p in the cylinder is proportional to the absolute temperature T and inversely proportional to the volume V of the gas (according to *Boyle-Mariotte law*):

$$p = \frac{n \cdot R \cdot T}{V}$$

The formula symbol n is the amount of substance of the gas in mols. The constant R is the general gas constant for ideal gases ($R = 8.314462618153 24\ \mathrm{J} \cdot \mathrm{mol}^{-1} \cdot K^{-1}$).

For the volume work done on the piston, the equation following applies:

$$\mathrm{d}W = p \cdot \mathrm{d}V$$

By integration, you'll obtain the following result:

$$W = n \cdot R \cdot T \int_{V_1}^{V_2} \frac{1}{V} dV = n \cdot R \cdot T \cdot \ln \frac{V_2}{V_1}$$

When heating with the temperature T_w, the volume work defined by the following equation is performed:

$$W_{12} = n \cdot R \cdot T_w \cdot \ln\frac{V_2}{V_1}$$

When cooling to temperature T_k, the following applies:

$$W_{34} = n \cdot R \cdot T_k \cdot \ln\frac{V_2}{V_1}$$

The index w stands for the warm state, and the index k stands for the cold state. The indexes for work W_{12} and W_{34} are shown in Figure 4.37.

The utilizable mechanical work ΔW is calculated from the difference:

$$\Delta W = W_{12} - W_{34} = n \cdot R \cdot (T_w - T_k) \cdot \ln\frac{V_2}{V_1}$$

The usual convention in thermodynamics that the work done (W_{12}) is given a negative sign is not followed in this case, for once. Just think of the volume of work as amounts.

For the efficiency level, the following applies:

$$\eta = \frac{W_{12} - W_{34}}{W_{12}} = 1 - \frac{T_w}{T_k}$$

Representing the Stirling Cycle

Figure 4.37 shows the individual process states 1 to 4 for the pressures and volumes of the expanded and compressed air of the Stirling cycle was created using Listing 4.36.

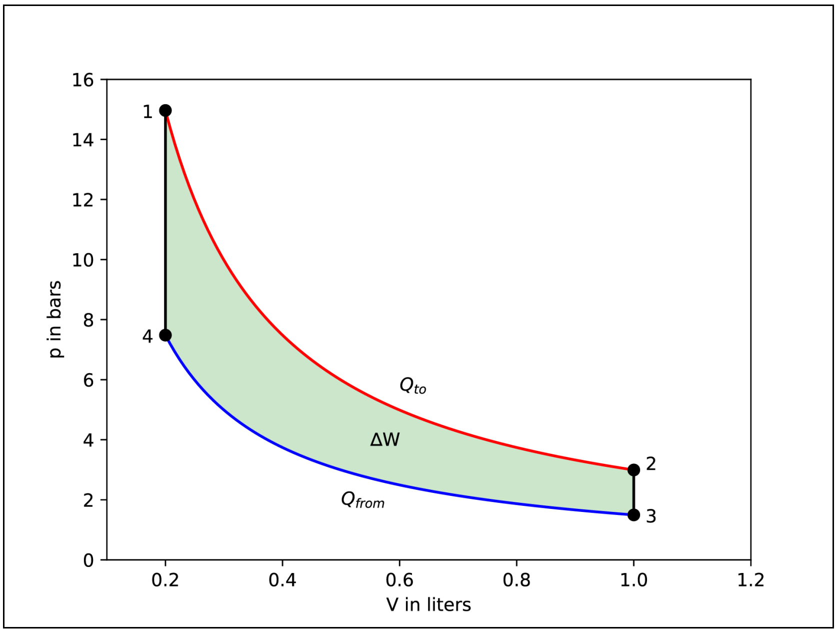


Figure 4.37 Stirling Cycle in p-V State Diagram

The dots symbolize the respective states for the pressures, volumes, and temperatures. The transitions from one state to another state are referred to as a *process*. The individual process steps run as follows:

- **1 to 2**: The air is heated by supplying thermal energy at a constant temperature T_w. It expands, so the air volume increases, and the piston moves upward. Because of $p \cdot V = const.$ the pressure decreases (isothermal change of state).
- **2 to 3**: The air is cooled down. The volume does not change (isochoric change of state).
- **3 to 4**: The volume decreases from V_2 to V_1 with the low temperature T_k being kept constant, and the pressure increases because $p \cdot V = const.$ (isothermal change of state).
- **4 to 1**: The temperature of the air is increased from the low temperature T_k to a higher temperature T_w by supplying thermal energy. The volume remains constant (isochoric change of state).

Listing 4.36 can generate the state diagram shown in Figure 4.37. The specification of the amount of substance in line 09 of n = 0.045 mol corresponds to the volume of one liter of air under normal conditions (T = 273.15 K, p = 1 bar).

```
01 #36_plot_cycle.py
02 import numpy as np
03 import matplotlib.pyplot as plt
04 Tw, Tk= 800, 400 #K
05 V1,V2=0.2,1 #dm^3
06 #Function definition p=f(V)
07 def p(V,T):
08     R=8.314   #J/(mol*K)
09     n=0.045   #mol
10     return 1e-2*n*R*T/V
11 #Graphics area
12 fig, ax = plt.subplots()
13 V=np.linspace(V1,V2,100)
14 ax.plot(V,p(V,Tw),'r-') #warm
15 ax.plot(V,p(V,Tk),'b-') #cold
16 #Dots
17 ax.plot([V1,V1],[p(V1,Tw),p(V1,Tk)],'ko')
18 ax.plot([V2,V2],[p(V2,Tw),p(V2,Tk)],'ko')
19 #vertical lines
20 ax.plot([V1,V1],[p(V1,Tk),p(V1,Tw)],'k-')
21 ax.plot([V2,V2],[p(V2,Tk),p(V2,Tw)],'k-')
22 ax.set_xlim(0.1,1.2)
```

```
23 ax.set_ylim(0,16)
24 #x,y labels
25 ax.text(V1-0.04,p(V1,Tw)-0.25,'1')
26 ax.text(V2+0.02,p(V2,Tw),'2')
27 ax.text(V2+0.02,p(V2,Tk)-0.25,'3')
28 ax.text(V1-0.04,p(V1,Tk)-0.25,'4')
29 ax.text(0.6,5.6,r'$Q_{to}$')
30 ax.text(0.5,1.8,r'$Q_{from}$')
31 ax.text(0.55,3.8,'ΔW')
32 ax.set(xlabel='V in liters',ylabel='p in bars')
33 ax.fill_between(V,p(V,Tw),p(V,Tk),alpha=0.2,color='green')
34 plt.show()
```

Listing 4.36 Generating a p-V State Diagram

Analysis

The temperatures must be specified in kelvin (line 4). The factor in line 10 `1e-2` causes the pressure to be converted from Pa to bar.

In lines 14 and 15, the `plot` method calculates the coordinate data for the warm and cold temperatures. The `r-` and `b-` parameters create a red and blue solid line, respectively.

The `ko` parameter in lines 17 and 18 draws four black dots. In lines 20 and 21, the `plot` method specifies the coordinate data for the volume boundary between volumes `V1` and `V2`. Two vertical lines are drawn.

In lines 25 to 28, the `text(x,y,'number')` method marks the numberings for the individual process states from 1 to 4.

Simulating the Stirling Cycle

Figure 4.38 shows the Matplotlib program interface with four slider controls. This program enables you to simulate the effects of the individual changes of state on the delivered volume work and the efficiency. The delivered volume work and the efficiency are displayed in the upper-right corner of the program interface.

If you change the slider markings (circles) of volumes V_1 and V_2, you'll notice that only the work done by the system changes, whereas the efficiency remains constant. By changing the temperatures, you can influence efficiency. In an upward direction, the temperature is limited by the material properties of the system. Generating lower temperatures below 400 K no longer makes physical sense because the energy required for this would no longer be justifiable. In real life, the Stirling engine achieves efficiencies of around 20% to 30%.

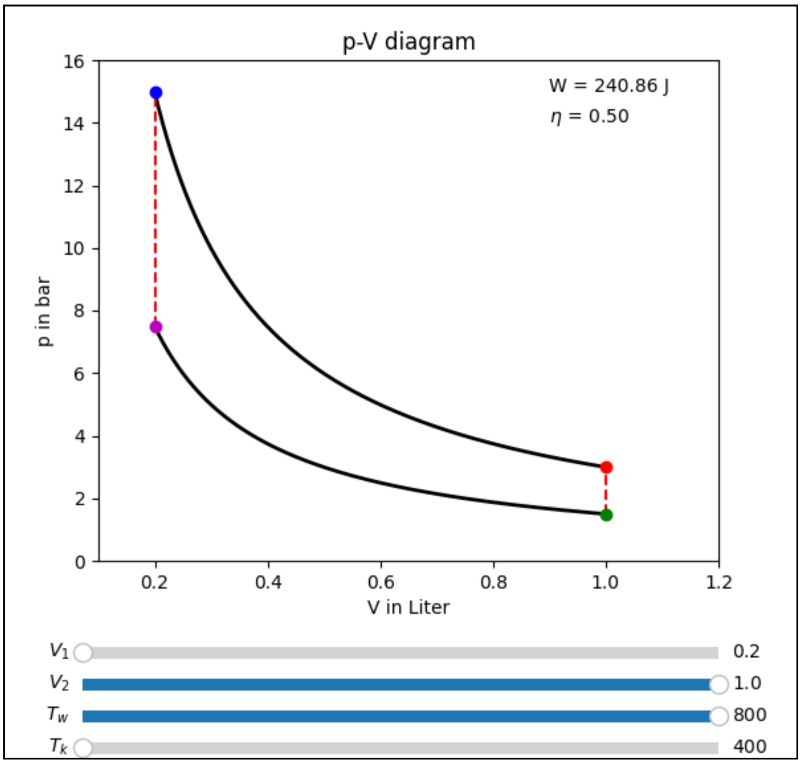


Figure 4.38 Simulating the Stirling Cycle

Listing 4.37 creates the program's user interface from Figure 4.38. Again, an air volume of one liter is assumed, which corresponds to a substance quantity of n = 0.045 mol (line 05). The gas constant `R` and the amount of substance `n` are already declared at the start of the program in lines 05 and 06 because they are needed in Python function `update(val)` for the calculation of the volume work (line 23).

```
01  #37_sld_p_V_diagram.py
02  import numpy as np
03  import matplotlib.pyplot as plt
04  from matplotlib.widgets import Slider
05  n=0.045  #mol
06  R=8.314  #J/(mol*K)
07  #p=f(V), isothermal, T as parameter
08  def p(V,T):
09      return 1e-2*n*R*T/V
10  #query slider
11  def update(val):
12      Tw, Tk = sldTw.val, sldTk.val #warm, cold
13      V1, V2 = sldV1.val,sldV2.val
```

```python
    Vx = np.arange(V1,V2,0.001)
    y1.set_data(Vx,p(Vx,Tw)) #isotherm
    y2.set_data(Vx,p(Vx,Tk))
    point1.set_data([V1],[p(V1,Tw)]) #point1
    point2.set_data([V2],[p(V2,Tw)]) #point2
    point3.set_data([V2],[p(V2,Tk)]) #point3
    point4.set_data([V1],[p(V1,Tk)]) #point4
    line1.set_data([V1,V1],[p(V1,Tk),p(V1,Tw)]) #vertical line
    line2.set_data([V2,V2],[p(V2,Tk),p(V2,Tw)]) #vertical line
    W=n*R*(Tw-Tk)*np.log(V2/V1)
    eta=1-Tk/Tw
    txtW.set_text('W = %.2f J' %W)
    txtEta.set_text(r'$\eta$ = %.2f' %eta)
#Graphics area
fig, ax = plt.subplots(figsize=(6,6))
txtW=ax.text(0.9,15,'')
txtEta=ax.text(0.9,14,'')
fig.subplots_adjust(left=0.12,bottom=0.25)
ax.set_xlim(0.1,1.2)
ax.set_ylim(0,16)
ax.set(xlabel='V in Liter',ylabel='p in bar',title='p-V diagram')
y1, = ax.plot([],[],'k-',lw=2) #ordinate
y2, = ax.plot([],[],'k-',lw=2) #ordinate
line1,line2 = ax.plot([],[],'r--',[],[],'r--')
point1,point2 = ax.plot([],[],'bo',[],[],'ro')
point3,point4 = ax.plot([],[],'go',[],[],'mo')
#x-, y-position, length, height
xyV1 = fig.add_axes([0.1, 0.12, 0.8, 0.03])
xyV2 = fig.add_axes([0.1, 0.08, 0.8, 0.03])
xyTw = fig.add_axes([0.1, 0.04, 0.8, 0.03])
xyTk = fig.add_axes([0.1, 0.0,  0.8, 0.03])
#create slider objects
sldTw=Slider(xyTw,r'$T_{w}$',501,800,valinit=800,valstep=1)   #warm
sldTk=Slider(xyTk,r'$T_{k}$',400,500, valinit=400,valstep=1)  #cold
sldV1=Slider(xyV1,r'$V_{1}$',0.2,0.5, valinit=0.2,valstep=0.01)
sldV2=Slider(xyV2,r'$V_{2}$',0.6,1, valinit=1.0,valstep=0.01)
#query changes
sldTw.on_changed(update)
sldTk.on_changed(update)
sldV1.on_changed(update)
sldV2.on_changed(update)
plt.show()
```

Listing 4.37 Simulation of a Stirling Cycle

Analysis

The n and R variables are already declared at the beginning of the program (line 05 and 06), so that they are available within the update(val) Python function (line 11). Alternatively, they could also be inserted as additional parameters in the custom p(V,T,R=8.314,n) Python function (line 08). Although this approach would follow better programming style, the programming effort would increase because a Python function would need to be defined for the calculation of the volume work (line 23).

In lines 12 and 13, the current values of the slider settings are assigned to the state variables Tw, Tk, V1, and V2. In line 14, the volume limits for V1 and V2 are adjusted.

The calculations for the volume work W and the efficiency eta are performed in lines 23 and 24. The set_text() method causes the currently calculated results (lines 25 and 26) to be output at the positions in the program's user interface that were specified in lines 29 and 30.

The on_changed(update) method (lines 51 to 54) calls the update Python function and passes it the current numerical values of the slider settings.

4.7 Project Task: Animating a Thread Pendulum

In this project task, the movement of a thread pendulum is to be animated, as shown in Figure 4.39.

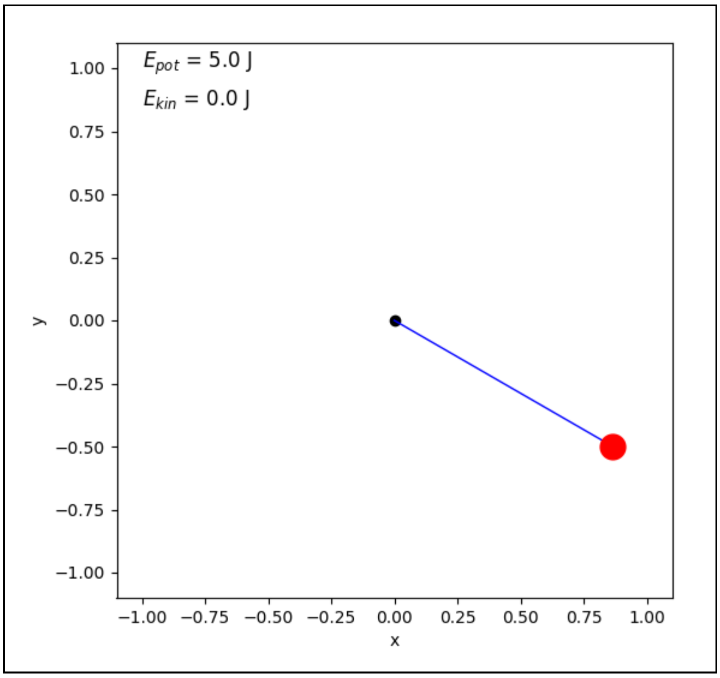


Figure 4.39 Snapshot of the Animation

The current potential energy $E_{pot} = m \cdot h$ and the current kinetic energy $E_{kin} = 0.5 \cdot m \cdot v^2$ need to be displayed in the user interface of the program. The height h can be calculated using the diagram shown in Figure 4.40.

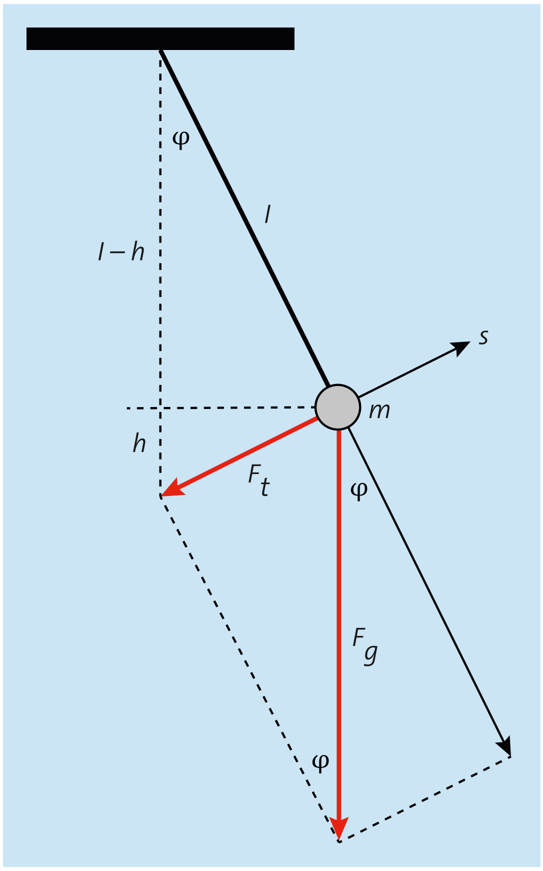


Figure 4.40 Forces on the Thread Pendulum

The total of the acceleration force $F_a = m \cdot a$ and the tangentially acting restoring force F_t must be equal to 0 at any time of the pendulum motion:

$F_a + F_t = 0$

Using $F_a = m \cdot l \cdot \ddot{\varphi}$ and $F_t = m \cdot g \cdot \sin\varphi$, you obtain the 2nd order nonlinear differential equation:

$m \cdot l \cdot \ddot{\varphi} + m \cdot g \cdot \sin\varphi = 0$

By rearrangement and using the abbreviation $\omega_0^2 = g/l$, we obtain:

$\ddot{\varphi} + \omega_0^2 \cdot \sin\varphi = 0$

To solve this differential equation numerically, it must be transformed into a 2nd order differential equation system:

$$\frac{d\varphi}{dt} = \omega$$
$$\frac{d\omega}{dt} = -\omega_0^2 \cdot \sin\varphi$$

Using the sum algorithm (Euler method), this differential equation system can be solved numerically in a particularly simple way:

```
repeat for t=0 to tmax with increment dt
      phi = phi + w*dt
        w = w - w0^2*sin(phi)*dt - d*w*dt
```

The additionally added term `d*w*dt` considers the effect of the damping *d*. For the damping, you can make the simplified assumption that it damps the oscillations proportionally to the angular velocity ω.

The sum algorithm is executed within a `for` loop.

Listing 4.38 shows the solution for this project task. In line 07, you can change the deflection angle.

```
01  #38_animation_thread_pendulum.py
02  import numpy as np
03  import matplotlib.pyplot as plt
04  from matplotlib.animation import FuncAnimation
05  #Data
06  l = 1.0         #pendulum length in m
07  angle = 60      #deflection angle
08  d = 0.0         #damping
09  m=10            #mass in kg
10  tmax = 50       #simulation duration
11  g = 9.81        #m/s^2
12  w02=g/l  #square of the circular frequency
13  #solution of the differential equation using the Euler method
14  dt = 1e-3 #increment
15  phi,w=np.radians(angle), 0.0  #initial values
16  t = np.arange(0, tmax, dt)
17  x,y = np.empty((len(t))),np.empty((len(t)))
18  v=np.empty((len(t)))
19  x[0]=y[0]=0
20  for i in range(len(t)):
21      phi = phi + w*dt           #deflection
22      w = w - w02*np.sin(phi)*dt - d*w*dt
23      v[i]=l*w
24      x[i],y[i] = l*np.sin(phi),-l*np.cos(phi) #x-y coordinates
25  #Animation function
26  def pendulum(j):
```

```
27      h=l+y[j]
28      Epot=m*h
29      Ekin=m*v[j]**2/2.0
30      txtEpot.set_text(f'$E_{{pot}}$={Epot:3.1f} J')
31      txtEkin.set_text(f'$E_{{kin}}$={Ekin:3.1f} J')
32      rod.set_data([0,x[j]],[0,y[j]])
33      sphere.set_data([x[j]],[y[j]])
34      return rod,sphere,txtEpot,txtEkin
35  #Graphics area
36  fig,ax= plt.subplots(figsize=(6, 6))
37  txtEpot=ax.text(-l,l,'',fontsize=12)
38  txtEkin=ax.text(-l,0.85,'',fontsize=12)
39  ival=1e3*dt
40  n=len(y)-1
41  width=1.1*l
42  ax.axis([-width,width,-width,width])
43  ax.set(xlabel='x',ylabel='y')
44  ax.set_aspect('equal')
45  ax.plot(0,0,'ko') #bearing
46  rod, = ax.plot([],[], 'b-', lw=1) #rod
47  sphere, =  ax.plot([],[], 'ro', markersize='15') #sphere
48  ani = FuncAnimation(fig, pendulum,frames=n,interval=ival,blit=True)
49  plt.show()
```

Listing 4.38 Animation of a Thread Pendulum

Analysis

The program consists of four parts:

1. Inputs (lines 06 through 09)
2. Solution of the differential equation (lines 14 to 24)
3. Definition of the animation function (lines 26 to 34)
4. Graphics area (lines 36 to 49)

To animate different scenarios, you can change the deflection angle in line 07, change the damping in line 08, and change the mass in line 09. You should not extend the setting for the pendulum length because the animation won't run as smoothly.

In line 14, you can adjust the increment dt to optimize the speed of the animation. Even on the same computer, program execution speeds differ when tested with different development environments. In lines 20 to 22, the sum algorithm is executed. For each individual support point i, the trajectory velocity v[i] of the pendulum is calculated from the angular velocity w and the pendulum length l in line 23. In line 24, the current trajectory coordinates x[i] and y[i] are calculated.

Within the custom animation function `pendulum(j)` (lines 26 to 34), the current values for the potential and kinetic energy (lines 28 and 29) are calculated and prepared for output on the screen in lines 30 and 31 using the `set_text()` method. In lines 32 and 33, the current coordinate data `x[j]` and `y[j]` are passed to the `set_data()` method and stored in the `rod` and `sphere` objects.

In line 48, the `FuncAnimation()` method processes the values of the `rod`, `sphere`, `txtEpot`, and `txtEkin` objects returned by the `pendulum()` function. In line 49, the `show()` method displays the animation on the screen.

4.8 Project Task: Animating a Transmission

A gear transmission is intended to reduce the rotational frequency of an electric motor from n_1 = 1500 min^{-1} to n_2 = 750 min^{-1}. The module chosen is m = 0.5. The pitch circle diameter (mean diameter) d_1 of the first gear is 8 cm. The transmission is supposed to be animated to illustrate the motion sequences.

The formulas necessary for dimensioning the transmission are listed in Table 4.3. The transmission ratio is i = 2.

Meaning	Formula
Pitch circle diameter of the second gear (mean diameter)	$d_2 = i \cdot d_1$
Distance between the gears	$a = \frac{d_1 + d_2}{2}$
Number of teeth of the first gear	$z_1 = \frac{d_1}{m}$
Number of teeth of the second gear	$z_2 = i \cdot z_1$
Tooth height	$h = \frac{13}{6} m$
Tooth clearance	$c = 0.2 \cdot m$

Table 4.3 Dimensioning of a Two-Stage Gear Transmission

The animation program is based on the original by Magnus Benjes. Only the identifiers of some variables were changed, a calculation part for the dimensioning of the transmission was added, and compact statements were distributed across several program lines.

In the program, the usual involute gearing is replaced by a trapezoidal one. The pitch circle of the gear is simulated by a polygon. The number of polygon sides corresponds to the number of teeth of a gear.

The x-y coordinates of the polygon corners are calculated using the Euler's formula:

$e^{j\alpha} = \cos\alpha + \mathrm{j}\sin\alpha$

The animation is performed using the `ArtistAnimation(fig,img,intv)` method. The `img` object must be a list. This method is used whenever geometric figures are to be animated. You can use Listing 4.39 to animate the motion of a gear transmission.

```
#39_animation_transmission.py
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as ani
from matplotlib.patches import Polygon
m=0.5   #module
i=2     #transmission ratio
d1=8    #mean diameter
d2=i*d1
a=(d1+d2)/2
z1=d1/m #number of teeth
z2=i*z1
h=13*m/6 #tooth height
c=0.2*m  #tooth clearance
i=complex(0,i)
x1=1/7
x2=1/3
tooth_shape=np.array([-x2,-x1,x1,x2])
frames=60
xmax=-11/16*d2,22/16*d2
ymax=-10/16*d2,10/16*d2
#Function definition for a gear
def gear(d,z,h):
    r=d/2
    alpha=2*np.pi/z #angle range
    sector=tooth_shape*alpha
    gear_section=np.array([r-h/2,r+h/2,r+h/2,r-h/2])-c
    tooth=gear_section*np.exp(1j*sector)
    return np.outer(np.exp(1j*alpha*np.arange(z)),tooth).ravel('C')
#Create gear objects
zr1=gear(d1,z1,h)
zr2=gear(d2,z2,h)*np.exp(1j*np.pi/z2)
step=2*np.pi/(z2*frames)
fig=plt.figure(figsize=(6,4))
ax=fig.add_axes([-0.2,-0.1,1.2,1.2])
image=[] #empty list
for k in range(frames):
    zr1=zr1*np.exp(-i*step) #right turning
```

```
39      zr2=zr2*np.exp(1j*step) #left turning
40      P1=Polygon(zr1.view(float).reshape(zr1.size,2),color='grey')
41      P2=Polygon(zr2.view(float).reshape(zr2.size,2)+[a,0],color='k')
42      image.append([ax.add_patch(P1),ax.add_patch(P2)])
43  an=ani.ArtistAnimation(fig,image,interval=20)
44  ax.set_aspect("equal")
45  ax.set_xlim(xmax)
46  ax.set_ylim(ymax)
47  plt.show()
```

Listing 4.39 Animation of a Gear Transmission

Output

Figure 4.41 shows a snapshot of the animation of a gear transmission.

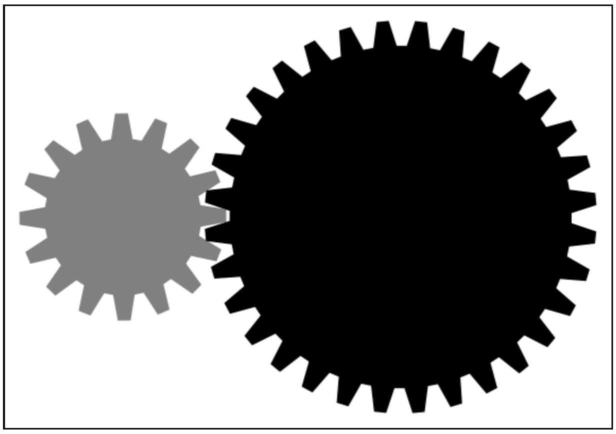


Figure 4.41 Snapshot of the Animation of a Gear Transmission

Analysis

The program consists of a total of six parts:

1. The input of the gear data for module `m`, the transmission ratio `i` and the pitch circle diameter `d1` of the first gear (line 06 to 08).
2. The calculation of the pitch circle diameter `d2` of the second gear, the distance `a` between the gears, the number of teeth `z1` and `z2`, the tooth height `h` and the tooth clearance `c` (line 09 to 14).
3. The definition of the `gear(d,z,h)` function for the calculation of the geometric data of a gear (line 23 to 29). The function expects three parameters when called: the diameter `d` of the gear, the number `z` of teeth, and the height `h` of a tooth. The `gear()` function returns the dyadic product of the term `np.exp(1j*alpha*np.arange(z))` and the `gear` object, flattened using NumPy method `ravel()`.

4. The creation of two gear objects, `zr1` and `zr2` (lines 31 and 32). The second gear is rotated one tooth position further by multiplying by the rotation factor `np.exp(1j*np.pi/z2)` so that the teeth do not overlap.
5. The generation of the images from two polygons (lines 36 to 42). Inside the `for` loop, the `Polygon()` constructors of the `Polygon` class create the `P1` and `P2` objects for the images of the two gears. The x and y coordinates for the polygon corners calculated in lines 38 and 39 are passed as the first parameter. The NumPy method `reshape(zr1.size,2)` transforms the coordinate data into a two-dimensional array. The `image.append()` method creates an array of 60 images from the two polygons because the `frames` variable was assigned the value 60 in line 19.
6. The `ArtistAnimation(fig,image,interval=20)` method that performs the animation (line 43). The first parameter passed is the `fig` object from line 34. The second parameter (`image`) contains all 60 images created within the `for` loop. They are displayed repeatedly with a delay of 20 ms.

4.9 Tasks

1. Write a program that represents the following two functions in a diagram:
   $y_1 = \cos x$
   $y_2 = x$
2. Write a program that plots the voltage drop $U = f(I)$ and power $P = f(I)$ for a 1-Ω resistor in a function plot. On the left axis, plot the voltage U in volts and, on the right axis, plot the power P in watts. To add scaling to the right axis, you must create a new object using the `twinx` method `a2=a1.twinx()`.
3. The gas consumption of a heating system for one week from Monday to Sunday is to be represented in a diagram as a line graph. For this period, the total gas consumption and its average value should be calculated and displayed within the diagram. You can save the days of the week in a list:

   ```
   days=['Mon','Tue','Wed','Thu','Fri','Sat','Sun']#label x-axis
   ```

   You can use the `set_xticks(np.arange(n),days)` method to label the x-axis. Write a program that meets these requirements.
4. A rectangular function is to be approximated by the Fourier series $y = \sum 10 \cdot \sin(kx)/k$ (for $k = 1, 3, 5, \ldots$). You can have the series calculated inside a `for` loop using `for k in range(1,n,2)`. The individual harmonics and the sum of these harmonics are to be represented. Write an appropriate program.
5. The AC resistance for an inductance, $X_L = 2\pi f L$ and a capacitance $X_C = \dfrac{1}{2\pi f C}$ is to be plotted in two subplots one below the other. Write an appropriate program.

6. Write a program that plots the time and location pattern of a sound wave ($f = 440$ Hz) in two subplots side by side as a cross of axes. The following applies:
   $y(t) = 10\sin(\omega t)\,, y(x) = 10\sin(kx)$

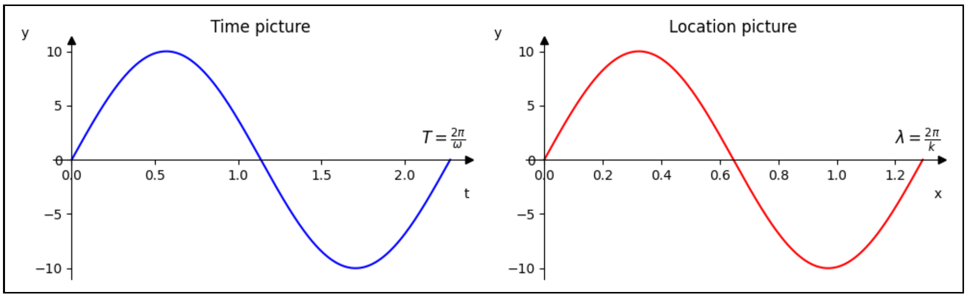


Figure 4.42 Time and Place Image of a Sound Wave

7. The function $y = \cos(1/x)$ is to be represented in the interval $10^{-6} \le x \le 1$. In an internal coordinate system, a detail of this function is to be displayed for the range from 0 to 0.2. For the Axes objects, the following shall apply:

   ```
   ax1=fig.add_axes([0.1,0.1,0.8,0.8])#external
   ax2=fig.add_axes([0.58,0.18,0.28,0.25])#internal
   ```

   Write a program that meets these requirements.
8. Write a program that draws a circle involute.
9. A sphere is to be represented using a 3D function plot. Write an appropriate program.
10. A pointer triangle is to be drawn for a series resonant circuit. Write an appropriate program.
11. A regular polygon is to be represented using the RegularPolygon() method. Write an appropriate program. The number of corners should be freely selectable.
12. For a damped vibration,

    $$y(t) = A \cdot e^{-\delta \cdot t} \cos\left(\frac{2\pi}{T} \cdot t\right)$$

    the amplitude, the damping and the period duration are to be changed using a slider control. Write a program that meets these requirements.
13. Fourier synthesis with a slider control is to be simulated for a rectangular signal. Only the sums of the individual harmonics are to be displayed. Write a program that meets these requirements.
14. The slider control is intended to simulate the superposition of two oscillations. The period duration is 20 ms. Both oscillations have an amplitude of 10. Only the phase angle of the second oscillation is to be changed. Write an appropriate program.
15. Modify Listing 4.20 so that the circuit operates as a phase section control.

16. Write a program that animates the motion of two sine functions in the direction of the x-axis. The first sine function should move from left to right, and the second sine function should move from right to left. The sum of both functions should also be represented.
17. A point is said to move on a sine curve $(0 \leq x \leq 2\pi)$. Write an animation program.
18. The Earth and Mars move in elliptical orbits around the Sun. The motion of Earth and Mars should be animated with a phase shift and with different velocities. You do not need to consider exact astronomical measurements. Only the qualitative correlations need to be illustrated. Write a program that displays the planets using the `mlt.patches.Circle()` method and animates them using the `FuncAnimation()` method.
19. Animate the Moon-Earth-Sun rotation system using the `FuncAnimation()` method. All three objects must be represented via the `mlt.patches.Circle()` method. You should only consider qualitative relationships.
20. The sine curve oscillates up and down in the direction of the y-axis. Write an animation program for the process described by the following equation:

    $$y(x,t) = 2y_m \sin(kx) \cdot \cos(\omega t) \text{ mit } k = 1, 2, 3, \ldots$$

Chapter 5
Symbolic Computation Using SymPy

In this chapter, you'll learn how to perform symbolic computations using the SymPy module. This chapter covers standard topics in engineering mathematics such as differentiations, integrations, differential equations, and Laplace transformations.

SymPy is a Python library for computer algebra. The SymPy module is written entirely in Python and consists of several hundred thousand program lines. The developers of SymPy pursued the goal of creating a complete *computer algebra system (CAS)*.

A CAS is a computer program for computing algebraic expressions. In this context, mathematical operations are not performed with numbers but with symbols. A minimal CAS consists of a user interface (Window-based version, terminal version); an interpreter for parsing mathematical commands; and a system kernel that executes the commands. For example, after entering a command such as `diff(x^2,x);`, you can press `Shift` + `Return` (in CAS, *Maxima*), and the result `2x` appears in the user interface. In this way, you can write even extensive mathematical papers: Variables and functions must be defined, then they are manipulated, linked, and parsed according to the previously designed specifications. Comments can be inserted between the symbolic computations. A user can then save the *worksheet* that results from these computations in LaTeX or other formats.

For custom extensions, every CAS provides a script language that can be integrated into a worksheet. The CAS is the base system, while the scripting language has a complementary function.

Python, however, goes the opposite way: The programming language is the base system, and the respective desired functionality is provided as a module and imported into the Python program as required. Thus, a particular advantage of the modular concept is its flexibility.

Due to the SymPy module, Python can also be used like a conventional CAS (e.g., Maxima) in the terminal. Let's consider a simple example like the following function:

$y = x^3 - 2x^2 + 10$

Now, we want to calculate the first and the second derivative as well as the antiderivative. Type the following statements into the Python console:

```
>>> from sympy import *
>>> x=symbols('x')
>>> y=x**3-2*x**2+10
>>> diff(y,x)
3*x**2 - 4*x
>>> diff(y,x,2)
2*(3*x - 2)
>>> integrate(y,x)
x**4/4 - 2*x**3/3 + 10*x
```

The first step is to import the SymPy module. The asterisk operator (*) specifies that all functions, methods, and mathematical constants of the `sympy` module should be loaded. As a beginner, you should prefer this module import option so that the execution of your scripts won't be blocked due to missing methods. As your experience grows, you can then incorporate the specific submodules and methods you need for your projects.

The second statement specifies the names of the mathematical variables. If multiple variables are to be used, the statement is, for example, `x,y,z=symbols('x y z')`.

In another CAS, the console input in the third line defines the function on which the mathematical operations are to be performed.

The `diff(y,x)` command calculates the first derivative of the polynomial. After pressing `Return`, the solution appears directly in the next line. The calculation of the second derivative follows the same pattern, the difference being that a `2` must exist after the independent `x` variable, separated by a comma. Notice in this context that the term of the second derivative has already been simplified.

The `integrate(y,x)` command computes the antiderivative of the `y` polynomial. SymPy does not output an integration constant.

To understand the next few examples, we've provided an overview of the most important functions of SymPy, which are listed in Table 5.1.

Function	Description
`apart(p)`	Decomposes polynomial `p` into its partial fractions.
`cancel(p)`	Generates a polynomial function from partial fractions `p`.
`diff(f,x,k)`	Computes the k-th derivative of function `f`.
`dsolve(eq,f(x))`	Solves an ordinary differential equation (eq) for function `f(x)`.
`N(Z,n)`	Generates a number with `n` digits for the number `Z` of type `Float`.

Table 5.1 Selected Functions of SymPy

Function	Description
`expand(T)`	Computes the term `T`.
`integrate(f,x)`	Computes the antiderivative `F(x)` of function `f(x)`.
`integrate(f,(x,a,b))`	Computes the definite integral of function `f` in the limits from `a` to `b`.
`limit(y,x,0)`	Computes the limit of a function.
`simplify(term)`	Simplifies a term.
`solve(F,x)`	Solves an equation.
`together()`	Combines two expressions.

Table 5.1 Selected Functions of SymPy (Cont.)

Table 5.2 contains a selection of frequently used SymPy methods.

Method	Description
`obj.doit()`	Analyzes `obj` objects that are not parsed by default, such as sums, products, limits, derivatives, and integrals. The analysis is recursive.
`Z.evalf(n)`	Generates a float with `n` digits for the number `Z` of type `Float`.
`f.series(x,0,n)`	Develops a series for the mathematical function `f` with `n` members at point $x_0 = 0$.
`s.subs(x,y)`	Replaces expressions.

Table 5.2 Selected Methods of SymPy

You can use certain SymPy functions as if they were methods, as the following console dialog shows:

```
>>> from sympy import *
>>> x=symbols('x')
>>> y=x**2
>>> y.diff(x)
2*x
>>> y.integrate(x)
x**3/3
```

For this reason, consistently referring to all SymPy functions as methods makes sense for the sake of better readability.

Note

In this chapter, all SymPy functions are referred to as *methods*. This choice of this term has the additional advantage of avoiding confusion with the mathematical term *function*.

SymPy also provides its own mathematical constants, listed in Table 5.3. Using the `pi.evalf(10)` method, for example, you can display the value of π with nine decimal places.

Constant	Symbol	Meaning
$\pi = 3.141592654$	`pi`	Pi
$e = 2.718281828$	`E`	Euler number
$\Phi = 1.618033989$	`GoldenRatio`	Golden ratio
∞	`oo`	Infinite

Table 5.3 Constants of SymPy

SymPy provides the trigonometric functions `cos`, `sin`, and `tan` as well as the e-function `exp()` and the hyperbolic functions `sinh`, `cosh`, and `tanh`, so you don't need to resort to using the NumPy module. In fact, using the corresponding NumPy functions is not recommended to avoid conflicts between the namespaces of both modules.

Note: Do Not Mix NumPy and SymPy

Do not use NumPy functions in the SymPy module.

Other common mathematical functions are listed in Table 5.4.

Function	Description
`Abs(x)`	Amount function
`binomial(n,k)`	Binomial coefficient
`factorial(n)`	Factorial
`fibonacci(n)`	*n*-th element of a Fibonacci sequence
`log(x)`	Natural logarithm
`log(x,a)`	Logarithm for basis *a*

Table 5.4 Important Built-In Functions of SymPy

5.1 Basic Mathematical Operations

Like other CAS, SymPy is adept at symbolic addition, subtraction, multiplication, division, and exponentiation operations. For basic mathematical operations, the known operators +, -, /, and * are available. Internally, SymPy uses the `Add()` and `Mul()` methods for basic arithmetic:

```
>>> Add(2,3)
5
>>> Mul(2,3)
6
>>> Mul(6,1/3)
2.00000000000000
```

However, you don't need to use this inconvenient notation. You can perform all basic arithmetic operations using the infix operators.

The following examples show how SymPy performs these operations on symbolic variables.

5.1.1 Addition

Listing 5.1 shows the addition of the four terms `T1`, `T2`, `T3`, and `T4`, which are composed of the variables `a`, `b`, `c`, and `d`.

```
01  #01_add.py
02  from sympy import *
03  a,b,c,d=symbols("a b c d")
04  T1=9*a+7*b-2*c+3*d
05  T2=8*a+2*b+3*c+4*d
06  T3=7*a-3*b+2*c+5*d
07  T4=4*a+2*b+5*c-6*d
08  T=T1+T2+T3+T4
09  print("Sum of terms")
10  pprint(T)
```

Listing 5.1 Summarizing Terms

Output

```
Sum of terms
28⋅a + 8⋅b + 8⋅c + 6⋅d
```

Indications and Procedures

Line 02 imports all methods of the SymPy module without taking into account that, in this example, actually only the `symbols` and `pprint` methods are needed. This approach is justified because, if SymPy wants to compete with another CAS, the effort would no longer be justifiable for a beginner if they had to explicitly import the required methods for each new task.

In line 02, you could use the following statement to import only the required methods:

```
from sympy import symbols, pprint
```

However, this option is more complex and error prone in complex programs.

Line 03 specifies the symbols for the mathematical variables the program should use to perform the computations. At this point, you can use the variable identifiers known from mathematics, which means that SymPy allows user-defined variable names. You're free to decide which variable names you want to use, and you're not locked into x-y math.

The assignments in lines 04 to 07 define the terms that are added up in line 08. The multiplication operator `*` must be placed between the factors and the variables, for example, `7*b`. The notation `7b` would trigger an error message.

In line 10, the `pprint()` method outputs the formatted result in which Unicode characters are used. The first `p` is supposed to stand for *pretty*.

You can check the correctness of the result by doing your own recalculation.

Both `print(type(a))` and `print(type(T))` enable you to print the types of the `a` and `T` variables, as shown in the following examples:

```
<class 'sympy.core.symbol.Symbol'> #a
<class 'sympy.core.add.Add'>       #T
```

You can use `print(srepr(T))` to display the internal structure of the term `T`, as shown in the following examples:

```
Add(Mul(Integer(28), Symbol('a')), Mul(Integer(8), Symbol('b')), Mul(Integer(8),
Symbol('c')), Mul(Integer(6), Symbol('d')))
```

The different parenthetical levels can be visualized in a tree structure. For an example of the visualization of a tree structure, refer to *https://docs.sympy.org/latest/tutorials/intro-tutorial/manipulation.html*.

5.1.2 Multiplication of Terms

Of course, SymPy is also adept at symbolic multiplication. Listing 5.2 shows how to multiply two terms (`T1` and `T2`) with the symbolic variables `a`, `b`, and `c`.

```
01  #02_mul.py
02  from sympy import *
03  a,b,c=symbols("a b c")
04  T1=2*a+4*b-5*c
```

```
05  T2=4*a+2*b+3*c
06  T=T1*T2
07  print("Products of the terms")
08  pprint(T)
09  print("Terms multiplied")
10  print(expand(T))
11  print("Formatted output")
12  pprint(expand(T))
```

Listing 5.2 Multiplication of Terms

Output

```
Products of the terms
(2·a + 4·b - 5·c)·(4·a + 2·b + 3·c)
Terms multiplied
8*a**2 + 20*a*b - 14*a*c + 8*b**2 + 2*b*c - 15*c**2
Formatted output
   2                     2             2
8·a  + 20·a·b - 14·a·c + 8·b  + 2·b·c - 15·c
```

Analysis

In line 06, the multiplication operation is executed, but the result that is output in line 08 does not match the expectation. For the parentheses to be multiplied out, the `T` object must be passed to the `expand(T)` method (line 12).

`print(type(T))` allows you to output the type of the `T` variable: `<class 'sympy.core.mul.Mul'>`.

5.1.3 Multiplication of Linear Factors

The multiplication of *n* linear factors `(x+x1)*(x+x2)* ... *(x+xn)` results in polynomials of the *n*-th degree. `x1` to `xn` represent the zeros of the polynomial. The number of linear factors determines the degree of the polynomial. Listing 5.3 generates a fourth-degree polynomial from three linear factors using the `expand()` method.

```
#03_mul_linear_factors.py
from sympy import *
x=symbols("x")
lf=(x-1)*(x-2)*(x-3)*(x-4)
p=expand(lf)
print("\nThe multiplication of the linear factors")
pprint(lf)
```

```
08 print("generates the polynomial of the 4th degree")
09 pprint(p)
```

Listing 5.3 Multiplication of Linear Factors

Output

```
The multiplication of the linear factors
(x - 4)⋅(x - 3)⋅(x - 2)⋅(x - 1)
generates the polynomial of the 4th degree
 4        3         2
x  - 10⋅x  + 35⋅x  - 50⋅x + 24
```

Analysis

In line 05, the linear factors from line 04 are multiplied using the `expand(lf)` method and stored in the `p` object. The output in line 09 confirms the expected result.

5.1.4 Division

Listing 5.4 demonstrates the division of terms. The symbolic division operation is performed using the / operator.

```
#04_div.py
from sympy import *
a,b,c=symbols("a b c")
T1=2*a+4*b-5*c
T2=4*a+2*b+3*c
T3=5*a-3*b+4*c
T=T1/(T2*T3)
print("Division of terms")
pprint(T)
print("Terms multiplied")
print(expand(T))
```

Listing 5.4 Division of Terms

Output

```
Division of terms
          2⋅a + 4⋅b - 5⋅c
─────────────────────────────────────────────────
(4⋅a + 2⋅b + 3⋅c)⋅(5⋅a - 3⋅b + 4⋅c)
Terms multiplied
```

```
2*a/(20*a**2 - 2*a*b + 31*a*c - 6*b**2 - b*c + 12*c**2) + 4*b/(20*a**2 - 2*a*b +
31*a*c - 6*b**2 - b*c + 12*c**2) - 5*c/(20*a**2 - 2*a*b + 31*a*c - 6*b**2 - b*c
+ 12*c**2)
```

Analysis

Line 07 performs the symbolic division operation. The `expand(T)` method is used to calculate three fractions in line 11 with the denominator terms that have been multiplied with each other. The example clearly shows how SymPy's CAS methods can greatly facilitate the necessary computational work.

If you output the type of the `T` variable via `print(type(T))`, you'll obtain the following output: `<class 'sympy.core.mul.Mul'>`. From this result, we can conclude that no `Div()` method exists.

5.1.5 Exponentiation

Symbolic exponentiation can be illustrated through an example with the binomial formula:

$(a + b)^n$

Listing 5.5 calculates the binomials for $n = 1$ through 6.

```
#05_binom.py
from sympy import *
a,b=symbols("a b")
for n in range(7):
    p=(a+b)**n
    print(expand(p))
```

Listing 5.5 Powers of Sums

Output

```
1
a + b
a**2 + 2*a*b + b**2
a**3 + 3*a**2*b + 3*a*b**2 + b**3
a**4 + 4*a**3*b + 6*a**2*b**2 + 4*a*b**3 + b**4
a**5 + 5*a**4*b + 10*a**3*b**2 + 10*a**2*b**3 + 5*a*b**4 + b**5
a**6 + 6*a**5*b + 15*a**4*b**2 + 20*a**3*b**3 + 15*a**2*b**4 +
6*a*b**5 + b**6
```

Analysis

This program illustrates that symbolic exponentiation is also possible using SymPy. In line 06, the `expand(p)` method calculates the powers of the binomials from line 05. Again, this example shows the capabilities of SymPy's CAS functionality.

If you output the type of the `p` variable via `print(type(p))`, you'll obtain the following output: `<class 'sympy.core.power.Pow'>`. Thus, a method exists for the exponentiation operation. You can test this method using `Pow(2,100)`, for example.

5.1.6 Usage Example: Analyzing an Electrical Power Transmission System

Let's now apply computation using symbolic variables for the analysis of an electrical network. Using a simple example, I will show you how to use SymPy to compute the efficiency of an electrical power transmission system for a direct current (DC), as shown in Figure 5.1. For this purpose, we need to compute the total resistance and the current.

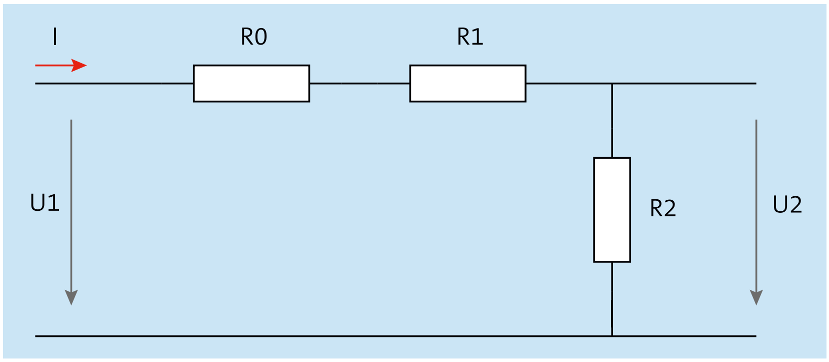


Figure 5.1 Equivalent Circuit Diagram for a Power Transmission System

Each DC power transmission system consists of the internal resistance of the voltage source R_0, the conductor resistance R_1, and the consumer resistance R_2. Based on this information, you can derive a formula to calculate the efficiency of the network. For the total resistance, the following equation applies:

$$R_g = R_0 + R_1 + R_2$$

The total current I_g is calculated from the input voltage and the total resistance:

$$I_g = \frac{U_1}{R_g}$$

The following equation applies to the input power P_1:

$$P_1 = R_g \cdot I_g^2$$

For the output power, P_2 applies accordingly:

$$P_2 = R_2 \cdot I_g^2$$

Using these equations, you can obtain the formula for the efficiency:

$$\eta = \frac{P_2}{P_1} = \frac{R_2}{R_0 + R_1 + R_2}$$

SymPy performs these symbolic arithmetic operations, as shown in Listing 5.6, and establishes a general formula for calculating the efficiency of a power transmission system for direct current.

```
01  #06_efficiency.py
02  from sympy import *
03  R0,R1,R2,U1,U2=symbols("R0 R1 R2 U1 U2")
04  Rg=R0+R1+R2
05  Ig=U1/Rg
06  P1=Rg*Ig**2
07  P2=R2*Ig**2
08  eta=P2/P1
09  print(u"\N{GREEK SMALL LETTER ETA}= ",eta)
```

Listing 5.6 Efficiency of a Direct Current Line

Output

```
η = R2/(R0 + R1 + R2)
```

Analysis

This example illustrates that you can use any identifier for the required symbolic variables (line 03). In lines 04 to 08, symbolic computations are executed according to the specifications. Line 09 outputs the expected result.

5.2 Multiplying Matrixes

SymPy also handles all arithmetic operations defined on matrixes. In electrical engineering, the analysis and synthesis of two-port networks can be performed particularly elegantly using the addition and multiplication of matrixes. For the analysis of catenary circuits from elementary two-port networks, only the matrix multiplication is needed.

5.2.1 Calculation Rule

Matrixes are multiplied based on the rule, "row vector multiplied by column vector," as illustrated in the following:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{pmatrix}$$

Listing 5.7 shows how to multiply matrixes symbolically with each other using SymPy.

```
01 #07_matrix_mul1.py
02 from sympy import *
03 a,b,c,d = symbols("a,b,c,d")
04 e,f,g,h = symbols("e,f,g,h")
05 A=Matrix([[a,b],
06           [c,d]])
07 B=Matrix([[e,f],
08           [g,h]])
09 C=A*B
10 D=B*A
11 print("Product A*B\n")
12 pprint(C)
13 print("\nProduct B*A\n")
14 pprint(D)
```

Listing 5.7 Matrix Multiplication

Output

```
Product A*B
⎡a⋅e + b⋅g  a⋅f + b⋅h⎤
⎣c⋅e + d⋅g  c⋅f + d⋅h⎦

Product B*A
⎡a⋅e + c⋅f  b⋅e + d⋅f⎤
⎣a⋅g + c⋅h  b⋅g + d⋅h⎦
```

Analysis

The symbolic definition of a matrix is performed in lines 05 and 07 using the `Matrix([[row1],[[row2]])` method. The multiplications of matrixes `A` and `B` in lines 09 and 10 show that the *commutative law* does not apply to matrixes.

5.2.2 Transmission Function of a Catenary Circuit

A transmission function describes the ratio of the output voltage to the input voltage of a two-port network, according to the following equation:

$$H(s) = \frac{U_2(s)}{U_1(s)}$$

If the transmission function and the input voltage are given, then you can calculate the output voltage using the following equation:

$$U_2(s) = H(s) \cdot U_1(s)$$

Figure 5.2 shows a catenary circuit consisting of three inductances as longitudinal links and two capacitances as cross links. The terminating resistor $R = 1\,\Omega$ is also a cross link.

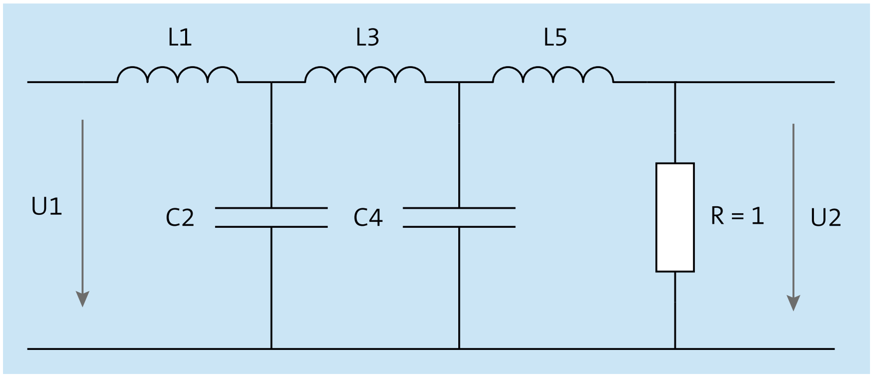


Figure 5.2 Catenary Circuit for a Fifth-Degree Low-Pass Filter

A catenary circuit is composed of n cross links and m longitudinal links. The chain parameters of these links are shown in Table 5.5.

Cross Link	Longitudinal Link
$\begin{pmatrix} 1 & 0 \\ Y & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & Z \\ 0 & 1 \end{pmatrix}$

Table 5.5 Chain Parameters for Elementary Links

Z can be a resistive, capacitive, or inductive alternating current (AC) resistance (impedance). The following equation applies to the inductive reactance X_L:

$$X_L = s \cdot L \text{ with } s = j\omega$$

The following equation applies to the capacitive conductance Y_C:

$$Y_C = s \cdot C \quad \text{with } s = j\omega$$

To calculate the resulting chain matrix, you must multiply the matrixes of the cross links and longitudinal links with each other.

Listing 5.8 multiplies the individual partial matrixes with each other. The resulting chain matrix and the denominator polynomial of the transmission function are output.

```
#08_matrix_mul2.py
from sympy import *
s,L1,C2,L3,C4,L5 = symbols("s L1 C2 L3 C4 L5")
A1=Matrix([[1, L1*s],
           [0, 1]])
A2=Matrix([[1,  0],
           [C2*s,1]])
```

```
08  A3=Matrix([[1,L3*s],
09             [0, 1]])
10  A4=Matrix([[1,   0],
11             [C4*s, 1]])
12  A5=Matrix([[1,L5*s],
13             [0,   1]])
14  A6=Matrix([[1, 0],
15             [1, 1]])
16  A=A1*A2*A3*A4*A5*A6
17  print("Chain parameters")
18  print(A)
19  print("Denominator polynomial of the transmission function")
20  print(expand(A[0,0]))
```

Listing 5.8 Parameters of a Catenary Circuit

Output

```
Chain parameters
Matrix([[C2*L1*s**2 + C4*s*(L1*s + L3*s*(C2*L1*s**2 + 1)) + L1*s +
L3*s*(C2*L1*s**2 + 1) + L5*s*(C2*L1*s**2 + C4*s*(L1*s + L3*s*(C2*L1*s**2 + 1)) +
1) + 1, L1*s + L3*s*(C2*L1*s**2 + 1) + L5*s*(C2*L1*s**2 + C4*s*(L1*s +
L3*s*(C2*L1*s**2 + 1)) + 1)], [C2*L3*s**2 + C2*s + C4*s*(C2*L3*s**2 + 1) +
L5*s*(C2*s + C4*s*(C2*L3*s**2 + 1)) + 1, C2*L3*s**2 + L5*s*(C2*s +
C4*s*(C2*L3*s**2 + 1)) + 1]])
Denominator polynomial of the transmission function
C2*C4*L1*L3*L5*s**5 + C2*C4*L1*L3*s**4 + C2*L1*L3*s**3 + C2*L1*L5*s**3 +
C2*L1*s**2 + C4*L1*L5*s**3 + C4*L1*s**2 + C4*L3*L5*s**3 + C4*L3*s**2 + L1*s +
L3*s + L5*s + 1
```

Analysis

The definition of the matrixes for the individual elementary two-port networks is performed in lines 04 to 15. The matrix multiplication is performed in line 16. Special care must be taken to ensure that the order of the multipliers corresponds to the circuit's structure. The outputs show that such complex operations can hardly be performed manually.

5.3 Equations

Systems of linear equations play a central role in the calculation of node voltages and mesh currents in electrical networks and the distribution of forces in trusses. In mathematics, partial fraction decomposition and the solving of linear differential equations with constant coefficients and higher orders also require equations to be solved.

5.3.1 Linear Systems of Equations

SymPy solves a linear system of equations using the following method:

```
solve((g1, g2, g3, g4), x1, x2, x3, x4)
```

The objects g1 to gn represent the rows of a linear system of equations. The variables x1 to xn are the unknowns.

The individual lines of the equation system, such as 7*x1+5*x2-2*x3+7*x4=9 are transformed so that the row elements of the result vector are on the left side of the equation system: 7*x1+5*x2-2*x3+7*x4-9=0, where the zero is no longer considered in the solve method.

Listing 5.9 shows the implementation.

```
01  #09_solve1.py
02  from sympy import *
03  x1,x2,x3,x4 = symbols('x1 x2 x3 x4')
04  #linear system of equations
05  g1=7*x1+5*x2-2*x3+7*x4-9
06  g2=6*x1+3*x2-4*x3+6*x4-8
07  g3=3*x1+2*x2-5*x3+5*x4-4
08  g4=2*x1+9*x2-6*x3+3*x4-2
09  #solution
10  L=solve((g1,g2,g3,g4),x1,x2,x3,x4)
11  #Output
12  print("Solution set\n",L)
```

Listing 5.9 Linear System of Equations

Output

```
Solution set
 {x1: 431/305, x2: -34/305, x3: -19/305, x4: -4/61}
```

Analysis

The rows of the rearranged equation system are assigned to objects g1 to g4 in rows 05 to 08. The solution follows in line 10 using the solve((g1,g2,g3,g4),x1,x2,x3,x4) method. The rows of the equation system are passed as tuples. Next, follow the parameters of the unknown variables x1 to x4 separated by commas. As expected, the solution set does not consist of real numbers but of rational numbers. Consequently, SymPy computes exactly in this case.

Usage Example: Nodal Analysis

Using the example of a low-pass filter in a π circuit, I will show you how a system of equations with two unknowns can generally be solved.

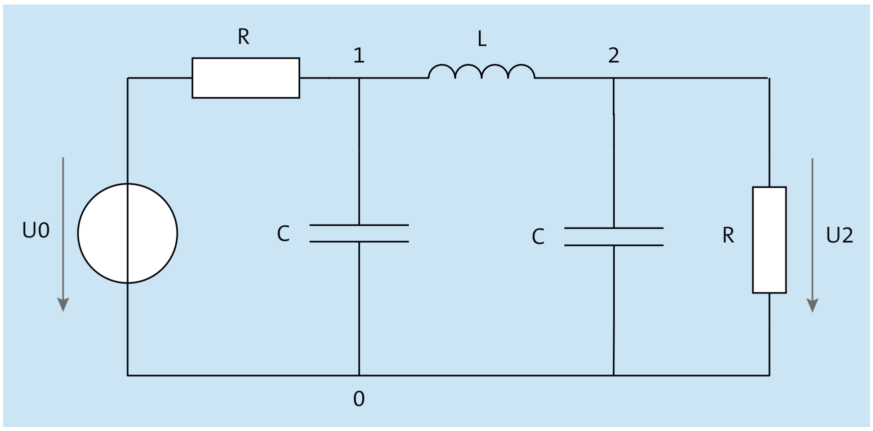


Figure 5.3 Low-Pass Filter in π Circuit

From the circuit shown in Figure 5.3, you can use nodal analysis to set up a system of equations for the node voltages $U_{1,0}$ and $U_{2,0}$, for example:

$$\left(\frac{1}{R}+C\cdot s+\frac{1}{L\cdot s}\right)U_{1,0}-\frac{1}{L\cdot s}\,U_{2,0}=\frac{U_0}{R}$$
$$-\frac{1}{L\cdot s}U_{1,0}+\left(\frac{1}{R}+C\cdot s+\frac{1}{L\cdot s}\right)U_{2,0}=0$$

Listing 5.10 computes these node voltages as general expressions.

```
#10_solve2.py
from sympy import *
s,U0,U1,U2,R,C,L = symbols("s,U0,U1,U2,R,C,L")
#node equations
I1=(1/R+C*s+1/(L*s))*U1-U2/(L*s)-U0/R
I2=-U1/(L*s)+(1/R+C*s+1/(L*s))*U2
#solving the node equations
U=solve((I1,I2),U1,U2)
print(U)
```

Listing 5.10 General Solution of a Second-Degree System of Equations

Output

```
{U1:-L*U0*s*(C*L*R*s**2+L*s+R)/(R**2-(C*L*R*s**2+L*s+R)**2),
 U2:-L*R*U0*s/(R**2-(C*L*R*s**2+L*s+R)**2)}
```

Analysis

In lines 05 and 06, the rows of the equation system are assigned to the I1 and I2 objects. Line 08 computes the general solution of the equation system. The output of the node voltages U1 and U2 is performed using the dictionary data structure.

5.3.2 Nonlinear Systems of Equations

Let's now consider the following nonlinear equations:

$$y_1 = x^4 - 7x^3 - 13x^2 + 79x + 84 = 0$$

$$y_2 = \ln\left(\sqrt{x} - 2\right) = 1$$

$$y_3 = e^{\sqrt{x}-2} = 1$$

$$y_4 = \sinh x = 10$$

$$y_5 = \cosh x = 10$$

For these equations, Listing 5.11 calculates the zeros.

```
#11_solve3.py
from sympy import *
a,b,x=symbols("a,b,x")
#Equations
y1=x**4-7*x**3-13*x**2+79*x+84
y2=log(sqrt(x)-2)-1
y3=exp(sqrt(x)-2)-1
y4=sinh(x)-10
y5=cosh(x)-10
#Outputs
print("Solutions")
print("f(x)=%s|f(x=0)=%s" %(y1,solve(y1,x)))
print("f(x)=%s|f(x=0)=%s" %(y2,solve(y2,x)))
print("f(x)=%s|f(x=0)=%s" %(y3,solve(y3,x)))
print("f(x)=%s|f(x=0)=%s" %(y4,solve(y4,x)))
print("f(x)=%s|f(x=0)=%s" %(y5,solve(y5,x)))
```

Listing 5.11 Nonlinear Systems of Equations

Output

```
Solutions
f(x)=x**4 - 7*x**3 - 13*x**2 + 79*x + 84|f(x=0)=[-3, -1, 4, 7]
f(x)=log(sqrt(x) - 2) - 1|f(x=0)=[(2 + E)**2]
f(x)=exp(sqrt(x) - 2) - 1|f(x=0)=[4]
f(x)=sinh(x)-10|f(x=0)=[log(-10+sqrt(101))+I*pi,
log(10 + sqrt(101))]
f(x)=cosh(x) - 10|f(x=0)=[log(10 - 3*sqrt(11)),
log(3*sqrt(11) + 10)]
```

Analysis

The mathematical functions defined in lines 05 to 09 are solved and output in lines 12 to 16 using the `solve()` SymPy method. The outputs provide the exact values (i.e., no floats).

In output `I*pi`, `I` stands for the imaginary unit. Using `plot(yi,(x,ug,og))`, you can use the function plots to illustrate the solutions (intersections with the x-axis).

Usage Example: Design of a Butterworth Low-Pass Filter

One method of design low-pass filters is to compare the electrical components L and C with the coefficients of the transmission function. The number of equations that are set up in the coefficient comparison corresponds to the degree of the filter. For example, to design a fifth-degree Butterworth lowpass filter, you'll obtain a nonlinear system of equations with five equations. Such a system of equations can most likely no longer be solved manually.

For Butterworth coefficients, the literature provides the following normalized transmission function:

$$\frac{1}{s^5 + 3.236s^4 + 5.236s^3 + 5.236s^2 + 3.236s + 1}$$

Using Listing 5.11, the denominator polynomial of the transmission function for a fifth-degree low-pass filter can be calculated in the following way:

```
C2*C4*L1*L3*L5*s**5 + C2*C4*L1*L3*s**4 + C2*L1*L3*s**3 + C2*L1*L5*s**3 +
C2*L1*s**2 + C4*L1*L5*s**3 + C4*L1*s**2 + C4*L3*L5*s**3 + C4*L3*s**2 + L1*s +
L3*s + L5*s + 1
```

As a result of the coefficient comparison, you can obtain the following system of equations:

$$\begin{aligned}
L_1 + L_3 + L_5 &= 3.236 \\
C_2L_1 + C_4L_1 + C_4L_3 &= 5.236 \\
C_2L_1L_3 + C_2L_1L_5 + C_4L_1L_5 + C_4L_3L_5 &= 5.236 \\
C_2C_4L_1L_3 &= 3.236 \\
C_2C_4L_1L_3L_5 &= 1
\end{aligned}$$

For ohmic resistance, the value of 1 Ω is set. The calculated values for the inductances and capacitances have the units 1 H (Henry) and 1 F (Farad), respectively. Listing 5.12 solves the nonlinear equation system we've set up and outputs the values for the various components as a dictionary.

```
01  #12_solve4.py
02  from sympy import *
03  s,L1,C2,L3,C4,L5 = symbols("s L1 C2 L3 C4 L5")
04  #Butterworth coefficients
05  a=[0,3.236,5.236,5.236,3.236,1]
06  #nonlinear system of equations
07  g1=L1+L3+L5-a[1]
08  g2=C2*L1 + C4*L1 + C4*L3-a[2]
09  g3=C2*L1*L3 + C2*L1*L5 + C4*L1*L5-a[3]
10  g4=C2*C4*L1*L3-a[4]
11  g5=C2*C4*L1*L3*L5-a[5]
12  components=solve((g1,g2,g3,g4,g5),L1,C2,L3,C4,L5,dict=True)
13  #Output
14  print(components[1])
15  print(components[1].keys())
16  print(components[1].values())
17  for item in components[1].items():
18      print("%s = %.3f"%item)
```

Listing 5.12 Dimensioning a Butterworth Low-Pass Filter

Output

```
{C2: 1.85739429156729, C4: 0.815341330818977, L1: 1.53414656573512, L3:
1.39282994847996, L5: 0.309023485784920}
dict_keys([C2, C4, L1, L3, L5])
dict_values([1.85739429156729, 0.815341330818977, 1.53414656573512,
1.39282994847996, 0.309023485784920])
C2 = 1.857
C4 = 0.815
L1 = 1.534
L3 = 1.393
L5 = 0.309
```

Analysis

Line 05 stores Butterworth coefficients in array object `a`. The value zero was provided as the first element to make how Python indexes arrays match the indexing of the Butterworth coefficients.

Lines 07 to 11 contain the individual equations of the nonlinear system of equations. In line 12, the system of equations is solved. The last parameter (`dict=True`) specifies that the solution set is stored as a dictionary in the `components` object.

Line 14 outputs the complete solution set as a dictionary. Line 15 outputs the keys, and line 16 outputs the dictionary values. The values for the individual components are output in line 18.

The values for the components are again the values normalized to 1 Ω, 1 F, and 1 H.

5.4 Simplifications of Terms

When deriving physical laws or calculating transmission functions for AC electrical networks, mathematical terms can arise that can still be greatly simplified. For this purpose, SymPy provides the `simplify(term)` method. Listing 5.13 is intended to simplify the following five terms:

$$e^{\ln x+\ln y}$$

$$\frac{nx^n}{x}$$

$$\frac{a^3}{(a-b)\cdot(a-c)}+\frac{b^3}{(b-a)\cdot(b-c)}+\frac{c^3}{(c-a)\cdot(c-b)}$$

$$2\sqrt{\frac{1}{x}}-\frac{1}{\sqrt{x}}$$

$$\frac{y^2+y}{y\cdot\sin^2 a+y\cdot\cos^2 a}$$

```python
#13_simplify.py
from sympy import *
a,b,c,n,x,y=symbols("a b c n x y")
#Terms
t1=exp(log(x)+log(y))
t2=n*x**n/x
t3=a**3/((a-b)*(a-c))+b**3/((b-c)*(b-a))+c**3/((c-a)*(c-b))
t4=2*sqrt(1/x)-1/sqrt(x)
t5=(y**2 + y)/(y*sin(a)**2 + y*cos(a)**2)
#Outputs
print("1: exp(log(x)+log(y)), simplified:",t1)
print("2:",t2,",simplified:",simplify(t2))
print("3:",t3,"\n   simplified:",simplify(t3))
print("4:",t4,",simplified",simplify(t4))
print("5:",t5,",simplified:",simplify(t5))
```

Listing 5.13 Simplification of Mathematical Terms

Output

```
1: exp(log(x)+log(y)), simplified: x*y
2: n*x**n/x, simplified: n*x**(n - 1)
3: a**3/((a-b)*(a-c))+b**3/((-a + b)*(b-c))+c**3/((-a+c)*(-b+c))
   simplified: a + b + c
4: 2*sqrt(1/x) - 1/sqrt(x), simplified 2*sqrt(1/x) - 1/sqrt(x)
5: (y**2 + y)/(y*sin(a)**2 + y*cos(a)**2), simplified: y + 1
```

Analysis

As expected, SymPy uses the `simplify()` method to simplify all terms correctly except the third term. For the third term, `2*sqrt(1/x)-1/sqrt(x)`, the expected output is `1/sqrt(x)`. Maple, for example, provides this result. Strictly speaking, however, this result is only correct if you assume that the positive sign of the root term was meant. This problem is also pointed out in the SymPy documentation. SymPy automatically simplifies the first term when output via `print()`.

5.5 Series Expansion

Using the `f(x).series(x,x0,n)` method, SymPy calculates n members for the series of function `f(x)` at position `x0`. Listing 5.14 proves Euler's formula with the series expansion for the sine and cosine functions:

$e^{jx} = \cos x + \mathrm{j} \sin x$

```python
#14_series_expansion.py
from sympy import *
x=symbols('x')
n=10
a=cos(x).series(x,0,n)
b=(sin(x)*I).series(x,0,n)
c=exp(x*I).series(x,0,n)
d=a+b
#Output
print("Series expansion cos\n",a)
print("\nSeries expansion sin\n",b)
print("\nSeries expansion cos+sin\n",c)
print("\nSeries expansion e-function\n",d)
```

Listing 5.14 Series Expansion

Output

```
Series expansion cos
1 - x**2/2 + x**4/24 - x**6/720 + x**8/40320 + O(x**10)
Series expansion sin
I*x - I*x**3/6 + I*x**5/120 - I*x**7/5040 + I*x**9/362880 + O(x**10)
Series expansion cos+sin
1 + I*x - x**2/2 - I*x**3/6 + x**4/24 + I*x**5/120 - x**6/720 -
I*x**7/5040 + x**8/40320 + I*x**9/362880 + O(x**10)
Series expansion e-function
1 + I*x - x**2/2 - I*x**3/6 + x**4/24 + I*x**5/120 - x**6/720 -
I*x**7/5040 + x**8/40320 + I*x**9/362880 + O(x**10)
```

Analysis

In lines 06 and 07, the sine function and the exponential function are multiplied by the imaginary unit `I`. In line 08, the addition of the series for the cosine and sine functions takes place. The outputs in lines 10 to 13 confirm the results from the literature. The values for the members from line 12 match the values of the series members from line 13. As expected, the imaginary unit `I` disappears in members with even exponents.

5.6 Partial Fractions

Any rational function can be decomposed into n partial fractions. If you calculate the main denominator for these partial fractions, you get back the original rational function with its numerator and denominator polynomial. In the synthesis of electrical inputs, transfer functions of complex conductance (admittance) *Y(s)* or complex resistance (impedance) *Z(s)* are given as rational functions. From these specifications, the individual elements of the circuit can be dimensioned using partial fraction decomposition. Based on two simple electrical networks, I will now demonstrate how the synthesis of one-ports is performed using the method of decomposing partial fractions. Figure 5.4 shows a parallel circuit of three R-L series circuits. For the ohmic resistance, $R = 1\,\Omega$ is specified again.

The total conductance can be read directly from the circuit using the following equation:

$$Y(s) = \frac{1}{L_1 s + 1} + \frac{1}{L_2 s + 1} + \frac{1}{L_3 s + 1}$$

By calculating the main denominator, the rational function for the complex conductance is obtained for $L_1 = 3$ H, $L_2 = 2$ H, and $L_3 = 1$ H:

$$Y(s) = \frac{11s^2 + 12s + 3}{6s^3 + 11s^2 + 6s + 1}$$

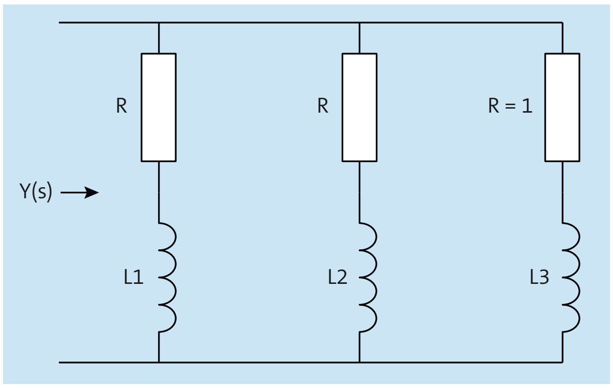


Figure 5.4 Parallel Circuit of R-L Elements

In network synthesis, transmission functions are given in this form. The individual components of the circuit must then be calculated using partial fraction decomposition. How we've described how to calculate the transmission function from a given circuit serves only to illustrate the synthesis procedure and to check the results for circuit synthesis. The reverse way of calculating the values for the components of a circuit from a transmission function $Y(s)$ or $Z(s)$ is thus easier to understand.

In the next circuit, shown in Figure 5.5, three parallel resonant circuits are connected in a series.

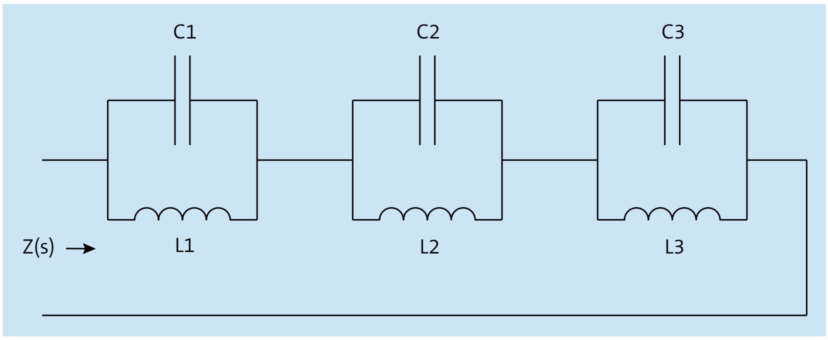


Figure 5.5 Series Circuit of L-C Elements

From this circuit, you can easily determine the partial fractions for the transmission function of the impedance:

$$Z(s) = \frac{L_1 s}{C_1 L_1 s^2 + 1} + \frac{L_2 s}{C_2 L_2 s^2 + 1} + \frac{L_3 s}{C_3 L_3 s^2 + 1}$$

For $L_1 = 3\text{ H}, C_1 = 4/3\text{ F}, L_2 = 2\text{ H}, C_2 = 3/2\text{ F}, L_3 = 1\text{ H}, C_3 = 2\text{ F}$, the transmission function of the impedance results as a rational function for the following:

$$Z(s) = \frac{46s^5 + 34s^3 + 6s}{24s^6 + 26s^4 + 9s^2 + 1}$$

From the partial fractions for $Y(s)$ and $Z(s)$, Listing 5.15 calculates the rational functions and, from the rational functions again, calculates the partial fractions through the `apart(rationalfunction)` method. By comparing coefficients, the normalized values of inductances and capacitances can be calculated.

```
#15_partial_fraction.py
from sympy import *
s=symbols("s")
#Parallel circuit from R-L series circuits
Yb1=1/(s+1)+1/(2*s+1)+1/(3*s+1)
Yp1=cancel(Yb1)
# Series circuit from L-C parallel circuits
Zb2=s/(2*s**2+1)+2*s/(3*s**2+1)+3*s/(4*s**2+1)
Zp2=cancel(Zb2)
#Calculation of partial fractions
pb1=apart(Yp1)
pb2=apart(Zp2)
#Output
print(Yp1,"=\n",pb1)
print("\n",Zp2,"=\n", pb2)
```

Listing 5.15 Partial Fraction Decomposition

Output

```
(11*s**2 + 12*s + 3)/(6*s**3 + 11*s**2 + 6*s + 1) =
1/(3*s + 1) + 1/(2*s + 1) + 1/(s + 1)
(46*s**5 + 34*s**3 + 6*s)/(24*s**6 + 26*s**4 + 9*s**2 + 1) =
3*s/(4*s**2 + 1) + 2*s/(3*s**2 + 1) + s/(2*s**2 + 1)
```

Evaluation and Analysis

For calculating admittance, the program calculates the three partial fractions:

$$Y(s) = \frac{1}{3s+1} + \frac{1}{2s+1} + \frac{1}{s+1}$$

By comparing the coefficients, you should obtain the following results for the inductances:

$L_1 = 3\text{ H}, \qquad L_2 = 2\text{ H}, \qquad L_3 = 1\text{ H}$

For calculating impedance, the program calculates the three partial fractions:

$$Z(s) = \frac{3s}{4s^2+1} + \frac{2s}{3s^2+1} + \frac{s}{2s^2+1}$$

By comparing the coefficients, you should obtain the following results for the inductances and capacitances:

$$L_1 = 3\text{ H},\quad C_1 = \frac{4}{3}\text{F},\quad L_2 = 2\text{ H},\quad C_2 = \frac{3}{2}\text{ F},\quad L_3 = 1\text{ H},\quad C_3 = 2\text{ F}$$

In lines 05 and 08, three partial fractions are provided in each case so you can more easily check the result of the partial fraction calculation step. In lines 06 and 09, these partial fractions are multiplied with the `cancel()` method in such a way that the two fractional rational functions $Y(s)$ and $Z(s)$ are obtained as a result. In lines 11 and 12, partial fraction decomposition is then performed using the `apart()` method. The outputs in lines 14 and 15 confirm the expected results.

5.7 Continued Fractions

Let's assume you need an ohmic resistor of exactly 37/14Ω, but you only have resistors of 1Ω available. This problem can be solved with a catenary circuit of six resistors, as shown in Figure 5.6.

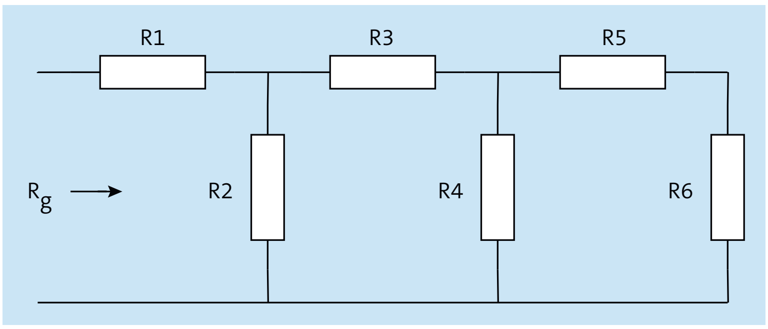


Figure 5.6 Continued Fraction Circuit

From this circuit, you can read the following continued fraction:

$$R_g = R_1 + \cfrac{1}{\cfrac{1}{R_2} + \cfrac{1}{R_3 + \cfrac{1}{\cfrac{1}{R_4} + \cfrac{1}{R_5 + R_6}}}}$$

You can calculate the continued fraction for 37/14 using the following Euclidean algorithm:

$$a = 37 = 2 \cdot 14 + 9$$
$$b = 14 = 1 \cdot 9 + 5$$
$$9 = 1 \cdot 5 + 4$$
$$5 = 1 \cdot 4 + 1$$
$$4 = 4 \cdot 1$$

This results in the continued fraction for the following:

$$\frac{37}{14} = 2 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \frac{1}{4}}}}$$

Or written in short notation:

$$\frac{37}{14} = [2; 1, 1, 1, 4]$$

By comparing the continued fraction decomposition of the circuit shown in Figure 5.6 with the continued fraction representation, you'll obtain the values for the resistors, as listed in Table 5.6.

R_1	R_2	R_3	R_4	R_5	R_6
2	1/1	1	1/1	2	2

Table 5.6 Resistance Values for the Catenary Circuit

Listing 5.16 uses the Euclidean algorithm to calculate the continued fraction for 37/14 and, from the coefficients of the continued fraction again, calculates the following continued fraction.

```python
#16_continued_fraction1.py
from sympy import *
z=37 #numerator
n=14 #denominator
#Calculate continued fraction
def kb(z,n):
    r=[]
    while n>0:
        r.append(z//n)
        z=z%n
        z,n=n,z
    return r
#Conversion to fraction
def ikb(ls):
    a = Integer(0)
    for i in reversed(ls[1:]):
        a=a+i
        a=1/a
    return ls[0] + a
#calculates continued fraction
kb1=kb(z,n)
```

```
22  #calculates fraction
23  ikb1=ikb(kb1)
24  #Output
25  print("Coefficients:",kb1)
26  print("Fraction:",ikb1)
```

Listing 5.16 Algorithms for Continued Fraction Decomposition

Output

```
Coefficients: [2, 1, 1, 1, 4]
Fraction: 37/14
```

Analysis

In lines 06 to 12, a function is defined that calculates the coefficients for a continued fraction according to the Euclidean algorithm. When calling function `kb(z,n)` in line 21, the numerator `z` and the denominator `n` of the fraction are passed as parameters. The `ikb(ls)` defined in lines 14 to 19 calculates the fraction again from the coefficients of the continued fraction.

SymPy facilitates the calculation of a continued fraction to a great extent. Listing 5.17 uses the `continued_fraction_periodic(z,n)` method to calculate the coefficients of a continued fraction, while it uses the `continued_fraction_reduce(continued fraction)` method from the coefficients to calculate again the continued fraction.

```
#17_continued_fraction2.py
from sympy import *
z=37 #numerator
n=14 #denominator
#Calculate continued fraction
kb=continued_fraction_periodic(z,n)
#Calculate fraction
fraction=continued_fraction_reduce(kb)
#Output
print("Coefficients:",kb)
print("Fraction:",fraction)
```

Listing 5.17 Continued Fraction Decomposition Using SymPy

Output

```
Coefficients: [2, 1, 1, 1, 4]
Fraction: 37/14
```

Analysis

The numerator z and the denominator n of the fraction are passed to the `continued_fraction_periodic(z,n)` method in line 06. The result of the continued fraction decomposition step is stored in the `kb` object. This object is passed to the `continued_fraction_reduce(kb)` method in line 08. The outputs confirm the expected result.

5.8 Limits

The concept of *limits* is fundamental to the understanding of differential and integral calculus. By calculating the limit, the slope of the tangent (first derivative) can be calculated in general from the secant slope of a function. Many antiderivatives (integrals) can also be calculated from the limits for the upper and lower sums. SymPy calculates the limit of a sequence or function using the `limit(f,x,g)` method. The `f` object stands for a mathematical sequence or function; `x` is the independent variable; and, for `g`, the limit can be specified.

5.8.1 Limits of Sequences

Let's use SymPy to calculate the limits of the five sequences shown in Table 5.7.

n	1	2	3	4	5	6	7	8	9	10
a_n	1	4	6	8	10	12	14	16	18	20
b_n	1	4	9	16	25	36	49	64	81	100
c_n	1	0.5	0.33	0.25	0.2	0.16	0.143	0.125	0.111	0.1
d_n	-1.33	2.57	1.81	1.73	1.74	1.75	1.78	1.8	1.81	1.828
e_n	2	2.25	2.37	2.44	2.49	2.52	2.55	2.57	2.58	2.59

Table 5.7 Sample Sequences

The sequences listed in Table 5.7 are in accordance with the following formation laws:

$$a_n = 2n$$

$$b_n = n^2$$

$$c_n = \frac{1}{n}$$

$$d_n = \frac{2n^2 + 2}{n^3 + n^2 - 5}$$

$$e_n = \left(1 + \frac{1}{n}\right)^n$$

An exception is the last sequence:

$$e_n = \left(1 + \frac{1}{n}\right)^n$$

For this sequence, the formation law cannot be read from the value table. The limit of this sequence for $n \to \infty$ is the Eulerian number e.

$$e = \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n \approx 2.7182818284590452354 \ldots$$

Listing 5.18 calculates the limits of the following sequences: a_n, b_n, c_n, d_n, and e_n.

```
01  #18_limit1.py
02  from sympy import *
03  n=symbols("n")
04  #Sequences
05  a1=2*n
06  a2=n**2
07  a3=1/n
08  a4=(2*n**3+2)/(n**3+n**2-5)
09  a5=(1+1/n)**n
10  #Output
11  print("Limits for n towards ∞")
12  print("Limit of %s is: %s" %(a1,limit(a1,n,oo)))
13  print("Limit of %s is: %s" %(a2,limit(a2,n,oo)))
14  print("Limit of %s is: %s" %(a3,limit(a3,n,oo)))
15  print("Limit of %s is: %s" %(a4,limit(a4,n,oo)))
16  print("Limit of %s is: %s" %(a5,limit(a5,n,oo)))
```

Listing 5.18 Limits of Sequences

Output

```
Limits for n towards ∞
Limit of 2*n is: oo
Limit of n**2 is: oo
Limit of 1/n is: 0
Limit of (2*n**3 + 2)/(n**3 + n**2 - 5) is: 2
Limit of (1 + 1/n)**n is: E
```

Analysis

In lines 12 to 16, the limits of the sequences defined in lines 05 to 09 are calculated and output using the `limit(a1,n,oo)` method. As a first parameter, this method expects the name of the sequence, the second parameter specifies the variable of the limit, and the third parameter specifies against which limit the sequence should strive. The outputs confirm the expected results.

5.8.2 Limits of Functions

SymPy can also calculate the limits of functions. Now, let's calculate the limits for the following functions:

$$\lim_{x\to 0}\frac{\sin x}{x}=1$$
$$\lim_{x\to 0}\frac{\tan x}{x}=1$$
$$\lim_{x\to \infty} a(1-e^{-x})=a$$
$$\lim_{x\to \infty}\frac{1}{x+2}=2$$
$$\lim_{x\to 1}\frac{x^2-1}{x-1}=2$$

Listing 5.19 calculates the limits for the listed functions.

```
01  #19_limit2.py
02  from sympy import *
03  a,x=symbols("a x")
04  #Functions
05  y1=sin(x)/x
06  y2=tan(x)/x
07  y3=a*(1-exp(-x))
08  y4=1/x+2
09  y5=(x**2-1)/(x-1)
10  #Output
11  print("Limit of %s toward 0 is: %s" %(y1,limit(y1,x,0)))
12  print("Limit of %s toward 0 is: %s" %(y2,limit(y2,x,0)))
13  print("Limit of %s toward ∞ is: %s" %(y3,limit(y3,x,oo)))
14  print("Limit of %s toward ∞ is: %s" %(y4,limit(y4,x,oo)))
15  print("Limit of %s toward 1 is: %s" %(y5,limit(y5,x,1)))
```

Listing 5.19 Limits of Functions

Output

```
Limit of sin(x)/x toward 0 is: 1
Limit of tan(x)/x toward 0 is: 1
Limit of a*(1 - exp(-x)) toward ∞ is: a
Limit of 2 + 1/x toward ∞ is: 2
Limit of (x**2 - 1)/(x - 1) toward 1 is: 2
```

Analysis

In lines 05 to 09, those functions are defined whose limits must be calculated using the `limit()` method. The arguments are passed in the same order as the sequences. The outputs in lines 11 to 15 again confirm the expected results.

5.8.3 Differential Quotient

For the following functions, we'll use SymPy to calculate the differential quotient by computing the limits from the difference quotient:

$$
\begin{aligned}
(x^n)' &= \lim_{h\to 0}\frac{(x+h)^n - x^n}{h} = nx^{n-1}\\
(a^x)' &= \lim_{h\to 0}\frac{a^{x+h} - a^x}{h} = a^{xlna}\\
(\sin x)' &= \lim_{h\to 0}\frac{\sin(x+h) - \sin x}{h} = \cos x\\
(\sinh x)' &= \lim_{h\to 0}\frac{\sinh(x+h) - \sinh hx}{h} = \cosh x\\
(\sin x\cos x)' &= \lim_{h\to 0}\frac{\sin(x+h)\cos(x+h) - \sin x\cos x}{h} = \cos 2x
\end{aligned}
$$

Listing 5.20 shows the implementation.

```
01  #20_limit3.py
02  from sympy import *
03  a,x,h,n=symbols('a x h n')
04  f1_1=((x+h)**n-x**n)/h      #power rule
05  f1_2=(a**(x+h)-a**x)/h      #exponential function
06  f1_3=(sin(x+h)-sin(x))/h    #trigonometric function
07  f1_4=(sinh(x+h)-sinh(x))/h  #hyberbolic function
08  f1_5=(sin(x+h)*cos(x+h)-sin(x)*cos(x))/h #product rule
09  #Output
10  print("Limits for h toward zero")
11  print("limes",f1_1," = ",simplify(limit(f1_1,h,0)))
12  print("limes",f1_2," = ",simplify(limit(f1_2,h,0)))
13  print("limes",f1_3," = ",simplify(limit(f1_3,h,0)))
14  print("limes",f1_4," = ",simplify(limit(f1_4,h,0)))
15  print("limes",f1_5," = ",simplify(limit(f1_5,h,0)))
```

Listing 5.20 Limits of Difference Quotients

Output

```
limes (-x**n + (h + x)**n)/h  =  n*x**(n - 1)
limes (-a**x + a**(h + x))/h  =  a**x*log(a)
limes (-sin(x) + sin(h + x))/h  =  cos(x)
limes (-sinh(x) + sinh(h + x))/h  =  cosh(x)
limes (-sin(x)*cos(x) + sin(h + x)*cos(h + x))/h  =  cos(2*x)
```

Analysis

Lines 04 to 08 define five difference quotients known from school math. The `limit()` method calculates the limits of these difference quotients for h toward zero in lines 11 to 15. These outputs have been simplified using `simplify()` but confirm the expected results.

5.9 Differentiation

SymPy can calculate the derivatives of functions using the `diff(f,x,k)` method, where the `f` object is the function to be derived, `x` is defined as the independent variable, and the natural number `k` stands for the *k*-th derivative of the function to be differentiated. If `k` does not occur, SymPy calculates the first derivative. Listing 5.21 calculates the first derivative for each of the functions.

$$y_1 = x^4 - 3x^3 + x^2 - 20$$

$$y_2 = \sin x \cos x$$

$$y_3 = \frac{x^3 - 4x + 3}{x + 4}$$

It also calculates the first, second and third derivatives for the following function:

$$y_4 = Ae^{-ax} \sin bx$$

```
#21_differential.py
from sympy import *
x,a,b,A=symbols("x a b A")
y1=x**4-3*x**3+x**2-20  #power rule
y2=sin(x)*cos(x)        #product rule
y3=(x**3-4*x+3)/(x+4)   #quotient rule
y4=A*exp(-a*x)*sin(b*x) #chain rule
#Calculations and outputs
print("1st derivative of:",y1,"\n", diff(y1,x))
print("1st derivative of:",y2,"\n", diff(y2,x))
print("1st derivative of:",y3,"\n", diff(y3,x))
print("1st derivative of:",y4,"\n", diff(y4,x,1))
print("2nd derivative of:",y4,"\n", diff(y4,x,2))
print("3rd derivative of:",y4,"\n", diff(y4,x,3))
```

Listing 5.21 Differentiation

Output

```
1st derivative of: x**4-3*x**3+x**2-20
4*x**3 - 9*x**2 + 2*x
1st derivative of: sin(x)*cos(x)
-sin(x)**2 + cos(x)**2
1st derivative of: (x**3-4*x+3)/(x+4)
(3*x**2 - 4)/(x + 4) - (x**3 - 4*x + 3)/(x + 4)**2
1st derivative of: A*exp(-a*x)*sin(b*x)
-A*a*exp(-a*x)*sin(b*x) + A*b*exp(-a*x)*cos(b*x)
2nd derivative of: A*exp(-a*x)*sin(b*x)
A*(a**2*sin(b*x) - 2*a*b*cos(b*x) - b**2*sin(b*x))*exp(-a*x)
3rd derivative of: A*exp(-a*x)*sin(b*x)
```

```
A*(-a**3*sin(b*x) + 3*a**2*b*cos(b*x) + 3*a*b**2*sin(b*x) -
b**3*cos(b*x))*exp(-a*x)
```

Analysis

The function types defined in lines 04 to 07 were selected based on the fact that they represent known differentiation rules. Hidden from the user, however, is whether SymPy uses these rules at all. You can check from the results that SymPy has differentiated correctly. But checking the higher derivatives of the damped sine function `y4` requires some effort.

Alternatively, you can calculate the derivative of a function such as y=*f*(x) using the following statements:

```
>>> from sympy import *
>>> x=symbols('x')
>>> y=x**2
>>> y.diff(x)
2*x
>>> Derivative(y,x).doit()
2*x
```

The `Derivative(y,x)` method initially accepts the term `x**2` without checking whether the derivative can be calculated at all (*unevaluated derivative*). Only the `doit()` method checks (*evaluates*) whether the calculation can be performed, and the derivative is then calculated if necessary. This delayed execution of a symbolic computation operation is useful when a term can be simplified further.

5.9.1 Usage Example: Curve Sketching

If a mathematical function $y = f(x)$ is given and its principal shape in a Cartesian coordinate system is to be examined, the mathematician refers to this process as a *curve sketching*. The shape of a function can be roughly estimated from the points of intersection with the x- and y-axis, the local places for the minimums and maximums, the limits for x toward $\pm\infty$ and the inflection points. In the following example, I deliberately omitted a detailed curve sketching to keep the source code as clean as possible. Listing 5.22 calculates the zeros, the extremums, and the inflection points for the following polynomial:

$y = -x^4 + 20x^2 - 64$

The function plot can be displayed on the screen for verification.

```
#22_curve_sketching.py
from sympy import *
x=symbols("x")

```

```
05 def f(x):
06     #y=4*x**3-16*x
07     #y=x**3-x**2-4*x+4
08     y=-x**4+20*x**2-64
09     return y
10 #Derivatives
11 f_1=diff(f(x),x,1)
12 f_2=diff(f(x),x,2)
13 x0=solve(f(x),x) #zeros
14 xe=solve(f_1,x)  #extremums
15 xw=solve(f_2,x)  #inflection points
16 #Output
17 print(f(x))
18 print("Zeros:",x0)
19 print("Extremums:",xe)
20 print("Inflection points",xw)
21 #plot(f(x),(x,-5,5))
```

Listing 5.22 Curve Sketching

Output

```
-x**4 + 20*x**2 - 64
Zeros: [-4, -2, 2, 4]
Extremums: [0, -sqrt(10), sqrt(10)]
Inflection points [-sqrt(30)/3, sqrt(30)/3]
```

Analysis

In lines 05 to 09, the function to be examined is defined. If the commented-out functions are supposed to be tested, you merely need to remove the corresponding comments. In lines 11 and 12, the `diff` method calculates the first and second derivatives. Line 13 calculates the zeros of the polynomial using the `solve` method. In line 14, the extremums are calculated by setting the first derivative equal to zero. Line 15 calculates the inflection points by setting the second derivative equal to zero. A case distinction was deliberately omitted in favor of better clarity. The "exact" numbers of the outputs remind the user that the calculations were performed using a CAS.

5.10 Integrations

Integrations represent the inverse operations of differentiations. If you form the derivative of a function, then the original function is the antiderivative, that is, the integral of the derived function. However, the reversal is not always clear. As a result, there is not an antiderivative F for every function f.

Mathematically, this relationship can be described by rearranging the differential quotient according to $\mathrm{d}y$. For the derivation, the following applies:

$$f'(x) = \frac{\mathrm{d}y}{\mathrm{d}x}$$

By rearranging according to $\mathrm{d}y$ and integrating on both sides, you obtain the following:

$$\mathrm{d}y = f'(x)\mathrm{d}x \iff y = \int f'(x)\mathrm{d}x$$

The product $f'(x)\mathrm{d}x$ can be interpreted as the surface element $\mathrm{d}A$. By summing up (i.e., through integration), you obtain the total surface under a curve.

You need integral calculus not only to calculate surfaces under nonlinear function graphs but also to calculate line lengths and volumes and to solve differential equations.

5.10.1 Indefinite Integral

Listing 5.23 calculates the antiderivatives for the following five functions:

$$f_1(x) = \frac{1}{x}$$

$$f_2(x) = e^{-x}$$

$$f_3(x) = x^2 e^{-x}$$

$$f_4(x) = \sin x$$

$$f_5(x) = e^{-x} \sin x$$

```
#23_integral1.py
from sympy import *
x=symbols("x")
print("∫%sdx=%s" %(1/x,integrate(1/x)))
print("∫%sdx=%s" %(exp(-x),integrate(exp(-x))))
print("∫%sdx=%s" %(exp(-x)*x**2,integrate(exp(-x)*x**2)))
print("∫%sdx=%s" %(sin(x),integrate(sin(x))))
print("∫%sdx=%s" %(exp(-x)*sin(x),simplify(integrate(exp(-x)*sin(x)))))
#plot(exp(-x),integrate(exp(-x)),(x,0,10))
```

Listing 5.23 Indefinite Integrals

Output

```
∫1/xdx=log(x)
∫exp(-x)dx=-exp(-x)
∫x**2*exp(-x)dx=(-x**2 - 2*x - 2)*exp(-x)
∫sin(x)dx=-cos(x)
∫exp(-x)*sin(x)dx=-sqrt(2)*exp(-x)*sin(x + pi/4)/2
```

Analysis

SymPy calculates the indefinite integrals (antiderivatives) in lines 04 to 08 using the `integrate()` method. With the `plot` method, you can output the function graphs and the graphs of the antiderivative.

Alternatively, you can also calculate integrals using the following statements:

```
>>> from sympy import *
>>> x=symbols('x')
>>> y=x**2
>>> y.integrate(x)
x**3/3
>>> Integral(y).doit()
x**3/3
```

5.10.2 Definite Integral

Using the definite integral, you can calculate the surface area of the surface enclosed by the lower limit a, the line of the function graph, the upper limit b, and the x-axis intercept. Listing 5.24 calculates the surface area of the five functions from Listing 5.23 in the limits from $a = 1$ to $b = 2$. For further testing purposes, you can of course change the limits according to your individual requirements.

```
#24_integral2.py
from sympy import *
x=symbols("x")
a,b=1,2 #lower limits
F1=integrate(1/x,(x,a,b))
F2=integrate(exp(-x),(x,0,oo))
F3=integrate(exp(-x)*x**2,(x,0,oo))
F4=integrate(sin(x),(x,0,pi))
F5=integrate(exp(-x)*sin(x),(x,0,oo))
#Output
print("∫%s from %s to %s = %s" %(1/x,a,b,F1))
print("∫%s from 0 to ∞ = %s" %(exp(-x),F2))
print("∫%s from 0 to ∞ = %s" %(exp(-x)*x**2,F3))
print("∫%s from 0 to π = %s" %(sin(x),F4))
print("∫%s from 0 to ∞ = %s" %(exp(-x)*sin(x),F5))
#plot(exp(-x)*sin(x),(x,0,10))
```

Listing 5.24 Definite Integrals

Output

```
∫1/x from 1 to 2 = log(2)
∫exp(-x) from 0 to ∞ = 1
```

```
∫x**2*exp(-x) from 0 to ∞ = 2
∫sin(x) from 0 to π = 2
∫exp(-x)*sin(x) from 0 to ∞ = 1/2
```

Analysis

In lines 05 to 09, the definite integrals of the test functions are calculated using the `integrate(func,(x,a,b))` method. The `func` function is passed as the first argument, followed by the integration variable `x`, the lower limit `a`, and the upper limit `b`. The integration variable and the integration limits must be placed in parentheses. Again, this example shows that you can also use constants as integration limits, such as `pi` (π) and `oo` (infinity).

5.10.3 Usage Example: Stored Electrical Energy

A capacitor with capacitance C is connected in series with a resistor R to a voltage source with voltage U_0, as shown in Figure 5.7. For this example, we need to determine the electrical energy W_{el} stored in the capacitor.

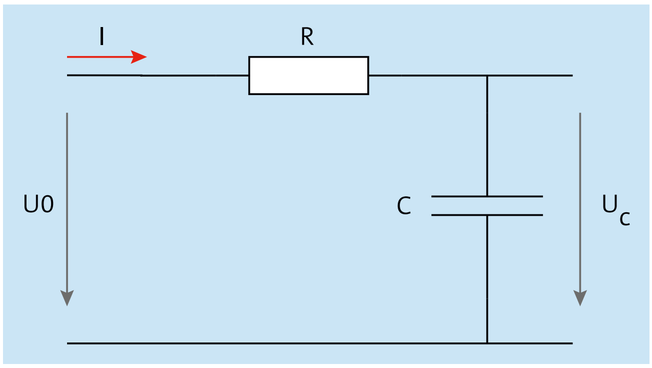


Figure 5.7 R-C Link

The following equation applies to the capacitor voltage u_c while charging:

$$u_c = U_0\left(1 - e^{-\frac{t}{\tau}}\right) \text{ with } \tau = RC$$

The capacitor current i_c decreases exponentially, according to the following formula:

$$i_c = \frac{U_0}{R}e^{-\frac{t}{\tau}}$$

The electrical power *p(t)* is the product of the voltage and current waveforms:

$$p(t) = u_c \cdot i_c$$

To calculate the stored electrical energy W_{el}, the surface area under the power curve must be calculated, according to the following formula:

$$W_{el} = \int_0^\infty p(t)\mathrm{d}t$$

Listing 5.25 calculates the stored electrical energy for a capacitor with a capacitance of 1F connected to a 10V voltage source.

```
01  #25_integral3.py
02  from sympy import *
03  t = symbols('t')
04  U0=10
05  R,C =1,1
06  I0=U0/R
07  tau=R*C
08  uc=U0*(1-exp(-t/tau))     #voltage curve
09  ic=I0*exp(-t/tau)         #current curve
10  p=uc*ic                   #electrical power
11  Wel=integrate(p,(t,0,oo)) #electrical energy
12  #Output
13  print("Stored el. energy:",Wel.evalf(3),"Ws")
14  plt=plot(uc,ic,p,(t,0,5*tau),show=False,legend=True)
15  plt[0].line_color = 'b'
16  plt[0].label='Voltage'
17  plt[1].line_color = 'r'
18  plt[1].label='Current'
19  plt[2].line_color = 'g'
20  plt[2].label='Power'
21  #plt.save('power.png')
22  plt.show()
```

Listing 5.25 Stored Energy of a Capacitor

Output

```
Stored el. energy: 50.0 Ws
```

Figure 5.8 shows what the output looks like in the function plot.

Analysis

In line 08, the voltage curve is stored in the `uc` object. In line 09, the current curve is stored in the `ic` object. To calculate the power curve, the only operation that needs to be performed is the multiplication `p=uc*ic` in line 10. Notice that—unlike what was necessary with NumPy—no array needs to be defined to make all power values available for the integration step in line 11. Also, the upper integration limit for this numerical problem is not five times the time constant, as is usually the case, but `oo` (∞). As a result, the program calculates the "exact" value for the stored electrical energy with 50 Ws without rounding errors.

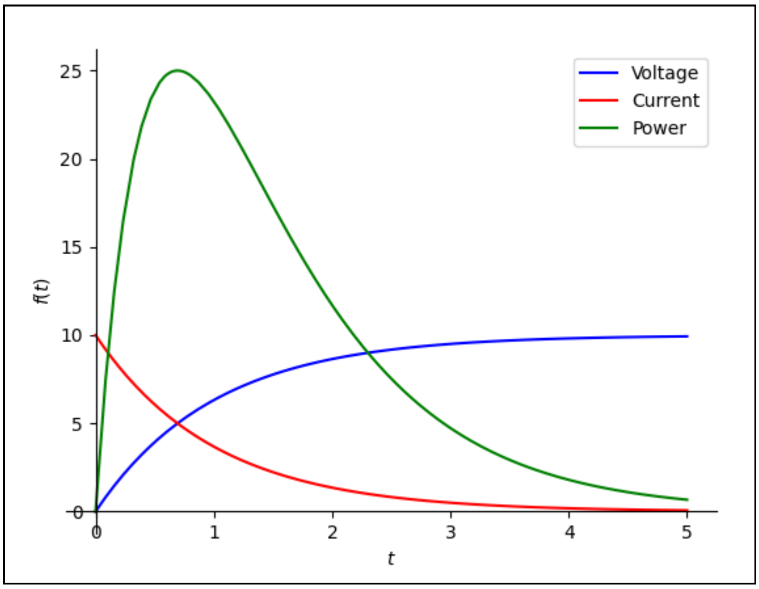


Figure 5.8 Stored Electrical Energy of a Capacitor

To illustrate this result, in line 14, the voltage, the current, and the power curve are stored in the `plt` variable. To display all three function plots in one diagram, you must set the `show=False` parameters in the `plot` method. Then, all three function plots will be saved into a list, and you can assign an individual color to each function plot. The stored electrical energy corresponds to the surface area below the green curve.

In line 21, you can save the graphic. Supported file formats include `eps`, `jpeg`, `jpg`, `pdf`, `pgf`, `png`, `ps`, `raw`, `rgba`, `svg`, `svgz`, `tif`, `tiff`, and `webp`.

5.11 Differential Equations

Ordinary differential equations contain the independent variable x and the dependent variable y as well as the derivatives $y^{(n)}(x)$ of the function, $y(x)$. The highest occurring derivative order determines the order of a differential equation. For a first-order differential equation, the formal notation is:

$$y'(x) + f(x) \cdot y(x) = g(x)$$

The function $f(x)$ can be any continuous function or a constant. The function $g(x)$ is referred to as a perturbation function.

A linear second-order differential equation with constant coefficients can be represented in the following way:

$$a_2 y''(x) + a_1 y'(x) + a_o y(x) = g(x)$$

The solution of a differential equation is a function. A differential equation can be solved by means of an integration operation. Since integration constants arise with each integration operation, theoretically, an infinite number of solution functions exist for any given differential equation. Solving a first-order differential equation results in *one* integration constant, and solving a differential equation of the nth order results in n integration constants. If the constants are not specified in more detail, mathematicians speak of a *general* solution to the differential equation. If the integration constants are determined by the initial conditions, the solution function $y(x)$ is a *special* solution of the differential equation. If the perturbation function $g(x)$ equals zero, then mathematicians refer to the differential equation as being *homogeneous*, otherwise as *inhomogeneous*.

Electrical networks, drive systems, and physical processes can be described via differential equations. If the input signal of a technical system and the differential equation of the system are known, the output signal can be calculated from it.

To describe a technical system in mathematical terms, you must use differential equations. This is because every non-trivial engineering system consists of energy stores, such as rotating masses with the moment of inertia J, capacitors with capacitance C, and coils with inductance L. The following laws apply to these energy stores:

$$M_b = J\frac{\mathrm{d}\omega}{\mathrm{d}t}, \quad i_C = C\frac{\mathrm{d}u_C}{\mathrm{d}t}, \quad u_L = L\frac{\mathrm{d}i}{\mathrm{d}t}$$

Thus, the listed energy stores differentiate the angular velocity $\omega(t)$ of the rotating mass, the voltage $u_c(t)$ across a capacitor, and the current $i(t)$ flowing through a coil. This relationship seems trivial. But real systems can be quite complex. The art of modeling a complex technical system consists of setting up the adequate differential equations. For this task, you need extensive expertise and a lot of hands-on experience. Once you have found the differential equation, SymPy does the often-tedious work of solving it for you. SymPy solves differential equations using the `dsolve(dgl,y)` method.

Note

SymPy can only solve *linear* differential equations.

5.11.1 Linear First-Order Differential Equations

In this section, I will describe the solution of a first-order differential equation using the example of discharging and charging a capacitor. A capacitor is charged to the voltage U_0 and then discharged through the resistor R.

At any time during the discharging process, the capacitor voltage must be equal to the voltage drop across the resistor:

$$u_c = R \cdot i$$

For the current *i(t)*, the capacitor current can be used:

$$i = -C\frac{\mathrm{d}u_c}{\mathrm{d}t}$$

When the capacitor is discharged, the current flows in the opposite direction than during the charging process, which is why the current receives a negative sign. By substituting the capacitor current in the output equation, you can obtain a first-order differential equation:

$$RC\frac{\mathrm{d}u_c}{\mathrm{d}t} = -u_c$$

By rearranging and using the time constant $\tau = RC$, we obtain the following equation:

$$\frac{\mathrm{d}u_c}{u_c} = -\frac{1}{\tau}\mathrm{d}t$$

Through integration on both sides, you get the following equation:

$$\ln u_c = -\frac{1}{\tau}t + K_1$$

The integration constant K_1 must not become zero because it represents the initial condition U_0. You must exponentiate both sides of the equation with the base *e*, and you'll get the general solution of the differential equation with the new integration constant K:

$$u_c = e^{-\frac{1}{\tau}t+K_1} = e^{K_1}e^{-\frac{1}{\tau}t} = Ke^{-\frac{1}{\tau}t}$$

At the point at which the discharging operation starts, the capacitor is charged to the voltage U_0. Thus, for the initial condition, $u_c(0) = U_0$ applies:

$$U_0 = Ke^0 = K$$

This results in the special solution of the differential equation for:

$$u_c = U_0 e^{-\frac{1}{\tau}t}$$

If you connect a series circuit of capacitor and resistor to a voltage source with voltage U_0, you obtain the following differential equation:

$$RC\frac{\mathrm{d}u_c}{\mathrm{d}t} + u_c = U_0$$

In the language of mathematics, the voltage U_0 is also referred to as the *perturbation function*. Separating the variables results in the following equation:

$$\frac{\mathrm{d}u_c}{U_0 - u_c} = \frac{1}{\tau}\mathrm{d}t$$

Then, both sides are integrated, as in the following example:

$$-\ln|U_0 - u_c| = \frac{1}{\tau}t + K_1$$

By exponentiating on both sides, you can thus obtain the general solution of the differential equation:

$$u_c = U_0 - e^{-\frac{1}{\tau}t+K_1} = U_0 - e^{K_1}e^{-\frac{1}{\tau}t} = U_0 - Ke^{-\frac{1}{\tau}t}$$

And for the initial condition $u_c(0) = 0$, you can obtain the special solution:

$$u_c = U_0\left(1 - e^{-\frac{1}{\tau}t}\right)$$

For any first-order linear differential equation with perturbation function $g(x)$

$$\frac{\mathrm{d}y}{\mathrm{d}x} = f(x)\cdot y + g(x)$$

you can find the following general solution formula in the literature:

$$y(x) = e^{F(x)}\cdot\int g(x)\cdot e^{-F(x)}\,\mathrm{d}x$$

Note that, in this context, F is an antiderivative of f.

Listing 5.26 calculates the voltage $u_c(t)$ on the capacitor for the discharging process and for the cases that the RC series circuit is connected to a voltage source with the following features:

- Constant voltage (voltage jump, step function)
- Linearly increasing voltage
- Exponentially increasing voltage
- Sinusoidal excitation

These voltage sources represent the perturbation functions of the differential equation and are called perturbation elements.

```
#26_first_order_differential_equation.py
from sympy import *
t=symbols("t")
u=Function("f")(t)
R,C=1,2
U0=10
tau=R*C
#first-order differential equation
dgl1=tau*Derivative(u,t)+u
dgl2=tau*Derivative(u,t)+u-U0
dgl3=tau*Derivative(u,t)+u-U0*t
dgl4=tau*Derivative(u,t)+u-U0*exp(t)
dgl5=tau*Derivative(u,t)+u-U0*sin(t)
```

```
14  #solution of the differential equation
15  L1=dsolve(dgl1,u)
16  L2=dsolve(dgl2,u)
17  L3=simplify(dsolve(dgl3,u))
18  L4=dsolve(dgl4,u)
19  L5=simplify(dsolve(dgl5,u))
20  #Output of the solution
21  print(L1,"\n",L2,"\n",L3,"\n",L4,"\n",L5)
```

Listing 5.26 Solutions of a First-Order Differential Equation with Different Perturbation Functions

Output

```
Eq(f(t), C1*exp(-t/2))
Eq(f(t), C1*exp(-t/2) + 10)
Eq(f(t), C1*exp(-t/2) + 10*t - 20)
Eq(f(t), (C1 + 10*exp(3*t/2)/3)*exp(-t/2))
Eq(f(t), C1*exp(-t/2) + 2*sin(t) - 4*cos(t))
```

Analysis

In line 04, the `Function("f")(t)` method specifies that the dependent variable `u` should be a function of time `t`. In lines 09 to 13, the differential equations are defined and assigned to the `dgl1` to `dgl5` objects. The definition of a differential equation is performed using the `Derivative(u,t)` method. This method represents the first derivative of the differential equation. The independent variable `u` is noted as the first parameter and the dependent variable `t` is noted as the second parameter. The first derivative is followed by the independent variable `u`. Thus, SymPy's syntax formally corresponds to the usual mathematical notation for a differential equation. The perturbation functions must all have a negative sign because they have been moved from the right-hand side of the differential equation to the left-hand side.

In lines 15 to 19, the `dsolve(dgl,u)` method solves the five differential equations. This method is passed the `dgl` object as the first parameter. Instead of the `dgl` object, the complete notation of the differential equation could also be written in this case. As a second parameter, `dsolve()` expects the independent variable `u`. The solutions are stored in objects `L1` to `L5`.

The outputs (line 21) confirm the expected solutions, which are performed using the `Eq(f(t),...)` method, where the abbreviation `Eq` stands for *equation*. `C1` stands for the integration constant—not be confused with a capacitance *C*. A procedure for determining this integration constant is described next.

5.11.2 General Solution of a Second-Order Differential Equation

Let's now use the example of a series resonant circuit to demonstrate the general solution of a second-order differential equation, as shown in Figure 5.9.

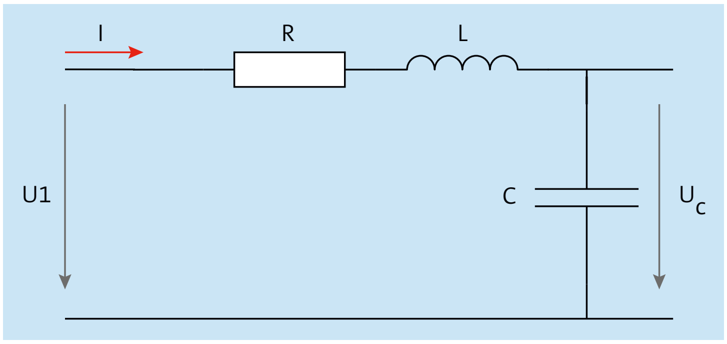


Figure 5.9 Series Resonant Circuit

Using the mesh rule for voltage drops, we can read the following from the circuit:

$$u_L + u_R + u_c = U_1$$

Due to the law of induction and the voltage drop across the ohmic resistor, we obtain the following:

$$L\frac{\mathrm{d}i}{\mathrm{d}t} + R \cdot i + u_c = U_1$$

By inserting the capacitor current, you obtain a second-order differential equation:

$$LC\frac{\mathrm{d}^2 u_c}{\mathrm{d}t^2} + RC\frac{\mathrm{d}u_c}{\mathrm{d}t} + u_c = U_1$$

If we divide both sides by LC, we obtain:

$$\frac{\mathrm{d}^2 u_c}{\mathrm{d}t^2} + \frac{R}{L}\frac{\mathrm{d}u_c}{\mathrm{d}t} + \frac{1}{LC}u_c = \frac{U_1}{LC}$$

You can use the characteristic equation to find the general solution of the differential equation:

$$\lambda_{1,2} = -\frac{R}{2L} \pm \sqrt{\frac{R^2}{4L^2} - \frac{1}{LC}}$$

The homogeneous solution is:

$$u_{ch} = K_1 e^{\lambda_1 t} + K_2 e^{\lambda_2 t}$$

For the inhomogeneous solution, if the perturbation function $g(x)$ is a constant K, then:

$$u_{cinh} = \frac{K}{a_0} = \frac{\frac{U_1}{LC}}{\frac{1}{LC}} = U_1$$

The special solution is the total of the homogeneous and the inhomogeneous solutions:

$$u_c = K_1 e^{\lambda_1 t} + K_2 e^{\lambda_2 t} + U_1$$

Depending on how large the values for R, L and C turn out, you have to distinguish three cases:

- **Case 1**: If the following applies:

  $$\frac{R^2}{4L^2} > \frac{1}{LC}$$

  Then the differential equation gets the following general solution:

  $$u_c = K_1 e^{\lambda_1 t} + K_2 e^{\lambda_2 t} + U_1$$

- **Case 2**: If the following case arises:

  $$\frac{R^2}{4L^2} = \frac{1}{LC}$$

  Then the differential equation gets the general solution with the following:

  $$u_c = (K_1 + K_2 t)e^{\lambda_1 t} + U_1$$

  $$\lambda_1 = -\frac{R}{2L}$$

- **Case 3**: If the following applies:

  $$\frac{R^2}{4L^2} < \frac{1}{LC}$$

  Then the differential equation looks as follows where parameter a must always have a negative value:

  $$u_c = e^{at} \cdot [K_1 \sin bt + K_2 \cos bt] + U_1$$

Listing 5.27 calculates the voltage across the capacitor for a series resonant circuit for all three cases:

```
#27_second_order_differential_equation.py
from sympy import *
t=symbols("t")
u=Function("f")(t)
U1=10 #input voltage
R1,L1,C1=5/2,1,1
R2,L2,C2=2,1,1
R3,L3,C3=2,1/2,1/4
#second-order differential equation
dgl1=L1*C1*Derivative(u,t,t)+R1*C1*Derivative(u,t)+u-U1
dgl2=L2*C2*Derivative(u,t,t)+R2*C2*Derivative(u,t)+u-U1
dgl3=L3*C3*Derivative(u,t,t)+R3*C3*Derivative(u,t)+u-U1
#solution of the differential equation
L1=dsolve(dgl1,u)
```

```
15  L2=dsolve(dgl2,u)
16  L3=dsolve(dgl3,u)
17  #Output of the solution
18  print(L1,"\n",L2,"\n",L3)
```

Listing 5.27 General Solution of a Second-Order Differential Equation

Output

```
Eq(f(t),  C1*exp(-2.0*t) + C2*exp(-0.5*t) + 10)
Eq(f(t), (C1 + C2*t)*exp(-t) + 10)
Eq(f(t), (C1*sin(2.0*t) + C2*cos(2.0*t))*exp(-2.0*t) + 10)
```

Analysis

In lines 06 to 08, the values for R, C, and L were chosen so that all three possible solutions occur. To make clear that the solution of the differential equation is a symbolic procedure, I deliberately avoided choosing real numbers.

The definitions of the differential equations in lines 10 to 12 follow the same pattern as shown in Listing 5.26, the only difference being that, for the second derivative in the `Derivative(u,t,t)` method, the independent variable `t` must occur twice.

The calculated solutions meet the expectations. `C1` and `C2` are again the integration constants.

5.11.3 Special Solution of a Second-Order Differential Equation

The solution set pertaining to the general solution of a differential equation can theoretically be infinitely large. However, practitioners are generally interested in the special solution of a differential equation. Such a special solution is defined by *initial conditions*. For example, was electrical or magnetic energy stored in the capacitor or coil before the interference element $g(t)$ was connected? If not the case, then all initial conditions have the value zero.

The differential equation for an oscillating circuit can be put into a generally valid form with the damping D and the angular frequency ω_0 in the following way:

$$\frac{\mathrm{d}^2 u_c}{\mathrm{d}t^2} + 2D\omega_0\frac{\mathrm{d}u_c}{\mathrm{d}t} + \omega_0^2 u_c = g(t)$$

This differential equation, referred to as a harmonic oscillator, also applies to a damped spring pendulum oscillating either horizontally or vertically. For better comprehensibility, in the differential equation that describes the motion of a spring pendulum, you should replace the u_c variable by x for a horizontal or by y for a vertical motion of the mass.

In real life, voltage jumps or periodic functions occur as perturbation functions $g(t)$.

By comparing coefficients, you can obtain the following result for the angular frequency:

$$\omega_0 = \frac{1}{\sqrt{LC}}$$

You can also obtain the following result for damping:

$$D = \frac{R}{2\omega_0 L}$$

SymPy also provides a different syntax for the notation of the differential equation:

```
dgl=Eq(uc(t).diff(t,2)+2*D*w0*uc(t).diff(t,1)+w0**2*uc(t),Us)
```

If initial conditions must be taken into account, then you must pass the `ics={…}` parameter to the `dsolve()` method in addition to the `dgl` object:

```
dsolve(dgl,uc(t),ics=aw)
```

The abbreviation `ics` stands for *initial conditions*. SymPy sets the initial conditions with a dictionary:

```
aw={uc(0):0, uc(t).diff(t,1).subs(t,0):0}
```

For the differential equation solved using Listing 5.28, the initial conditions $\dot{u}_c(0) = 0$ and $u_c(0) = 0$ are valid.

```
#28_differential_equation_special_solution.py
from sympy import *
t=symbols("t")
uc=Function("uc")
U1=10
R=2
L=1/2
C=1/4
Us=U1/(L*C)       #perturbation function
w0=sqrt(1/(L*C)) #angular frequency
D=R/(2*L*w0)      #damping
dgl=Eq(uc(t).diff(t,2)+2*D*w0*uc(t).diff(t,1)+w0**2*uc(t),Us)
#initial values
aw={uc(0):0, uc(t).diff(t,1).subs(t,0):0}
ua_t=dsolve(dgl,uc(t))  #general solution
us_t=dsolve(dgl,uc(t),ics=aw)#special solution
uc_t=us_t.rhs
#plot(uc_t,(t,0,5))
#Output
print("general solution\n",ua_t)
```

```
21  print("special solution\n",us_t)
22  print("right side of function uc(t) =",uc_t)
```

Listing 5.28 Special Solution of a Second-Order Differential Equation

Output

```
general solution
 Eq(uc(t), (C1*sin(2.0*t) + C2*cos(2.0*t))*exp(-2.0*t) + 10.0)
special solution
 Eq(uc(t), (-10.0*sin(2.0*t) - 10.0*cos(2.0*t))*exp(-2.0*t) + 10.0)
right side of function
 uc(t) = (-10.0*sin(2.0*t) - 10.0*cos(2.0*t))*exp(-2.0*t) + 10.0
```

Analysis

In line 12, the differential equation is defined using the `diff()` method within the `Eq()` method and assigned to the `dgl` object. The searched `uc(t)` function is linked via the dot operator with the `diff()` method to `uc(t).diff(t,2)`. The first parameter is the independent variable `t`, and the second parameter specifies which derivative is meant. A 2 represents the second derivative, while a 1 represents the first derivative. The parameters for the damping and the angular frequency of the differential equation are adopted according to the mathematical representation. The perturbation function `Us` is the second parameter, separated by a comma, after the definition of the differential equation in the parentheses of the `Eq()` method. The initial values are set as a dictionary in line 14 and assigned to the `aw` object. For $t = 0$, the first derivative of the capacitor voltage and the voltage across the capacitor should be zero. `uc(0)` and `uc(t).diff(t,1).subs(t,0)` are the keys of the dictionary.

In line 16, the `dsolve(dgl,uc(t),ics=aw)` method solves the differential equation with the given initial values. The third parameter contains the initial values: The `ics` object is assigned the initial values `aw`, where `ics` is supposed to stand for *initial conditions*.

In line 17, `rhs` (*right hand side*) is used to determine the right-hand side of the function equation $_{uc} = f(t)$. If you remove the comment in line 18, you can also display the course of the function graphically.

In lines 20 and 21, the general and the special solutions are output. In line 22, the right-hand side of the function equation is output.

5.12 Laplace Transform

The *Laplace transform* is an integral transform which greatly simplifies the solving of differential equations and the analysis of AC networks. In both cases, you only need to set up algebraic equations according to the computational rules of the Laplace transform or according to the theorems of network theory in the image. The solution in the image is an algebraic function consisting of a numerator and denominator polynomial. This algebraic function is then transformed back into the time domain using correspondence tables.

5.12.1 Solving Differential Equations

Consider the following linear differential equation with constant third-order coefficients:

$$f'''(t) + Af''(t) + Bf'(t) + C = g(t)$$

Three steps are required to solve this equation and any other differential equation of this type:

1. Transforming the differential equation into the image
2. Solving the differential equation in the image
3. Transforming the image function back to the original domain

Step 1: Transforming the Differential Equation into the Image

The transformation of the derivatives and the perturbation function $g(t)$ into the image is performed using the improper integral, which performs the transformation from the time domain into the image domain:

$$\mathcal{L}\{f(t)\} = \int_0^\infty f(t) \cdot e^{-st} \mathrm{d}t = F(s)$$

The transformation of the first derivative is:

$$\mathcal{L}\{f'(t)\} = \int_0^\infty f'(t) \cdot e^{-st} \mathrm{d}t = sF(s) - f(0)$$

The transformation of the second derivative is:

$$\mathcal{L}\{f''(t)\} = \int_0^\infty f''(t) \cdot e^{-st} \mathrm{d}t = s^2F(s) - sf(0) - f'(0)$$

The transformation of the third derivative is:

$$\mathcal{L}\{f'''(t)\} = \int_0^\infty f'''(t) \cdot e^{-st} \mathrm{d}t = s^3F(s) - s^2f(0) - sf'(0) - f''(0)$$

Transformation of the Perturbation Function g(t)

Voltage sources with constant, linearly increasing, and sinusoidal voltage occur as a perturbation function in electrical networks. The step function $\sigma(t)$ is used particularly frequently in real life:

$$\sigma(t) = \begin{cases} 0 & \text{for } t \leq 0 \\ 1 & \text{for } t \geq 0 \end{cases}$$

The input of a (two-port) network is connected to a voltage source with voltage $U_1 = a$ at time $t = 0$. The course of the output voltage $U_2(t)$, called the step response, must now be calculated.

For a voltage jump $\sigma(t)$ the following applies:

$$\mathcal{L}\{a\} = \int_0^\infty a \cdot e^{-st} \mathrm{dt} = \frac{a}{s}$$

SymPy calculates these and other important transforms of perturbation functions in the following way:

```
>>> from sympy import *
>>> a,s,t = symbols("a,s,t")
>>> laplace_transform(a, t, s)
(a/s, 0, True)
>>> laplace_transform(a*t, t, s)
(a/s**2, 0, True)
>>> laplace_transform(exp(-a*t), t, s)
(1/(a + s), 0, Abs(arg(a)) < pi/2)
>>> laplace_transform(sin(a*t), t, s)
(a/(a**2 + s**2), 0, Eq(2*Abs(arg(a)), 0))
>>> laplace_transform(cos(a*t), t, s)
(s/(a**2 + s**2), 0, Eq(2*Abs(arg(a)), 0))
```

The perturbation functions calculated using SymPy are listed in Table 5.8.

Original Function $g(t)$	Image Function $F(s)$
a	$\frac{a}{s}$
$a \cdot t$	$\frac{a}{s^2}$
e^{-at}	$\frac{1}{s+a}$
$\sin at$	$\frac{a}{s^2+a^2}$
$\cos at$	$\frac{s}{s^2+a^2}$

Table 5.8 Correspondence Table for Transforming the Perturbation Functions into the Image Domain

Step 2: Solving the Differential Equation in the Image

The general solution for an ordinary third-order differential equation is the following fractional rational function:

$$F(s) = \frac{\mathcal{L}\{f(t)\} + s^2 f(0) + sf'(0) + f''(0)}{s^3 + As^2 + Bs + C}$$

Step 3: Transforming the Image Function Back into the Time Domain

The inverse Laplace transform $\mathcal{L}^{-1}$ is performed using the following inverse integral:

$$\mathcal{L}^{-1}\{F(s)\} = \frac{1}{2\pi j} \int_{c-j\infty}^{c+j\infty} F(s) \cdot e^{st} \mathrm{d}t = f(t)$$

SymPy calculates the inverse Laplace transform using the following method:

```
>>>inverse_laplace_transform(Fs,s,t)
```

Except for equation No. 9 from Table 5.9, all transforms from the image domain to the time domain were performed using SymPy and compared to the data from the literature.

No.	Image Function $F(s)$	Original Function $f(t)$
1	$\frac{a}{s}$	a
2	$\frac{a}{s^2}$	$a \cdot t$
3	$\frac{1}{s+a}$	e^{-at}
4	$\frac{a}{s(s+a)}$	$1 - e^{-at}$
5	$\frac{a}{s^2+a^2}$	$\sin at$
6	$\frac{s}{s^2+a^2}$	$\cos at$
7	$\frac{b}{(s+a)^2+b^2}$	$e^{-at} \sin bt$
8	$\frac{s+a}{(s+a)^2+b^2}$	$e^{-at} \cos bt$
9	$\frac{\omega_0^2}{s(s^2+2D\omega_0+\omega_0^2)}$	$1 - \frac{e^{-\delta t}}{\omega_e}(\delta \sin \omega_e t + \omega_e \cos \omega_e t)$
	with $\delta = D\omega_0$ and $\omega_e = \omega_0\sqrt{1-D^2}$ for $D < 1$	

Table 5.9 Correspondence Table for the Transformation into the Time Domain

SymPy calculates all transformations of the image functions 1 through 8 (see Table 5.9) without any problem in a runtime that's still acceptable. However, SymPy fails at the transformation of the third-degree polynomial (No. 9) with the symbolic variables D and ω_0. If, on the other hand, numerical values are used for the damping and the angular frequency, SymPy provides the correct result, as the following example shows.

Usage Example: Capacitor Voltage at the Series Resonant Circuit

A simple example will illustrate the transformation of the original domain (time domain) into the image domain. A series resonant circuit is connected to a voltage source with voltage U_1 = 8V. Thus, the step function is a constant for $t \geq 0$. We are searching the step response $u_c(t)$ for the voltage across the capacitor. According to the mesh rule, the following differential equation results:

$$\ddot{u}_c + \frac{R}{L}\dot{u}_c + \frac{1}{LC}u_c = \frac{U_1}{LC}$$

The voltage u_c is a function of time $u_c = f(t)$ in the original range (time domain).

This differential equation must now be transformed from the original domain to the image domain. The first step consists of defining the initial conditions. The following initial conditions are supposed to apply at time $t = 0$: $\ddot{u}_c(0) = 0$, $\dot{u}_c(0) = 0$, and $u_c(0) = 0$.

Next, the differential operators are replaced by the Laplace operators. For the second derivative, we use $s^2 U_C(s)$, for the first derivative $sU_C(s)$, and for u_c, we use $U_C(s)$. For each term, the corresponding initial condition is also written:

$$\mathcal{L}\left\{\frac{\mathrm{d}u_c^2}{\mathrm{d}t^2}\right\} = s^2 U_c(s) - s \cdot u_c(0) - \dot{u}_c(0) = s^2 U_c(s) - 0 - 0 = s^2 U_C(s)$$

$$\mathcal{L}\left\{\frac{R}{L}\frac{\mathrm{d}u_c}{\mathrm{d}t}\right\} = \frac{R}{L}sU_C(s) - u_c(0) = \frac{R}{L}sU_C(s) - 0 = \frac{R}{L}sU_C(s)$$

$$\mathcal{L}\left\{\frac{1}{LC}u_c\right\} = \frac{1}{LC}U_C(s)$$

$$\mathcal{L}\left\{\frac{U_1}{LC}\right\} = \frac{U_1}{LCs}$$

Putting the individual transforms together yields the image function of the differential equation:

$$s^2 U_C(s) + \frac{R}{L}sU_C(s) + \frac{1}{LC}U_C(s) = \frac{U_1}{LCs}$$

By rearranging, you obtain the image function with the values for the components R = 2Ω, L = ½H and C = ¼F:

$$U_C(s) = \frac{\frac{U_1}{LC}}{s\left(s^2 + \frac{R}{L}s + \frac{1}{LC}\right)} = \frac{10 \cdot 8}{s(s^2 + 4s + 8)}$$

The transformation back into the time domain is done using equation No. 9 from the correspondence table:

$$\mathcal{L}^{-1}\left\{\frac{\omega_0^2}{s(s^2+2D\omega_0+\omega_0^2)}\right\} = 1 - \frac{e^{-\delta t}}{\omega_e}(\delta \sin \omega_e t + \omega_e \cos \omega_e t)$$

The damping D is determined by means of a coefficient comparison:

$$D = \frac{4}{2\omega_0} = \frac{2}{\sqrt{8}} = \frac{\sqrt{2}}{2}$$

The following applies to the damping constant, δ :

$$\delta = D\omega_0 = \frac{\sqrt{2}}{2}\sqrt{8} = 2$$

And it applies to the natural frequency:

$$\omega_e = \omega_0\sqrt{1-D^2} = \sqrt{8}\sqrt{1-\left(\frac{\sqrt{2}}{2}\right)^2} = 2$$

By inserting it into the time function from the correspondence table, the time function is then obtained for the voltage curve at the capacitor:

$$u(t) = 10\text{ V}[1 - e^{-2t}(\sin 2t + \cos 2t)]$$

SymPy calculates the following for the transformation back into the time domain:

```
>>> from sympy import *
>>> s,t=symbols("s t", positive=True)
>>> Fs=80/(s*(s**2+4*s+8))
>>> LT_inv=inverse_laplace_transform(Fs,s,t)
>>> LT_inv
10 - 10*exp(-2*t)*sin(2*t) - 10*exp(-2*t)*cos(2*t)
>>> simplify(LT_inv)
10 - 10*sqrt(2)*exp(-2*t)*sin(2*t + pi/4)
```

5.12.2 Analyzing Networks with Transmission Functions

The step response of a two-port network can also be calculated without setting up a differential equation. Only the transmission function of the two-port network and the image function of the input voltage $U_1(s)$ are required. The transmission function $H(s)$ is the ratio between the output voltage and the input voltage, as in the following equation:

$$H(s) = \frac{U_2(s)}{U_1(s)}$$

By rearranging, you can obtain the output voltage $U_2(s)$ in the image domain:

$$U_2(s) = H(s) \cdot U_1(s)$$

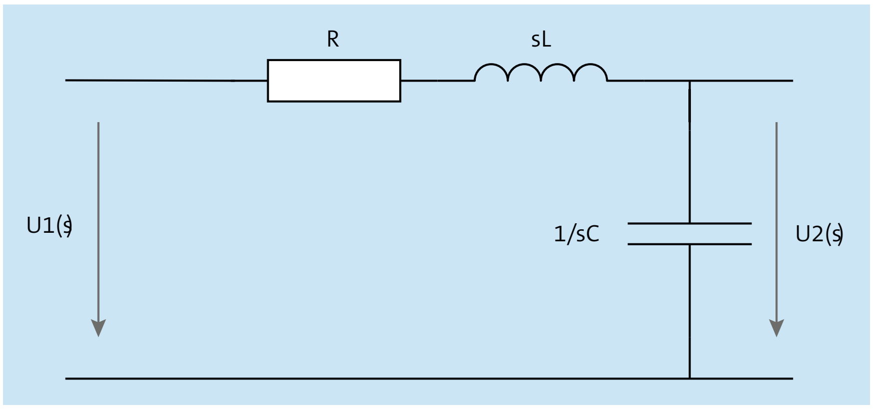


Figure 5.10 R-L-C Two-Port Network

Based on the voltage divider rule, the transmission function *H*(*s*) shown in Figure 5.10 can be set up with the following equation:

$$H(s) = \frac{\frac{1}{Cs}}{R + Ls + \frac{1}{Cs}} = \frac{1}{LCs^2 + RCs + 1} = \frac{1}{s^2 + \frac{R}{L}s + \frac{1}{LC}}$$

Using the input voltage in the image

$$U_2(s) = \frac{U_1}{s}$$

you obtain the output voltage for the image:

$$U_2(s) = \frac{U_1}{s} \cdot \frac{1}{s^2 + \frac{R}{L}s + \frac{1}{LC}}$$

In the image, the current *I*(*s*) can also be calculated using Ohm's law:

$$I(s) = \frac{U_1(s)}{Z(s)} = \frac{\frac{U_1}{s}}{R + Ls + \frac{1}{Cs}}$$

The transformation back into the time domain is performed using the following method:

```
inverse_laplace_transform(F(s),s,t)
```

Listing 5.29 calculates the step response of the output voltage and the capacitor current for the circuit from Figure 5.10 using the inverse Laplace transform method.

```
01  #29_inv_laplace1.py
02  from sympy import *
03  s,C,L,R = symbols("s C L R")
04  t = symbols("t",positive=True)
05  U1=10
```

```
06  R=2
07  L=Rational(1,2)
08  C=Rational(1,4)
09  U1_s=U1/s
10  Z_s=R+L*s+1/(C*s)
11  I_s=U1_s/Z_s
12  H_s=1/(R + L*s + 1/(C*s))/(C*s)
13  H_s=expand(H_s) #transmission function
14  U2_s=U1_s*H_s   #step response
15  uc=inverse_laplace_transform(U2_s,s,t)
16  ic=inverse_laplace_transform(I_s,s,t)
17  #Outputs
18  print("Transmission function\n",H_s)
19  print("Voltage at capacitor\n","uc =",simplify(uc))
20  print("Capacitor current\n","ic =",ic)
21  plt=plot(uc,ic,(t, 0, 5),show=False)
22  plt[0].line_color = 'b'
23  plt[1].line_color = 'r'
24  plt.show()
```

Listing 5.29 Inverse Laplace Transform

Output

```
Transmission function
 4/(s**2/2 + 2*s + 4)
Voltage at capacitor
 uc = 10 - 10*sqrt(2)*exp(-2*t)*sin(2*t + pi/4)
Capacitor current
 ic = 10*exp(-2*t)*sin(2*t)
```

Figure 5.11 shows the output as a function plot.

Analysis

In line 03, the symbols for the components and the variables for the time and image domains are defined. The `positive=True` parameter causes the suppression of the output of the `Heaviside(t)` expression. In line 05, the level of the voltage jump for the input voltage is set to 10V.

In lines 06 to 08, you can enter other values for the components. You can use only values of the `int` type. For example, if you assign the value `2.` to the `R` variable, the program will not be executed. If you want to assign rational numbers as values to the components, you must convert them into a fraction using the `Rational(numerator,denominator)` method. You can use `print(type(L))` to display the type of the variable `L`: `<class 'sympy.core.numbers.Half'>`.

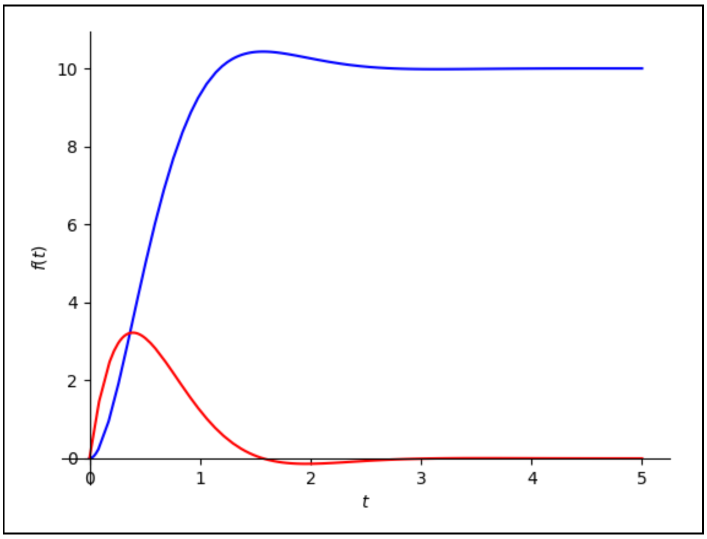


Figure 5.11 Step Response for an R-L-C Two-Port Network

Line 09 defines the step function for the image. Line 10 shows the total resistance in the image. The current that flows through all components in the image (series connection) is calculated in line 11 using Ohm's law. Line 12 contains the transmission function. It was deliberately not brought to the usual form as a fractional rational function to show that SymPy can cope with elementary terms (double fractions, partial fractions). In line 13, the `expand()` method converts the term from line 12 into the common fractional rational function. In line 14, the step response in the image is calculated. In lines 13 and 14, the transformation from the image domain to the time domain is performed using the `inverse_laplace_transform(F(s),s,t)` method.

The results calculated by the inverse Laplace transform for the current and voltage curves at the capacitor are also output as a function plot (lines 21 to 24).

This example clearly shows how effectively a step response can be simulated using the inverse Laplace transform: The transmission function is set up directly in the image domain according to the rules of network theory, multiplied by the image function 1/s for the unit step, and then transformed into the time domain using the `inverse_laplace_transform()` method. As elegant and effective as this procedure seems, it unfortunately comes up against SymPy's limited resources, as the next project task example shows.

5.13 Project Task: Step Response of a Catenary Circuit

For the catenary circuit shown in Figure 5.12, which consists of three capacitors (as cross links) and two coils (as longitudinal links), we want to calculate the transmission function and the step response of the output voltage using the Laplace transform method, which means that the circuit is a fifth-order low-pass filter. For the internal resistance of the voltage source and the terminating resistor, $R = 1\Omega$ is specified in each case.

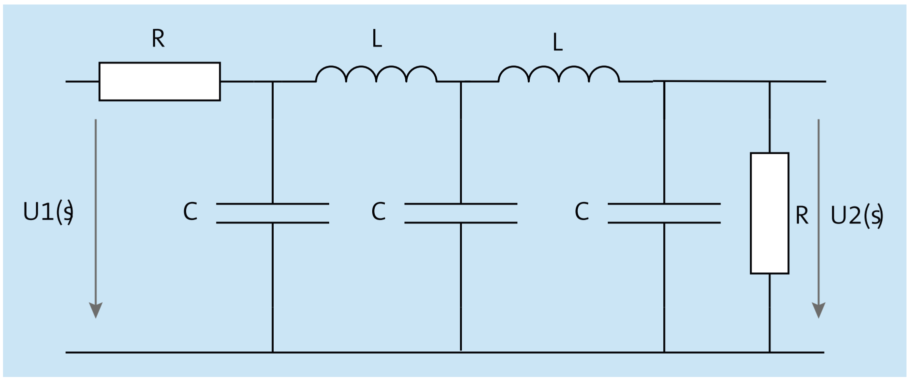


Figure 5.12 L-C Catenary Circuit as a Fifth-Degree Low-Pass Filter

Listing 5.30 calculates the transmission function for the circuit shown in Figure 5.12 using symbolic matrix multiplication from the elementary two-port elements of the catenary circuit (longitudinal and cross links). SymPy calculates the course of the output voltage (step response) using the inverse Laplace transform.

```
#30_inv_laplace2.py
from sympy import *
s,C,L,R = symbols("s C L R")
t = symbols("t",positive=True)
#Values of the components
R=1
L=5
C=10
#Matrixes of the longitudinal and cross links
A1=Matrix([[1, R],
           [0, 1]])
A2=Matrix([[1,  0],
           [C*s,1]])
A3=Matrix([[1,L*s],
           [0, 1]])
A4=Matrix([[1,   0],
           [C*s, 1]])
A5=Matrix([[1, L*s],
           [0,   1]])
A6=Matrix([[1,  0],
           [C*s,1]])
A7=Matrix([[1,  0],
           [1/R,1]])
```

```
#Matrix multiplication
A=A1*A2*A3*A4*A5*A6*A7
#Transmission function
H_s=1/A[0,0]
U2_s=H_s/s
u2=inverse_laplace_transform(U2_s,s,t)
#Outputs
print("Transmission function\n",expand(H_s))
plot(u2,(t,0,100))
```

Listing 5.30 Step Response of a Catenary Circuit

Output

```
Transmission function
1/(25000*s**5+5000.0*s**4+2250.0*s**3+300.0*s**2+40.0*s+2.0)
```

A graphical representation of the step response is shown in Figure 5.13.

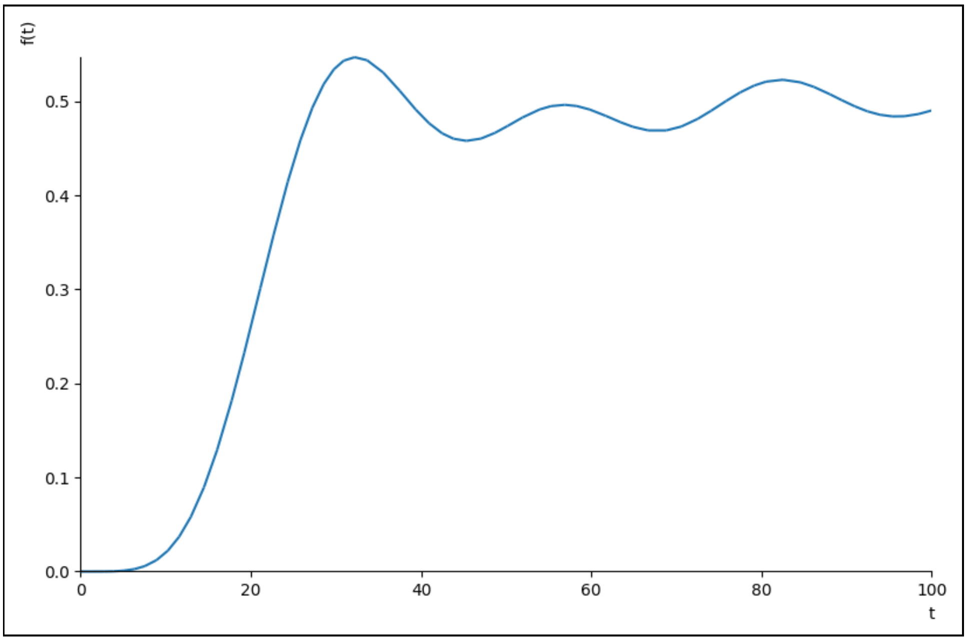


Figure 5.13 Step Response for a Fifth-Degree Low-Pass Filter

Analysis

The circuit consists of three capacitors as cross links and two coils as longitudinal links, which can store electrical energy and magnetic energy, respectively. Thus, the denominator polynomial of the transmission function is a fifth-degree polynomial.

This project task combines all the programming techniques covered so far: the symbolic multiplication of matrixes and the inverse Laplace transform. The matrixes of the A parameters for the longitudinal and cross links are defined in lines 10 to 23. The multiplication is performed in line 25. For the calculation of the transmission function in line 27, only the `A[0,0]` parameter is needed. In line 28, the transmission function is multiplied by the step function of the unit step, `1/s`. The transformation to the time domain is carried out in line 29.

When running the program, the long translation time shows that SymPy has almost reached its limits when transforming a fifth-degree transmission function into the time domain.

5.14 Project Task: Bending a Beam That Is Fixed at One End

For a beam of length l that is fixed at one end, we want to calculate the deflection $w = f(x)$. The deflection (bending line) depends on the cross-section and length of the beam, the elastic modulus E of the material and the force F that's acting on the beam. The influence of the cross-section on the deflection is determined by the second axial moment of area, I_y. The second moment of area and the deflection of a beam are supposed to be computed symbolically using SymPy. The deflection should also be represented as a function graph.

5.14.1 Second Moment of Area

The second moment of area determines how the shape of the cross-sectional area of a beam (girder) affects its stiffness. Figure 5.14 shows a beam with a rectangular cross-section. The load should only act in the z direction.

The second moment of area (I_y) is defined as the surface integral of z^2:

$$I_y = \int_A z^2 \, dA$$

To ensure that the entire cross-sectional area is covered, you must integrate in the y and z directions. For this reason, you must compute a double integral in the following way:

$$I_y = \int_{z=-\frac{h}{2}}^{\frac{h}{2}} \left(\int_{y=-\frac{b}{2}}^{\frac{b}{2}} z^2 \, dy \right) dz = \int_{-\frac{h}{2}}^{\frac{h}{2}} b \cdot z^2 \, dz = \frac{b \cdot h^3}{12}$$

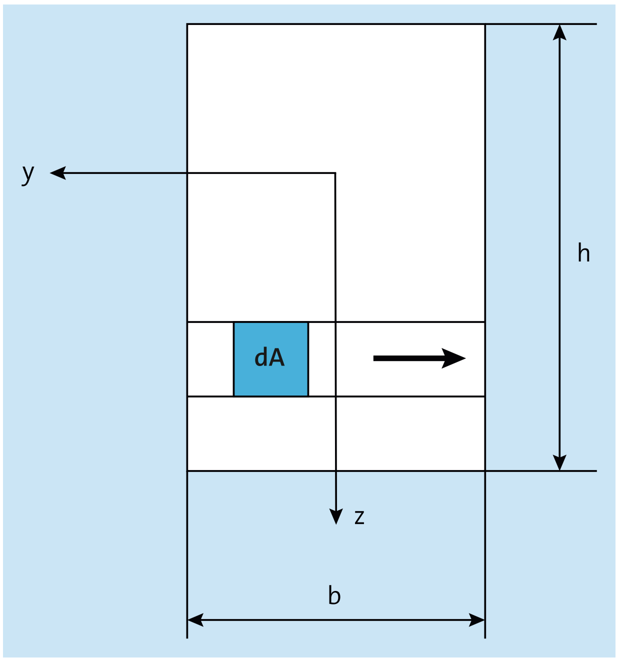


Figure 5.14 Cross-Section of a Beam

SymPy computes this double integral using the `integrate(fz,(y,y1,y2),(z,z1,z2))` method. The function `fz` to be integrated is passed as the first argument; then, the integration variable `y` with the associated lower and upper limits for the inner integral is passed as a tuple. The tuple for the integration variable `y` of the outer integral with the associated limits is passed as the third argument. The index 1 stands for the lower integration limit, and the index 2 for the upper integration limit. Instead of parentheses, you can also use square brackets.

Listing 5.31 uses the double integral (line 05) to calculate the general formula for the second moment of area (`Iy`) of a beam with a rectangular cross-section.

```
01  #31_moment_of_area.py
02  from sympy import *
03  Iy,y,z,h,b =symbols('Iy,y,z,h,b')
04  Iy=z**2
05  zI=integrate(Iy,(y,-b/2,b/2),(z,-h/2,h/2))
06  print("Moment of area\n Iy =",zI)
```

Listing 5.31 Moment of Area for a Rectangular Cross-Section

Output

```
Moment of area
 Iy = b*h**3/12
```

Analysis

In line 03, the symbolic variables are declared. Line 04 contains the calculation rule for the second moment of area. In line 05, the integration is carried out using the `integrate(Iy,(y,-b/2,b/2),(z,-h/2,h/2))` method. First, the inner integral with the integration variable `y` and then the outer integral with the integration variable `z` are inserted. The integrals must be enclosed in parentheses or square brackets with their lower and upper limits. If you swap the order of inner and outer integrals, you'll get the same result.

5.14.2 Equation of the Bending Line

Figure 5.15 shows a beam (cantilever) with a rectangular cross-section firmly clamped in the wall. This beam has length *l* and is loaded with force *F*.

The bending line of the beam is described by the following second-order linear differential equation:

$$\frac{\mathrm{d}^2 w}{\mathrm{d}x^2} = -\frac{M}{E \cdot I_y}$$

For the calculation of bending girders, a common practice is to choose a coordinate system where the *x*-axis points in the direction of the beam. The *y*-axis is perpendicular to the drawing plane, and the *z*-axis points downwards.

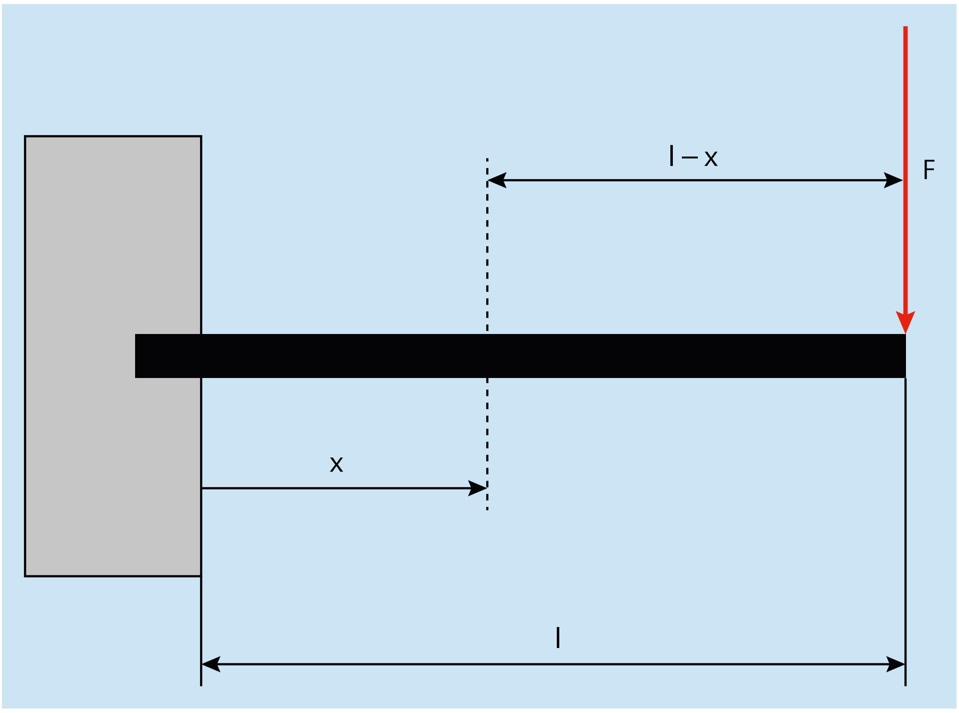


Figure 5.15 Beam, Fixed on One Side

The deflection $w(x)$ depends on the modulus of elasticity E, the second moment of area I_y, and the bending moment M. The bending moment, acting at point x, is described by the following linear equation:

$$M(x) = -F(l - x)$$

The bending moment is greatest at the fastening point $x = 0$. Its magnitude decreases linearly down to the point $x = l$, where its value is zero.

If you insert this torque equation into the above differential equation, you obtain the following:

$$\frac{d^2w}{dx^2} = \frac{F \cdot (l - x)}{E \cdot I_y}$$

This differential equation can be solved simply by integrating twice. Due the two boundary conditions $w(0) = 0$ und $w'(0) = 0$, you can determine the special solution.

Listing 5.32 calculates the general and the special solutions of the differential equation. The course of the bending line is also shown as function plot w = *f*(x). For the elasticity modulus, let's use the value for steel $E = 2{,}1 \cdot 10^5$ N/mm^2 (see line 21).

```
01  #32_bending_line.py
02  from sympy import *
03  F,Iy,E,l=symbols('F,Iy,E,l')
04  x = symbols('x')
05  w = Function('w')(x)
06  #Inputs
07  b=20   #width in mm
08  h=30   #height in mm
09  lx=1e3 #length in mm
10  Fz=1e2 #force in N
11  #solution of the differential equation
12  dgl=Eq(w.diff(x,2),F/(E*Iy)*(l-x))
13  aL=dsolve(dgl)        #general solution of the differential equation
14  rb={                  #boundary conditions
15      w.subs(x,0):0,
16      w.diff(x,1).subs(x,0):0
17      }
18  sL=dsolve(dgl,ics=rb) #special solution
19  rL=sL.rhs             #right-hand side of the equation
20  mL=sL.rhs.subs(x,l)
21  wmax=mL.subs(F,Fz).subs(l,lx).subs(E,2.1e5).subs(Iy,b*h**3/12)
22  wx=rL.subs(F,Fz).subs(l,lx).subs(E,2.1e5).subs(Iy,b*h**3/12)
23  #Outputs
24  print("Width b =",b, "mm")
25  print("Height h =",h, "mm")
26  print("Length l =",lx, "mm")
```

```
print("Force F =",Fz, "N")
print("General solution\n",aL)
print("Special solution\n",sL)
print("Right-hand side of the equation\n w(x) =",rL)
print(" w(x) =",wx)
print("Maximum deflection\n w(x=l) =",mL,"=",N(wmax,3),"mm")
p=plot(wx,(x,0,lx),ylabel='w(x)',show=False)
#p.save('bending_line.png')
#p.save('bending_line.svg')
p.show()
```

Listing 5.32 Calculation of the Bending Line

Output

```
Width b = 20 mm
Height h = 30 mm
Length l = 1000.0 mm
Force F = 100.0 N
General solution
 Eq(w(x), C1 + C2*x + F*l*x**2/(2*E*Iy) - F*x**3/(6*E*Iy))
special solution
 Eq(w(x), F*l*x**2/(2*E*Iy) - F*x**3/(6*E*Iy))
Right-hand side of the equation
 w(x) = F*l*x**2/(2*E*Iy) - F*x**3/(6*E*Iy)
 w(x) = -1.7636684303351e-9*x**3 + 5.29100529100529e-6*x**2
Maximum deflection
 w(x=l) = F*l**3/(3*E*Iy) = 3.53 mm
```

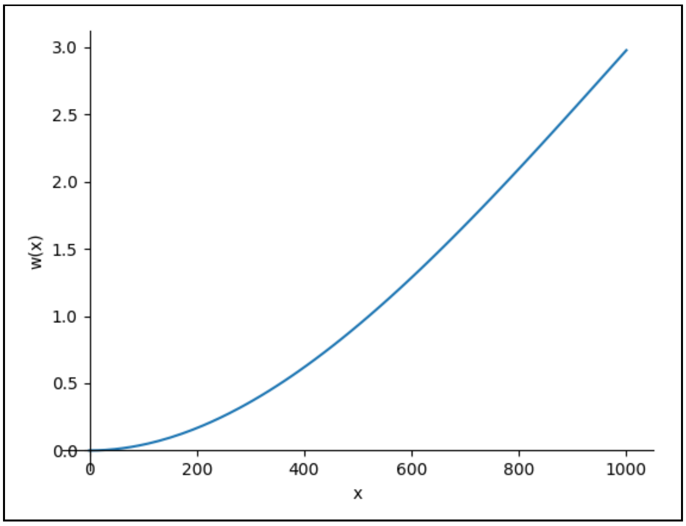


Figure 5.16 Course of the Bending Line w=*f*(x) in Millimeters

Analysis

In line 03, the symbolic variables are determined. A separate line is reserved for the independent variable `x` to highlight its importance (line 04). Line 05 specifies the functional relationship between the deflection `w` and the variable `x`.

In lines 07 to 09, you can change the geometric data defining the beam. Line 10 specifies the load `Fz` at the end of the beam in the direction of the z-axis.

Line 12 contains the mathematical term of the differential equation; it is assigned to the `dgl` object. In line 13, the `dsolve(dgl)` method solves the differential equation to derive its general solution.

In lines 14 to 17, the boundary conditions are specified for the deflection at point $x = 0$, i.e., $w(0) = 0$, and for the first derivative at point $x = 0$, i.e., $w'(0) = 0$. They are stored in a dictionary—`{w.subs(x,0):0, w.diff(x,1).subs(x,0):0}`. In line 18, the `dsolve()` method calculates the special solution of the differential equation. The expression `ics` (*initial conditions*) stands for the boundary conditions of the differential equation.

Using `rhs` (*right-hand side*) the right-hand side of the special solution `sL` is determined (line 19). Lines 20 and 21 use the `mL` object to calculate the maximum deflection of the beam at point $x = l$. In line 22, the course of the function, $w = f(x)$is calculated using the notation `rL.subs(variable,value)`.

In line 33, the `plot` data is stored in the `p` object. The `p.show()` method causes the graphic to be displayed on the screen (line 36). You can also use the `p` object to save the function plot in a file format of your choice (`png` or `svg`) (lines 34 and 35). To determine which file formats are still supported, you can try saving the graphic in `gif` format. This format is not supported, and you'll receive an error message indicating what other file formats are available as storage options.

5.15 Project Task: Reaction Kinetics

In this project task, we must solve a linear differential equation system with three differential equations. For this purpose, I chose an example from physical chemistry. Let's look at the following consecutive reaction:

$$A \overset{k_1}{\rightarrow} B \overset{k_2}{\rightarrow} C$$

In a chemical reaction, substance A (the *reactant*) gives rise to substance B (the *intermediate*), which in turn gives rise to substance C (the *product*). Each substance has the molar concentration c at a certain point in time. This value is the ratio of the amount of substance n and the volume V of the substance and is defined in the following way:

$$c = \frac{n}{V}$$

The unit mol/liter is often used for units of molar concentration.

The rates at which the reactions proceed are determined by the reaction rate constant k. Its unit is s^{-1}.

A consecutive chemical reaction produces substance B from substance A; and substance C, from substance B. During the course of this reaction, the molar concentrations c_A, c_B, and c_C of these reactants change. The reaction rate constants k_1 and k_2 influence the time course of the molar concentrations. This kinetic reaction can be described by the following linear differential equation system:

$$\frac{\mathrm{d}c_A}{\mathrm{d}t} = -k_1 \cdot c_A$$
$$\frac{\mathrm{d}c_B}{\mathrm{d}t} = k_1 \cdot c_A - k_2 \cdot c_B$$
$$\frac{\mathrm{d}c_C}{\mathrm{d}t} = k_2 \cdot c_B$$

Using the example of the breeding of plutonium-239 from uranium-238, let's look at how such a differential equation system can be solved using SymPy. For the start reaction, the following applies:

$$^{238}_{92}U + n \rightarrow {}^{239}_{92}U$$

This results in the consecutive reaction:

$$^{239}_{92}U \xrightarrow{23{,}5\ \text{min}} {}^{239}_{93}Np \xrightarrow{3384\ \text{min}} {}^{239}_{94}Pu$$

The half-lives are indicated above the arrows. From the half-lives, the rate constants can be calculated:

$$k = \frac{\ln 2}{T_{1/2}}$$

Then you obtain the following values for the rate constants k_1 and k_2:

$$k_1 = \frac{\ln 2}{23{,}5\ \text{min}} = 0.0295\ \frac{1}{\text{min}}$$
$$k_2 = \frac{\ln 2}{3384\ \text{min}} = 2.0483 \cdot 10^{-4}\ \frac{1}{\text{min}}$$

In Listing 5.33, the `dsolve_system(equations)` SymPy method solves the linear differential equation system for the breeding of plutonium-239 from uranium-238. In lines 08 and 09, you can enter the corresponding rate constants for other consecutive reactions.

```
01  #33_differential_equation_system.py
02  from sympy import symbols,Eq,Function,plot,N
03  from sympy.solvers.ode.systems import dsolve_system
04  t = symbols("t")
05  cA = Function("cA") #mol/dm^3
06  cB = Function("cB")
07  cC = Function("cC")
```

```
k1=0.0295     #1/min, U in Np
k2=2.0483e-4  #1/min, Np in Pu
#differential equation system
dgl1=Eq(cA(t).diff(t,1),-k1*cA(t)) #reactant
dgl2=Eq(cB(t).diff(t,1), k1*cA(t)-k2*cB(t)) #intermediate
dgl3=Eq(cC(t).diff(t,1), k2*cB(t)) #product
#initial values
aw={
    cA(0): 1,
    cB(0): 0,
    cC(0): 0
    }
#Solution of the differential equation system
equations = [dgl1,dgl2,dgl3]
aL=dsolve_system(equations)        #general solution
sL=dsolve_system(equations,ics=aw) #special solution
gA=sL[0][0].rhs #reactant
gB=sL[0][1].rhs #intermediate
gC=sL[0][2].rhs #product
#Outputs
print("General solution\n",aL)
print("Special solution")
print("cA(t) =",N(gA,3))
print("cB(t) =",N(gB,3))
print("cC(t) =",N(gC,3))
p=plot(gA,gB,gC,(t,0,600),show=False,legend=True)
p.title='Consecutive reaction'
p.xlabel='t in min'
p.ylabel='Concentration'
p[0].line_color='blue'
p[0].label='Uranium'
p[1].line_color='green'
p[1].label='Neptunium'
p[2].line_color='red'
p[2].label='Plutonium'
p.show()
```

Listing 5.33 Solving a Linear Differential Equation System of Equations Using SymPy

Output

```
General solution
 [[Eq(cA(t), 143.021871796124*C1*exp(-0.0295*t)), Eq(cB(t),
-144.021871796124*C1*exp(-0.0295*t) - 1.0*C2*exp(-0.00020483*t)), Eq(cC(t),
1.0*C1*exp(-0.0295*t) + 1.0*C2*exp(-0.00020483*t) + 1.0*C3)]]
```

```
Special solution
cA(t) = 1.0*exp(-0.0295*t)
cB(t) = -1.01*exp(-0.0295*t) + 1.01*exp(-0.00020483*t)
cC(t) = 1.0 + 0.00699*exp(-0.0295*t) - 1.01*exp(-0.00020483*t)
```

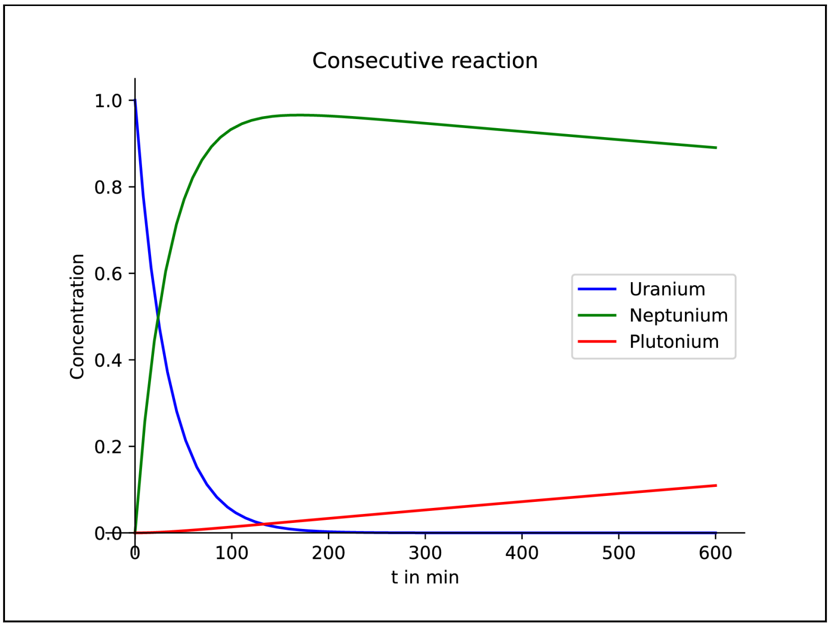


Figure 5.17 Consecutive Reaction for Uranium → Neptunium → Plutonium

Analysis

In line 02, all methods necessary for the program are imported. Line 03 imports the `dsolve_system` method, which is to solve the differential equation system. We set `t` as the independent variable (line 04). In lines 05 to 07, the concentrations `cA`, `cB` and `cC` are declared as functions.

In lines 08 and 09, you can assign different values to the rate constants `k1` and `k2` for other consecutive reactions.

Lines 11 to 13 contain the terms of the differential equations of the differential equation system. The data of the individual differential equations are stored in objects `dgl1`, `dgl2`, and `dgl3`.

In lines 15 to 19, the initial values for the individual concentrations are specified. Only reactant *A* has an initial value. The concentrations of intermediate *B* and product *C* are zero at the beginning of the reaction.

In line 21, the `dgl1`, `dgl2`, and `dgl3` objects are combined into a list and stored in the `equations` object.

In lines 22 and 23, the `dsolve_system(equations)` method calculates the general and special solutions of the differential equation system.

In lines 24 to 26, the data of the right-hand side of the equation is determined and stored in objects `gA`, `gB`, and `gC`. The `print` function outputs the general and special solutions stored in these objects in lines 28 to 32.

In line 33, the data of the function plot is stored in the `p` object. Lines 34 to 42 use this object to specify the colors of the function graphs and the labels for the legend.

5.16 Project Task: Dual Mass Oscillator

The model of the coupled oscillator can be used to study complex technical systems such as electromechanical drives and structures. In this task, we want to example the dual mass oscillator shown in Figure 5.18.

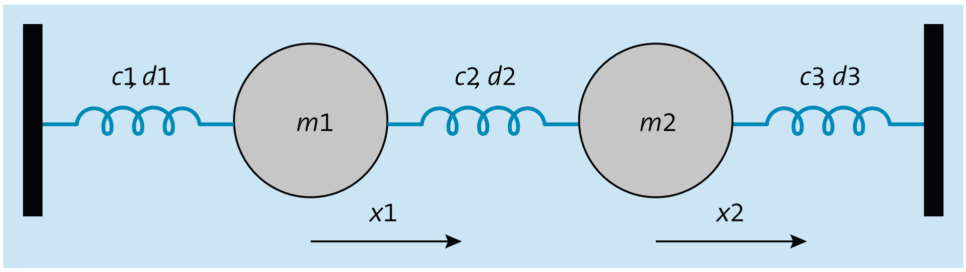


Figure 5.18 Dual Mass Oscillator

The dual mass oscillator consists of two masses (m_1 and m_2), which are connected (coupled) by three springs. The vibration behavior of the system is influenced by the masses m, the spring constants c, and the damping d.

The vibration behavior of the dual mass oscillator can be described by the following differential equation system (see Vöth: 80):

$$m_1\ddot{x}_1 + d_1\dot{x}_1 + d_2(\dot{x}_1 - \dot{x}_2) + c_1x_1 + c_2(x_1 - x_2) = 0$$
$$m_2\ddot{x}_2 + d_3\dot{x}_2 + d_2(\dot{x}_2 - \dot{x}_1) + c_3x_2 + c_2(x_2 - x_1) = 0$$

The `dsolve_system()` SymPy method calculates the deflection $x(t)$, the velocity $v(t)$, and the acceleration of the masses $a(t)$. You can pass the left-hand side of the equation terms to this method as arguments without transformations.

Listing 5.34 solves the differential equation system of the dual mass oscillator and displays the deflections x_1 and x_2 of the masses graphically on the screen.

```
01  #34_differential_equation_dual_mass_oscillator.py
02  from sympy import symbols,Eq,Function,plot,N
03  from sympy.solvers.ode.systems import dsolve_system
04  t = symbols("t")
```

```
x1 = Function("x1")(t)
x2 = Function("x2")(t)
m=1000 #kg
c=1e7  #N/m
d=1e3  #kg/s
m1,m2=m,2*m
c1,c2,c3=c,c,c
d1,d2,d3=d,d,d
#differential equation system
dgl1=Eq(m1*x1.diff(t,2)+d1*x1.diff(t,1)+d2*(x1.diff(t,1)\
        -x2.diff(t,1))+c1*x1+c2*(x1-x2),0)
dgl2=Eq(m2*x2.diff(t,2)+d2*(x2.diff(t,1)-x1.diff(t,1))\
        +d3*x2.diff(t,1)+c2*(x2-x1)+c3*x2,0)
#initial values
aw={
    x1.subs(t,0): 0.01, #m
    x2.subs(t,0): 0,
    x1.diff(t,1).subs(t,0):0,
    x2.diff(t,1).subs(t,0):0
    }
#Solution of the differential equation system
equations = [dgl1,dgl2]
aL=dsolve_system(equations)        #general solution
sL=dsolve_system(equations,ics=aw) #special solution
gX1=sL[0][0].rhs #deflection for m1
gX2=sL[0][1].rhs #deflection for m2
#Outputs
#print("General solution\n",aL)
#print("Special solution")
#print("x1(t) =",N(gX1,3))
#print("x2(t) =",N(gX2,3))
p=plot(gX1,gX2,(t,0,0.2),show=False,legend=True)
p.xlabel='t'
p.ylabel='Deflection in m'
p[0].line_color='blue'
p[0].label='x1'
p[1].line_color='red'
p[1].label='x2'
p.show()
```

Listing 5.34 Solution of the Differential Equation System for the Dual Mass Oscillator

The general and special solutions have not been printed here because they are quite long and complex.

Output

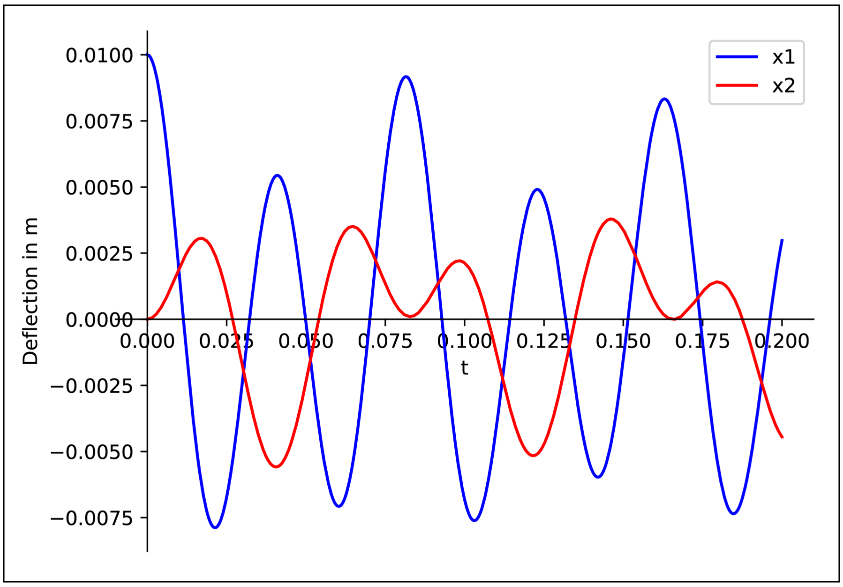


Figure 5.19 Deflection of the Dual Mass Oscillator

Analysis

The numerical values for the masses, spring constants, and damping are taken from Vöth: 79f (lines 07 to 09). In that source, for the spring constants, $c_2 = 2c$, c_3=3c and for the damping d_2= 2d, d_3 = 3d is used. If you test the program with these values, you'll find that the differential equation system is no longer solved or that the computation takes an unacceptable amount of time.

In lines 14 and 16, the two differential equations are passed as arguments to the `Eq()` method. All symbolic data of these differential equations is stored in the `dgl1` and `dgl2` objects.

The initial values are defined in lines 19 to 24. Only the mass m_1 is shifted with the initial value x_1 = 0.01 m. All other initial values are set to 0.

In lines 27 and 28, the `dsolve_system()` SymPy method solves the differential equation system. The general solution and the special solution are stored in the `aL` and `sL` objects.

The statements in lines 29 and 30 cause only the deflections x_1 and x_2 to be selected from the entire solution set.

Lines 36 to 43 contain the statements for the graphical output. You can use `p.save('dual_mass_oscillator.png')` to save the graphic in PNG format.

Note

You should not use SymPy to solve differential equation systems if their equations are composed of complicated mathematical terms. In Chapter 6, I show you how to solve the differential equation system of the dual mass oscillator more effectively numerically using the SciPy function `solve_ivp()`.

5.17 Tasks

1. Simplify the following term:

   $3x^4 + 6x^2 + 3 + 3x^4 + 3x^2 + 3x^2 + 3 - 6x^4 - 6x^2 + 2x^2 + 2 - 8x^2$

2. Calculate the limits for the following functions:

   $$\lim_{x\to 0}\frac{\sin nx}{x}$$
   $$\lim_{x\to 0}\frac{\tan nx}{x}$$
   $$\lim_{x\to 0}\frac{\sin x}{x\sqrt[3]{\cos x}}$$
   $$\lim_{x\to 0}\frac{\sin 2x - 2\sin x}{2e^x - x^2 - 2x - 2}$$

3. Calculate the following derivatives:

   $$y_1 = \sin ax \cdot \cos bx$$
   $$y_2 = \frac{\sin ax}{\cos bx}$$
   $$y_3 = \frac{x^3 + 2x}{4x^2 - 7}$$
   $$y_4 = (1 - \cos^4 x)^2$$
   $$y_5 = \arctan x$$

4. Calculate the following integrals:

   $$\int e^{\mathrm{ax}}\mathrm{d}x \int x^2 e^{\mathrm{x}} \cos x \,\mathrm{d}x$$
   $$\int \frac{1}{1 + \cos x}\mathrm{d}x$$
   $$\int \tan x \,\mathrm{d}x$$
   $$\int \sin x \cos x \,\mathrm{d}x$$

5. Solve the following differential equations:

   $$2y' + 8y = 0$$
   $$\frac{1}{5}y' = 6y$$

$$3y' = 6y$$
$$7y'' - 4y' - 3y = 6$$
$$y'' - 10y' + 9y = 9x$$

6. Decompose the image function into its partial fractions:

   $$F(s) = \frac{s^3 + 16s - 24}{s^4 + 20s^2 + 64}$$

   Then, transform the individual partial fractions into the time domain using a correspondence table.

   Check the result using the following method:

   ```
   inverse_laplace_transform(Fs,s,t)
   ```

7. Calculate the transfer function for an unbalanced third-order low-pass filter. The internal resistance of the voltage source and the terminating resistor each have a value of 1Ω.

Chapter 6

Numerical Computations and Simulations Using SciPy

This chapter describes how you can carry out numerical differentiations and integrations using SciPy, how to numerically solve differential equations, and how SciPy can support you in analyzing non-sinusoidal waves.

The acronym SciPy stands for **Sci**entific **Py**thon. SciPy is a Python-based collection of open-source software for mathematics, science, and engineering.

This module provides users with numerous user-friendly and efficient routines for numerical differentiation, integration, interpolation, optimization, linear algebra, and statistics.

In contrast to SymPy, the SciPy module performs engineering mathematical calculations numerically rather than symbolically. Since the numerical representation of the results of extensive numerical computations (such as solving differential equations) is not particularly meaningful, the SciPy module is almost always used in combination with the Matplotlib module to visualize the solutions. The NumPy module provides the `linspace()` and `arange()` functions for creating one-dimensional arrays. The Matplotlib method `plot()` is used for visualizing the results. Thus, the NumPy and Matplotlib modules form the basis of SciPy.

From the large range of functions, only a limited selection can be discussed in this book, covering as broad a range of topics in engineering mathematics as possible. For this reason, this chapter is limited to the exemplary treatment of submodules most relevant to engineering and science: `optimize`, `interpolate`, `integrate`, `fft`, and `signal`.

In accordance with convention, the modules for NumPy and Matplotlib can be included in a Python source code using the names `np` and `plt`:

```
import numpy as np
import matplotlib.pylot as plt
```

The following notation is common for importing submodules:

```
from scipy.submodule import function1, function2, etc.
```

6.1 Numerical Computation of Zeros

There are many mathematical functions whose zeros cannot be computed analytically. In these cases, the zeros (i.e., the points of intersection with the x-axis) must be calculated using numerical methods, such as secants or the Newton method. In addition to the Newton method, SciPy also provides other methods. The SciPy functions for calculating zeros are part of the `optimize` subpackage. To use the Newton method, you must transfer the mathematical function `fun` and the start values `x0` to the SciPy function `newton(fun, x0, ...)`. If, on the other hand, you want to use other methods, you must also pass the name of the mathematical method `method`, which is used to calculate the zeros, to the SciPy function `root(fun, x0, method, ...)`. The possible mathematical methods, such as `hybr`, `lm`, and so on, are listed in the commented-out lines 13 and 14. They can be tested if necessary.

Listing 6.1 calculates the zeros for the damped sinusoidal oscillation:

$$y = 10e^{-\frac{x}{2}}\sin x$$

```
01  #01_zeros.py
02  import numpy as np
03  import matplotlib.pyplot as plt
04  from scipy.optimize import root
05
06  def f(x):
07      #y=(x-1)*(x-2)*(x-5)*(x-10)
08      y=10*np.exp(-0.5*x)*np.sin(x)
09      return y
10
11  x = np.linspace(0,15,100)
12  x0=[0,2,6,9,12] #start values
13  #hybr,lm,broyden1,broyden2,anderson
14  #linearmixing,diagbroyden,excitingmixing,krylov,df-sane
15  xn=root(f,x0,method='hybr')
16  print(xn.x)
17  fig, ax = plt.subplots()
18  ax.plot(x,f(x),"r-",lw=2)
19  ax.scatter(xn.x,[0,0,0,0,0],color="k",marker="x")
20  ax.set(xlabel="x",ylabel="y")
21  ax.grid(True)
22  plt.show()
```

Listing 6.1 Calculation of Zeros

Output

```
[0. 3.14159265  6.28318531  9.42477796 12.5663707]
```

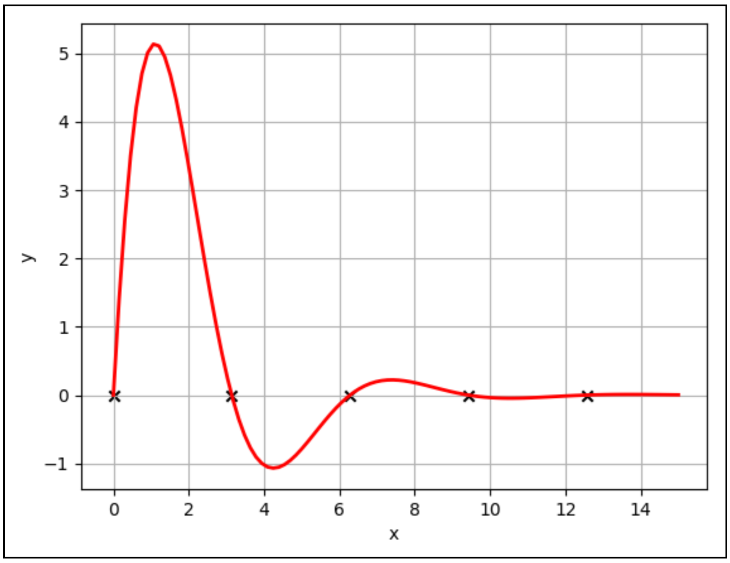


Figure 6.1 Zeros of a Damped Sinusoidal Oscillation

Analysis

Line 04 imports the `root` function from the `optimize` submodule. The commented-out polynomial in line 07 serves as a test function. The list in line 12 contains the initial values for the calculation of the zeros. The specification of the start values is actually the critical part of the program because their approximate position must be known before the calculation of the exact values. Here, the function plot from Figure 6.1 is useful. The commented-out lines 13 and 14 contain the names of the possible mathematical methods to calculate zeros. For testing purposes, you can use them in line 15 to replace `hybr`. In this line, the SciPy function `root(f,x0,method='hybr')` computes the zeros of function `f` using the mathematical method, `hybr`. More details about the mathematical methods of zero calculation can be found in the SciPy documentation. The `x0` parameter contains the list of start values. The zeros are stored in the `xn` object and output with `xn.x` in line 16.

The Matplotlib method `scatter()` in line 19 marks the zeros with a cross. The length of the list with zeros (second parameter) must correspond to the length of the `x0[]` list from line 12.

6.2 Optimizations

From school mathematics, we know the problem of how to calculate the smallest possible surface area for a tin can (cylinder) to minimize material consumption. SciPy provides the `minimize(fun,x0,...)` function from the `optimize` submodule for this kind of optimization task. This function must be passed an initial value `x0` in addition to a mathematical function `fun`, which describes the optimization problem.

The following applies to the volume of a cylinder:

$$V = \pi \cdot r^2 \cdot h$$

The surface is composed of the two flat surfaces and the curved surface:

$$A = 2\pi \cdot r^2 + 2\pi \cdot r \cdot h$$

The height h can be replaced by $\frac{2 \cdot V}{r}$:

$$A(r) = 2\pi \cdot r^2 + \frac{2 \cdot V}{r}$$

If you set the first derivative of the surface function to equal zero, you'll obtain the optimum radius of a tin can with minimal surface:

$$A'(r) = 4\pi \cdot r - \frac{2 \cdot V}{r^2} = 0$$

$$r_{\text{opt}} = \sqrt[3]{\frac{V}{2\pi}}$$

The calculation in Listing 6.2 confirms this result.

```
#02_minimum.py
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize
V=1 #volume

def f(r):
    h=V/(np.pi*r**2)
    A=2*np.pi*r**2+2*np.pi*r*h
    return A

x = np.linspace(0.1, 2, 100)
opt=minimize(f,0.5)
#Output
print("r=%4.6f A=%4.6f" %(opt.x[0],opt.fun))
```

```
16  fig, ax = plt.subplots()
17  ax.plot(x,f(x),"b-",lw=2)
18  ax.plot(opt.x[0],opt.fun,"rx",lw=2)
19  ax.set(xlabel="Radius",ylabel="Surface")
20  ax.grid(True)
21  plt.show()
```

Listing 6.2 Optimization of a Cylinder Surface

Output

```
r=0.541926 A=5.535810
```

Figure 6.2 shows the optimal radius in a function plot.

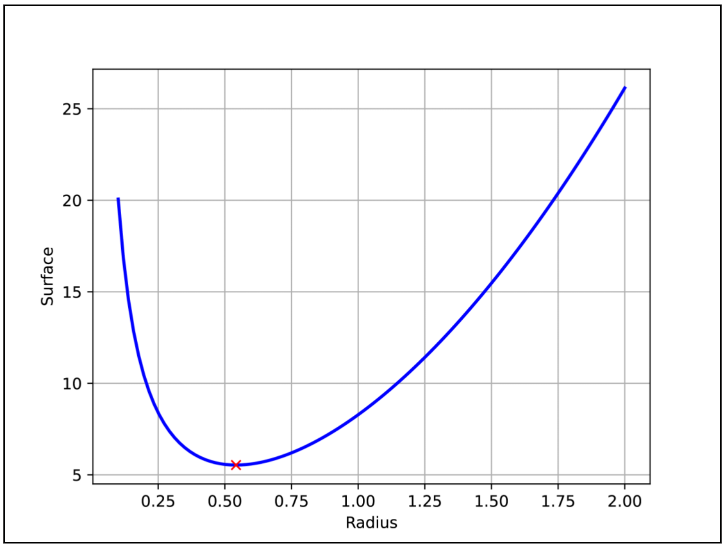


Figure 6.2 Optimal Radius

Analysis

In line 04, the SciPy function `minimize` is imported from the `optimize` submodule. Line 05 sets the volume of the cylinder to 1 volume unit (V). In line 13, the `minimize(f,0.5)` function calculates the optimal radius for a minimum cylinder surface for the `f(r)` function defined in line 07 and stores the result in the `opt` object. With the help of this object, the optimal values `opt.x[0]`, `opt.fun` for the radius `x` and the cylinder surface `fun` can be output in line 15 and used to mark the minimum in line 18.

6.3 Interpolations

Interpolation determines a value between two points on a curve. For any given discrete data set (e.g., measured values), a continuous function must be found that maps this data. Interpolation is used to predict trends or curve progressions.

The SciPy function `interp1d(x,y,kind='linear',...)` from the `interpolate` submodule can solve interpolation problems in the plane. The x and y parameters are arrays with the x-y coordinates of the measuring points. The third parameter specifies the interpolation method, such as `linear`, `next`, `previous`, `quadratic`, or `cubic`.

The discrete data shown in Figure 6.3 represents some sampled values of a sine wave. Let's now search for the interpolating function.

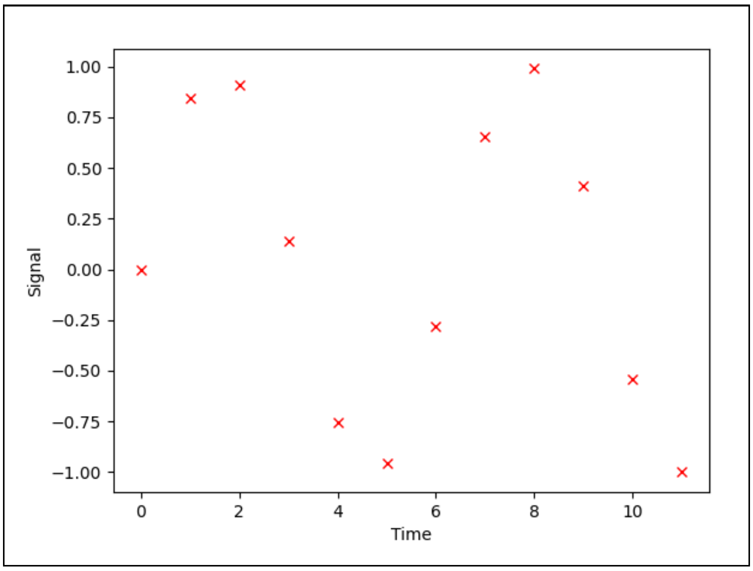


Figure 6.3 Sampled Signal

Listing 6.3 calculates the interpolating function using the `kind='cubic'` interpolation method.

```
#03_interpolation.py
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d
ta=np.arange(0,12)
ti=np.arange(0,11,0.01)
#sampled signal
s=np.sin(ta)
```

```
09  #interpolation methods
10  #linear,next,previous,quadratic,cubic
11  f = interp1d(ta, s, kind='cubic')
12  fig, ax = plt.subplots()
13  ax.plot(ta,s, 'rx')    #points
14  ax.plot(ti,f(ti),'b-') #interpolated
15  ax.set(xlabel='Time', ylabel='Signal')
16  plt.show()
```

Listing 6.3 Interpolation of Sampling Signals

Output

Figure 6.4 shows the recovered signal.

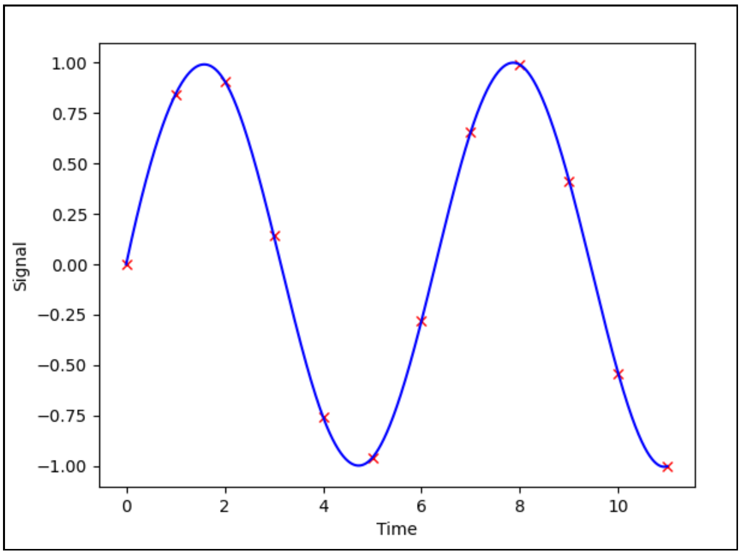


Figure 6.4 Recovered Signal by Interpolation

Analysis

Line 04 imports SciPy function `interp1d`. The array in line 05 contains twelve interpolation values. From these values, the sine function in line 08 generates the sampled amplitudes. In line 11, the SciPy function `interp1d(ta,s,kind='cubic')` reconstructs the original signal from these sampled amplitude values. The interpolation method used is cubic interpolation (`cubic`), which uses a third-degree polynomial as the interpolation function. For further testing purposes, you can try the interpolation methods commented out in line 10.

6.4 Numerical Differentiation

Numerical differentiation is the approximate calculation of the slope of a mathematical function for a given point (x_s,y_s). Since the slopes are calculated using the difference quotient Δy/Δx, there is always an error, which cannot be eliminated even by reducing Δx.

6.4.1 Methods of Numerical Differentiation

The accuracy of numerical differentiation by calculating the difference quotient,

$$\frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta y}{\Delta x} = \frac{f(x+h) - f(x)}{h}$$

can be improved by using the central difference quotient,

$$\frac{f(x+h) - f(x-h)}{2h}$$

Note

As of SciPy version 2.0.0, the `misc` submodule has been removed. Thus, the `derivative()` function is no longer available. The SciPy documentation recommends using the `Derivative()` function from the `numdifftools` module instead. In the download area of *https://www.rheinwerk-computing.com/5852/* or *https://drsteinkamp.de*, you can find source code versions in which derivatives are still calculated using the SciPy function `derivative()`.

The `numdifftools` function `Derivative(fun,n=1)` expects at least one argument to be passed: a mathematical function `fun`. The `n` parameter specifies that the nth derivative must be calculated, but this step is optional. If you omit this parameter, the first derivative will be calculated.

If you can't install the `numdifftools` module or don't want to use it, you can also test the differentiation examples covered in this section using the following custom Python function:

```
def derivative(f,x,h=1e-9):
    return (f(x+h)-f(x-h))/(2.0*h)
```

Listing 6.4 calculates for the following parabola, $y = 0.25x^2$, the slope of the secant or tangent at point $x_s = 2$ for the distance $h = 10^{-6}$ on the x-axis using the central difference quotient and the `Derivative()` function from the `numdifftools` module.

```
#04_slope1.py
import numpy as np
from numdifftools import Derivative
```

```
04  #Function
05  def f(x):
06      return 0.25*x**2
07  #central difference quotient
08  def df(x,h):
09      return (f(x+h)-f(x-h))/(2*h)
10
11  xs=2   #place of the slope
12  h=1e-6 #accuracy
13  mS=df(xs,h) #secant slope
14  mT=Derivative(f,n=1) #tangent slope
15  a1=np.degrees(np.arctan(mS))
16  a2=np.degrees(np.arctan(mT(xs)))
17  print("Secant slope  m=%2.6f %s=%2.1f°"%(mS,chr(945),a1))
18  print("Tangent slope m=%2.6f %s=%2.1f°"%(mT(xs),chr(945),a2))
```

Listing 6.4 Comparison of Secant and Tangent Slopes

Output

```
Secant slope  m=1.000000 α=45.0°
Tangent slope m=1.000000 α=45.0°
```

Analysis

Line 03 imports the `Derivative` function from the `numdifftools` module. The slope at point `xs=2` (line 11) is calculated in line 13 using the custom Python function `df()` and in line 14 using the `Derivative(f,n=1)` function. The accuracy (increment) of the slope calculation has been set to a small value (`h=1e-6` in line 12). Surprisingly, the custom function `df(x,h)` from line 08 provides the same result as the `Derivative` function.

6.4.2 Drawing a Tangent Slope

To illustrate numerical differentiation, let's draw the slope tangent of a parabola at point (x_s,y_s). For the intersection of the tangent with the x-axis, the following applies:

$$x_0 = x_s - \frac{y_s}{m} = x_s - \frac{f(x_s)}{y'}$$

From the slope m at point (x_s,y_s) the straight-line equation of the tangent can be derived.

$$m = \frac{y_s}{x_s - x_0}$$

$$y_s = m(x - x_0)$$

Figure 6.5 shows for the parabola the tangent calculated using Listing 6.5 at point $x = 2$.

$$y = \frac{1}{2}x^2$$

```python
#05_slope2.py
import numpy as np
import matplotlib.pyplot as plt
from numdifftools import Derivative
#Function
def f(x):
    return x**2/2
#Tangent
def f2(x,xs):
    m=Derivative(f,n=1)
    x0=xs-f(xs)/m(xs)
    return m(xs)*(x-x0)

x = np.linspace(0, 5, 100)
xs=2 #place of the slope
fig, ax = plt.subplots()
ax.plot(x, f(x),"g-", lw=2) #function
ax.plot(x, f2(x,xs),"b-", lw=1) #tangent
ax.plot(xs, f(xs), "or") #red point
ax.set(xlabel="x",ylabel="y")
ax.grid(True)
plt.show()
```

Listing 6.5 Tangent Slope

Output

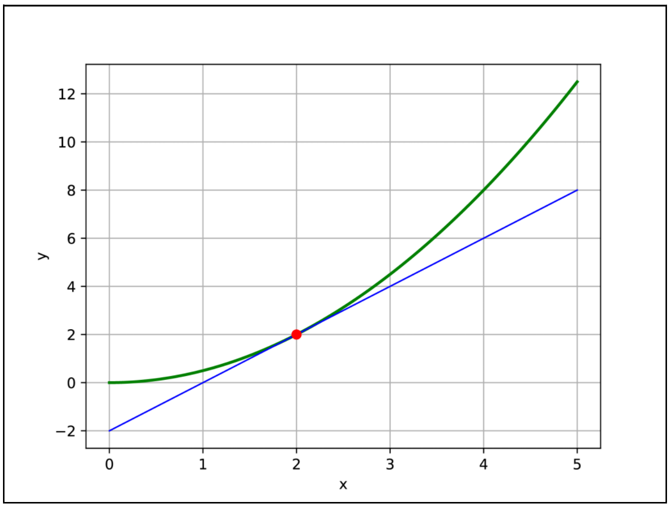


Figure 6.5 Tangent Slope

Analysis

In lines 09 to 12, the function `f2(x,xs)` is defined for the calculation of the tangent slope. Figure 6.5 shows the result.

The tangent has a slope of 2 at point $x_s = 2$, which corresponds to a slope angle of 63.43°. Note that the x and y axes are scaled unequally.

6.4.3 Derivative of a Sine Function

Figure 6.6 shows a coil through which flows an impressed current. Impressed currents are generated by power sources.

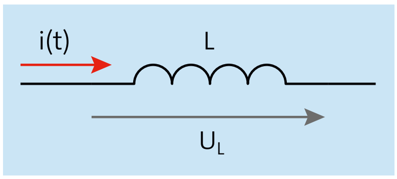


Figure 6.6 Inductive Voltage Drop

If an impressed current $i = f(t)$ flows through a coil with inductance L, then an inductive voltage drop is created at the coil according to the induction law:

$$u_L(t) = L\frac{\mathrm{d}i_L}{\mathrm{d}t}$$

Listing 6.6 calculates and visualizes this inductive voltage drop.

```
#06_slope3.py
import numpy as np
import matplotlib.pyplot as plt
from numdifftools import Derivative
L=1     #H
f=50    #frequency in Hz
omega=2*np.pi*f #1/s
imax=1. #A
#power source
def i(t):
    return imax*np.sin(omega*t*1e-3)#t in ms!
#inductive voltage drop
def u(t):
    df=Derivative(i)
    return 1e3*L*df(t)

t = np.linspace(0,20,500) #ms
fig,(ax1,ax2)=plt.subplots(2,1)
ax1.plot(t, i(t), 'r-', lw=2)
```

```
20  ax1.set(ylabel='Current in A',title='impressed current')
21  ax1.grid(True)
22  #inductive voltage drop
23  ax2.plot(t, u(t), 'b-',lw=2)
24  ax2.set(xlabel='t in ms',ylabel='Voltage in V',title='inductive voltage 
drop')
25  ax2.grid(True)
26  fig.tight_layout()
27  plt.show()
```

Listing 6.6 Derivative of the Current: Inductive Voltage Drop

Output

The output of Listing 6.6 is shown in Figure 6.7.

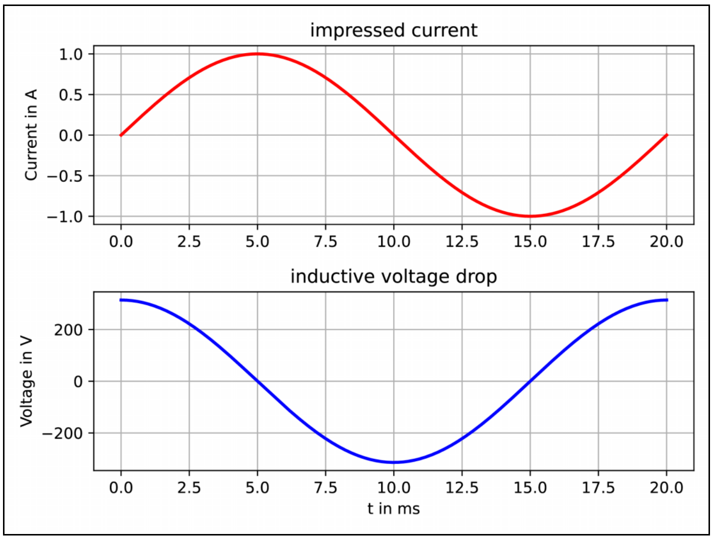


Figure 6.7 Derivative of a Sine Function

Analysis

In line 14, the `Derivative(i)` function calls function `i(t)` from line 10. The `t` parameter must not be passed here. All information of function `i(t)` is stored in the `df` object. In line 15, this `df(t)` object is passed the argument `t`, and it calculates the first derivative for the `i(t)` current for the values of the array from line 17. The scaling factor `1e3` undoes the scaling of the time axis in ms from line 11.

As expected, a cosine function is displayed in line 23 as the result.

6.4.4 Usage Example: Free Fall

Galileo Galilei (1564–1641) allegedly figured out the law of falling bodies by experimenting with an inclined plane: The traveled path of a sphere obeys the law of a geometric time series. The fact that this finding is basically correct can be experimentally proven with today's technical means. But by mere measurements, no matter how accurate they may be, the following formula cannot be precisely confirmed:

$$s = \frac{1}{2} g \cdot t^2$$

The exact formulation of the distance-time law can only be derived from the measured acceleration due to gravity g by means of the integral calculus. In our case, we'll use the distance-time law to illustrate using the `Derivative()` function to compute from the distance s the velocity and from the velocity the acceleration a of a falling sphere.

$$v = \frac{\mathrm{d}s}{\mathrm{d}t}$$

$$a = \frac{\mathrm{d}v}{\mathrm{d}t}$$

Listing 6.7 shows the implementation.

```
#07_slope4.py
import numpy as np
import matplotlib.pyplot as plt
from numdifftools import Derivative
g=9.81
#distance
def s(t):
    return g*t**2/2
#velocity
def v(t):
    df=Derivative(s,n=1)
    return df(t)
#acceleration
def a(t):
    df=Derivative(v,n=1)
    return df(t)
#time
t = np.linspace(0,5,100)#seconds
fig, ax = plt.subplots(3,1,figsize=(6,6))
#distance
ax[0].plot(t, s(t), 'b-', lw=2)
ax[0].set(ylabel='s in m',title='Distance')
#velocity
ax[1].plot(t, v(t), 'r-', lw=2)
ax[1].set(ylabel='v in m/s',title='Velocity')
```

```
26 #acceleration
27 ax[2].plot(t, a(t), 'g-', lw=2)
28 ax[2].set(xlabel='t in s',ylabel='a',title='Acceleration')
29 ax[2].set_ylim(0,12)
30 [ax[i].grid(True) for i in range(len(ax))]
31 fig.tight_layout()
32 plt.show()
```

Listing 6.7 Free Fall

Output

The free fall of the sphere is graphically summarized by the diagrams for distance, velocity, and acceleration shown in Figure 6.8.

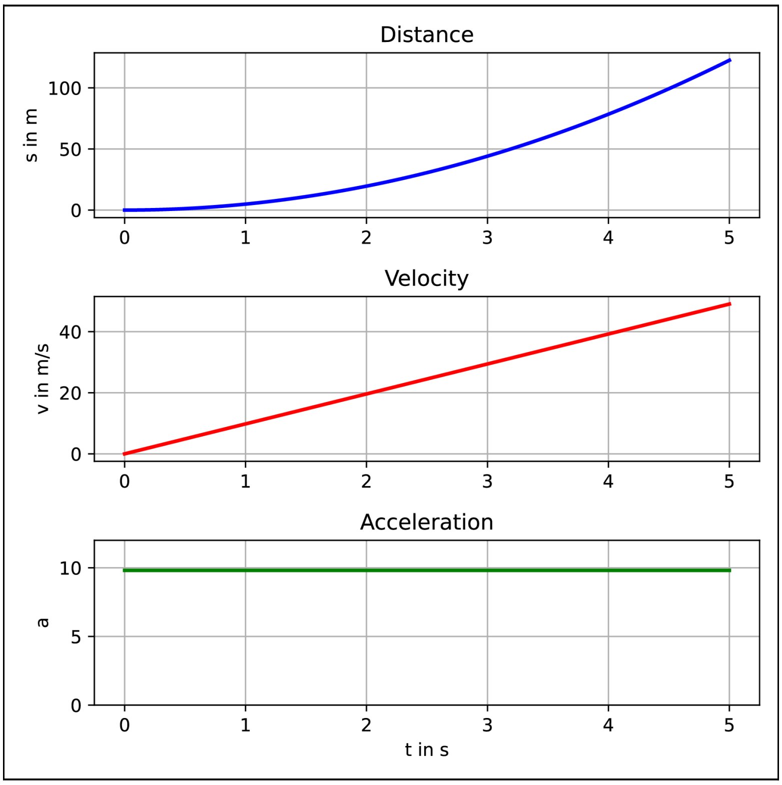


Figure 6.8 Free Fall: Distance, Velocity, Acceleration

Analysis

In lines 12 and 16, the derivatives for velocity `v(t)` and acceleration `a(t)` are calculated and returned via the `return` statement in the function calls in lines 24 and 27. They are shown as expected.

In line 15, you can also calculate the acceleration `a(t)` from the second derivative of the distance function `s(t)` using the `df=Derivative(s,n=2)` statement.

6.5 Numerical Integration

Numerical integration is the approximate calculation of certain integrals. Numerical methods are always used when integrals cannot be solved analytically or when only the numerical solutions are relevant. The latter is the case with engineering tasks.

SciPy provides numerous numerical integration methods in the `integrate` submodule, which can also compute double and triple integrals. The user is therefore free to select a method that suits their purposes. You can use the following statement to import the `integrate` submodule:

```
import scipy.integrate as integral
```

6.5.1 Methods of Numerical Integration

From school mathematics, you already know the numerical method of rectangle sums. But even with very small selected axis intercepts on the x-axis, this method is too inaccurate for practical purposes. For this reason, optimized methods approximate the curve section of the function to be integrated by suitable polynomials. Table 6.1 shows an overview of the numerical integration methods of the `integrate` submodule of SciPy.

Method	Description
`quad`	Standard procedure for single integration
`dblquad`	Standard procedure for double integration
`tplquad`	Standard procedure for triple integration
`fixed_quad`	Calculates a given integral using the Gaussian quadrature rule for a given order n
`quadrature`	Calculates a given integral using the Gaussian quadrature rule for a given tolerance
`romberg`	Calculates a given integral using the Romberg method

Table 6.1 Numerical Integration Methods of SciPy

The SciPy function `quad(func, a, b, ...)` expects three parameters. The name of the mathematical function `func` is passed as the first parameter without a function argument, so instead of `f(x)` simply `f` is passed. The second and third parameters set the lower and upper integration limits. Other parameters are optional. They can be taken from the SciPy documentation.

Listing 6.8 calculates the definite integral using a custom function of rectangle sums and the integration methods listed in Table 6.1.

$$\int_0^2 x^2 \mathrm{d}x = \frac{8}{3} \approx 2.6666666667$$

```
#08_integral_comparison.py
import scipy.integrate as integral

def f(x):
    return x**2
#rectangle sums
def rect(f,a,b,h=1e-6):
    n=int((b-a)/h)
    s=0
    for k in range(1,n):
        x=a+k*h
        s=s+f(x)
    return s*h

a=0 #lower limit
b=2 #upper limit
A1=rect(f,a,b)
A2=integral.quad(f,a,b)#[0]
A3=integral.fixed_quad(f,a,b,n=4)#[0]
A4=integral.quadrature(f,a,b,tol=1e-6)#[0]
A5=integral.romberg(f,a,b,tol=1e-6,show=False)
#Outputs
print("Rectangle sums\t: ",A1)
print("quad\t\t:",A2)
print("fixed_quad\t:",A3)
print("quadrature\t:",A4)
print("romberg\t\t: ",A5)
```

Listing 6.8 Methods of Numerical Integration

Output

```
Rectangle sums: 2.6666646666669664
quad          : (2.666666666666667, 2.960594732333751e-14)
fixed_quad    : (2.6666666666666665, None)
quadrature    : (2.6666666666666665, 4.440892098500626e-16)
romberg       :  2.6666666666666665
```

Analysis

As expected, the custom function `rect()` calculates the integral less accurately than the SciPy functions. Line 02 imports the `integrate` submodule and sets the `integral` alias. In lines 18 to 21, this alias is used to access the SciPy integration functions. The results are saved in objects `A2` to `A5`. The `quadrature()` and `romberg()` functions enable you to specify a value for the error tolerance. In lines 24 and 26, the `quad()` and `quadrature()` functions output a second value with an error estimation in addition to the surface areas. You can suppress the output of this error estimation if you remove the comments in lines 18 to 20.

6.5.2 Definite Integral

The next example, in Listing 6.9, calculates for the parabola

$$y = -(x-4)^2 + 10$$

the definite integral between the limits x_{01} and x_{02}:

$$\int_{x_{01}}^{x_{02}} [-(x-4)^2 + 10]\,\mathrm{d}x$$

The zeros of the parabola were chosen as integration limits.

```python
#09_area_parabola.py
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate as integral
from scipy.optimize import root

def f(x):
    return -(x-4)**2+10

x = np.linspace(0,8,100)
x0=[0,5]
xn=root(f,x0,method='hybr')
a,b=xn.x[0],xn.x[1]
A=integral.quad(f,a,b)[0]
#Output
print("Zeros:",a,b)
print("Area:",A)
#Display
fig, ax = plt.subplots()
ax.plot(x, f(x), "b-", lw=2)
ax.grid(True)
ax.set(xlabel="x",ylabel="f(x)")
ax.fill_between(x,f(x),where=f(x)>=0,color='g',alpha=0.2)
plt.show()
```

Listing 6.9 Area between Zeros

Output

```
Zeros: 0.8377223398316203 7.162277660168379
Area: 42.1637021355784
```

Figure 6.9 shows the area between the zeros colored.

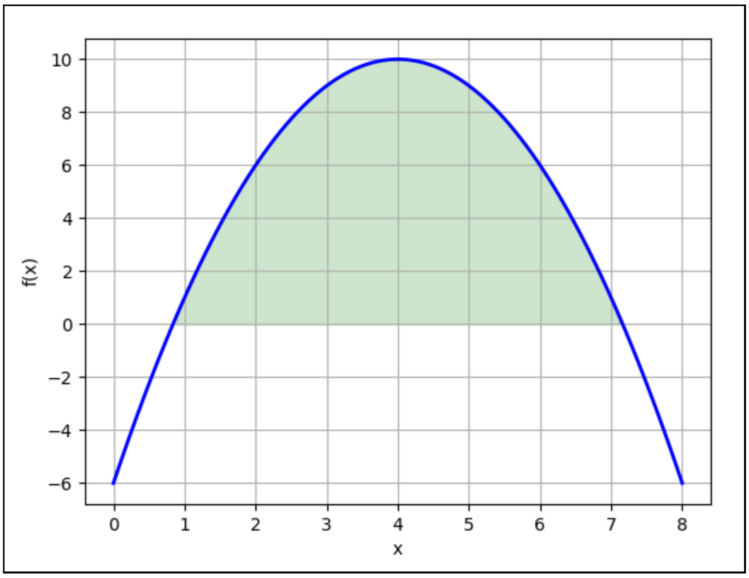


Figure 6.9 Area under a Parabola

Analysis

This example should illustrate on the one hand how you must calculate a certain integral with SciPy and on the other hand, how you can color the area between the zeros. As expected, the values calculated by the program for the zeros and the area meet the requirements for sufficient accuracy.

6.5.3 Integrating a Constant

The integration of a constant results in an ascending straight line. An uncharged capacitor is connected to a constant current source supplying a constant current of $i(t) = 10$ A, as shown in Figure 6.10.

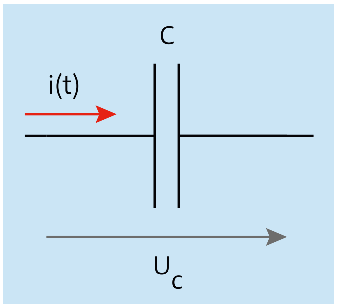


Figure 6.10 Voltage at the Capacitor

Using an oscilloscope, you can then measure a linearly increasing voltage.

To simulate this voltage profile, Listing 6.10 is used to calculate the following determined integral:

$$u_c(t) = \frac{1}{C}\int_0^t i_c \mathrm{d}\tau$$

```python
#10_constant_integration1.py
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate as integral
C=1 #F
imax=10
#Constant current source
@np.vectorize
def i(t):
    return imax
#Capacitor voltage
@np.vectorize
def u(t):
    uc=(1/C)*integral.quad(i,0,t)[0]
    return uc

x = np.linspace(0, 20, 500)
fig, (ax1,ax2)=plt.subplots(2,1)
ax1.plot(x, i(x), 'r-', lw=2)
ax1.set(ylabel='Current in A',title='Voltage at capacitor')
ax2.plot(x, u(x),'b-',lw=2)
ax2.set(xlabel='t in s',ylabel='$u_c(t)$ in V')
ax1.grid(True);ax2.grid(True)
plt.show()
```

Listing 6.10 Simulating the Voltage at the Capacitor

Output

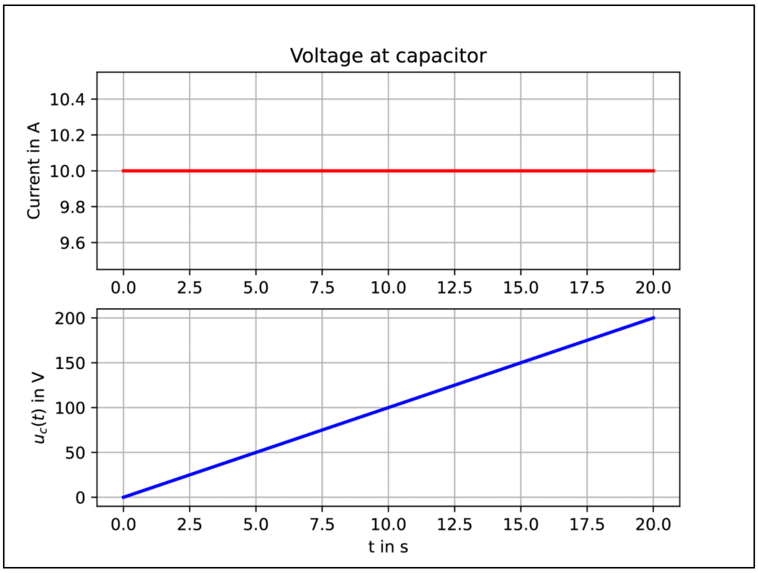


Figure 6.11 Voltage Profile at the Capacitor

Analysis

The integration of a constant is by no means trivial. In line 10, the custom function `i(t)` returns a constant named `imax`. This constant must be converted to an array via `@np.vectorize` so that it can be numerically integrated and `plotted` using the `plot` function, as shown in Figure 6.11. In lines 08 and 12, the `@np.vectorize` statement creates a NumPy array with 500 elements from the custom functions `i(t)` and `u(t)`. The number of elements is specified using NumPy function `np.linspace(0, 20, 500)` in line 17.

6.5.4 Usage Example: Free Fall

In Section 6.4.4, we solved this free fall problem by means of differential calculus. This time, the fall velocity and the distance will be calculated by integrating the acceleration caused by gravity. The following equation thus applies to the velocity:

$$v(t) = \int_0^t g\mathrm{d}\tau = gt$$

For the distance, the following equation applies:

$$s(t) = \int_0^t v\mathrm{d}\tau = \frac{1}{2}gt^2$$

Listing 6.11 shows the implementation.

```
01  #11_constant_integration2.py
02  import numpy as np
03  import matplotlib.pyplot as plt
04  import scipy.integrate as integral
05  g=9.81
06  @np.vectorize
07  def a(t):
08      return g
09  @np.vectorize
10  def v(t):
11      return integral.quad(a, 0, t)[0]
12  @np.vectorize
13  def s(t):
14      return integral.quad(v, 0, t)[0]
15  #time values
16  t = np.linspace(0,5,100)
17  fig, ax=plt.subplots(3,1,figsize=(6,8))
18  ax[0].plot(t, a(t), 'g-', lw=2)
19  ax[0].set(ylabel='a in $m/s^2$',title='Acceleration')
20  ax[0].set_ylim(0,12)
21  ax[1].plot(t, v(t), 'r-',lw=2)
22  ax[1].set(ylabel='v in m/s',title='Velocity')
23  ax[2].plot(t, s(t), 'b-', lw=2)
24  ax[2].set(xlabel='t in s',ylabel='s in m',title='Distance')
25  [ax[i].grid(True) for i in range(len(ax))]
26  fig.tight_layout()
27  plt.show()
```

Listing 6.11 Free Fall: Acceleration, Velocity, Distance

Output

The graphical output of this free fall computation is shown in Figure 6.12.

Analysis

Basically, the program does not contain any new programming elements. The custom functions `a(t)`, `v(t)`, and `s(t)` must be vectorized again using `@np.vectorize` so that they can be integrated numerically and their time courses can be displayed. The functions `a(t)` and `v(t)` can be called and integrated directly after the `return` statement (lines 11 and 14).

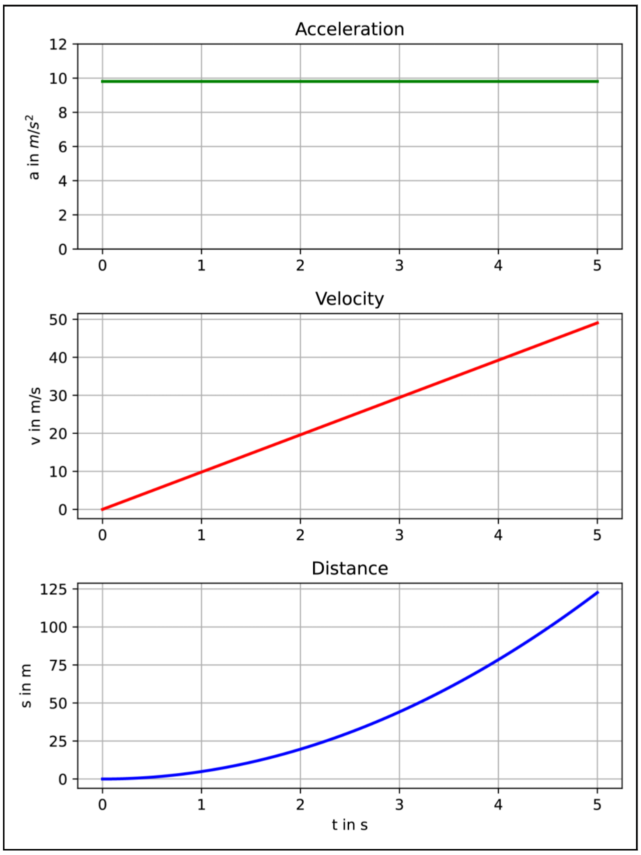


Figure 6.12 Free Fall: Acceleration, Velocity, and Distance

6.5.5 Improper Integral

Simply put, an improper integral is an integral where the integration limits can be between $-\infty$ and $+\infty$. For example, when a capacitor is charged, as shown in Figure 6.13, the charging process theoretically never ends. However, real-life everyday experience shows that a capacitor is charged after about five times the time constant RC.

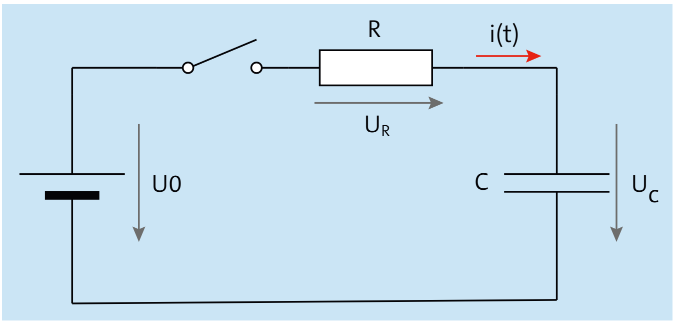


Figure 6.13 Charging an R-C Element

You can compute the stored electrical energy using the improper integral from the electrical power:

$$W_{el} = \int_0^\infty p(t)\mathrm{d}t$$

The NumPy constant `np.inf` is used as the upper limit for the term ∞. Listing 6.12 shows the implementation.

```python
#12_improper_integral.py
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate as integral
U0=10
R=1
C=1
tau=R*C
#voltage profile at the capacitor
def u(t):
    return U0*(1-np.exp(-t/tau))
#capacitor current
def i(t):
    return U0*np.exp(-t/tau)/R
#power
def p(t):
    return u(t)*i(t)
#upper limit->infinity
g=np.inf
t = np.linspace(0,5,1000)
W=integral.quad(p,0,g)[0]
print("Stored electrical energy:",W,"Ws")
fig, ax = plt.subplots()
ax.plot(t,u(t),"b-",lw=2,label="Voltage")
ax.plot(t,i(t),"r-",lw=2,label="Current")
ax.plot(t,p(t),"k-",lw=1,label="Power")
ax.fill_between(t,p(t),where=p(t)>=0,color='g',alpha=0.2)
ax.legend(loc="best")
ax.annotate(r"$W_{el}$",xy=(2,1),xytext=(1,10))
ax.set(xlabel="Time",ylabel="i, u, p")
plt.show()
```

Listing 6.12 Computing an Area under a Curve

Output

```
Stored electrical energy: 50.00000000000001 Ws
```

Figure 6.14 shows what the whole thing looks like in a function plot.

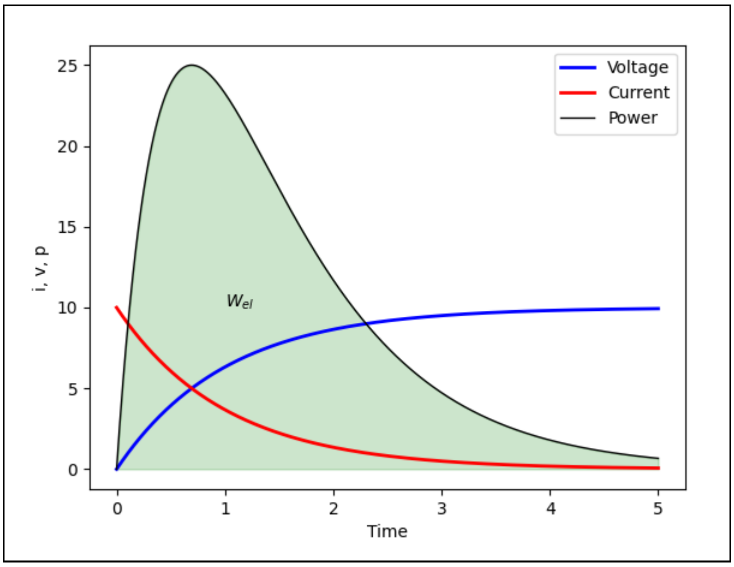


Figure 6.14 Stored Electrical Energy

Analysis

The custom functions `u(t)` and `i(t)` for the voltage and current profiles already contain the NumPy function `np.exp()` as an array, which is why they do not need to be vectorized anymore. Line 19 sets the limit `g=np.inf`. In line 21, this value is passed to SciPy function `integral.quad(p,0,g)[0]`. The zero `[0]` suppresses the output of the error estimation. The numerically calculated value for the stored electrical energy comes close to the theoretically calculated value of 50 Ws.

6.5.6 Calculating Arc Lengths

The calculation of arc lengths is also a typical task of the integral calculus. The following formula for the calculating arc lengths is derived from the Pythagorean theorem:

$$l = \int_a^b \sqrt{1 + [f'(x)]^2}\,\mathrm{d}x$$

A root expression must be integrated that contains the first derivative of a function *f*(*x*), which describes the course of the line. A typical application is the calculation of the length of a catenary, which is mathematically described by the *hyperbolic cosine* with the following curvature radius a :

$$f(x) = a \cosh \frac{x}{a}$$

Listing 6.13 shows how to calculate the length of a rope suspended between two posts with a distance of 20 meters.

```
01  #13_catenary.py
02  import numpy as np
03  import scipy.integrate as integral
04  from numdifftools import Derivative
05  a=10 #curvature radius
06  def k(x):
07      return a*np.cosh(x/a)
08  #calculate length
09  def dl(x):
10      df=Derivative(k)
11      return np.sqrt(1+df(x)**2)
12  #Distances
13  x1,x2=-10,10
14  length=integral.quad(dl,x1,x2)[0]
15  print("Length of the catenary %3.2f m" %length)
```

Listing 6.13 Arc Length of a Catenary

Output

```
Length of the catenary 23.50 m
```

Analysis

Lines 06 and 07 contain the function definition for catenary `a*np.cosh(x/a)`. In lines 09 to 11, the `dl(x)` function is defined for a line element of the catenary. In line 10, the `Derivative(k)` function calls the custom Python function `k(x)` from line 06. The `x` parameter must be omitted. The data of the `cosh` function is stored in the `df` object. In line 11, the derivative is squared. Line 14 uses the `quad(dl,x1,x2)[0]` function to calculate the defined integral of the catenary for the given limits between ±10 m. Thus, the chain is 3.5 m longer than the distance.

6.5.7 Volume and Surfaces of Rotating Bodies

The volume of a symmetrical body rotating the x-axis can be calculated using the following integral:

$$V_x = \pi \int_a^b [f(x)]^2 \mathrm{d}x$$

The following equation applies to the curved surface:

$$M_x = 2\pi \int_a^b f(x)\sqrt{1+[f'(x)]^2}\mathrm{d}x$$

Listing 6.14 shows how to calculate the volume and the surface area of a rotating paraboloid within the limits from 0 to 1.

```
01 #14_integral7.py
02 import numpy as np
03 import scipy.integrate as integral
04 from numdifftools import Derivative
05 #function definition
06 def f(x):
07     #return x
08     return np.sqrt(x)
09 #square function f(x)
10 def f2(x):
11     return f(x)**2
12 #for curved surface
13 def m(x):
14     df=Derivative(f)
15     return f(x)*np.sqrt(1+df(x)**2)
16 #limits
17 a,b=0,1
18 V=np.pi*integral.quad(f2,a,b)[0]  #volume
19 M=2*np.pi*integral.quad(m,a,b)[0] #curved surface
20 print("Volume      %3.6f" %V)
21 print("Curved surface %3.6f" %M)
```

Listing 6.14 Volume and Curved Surface of Rotating Bodies

Output

```
Volume      1.570796
Curved surface 5.330414
```

Analysis

In line 18, the volume `V` and in line 19, the curved surface `M` of the rotating paraboloid is calculated for the limits 0 to 1 using the `quad()` function. The commented-out function in line 07 (`f(x)=x`) creates a cone that rotates around the x-axis. This function makes it easy to check if the program calculates correctly.

6.5.8 Double Integrals

A typical application for a double integral is the calculation of the second area moment. The second area moment indicates the stiffness of a beam based on its cross-sectional area. For the second area moment of a rectangular beam of width b and height h, the following equation applies:

$$I_y = \int\limits_{z=-\frac{h}{2}}^{\frac{h}{2}} \left(\int\limits_{y=-\frac{b}{2}}^{\frac{b}{2}} z^2 \, \mathrm{d}y \right) \mathrm{d}z = \int\limits_{-\frac{h}{2}}^{\frac{h}{2}} b \cdot z^2 \, \mathrm{d}z = \frac{b \cdot h^3}{12}$$

SciPy calculates a double integral using the following function:

```
dblquad(func, a, b, c, d, ...)
```

In this context, the `func` parameter expects a mathematical function of the form $z = f(x,y)$. Parameters `a` and `b` represent the outer integration limits, and parameters `c` and `d` represent the inner integration limits.

Listing 6.15 calculates the second area moment for a beam with a rectangular cross-section. The girder has a width of 5 centimeters and a height of 10 centimeters.

```
01  #15_double_integral.py
02  import scipy.integrate as integral
03  b=5   #width in cm
04  h=10  #height in  cm
05  #function definition
06  def f(y,z):
07      return z**2
08  #calculation
09  Iy=integral.dblquad(f,-h/2,h/2,-b/2,b/2)[0]
10  print("Iy =",Iy,"cm^4")
11  print("Iy =",b*h**3/12,"cm^4 exactly")
```

Listing 6.15 Volume Calculation with Double Integral

Output

```
Iy = 416.66666666666674 cm^4
Iy = 416.6666666666667 cm^4 exactly
```

Analysis

In lines 06 and 07, the Python function `f(y,z)` is used to define the second area moment. In line 09, the SciPy function `dblquad(f,-h/2,h/2,-b/2,b/2)[0]` calculates the second area moment `Iy`. The accuracy of the result is stunning.

For further program testing, try reversing the order of the arguments in line 06 and the integration limits in line 09.

6.5.9 Triple Integrals

Triple integrals have the following general form:

$$\int_e^f \int_c^d \int_a^b u(x,y,z)\mathrm{d}x\mathrm{d}y\mathrm{d}z$$

A typical example of using a triple integral is the calculation of the air mass for a given base area and height of an air column. The density ρ of the air decreases exponentially with increasing height h:

$$\rho = \rho_0 e^{-\alpha h}$$

In this context, α stands for:

$$\alpha = \frac{\rho_0}{p_0} g$$

The mass m of an air column with the base area ab and the height h is calculated using the following the triple integral:

$$m = \int_0^h \int_0^b \int_0^a \rho_0 e^{-\alpha z}\mathrm{d}x\mathrm{d}y\mathrm{d}z = \frac{ab\rho_0}{\alpha}(1 - e^{-\alpha h})$$

SciPy calculates triple integrals using the following function:

```
tplquad(func, x1, x2, y1, y2, z1, z2, ...)
```

The `func` parameter expects a mathematical function of the following form:

$$u = f(z,x,y)$$

The `x1` and `x2` parameters define the integration limits on the x-axis. The `y1` and `y2` parameters determine the integration limits on the y-axis. The `z1` and `z2` parameters define the integration limits on the z-axis.

Listing 6.16 calculates the mass of an air column for a height of 8,223 meters with a base area of $1\mathrm{m}^2$ and compares the result with the analytically calculated value.

```
#16_triple_integral.py
import numpy as np
import scipy.integrate as integral
g=9.81      #gravitational acceleration
rho_0=1.28 #air density
p0=10e5     #air pressure
alpha=g*rho_0/p0

```

```
09  def density(z,x,y):
10      return rho_0*np.exp(-alpha*z)
11
12  a=1 #x2
13  b=1 #y2
14  h=8.223e3 #z2 height of the air column in m
15  #mass of the air column
16  m1=a*b*rho_0*(1-np.exp(-alpha*h))/alpha
17  #x1,x2,y1,y2,z1,z2
18  m2=integral.tplquad(density,0,a,0,b,0,h)[0]
19  print("Mass of air column m1:",m1, "kg")
20  print("Mass of air column m2:",m2, "kg")
```

Listing 6.16 Triple Integral

Output

```
Mass of air column m1: 10000.269980245075 kg
Mass of air column m2: 10000.269980245075 kg
```

Analysis

In line 09, the `density(z,x,y)` function is defined for the calculation of the air mass. An important requirement is that the function arguments are passed in the specified order. The SciPy function `tplquad(density,0,a,0,b,0,h)[0]` calculates the air mass for a height of 8,223 meters on a footprint of $1m^2$ in line 18. The comparison with the calculation in line 16 shows that the triple integral was calculated correctly.

6.6 Solving Differential Equations Numerically

Numerous types of differential equations cannot be solved analytically. These equations include, among others, nonlinear differential equations. However, these types of differential equations can be solved numerically with sufficient accuracy and relatively little effort. A particular advantage of numerical solution methods is that the initial values are a necessary part of the solution algorithm. Thus, the initial value problem no longer needs to be solved explicitly. Another advantage of numerical solution methods is that the calculated solution set is available as an array and can thus be visualized using the `plot` method.

6.6.1 Numerical Solution of Differential Equations

If a differential equation is to be solved numerically, the equation must first be converted into the explicit form:

$$y' = f(x, y)$$

In other words, y' must be on the left side of the equal sign, and the term of the differential equation must be on the right side of the equal sign (rhs). For starters, let's consider an analytical, easily solvable linear first-order differential equation:

$$y' = \frac{\mathrm{d}y}{\mathrm{d}x} = x \cdot y$$

This equation will serve as a test function for checking the accuracy of the numerical solution methods. The solution function $y = f(x)$ can be determined by separating the variables:

$$y = e^{\frac{x^2}{2}}$$

The simplest numerical method for solving first-order differential equations is the *Euler method*. The basic idea of this method to approximate the course of the solution function with a polygonal line.

The Euler algorithm computes the values for the independent variable x_k from the k-fold of increment h:

$$x_k = x_0 + k \cdot h$$

The algorithm can be derived from the following approach:

$$\frac{y_{k+1} - y_k}{h} = f(x_k, y_k)$$

By rearranging, we obtain the following:

$$y_{k+1} = y_k + f(x_k, y_k)h$$

The respective new discrete function value y_{k+1} of the solution function is calculated using the sum algorithm from the sum of the old function value y_k and the slope $f(x_k, y_k)$:

Thus, the slopes are added to the respective function values of the solution function. Listing 6.17 and Figure 6.15 visualize this process.

```
#17_differential_equation1.py
import math as math
import matplotlib.pyplot as plt

def f(x,y):
    return x*y

x0=0
xk=2
y=1    #initial value
h=0.25 #increment
n=int((xk-x0)/h)
lx,lyu,lyg =[],[],[]
```

```
14 for k in range(n):
15     x=x0+k*h
16     y=y+h*f(x,y) #Euler method
17     yg=math.exp(x**2/2) #exact
18     lx.append(x)
19     lyu.append(y)
20     lyg.append(yg)
21 fig, ax = plt.subplots()
22 ax.plot(lx,lyu,"b--",label="inexact")
23 ax.plot(lx,lyg,"r-",label="exact")
24 ax.set(xlabel="x",ylabel="y")
25 ax.legend(loc="best")
26 plt.show()
```

Listing 6.17 Numerical Integration according to the Euler Method

Output

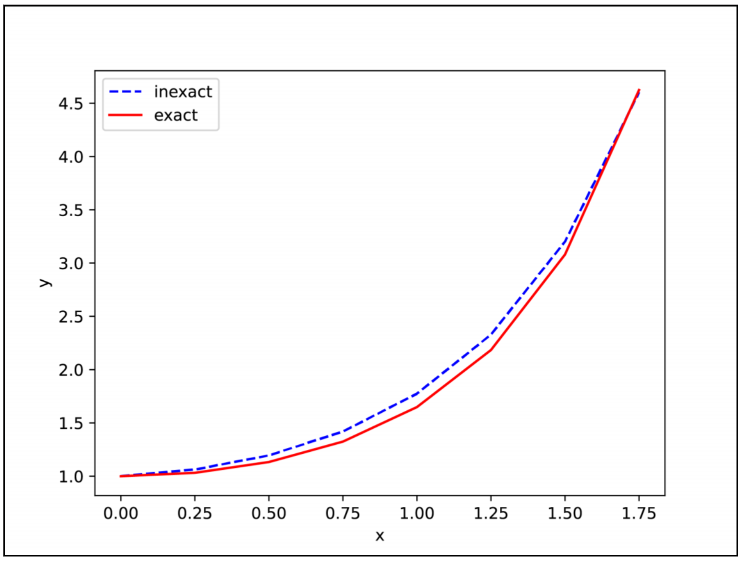


Figure 6.15 Numerical Solution of a Differential Equation Based on the Euler Method

Analysis

Line 11 defines the increment h. In this case, the increment must not be too small, so that the polygonal lines are also clearly visible. The empty lists created in line 13 are used to store the results for saving the exact values and the values calculated using the Euler method. In lines 14 to 20, the calculations within the for loop are then performed, and their results are stored in the empty lists. In lines 22 and 23, the output is done using the plot method. Clearly visible is the fact that the values calculated using the Euler method are somewhat smaller than the exact values of the solution.

Comparing SciPy Functions: odeint versus solve_ivp

The `integrate` submodule provides a variety of functions and methods for the numerical solution of differential equations. Since almost all methods are based on the *Runge–Kutta algorithm*, I want to introduce it briefly at this point.

The fourth-order Runge–Kutta method is based on the following algorithm:

$$y_{k+1} = y_k + \frac{h}{6}(a_k + 2b_k + 2c_k + d_k)$$

For this algorithm, the following values are given:

$$a_k = f(x_k, y_k)$$
$$b_k = f\left(x_k + \frac{h}{2},\ y_k + h\frac{a_k}{2}\right)$$
$$c_k = f\left(x_k + \frac{h}{2},\ y_k + h\frac{b_k}{2}\right)$$
$$d_k = f(x_k + h,\ y_k + hc_k)$$

The Runge-Kutta algorithm exists in many different variants. As you can easily see from its structure, it optimizes the Euler algorithm in terms of accuracy but worsens runtimes. When you apply these algorithms, you must be aware of the restriction that only first-order differential equations can be solved with them. Higher-order differential equations must be transformed into a first-order system, and they must be in the explicit form.

Note

- All higher-order differential equations must be transformed into a first-order differential equation system.
- These differential equations must be in explicit form.

You can use the SciPy functions `odeint()` and `solve_ivp()` to solve differential equations and systems of differential equations numerically. The `odeint()` function is an interface function that accesses the ODEPACK Fortran library. From this library, `odeint()` uses the LSODA solution method, which switches between optimal solution methods in case of problem behavior (numerical instability, too large error).

However, the `odeint()` function is now considered deprecated. The SciPy documentation recommends using the `solve_ivp()` function for new code instead. Although `odeint()` has been declared obsolete, both methods will be compared here.

The `odeint(func, y0, t, args=(...), ...)` function requires at least three parameters: The `func` name of a first-order differential equation is passed as the first parameter. The second parameter `odeint()` expects an array with the initial values `y0`. The third parameter represents the independent variable `t`. The fourth parameter, which is optional, can contain additional arguments necessary for the solution of the differential equation.

The SciPy function `solve_ivp(fun, t_span, y0, method='RK45', ...)` must be passed at least the first three arguments when called: The first parameter `fun` stands for the name of the differential equation. The second parameter `t_span` is the integration interval. The third parameter to be passed (`y0`) must be an array with the initial values of the differential equation. The default value for the integration method is the Runge-Kutta method `RK45` of order 5(4).

Listing 6.18 solves the differential equation $y' = xy$ in the interval `[0,1]` using `odeint()` and `solve_ivp()`. The exact value is also output so that the accuracy of both integration methods can be compared.

```python
#18_differential_equation_comparison.py
import numpy as np
from scipy.integrate import odeint,solve_ivp

def dgl(x,y):
    dy_dx=y*x
    return dy_dx

n=5
xmax=1
y0=[1]      #initial value
xi=[0,xmax] #integration range
x = np.linspace(0,xmax,n)
#solutions
y1 = np.exp(x**2/2)#exact
y2 = odeint(dgl,y0,x)
#methods for solve_ivp
#RK45, RK23, DOP853, Radau, BDF, LSODA
z = solve_ivp(dgl,xi,y0,method='RK45',dense_output=True)
y3 = z.sol(x)
#comparison
print("Exactly :",y1)
print("odeint:",y2.reshape(n,))
print("ivp    :",y3.reshape(n,))
```

Listing 6.18 Comparison of odeint() and solve_ivp()

Output

```
Exactly : [1. 1.03174341 1.13314845 1.32478476 1.64872127]
odeint:   [1. 1.03174355 1.13314863 1.32478499 1.6487216 ]
ivp   :   [1. 1.03209666 1.13355731 1.32438841 1.64883123]
```

Analysis

A comparison of the outputs clearly shows that the deprecated interface function `odeint()` provides much more accurate values than the SciPy function `solve_ivp()`. However, you can fix this deficiency by passing two additional arguments to the `solve_ivp()` function in line 19, namely, the constant for the relative tolerance `rtol=1e-12` and the constant for the absolute tolerance `atol=1e-12`. In this case, the `solve_ivp()` function calculates the exact value. You can use the commented-out solution methods in line 18 for testing purposes in line 19.

The `dense_output=True` parameter and the `y3 = z.sol(x)` statement in line 20 are explained in the next example.

In lines 23 and 24, the outputs are linearized. You should test the outputs also without linearization by using the NumPy method `flatten()`.

6.6.2 First-Order Linear Differential Equation

In the following sections, I'll describe how a linear first-order differential equation can be solved using the SciPy function `solve_ivp()`. For this purpose, our example involves a series circuit of a resistor and an inductor, as shown in Figure 6.16.

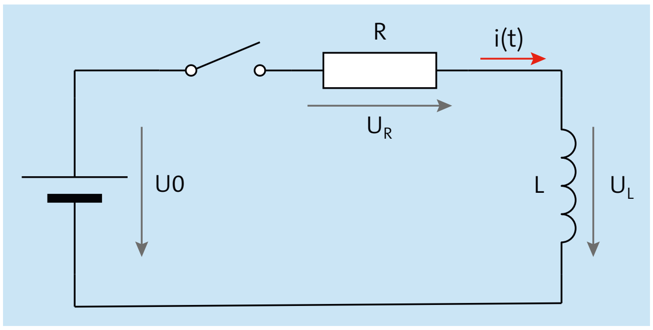


Figure 6.16 Activating a Coil

At time $t = 0$, the switch will be closed. We are searching for the current profile $i(t)$.

At any point in the transient state, the following equation applies:

$$u_L + u_R = U_0$$

With the voltage drops at the coil and at the resistor, we obtain the following equation:

$$L\frac{\mathrm{d}i}{\mathrm{d}t} + R \cdot i = U_0$$

By rearranging, we get the explicit form of the differential equation for the current profile:

$$\frac{\mathrm{d}i}{\mathrm{d}t} = \frac{U_0 - R \cdot i}{L}$$

Listing 6.19 solves the differential equation and displays it graphically, as shown in Figure 6.17.

```
01  #19_differential_equation3.py
02  import numpy as np
03  import matplotlib.pyplot as plt
04  from scipy.integrate import solve_ivp
05  U0=10
06  R,L=1,1
07  Tau=L/R
08  tmax=5*Tau
09  ti=[0,tmax] #integration interval
10  IL0 =[0]    #initial value
11  #First-order differential equation
12  def dgl(t,ia):
13      i=ia #initial value
14      di_dt=(U0-R*i)/L
15      return di_dt
16  #t.shape (500, )
17  t = np.linspace(0,tmax,500)
18  #solution of the differential equation
19  z = solve_ivp(dgl,ti,IL0,dense_output=True)
20  iL = z.sol(t) #iL.shape (1,500)
21  #Display of the solution
22  fig, ax = plt.subplots()
23  ax.plot(t, iL.flatten(),"r-",lw=2) #iL.flatten().shape (500, )
24  ax.set(xlabel="t",ylabel="$I_L(t)$")
25  ax.grid(True)
26  plt.show()
```

Listing 6.19 Activating a Coil

Output

The graphical output is shown in Figure 6.17.

Analysis

Line 09 sets the integration interval `ti=[0,tmax]`, and line 10 determines the initial value `IL0=[0]` of the current. Note that the initial values are enclosed in square brackets.

Line 19 solves the differential equation and stores the data of the solution in the `z` object. This object can then be used to access the `sol(t)` function in line 20. The solution vector is stored in the `iL` object and prepared for plotting in line 23 using the Matplotlib method `plot(t,iL.flatten())`. The NumPy method `flatten()` converts the two-dimensional `iL` array with the `shape (1,500)` into a one-dimensional array with the `shape (500, )`. If you do not perform this operation, you'll get an error message because, in the plot method, only one-dimensional arrays are allowed as arguments for the independent and dependent variables. You can determine the dimension and type (shape) of the arrays using print(name.ndim) and print(name.shape), respectively.

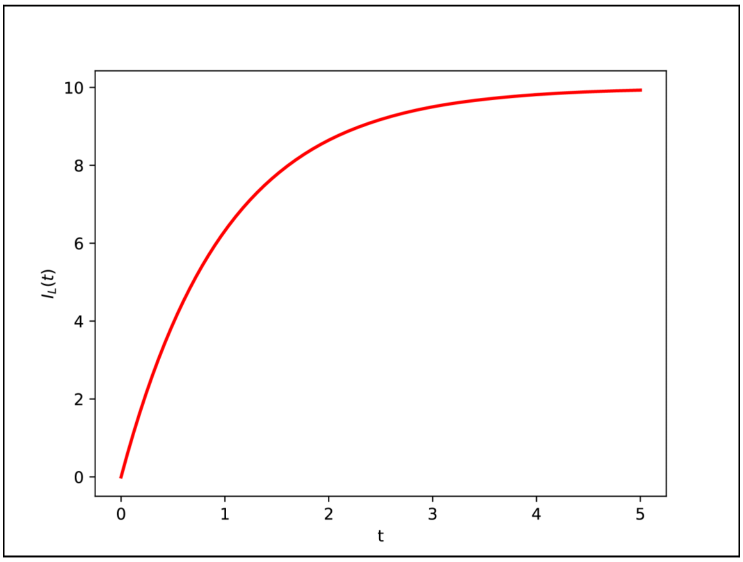


Figure 6.17 Activating a Coil

The dense_output=True parameter is of particular importance. The default value of dense_output is False. If you omit this parameter or set it to False, the following error message will appear:

```
TypeError: 'NoneType' object is not callable
```

This parameter allows you to use the sol(t) function. In the documentation, this function is referred to as an object (*Found solution as `OdeSolution` instance*). In fact, sol() should be referred to as a method because it accesses the z object. sol(t) computes the numerical solution of the differential equation for the time values specified in NumPy array t. You can use print(z) to display the contents of the z object:

```
message: The solver successfully reached the end of the integration interval.
  success: True
   status: 0
        t: [0.000e+00  1.000e-04 ... 5.000e+00]
        y: [[ 0.000e+00  1.000e-03 ... 9.735e+00  9.931e+00]]
      sol: <scipy.integrate._ivp.common.OdeSolution object at 0x141659c30>
 t_events: None
 y_events: None
     nfev: 56
     njev: 0
      nlu: 0
```

The contents of `z` can also be output individually in the Python shell via `print(z.message)`. You can find out the meaning of each statement by typing `help(solve_ivp)` in the Python shell.

6.6.3 Second-Order Linear Differential Equation

Based on our example series resonant circuit, let's now solve a second-order linear differential equation, as shown in Figure 6.18. We need to calculate the voltage profile $u_c(t)$ for the transient state (step response).

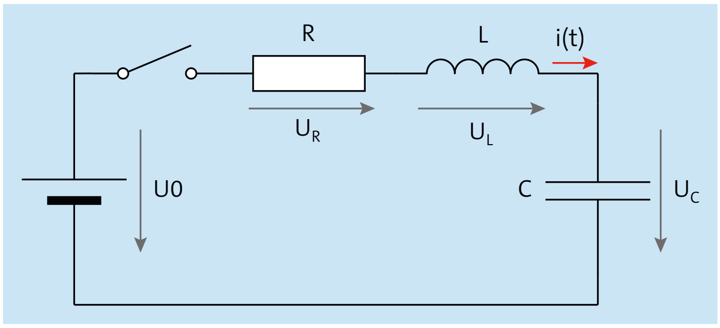


Figure 6.18 Equivalent Circuit Diagram for a DC Motor with Separate Excitation

Our series circuit of a resistor, an inductor, and a capacitor can model the dynamic behavior of a DC motor with separate excitation. The resistance R, the inductance L, and the capacitance C represent the ohmic resistance of the copper winding, the armature inductance, and the what's called the *dynamic capacitance* of a DC motor with separate excitation. The dynamic capacitance describes the mechanical inertia of the armature (rotor) of a DC motor. The rotational energy stored in the armature is equal to the electrical energy stored in a capacitor.

The dynamic capacitance is calculated from the moment of inertia of the working machine J, the rated armature current I_{AN}, and the rated torque M_N:

$$C = J\left(\frac{I_{\mathrm{AN}}}{M_N}\right)^2$$

The course of the rotational frequency $n(t)$ corresponds to the voltage profile at the capacitor $i_c(t)$. If the value of the battery voltage U_0 is 100V, then the normalized rotational frequency profile can be read on the y-axis as a percentage value for each time of the transient state.

To obtain the differential equation for the capacitor voltage, you must first set up the mesh equation for the voltage drops on the components:

$$u_L + u_R + u_C = U_0$$

The following applies for the capacitor current:

$$i = C\frac{\mathrm{d}u_C}{\mathrm{d}t}$$

The voltage drop across the coil is determined by the law of induction:

$$u_L = L\frac{\mathrm{d}i}{\mathrm{d}t}$$

Substituting the coil voltage into the mesh equation provides:

$$L\frac{\mathrm{d}i}{\mathrm{d}t} + Ri + u_C = U_0$$

And substituting the capacitor current into the mesh equation provides:

$$L\frac{\mathrm{d}i}{\mathrm{d}t} + RC\frac{\mathrm{d}u_C}{\mathrm{d}t} + u_C = U_0$$

By rearranging, you obtain a system of two differential equations. The first is for the capacitor voltage:

$$\frac{\mathrm{d}u_C}{\mathrm{d}t} = \frac{1}{C}i$$

The second is for the coil current:

$$\frac{\mathrm{d}i}{\mathrm{d}t} = \frac{1}{L}\left(U_0 - RC\frac{\mathrm{d}u_C}{\mathrm{d}t} - u_C\right)$$

Because this differential equation system was set up in explicit form, it can be coded directly as source code, as shown in Listing 6.20.

```
#20_differential_equation_second_order.py
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
U0 = 100  #input voltage in V
R = 1.5   #armature resistance in Ohms
L = 0.025 #armature inductance in H
Mn=150    #rated torque in Nm
In=50     #rated current in A
J=0.2     #moment of inertia in kgm^2
tmax=0.5  #time in seconds
#system of differential equations
def dgl(t,initial_values,R,L,C):
    uc,i = initial_values
    duc_dt = i/C
    di_dt = (U0 - R*C*duc_dt-uc)/L
    return [duc_dt, di_dt]

C = J*(In/Mn)**2 #dynamic capacitance
```

```
a0 = [0,0]         #initial values
ti=[0,tmax]        #integration interval
t = np.linspace(0,tmax,500)
z=solve_ivp(dgl,ti,a0,args=(R,L,C),dense_output=True)
uc,ic = z.sol(t)
fig,axes=plt.subplots(2,1,figsize=(6,6))
#capacitor voltage
axes[0].plot(t, uc,'b-',lw=2)
axes[0].set_title("Step response of a DC motor with separate excitation")
axes[0].set_ylabel('Output voltage in V')
#current_profile
axes[1].plot(t, ic,'r-',lw=2)
axes[1].set(xlabel='Time in seconds',ylabel='Armature current in A')
axes[0].grid(True);axes[1].grid(True)
print("Dynamic capacitance:",C, "F")
fig.tight_layout()
plt.show()
```

Listing 6.20 Step Response of a DC Motor with Separate Excitation

Output

```
Dynamic capacitance: 0.0222 F
```

The graphical output of the step response is shown in Figure 6.19.

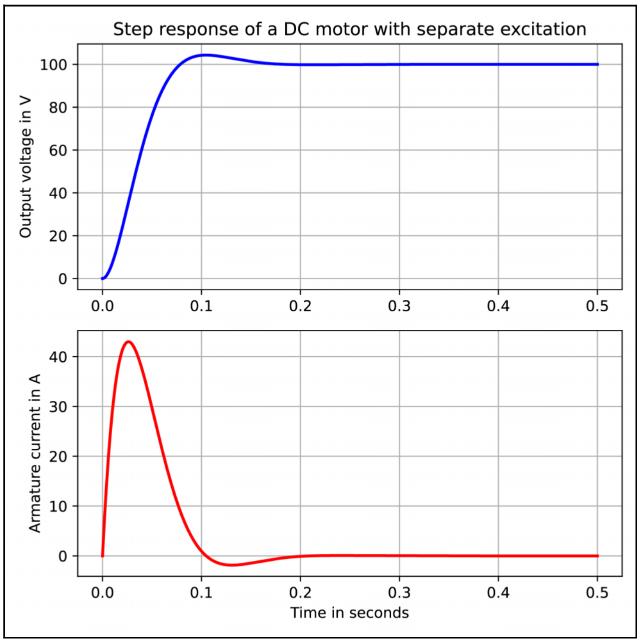


Figure 6.19 Step Response of a DC Motor with Separate Excitation

Analysis

In lines 13 to 17, the function for the differential equation system `dgl(t,initialval-ues,R,L,C)` is defined. This function returns the array `[duc_dt, di_dt]` as the solution set.

In line 23, this differential equation is solved. The additional parameters for the values of the components are passed as tuples `args=(R,L,C)`. Since the `z` object contains the solution set for the voltage and current profiles, they must be separated in line 24. The solution of the differential equation is stored in the tuple `uc, ic`.

6.6.4 Nonlinear Second-Order Differential Equation

The particular strengths of numerical solution methods are that they can also solve nonlinear differential equations. For example, the pendulum motion of a mathematical pendulum can be described by a nonlinear differential equation. We now want to simulate the movement of the pendulum shown in Figure 6.20. For this purpose, the differential equation of the oscillating system must be set up and converted into the explicit form.

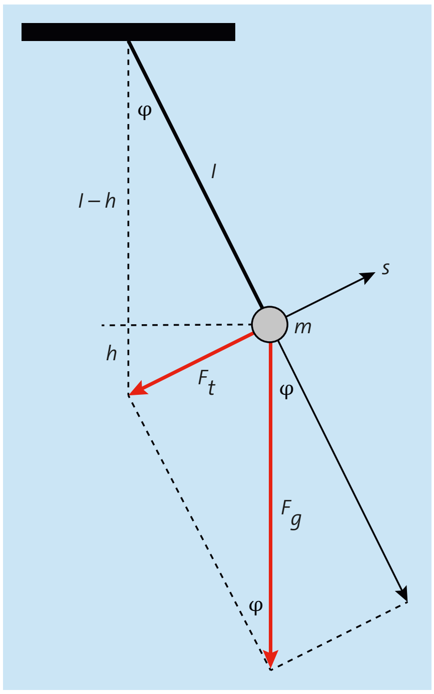


Figure 6.20 Simple Pendulum

A sphere of mass *m* is attached to a rod. The rod is assumed to be massless, while the entire mass of the system is concentrated as a point mass in the sphere. Bearing friction is also neglected. However, the air friction of the sphere, which is described by the following equation, should not be neglected:

$$F_r = -\frac{1}{2} \cdot \rho \cdot c_w \cdot A \cdot v^2 = -\frac{1}{2} \cdot \rho \cdot c_w \cdot A \cdot l^2 \cdot \omega^2$$

In detail, this equation means the following:

- ρ: air density 1.28kg/m^3
- c_w: drag coefficient
- A: cross-section of the sphere
- l: length of the pendulum
- ω: angular velocity of the pendulum in 1/s

The tangential force F_t acts against the acceleration:

$$F_t = -F_g \cdot \sin\varphi = -m \cdot g \cdot \sin\varphi$$

For the force component of the acceleration, the following equation applies:

$$F_b = m \cdot a = m \cdot l \cdot \ddot{\varphi}$$

At any point in time, the sum of all forces must equal zero:

$$F_b(t) + F_r(t) + F_t(t) = 0$$

By inserting the values, you obtain a nonlinear second-order differential equation:

$$m \cdot l \cdot \ddot{\varphi} \ + \frac{1}{2} \cdot \rho \cdot c_w \cdot A \cdot l^2 \cdot |\dot{\varphi}|\dot{\varphi} + m \cdot g \cdot \sin\varphi = 0$$

Dividing by ml provides the following result:

$$\ddot{\varphi} + \frac{\rho \cdot c_w \cdot A \cdot l}{2m}|\dot{\varphi}|\dot{\varphi} + \frac{g}{l}\sin\varphi = 0$$

Using the abbreviations $b = \frac{\rho \cdot c_w \cdot A \cdot l}{2m}$ for the friction and $\omega_0^2 = \frac{g}{l}$ for the angular frequency provides the following clear form:

$$\ddot{\varphi} + b|\dot{\varphi}|\dot{\varphi} + \omega_0^2 \ \sin\varphi = 0$$

The frequency of the pendulum movement decreases with the pendulum length and the period duration increases with the length of the pendulum:

$$f = \frac{1}{2\pi}\sqrt{\frac{g}{l}}$$

$$T = 2\pi\sqrt{\frac{l}{g}}$$

The pendulum swings back and forth on a circular path after being deflected because the potential energy of the sphere is converted into kinetic energy and the kinetic energy of the sphere is in turn converted into potential energy. This process is damped by air friction and bearing friction.

The sphere of the pendulum moves along a circular path. The maximum orbital velocity can be calculated from the energy balance with the following equation:

$$v_{\max} = \sqrt{2gl(1 - \cos\varphi_0)}$$

The second-order differential equation of the pendulum motion can be solved using the `solve_ivp()` function only if it is transformed into a system of two first-order differential equations.

With $\dot{\varphi} = \omega$, we can obtain the explicit first-order differential equation system:

$$\frac{\mathrm{d}\varphi}{\mathrm{d}t} = \omega$$

$$\frac{\mathrm{d}\omega}{\mathrm{d}t} = -b|\omega|\omega - \omega_0^2 \sin\varphi$$

As shown in Listing 6.21, we can simulate Foucault's pendulum, which was a famous experiment held at the Panthéon in Paris on March 26, 1851. This pendulum had a length of 67 meters and a spherical mass of 28 kilograms.

Although the deflection angle of 179° specified in line 07 is unrealistic for a pendulum length of 10 meters (line 05), it was deliberately chosen so that the nonlinearities of the motion sequence become clearly visible, as shown in Figure 6.21.

```
01  #21_differential_equation_simple_pendulum.py
02  import numpy as np
03  import matplotlib.pyplot as plt
04  from scipy.integrate import solve_ivp
05  l=10        #length of the pendulum in m
06  d=1         #diameter of the sphere in dm
07  phi0=179    #deflection
08  cw=0.3      #drag coefficient for the sphere
09  rho_K=7.85  #density of steel kg/dm^2
10  rho_L=1.28  #Density of air kg/m^3
11  g=9.81      #gravitational acceleration
12  tmax=50
13  #system of differential equations
14  def dgl(t,ya,b,w02):
15      phi, w = ya
16      dphi_dt = w
17      dw_dt = -b*np.abs(w)*w-w02*np.sin(phi)
18      return [dphi_dt,dw_dt]
19  #computations
20  r=d/2                   #radius of the sphere in dm
21  A=np.pi*(0.1*r)**2      #circle area in m^2
22  m=rho_K*4/3*np.pi*r**3  #mass of the sphere in kg
23  b=cw*rho_L*A*l/(2*m)    #damping constant
24  w02=g/l
```

```
25  T=2.0*np.pi*np.sqrt(l/g)
26  f=1.0/T    #frequency
27  vmax=np.sqrt(2*g*l*(1-np.cos(np.radians(phi0))))
28  #solution of the differential equation
29  y0 = [np.radians(phi0),0]
30  t = np.linspace(0, tmax, 500)
31  z=solve_ivp(dgl,[0,tmax],y0,args=(b,w02),dense_output=True)
32  phi, w = z.sol(t)
33  v=l*w
34  #Output
35  fig,ax=plt.subplots(2,1)
36  #Deflection
37  ax[0].plot(t, np.degrees(phi),'r-',lw=2)
38  ax[0].set(ylabel=r"$\varphi$ in °",title="Simple pendulum")
39  #Velocity
40  ax[1].plot(t, v,'b-',lw=2)
41  ax[1].set(xlabel='Time in s',ylabel="v in m/s")
42  [ax[i].grid(True) for i in range(len(ax))]
43  fig.tight_layout()
44  print("Mass of the sphere %3.2f kg"%m)
45  print("Period duration %3.2f s"%T)
46  print("Frequency %3.2f Hz"%f)
47  print("Damping %3.4f"%b)
48  print("Maximum velocity %3.2f m/s"%vmax)
49  plt.show()
```

Listing 6.21 Simple Pendulum

Output

Figure 6.21 shows the graphical output of the program.

```
Mass of the sphere 4.11 kg
Period duration 6.34 s
Frequency 0.16 Hz
Damping 0.0037
Maximum velocity 19.81 m/s
```

Analysis

The structure of the simulation program results from the previously derived formulas. At a deflection angle of 179°, the nonlinearity of the curves is clearly visible. If you reduce the angle in line 07 to about less than 90°, then the nonlinearity is already no longer visible to the naked eye. For the variation of the simulations, you can change the diameter of the sphere in line 06 and the density of the material in line 09.

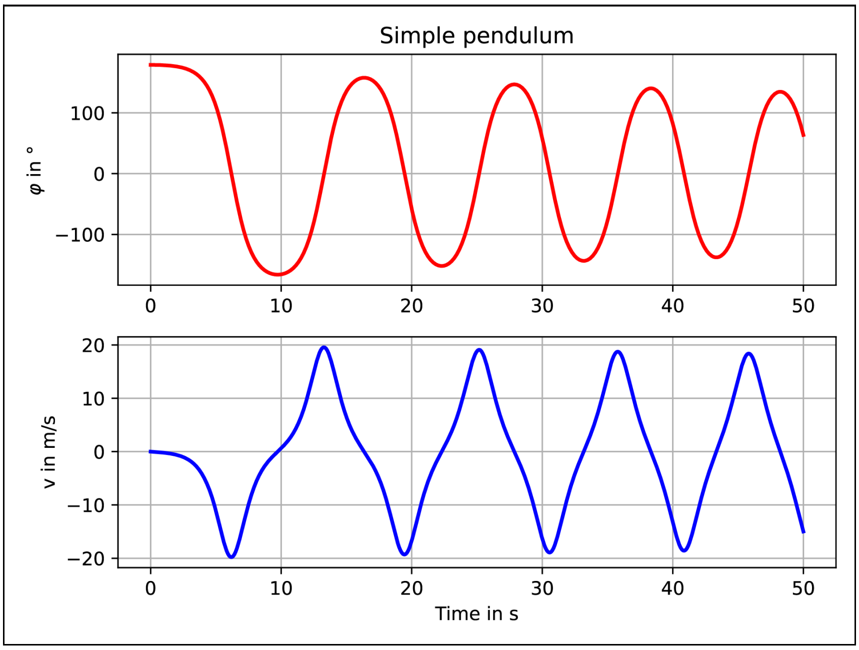


Figure 6.21 Deflection and Velocity of a Simple Pendulum

If you want to simulate Foucault's pendulum, then you must enter 67m for the pendulum length and 1.896dm for the sphere diameter. A deflection angle of 1.71° corresponds to a deflection of about 2m.

6.6.5 Second-Order Differential Equation System: Coupled Spring-Mass System

A coupled spring-mass system, shown in Figure 6.22, forms an oscillating system in which the two masses influence each other's movements.

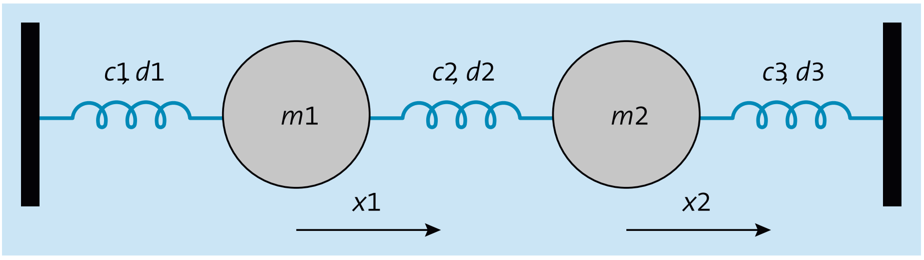


Figure 6.22 Two-Mass Oscillator with Damping

The masses m_1 and m_2 are supposed to only move horizontally along the x-axis, in the horizontal direction. The deflections x_1 and x_2 refer to the idle state. The physical behavior of the two springs is determined by the damping constants d_1, d_2, and d_3 and the spring constants c_1, c_2, and c_3 .

The spring-mass system is described by the following differential equation system (from Vöth: 80):

$$m_1\ddot{x}_1 = -d_1\dot{x}_1 - d_2(\dot{x}_1 - \dot{x}_2) - c_1x_1 - c_2(x_1 - x_2)$$
$$m_2\ddot{x}_2 = -d_3\dot{x}_2 - d_2(\dot{x}_2 - \dot{x}_1) - c_3x_2 - c_2(x_2 - x_1)$$

Both sides must be divided by the masses to obtain the explicit form:

$$\ddot{x}_1 = -\frac{1}{m_1}[d_1\dot{x}_1 + d_2(\dot{x}_1 - \dot{x}_2) + c_1x_1 + c_2(x_1 - x_2)]$$
$$\ddot{x}_2 = -\frac{1}{m_2}[d_3\dot{x}_2 + d_2(\dot{x}_2 - \dot{x}_1) + c_3x_2 + c_2(x_2 - x_1)]$$

With the substitutions

$$\dot{x}_1 = v_1\,, \qquad \ddot{x}_1 = \dot{v}_1, \qquad \dot{x}_2 = v_2, \qquad \ddot{x}_2 = \dot{v}_2$$

we obtain the following differential equation system:

$$\dot{x}_1 = v_1$$
$$\dot{v}_1 = -\frac{1}{m_1}[d_1v_1 + d_2(v_1 - v_2) + c_1x_1 + c_2(x_1 - x_2)]$$
$$\dot{x}_2 = v_2$$
$$\dot{v}_2 = -\frac{1}{m_2}[d_3v_2 + d_2(v_2 - v_1) + c_3x_2 + c_2(x_2 - x_1)]$$

Listing 6.22 solves this differential equation system using the `solve_ivp()` function and visualizes the motion sequences of the coupled spring-mass system, as shown in Figure 6.23.

```
#22_differential_equation_two-mass_oscillator.py
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
m=1e3     #mass in kg
c=1e7     #spring constant N/m
d=1e3     #damping kg/s
m1,m2=m,2*m
c1,c2,c3=c,2*c,3*c
d1,d2,d3=d,2*d,3*d
tmax=0.2 #seconds
#system of differential equations
def dgl(t,xa,c1,c2,c3,d1,d2,d3,m1,m2):
    x1,v1,x2,v2=xa
    dx1_dt=v1
    dv1_dt=-(d1*v1+d2*(v1-v2)+c1*x1+c2*(x1-x2))/m1
    dx2_dt=v2
    dv2_dt=-(d3*v2+d2*(v2-v1)+c3*x2+c2*(x2-x1))/m2
    return np.array([dx1_dt,dv1_dt,dx2_dt,dv2_dt])
```

```
20 #Solution of the differential equation system
21 t = np.linspace(0,tmax,500)
22 x0 = [0.1, 0.0, 0.0, 0.0]#initial values
23 parameter=(c1,c2,c3,d1,d2,d3,m1,m2)
24 z=solve_ivp(dgl,[0,tmax],x0,args=parameter,dense_output=True)
25 x1,v1,x2,v2 = z.sol(t)
26 fig,ax=plt.subplots()
27 #deflection m1
28 ax.plot(t, 1e3*x1,'r-',lw=1.5,label=r'$m_{1}$')
29 #deflection m2
30 ax.plot(t, 1e3*x2,'b-',lw=1.5,label=r'$m_{2}$')
31 ax.legend()
32 ax.set(xlabel='t in s',ylabel='Deflection in mm')
33 ax.grid(True)
34 plt.show()
```

Listing 6.22 Solution of the Differential Equation System for a Two-Mass Oscillator

Output

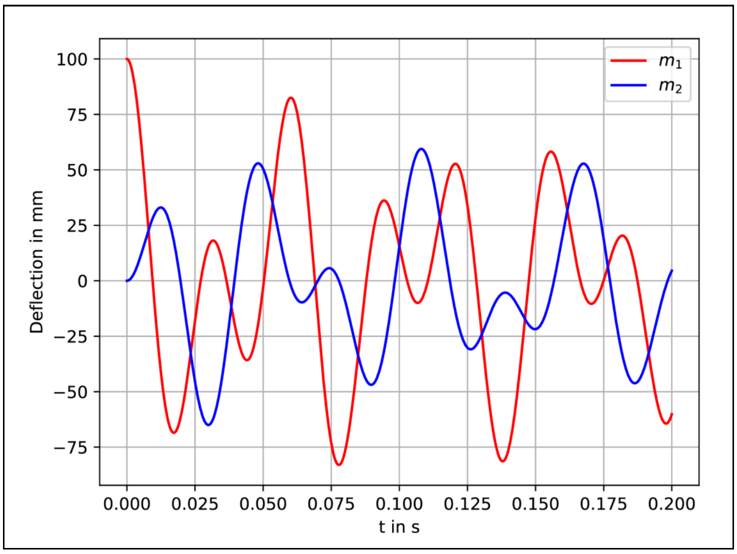


Figure 6.23 Motion Profile of a Two-Mass Oscillator

Analysis

The values for the masses, damping, and spring constants were taken from Vöth: 79f. (lines 05 to 10).

The function definition dgl() in lines 13 to 19 contains the code for the differential equation system in explicit form. The derivatives are returned as a NumPy array in line 19. This approach shortens the computation time compared to the return with lists.

In line 24, the `solve_ivp()` function solves the differential equation system. The solution data is stored in the `z` object. In line 25, the solutions are separated and stored in the tuple `x1,v1,x2,v2`.

The graphics area for the graphic representation of the mass deflections m_1 and m_2 begins in line 26. The factor `1e3` in lines 28 and 30 causes the values for the x-axis to be converted to ms and the values for the y-axis to be converted to mm.

Compared to the SymPy method `dsolve_system()`, the numerical calculation using the SciPy function `solve_ivp()` is much faster and more effective.

6.6.6 Nonlinear Second-Order Differential Equation System: Double Pendulum

In the next example, the motion sequences of a double pendulum will be simulated, as shown in Figure 6.24.

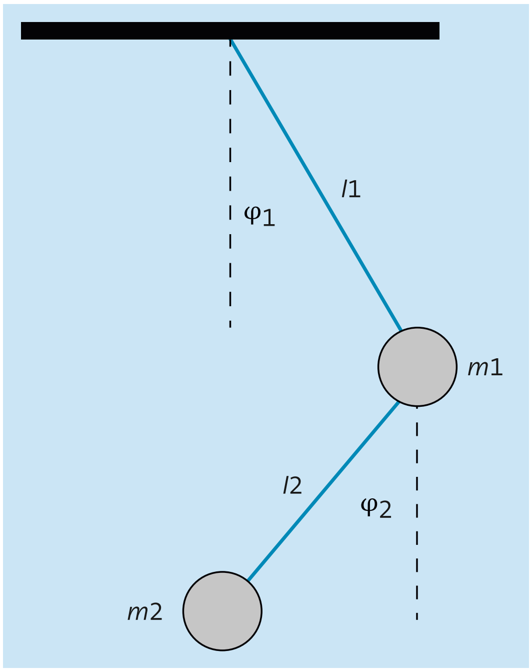


Figure 6.24 Double Pendulum

The two masses are connected by rigid rods. The bearing friction and air friction as well as the moment of inertia of the rods can be neglected.

A double pendulum is a complex oscillating system whose motion can be described by two nonlinear differential equations, in this scenario from *http://www.physics.usyd.edu.au/~wheat/dpend_html/*.

For the first pendulum, the following equation applies:

$$\dot{\varphi}_1 = \omega_1$$
$$\dot{\omega}_1 = \frac{m_2 l_1 \omega_1^2 \sin\Delta\cos\Delta + m_2 g \sin\varphi_2 \cos\Delta + m_2 l_2 \omega_2^2 \sin\Delta - mg\sin\varphi_1}{m l_1 - m_2 l_1 \cos^2\Delta}$$

For the second pendulum, the following equation applies:

$$\dot{\varphi}_2 = \omega_2$$
$$\dot{\omega}_2 = \frac{-m_2 l_2 \omega_2^2 \sin\Delta\cos\Delta + m(g\sin\varphi_1\cos\Delta - l_1\omega_1^2\sin\Delta - g\sin\varphi_2)}{m l_2 - m_2 l_2 \cos^2\Delta}$$

In this case, the abbreviations Δ and m stand for $\Delta = \varphi_2 - \varphi_1$ and $m = m_1 + m_2$.

Listing 6.23 solves this differential equation system using the `solve_ivp()` function and visualizes the deflections of a double pendulum in x and y direction, as shown in Figure 6.25, as a function of time and as a trajectory.

```
#23_differential_equation_double_pendulum.py
import numpy as np
from numpy import sin,cos
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
#pendulum data
g = 9.81    #gravitational acceleration
l1,l2 = 2,1 #pendulum lengths
m1,m2 = 5,1 #pendulum masses
phi1, phi2 = 120, -10 #deflection
tmax=20
#system of differential equations
def dgl(t,ya,l1, l2, m1, m2):
    phi1,w1,phi2,w2 = ya
    delta=phi2-phi1; m=m1+m2 #abbreviation for angle and mass
    phi1_dt = w1 #1. Derivative top angle
    w1_dt=(m2*l1*w1**2*sin(delta)*cos(delta)\
      +m2*g*sin(phi2)*cos(delta)+m2*l2*w2**2*sin(delta)-m*g*sin(phi1))\
           /(m*l1-m2*l1*cos(delta)**2)
    phi2_dt = w2 #1. Derivative bottom angle
    w2_dt=(-m2*l2*w2**2*sin(delta)*cos(delta)\
       + m*(g*sin(phi1)*cos(delta)-l1*w1**2*sin(delta)-g*sin(phi2)))\
           /(m*l2-m2*l2*cos(delta)**2)
    return np.array([phi1_dt, w1_dt, phi2_dt, w2_dt])
#Solution of the differential equation system
omega1 = omega2 = 0
ya =[np.radians(phi1),omega1,np.radians(phi2),omega2]
t = np.linspace(0,tmax,1000)
z=solve_ivp(dgl,[0,tmax],ya,args=(l1,l2,m1,m2),dense_output=True)
```

```
phi1, w1, phi2, w2 = z.sol(t) #solutions
#calculation of the x,y coordinates, l1 is anchored in the origin
x1,y1 =    l1*sin(phi1),  -l1*cos(phi1) #1st pendulum
x2,y2 = x1+l2*sin(phi2),y1-l2*cos(phi2) #2nd pendulum
fig, ax = plt.subplot_mosaic([['upper left', 'right'],
                              ['lower left', 'right']],
                              figsize=(8,4), layout="constrained")
#Deflection pendulum l1 (top)
width=1.1*(l1+l2)
ax['upper left'].plot(t,x1,'r-',lw=1)#x-direction top
ax['upper left'].plot(t,y1,'b-',lw=1)#y-direction top
ax['upper left'].set(ylabel='$x_1, y_1$',title='Pendulum 1')
#Deflection pendulum l2 (bottom)
ax['lower left'].plot(t,x2,'r-',lw=1) #x-direction bottom
ax['lower left'].plot(t,y2,'b-',lw=1) #y-direction bottom
ax['lower left'].set(xlabel='t',ylabel='$x_2, y_2$',title='Pendulum 2')
#Trajectories
width=1.1*(l1+l2)
ax['right'].plot(x1,y1,'r-',lw=1,label='Pendulum 1')
ax['right'].plot(x2,y2,'b-',lw=1,label='Pendulum 2')
ax['right'].set(xlabel='x',ylabel='y',title='Trajectories')
ax['right'].legend(loc='best')
ax['right'].set_xlim(-width,width)
ax['right'].set_ylim(-width,width)
plt.show()
```

Listing 6.23 Double Pendulum

Output

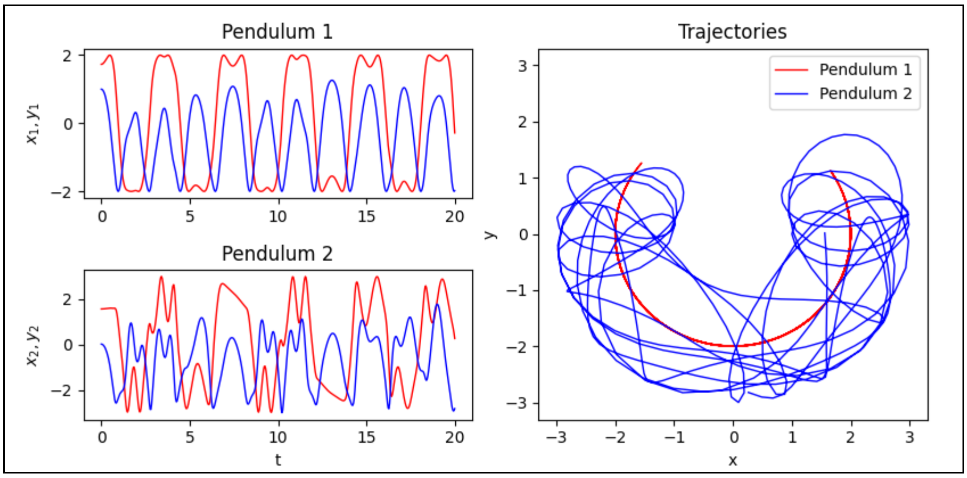


Figure 6.25 Motion Profile of a Double Pendulum

Analysis

As clearly indicated by the trajectories shown in Figure 6.25, pendulum 1 is forced to move on a circular path, while pendulum 2 performs chaotic movements. By changing the pendulum lengths in line 08 and the pendulum masses in line 09, you can perform various simulations.

In lines 13 to 24, the differential equation system `dgl` for the pendulum motions is implemented as required by the specifications. In line 29, the `solve_ivp()` function solves this differential equation system and stores the solution vector in the `z` object.

In line 30, the solutions are separated and stored in the tuple `phi1, w1, phi2, w2`.

In line 32, the x-y coordinates of the pendulum motion are calculated for pendulum 1 and in line 33 for pendulum 2.

6.7 Discrete Fourier Transform

Any periodic non-sine wave can be approximated by an infinite series of sinusoidal oscillations. For example, a rectangular periodic signal consists of a fundamental frequency and a sum of an infinite number of harmonics, whose frequencies consist of an odd multiple of the fundamental frequency. The amplitudes of the harmonic decrease by an odd fraction:

$$u(t) = \frac{4\hat{u}}{\pi}\left[1 + \sin\omega t + \frac{1}{3}\sin 3\omega t + \frac{1}{5}\sin 5\omega t + \ldots\right]$$

In amplifiers, sinusoidal signals (voltages or currents) are distorted by the nonlinear characteristic curves of the transistors. This results in harmonics. To assess the deviation of a signal from the sine wave, the term *total harmonic distortion* was introduced:

$$k_u = \frac{\sqrt{\hat{u}_2^2 + \hat{u}_3^2 + \hat{u}_4^2 + \hat{u}_5^2 + \ldots + \hat{u}_n^2}}{\hat{u}_1^2}$$

The smaller the total harmonic distortion, the better the “quality” of the sine wave.

For a rectangular function, the following equation applies:

$$k_u = \sqrt{\frac{\pi^2}{8} - 1} = 0.483$$

According to *Shannon’s sampling theorem*, the signal must be sampled with at least twice the frequency to detect these harmonics. Mathematically, the sampling process is described by the *discrete Fourier transform*. This transformation determines the frequency components of a non-sine wave. This approach transforms a signal $s(t)$ from the time domain to the frequency domain $\underline{S}(\omega)$, using the following algorithm:

$$\underline{S}(\omega) = \underline{S}\left(\frac{2\pi k}{NT_a}\right) = \sum_{n=0}^{N-1} s(nT_a)\, e^{-\frac{j2\pi kn}{N}}$$

The inverse Fourier transform is used to transform back into the time domain:

$$s(nT_a) = \frac{1}{N}\sum_{n=0}^{N-1} \underline{S}\left(\frac{2\pi k}{NT_a}\right) e^{\frac{j2\pi kn}{N}}$$

SciPy provides the Fourier transform functions in the `fft` subpackage. The `fft(x)` function transforms the `x` array from the time domain to the frequency domain. It calculates the one-dimensional discrete Fourier transform (DFT) for N sample points using the efficient **f**ast Fourier **t**ransform (FFT) algorithm. The first letter f (*fast*) in the identifier of the $fft()$ function refers to the efficiency of the algorithm.

The inverse `ifft(x)` function transforms the signal $\underline{S}(\omega)$ back to the time domain $s(t)$.

6.7.1 Basic Use of the Fast Fourier Transform Algorithm

Listing 6.24 shows how the discrete voltage values `u_t1` are transformed from the time domain to the frequency domain using the `fft(u_t1)` function and then transformed back to the time domain using the `ifft(U_fft)` function.

```
01  #24_fourier1.py
02  from scipy.fft import fft,ifft
03  u_t1=[0,1,2,3,4,5]
04  #transformation to the frequency domain
05  U_fft=fft(u_t1)
06  #transformation to the time domain
07  u_t2=ifft(U_fft)
08  print("Original signal :\n",u_t1)
09  print("Transformed signal:\n",U_fft)
10  print("Reconstructed signal:\n",u_t2.real)
```

Listing 6.24 Fourier Transform and Inverse Transform

Output

```
Original signal:
 [0, 1, 2, 3, 4, 5]
Transformed signal:
[15.-0.j  -3.+5.19615242j -3.+1.73205081j -3.-0.j
 -3.-1.73205081j -3.-5.19615242j]
Reconstructed signal:
 [0. 1. 2. 3. 4. 5.]
```

Analysis

In line 05, the fast Fourier transform function `fft(u_t1)` from the `fft` submodule transforms the `u_t1` list from line 03 to the frequency domain. The output in line 09 is an array of complex numbers whose practical usability cannot be seen at this point. In line

07, the inverse Fourier transform function `ifft(U_fft)` transforms the `U_fft` function from the frequency domain back to the time domain. The output in line 10 shows that the reverse transformation is exact. By specifying the `real` property, the outputs of the imaginary parts, which are zero anyway, are suppressed.

6.7.2 Frequency Spectra of Non-Sinusoidal Periodic Signals

In our next example, Listing 6.25 shows the computation of frequency spectra and the total harmonic distortion for rectangular, sawtooth, triangular, and parabolic signals using the Fourier transform. In line 43, the desired waveform can be tested by changing the function name.

```
01  #25_fourier_rectangle.py
02  import numpy as np
03  import matplotlib.pyplot as plt
04  from scipy.fft import fft,fftfreq
05  f=50      #frequency in Hz
06  T=1/f     #period duration
07  umax=325 #amplitude
08  N=6000    #number of samplings
09  Ta=T/N    #sampling time
10  t=np.linspace(0,T,N)
11  #compute total harmonic distortion
12  def total_harmonic_distortion(u):
13      z=0
14      for i in range(1,len(u)):
15          z=z+u[i]**2
16      return np.sqrt(z)/u[0]
17  #rectangle function
18  @np.vectorize
19  def ur(t):
20      if t < T/2:
21          return umax
22      else:
23          return -umax
24  #sawtooth
25  @np.vectorize
26  def us(t):
27      return 1e4*(t-T/2)
28  #triangle
29  @np.vectorize
30  def ud(t):
31      m=100
32      if t < T/2:
```

```
33          return m*t
34      else:
35          return -m*(t-T/2)+m*T/2
36  #parabolic arcs
37  @np.vectorize
38  def up(t):
39      if t < T/2:
40          return -(t-T/4)**2+(T/4)**2
41      else:
42          return (t-3*T/4)**2-(T/4)**2
43  u_t=ur(t)
44  #transformation to the frequency domain
45  U_fft = fft(u_t)
46  #magnitudes of the amplitudes in the frequency domain
47  U_f=2*np.abs(U_fft)/N
48  #compute harmonics
49  fk=fftfreq(N,Ta)
50  pos=np.where(fk>0) #only positive frequencies
51  #create subdiagrams
52  print("Total harmonic distortion %2.3f"%total_harmonic_distortion(U_f[pos]))
53  fig,ax=plt.subplots(2,1)
54  #display square wave display
55  ax[0].plot(1e3*t,u_t,"b-",lw=2)
56  ax[0].set(xlabel="t in ms",ylabel="U in V")
57  #display frequency spectrum
58  ax[1].stem(fk[pos],U_f[pos])
59  ax[1].set(xlabel="f in Hz",ylabel="Amplitudes")
60  ax[1].set_xlim(0,1000)
61  fig.tight_layout()
62  plt.show()
```

Listing 6.25 Computing Frequency Spectra of Non-Sinusoidal Signals

Output

Figure 6.26 shows the graphical output of the program.

```
Total harmonic distortion 0.483
```

Analysis

All custom functions must be vectorized via `@np.vectorize` so that they can be represented on the time axis. In line 43, the function call of the custom signal function takes place. By changing the function name, you can select the desired waveform (rectangle `ur(t)`, sawtooth `us(t)`, triangle `ud(t)`, or parabolic arcs `up(t)`) for the calculation of the total harmonic distortion and the display of the frequency spectrum.

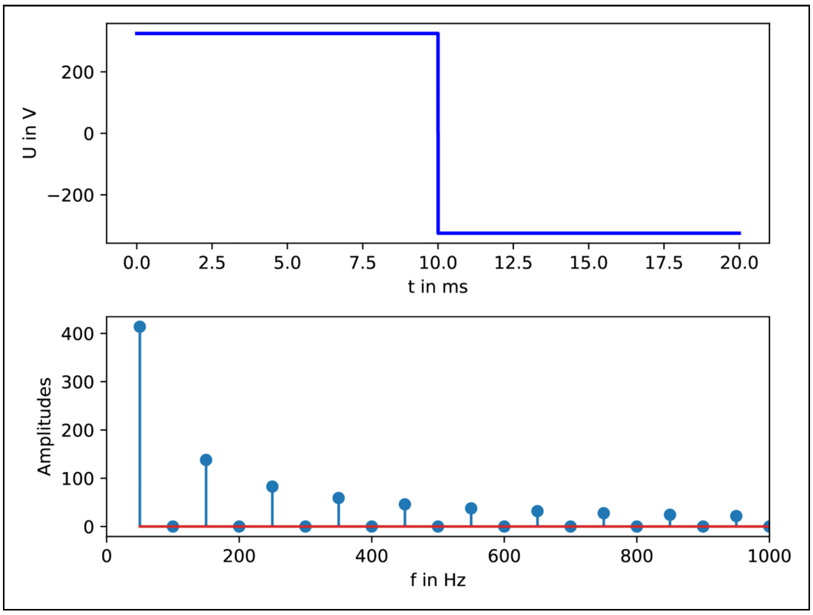


Figure 6.26 Frequency Spectrum of a 50 Hz Square Wave

The total harmonic distortion `(u)` function for the calculation of the total harmonic distortion is defined in lines 12 to 16. The algorithm sums up the individual amplitudes of the frequency spectrum (so-called *harmonics*) within a `for` loop. Note that the loop pass must start with index 1 so that the fundamental frequency `u[0]` is not added up with it.

In line 45, the SciPy function `fft(u_t)` transforms the time function `u_t` into the frequency domain. Since the transformation produces complex frequencies, the NumPy function `abs()` in line 47 computes the amounts of the amplitudes. In line 49, the `fftfreq(N,Ta)` function determines the harmonics from the number of sampling points `N` and from the sampling time `Ta`. In line 50, the NumPy function `where(fk>0)` suppresses all negative harmonics and stores them in the `pos` variable.

In line 55, the `plot` method prepares the display of the temporal signal wave `u_t`. The time axis was realistically scaled to milliseconds. In line 58, the matplotlib method `stem(x,y)` plots the amplitudes of the frequency spectrum (i.e., the harmonics) as vertical lines for each calculated x-y position, as shown in Figure 6.26.

The total harmonic distortion of 0.483 calculated for the square wave function corresponds to the theoretically determined value.

6.7.3 Reconstructing a Noisy Signal

If for any periodic non-sinusoidal function the frequency spectrum can be calculated using the Fourier transform, then it must also be possible to suppress unwanted frequencies (such as noise) in the frequency domain. Above a certain frequency, what's called the cutoff frequency f_g, these frequencies are no longer taken into account in the frequency domain by a comparison $f<f_g$ when transforming back into the time domain. This method allows you to suppress certain harmonics or noise. Listing 6.26 shows how a noisy sine wave is filtered in the frequency domain and transformed back into the time domain using the inverse Fourier transform `ifft(F)`.

```python
#26_low-pass_filter.py
import numpy as np
from matplotlib import pyplot as plt
from scipy.fft import fft,ifft,fftfreq
f=50     #frequency in Hz
T = 1/f  #period duration
fg=1.1*f #cutoff frequency
N=6000   #number of samplings
Ta=T/N   #sampling time
t = np.linspace(0,T,N)
u_t=10*np.sin(2*np.pi*f*t)+8*np.random.randn(t.size)
#transformation to the frequency domain
U_fft = fft(u_t)
#compute sampling frequency
fk = fftfreq(u_t.size,Ta)
#filter signal
F_g=U_fft*(np.abs(fk) < fg)
#transformation back to the time domain
u_g = ifft(F_g)
fig, ax=plt.subplots(2,1,figsize=(6,6))
#noisy signal
ax[0].plot(1e3*t, u_t)
ax[0].set(xlabel="t in ms",ylabel="u(t)",title="Noisy signal")
#filtered signal
ax[1].plot(1e3*t, u_g.real,lw=2)
ax[1].set(xlabel="t in ms",ylabel="u(t)",title="Filtered signal")
fig.tight_layout()
plt.show()
```

Listing 6.26 Simulation of a Low-Pass Filter

Output

The graphical output of the noisy and filtered signals is shown in Figure 6.27.

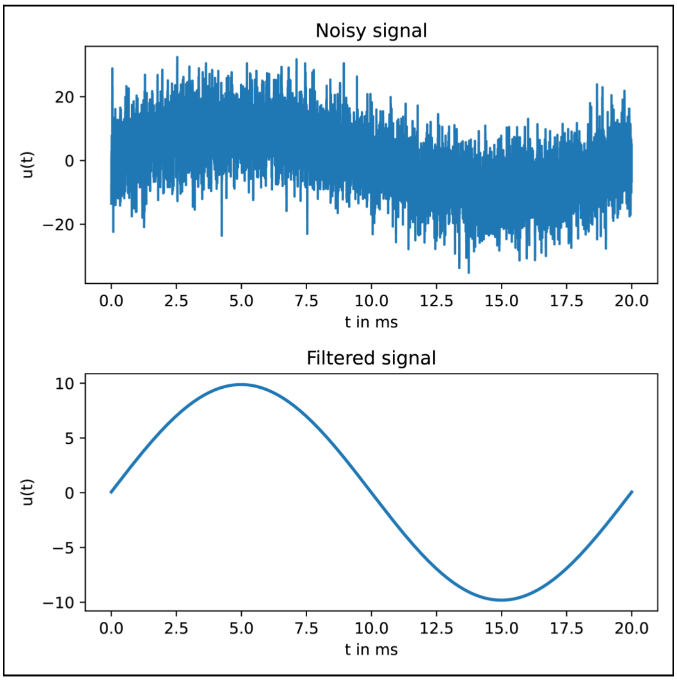


Figure 6.27 Reconstruction of a Noisy Signal

Analysis

In line 11, a sine function `u_t` with a superimposed noisy signal is defined. In line 13, the transformation into the frequency domain takes place.

The crucial operation takes place in line 17 where all harmonics greater than the cutoff frequency `fg` are suppressed by a simple mathematical comparison operation. In the `F_g` variable, only those harmonics that are below the cutoff frequency are stored.

The original sine wave is reconstructed by the reverse transformation into the time domain (line 19).

6.8 Writing and Reading Sound Files

SciPy has many submodules, classes, and functions that you can use to read and write data from various file formats (e.g., MATLAB or Fortran files). This section focuses on the treatment of the `write()` and `read()` functions of the `io` submodule for saving and reading sound files in the WAV format.

6.8.1 Generating and Saving Signals

The `write(filename,rate,data)` function expects three arguments when called: The first parameter `filename` creates a `wav` file from a NumPy array. As the second argument, the function expects the sampling rate `rate`. The third parameter (`data`) expects a NumPy array containing the sampled audio signals.

Listing 6.27 produces the concert pitch A (440 Hz). A program that can play WAV files can make this sound audible. The program can also be used for a hearing test if you increase the frequencies in line 05 step by step up to about 20,000 Hz.

```
01  #27_pitch_generation.py
02  import numpy as np
03  from scipy.io import wavfile
04  samplingrate = 44100
05  f=440 #frequency in Hz
06  t = np.linspace(0,1,samplingrate)
07  amp = np.iinfo(np.int16).max
08  pitch = amp*np.sin(2*np.pi*f*t)
09  print("Amplitude:",amp)
10  wavfile.write("sinus440Hz.wav", samplingrate, pitch)
```

Listing 6.27 Generating and Saving Concert Pitch A

Output

```
Amplitude: 32767
```

Analysis

Line 04 specifies that the signal is sampled 44,100 times in 1 second. Line 05 determines the frequency `f` of the pitch to be generated. If necessary, you can vary the frequency of the sound in this line. In line 06, 1 second must be entered for the duration of the tone because the sampling rate refers to the interval from 0 to 1 second. In line 07, you can also enter other values for the amplitude (`amp`) of the signal. By superimposing different sinusoidal oscillations, other sounds can also be generated in line 08. In line 10, the following SciPy function generates a sound file in WAV format:

```
wavefile.write("sinus440Hz.wav",samplingrate,pitch)
```

This file is saved in binary format as `sinus440Hz.wav` to the hard disk. The sampling rate (`samplingrate`) and the `pitch` object must be passed to the `wavefile.write()` function.

6.8.2 Reading and Displaying Signals

The `read(filename)` function is used to read WAV files. Listing 6.28 reads the signal data from the `sinus440Hz.wav` file and displays it graphically as a function of time, as shown in Figure 6.28.

```
01 #28_read_pitch.py
02 import numpy as np
03 import matplotlib.pyplot as plt
04 from scipy.io import wavfile
05 sampling_rate,data = wavfile.read("sinus440Hz.wav")
06 t = np.linspace(0, 1, sampling_rate)
07 fig, ax=plt.subplots()
08 ax.plot(t,data)
09 ax.set_xlim(0,0.01)
10 ax.set(xlabel="Time in seconds")
11 plt.show()
```

Listing 6.28 Reading and Graphically Displaying Sound Data

Output

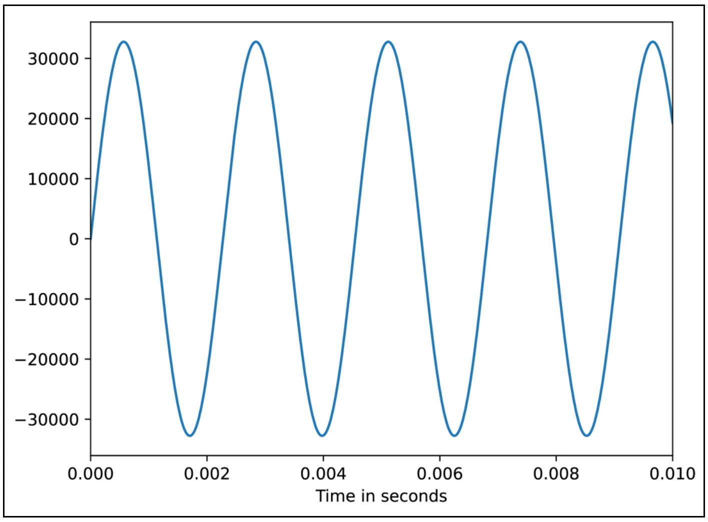


Figure 6.28 440 Hz Signal of a WAV File

Analysis

The output shows that the data of the sound file `sinus440Hz.wav` was read correctly: In 10 ms, about four periods of the 440 Hz signal with the amplitude of about 33,000 are displayed.

In line 05, the SciPy function `wavfile.read("sinus440Hz.wav")` reads the data of the `sinus440Hz.wav` file and saves it in the `samplingrate,data` tuple. The sampling rate (`samplingrate`) is needed in line 06 to define the distances on the t-axis.

In line 08, the coordinate data `t,data` is passed to the `plot` method. To ensure that individual oscillations of the signal remain recognizable, the t-axis was limited to a final value of 10 ms in line 09.

6.9 Signal Processing

The signal subpackage provides a variety of signal processing functions such as convolution, B-splines, spectral analysis, filter design, and many others. For more information, I recommend reading the SciPy documentation as I will describe only the Butterworth filter in this section.

6.9.1 Frequency Response of a Butterworth Lowpass

The following function computes the numerator and denominator coefficients of a Butterworth filter:

```
b,a=butter(N,Wn,btype='low',analog=False,output='ba',fs=None)
```

The meaning of the individual parameters is described in Table 6.2.

Parameter	Description
N	Degree of the filter
Wn	Cutoff frequency (-3 dB for Butterworth filters). For analog filters, Wn is an angular frequency (rad/s). For lowpass and highpass filters, Wn is a scalar. For bandpass and bandstop filters, Wn is an array containing the lower and upper cutoff frequencies.
btype	Type of filter: Options include lowpass, highpass, bandpass, and bandstop. The default setting is lowpass.
analog=False	Selects an analog or digital filter. If the parameter is omitted, a digital filter will be implemented.
output='ba'	Type of output: ■ ba: Numerator/denominator coefficients (default setting) ■ zpk: Poles and zeros ■ sos: Used for the filter function (**s**econd **o**rder **s**ection)
fs=None	Sampling frequency of a digital filter

Table 6.2 Parameters for a Butterworth Filter

To plot the frequency response of a filter, you'll need the w,h=freqs(b,a) function. This function returns two arrays with the values of the frequency response h calculated from the numerator and denominator coefficients b and a and the corresponding frequency values w as tuples. Using the Matplotlib method plot() or even better, the semilogx() method, the frequency response of an analog filter can be displayed as a function plot.

Listing 6.29 represents the frequency response of a third-degree Butterworth lowpass filter for the cutoff frequency 1 Hz with logarithmic scale division. The 'lowpass' parameter in line 08 specifies that a lowpass is supposed to be simulated.

```
01  #29_tp_frequency_response.py
02  import numpy as np
03  import matplotlib.pyplot as plt
04  from scipy.signal import butter,freqs
05  g=3     #degree of the filter
06  fg=1    #cutoff frequency in Hz
07  #numerator, denominator coefficients
08  b,a=butter(g,fg,'lowpass',analog=True)
09  #angular frequency, frequency response
10  omega,h_t = freqs(b,a)
11  fig, ax=plt.subplots(figsize=(8,6))
12  ax.semilogx(omega, 20*np.log10(abs(h_t)))
13  #ax.plot(omega, 20*np.log10(abs(h_t)))
14  ax.set_title('Butterworth lowpass')
15  ax.set_xlabel('f in Hz')
16  ax.set_ylabel('Amplitude in dB')
17  ax.margins(0, 0.1)
18  ax.grid(which='both', axis='both')
19  ax.axvline(fg, color='red') #cutoff frequency
20  plt.show()
```

Listing 6.29 Frequency Response of a Butterworth Lowpass

Output

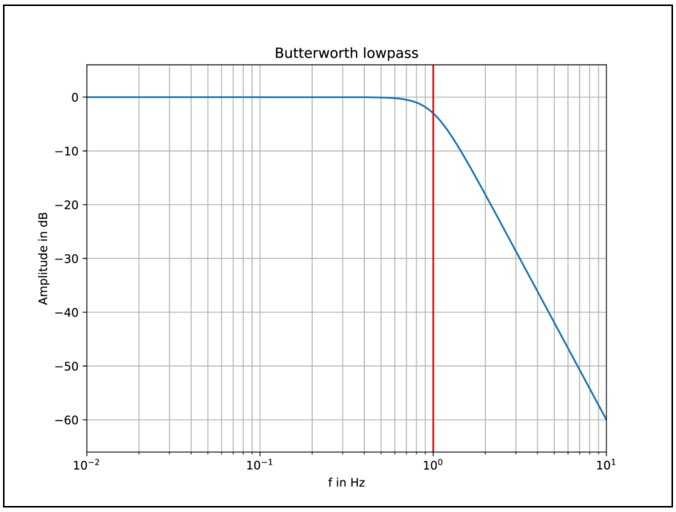


Figure 6.29 Frequency Response of a Butterworth Lowpass

Analysis

A third-degree lowpass has an attenuation of 60 dB per decade, which is confirmed by the function plot shown in Figure 6.29.

Strictly speaking, the cutoff frequency `fg` in line 06 is an angular frequency. However, since the angular frequency is plotted on the frequency axis, the "error" undoes itself.

In line 08, the `butter(g,fg,'lowpass',analog=True)` function computes from the `signal` submodule the coefficients `a` of the denominator polynomial and the coefficients `b` of the numerator polynomial of the Butterworth lowpass of the transmission function. These coefficients are used in line 10 by the `freqs(b,a)` function to compute the frequency response. The frequency response data is stored in the `omega,h_t` tuple and passed to the Matplotlib method `semilogx()` in line 12.

You can also use the `butter()` function to compute Butterworth coefficients for various filter types such as lowpass, highpass, bandpass, and bandreject. The results can be output as `numpy.ndarray` via the `print` function.

6.9.2 Frequency Response of a Crossover

Crossovers for three-way speaker systems filter the low frequencies for the woofer (up to about 500 Hz), the mid frequencies (about 500 to 5,000 Hz) for the midrange speaker, and the high frequencies (from about 5,000 Hz) for the tweeter from the entire signal from the amplifier.

Listing 6.30 simulates the frequency response of such a crossover with Butterworth filters.

```python
#30_crossover.py
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import butter,freqs
g=3       #degree of the filter
fgu=500   #lower cutoff frequency in Hz
fgo=5000 #upper cutoff frequency in Hz
#numerator, denominator coefficients
bt,at=butter(g,fgu,'lowpass',analog=True)
bb,ab=butter(g,[fgu,fgo],'bandpass',analog=True)
bh,ah=butter(g,fgo,'highpass', analog=True)
#angular frequency, frequency response
f1, ht_t = freqs(bt,at)
f2, hb_t = freqs(bb,ab)
f3, hh_t = freqs(bh,ah)
fig, ax = plt.subplots(figsize=(8,6))
ax.semilogx(f1,20*np.log10(abs(ht_t)))
ax.semilogx(f2,20*np.log10(abs(hb_t)))
```

```
19  ax.semilogx(f3,20*np.log10(abs(hh_t)))
20  ax.set_title('Crossover')
21  ax.set_xlabel('f in Hz')
22  ax.set_ylabel('Amplitude in dB')
23  ax.set_xlim(50,20e3)
24  ax.set_ylim(-20,3)
25  ax.margins(0, 0.1)
26  ax.grid(which='both', axis='both')
27  ax.axvline(fgu, color='red')
28  ax.axvline(fgo, color='red')
29  plt.show()
```

Listing 6.30 Frequency Response of a Crossover

Output

Figure 6.30 shows the frequency response of the crossover.

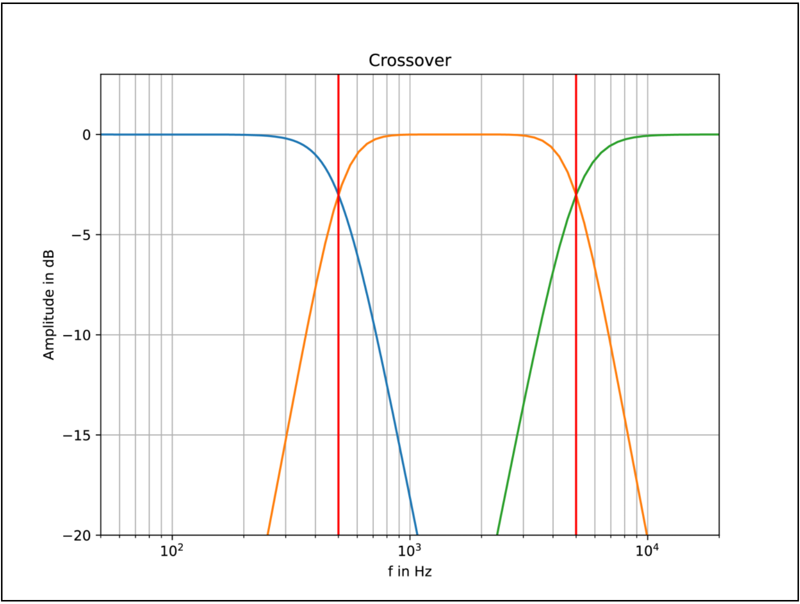


Figure 6.30 Frequency Response of a Crossover

Analysis

Basically, the program does not contain any new code. In lines 09 to 11, the `butter()` function computes the coefficients of the transmission functions for the lowpass, the bandpass, and the highpass. In lines 13 to 15, the `freqs()` function computes the frequency responses of the filters.

6.9.3 Filtering Signals

From an array x, the sosfilt(sos,x) function filters out the signal components specified by the filter coefficients sos of a certain filter characteristic. Listing 6.31 shows how a digital Butterworth lowpass filter filters out the low frequency from a 10 Hz and 30 Hz frequency mixing.

```
01  #31_tpfilter.py
02  import numpy as np
03  import matplotlib.pyplot as plt
04  import scipy.signal as signal
05  g=10  #degree of the filter
06  f1=10
07  f2=30
08  fs=1e3    #sampling frequency
09  fg=1.1*f1 #cutoff frequency
10  tmaxes=1
11  t = np.linspace(0,tmaxes,2000)
12  u_t = np.sin(2*np.pi*f1*t) + np.sin(2*np.pi*f2*t)/3
13  #u_t = 10*np.sin(2*np.pi*f1*t) + 5*np.random.randn(t.size)
14  TPK = signal.butter(g,fg,'low',analog=False,output='sos',fs=fs)
15  filter = signal.sosfilt(TPK, u_t)
16  fig, ax = plt.subplots(2, 1)
17  #mixed signal
18  ax[0].plot(t, u_t)
19  ax[0].set(ylabel='u(t)',title='10-Hz und 30-Hz Signal')
20  #filtered signal
21  ax[1].plot(t, filter)
22  ax[1].set(xlabel='t in s',ylabel='u(t)',title='Filtered signal:
 10 Hz')
23  plt.tight_layout()
24  plt.show()
```

Listing 6.31 Signal Filtered with a Lowpass Filter

Output

Figure 6.31 shows the distorted and the filtered signals.

Analysis

As expected, Figure 6.31 shows ten periods within a time of 1 second. The filtered signal therefore has a frequency of 10 Hz.

In line 14, the following function computes the data for the filtered signal and stores it in the TPK object:

```
butter(g,fg,'low',analog=False,output='sos',fs=fs)
```

The `output` parameter must be assigned the `sos` (*second-order sections*) value. As a result, the filter function of `butter()` will be activated. The lowpass is a digital filter because the `analog` parameter has not been set to `True`. The last parameter is assigned the sampling frequency `fs` defined in line 08. In line 15, the `sosfilt(TPK,u_t)` function suppresses all frequency components that are above the cutoff frequency of 11 Hz.

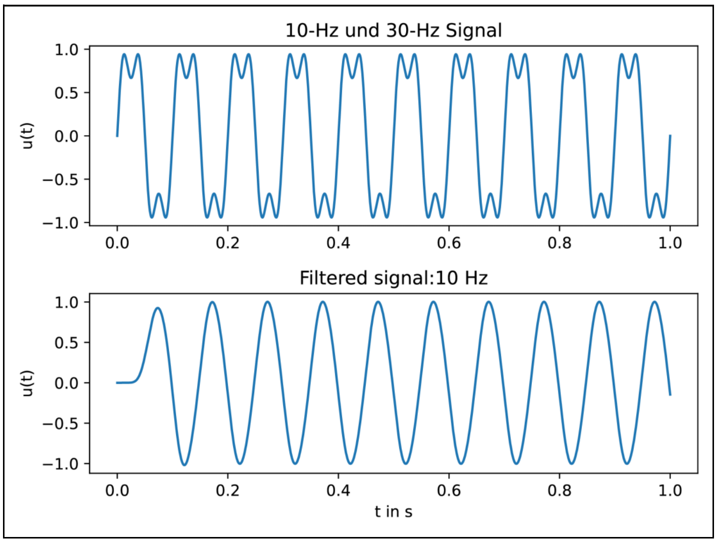


Figure 6.31 Signal Filtered with a Lowpass Filter

6.10 Project Task: Simulation of a Rolling Bearing Damage

Every machine tool generates mechanical vibrations during operation, generally caused by the rotary movements of its shafts. These vibrations can be captured by a microphone, converted to digital signals by an analog-to-digital converter, and displayed on a laptop screen using special software. These oscillations are accelerations that become visible on the screen as non-specific signals, which we call “noise” due to the large number of bearings and the complexity of the movements.

However, bearing damage generates additional machine noise that cannot be detected in the frequency mixing of the time signal. This scenario is exactly where the Fourier transform finds an important application. Special software can transform the noisy time signals of machine vibrations into the frequency domain. Then, the diagnosis of whether bearing damage is present at all is made based on the distribution of the amplitudes of the frequency spectrum (i.e., its harmonics). If only one particular expression of an amplitude exists, called the fundamental vibration, then the bearing is not damaged. In the case of bearing damage, several harmonics occur in addition to the fundamental frequency.

In this project task, we need to develop a program that simulates bearing damage. For this purpose, only two waveforms, one with a lot of harmonics and one with only a little, must be generated from sine functions. The SciPy function `fft()` then transforms both waveforms into the frequency domain. By displaying the harmonics, a diagnosis is then made as to whether a bearing is damaged. If the signals were available in WAV format, they could also be read using the `read` function and analyzed using the `fft()` function. Listing 6.32 simulates the frequency spectrum of a damaged bearing and compares it with the frequency spectrum of an undamaged bearing. The shaft rotates at a frequency of 1,200 rpm.

```
#32_bearing_damage.py
import numpy as np
import matplotlib.pyplot as plt
from scipy.fft import fft,fftfreq
f1=20 #1200 1/min
fn=[f1,2*f1,3*f1,4*f1,5*f1]
a1=[4,5,4,3,2,1] #defect
a2=[4,0.22,0.21,0.15,0.11] #undamaged
#Frequency
T=20
N=5000
Ta=T/N #sampling time
t = np.linspace(0,T,N)
#noisy signals
ur = a1[0]*np.sin(2*np.pi*fn[0]*t)\
    +a1[1]*np.sin(2*np.pi*fn[1]*t)\
    +a1[2]*np.sin(2*np.pi*fn[2]*t)\
    +a1[3]*np.sin(2*np.pi*fn[3]*t)\
    +a1[4]*np.sin(2*np.pi*fn[4]*t)\
    +np.random.normal(size=N)

ug = a2[0]*np.sin(2*np.pi*fn[0]*t)\
    +a2[1]*np.sin(2*np.pi*fn[1]*t)\
    +a2[2]*np.sin(2*np.pi*fn[2]*t)\
    +a2[3]*np.sin(2*np.pi*fn[3]*t)\
    +a2[4]*np.sin(2*np.pi*fn[4]*t)\
    +np.random.normal(size=N)
#transformation to the frequency domain
U_fftd = fft(ur) #defect
U_fftg = fft(ug)
fk=fftfreq(N,Ta)
pos=np.where(fk>0)
#amplitude magnitudes
Usd=2.0/N*np.abs(U_fftd) #defect
```

```
Usf=2.0/N*np.abs(U_fftg)
#Frequency spectrum
fig,ax=plt.subplots(3,1,figsize=(6,6))
#Time domain
ax[0].plot(t,ur,"b-",lw=1)
ax[0].set_xlim(0,10)
ax[0].set(xlabel="t in s",ylabel="a",title="Noisy signal")
#damaged bearing
ax[1].plot(fk[pos],Usd[pos],"r-",lw=2)
ax[1].set(xlabel="f in Hz",ylabel="a",title="Damaged bearing")
#undamaged bearing
ax[2].plot(fk[pos],Usf[pos],"g-",lw=2)
ax[2].set(xlabel="f in Hz",ylabel="a",title="Undamaged bearing")
fig.tight_layout()
plt.show()
```

Listing 6.32 Simulation of a Bearing Damage

Output

Figure 6.32 shows the frequency spectrum of an undamaged and a damaged bearing.

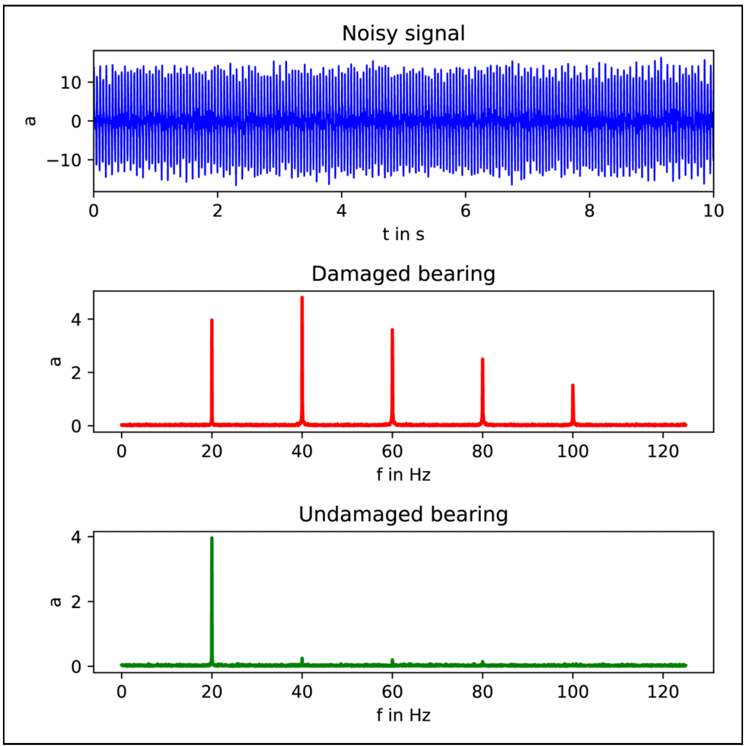


Figure 6.32 Frequency Spectrum of a Damaged and an Undamaged Rolling Bearing

Analysis

The rotation frequency of 20 Hz specified in line 05 corresponds to a speed of 1,200 rpm. In line 06, in the `fn` list, you can enter which harmonics should occur for the simulation of a damaged bearing. The `a1` list in line 07 contains the Fourier coefficients for the simulation of a damaged bearing, and the `a2` list in line 08 stores the Fourier coefficients for an undamaged bearing.

Lines 15 to 20 contain the code for the Fourier series of the defective bearing. Lines 22 to 27 contain the code for the Fourier series of the undamaged bearing. A noisy signal is superimposed on both signals to simulate the disturbances of real operation.

In lines 29 and 30, the `fft()` function transforms both time signals `ur` and `ug` into the frequency domain. In lines 34 and 35, the amplitudes for the amplitude spectra are calculated.

In the output shown in Figure 6.32, notice the clear difference between the defective bearing and the undamaged bearing. The undamaged bearing has only one amplitude, at 20 Hz, which is the specified rotational frequency, while in the case of the defective bearing, multiple harmonics are pronounced.

6.11 Project Task: Predator-Prey Model

Several animal species eat a purely plant-based diet. Other animal species (predators), on the other hand, depend on live animals (prey) as a food source. In this project task, the relationship between the predator and prey populations will be studied in its development over time. This development can be described by means of a first-order nonlinear differential equation system. The following idealized assumptions will be made:

- In the considered area, no in-migrations and no out-migrations of either population occur.
- Neither predators nor prey are decimated by pathogens.
- Predators specialize in one prey species.

For the derivation of the differential equation system, the notion of the *growth rate* is crucial. This rate is the number of births minus the number of deaths related in a given time period divided by the total number of the population.

For example, let's suppose a population of 1,000 individuals (equivalent to 500 pairs) were to have 1,000 offspring in 1 year. Then, 500 deaths would result in the following growth rate:

$$r = \frac{\frac{\Delta N}{\Delta t}}{N} = \frac{\frac{1000 - 500}{1J}}{1000} = 0{,}5\frac{1}{J}$$

6.11.1 Exponential Growth

For $\varDelta t \to 0$, you obtain the following basic equation of population dynamics by rearranging the following equation:

$$\frac{\mathrm{d}N}{\mathrm{d}t} = rN$$

Resulting in the following solution:

$$N = N_0 e^{rt}$$

This equation describes exponential growth for a positive growth rate, which can realistically occur only exceptionally and temporarily in nature.

6.11.2 Logistic Growth

Any real growth is constrained by a capacity limit K. The following then applies to the growth rate:

$$\frac{\frac{\mathrm{d}N}{\mathrm{d}t}}{N} = r\frac{K - N}{K}$$

By rearranging, you get the differential equation for what's called logistic growth:

$$\frac{\mathrm{d}N}{\mathrm{d}t} = r\frac{K - N}{K}N = rN - \frac{r}{K}N^2$$

For time $t = 0$ and when the number N of a population has reached the capacity limit K, the slope of the searched function $N(t)$ takes the value zero. Thus, an S-shaped course for $N(t)$ between the bounds $N = 0$ and $N = K$ is to be assumed.

In task 10 (see Section 6.13), we'll simulate both exponential growth and logistic growth.

6.11.3 Predator-Prey Relationship for Exponential Growth

If N_2 prey animals live in a territory where there are also N_2 predators, their growth rate will be limited not only by the capacity limit of the available resources, but also by the existence of the predators. In other words, the growth rate of the prey species is limited by the predators' hunting success. The situation in which predator and prey meet and the predator is successful in its hunt is to be modeled by the prey probability b of hunting success:

$$\frac{\frac{\mathrm{d}N_1}{\mathrm{d}t}}{N_1} = r - bN_2$$

By rearranging, you obtain the following differential equation for the prey animals:

$$\frac{\mathrm{d}N_1}{\mathrm{d}t} = rN_1 - bN_2N_1$$

For the predator, a mortality rate s is assumed instead of the growth rate because, without the existence of the prey animals, it would become extinct. The population of predators can only grow if prey animals also exist and if these could be hunted successfully. The probability of hunting success is again to be described by the prey probability b. For the predator growth rate, the following applies:

$$\frac{\frac{\mathrm{d}N_2}{\mathrm{d}t}}{N_2} = -s + bN_1$$

By rearranging, you obtain the following differential equation for the predators:

$$\frac{\mathrm{d}N_2}{\mathrm{d}t} = -sN_2 + bN_1N_2$$

This differential equation system is also referred to as a *Lotka-Volterra system*. In the SciPy documentation, a source code example demonstrates how you can solve this differential equation system using the SciPy function `solve_ivp()` at *https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html*.

However, you can also simulate the predator-prey relationship for exponential growth using Listing 6.33 by setting the constants c and d equal to 0 in line 12.

6.11.4 Predator-Prey Relationship for Logistic Growth

If the number of herbivores and predators increases, the competitive situation also worsens for both genera: The dwindling food supply reduces both growth rates. Different capacity limits are assumed for the two genera. For prey, the growth rate is reduced by $c = \frac{r}{K_1}$; and for predators, by $d = \frac{s}{K_2}$. These considerations specify the predator-prey model for logistic growth:

$$\frac{\mathrm{d}N_1}{\mathrm{d}t} = rN_1 - bN_1N_2 - cN_1^2$$
$$\frac{\mathrm{d}N_2}{\mathrm{d}t} = -sN_1 + bN_1N_2 - dN_2^2$$

Listing 6.33 simulates the predator-prey relationship for the assumption that the growth of both populations is limited by capacity constraints.

```
01  #33_predator_prey.py
02  import numpy as np
03  import matplotlib.pyplot as plt
04  from scipy.integrate import solve_ivp
```

```
K1=10e3 #prey capacity limit
K2=1e3 #predator capacity limit
prey=500
predator=50
r=0.25   #prey reproduction rate
b=0.001  #prey probability
s=0.5    #predator mortality rate
c,d = r/K1, s/K2
tmax=50  #period
xy0=[prey,predator]
#system of differential equations
def dgl(t,xy,r,b,s,c,d):
    N1,N2 = xy   #initial values
    dN1_dt= r*N1-b*N1*N2-c*N1**2 #prey
    dN2_dt=-s*N2+b*N1*N2-d*N2**2 #predator
    return [dN1_dt,dN2_dt]
#Solution of the differential equation system
t = np.linspace(0,tmax,500)
z=solve_ivp(dgl,[0,tmax],xy0,args=(r,b,s,c,d),dense_output=True)
N1, N2 = z.sol(t) #separation of the solutions
mbmean=int(np.mean(N1)) #prey, mean value
mrmean=int(np.mean(N2)) #predator, mean value
fig,ax=plt.subplots(2,1,figsize=(6,6))
#time chart
ax[0].plot(t, N1,"g--",lw=2,label="Prey")
ax[0].plot(t, N2,"r-",lw=2,label="Predator")
ax[0].plot([0,tmax],[mbmean,mbmean],"g-.",lw=1)
ax[0].plot([0,tmax],[mrmean,mrmean],"r-.",lw=1)
ax[0].set(xlabel="Time",ylabel="$N_{1}, N_{2}$")
ax[0].legend(loc="best")
#phase diagram
ax[1].plot(N1, N2,'b-',lw=1)
ax[1].set(xlabel="Prey",ylabel="Predator",title="Phase diagram")
fig.tight_layout()
plt.show()
```

Listing 6.33 Simulation of the Predator-Prey Model with Logistic Growth

Output

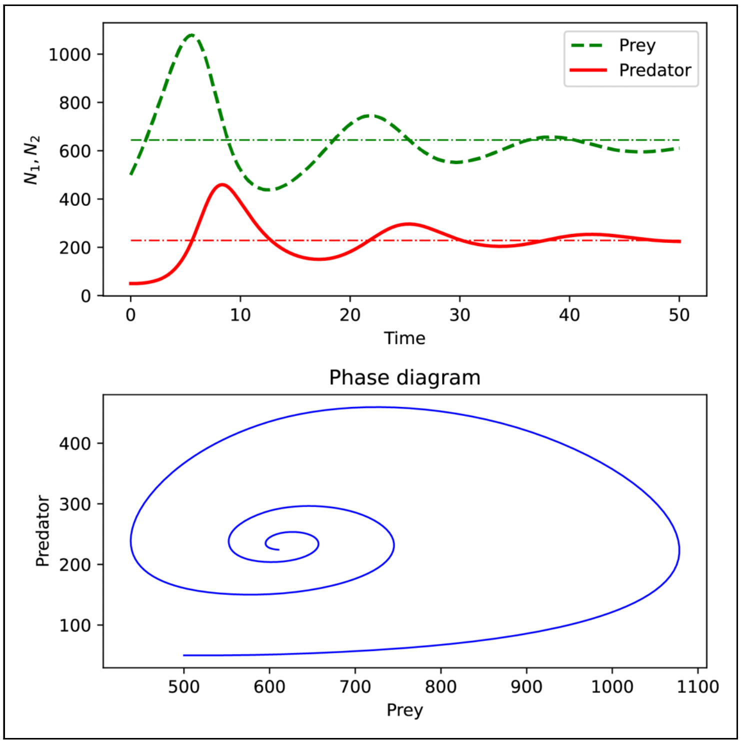


Figure 6.33 Simulation of the Predator-Prey Model with Logistic Growth

Analysis

The predator-prey relationship is described in the phase diagram by a spiral running from the outside to the inside, as shown in Figure 6.33. The number `N1` of prey stabilizes at a mean of 644 individuals, and the number `N2` of predators stabilizes at a mean of 229 individuals. These values are illustrated by the predator-prey ratio endpoint in the phase diagram.

In lines 16 to 20, the differential equation system `dgl` for logistic growth is implemented according to the specifications and solved in line 23. The separation of the solution is performed in line 24: `N1,N2= z.sol(t)`. In lines 29 and 30, the `plot` method prepares the visualization for the relation $N_1, N_2 = f(t)$.

In the first graph, the mean value of both populations is visualized by a dash-dot line.

6.12 Project Task: Simulation of an Epidemic

The spread of an epidemic can be described by means of a differential equation system that consists of three differential equations. In this case, a population N is divided into three groups: the healthy S (*Susceptible*), the infected I (*Infective*), and the recovered R (*Removed*).

The following conditions should apply when we create the model:

- Deaths and births are to be disregarded. As a result, at all times, the following applies: $N = S(t) + I(t) + R(t)$.
- Infected individuals are immediately contagious.
- Healthy individuals are infected at an infection rate of $b > 0$.
- Infected individuals recover at a recovery rate of $g > 0$.

The differential equation system of the *Susceptible-Infected-Removed model* (SIR) of Kermack and McKendrick (1927) describes the spread of infectious diseases in this way:

$$\frac{\mathrm{d}S}{\mathrm{d}t} = -b\frac{S \cdot I}{N}$$

$$\frac{\mathrm{d}I}{\mathrm{d}t} = b\frac{S \cdot I}{N} - g \cdot I$$

$$\frac{\mathrm{d}R}{\mathrm{d}t} = g \cdot I$$

Listing 6.34 simulates the infection process for a population of $N = 1{,}000$ individuals over a period of 120 days. For the recovery rate and infection rate, the assumptions are $g = 0.04$ and $b = 0.4$.

```python
#34_epidemic.py
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
tmax=120
S0=997  #not immune healthy individuals
I0=3    #infected
R0=0    #recovered
N=S0+I0+R0 #Population
b=0.4  #infection rate
g=0.04 #recovery rate
#system of differential equations
def dgl(t,ya):
    S,I,R=ya
    dS_dt=-b*S*I/N    #not immune healthy individuals
    dI_dt=b*S*I/N-g*I #infected
    dR_dt=g*I         #recovered
    return [dS_dt,dI_dt,dR_dt]
#Initial values
y0 = [S0,I0,R0]
t = np.linspace(0, tmax, 500)
z=solve_ivp(dgl,[0,tmax],y0,dense_output=True)
```

```
23  S, I, R = z.sol(t)
24  fig, ax = plt.subplots()
25  ax.plot(t, S,'b-',label="Healthy")
26  ax.plot(t, I,'r--',label="Infected")
27  ax.plot(t, R,'g-.',label="Recovered")
28  ax.legend(loc='best')
29  ax.set(xlabel="Time",ylabel="Individuals")
30  ax.grid(True)
31  plt.show()
```

Listing 6.34 Simulation of an Epidemic

Output

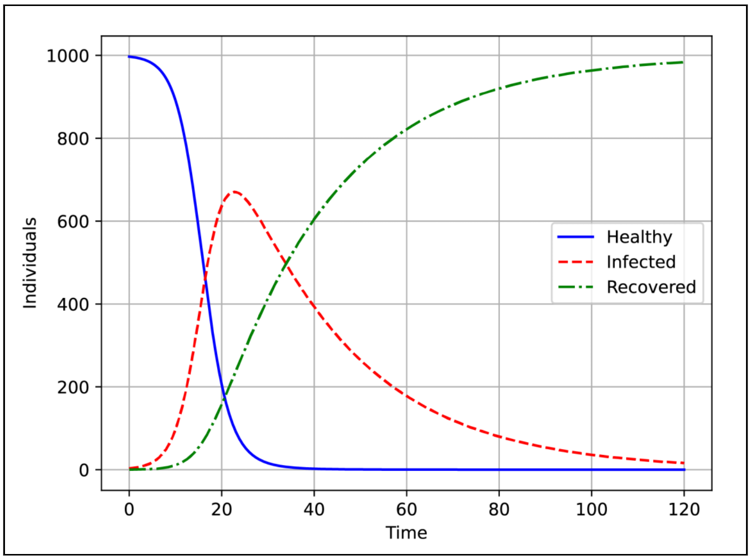


Figure 6.34 Simulation of an Epidemic

Analysis

In lines 06 to 08, the initial values for the healthy `S0`, infected `I0`, and recovered `R0` are defined. Line 09 uses these values to compute the total number `N` of the population in which the epidemic is spreading. By changing the infection rate `b` in line 10 and the recovery rate `g` in line 11, you can simulate different infection profiles.

The code of the differential equation system consists of three lines (lines 15 to 17). In line 23, the entire solution set of the differential equation system is assigned to the `S, I,R` tuple.

The output shown in Figure 6.34 clearly and plausibly shows that, during the course of infection, the number of uninfected `S` decreases and the number of infected `I` increases to a maximum. The number of recovered `R` increases in the shape of an S curve until it reaches the threshold value of `N`.

6.13 Tasks

1. A triangular current with $i_{max} = 1$ A runs through a coil. The period duration is 20 ms. Write a program that visualizes the current profile and calculates and graphs the voltage profile.
2. Write a program that calculates the length of a logarithmic spiral defined by the following:

   $r = ae^{b\varphi}$

   Use the SciPy function `quad()` in the limits between 0 to 2π.
3. Write a program that calculates the area of a circle using the SciPy function `quad()`. The program should also calculate the error that arises during the numerical integration.
4. Write a program that calculates the circumference of a circle using the SciPy function `quad()`. The program should also calculate the error that arises during the numerical integration.
5. Write a program that calculates the surface area and volume of a sphere using the SciPy function `quad()`. The program should also calculate the error that arises during the numerical integration.
6. In a rectangular conductor with $a = 4$mm and $b = 2$mm, a current flows with the current density of $J(x,y) = x^2 y \frac{\mathrm{A}}{\mathrm{mm}^5}$. Write a program that computes the current using the `dblquad()` function.
7. A cuboid with the dimensions of $x = 0.3$m, $y = 0.4$m, $z = 0.5$m contains the volume charge density defined by the following:

   $$\rho = (xy + xz + yz)\frac{\mathrm{As}}{\mathrm{m}^3}$$

   Write a program that computes the amount of charge Q contained in the cuboid using the `tplquad()` function.
8. Write a program that solves the following differential equations in the interval [-1,1] numerically:

   $y' = -\frac{x}{y}$

   $y' = 1 + x - y$

   $y' = \tan x - \sin x$

   $y' = e^{-xy}$

   The results should be presented as a function plot. The following applies to the initial values: y(0) = 1.

9. The pendulum motion of a rod with length l is described by the following differential equation:

   $$\ddot{\varphi} + d\dot{\varphi} + \frac{3g}{2l}\sin\varphi = 0$$

   The damping d has the value 0.5. Write a program that computes the deflection φ, the angular velocity $\dot{\varphi}$, and the trajectory. The solutions of the differential equation should be represented in each case in a subdiagram.
10. Write a program that simulates exponential growth and logistic growth. The differential equations should be solved using the `solve_ivp()` function.
11. A spring pendulum oscillating in the direction of the y-axis is described by the differential equation $m \cdot \ddot{y} + d \cdot \dot{y} + c \cdot y = 0$. The mass is m = 0.5kg. The damping has a value of d = 0.5. The value of the spring constant is c = 10 N/m. Solve the initial value problem using `solve_ivp()` for y_0 = 0.1m and v_0 = 0. The deflection $y(t)$, the velocity $v(t)$, and the phase diagram $v=f(y)$ should each be shown in a subdiagram.
12. Simulate Foucault's pendulum. The pendulum has a length of l = 67m. The steel sphere has a diameter of d = 1.896dm. The pendulum is deflected by 2m from the rest position in the direction of the x-axis. In addition, the trajectory and the phase diagram $v = f(x)$ should be shown in two additional subdiagrams. Supplement the code provided in Listing 6.21 accordingly.
13. An oscillatory system consisting of two string pendulums whose masses are coupled with a helical spring is described by the following system of differential equations:

   $$\dot{\varphi}_1 = \omega_1$$
   $$\dot{\omega}_1 = -\frac{g}{l}\varphi_1 - \frac{c}{m}(\varphi_1 - \varphi_2)$$
   $$\dot{\varphi}_2 = \omega_2$$
   $$\dot{\omega}_2 = -\frac{g}{l}\varphi_2 + \frac{c}{m}(\varphi_1 - \varphi_2)$$

   The spring constant has a value of c = 1N/m. The two masses each have a value of m = 0.2kg. The threads both have a length of l = 0.2m. The differential equation system is to be solved using the `solve_ivp()` function. The deflections of angles φ_1 and φ_2 should each be plotted in a subplot. Write an appropriate program.
14. Write a program that will solve the following differential equation system in the interval of [0,10]:

   $$y_1'(x) = ay_1 - by_1y_2$$
   $$y_2'(x) = cy_1 - dy_1y_2x^2$$

   Use the `solve_ivp()` function for $y_1(0) = y_2(0) = 1$. For the coefficients: a = b = c = 1 and d = 0.1. The result should be displayed as a function plot.

15. Write a program that will solve the following differential equation system in the interval of [0,1]:

    $$y_1'(x) = y_2 + 2y_3$$
    $$y_2'(x) = -xy_1 + y_2$$
    $$y_3'(x) = y_1 + y_2$$

    Use the `solve_ivp()` function for $y_1(0)=1$ and $y_2(0)=0$. The result should be presented in three subdiagrams.
16. Write a program that solves the following fourth-order differential equation in the interval of [0,5]:

    $$3y'''' + 2y'' + y' + 4y = \cos x$$

    Use the `solve_ivp()` function. All initial values are zero.
17. Extend the SIR model to a SIRD model in which deaths D are also to be considered:

    $$\frac{\mathrm{d}S}{\mathrm{d}t} = -b\frac{S \cdot I}{N}$$
    $$\frac{\mathrm{d}I}{\mathrm{d}t} = b\frac{S \cdot I}{N} - g \cdot I - m \cdot I$$
    $$\frac{\mathrm{d}R}{\mathrm{d}t} = g \cdot I$$
    $$\frac{\mathrm{d}D}{\mathrm{d}t} = m \cdot I$$

    The following assumptions apply: N = 1,000, t = 120 days, recovery rate g = 0.035, infection rate b = 0.4, and mortality rate m = 0.005.
18. Write a program that computes the harmonic distortion and frequency spectrum of a sinusoidal AC voltage rectified by a one-way rectifier.
19. Write a program that computes the Butterworth coefficients for a fifth-degree low-pass for a cutoff frequency of 1Hz using the `butter()` function.
20. Simulate the frequency response of a third-degree Butterworth bandstop using the `butter()` function for the cutoff frequencies 50 Hz and 500 Hz.

Chapter 7
3D Graphics and Animations Using VPython

In this chapter, you'll learn how to represent and animate objects in 3D space using the VPython module. Sophisticated animations of physical processes will showcase the capabilities of VPython.

The letter "V" in the name of the VPython module stands for *visual*, referring to the representation and movement of objects in 3D space. The dynamics of physical relationships and processes are no longer depicted as function plots, as done with the SciPy module, but rather as how an observer perceives them in reality. It is important to consider that many physical processes, such as an oblique throw, may not be captured in detail by the human eye. VPython provides the ability to slow down (slow motion) or speed up processes using the `rate(frequency)` function as needed. When real movements are simulated on a computer, the field of computer graphics refers to this process as animation, which means giving "life" to objects.

The canvas where the objects are represented is referred to as a *scene*. Within a scene, you can rotate the displayed object around the *x*-*y*-*z* axis by holding down the right mouse button and moving the mouse cursor. This task allows the observer to view an object from different perspectives.

Starting from version 7, the VPython module is imported using the `from vpython import *` statement. Other modules do not need to be imported. After starting the program, the default browser opens, and the program runs within *Web Graphics Library (WebGL)*. WebGL is a JavaScript programming interface that allows hardware-accelerated 3D graphics to be displayed in a web browser without additional extensions.

The position of an object in 3D space can be determined using the `vector(x, y, z)` method and can be changed if the object is supposed to move. Since the concept of vectors plays a central role in VPython, let's briefly discuss this topic using a small console example:

```
>>> from vpython import *
>>> v1=vector(1,2,3)
>>> v2=vector(4,5,6)
>>> v1+v2
```

```
<5, 7, 9>
>>> type(v1)
<class 'vpython.cyvector.vector'>
```

In the second and third lines, the `vector(x,y,z)` method creates the `v1` and `v2` objects as three-dimensional vectors with the data for the x-y-z coordinates. In the fourth line, the addition of these vectors takes place. Besides the addition, the scalar product and the cross product are also implemented. Instead of `vector()`, the abbreviation `vec()` is also allowed.

A body object `obj` is created using the following statement:

```
obj=body(pos=vec,size=vec,axis=vec,color=color.color,...)
```

In this context, the `body` method represents basic shapes such as `box()`, `sphere()`, `cylinder()`, `cone()`, and so on, which are provided by the VPython module. If you need other bodies, you can create them from the basic bodies (`k1`, `k2`, etc.) using the `compound ([k1,k2, ...])` method.

7.1 The Coordinate System

Figure 7.1 shows the coordinate system of VPython created using Listing 7.1. The coordinate origin is located in the center of the *canvas*. The x-axis points from left to right, the y-axis from bottom to top, and the z-axis is perpendicular to the canvas.

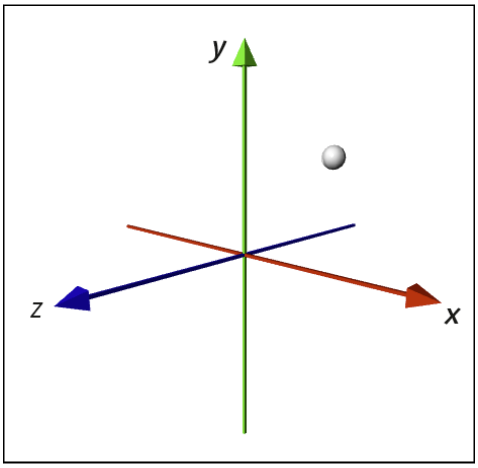


Figure 7.1 Coordinate System in VPython

Start the program, which will run in your default browser after a short time delay. Then, right-click on the canvas, hold the right mouse button down, and rotate the coordinate system so that you can observe as many perspectives as possible. The gray point with the coordinates (5,5,0) inserted in the coordinate system facilitates your orientation in space and to clarify the mode of action of the coordinate transformation in VPython.

```
01  #01_coordinates.py
02  from vpython import *
03  h=10. #height
04  b=10. #width
05  t=10. #depth
06  scene.title="<h2>Coordinate system of VPython</h2>"
07  scene.width=scene.height=600
08  scene.background=color.white
09  scene.center=vector(0,0,0)
10  scene.range=1.5*b
11  x0=vector(-b,0,0)
12  y0=vector(0,-h,0)
13  z0=vector(0,0,-t)
14  #x-axis is red
15  arrow(pos=x0,axis=vector(2*b,0,0),shaftwidth=0.15,color=color.red)
16  #y-axis is green
17  arrow(pos=y0,axis=vector(0,2*h,0),shaftwidth=0.15,color=color.green)
18  #z-axis is blue
19  arrow(pos=z0,axis=vector(0,0,2*t),shaftwidth=0.15,color=color.blue)
20  label( pos=vec(b,-1,0),text="x",height=30,box=False,opacity=0)
21  label( pos=vec(-1,h,0),text="y",height=30,box=False,opacity=0)
22  label( pos=vec(-1,0,t),text="z",height=30,box=False,opacity=0)
23  points(pos=vector(5,5,0))
24  scene.caption="\nPress right mouse button and drag"
```

Listing 7.1 The Coordinate System of VPython

Analysis

In line 02, the `vpython` module is imported with all available methods and properties.

Lines 03 to 05 define the height, width, and depth of the display window (canvas).

The `scene.title="..."` command in line 06 outputs the headline of the program with the HTML tag `<h2> ... </h2>` in the browser.

In line 07, the `scene.width=scene.height=600` command sets the width and height of the display window to 600 pixels each.

In line 08, `scene.background=color.white` sets the background of the canvas to the color white. The default setting is the color black.

The `scene.center=vector(0,0,0)` statement in line 09, which sets the origin of the coordinate system, is actually not necessary because the default value places the origin exactly in the center of the canvas. We only included it in this example so we could carry out further program tests.

Due to `scene.range=1.5*b` (line 10), the display range is enlarged so that enough space on the canvas exists when you rotate the coordinate system.

In lines 11 to 13, the VPython method `vector()` is used to move the coordinate axes. A negative sign causes a shift in positive direction of the coordinate axis to the right (x-axis) or upwards (y-axis) or forwards (z-axis). All three axes have a length of 20 units.

The `arrow()` method draws the coordinate axes as arrow objects in lines 15, 17, and 19.

The `label()` method labels the coordinate axes (lines 20 to 22).

The point created using the `points(pos=vector(5,5,0))` method should confirm the correctness of the coordinate transformations: The point lies exactly on half of the intercept of the x- and y-axis.

Using `scene.caption="..."`, you can output any text in the browser (line 24).

Exercise

The point should be represented in the second, third, and fourth quadrants. Change the source code in each case and restart the program. Check the positions of the point by rotating the coordinate system within the scene.

7.2 Basic Shapes, Points, and Lines

The following basic shapes can be created using VPython: cuboid, sphere, cylinder, cone, pyramid, ellipsoid, and circular ring. The general syntax for creating a body object is:

```
obj=body(pos=vec(x0,y0,z0),axis=vec(x,y,z),size=vec(a,b,c),
color=color.red)
```

The first parameter specifies the position of the body in 3D space. The default setting for the position is `pos=vector(0,0,0)`. The position can be changed within the animation loop using `obj.pos=vector(x,y,z)`. If only the position in x-direction needs to be changed, the `obj.pos.x= value` statement is sufficient.

The `axis` vector defines the orientation of the body object. For example, if `vector (1,0,0)` is assigned to `axis`, the body will be aligned in the direction of the x-axis. The same applies to the y- and z-axes.

The `size` vector determines the dimensions of the body object. For example, the `box(size=vector(10,5,2)` method creates a cuboid object with a width of 10 units of length (LE), a height of 5 LE, and a depth of 2 LE.

The color property can be varied via the obj.color=vector(R,G,B) vector. The colors *red*, *green*, and *blue* can have values between 0 and 1. The default color setting is *gray*.

7.2.1 Cylinder

A cylinder object has the following properties: position, length, orientation, radius, and color. The pos=vector(x0,y0,z0) vector defines the position (center of the base of the cylinder) in space. The axis=vector(x,y,z) vector determines the length and orientation of a cylinder object. Using the following method, a cylinder object of the cylinder() class can be created from the VPython module:

```
cylinder(pos=vector(x0,y0,z0),axis=vector(x,y,z),radius=r,...)
```

The default color (the default value) is gray. The radius has the default value radius=1.

For example, if x0=-20, x=40, and all other values of the position and axis vector are zero, then a cylinder object with the length of 40 units of length (LE) is created, shifted left by 20 LE on the x-axis. The centerline of the cylinder lies exactly on the x-axis.

Alternatively, the dimensions of a cylinder object can be specified using the size=vector(length,height,width) property.

Listing 7.2 shows how to create a cylinder object in VPython.

```
#02_cylinder.py
from vpython import *
scene.title="<h2>Cylinder</h2>"
scene.autoscale=True
scene.background=color.white
scene.width=600
scene.height=600
scene.center=vector(0,0,0)
scene.range=30
#Position: x0,y0,z0
p=vector(-20,0,0)
#alignment and length
a=vector(40,0,0)
r=10.  #radius
#col=color.gray(0.5)
#red, green, blue
col=vector(1,0,0)
cylinder(pos=p,axis=a,radius=r,color=col,opacity=0.5)
#length, height, width
#cylinder(pos=p,size=vector(40,20,20),color=col)
scene.caption="\nPress the right mouse button and rotate the object"
```

Listing 7.2 Cylinder

Output

Figure 7.2 shows the cylinder created by Listing 7.2 in the canvas.

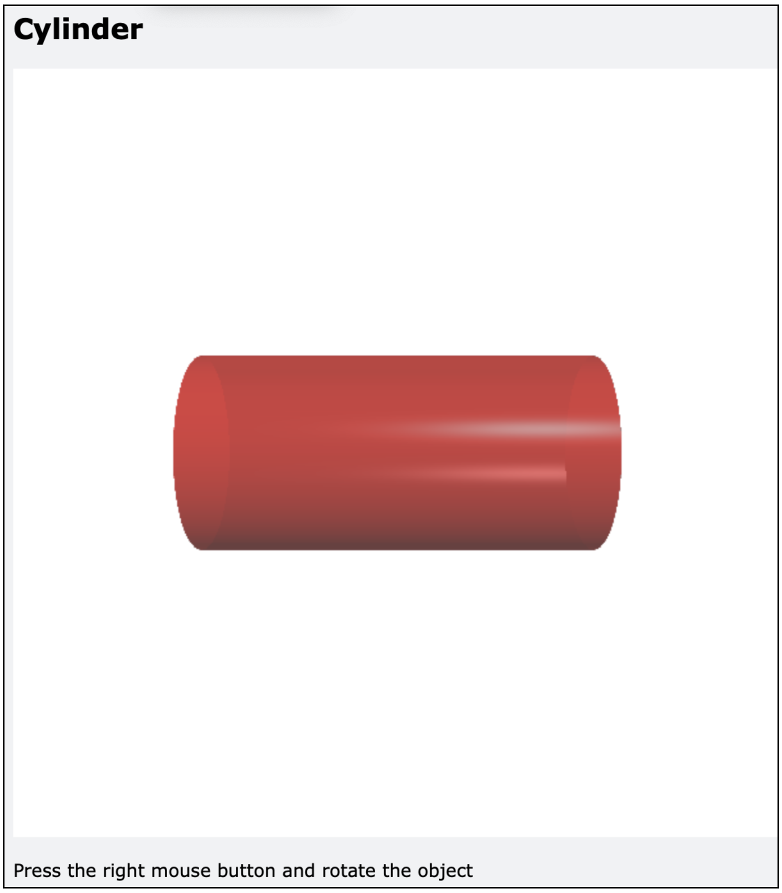


Figure 7.2 Cylinder

Analysis

The `scene.range=30` property (line 09) changes the width of the display area. A value smaller than 30 (line 09) enlarges the cylinder object, and a value larger than 30 reduces the cylinder object.

The `p=vector(-20,0,0)` vector in line 11 specifies that the cylinder is moved 20 LE to the left on the x-axis. The `a=vector(40,0,0)` vector (line 13) defines the length of the cylinder of 40 LE. The centerline of the cylinder lies on the x-axis because the y- and z-components of the alignment vector `axis` have the value zero.

The color of a body is determined as an RGB value in line 17 by the `col=vector(1,0,0)` vector. The color saturation can be set with values from 0 to 1. The `opacity` property determines the `opacity` of a color. A value between 0 and 1 is permissible. If `opacity` has the value 0, the body is completely transparent.

In line 18, the following method generates a cylinder object:

```
cylinder(pos=p,axis=a,radius=r,color=col,opacity=0.5)
```

This cylinder object has its properties stored in the objects `p`, `a`, `r`, and `col` (lines 11, 13, 14, and 17).

Exercise

Change the property of the `scene.range` canvas in line 09 to 40, 50, and 60. What exactly is happening here?

Change the colors, the positions, and the dimensions of the cylinder. Restart the program after each individual change. Check if the changes have the expected effects.

Comment out line 18 and remove the comment in line 20. Restart the program and observe the result by rotating the cylinder within the scene. Vary the values of the `size()` vector and restart the program after each change.

7.2.2 Cuboid

A cuboid object can be created using the following method:

```
box(pos=vec(x0,y0,z0),axis=vec(x,y,z),size=vec(L,H,B), ...)
```

The `pos` vector defines the position of the cuboid. In contrast to the cylinder object, the position in this case does not refer to one end of the object, but to the center of the cuboid. The `axis` vector defines the orientation, while the `size` vector defines the dimensions (length, height, width) of the cuboid object.

Listing 7.3 creates a cuboid with length 40, height 20, and width 10.

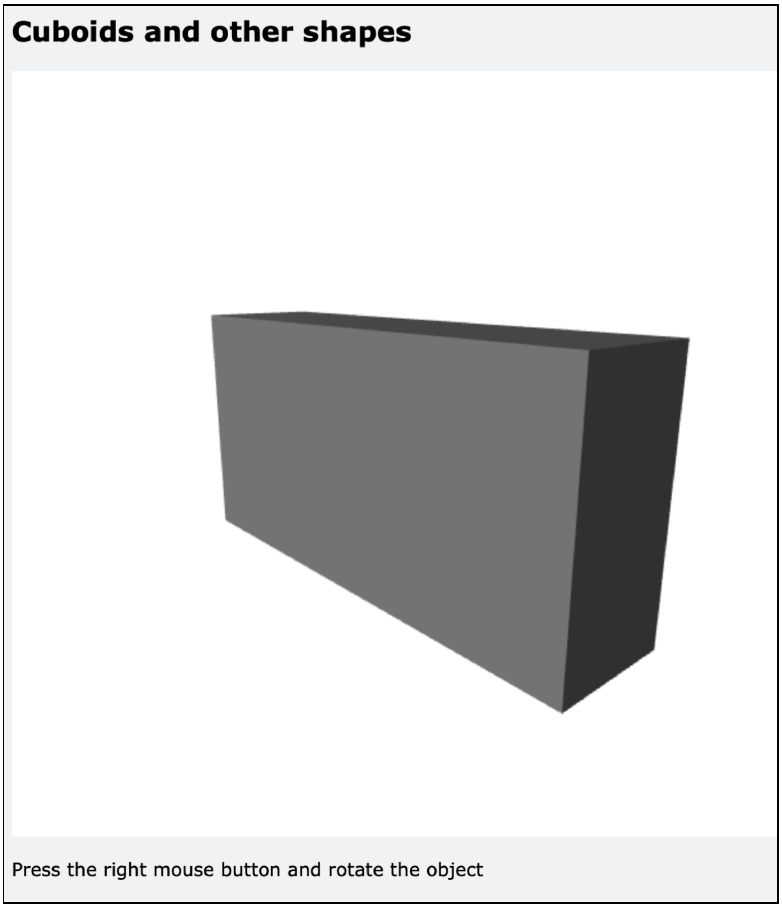
```
01  #03_cuboid.py
02  from vpython import *
03  scene.title="<h2>Cuboids and other shapes</h2>"
04  scene.autoscale=True
05  scene.background=color.white
06  scene.width=600
07  scene.height=600
08  scene.center=vector(0,0,0)
09  #x0,y0,z0
10  p=vector(0,0,0)
11  #alignment
12  a=vector(1,0,0)
13  #dimensions: length, height, width
14  dim=vector(40,20,10)
15  scene.range=30
```

```
#rotation
d=vector(0,0,0)
c=color.gray(0.5)
box(pos=p,axis=a,size=dim,up=d,color=c)
#cone(pos=vec(-5,0,0),axis=vector(10,0,0),radius=5,color=c)
#ellipsoid(pos=vec(0,0,0),axis=vec(1,0,0),size=vec(10,5,5),color=c)
#pyramid(pos=vec(0,5,0),axis=vec(0,1,0),size=vec(10,12,12),color=c)
#ring(pos=vec(0,0,0),axis=vec(0,0,1),radius=10,thickness=3,color=c)
scene.caption="\nPress the right mouse button and rotate the object"
```

Listing 7.3 Cuboids and Other Shapes

Output

The cuboid that is output to the canvas is shown in Figure 7.3.


Cuboids and other shapes

Press the right mouse button and rotate the object

Figure 7.3 Cuboid

Analysis

In line 10, the `p=vector(0,0,0)` vector specifies that the cuboid is positioned exactly in the center of the drawing area. The `a=vector(1,0,0)` vector (line 12) determines the orientation of the cuboid: Its centerline lies on the x-axis. The `dim=vector(40,20,10)` vector in line 14 provides the dimensions of the box: It has a length of 40 LE, a height of 20 LE, and a width of 10 LE. Using the `d=vector(0,0,0)` vector in line 17, you can rotate the object around its own axis.

In line 19, the `box()` method creates the cuboid object with the given properties.

Exercise

Change the alignment vector in line 12 `a=vector(0,1,1)` and restart the program. What exactly is happening here?

Change the `d=vector(0,0,1)` vector in line 17 for the property `up` in line 19 and restart the program. What exactly is happening now?

Test the program with the basic shapes that have been commented out.

7.2.3 Points

Point objects can be created using the following method:

```
points(pos=[vector(-1,0,0), vector(1,0,0)], radius=0, ...)
```

The `pos` property expects a list of vectors containing the positions of the point objects. Specifying the `radius` property is not necessary. According to the VPython documentation, the radius should have a default value of 2.5 pixels, even if assigned the value zero.

Listing 7.4 represents the center and the vertices of a cube as point objects, resulting in the graphical output shown in Figure 7.4.

```
#04_points.py
from vpython import *
scene.width=600
scene.height=600
scene.background=color.white
e=1.
scene.center=vector(e/2,e/2,e/2)
scene.range=1.2*e
v=[(0,0,0),(0,0,e),(0,e,0),(0,e,e),
    (e,0,0),(e,0,e),(e,e,0),(e,e,e),(e/2,e/2,e/2)]
box(pos=vector(e/2,e/2,e/2),size=vector(e,e,e),
    axis=vector(1,0,0),opacity=0.5)
points(pos=v,color=color.red)
```

Listing 7.4 Vertices of a Cube

Output

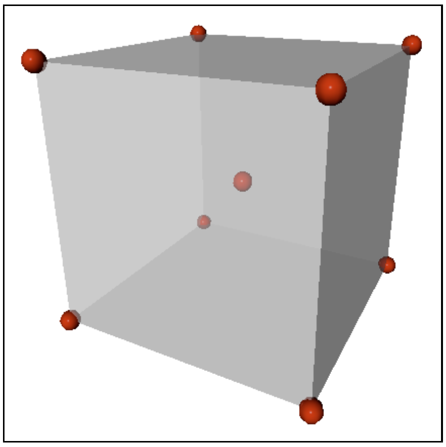


Figure 7.4 Vertices of a Cube

Analysis

Line 06 specifies the edge length `e` of the cube. In line 07, the center of the coordinate system is scaled to half the edge length. Line 09 contains a list named `v` containing the positions for the center and the vertices of the cube. In line 11, the `box()` method creates a transparent cube with the edge length `e`. In line 13, the `points()` method creates nine red point objects with the default radius.

7.2.4 Lines

The following method draws a line between two points:

```
curve(vector(-1,0,0), vector(1,0,0), ...)
```

For example, let's suppose the two vectors `v1` and `v2` are given. Then, you can draw a connecting line between the two vectors using five different syntax variants:

```
>>> curve(v1,v2)       #1
>>> curve([v1,v2])     #2
>>> curve(pos=[v1,v2]) #3
>>> c = curve(v1)      #4
>>> c.append(v2)
>>> c=curve()          #5
>>> c.append(v1,v2)
```

Listing 7.5 shows how the `curve()` method connects the vertices of a tetrahedron with lines of the length e . The base of the tetrahedron is placed in the x-y plane. From the radius *r* of the circumcircle of the base area, the edge length can be computed in the following way:

$$e = \sqrt{3}r$$

The height of the tetrahedron

$$z = \sqrt{6}\frac{e}{3}$$

points in the direction of the positive z-axis.

```
01  #05_lines.py
02  from vpython import *
03  scene.width=600
04  scene.height=600
05  scene.background=color.white
06  r=10. #radius of the plane
07  e=sqrt(3.)*r #edge length
08  scene.center=vector(0,0,0)
09  scene.range=1.8*r
10  x=r*cos(pi/6.)
11  y=r*sin(pi/6.)
12  z=sqrt(6.)*e/3. #height
13  #triangle: bottom left-top, bottom right-bottom left
14  v1=[(-x,-y,0),(0,r,0),(x,-y,0),(-x,-y,0)]
15  #star: bottom left-center, top-center, bottom right-center
16  v2=[(-x,-y,0),(0,0,z),(0,r,0),(0,0,z),(x,-y,0)]
17  points(pos=v2,radius=10.,color=color.red)
18  c=curve(pos=v1,color=color.green)
19  c.append(v2,color=color.yellow)
```

Listing 7.5 Vertices of a Tetrahedron Connected by Lines

Output

Figure 7.5 shows the result of Listing 7.5 on the canvas.

Analysis

Line 06 sets the radius `r` of the x-y plane circumcircle (circumscribed circle) to 10 units of length. Line 07 calculates the edge length `e`. In lines 10 and 11, the x and y coordinates of the x-y plane are calculated. The statement in line 12 computes the height `z` of the tetrahedron in the direction of the z-axis. The `v1` list in line 14 contains the coordinate data of the triangular base area of the x-y plane. The `v2` list in line 16 contains the x-y-z coordinates of the tetrahedron corners. In line 17, the `points()` method draws the four vertices of the tetrahedron. The `curve()` method draws the line objects of the `v1` list as a triangle in line 18. In line 19, the star-shaped lines of the `v2` list are added to the polyline `c`.

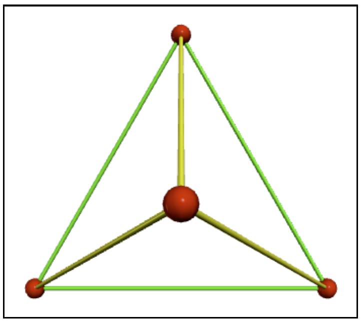


Figure 7.5 Vertices of a Tetrahedron

Exercise

Comment out lines 17 and 19, and restart the program. What do you see now?

Only the polyline of `v2` should be displayed. Change the program accordingly and restart the program.

7.2.5 Sphere

The `sphere(pos=vector(x0,y0,z0),radius=r,...)` method creates a sphere object. The `pos` vector determines the coordinates of the center of a sphere in space, and the `radius` property determines its radius.

Listing 7.6 shows how the `sphere()` method can be used to arrange nine spheres symmetrically in space.

```
#06_spheres1.py
from vpython import *
scene.width=600
scene.hight=600
scene.background=color.white
x0=5.
y0=5.
z0=5.
R1=1.
R2=0.25
#center
sphere(pos=vector(0,0,0),radius=R1,color=color.red)
#top
sphere(pos=vector(0,y0,0),radius=R2,color=color.blue)
#bottom
sphere(pos=vector(0,-y0,0),radius=R2,color=color.blue)
```

```
17  #left
18  sphere(pos=vector(-x0,0,0),radius=R2,color=color.blue)
19  #right
20  sphere(pos=vector(x0,0,0),radius=R2,color=color.blue)
21  #back
22  sphere(pos=vector(0,0,-z0),radius=R2,color=color.blue)
23  #front
24  sphere(pos=vector(0,0,z0),radius=R2,color=color.blue)
25  label( pos=vec(0,0,0), text="O",height=30,box=False,opacity=0)
26  sphere(pos=vector(x0/2,0,0),radius=R2,color=color.blue)
27  sphere(pos=vector(-x0/2,0,0),radius=R2,color=color.blue)
```

Listing 7.6 Nine Spheres in Space

Output

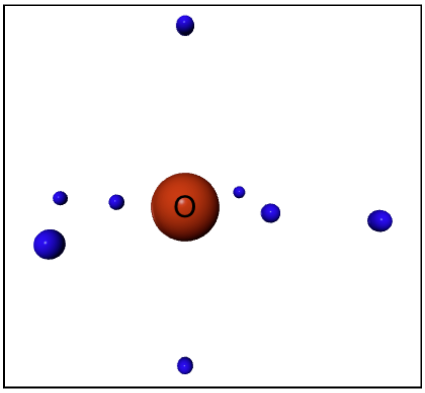


Figure 7.6 Spheres in Space

Analysis

Lines 06 to 08 define the coordinates of the small blue spheres in space. The center red sphere has a radius of `R1=1` (line 09), and the six other spheres have a radius of `R2=0.25` each. In lines 12 to 24, the `sphere()` method creates the individual sphere objects. The comments describe the positions of the individual spheres. In line 25, a `label` object named `O` is created. This label suggests that a Bohr model of an oxygen atom (without the inner shell) is represented by this object, as shown in Figure 7.6.

Exercise

Start the program and view the scene from different perspectives.

Add the statements for the two electrons of the inner shell of the oxygen atom to the source code and start the program.

Crystal Lattice

The example shown in Listing 7.7 illustrates how you can place spheres within space using three nested `for` loops and the `sphere()` method.

```
01  #07_spheres2.py
02  from vpython import *
03  scene.background=color.white
04  scene.width=600
05  scene.height=600
06  e = 5   #lattice spacing
07  R = 0.5 #radius of an atomic nucleus
08  for x in range(-e,e):
09      for y in range(-e,e):
10          for z in range(-e,e):
11              sphere(pos=vector(x,y,z),radius=R,color=color.red)
```

Listing 7.7 Spheres as a Lattice Structure

Output

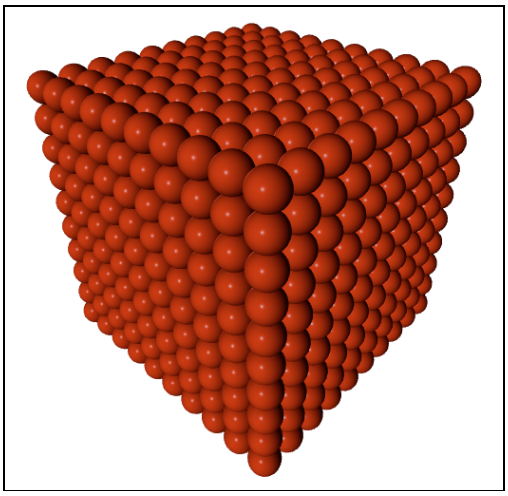


Figure 7.7 Crystal Lattice

Analysis

Three nested `for` loops (lines 08 to 11) compute the x-y-z coordinates for the red spheres. The first `for` loop in line 08 sets the positions of the sphere objects on the x-axis. The second `for` loop in line 09 sets the positions on the y-axis. The third `for` loop in line 10 sets the positions on the z-axis. In line 11, the `sphere(pos=vector(x,y,z),...)` method creates the individual red sphere objects. The representation in shown in Figure 7.7 can be interpreted as a model for a crystal lattice, with the red spheres representing positively charged atomic nuclei.

Exercise

Start the program and view the scene from different perspectives.

How many spheres are represented?

7.2.6 Penetration

A problem known from descriptive geometry, penetration explores how mutually penetrating bodies must be represented in three views, a concept often difficult for beginners. For this purpose, a program that illustrates the spatial representation of interpenetrating bodies is provided in Listing 7.8. Its graphical output is shown in Figure 7.8, which spatially represents the interpenetration of a cone and a cylinder. This program can be used to simulate different views of this penetration.

```python
#08_penetration.py
from vpython import *
rc=10.     #radius of the cone
hc=3.*rc  #height of the cone
scene.background=color.white
scene.width=600
scene.hight=600
scene.range=2.1*rc
rz=rc/2.  #radius of the cylinder
lz=2.5*rc #length of the cylinder
z=rc/1.5  #displacement of the cylinder
cone(pos=vec(0,-hc/2.5,0),axis=vec(0,hc,0),radius=rc)
cylinder(pos=vec(-lz/2.,0,z),axis=vec(lz,0,0),radius=rz)
```

Listing 7.8 Interpenetration of a Cone and a Cylinder

Output

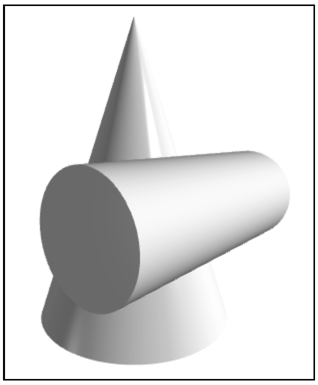


Figure 7.8 Interpenetration of a Cylinder and a Cone

Analysis

The radius `rc` and the height `hc` of the cone are the reference values (lines 03 and 04). The radius `rz` and the length `lz` of the cylinder (lines 09 and 10) depend on these values.

In line 12, a cone object is created using the `cone()` method. The `axis=vec(0,hc,0)` vector defines its orientation in the direction of the y-axis.

In line 13, a cylinder object is created using the `cylinder()` method. The `axis=vec (lz,0,0)` vector defines its orientation in the direction of the x-axis.

Exercise

Start the program and view the scene of the cone-cylinder penetration from the perspective of the front view, the side view from the left and of the top view.

Simulate different distances on the z-axis and different penetration angles.

7.2.7 Composite Bodies

You can use the `compound([G1,G2,G3, ...])` method to create solid objects from the basic shapes `G1`, `G2`, and `G3`. Listing 7.9 shows how a compound body can be created from the combination of two basic shapes: a cube and a pyramid.

```
01  #09_combination.py
02  from vpython import *
03  scene.background=color.white
04  scene.width=scene.height=600
05  a=5.
06  b=10.
07  scene.range=1.5*b
08  scene.autocenter=True
09  p1=pyramid(pos=vec(0,a,0),axis=vec(0,1,0),size=vec(a,a,a),color=color.green)
10  q1=box(pos=vector(0, a/2,0),size=vector(a,a,a),color=color.red)
11  q2=box(pos=vector(0,-b/2,0),size=vector(b,b,b),color=color.blue)
12  werkstueck=compound([p1,q1,q2])
```

Listing 7.9 Composite Body

Output

Figure 7.9 shows the resulting body composed of two cubes and a pyramid.

Analysis

In line 09, the `p1` object is created using the following method:

```
pyramid(pos=vec(0,a,0),axis=vec(0,1,0),size=vec(a,a,a)
```

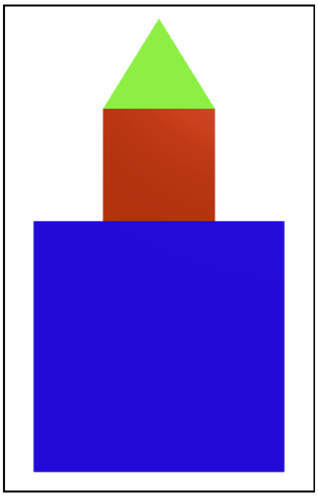


Figure 7.9 Composite Body

The pyramid object `p1` is shifted upwards by 5 units of length in the direction of the positive y-axis. The alignment is in the direction of the y-axis. The length, width, and height of the pyramid each have a value of 5 units of length.

The `q1` cube is shifted upwards by 2.5 units of length on the positive y-axis (line 10). The `q2` cube is shifted downwards by 5 units of length on the negative y-axis (line 11).

In line 12, the `compound([p1,q1,q2])` method creates the `workpiece` object as a compound body.

Exercise

Start the program and view the composite body in the front view, side view, and top view.

Test the program with different dimensions.

7.3 Bodies in Motion

The main purpose of an animation is to move bodies. As in the real world, the body objects created using VPython methods should be able to move through 3D space in accordance with the laws of physics. All mathematical operations of location changes usually run within an infinite loop, which I also refer to as an animation loop in the further course of this book. You can use the `rate(frequency)` method to set how often the animation should be executed in 1 second. The `body.pos=vector(x,y,z)` property sets the current position of the body in 3D space.

7.3.1 Vertical Movement

For a vertical movement, the x and z components of the 3D coordinates have the value zero. The `body.v=vector(0,0,0)` property initializes the velocity vector. The `body.v*dt` property computes the current position from the product of the current velocity and a freely selectable time interval `dt`. The identifier `v` is freely selectable.

Listing 7.10 animates the motion sequence of a bouncing ball. Damping influences were not taken into account in our example. The result is shown in Figure 7.10.

```python
#10_ball_vertical.py
from vpython import *
r=1. #radius
h=5. #height
scene.background=color.white
scene.center=vector(0,h,0)
box(pos=vector(0,0,0),size=vector(2*h,r/2,h), color=color.green)
ball = sphere(radius=r, color=color.yellow)
ball.pos=vector(0,2*h,0)
ball.v = vector(0,0,0) #velocity vector
g=9.81 #gravitational acceleration
dt = 0.01
while True:
    rate(100)
    ball.pos = ball.pos + ball.v*dt
    if ball.pos.y < r:
        ball.v.y = abs(ball.v.y)   #upwards
    else:
        ball.v.y = ball.v.y - g*dt #downwards
```

Listing 7.10 Bouncing Ball

Output

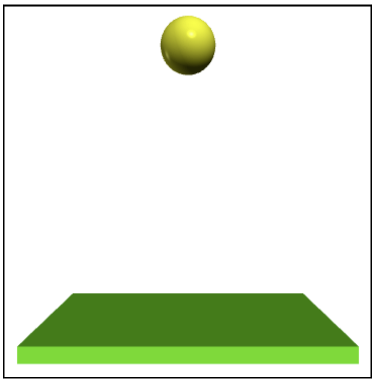


Figure 7.10 Animation of a Bouncing Ball

Analysis

Line 06 causes the floor object `box()` from line 07 to be moved down by 5 units of length. In line 08, the `ball` object is created. Line 09 determines the drop height of 10 units of length. In line 10, the velocity of the ball is initialized with the `ball.v=vector(0,0,0)` property. In line 12, the time interval `dt=0.01` was set to a realistic value.

The infinite loop runs between lines 13 to 19. This loop is executed 100 times per second because of `rate(100)` (line 14). In line 15, the new ball position ball `ball.pos` is computed from the old ball position and the product of velocity `ball.v` and time interval `dt`. If the ball position is smaller than the ball radius `r`, the ball object `ball` moves upwards (lines 16 and 17); otherwise, it moves downwards (lines 18 and 19).

Exercise

Test the program with different time intervals and frame rates.

Test the program with different radii and heights.

Test the program with different accelerations (such as Moon and Jupiter).

7.3.2 Horizontal Movement

As an example for a horizontal movement, I chose the process of induction. As shown in Listing 7.11, a bar magnet moves back and forth in the direction of the x-axis inside a coil. The coil object is created using the `helix()` method.

```
#11_cylinder_horizontal.py
from vpython import *
scene.background=color.white
scene.width=600
scene.height=300
l=10.   #length of the coil
r=l/5.  #radius of the core
scene.center=vector(0,0,0)
cs=vector(1,0.7,0.2)   #copper-colored
helix(pos=vec(-l/2,0,0),axis=vec(l,0,0),radius=1.25*r,
coils=10,thickness=0.3,color=cs)
np = cylinder(pos=vec(l/2,0,0),axis=vec(l/2,0,0),radius=r, color=color.red)
sp = cylinder(pos=vec(0,0,0),axis=vec(l/2,0,0),radius=r, color=color.green)
magnet=compound([np,sp])
magnet.pos=vector(0,0,0)
dx = 0.1
while True:
    rate(50)
    x = magnet.pos
```

```
19        x = x+vector(dx,0,0)
20        magnet.pos = x
21        if x.x>l/4. or x.x<=-l/4.:
22            dx = -dx
```

Listing 7.11 Bar Magnet Moves in Coil

Output

A snapshot of the resulting animated magnet movement is shown in Figure 7.11.

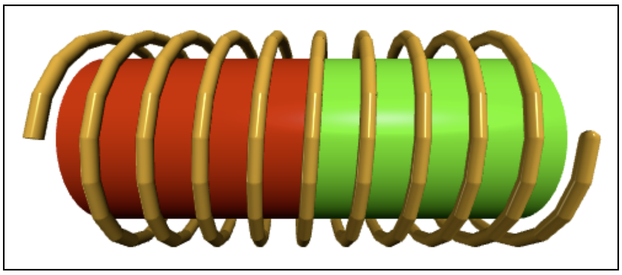


Figure 7.11 A Bar Magnet Moving in a Coil

Analysis

In line 10, the coil object is created using the `helix()` method. The left edge is shifted 5 units of length to the left on the negative x-axis. The length `l` of the coil is 10 units of length. The coil has ten turns. In line 11, the red-colored north pole `np` of the bar magnet is created using the `cylinder()` method. The same process is repeated in line 12 for the green-colored south pole `sp` of the bar magnet. In line 13, the `compound([np,sp])` method creates the `magnet` object.

The animation of the horizontal movement of the bar magnet is performed in the `while` loop (lines 16 to 22). In line 18, the `x` variable is assigned the `magnet.pos` position of the magnet initialized in line 14. The sum algorithm in line 19 calculates the new `x` position of the magnet. This position is assigned to the `magnet.pos` property in line 20. If the deflection is greater than `l/4` or less than `-l/4` (line 21), a reversal of the direction of motion occurs due to the change in sign `dx = -dx` (line 22).

7.3.3 Movement in Space

Our next example, shown in Listing 7.12, was taken from the VPython documentation. This example shows how a ball, driven by an impulse, moves back and forth between the walls of a room.

```
#12_ball_wall.py
from vpython import *
```

```
scene.width=scene.height=600
scene.background=color.white
cw=color.gray(0.9) #color of the walls
b = 5.0 #width
d = 0.3 #thickness of the wall
r=0.4   #ball radius
s2 = 2*b - d
s3 = 2*b + d
#right hand wall
box (pos=vec(b, 0, 0), size=vec(d, s2, s3), color = cw)
#left hand wall
box (pos=vec(-b, 0, 0), size=vec(d, s2, s3), color = cw)
#bottom wall
box (pos=vec(0, -b, 0), size=vec(s3, d, s3), color = cw)
#top wall
box (pos=vec(0, b, 0), size=vec(s3, d, s3), color = cw)
#back wall
box(pos=vec(0, 0, -b), size=vec(s2, s2, d), color = cw)
ball = sphere(radius=r,color=color.yellow)
ball.m = 2.0 #mass of the ball
ball.p = vec(-0.15, -0.23, 0.27) #impulse
#ball.p = vec(0,-1,0)
#ball.p = vec(-1,0,0)
#ball.p = vec(0,-1,-1)
b = b - d*0.5 - ball.radius
dt = 0.2
while True:
    rate(100)
    ball.pos = ball.pos + (ball.p/ball.m)*dt
    if not (b > ball.pos.x > -b):
        ball.p.x = -ball.p.x
    if not (b > ball.pos.y > -b):
        ball.p.y = -ball.p.y
    if not (b > ball.pos.z > -b):
        ball.p.z = -ball.p.z
```

Listing 7.12 A Ball Moving in Space

Output

A snapshot of the resulting animated ball movement is shown in Figure 7.12.

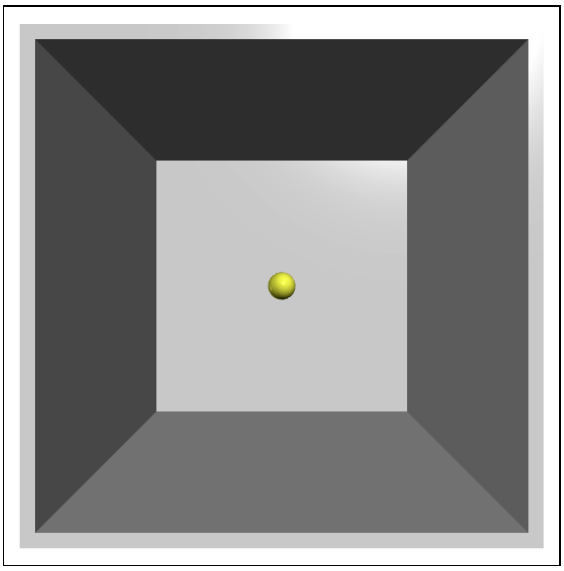


Figure 7.12 A Ball Moving in Space

Analysis

In lines 12 to 20, the box objects for the walls are created. The yellow sphere object ball created in line 21 has a radius of r=0.4 (line 08). Line 22 defines a new property m for the ball object. The identifier m is freely selectable. One could have chosen the designator mass for the mass of the ball. The ball.p vector defined and initialized in line 23 represents the impulse of a mass. The identifier p is also freely selectable. As a reminder, the following equation applies for the impulse:

$$p_x = mv_x = m\frac{\mathrm{d}x}{\mathrm{d}t}$$

This equation enables you to compute the distance traveled:

$$\mathrm{d}x = \frac{p_x}{m}\mathrm{d}t$$

The same applies to the y and z directions.

The animation of the ball movement can be performed within the while loop (lines 29 to 37). In line 31, the following sum algorithm calculates the current position of the ball from the impulse p, the mass m, and the time interval dt:

```
ball.pos = ball.pos + (ball.p/ball.m)*dt
```

When the ball bounces on the side, top, bottom, or back walls, respectively, the if queries in lines 32 to 37 cause the ball motion to reverse its direction.

Exercise

Test the program with the commented-out lines 24 to 26. What exactly is happening?

In line 21, add the `make_trail=True, retain=200` properties and restart the program.

Test the program with other masses.

7.3.4 Composite Motion

An oblique throw provides a good example of animating a composite motion. For the x- and y-components of the throwing motion, let's say the following equations apply:

$$x = v_0 \mathrm{t} \cos \alpha$$

$$y = h + v_0 \,\mathrm{t} \sin \alpha - \frac{1}{2} g t^2$$

The trajectory depends on the initial velocity $_{\mathrm{v0}}$, the throwing angle α, and the throwing height h.

Listing 7.13 shows the implementation of the animation of the oblique throw. The program does not start the motion sequence until the left mouse button is clicked on the canvas.

```
#13_oblique_throw.py
from vpython import *
h=1.2 #throwing height
b=60. #width of the reference plane
v0=22.5 #initial velocity
alpha=45. #throwing angle
alpha=radians(alpha)
g=9.81
r=b/40.
h=h+r
scene.background=color.white
scene.width=600
scene.height=600
scene.center=vector(0,b/4.,0)
ball = sphere(pos=vector(-b/2.,h,0),radius=r,color=color.yellow)
box(pos=vec(0,-b/50.,0),size=vec(b,b/25.,b/2.),color=color.green)
scene.caption="\nStart with mouse click"
scene.waitfor('click')
dt=0.01
t=0.0
while True:
    rate(50)
```

```
23        x = v0*t*cos(alpha)
24        y = h + v0*t*sin(alpha) - 0.5*g*t**2
25        ball.pos = vector(x-b/2.,y+r,0)
26        if y<=0.0:
27            break
28        t=t+dt
```

Listing 7.13 Animation of the Oblique Throw

Output

A snapshot of the resulting animated oblique throw is shown in Figure 7.13.

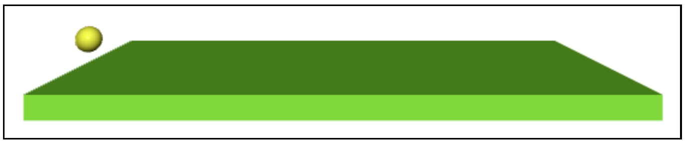


Figure 7.13 Graphic Implementation of the Animation of the Oblique Throw

Analysis

In line 18, the `scene.waitfor('click')` statement causes the animation not to be executed until after a mouse click within the scene. With this interruption, it is possible to better identify the throwing height. Within the animation loop (lines 21 to 28), the x and y components of the ball motion are computed in lines 23 and 24. In line 25, the current vector of the motion position is assigned to the `ball.pos` property. When the ball reaches the reference plane (line 26), the `break` statement aborts the animation.

Exercise

Test the program with different throwing angles and throwing velocities.

Vary the throwing heights as well.

7.3.5 Rotational Motion

The rotational motion of a body on an elliptical trajectory can be animated by the x-y coordinates that change in time:

$$x = a \cos \omega t$$
$$y = b \sin \omega t$$

If the semiaxes a and b are equal, a circular motion is animated; otherwise, the body moves on an elliptical trajectory.

Rotation of a Body on an Elliptical Trajectory

Listing 7.14 animates the movement of the Moon on an elliptical orbit around the Earth. The mean orbital eccentricity of the Moon (0.0549) has been greatly exaggerated to illustrate that the Earth is not at the center of the ellipse.

```
01  #14_elliptical_orbit.py
02  from vpython import *
03  scene.width=600
04  scene.height=600
05  b=10.     #semiaxis of the ellipse
06  a=1.157*b #semiaxis of the ellipse
07  Rm=1.     #Moon radius
08  Re=3.7*Rm #Earth radius
09  rem=10.*Re #Earth-Moon distance
10  scene.background=color.white
11  earth = sphere(pos=vector(0.1*a,0,0),radius=Re,texture=textures.earth)
12  moon = sphere(pos=vector(rem,0,0),radius=Rm,color=color.gray(0.8))
13  w=1.0 #angular velocity
14  t=0
15  dt=1e-3
16  while  True:
17      rate(100)
18      x = a*cos(w*t)
19      y = b*sin(w*t)
20      moon.pos = vector(x,y,0)
21      t=t+dt
```

Listing 7.14 Animation of the Elliptical Orbit of the Moon

Output

A snapshot of the resulting animated elliptical orbit of the moon is shown in Figure 7.14.

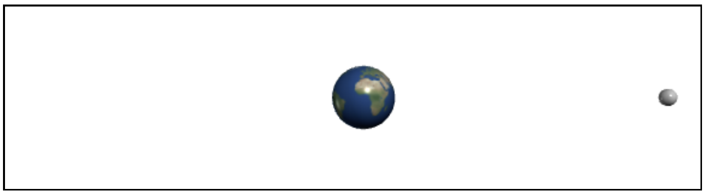


Figure 7.14 Animation of the Elliptical Orbit of the Moon

Analysis

In lines 05 to 09, the data for our Earth-Moon planetary system is defined. In line 11, the `sphere()` method creates the `earth` object. The spherical `earth` object is moved to the right on the positive x-axis by the amount `0.1*a`. We chose this unrealistically high value to illustrate that the geometric location of the Earth does not coincide with the center of the ellipse. The surface of a body object can represented visually with the `texture` property; obviously, we chose the `textures.earth` value. The `earth` object would not actually need to be created explicitly because it will no longer be used further down in the source code. The situation is different with the Moon. An object must be created for the Moon because it is needed in the animation loop. The creation of the `moon` object takes place in line 12.

The angular velocity `w=1.0` does not correspond to reality (line 13). We set this value arbitrarily to better understand the motion of the moon.

The `while` loop (lines 16 to 21) implements the animation. In line 20, for each time `t` the `moon.pos` property is assigned the x-y coordinates determined in lines 18 and 19.

Rotation of a Body on a Circular Trajectory

VPython also provides a simple method to animate circular motion. The following method animates the motion of a circular trajectory:

```
body .rotate(angle=w*dt,axis=vec(0,1,0),origin=vec(0,0,0))
```

The `angle` property must be assigned the angle `w*dt`. The rotational motion comes about by computing a new angle for each time (`dt`) from the angular velocity `w`, which is assumed to be constant. The axis vector `axis` specifies the axis of rotation, and the `origin` property specifies the center of rotation.

Listing 7.15 animates the rotation of a cube. The commented-out lines are for testing purposes.

```
#15_rotation1.py
from vpython import *
scene.width=scene.height=600
scene.background=color.white
scene.center=vec(0,0,0)
r=1.
col=color.green
scene.range=1.5*r
red = box(pos=vec(0,0,0),axis=vec(0,1,0),size=vec(r,r,r),color=col)
#red =ring(pos=vec(0,0,0),axis=vec(0,0,1),radius=r,thickness=r/5.)
#red=ellipsoid(pos=vec(0,0,0),axis=vec(1,0,0),size=vec(2.0*r,r,r))
#red = arrow(pos=vec(0,0,0), axis=vec(r,0,0), color=col)
dt = 0.05
w=0.5     #angular velocity
```

```
15  while True:
16      rate(25)
17      red.rotate(angle=w*dt,axis=vec(0,1,0),origin=vec(0,0,0))
```

Listing 7.15 A Rotating Cube

Output

A snapshot of the resulting animation of a rotating cube is shown in Figure 7.15.

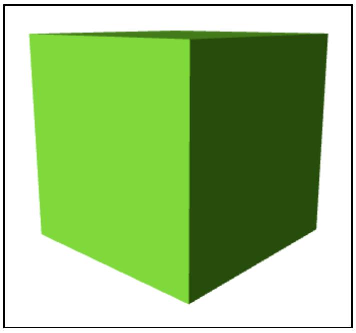


Figure 7.15 Animation of a Rotating Cube

Analysis

In line 09, the `box()` method creates the `red` object with edge length `r`.

In line 17, the following method makes the cube rotate around the y-axis:

```
rotate(angle=w*dt,axis=vec(0,1,0),origin=vec(0,0,0))
```

By changing the angular velocity `w` in line 14, you can change the rotation frequency. The center of rotation is located at the origin (zero).

> **Exercise**
>
> Test the program with different angular velocities. Also try different axes of rotation.
>
> Test the program with the bodies commented out.

Rotation of Multiple Bodies

Within an animation loop, multiple bodies can also be moved on a circular trajectory with different angular velocities. Listing 7.16 shows how to animate the rotational motions of the inner planets Mercury, Venus, and the Earth around the Sun and display the result, as shown in Figure 7.16. The relative distances between the planets do not correspond to reality, and all the planets rotate around the z-axis.

```
01  #16_rotation2.py
02  from vpython import *
03  scene.width=scene.height=600
04  scene.background=color.white
05  R=5.0 #radius of the sun
06  r=10.0 #Sun-Mercury distance
07  sphere(pos=vec(0,0,0),axis=vec(1,0,0),radius=R,color=color.yellow)
08  mercury=sphere(pos=vec(r,0,0),axis=vec(1,0,0),
radius=0.2*R,color=color.red)
09  venus=sphere(pos=vec(2*r,0,0),axis=vec(1,0,0),
radius=0.3*R,color=color.green)
10  earth=sphere(pos=vec(3*r,0,0),axis=vec(1,0,0),
radius=0.5*R,texture=textures.earth)
11  dt = 0.05
12  w1=0.3
13  w2=0.2
14  w3=0.1 #angular velocity
15  while True:
16      rate(25)
17      mercury.rotate(angle=w1*dt,axis=vec(0,0,1),origin=vec(0,0,0))
18      venus.rotate(angle=w2*dt,axis=vec(0,0,1),origin=vec(0,0,0))
19      earth.rotate(angle=w3*dt,axis=vec(0,0,1),origin=vec(0,0,0))
```

Listing 7.16 Animation of the Planetary Motions in the Solar System

Output

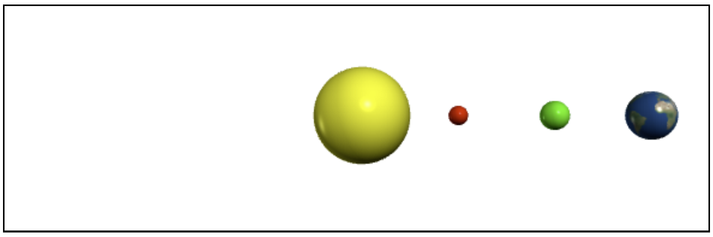


Figure 7.16 Animation of the Planetary Motions in the Solar System

Analysis

Lines 05 and 06 define the radius of the Sun `R` and the distance between Mercury and the Sun `r`. The radii of the planets are defined as fractions of the solar radius, and their distances are defined as a multiple of `r`.

The planetary objects, `mercury`, `venus`, and `earth` are generated in lines 08 to 10.

Lines 12 to 14 define the angular velocities of the planetary objects. The greater the distance between a planet and the Sun, the smaller its angular velocity must be.

Within the `while` loop (lines 15 to 19), the `rotate()` method is applied to the planetary objects `mercury`, `venus`, and `earth`. The `axis=vec(0,0,1)` vector specifies that the planets rotate around the z-axis.

Exercise

Test the program with different angular velocities and distances.

Let the planets rotate around the y-axis.

7.3.6 Random Motion

In Brownian motion, small particles, such as pollen grains, move randomly and jerkily in different directions in a fluid. These movements are caused by the thermal motion of the liquid molecules that are in the vicinity of a particle. Such random movements can be animated in VPython by generating random numbers for the x-y coordinates using the `random()` function. This function generates random numbers between 0 and 1.

The x-y coordinates are recomputed with each loop pass:

$$x = \sin\alpha\cos\varphi$$
$$y = \sin\alpha\sin\varphi$$

The angle α is generated with a random number generator just like the angle φ. The first angle has a value between 0 and π, and the second angle has a value between 0 and 2π. The sin α function calculates the distances of the particles from the origin of the coordinate system. The $\cos\varphi$ and $\sin\varphi$ functions compute the randomly distributed x-y coordinates.

Listing 7.17 shows how to animate the random, two-dimensional motion of a sphere. The trajectories of the sphere are displayed as thin blue lines using the `attach_trail(object, ...)` method.

```
#17_random.py
from vpython import *
a=10.
r=a/20.
scene.background=color.white
scene.width=scene.height=600
scene.center=vector(0,0,0)
scene.range=a
part = sphere(radius=r,color=color.red)
part.pos=vector(0,0,0)
attach_trail(part,radius=0.05,color=color.blue)
i=0
while i<10:
    sleep(0.8)
```

```
15      alpha = pi*random()
16      phi = 2.0*pi*random()
17      x = a*sin(alpha)*cos(phi)
18      y = a*sin(alpha)*sin(phi)
19      part.pos=vector(x,y,0)
20      i=i+1
```

Listing 7.17 Animation of Random Motions

Output

A snapshot of the resulting animation of a randomly moving sphere is shown in Figure 7.17.

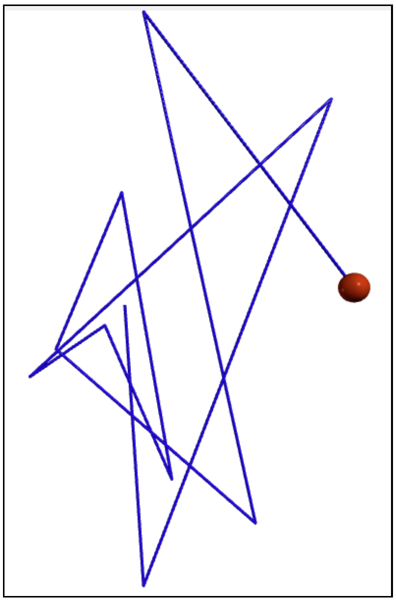


Figure 7.17 Random Motions

Analysis

In line 09, the `sphere()` method creates the `part` object. In line 10, the `part.pos=vector(0,0,0)` property places the initial position of the `part` sphere in the origin of the coordinate system. The `attach_trail(part, ...)` method in line 11 causes the trajectory curves to be traced as blue lines with `radius=0.05`. The animation is executed ten times within the `while` loop (lines 13 to 20). In line 14, the `sleep(0.8)` function ensures that the loop pass gets interrupted for 0.8 seconds.

In line 15, the `random()` function generates random numbers between 0 and π for computing the randomly distributed distances of the sphere from the coordinate origin. The random numbers generated in line 16 lie in the interval from 0 to 2π. In lines 17 and 18, the randomly distributed x-y coordinates are computed.

Exercise

Test the program with different interruption times.

Test the program with different radii for the motion lines (line 11).

7.4 Animation of Oscillations

A pendulum deflected from its rest position performs oscillations. These pendulum motions can be animated just as well as any other motion. When animating a pendulum in motion, you must first set up the differential equation system that describes the motion of the pendulum. This differential equation system is then solved within the animation loop using a simple summation algorithm. Based on the solution of the differential equation system, the x-y positions of the pendulum can then be calculated.

7.4.1 Simple Pendulum

The motion of an ideal simple pendulum (mathematical pendulum) can be described by the following differential equation:

$$\frac{\mathrm{d}^2\varphi}{\mathrm{d}t^2}+\frac{g}{l}\sin\varphi=0$$

Next, we'll use the following substitution:

$$\frac{\mathrm{d}^2\varphi}{\mathrm{d}t^2}=\frac{\mathrm{d}\omega}{\mathrm{d}t}$$

Thus, you'll obtain a differential equation system with two first-order differential equations:

$$\frac{\mathrm{d}\varphi}{\mathrm{d}t}=\omega$$
$$\frac{\mathrm{d}\omega}{\mathrm{d}t}=-\frac{g}{l}\sin\varphi$$

Since accuracy is not absolutely essential for animations but instead the efficiency of the algorithm is most important, a convenient way to solve this differential equation system is using the Euler method.

Listing 7.18 animates the motion of a simple pendulum. The differential equation system is solved within the animation loop using the sum algorithm of the Euler method. The pendulum consists of a cylinder object (`cylinder`), which represents the thread, and a sphere object (`sphere`), which represents the mass.

```
#18_pendulum.py
from vpython import *
y0=-5.  #shift on the y-axis
```

```
b=5.    #width of the ceiling
l=8.    #length of the pendulum
phi=45. #deflection
r=0.5   #radius of the sphere
scene.width=600
scene.height=600
scene.center =vector(0,y0,0)
scene.range=1.5*b
scene.background = color.white
box(size=vector(b,b/20.,b/2.),color=color.gray(0.8)) #ceiling
rod=cylinder(axis=vector(0,l,0),radius=0.05)
mass = sphere(radius=r,color=color.red)
mass.pos=vector(0,rod.pos.y,0)
g=9.81   #gravitational acceleration
w02=g/l  #square of the angular frequency
phi=radians(phi)
w=0.  #initial angular velocity
dt=0.02
while True:
    rate(100)
    phi=phi+w*dt
    w=w-w02*sin(phi)*dt
    x= l*sin(phi)
    y=-l*cos(phi)
    rod.axis=vector(x,y,0)
    mass.pos  =vector(x,y,0)
```

Listing 7.18 Simple Pendulum

Output

A snapshot of the resulting animation of a mathematical pendulum is shown in Figure 7.18.

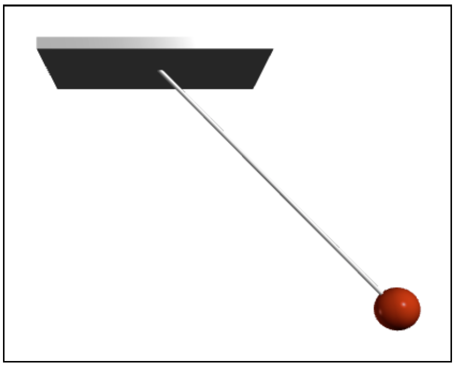


Figure 7.18 Animation of a Pendulum Motion

Analysis

Line 06 defines the deflection angle `phi`. The differential equation system of the pendulum motion (lines 24 and 25) is solved within the animation loop (lines 22 to 29) using the sum algorithm `phi=phi+w*dt` and `w=w-w02*sin(phi)*dt`. Lines 26 and 27 calculate the x and y coordinates of the current sphere position. In lines 28 and 29, the positions of the rod and of the sphere are updated.

Exercise

Test the program with different deflection angles.

Test the program with different pendulum lengths.

7.4.2 Spring Pendulum

The pendulum motion of a spring pendulum can be described by the following differential equation:

$$\frac{\mathrm{d}^2y}{\mathrm{d}t^2}+\frac{c}{m}y=0$$

Using the substitution

$$\frac{\mathrm{d}^2y}{\mathrm{d}t^2}=\frac{\mathrm{d}v}{\mathrm{d}t}$$

the following differential equation system is obtained:

$$\frac{\mathrm{d}y}{\mathrm{d}t}=v$$
$$\frac{\mathrm{d}v}{\mathrm{d}t}=-\frac{c}{m}y$$

Listing 7.19 animates oscillations of a spring-mass system. The mass is represented by a spherical object.

```
01  #19_spring_pendulum.py
02  from vpython import *
03  y0=-5. #shift on the y-acis
04  b=8.   #width of the ceiling
05  l=0.8*y0 #length of the spring
06  r=1.2  #radius of the mass
07  c=1.1  #spring constant
08  m=1.5  #mass of the sphere
09  scene.width=600
10  scene.height=600
11  scene.center =vector(0,y0,0)
12  scene.background = color.white
13  box(pos=vector(0,b/40.,0),size=vector(b,b/20.,b/2.),
color=color.gray(0.8)) #ceiling
14  spring=helix(axis=vector(0,l,0),radius=0.6,color=color.yellow)
15  spring.thickness=0.2
16  spring.coils=8
17  mass=sphere(pos=spring.pos,radius=r,color=color.red)
18  w02=c/m   #square of the angular frequency
19  y=-0.6*l  #deflection
20  v=0.      #initial velocity
21  dt=0.02
22  while True:
23      rate(100)
24      y=y+v*dt
25      v=v-w02*y*dt
26      spring.axis=vector(0,y+l,0)
27      mass.pos =vector(0,y+l-r,0)
```

Listing 7.19 Spring-Mass Oscillator

Output

A snapshot of the resulting animation of a spring pendulum is shown in Figure 7.19.

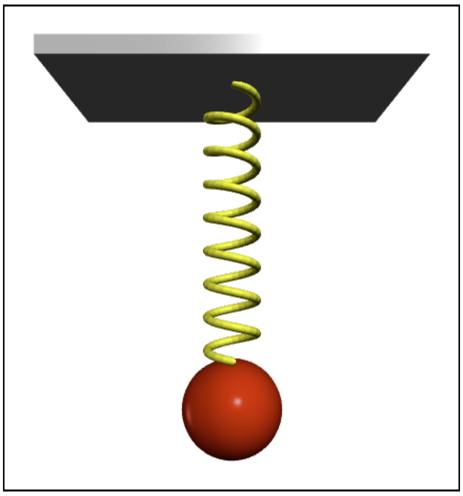


Figure 7.19 Animation of a Spring Pendulum

Analysis

Lines 03 and 11 cause the shift of the coordinate origin by 5 units of length upwards. In lines 05 to 08, the data of the spring-mass oscillator is defined.

In line 14, the `helix()` method creates the spring object `spring`. The properties of the spring object are complemented in lines 15 and 16. In line 17, the `sphere()` method creates the sphere object `mass`.

The deflection (initial value) is set to 60% of the spring length `l` (line 19).

Within the infinite loop (lines 22 to 27), the differential equation system (lines 24 and 25) is solved using the Euler method.

The positions of the end of the spring and the center of the sphere are updated in lines 26 and 27.

Exercise

Test the program with different masses.

Test the program with different spring constants.

7.5 Event Processing

For event processing, VPython also provides controls such as command buttons (`button`), radio buttons (`radio`), multiple selection options (`checkbox` and `menu`), and sliders (`slider`).

For each event, a function must be defined to perform the relevant action. Events are always implemented according to the following schema:

```
control(bind=function, ...)
```

The `control` identifier can be the name of a control, such as a `button`, `slider`, `checkbox`, or `radio` button. To enable the `controlelement` method to trigger an event, a custom function must be passed to it as a parameter. This function is assigned to the `bind` property. The parentheses of the custom function must be omitted. All other parameters depend on the type of control. Listing 7.20 shows how to use the `slider()` method to change the rotational frequency of a voltage pointer. The `checkbox()` method enables the activation of a power pointer that rotates with double frequency. The `button()` method can be used to pause and restart the animation.

```
#20_event-processing.py
from vpython import *
scene.title="<h2>Rotating voltage and power pointer</h2>"
scene.width=scene.height=600
scene.background=color.white

runs = True
col=color.yellow

def start(b):
    global runs
    runs = not runs
    if runs: b.text = "Pause"
    else: b.text = "Start"

def omega(s):
    txtA.text = "{:1.2f}".format(s.value)

def visibleP(b):
    if b.checked:
        p.visible = True
    else:
        p.visible = False

u_s=2.
p_s=1.5
d=0.025
scene.range = 1.2*u_s
u=arrow(pos=vec(0,0,0),axis=vec(0,u_s,0),color=color.blue)
p=arrow(pos=vec(0,0,0),axis=vec(p_s,0,0),color=col)
p.visible=False
u.shaftwidth=d
p.shaftwidth=d
button(text="Pause",pos=scene.title_anchor,bind=start)
scene.append_to_caption("\n\n")
scene.caption="\n  Change frequency:\n\n"
sldF=slider(min=0,max=6.28,value=1,length=300,bind=omega,right=4)
txtA=wtext(text="{:1.2f}".format(sldF.value))
scene.append_to_caption(" rad/s\n\n")
checkbox(bind=visibleP, text="Show power pointer\n\n")
dt=0.01
w=1.
while True:
    rate(1/dt)
    if runs:
        w=sldF.value
        u.rotate(angle=w*dt,axis=vec(0,0,1))
        p.rotate(angle=2.0*w*dt,axis=vec(0,0,1))
```

Listing 7.20 Event Processing

Output

A snapshot of the resulting rotating pointers animation is shown in Figure 7.20.

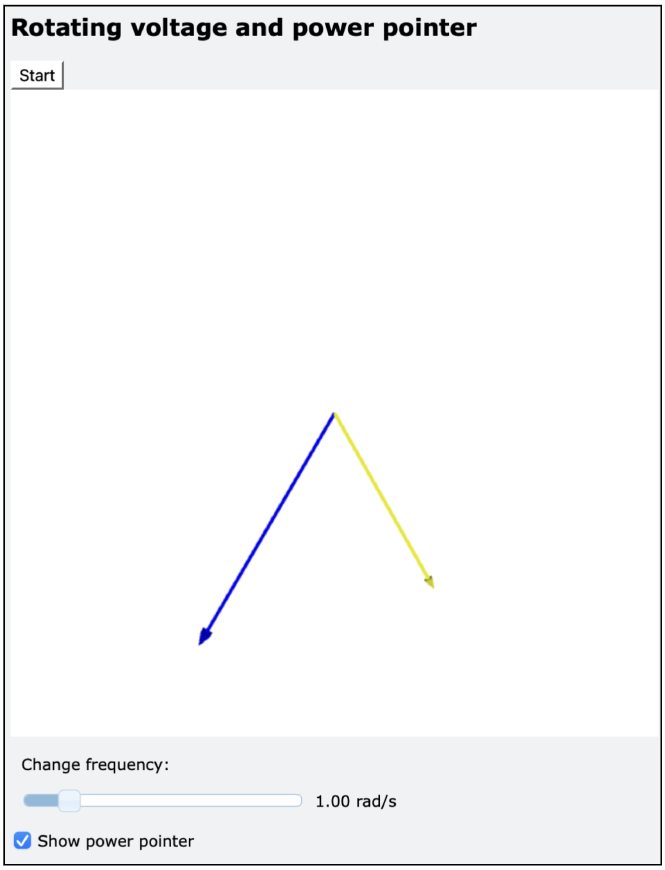


Figure 7.20 Event Processing

Analysis

If the value of the global `runs` variable (lines 07 and 11) is `True`, the animation is executed within the `while` loop (lines 43 to 48). If you want to pause the animation, you must click the **Pause** button. Then, the label of the button changes to **Start**. In this case, the `button()` method from line 34 calls the custom `start(b)` function from lines 10 to 14. The command button is placed above the scene in the upper-left corner. The `start(b)` function is called via `bind=start`. The parentheses of the function definition and the function argument `b` must be omitted.

In line 37, the `slider()` method calls the custom `bind=omega` function from lines 16 and 17. The set values are stored in the `sldF` object and displayed in the `txtA` text field in line 38. In line 46, the change of the rotation frequency is performed using the `w=sldF.value` assignment.

The `checkbox(bind=visible,...)` method in line 40 calls the custom function `visible(b)` from lines 19 to 23. If you activate the `checkbox` control, the power pointer will be switched on.

Exercise

Test the program with all settings.

Comment out line 11. Restart the program and click the **Pause** button. Analyze the error message.

7.6 Project Task: Animation of a Coupled Spring Pendulum

In this task, we need to animate the oscillations of a coupled spring pendulum consisting of two spring-mass systems with spring constants c_1 and c_2 and masses m_1 and m_2 in VPython. The spring-mass system oscillates in the direction of the y-axis. Damping should be neglected for the time being.

This solution consists of three steps:

1. Set up the differential equation system:

$$\frac{\mathrm{d}^2 y_1}{\mathrm{d}t} + \frac{c_1 + c_2}{m_1} y_1 - \frac{c_2}{m_1} y_2 = 0$$

$$\frac{\mathrm{d}^2 y_2}{\mathrm{d}t^2} + \frac{c_2}{m_2} (y_2 - y_1) = 0$$

2. Convert this equation system into a first-order differential equation system using the following substitutions:

$$\frac{\mathrm{d}^2 y_1}{\mathrm{d}t^2} = \frac{\mathrm{d}v_1}{\mathrm{d}t}$$

and

$$\frac{\mathrm{d}^2 y_2}{\mathrm{d}t^2} = \frac{\mathrm{d}v_2}{\mathrm{d}t}$$

Thus, you obtain the following first-order differential equation system:

$$\frac{\mathrm{d}y_1}{\mathrm{d}t} = v_1$$

$$\frac{\mathrm{d}v_1}{\mathrm{d}t} = -\frac{c_1 + c_2}{m_1} y_1 + \frac{c_2}{m_1} y_2$$

$$\frac{\mathrm{d}y_2}{\mathrm{d}t} = v_2$$

$$\frac{\mathrm{d}v_2}{\mathrm{d}t} = -\frac{c_2}{m_2} (y_2 - y_1)$$

3. Set up the solution algorithm for the differential equation system using the Euler method:

   ```
   y1=y1+v1*dt
   v1=v1-(c1+c2)/m1*y1*dt + c2/m1*y2*dt
   y2=y2+v2*dt
   v2=v2-c2/m2*(y2-y1)*dt
   ```

This algorithm must then be inserted within the animation loop. Listing 7.21 shows the implementation.

```python
#21_double_spring_pendulum.py
from vpython import *
y0=-5. #shift on the y-axis
b=10.  #width of the ceiling
r=1.2  #radius of the mass
l=0.9*y0
c1=1.  #spring constant
m1=1.  #mass of the sphere
c2=1.
m2=1.
scene.width=600
scene.height=800
scene.center =vector(0,2*y0,0)
scene.background = color.white
box(pos=vector(0,b/40.,0),size=vector(b,b/20.,b/2.),
color=color.gray(0.8)) #ceiling
spring1 = helix(pos=vector(0,0,0),axis=vector(0,l,0),
                color=color.yellow,radius=0.5*r,thickness=0.2,coils=10)
mass1 = sphere(pos=spring1.feder1.pos,radius=r, color=color.red)
spring2 = helix(pos=vector(0,l,0),axis=vector(0,l,0),
                color=color.green,radius=0.5*r,thickness=0.2,coils=10)
mass2 = sphere(pos=vector(0,2*l,0),radius=r, color=color.blue)
y1=-0.6*l #deflection
y2=0
v1=v2=0   #initial velocity
lk=l-r
dt=0.02
while True:
    rate(50)
    y1=y1 + v1*dt
    v1=v1-(c1+c2)/m1*y1*dt+c2/m1*y2*dt #-0.05*v1*dt
    y2=y2 + v2*dt
    v2=v2-c2/m2*(y2-y1)*dt #-0.05*v2*dt
```

```
33      spring1.axis=vector(0,y1+l,0)
34      mass1.pos =vector(0,y1+lk,0)
35      spring2.axis=vector(0,y1+y2+l,0)
36      spring2.pos.y =mass1.pos.y
37      mass2.pos =spring2.pos+vector(0,y1+y2+lk,0)
```

Listing 7.21 Coupled Spring Pendulum

Output

A snapshot of the resulting animation of a coupled spring pendulum is shown in Figure 7.21.

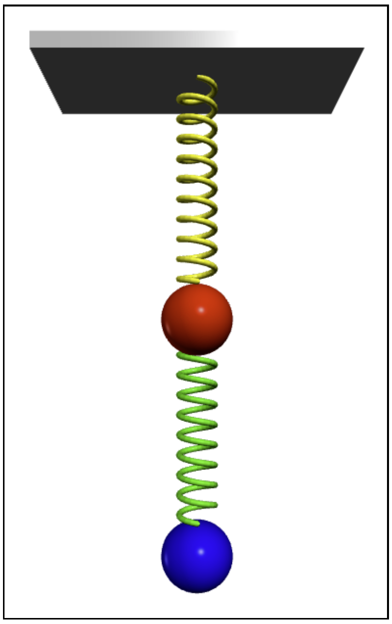


Figure 7.21 Coupled Spring Pendulum

Analysis

In lines 07 to 10, the masses and spring constants of the two coupled springs can be changed.

In lines 16 to 21, the `sphere()` and `helix()` methods create the spring and mass objects.

In lines 29 to 32, the differential equation system is solved using the Euler method.

The `axis` property of `spring1` and `spring2` causes the two springs to deflect only in the direction of the y-axis (lines 33 and 35).

In line 36, the `spring2.pos.y` property is assigned the current position of `mass1`.

Line 37 causes `mass2` to be positioned at the end of `spring2`.

Exercise

Test the program with different masses.

Test the program with different spring constants.

In certain constellations, the spring-mass system reacts unstably.

Test the program with different damping values by removing the comments in lines 30 and 32.

7.7 Project Task: Animation of Two Coupled Simple Pendulums

For this next task, we need to animate a pendulum system that consists of two simple mathematical pendulums, whose masses m are connected via a spring (spring constant c), as shown in Figure 7.22. First, you must set up the differential equation system of the spring-mass system again:

$$\frac{\mathrm{d}^2\varphi_1}{\mathrm{d}t^2}+\frac{g}{l}\varphi_1-\frac{c}{m}(\varphi_2-\varphi_1)=0$$
$$\frac{\mathrm{d}^2\varphi_2}{\mathrm{d}t^2}+\frac{g}{l}\varphi_2+\frac{c}{m}(\varphi_2-\varphi_1)=0$$

Then, you'll use the following substitutions:

$$\frac{\mathrm{d}^2\varphi_1}{\mathrm{d}t^2}=\frac{\mathrm{d}\omega_1}{\mathrm{d}t}$$

and

$$\frac{\mathrm{d}^2\varphi_2}{\mathrm{d}t^2}=\frac{\mathrm{d}\omega_2}{\mathrm{d}t}$$

Thus, you'll obtain the following first-order differential equation system:

$$\frac{\mathrm{d}\varphi_1}{\mathrm{d}t}=\omega_1$$
$$\frac{\mathrm{d}\omega_1}{\mathrm{d}t}=-\frac{g}{l}\varphi_1+\frac{c}{m}(\varphi_2-\varphi_1)$$
$$\frac{\mathrm{d}\varphi_2}{\mathrm{d}t}=\omega_2$$
$$\frac{\mathrm{d}\omega_2}{\mathrm{d}t}=-\frac{g}{l}\varphi_2-\frac{c}{m}(\varphi_2-\varphi_1)$$

Using the abbreviations

$$\omega_0^2=\frac{g}{l}$$

and

$$k=\frac{c}{m}$$

you develop the algorithm according to the Euler method from the first-order differential equation system:

```
phi1=phi1+w1*dt
w1=w1-w02*phi1*dt+k*(phi2-phi1)*dt #-0.05*w1*dt
phi2=phi2+w2*dt
w2=w2-w02*phi1*dt-k*(phi2-phi1)*dt
```

This algorithm is inserted into the animation loop of the program. Listing 7.22 shows the implementation.

```
#22_double_pendulum.py
from vpython import *
phi1=radians(-5.)
phi2=radians(5.)
b=12.   #width of the ceiling
y0=-b/2.#shift on the y-axis
a=b/2.  #distance between the pendulums
l=0.9*b #length of the pendulums
r=b/15. #radius of the spheres
m=10.   #mass of the spheres
c=4.5   #spring constant
scene.width=600
scene.height=600
scene.center=vector(0,y0,0)
scene.range=0.8*b
scene.background = color.white
box(size=vector(b,b/20.,b/4.),color=color.gray(0.8)) #ceiling
rod1=cylinder(axis=vector(0,l,0),radius=0.05)
rod1.pos=vector(-a/2.,0,0)
rod2=cylinder(axis=vector(0,l,0),radius=0.05)
rod2.pos=vector(a/2.,0,0)
mass1 = sphere(radius=r,color=color.red)
mass2 = sphere(radius=r,color=color.blue)
spring=helix(axis=vector(a,0,0),radius=0.4)
spring.thickness=0.1
spring.coils=10
g=9.81   #gravitational acceleration
w02=g/l  #pendulum frequency
k=c/m    #spring frequency
w1=w2=0  #angular velocity
dt=0.02
while True:
    rate(100)
    phi1=phi1+w1*dt
```

```
35        w1=w1-w02*phi1*dt+k*(phi2-phi1)*dt #-0.05*w1*dt
36        phi2=phi2+w2*dt
37        w2=w2-w02*phi1*dt-k*(phi2-phi1)*dt #-0.05*w2*dt
38        x1= l*sin(phi1)
39        y1=-l*cos(phi1)
40        x2= l*sin(phi2)
41        y2=-l*cos(phi2)
42        rod1.axis=vector(x1,y1,0)
43        mass1.pos =vector(x1-a/2.,y1,0)
44        rod2.axis=vector(x2,y2,0)
45        mass2.pos =vector(x2+a/2.,y2,0)
46        spring.pos=mass1.pos+vector(r,0,0)
47        spring.axis.x=x2-x1+a-2*r
48        spring.axis.y=y2-y1
```

Listing 7.22 Coupled Simple Pendulums

Output

A snapshot of the resulting animation of a simple coupled pendulum is shown in Figure 7.22.

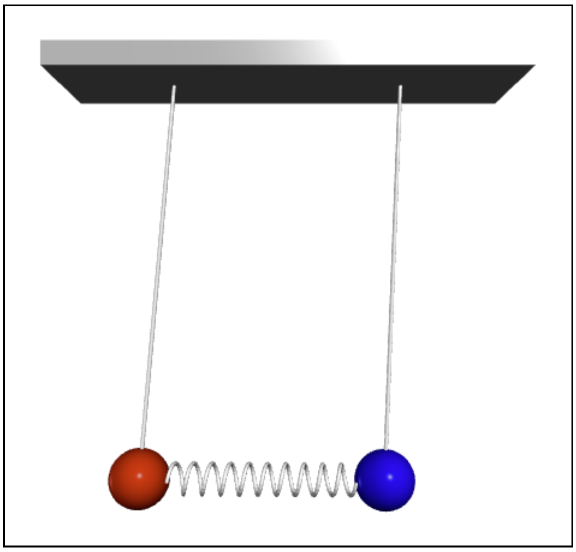


Figure 7.22 Coupled Simple Pendulums

Analysis

In lines 03 and 04, you can change the deflection angles phi1 and phi2 of both pendulums.

Lines 10 and 11 define the masses of the pendulums and the spring constant of the coupling spring.

The differential equation system is solved in lines 34 to 37. The damping values are commented out. They can be removed for testing purposes.

In lines 38 to 41, the current x-y coordinates are calculated from the deflection angles `phi1` and `phi2`.

In lines 42 to 48, the current positions are assigned to each pendulum and the coupling spring.

Exercise

Test the program with different masses.

Test the program with different spring constants.

With certain settings, the double pendulum reacts unstably.

Test the program with different damping values by removing the comments in lines 35 and 37.

7.8 Tasks

1. Four spheres are supposed to touch each other in space. Write a VPython program to represent this.
2. A graph consists of four nodes and has the shape of a parallelogram. All nodes are connected to each other. The nodes are to be simulated as *points*. Write a VPython program to represent this.
3. A cylinder oriented in the direction of the x-axis penetrates another cylinder oriented in the direction of the y-axis. Write a VPython program to represent this.
4. Write a VPython program that creates an octahedron.
5. Write a VPython program that animates the motions of the Moon and the Earth around the Sun. To simplify things, assume that the planets move along circular orbits.

Chapter 8

Computing with Complex Numbers

This chapter describes how you can compute alternating current (AC) networks, frequency responses and locus curves using complex calculus.

Complex numbers extend the real number range into the range of imaginary numbers. A complex number z consists of a real part a and an imaginary part b:

$z = a + jb$

Complex numbers can be represented in the complex number plane, also called the *Gaussian number plane*, as shown in Figure 8.1.

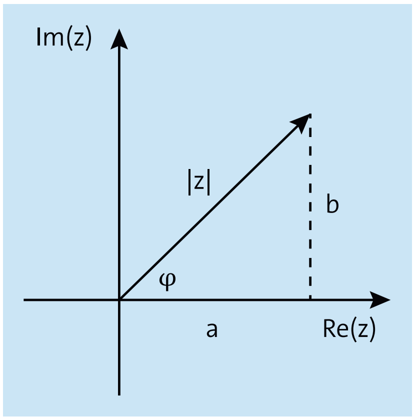


Figure 8.1 Complex Numbers

Complex numbers can be added, subtracted, divided, and multiplied directly in the Python console. You can enter complex numbers together with the mathematical operators into the console and then perform arithmetic operations by pressing `Return`. The integration of a module is not required for basic mathematical operations, such as the following:

```
>>> z1=1+2j
>>> z2=3+4j
>>> z1+z2
(4+6j)
>>> z1-z2
(-2-2j)
```

```
>>> z1/z2
(0.44+0.08j)
>>> z1*z2
(-5+10j)
```

You can also perform other mathematical operations defined for complex numbers, for instance, exponentiation, root calculation, logarithmizing, and so on.

8.1 Mathematical Operations

Listing 8.1 shows how selected mathematical operations on two complex numbers $z_1 = 5 - j12$ and $z_2 = 3 + j4$ must be implemented using the NumPy module.

```python
#01_operations.py
import numpy as np
n=3
z1=5-12j
z2=complex(3,4)
z2conj=np.conjugate(z2)
rez2=np.real(z2)
imz2=np.imag(z2)
absolutevalue=np.abs(z2)
angle=np.angle(z2)
s=z1+z2
d=z1-z2
p=z1*z2
q=z1/z2
pot=z2**n
w=np.sqrt(z2)
lg=np.log(z2)
hs=np.sinh(z2)
#Outputs
print("Complex number z1:",z1)
print("Complex number z2:",z2)
print("Conjugated     z2:",z2conj)
print("Real part      z2:",rez2)
print("Imaginary part z2:",imz2)
print("Absolute value of    z2:",absolutevalue)
print("Angle of   z2:",np.angle(z2,deg=True),"°")
print("Sum of      z1+z2:",s)
print("Difference z1-z2:",d)
print("Product    z1*z2:",p)
print("Quotient   z1/z2:",q)
print("%1d.Power"%n,"of  z2:",pot)
```

```
32 print("Square root of    z2:",w)
33 print("Logarithm   z2:",lg)
34 print("sinh z2:",hs)
35 print("Type of z1:",type(z1))
36 print("Type of z2:",type(z2))
```

Listing 8.1 Basic Mathematical Operations with Complex Numbers

Output

```
Complex number z1: (5-12j)
Complex number z2: (3+4j)
Conjugated   z2: (3-4j)
Real part      z2: 3.0
Imaginary part  z2: 4.0
Absolute value of    z2: 5.0
Angle of     z2: 53.13010235415598°
Sum of  z1+z2: (8-8j)
Difference  z1-z2: (2-16j)
Product     z1*z2: (63-16j)
Quotient    z1/z2: (-1.32-2.24j)
Cube of  z2: (-117+44j)
Root of     z2: (2+1j)
Logarithm    z2: (1.6094379124341003+0.9272952180016122j)
sinh           z2: (-6.5481200409110025-7.61923172032141 1j)
Type of z1: <class 'complex'>
Type of z2: <class 'complex'>
```

Analysis

Complex numbers can be defined with `z1=5-12j` (line 04) or with `complex(3,4)` (line 05).

The NumPy function `np.conjugate(z2)` generates the conjugate complex number `z2conj` from the complex number `z2` (line 06).

The real part and the imaginary part of the complex number `z2` can be calculated using the `np.real(z2)` and `np.imag(z2)` functions (lines 07 and 08).

To determine the absolute value and the angle of `z2`, you must apply the `np.abs(z2)` and `np.angle(z2)` functions (lines 09 and 10).

Lines 11 to 14 perform some basic mathematical operations on `z1` and `z2`.

Line 15 calculates the cube of `z2`, and line 16 calculates its root.

Lines 17 and 18 compute the natural logarithm and the sine hyperbolic of `z2`.

The output of the results for the mathematical operations on the complex numbers `z1` and `z2` occurs in lines 20 to 34.

8.2 Euler's Formula

Euler's formula describes the projections of a complex number z with the magnitude $r = |z|$ and the angle φ on the real and imaginary axes of the Gaussian number plane. The following equation applies:

$$z = re^{j\varphi} = r\cos\varphi + j\,r\sin\varphi$$

Listing 8.2 compares whether the following calculations provide the same values:

```
z1=r*np.exp(1j*phi)
z2=r*np.cos(phi)+1j*r*np.sin(phi)
```

```
01  #02_euler.py
02  import numpy as np
03  r=10
04  phi=np.radians(30)
05  z1=r*np.exp(1j*phi)
06  z2=r*np.cos(phi)+1j*r*np.sin(phi)
07  #Outputs
08  print("z1:",z1)
09  print("z2:",z2)
10  print("Magnitude z1:",np.abs(z1))
11  print("Magnitude z2:",np.abs(z2))
12  print("Type of z1:",type(z1))
13  print("Type of z2:",type(z2))
```

Listing 8.2 Implementation of Euler's Formula

Output

```
z1: (8.660254037844387+4.999999999999999j)
z2: (8.660254037844387+4.999999999999999j)
Magnitude z1: 10.0
Magnitude z2: 10.0
Type of z1: <class 'numpy.complex128'>
Type of z2: <class 'numpy.complex128'>
```

Analysis

Line 03 specifies the magnitude `r` of the complex numbers `z1` and `z2`. In line 04, the `np.radians(30)` function converts the angle of 30° into the radian. In line 05, the exponential function `np.exp(1j*phi)` is passed the imaginary part of a complex number (the angle!) as an argument. The real and imaginary parts of the angle, multiplied by the amount `r`, are stored in the `z1` variable. Note that the imaginary unit `j` must always be preceded by a `1`.

In line 06, the term of the right hand side of Euler's formula is assigned to the `z2` variable.

As expected, the outputs of lines 08 and 09 prove that the results computed in lines 05 and 06 match.

In lines 10 and 11, the NumPy function `np.abs()` calculates the amounts of the complex numbers `z1` and `z2`. Both results are identical.

8.2.1 Symbolic Method

Euler's formula entails an important consequence for the computation of AC networks.

Due to $\varphi = \omega t$, voltages and currents can be represented as pointers rotating with an angular velocity of ω .

The following equation then applies to the voltage pointer:

$$\underline{U} = Ue^{\mathrm{j}\omega t} = U(\cos\omega t + \mathrm{j}\sin\omega t)$$

For the current pointer, the following equation applies:

$$\underline{I} = Ie^{\mathrm{j}\omega t} = I(\cos\omega t + \mathrm{j}\sin\omega t)$$

The phase shift between voltage and current, which is usually always present, has not yet been taken into account.

If you apply the rules of complex calculus, you can compute AC networks with sinusoidal feed as if they were DC networks. In the literature, this method is also referred to as the symbolic method.

Based on this method, total voltages and currents are calculated by adding up the real and imaginary parts of the individual partial voltages and currents separately in each case according to the rules of complex calculus. The multiplication and division of complex resistors must also be performed according to the calculation rules of the complex calculus. You can use the symbolic method only if all voltages and currents of the network are sinusoidal.

8.3 Calculating with Complex Resistors

Of course, Ohm's law also applies to AC networks:

$$\underline{Z} = \frac{\underline{U}}{\underline{I}} = \frac{Ue^{\mathrm{j}(\omega t + \varphi_u)}}{Ie^{\mathrm{j}(\omega t + \varphi_i)}} = Ze^{\mathrm{j}(\varphi_u - \varphi_i)} = Z(\cos\varphi + \mathrm{j}\sin\varphi)$$

When dividing the voltage pointer by the current pointer, the frequency is reduced. Thus, pointers of complex resistors do not rotate. In any AC network, only ohmic, inductive, and capacitive resistances occur. For a given angular frequency ω, the following applies with regard to the total resistance:

$$\underline{Z} = R + \mathrm{j}\omega L - \mathrm{j}\frac{1}{\omega C} = R + \mathrm{j}X_L - \mathrm{j}X_c$$

The inductive and the capacitive parts can be combined to an imaginary part X:

$$\underline{Z} = R \pm \mathrm{j}X$$

The real part R is formed by all ohmic resistances. If the inductive component predominates, the imaginary part of the complex resistance has a positive sign. The current lags behind the voltage. If the capacitive part predominates, the imaginary part of the complex resistance has a negative sign. The current rushes ahead of the voltage. If the imaginary part disappears, the network behaves like an ohmic resistor.

Complex currents

$$\underline{I} = \frac{\underline{U}}{\underline{Z}}$$

and complex powers

$$\underline{P} = \underline{U} \cdot \underline{I}$$

can be computed using Python by declaring the AC resistors $\underline{Z}$ as complex variables. For a series circuit of an ohmic resistor of R = 10 Ω and an inductive reactance X_L = 5 Ω, the impedance is declared as a complex variable: `Z=complex(10,5)`. Alternatively, you can write `Z=10+5j`.

Based on the example T-circuit shown in Figure 8.2, I will demonstrate how the complex calculus is carried out in real-life situations.

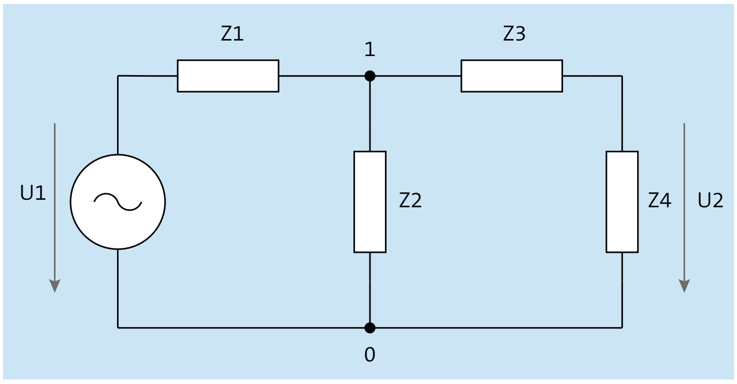


Figure 8.2 T-Circuit with Complex Resistors

From this T-circuit, you can read the equivalent resistance $\underline{Z}_i$:

$$\underline{Z}_i = \underline{Z}_3 + \frac{\underline{Z}_1 \underline{Z}_2}{\underline{Z}_1 + \underline{Z}_2}$$

The three impedances of the T-circuit can be reduced to a single complex resistor $\underline{Z}_i$ using the equivalent voltage source method. The original T-circuit then consists only of a series circuit of the equivalent resistance, which can be interpreted as the internal resistance of the voltage source, and the complex load resistance $\underline{Z}_4$.

In this way, the calculation of the current and the complex output voltage as well as the complex output power is considerably simplified.

$$\underline{I} = \frac{\underline{U}_1}{\underline{Z}_i + \underline{Z}_4}$$

$$\underline{U}_2 = \underline{Z}_4 \cdot \underline{I}$$

$$\underline{P}_2 = \underline{U}_2 \cdot \underline{I}$$

Listing 8.3 computes the output voltage, current, and output power of the T-circuit.

```
01  #03_t_circuit.py
02  import numpy as np
03  U1=230
04  Z1=1+2j
05  Z2=10-12j
06  Z3=1+2j
07  Z4=10-10j
08  Zi=Z1*Z2/(Z1+Z2)+Z3
09  I2=U1/(Zi+Z4)
10  U2=Z4*I2
11  P2=U2*I2
12  print("Internal resistance:",np.round(Zi,decimals=2), "ohms")
13  print("Output current: ",np.round(I2,decimals=2), "A")
14  print("Output voltage:",np.round(U2,decimals=2), "V")
15  print("Output power:",np.round(P2,decimals=2), "W")
```

Listing 8.3 T-Circuit

Output

```
Internal resistance: (2.33+3.94j) ohms
Output current:   (15.02+7.39j) A
Output voltage: (224.07-76.35j) V
Output power: (3929.73+508.34j) W
```

Analysis

The input voltage `U1` is defined as `int` in line 03. In lines 04 to 07, the AC resistors `Z1` to `Z4` are defined. The AC resistors have the `complex` data type because their imaginary parts have been marked by a `j`.

Line 08 computes the equivalent resistance `Zi` of the T-circuit as a complex resistance because the variables `Z1` to `Z4` on the right-hand side of the assignment were defined as complex magnitudes.

Line 09 computes the current `I2` flowing through the load resistor `Z4`.

In lines 10 and 11, the output voltage `U2` and the power consumption `P2` of the load resistor are calculated.

The output of the results is performed in lines 12 to 15. The sign of the imaginary part of the output power `P2` is positive, which means that inductive reactive power is "implemented" in the complex load resistor `Z4`. Because it is reactive power, the inductive portion of `Z4` absorbs the amount of 508.34W in one half-period and feeds it back into the grid in the second half-period.

8.4 Function Plots with Complex Magnitudes

You can also use Python to represent frequency responses for complex resistances. However, one prerequisite is that you must divide the complex magnitudes into their real and imaginary parts. Let's use the example of a series resonant circuit and a two-port network, which is constructed from two inductors—a resistor and a capacitor—to illustrate how to implement function plots of frequency responses and locus curves.

8.4.1 Complex Frequency Response of a Series Resonant Circuit

Figure 8.3 shows the circuit of a series resonant circuit. For this circuit, the real and imaginary parts of the current are represented as a function of the angular frequency.

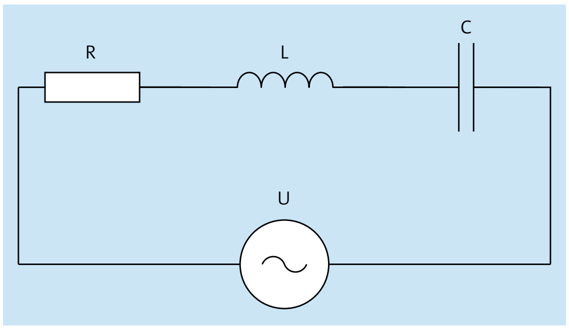


Figure 8.3 Series Resonant Circuit

The complex resistance of the series resonant circuit consists of the real part R and the two imaginary parts $\mathrm{j}\omega L$ and $1/\mathrm{j}\omega C$ with the following equation:

$$Z(\mathrm{j}\omega) = R + \mathrm{j}\omega L + \frac{1}{\mathrm{j}\omega C}$$

To compute the complex frequency response of the current I(jω), the voltage must be divided by the complex resistance, with the following equation:

$$I(\mathrm{j}\omega) = \frac{U}{Z(\mathrm{j}\omega)}$$

Listing 8.4 computes the frequency response of the complex current flowing through the series resonant circuit and plots its real and imaginary parts as a function of the angular frequency.

```
01  #04_series_resonant_circuit.py
02  import numpy as np
03  import matplotlib.pyplot as plt
04  U=230 #int!
05  R=10  #int!
06  C=1e-6
07  L=1e-2
08  w=2.0*np.pi*np.linspace(0.01,3e3,1000)
09  Z=R+1j*w*L+1.0/(1j*w*C)
10  I=U/Z
11  fig, ax=plt.subplots(figsize=(8,6))
12  ax.plot(w,np.real(I),lw=2,color="red",label="Real part")
13  ax.plot(w,np.imag(I),lw=2,color="green",label="Imaginary part")
14  ax.set(xlabel="$\omega$ rad/s",ylabel=" I in A ")
15  ax.legend(loc="best")
16  ax.grid(True)
17  plt.show()
```

Listing 8.4 Frequency Response of the Current Waveform of a Series Resonant Circuit

Output

The resulting output is the current waveform of the real and the imaginary parts represented as a curve, as shown in Figure 8.4.

Analysis

Lines 04 to 07 provide the data. Line 08 creates a NumPy array for the angular frequency `w`.

Line 09 contains the formula for the impedance of the series resonant circuit. In line 10, the current `I` is calculated. A number of the `int` type is divided by a number of the `complex` type. The result is a number of the `complex` type.

In lines 12 and 13, the Matplotlib method `plot()` prepares the function plot for displaying the real and imaginary parts. In line 17, the function plot is displayed on the screen via `plt.show()`.

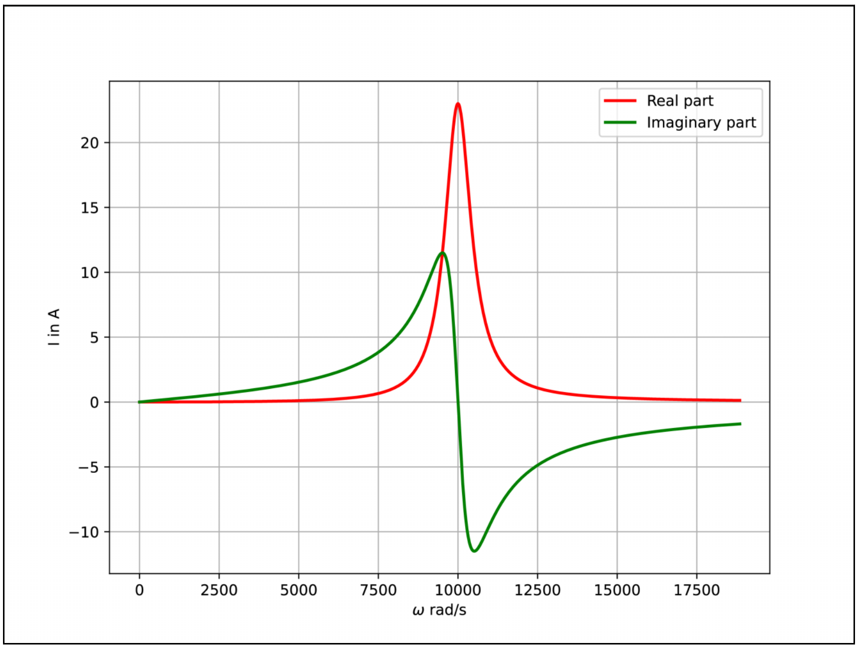


Figure 8.4 Current Waveform of Real and Imaginary Parts

8.4.2 Locus Curves

Locus curves represent the course of a complex resistance (*impedance*) or the course of a complex conductance (*admittance*) in the complex number plane as a function of the angular frequency. For each discrete value of an angular frequency, the real and imaginary parts must be computed from the impedance or admittance. These values are drawn into the complex number plane as points. If you connect these points, you obtain the locus curve.

For the two-port network shown in Figure 8.5, we need to compute the locus curve of the complex resistance and represent this curve graphically as a function plot.

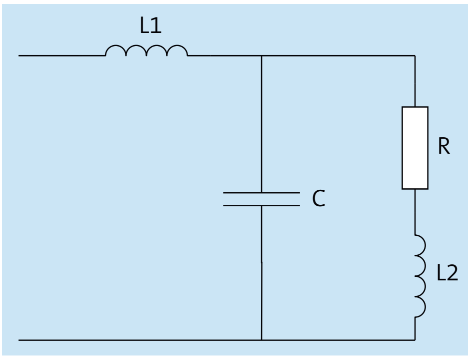


Figure 8.5 Two-Port Network

From the circuit, you can directly read the complex resistance in the following way:

$$Z(\mathrm{j}\omega) = \mathrm{j}\omega L_1 + \frac{(R + \mathrm{j}\omega L_2)\dfrac{1}{\mathrm{j}\omega C}}{R + \mathrm{j}\omega L_2 + \dfrac{1}{\mathrm{j}\omega C}}$$

To manually draw the locus curve of this impedance, you must split the complex term into its real and imaginary parts, which would be quite tedious.

Listing 8.5 carries out this task using the NumPy functions `np.real(Z)` and `np.imag(Z)`.

```
01  #05_locus_curve.py
02  import numpy as np
03  import matplotlib.pyplot as plt
04  R=20
05  C=1e-6
06  L1=5e-2
07  L2=1e-2
08  wp=10e3 #angular frequency at point p
09  #complex resistance
10  def Z(w):
11      return 1j*w*L1+(R+1j*w*L2)*(1/(1j*w*C))/(R+1j*w*L2+1/(1j*w*C))
12
13  w=2.0*np.pi*np.linspace(0.01,3e3,500)
14  Zp=Z(wp)
15  fig, ax = plt.subplots(figsize=(8,6))
16  ax.plot(np.real(Z(w)),np.imag(Z(w)),lw=2)
17  ax.plot(np.real(Zp),np.imag(Zp),"o",c="red")
18  ax.set(title="Locus curve",xlabel="Real part in $\Omega$", ylabel=
"Imaginary part in $\Omega$")
19  ax.grid(True)
20  plt.show()
```

Listing 8.5 Locus Curve

Output

Figure 8.6 shows the output in a function plot.

Analysis

In lines 04 to 07, you can define the values for the components. In line 08, you can enter a selected location of the angular frequency `wp`, which will be shown in the locus curve in connection with line 14 `Zp=Z(wp)` and line 17 as a red point in the function plot. At the angular frequency of $10 \cdot 10^3$ s^{-1}, the real part of the two-port network has a value of 500Ω, and the imaginary part has a value of 400Ω. Thus, Z = 500Ω + j400Ω.

Line 10 defines the function for the impedance $Z(\omega)$ of the two-port network.

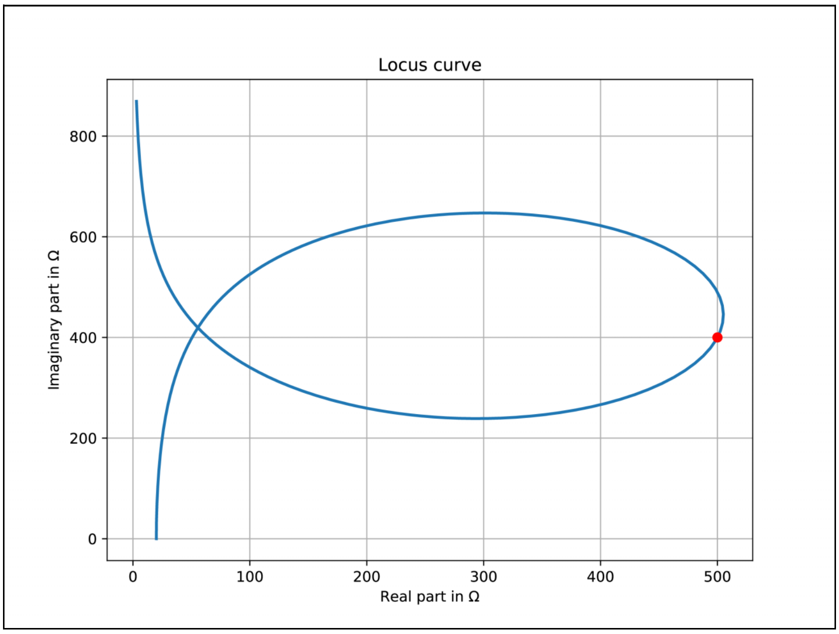


Figure 8.6 Locus Curve for a Two-Port Network

In line 13, a NumPy array for the range of the angular frequency `w` is created.

In line 16, the display of the locus curve is prepared and displayed on the screen in line 20 by using `plt.show()`.

8.5 Project Task: Electric Power Transmission System

Let's say, for a 50 Hz three-phase line, the line-to-line voltage (e.g., 380 kV) and the power P_2 at the end of the line are given. Now, we need to determine the input voltage U_1 and the current I_1 at the sending end.

These input variables depend on the transmission characteristics of the line and the line length. Figure 8.7 shows the schematic representation of an electrical line.

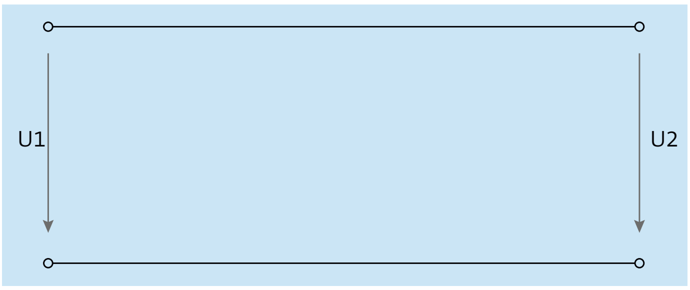


Figure 8.7 Schematic Representation of an Electrical Line

Four electrical constants, called *primary line constants*, completely determine the transmission characteristics of a line. These constants include the resistance coating R', the inductance coating L', the dielectric coating G', and the capacitance coating C'. The magnitudes refer to 1km of line length.

The following data is given:

- Line length: $l = 400$km
- Line-to-line voltage at line end: $U_2 = 380$kV
- Active power at line end: $P_2 = 360$MW
- Power factor: $\cos\varphi = 1$
- Primary line constants:
  - $R' = 31$mΩ/km
  - $L' = 0.8$mH/km
  - $G' = 0.02$µS/km
  - $C' = 14.3$nF/km

8.5.1 Task

a) The input voltage U_1, the input current I_1, the input power S_1, and the efficiency of the line should be computed using a Python program.

b) Using a T-equivalent circuit diagram of the line, the results from a) must also be checked using a Python program.

Solution to A

1. Determine the phase voltage U_2 and the line-to-line current I_2 at the end of the line:

$$U_2 = \frac{U_{2\triangle}}{\sqrt{3}} \approx 220 \text{ kV}$$

$$I_2 = \frac{P_{2\triangle}}{\sqrt{3}\, U_{2\triangle} \cos\varphi} \approx 547 \text{ A}$$

Set up the formulas for the propagation constant y and the characteristic impedance $\underline{Z}_w$. Based on the primary line constants, the propagation constant and the characteristic impedance can be determined.

$$\gamma = \sqrt{(R + \mathrm{j}\omega L)(G + \mathrm{j}\omega C)}$$

$$\underline{Z}_w = \sqrt{\frac{R' + \mathrm{j}\omega L'}{G' + \mathrm{j}\omega C'}}$$

2. Set up the formulas for computing the input voltage $\underline{U}_1$ and the input current $\underline{I}_1$.

At the sending end, the following applies to the voltage and current:

$$\underline{U}_1 = \cosh \gamma l \cdot \underline{U}_2 + \underline{Z}_w \sinh \gamma l \cdot \underline{I}_2$$

$$\underline{I}_1 = \frac{\sinh \gamma l}{\underline{Z}_w} \cdot \underline{U}_2 + \cosh \gamma l \cdot \underline{I}_2$$

3. Implement the equations as Python source code.

The line data is implemented as assignments, as is usually the case. For the line equations, we can use the NumPy functions `np.sinh()` and `np.cosh()`.

Listing 8.6 computes the complex voltage $\underline{U}_1$, the complex current $\underline{I}_1$, the complex input power $\underline{S}_1$, and the efficiency of the line.

```
#06_line1.py
import numpy as np
P2a=360e6  #power at line end
U2a=380e3  #line-to-line voltage at line end
l=400      #line length
f=50       #frequency
phi=0      #phase shift
R=31e-3    #resistance coating
L=0.8e-3   #inductance coating
G=0.02e-6  #dielectric coating
C=14.3e-9  #capacitance coating
#Computations
w=2*np.pi*f
U2=U2a/np.sqrt(3)
I2=P2a/(np.sqrt(3)*U2a*np.cos(phi))
Zw=np.sqrt((R+1j*w*L)/(G+1j*w*C))
g=np.sqrt((R+1j*w*L)*(G+1j*w*C))
U1=   np.cosh(g*l)*U2 + Zw*np.sinh(g*l)*I2
I1=np.sinh(g*l)/Zw*U2 +    np.cosh(g*l)*I2
S1=3*U1*np.conjugate(I1)/1e6
eta=1e-4*P2a/np.real(S1)
#Output
print("Characteristic impedance: %5.2f\u03A9, %5.1f°" \
      %(np.abs(Zw),np.angle(Zw,deg=True)))
print("Input voltage: %5.2f V, %5.1f°"\
      %(np.abs(U1),np.angle(U1,deg=True)))
print("Input current   : %5.2f A, %5.1f°"\
      %(np.abs(I1),np.angle(I1,deg=True)))
print("Input power: %5.0f MW %5.0f Mvar"\
      %(np.real(S1),np.imag(S1)))
print("Efficiency \u03B7 = %5.0f percent" %(eta))
```

Listing 8.6 Input Variables of a Three-Phase Line

Output

```
Characteristic impedance: 237.42Ω,  -3.4°
Input voltage: 213673.23 V,  15.1°
Input current   : 632.10 A,  37.9°
Input power:   374 MW  -157 Mvar
Efficiency η = 96 percent
```

Analysis

Lines 03 to 11 define the data of the line according to the specifications.

In lines 16 and 17, the characteristic impedance `Zw` and the propagation constant `g` are calculated, in lines 18 and 19 the input voltage `U1` and the input current `I1` of the power transmission system with the line equations.

Line 20 calculates the apparent power `S1` at the input of the line. From the real part of `S1`, the efficiency `eta` of the line can then be calculated in line 21.

The outputs are made in lines 23 through 31. The NumPy function `np.angle(Zw,deg=True)` calculates the phase shift angle and converts the angle from radians to degrees. The results exactly match the data from the technical literature.

8.5.2 Equivalent Circuit Diagram of a Three-Phase Power Line

The equivalent circuit describes only the transmission behavior of a phase in the three-phase line.

Solution to B

You can determine the complex resistances $\underline{Z}_1$ and $\underline{Z}_3$ for the T-equivalent circuit of a line, shown in Figure 8.2, using the characteristic impedance, the hyperbolic tangent, and the propagation constant:

$$\underline{Z}_1 = \underline{Z}_3 = \underline{Z}_w \tanh\frac{\gamma l}{2}$$

The complex resistance $\underline{Z}_2$ can be computed using the hyperbolic sine:

$$\underline{Z}_2 = \frac{\underline{Z}_w}{\sinh \gamma l}$$

You can determine the load resistance $\underline{Z}_4$ from the output voltage $\underline{U}_2$ and the output current $\underline{I}_2$:

$$\underline{Z}_4 = \frac{\underline{U}_2}{\underline{I}_2}$$

The voltage $\underline{U}_{1,0}$ at the resistor $\underline{Z}_2$ is composed of the voltage drop at the resistor $\underline{Z}_3$ and the voltage drop at the load resistor $\underline{Z}_4$:

$$\underline{U}_{1,0} = (\underline{Z}_3 + \underline{Z}_4)\underline{I}_2$$

The index 1.0 represents the two nodes to which the resistor $\underline{Z}_2$ is connected. The voltage $\underline{U}_{1,0}$ enables you to determine the current $\underline{I}_{1,0}$, which flows through the resistor $\underline{Z}_2$:

$$\underline{I}_{1,0} = \frac{\underline{U}_{1,0}}{\underline{Z}_2}$$

The input current $\underline{I}_1$ represents the sum of $\underline{I}_{1,0}$ and $\underline{I}_2$:

$$\underline{I}_1 = \underline{I}_{1,0} + \underline{I}_2$$

For the input voltage, we obtain:

$$\underline{U}_1 = \underline{Z}_1 \underline{I}_1 + \underline{U}_{1,0}$$

You only need to enter these equations as Python source code into the editor of the development environment. Listing 8.7 shows the implementation.

```
#07_line2.py
import numpy as np
P2a=360e6  #power at line end
U2a=380e3  #line-to-line voltage at line end
l=400      #line length
f=50       #frequency
phi=0      #phase shift
R=31e-3    #resistance coating
L=0.8e-3   #inductance coating
G=0.02e-6  #dielectric coating
C=14.3e-9  #capacitance coating
#Computations
w=2*np.pi*f
U2=U2a/np.sqrt(3)
I2=P2a/(np.sqrt(3)*U2a*np.cos(phi))
Zw=np.sqrt((R+1j*w*L)/(G+1j*w*C))
g=np.sqrt((R+1j*w*L)*(G+1j*w*C))
Z1=Zw*np.tanh(0.5*g*l)
Z2=Zw/np.sinh(g*l)
Z3=Z1
Z4=U2/I2
U10=(Z3+Z4)*I2
I10=U10/Z2
I1=I10+I2
U1=Z1*I1+U10
#Output
print("Z1= %3.2f \u03A9  %3.2fj \u03A9"\
    %(np.real(Z1),np.imag(Z1)))
print("Z2= %3.2f \u03A9  %3.2fj \u03A9"\
    %(np.real(Z2),np.imag(Z2)))
print("Z3= %3.2f \u03A9  %3.2fj \u03A9"\
32      %(np.real(Z3),np.imag(Z3)))
33  print("Output current %5.3f A" %(I2))
34  print("Input voltage: %5.2f V, %5.1f°"\
35        %(np.abs(U1),np.angle(U1,deg=True)))
36  print("Input current : %5.2f A, %5.1f°"\
37        %(np.abs(I1),np.angle(I1,deg=True)))
```

Listing 8.7 Calculation with Equivalent Circuit

Output

```
Z1= 6.40 Ω  51.02j Ω
Z2= 0.32 Ω  -573.58j Ω
Z3= 6.40 Ω  51.02j Ω
Output current 546.963 A
Input voltage: 213673.23 V,  15.1°
Input current   : 632.10 A,  37.9°
```

Analysis

The values for the longitudinal members `Z1` and `Z3`, as well as the transverse member `Z2`, are calculated in lines 18 and 19.

Lines 21 to 25 calculate step by step the input voltage `U1` and the input current `I1`, starting from the end of the equivalent circuit.

The outputs are made in lines 27 through 37. The results from lines 34 and 36 are consistent with the results from Listing 8.6.

8.6 Tasks

1. Calculate the square root of $\sqrt{10-\mathrm{j}5}$ using the Python console.
2. Write a program to convert complex resistors connected as a star into a delta circuit.
3. For a parallel resonant circuit, the frequency response $I = f(\mathrm{j}\omega)$ must be represented as real and imaginary parts. Write a program that solves this task.
4. Write a program for plotting the locus curve of a characteristic impedance (see Task 5 for data).
5. Let's say we have a telephone cable with the primary line constants:
   - $R' = 60\Omega/\mathrm{km}$
   - $L' = 0.6\mathrm{mH/km}$
   - $G' = 1\mu\mathrm{S/km}$
   - $C' = 50\mathrm{nF/km}$

Now, calculate the voltage U_2 and the current I_2 at the end of the line at a frequency of $f = 1$ kHz. The input voltage is $U_1 = 10$V, and the input current has a value of $I_1 = 10$ mA. For these line equations, the following applies:

$$\underline{U}_2 = \cosh \gamma l \cdot \underline{U}_1 - \underline{Z}_w \sinh \gamma l \cdot \underline{I}_1$$

$$\underline{I}_2 = -\frac{\sinh \gamma l}{\underline{Z}_w} \cdot \underline{U}_1 + \cosh \gamma l \cdot \underline{I}_1$$

Chapter 9

Statistical Computations

In this chapter, you'll learn how to calculate and analyze important statistical characteristics from normally distributed random numbers using NumPy and SciPy. I'll show you how to use a simulation program to graphically represent the means and standard deviations of samples in a two-track quality control chart.

Statistical analyses are responses to the ever-increasing complexity of society and technology. Complexity unsettles and awakens the (unfulfillable) desire to know the future. The quantitative description of the actual state of affairs forms the basis for decisions in politics and management. *Empirical social research* aims to identify social trends and risks. Election research tries to predict the outcome of an election, and epidemiology tries to assess the course and dangers of an epidemic. Statistics almost always venture a glimpse into the future, even if they seem to observe and describe only the conditions of the present. Because we cannot capture the full extent of a population's dataset, we take samples, analyze them, and provide interpretive guidance for future planning.

Important key figures include the *mean values* and the *scatter values* of a statistical characteristic. Python provides a wide range of options for statistical computations with its `statistics`, `numpy`, and `scipy` modules. The `statistics` module is part of the standard Python package. The `numpy` and `scipy` modules must be installed subsequently. Table 9.1 contains an overview of selected statistical functions.

Terminology	Python	NumPy	SciPy
Median	`median(a)`	`median(a)`	
Modal value	`mode(a)`		`mode(a)`
Arithmetic mean value	`mean(a)`	`mean(a)`	
Harmonic mean value	`harmonic_mean(a)`		`hmean(a)`
Geometric mean value	`geometric_mean(a)`		`gmean(a)`

Table 9.1 The Statistical Methods of Python, NumPy, and SciPy

Terminology	Python	NumPy	SciPy
Standard deviation	`stdev(a)`	`std(a)`	`tstd(a)`
Skew			`skew(a)`

Table 9.1 The Statistical Methods of Python, NumPy, and SciPy (Cont.)

The data to be analyzed is stored in an array `a`. Above all, the SciPy submodule `stats`, with its overwhelming number of statistical functions, is a powerful tool for extensive statistical analyses. In this chapter, I'll focus on the treatment of important statistical mean values, the standard deviation, and the regression analysis of normally distributed physical measurements. The dimensions of a workpiece (such as the gear shaft shown in Figure 9.1) are supposed to represent these measurement values. Since no real measurement values are available, they are generated using a random number generator, stored in a file, and read from the file for statistical analysis. The results of the statistical analysis are then visualized by histograms and quality control charts.

In mechanical engineering, the process quality of a manufacturing process must be permanently monitored to ensure the quality required by the customer. For this purpose, a random sample of usually five workpieces is taken every hour from running production. Software then analyzes the samples using statistical tools. The process quality can be thus assessed based on the profile of the mean values and standard deviations from the individual samples.

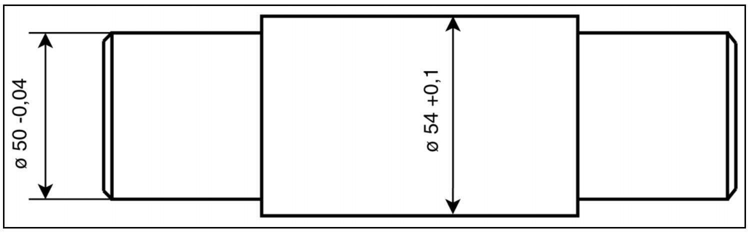


Figure 9.1 Gear Shaft

Figure 9.1 shows a technical drawing of a gear shaft with the required dimensions. Basically, all dimensions given in the drawing can be the subject of statistical analysis. However, we assume that the measurement values are approximately normally distributed, which in real life is almost always the case. Each technical drawing provides all relevant values for statistical analysis, such as the following values:

- Length and width dimensions
- Diameter of the shaft
- Roughness depths
- Hardness according to the Rockwell scale (HRC)

For other production processes, the following values may also be relevant:

- Filling quantities of certain foods or substances
- Coating thicknesses
- Resistance values of ohmic resistors
- Capacitance values of capacitors
- Coil inductance values

The basic idea behind *statistical process control* (SPC) is to infer the process quality of the entire manufacturing process based on the statistical analysis of a relatively small number (sample) of workpieces. Corrective action can thus be taken in the process, before further rejects are produced.

9.1 Generating, Saving, and Reading Measurement Values

Since no measurement values from a real manufacturing process are available, a random number generator must generate them. Python's NumPy module provides the `random.normal()` function for this purpose. The measurement values are thus simulated by software.

9.1.1 Generating Measurement Values

The NumPy module provides an efficient function for the simulation of normally distributed numbers. The general syntax for generating *n* normally distributed random numbers is the following:

```
values=np.random.normal(setpoint,standarddeviation,size=n)
```

This statement creates a one-dimensional array of n normally distributed random numbers with the mean `setpoint` value and the standard deviation `standarddeviation`. The random numbers generated in this way are stored in the `values` variable. Using the `[]` operator, you can access the individual elements of the array. Listing 9.1 shows the implementation for generating ten normally distributed random numbers.

```
01  #01_generate.py
02  import numpy as np
03  n=10
04  setpoint=50
05  s=1
06  values=np.random.normal(setpoint,s,size=n)
07  rvalues=np.around(values,decimals=2)
08  print("Normally distributed values:")
09  print(values)
```

```
10  print("Rounded values:")
11  print(rvalues)
12  print("Type of values:",type(values))
```

Listing 9.1 Generating Normally Distributed Random Numbers

Output

```
Normally distributed values:
[48.8918586  50.38567219 49.46935897 51.13788815 49.80800167 47.74986333
51.54947603 50.14987419 49.50516999 49.43358825]
Rounded values:
[48.89 50.39 49.47 51.14 49.81 47.75 51.55 50.15 49.51 49.43]
Type of values: <class 'numpy.ndarray'>
```

Analysis

Line 02 imports the `numpy` module. As usual, the `np` identifier is assigned as an alias. Line 03 specifies the number of random numbers to be generated. We arbitrarily specified the number 50 as the setpoint of a measured variable (line 04). The dispersion measure of the standard deviation `s` has a reference value of 1 (line 05), which has also been specified arbitrarily. Line 06 generates ten normally distributed random numbers using the `np.random.normal(setvalue,s,size=10)` NumPy function and assigns these numbers to the `values` variable. In line 07, the decimal numbers are rounded to 2-digit precision using the `np.arrond(values,decimals=2)` NumPy function.

Lines 09 and 11 output the random numbers. Line 12 specifies the type of variable values with `<class 'numpy.ndarray'>`. The `ndarray` data structure is typical of NumPy: In this case, this array object represents a one-dimensional, homogeneous array with elements that have fixed sizes.

9.1.2 Converting a Measurement Series into a Table

In real-life situations, the measurement values are often stored sequentially as a series in a file during the manufacturing process. For the statistical analysis, however, you often need a table with n rows and m columns. You can use the `reshape()` NumPy function to convert a one-dimensional array into a two-dimensional array. Listing 9.2 shows how to implement such a reshaping.

```
01  #02_reshape.py
02  import numpy as np
03  rows=5
04  columns=10
05  n=columns*rows
06  setpoint=10
07  s=1
```

```
08 values=np.random.normal(setpoint,s,size=n)
09 rvalues=np.around(values,decimals=2)
10 svalues=np.sort(rvalues)
11 table=np.reshape(svalues,(rows,columns),order='F')
12 print("Measurement values:\n",svalues)
13 print("Table:\n",  table)
14 print("Minimum value:", np.amin(rvalues))
15 print("Maximum value:", np.amax(rvalues))
```

Listing 9.2 Converting a Measurement Series into a Table

Output

```
Measurement values:
[8.23  8.5   8.73  8.83  8.93  8.98  9.13  9.2   9.3   9.52  9.52  9.6   9.63  9.71
9.72  9.74  9.82  9.86  9.87  9.91  9.91  9.92  9.94  9.98 10.03 10.04 10.05
10.05 10.2  10.36 10.46 10.55 10.57 10.72 10.74 10.75 10.76 10.78 10.83 10.86
10.96 11.01 11.11 11.18 11.21 11.26 11.35 11.49 12.27 12.49]
Table:
[[8.23  8.98  9.52  9.74  9.91 10.04 10.46 10.75 10.96 11.26]
[ 8.5   9.13  9.6   9.82  9.92 10.05 10.55 10.76 11.01 11.35]
[ 8.73  9.2   9.63  9.86  9.94 10.05 10.57 10.78 11.11 11.49]
[ 8.83  9.3   9.71  9.87  9.98 10.2  10.72 10.83 11.18 12.27]
[ 8.93  9.52  9.72  9.91 10.03 10.36 10.74 10.86 11.21 12.49]]
Minimum value: 8.23
Maximum value: 12.49
```

Analysis

The program generates 50 normally distributed random numbers (line 08). Line 10 sorts the numbers to improve the verification of the reshaping. In real-life applications, for example, in process monitoring via quality control charts, the real measurement data must not be sorted, of course. The statistical analysis of the individual table columns would provide a completely false picture of the manufacturing process.

In line 11, the `reshape()` NumPy function converts the one-dimensional array into a table with five rows and ten columns. The `svalues` array is passed as the first parameter. The second parameter contains the number of rows and columns as tuples. The third parameter (`order='F'`) determines how the table columns are formed from a section of a row. “F” means that the numbers are read or written in the Fortran-like index order, with the first index changing first and the last index changing last. The first column is formed from the first five numbers in the series. The next five numbers are used to form the second column of the measurement series, and so on.

In line 12, the sorted numbers are output as a one-dimensional array. Line 13 outputs these numbers as a table with five rows and ten columns.

9.1.3 Writing Measurement Values to a File

In reality, the measurement values determined in the manufacturing process are available as persistently stored data on hard disks. This data is used by SPC programs, so that corrective action can be taken in the process if necessary. For the analysis program to simulate process monitoring in a realistic way, the numbers generated by the random number generator must also be stored on the hard disk. For this purpose, the `savetext(param1,param2,param3)` NumPy function is used. Listing 9.3 stores the random numbers persistently on a hard disk.

```
01  #03_write.py
02  import numpy as np
03  n=50
04  setpoint=50
05  s=1
06  values=np.random.normal(setpoint,s,size=n)
07  rvalues=np.around(values,decimals=2)
08  np.savetxt("data.txt", values,fmt="%4.2f")
09  print("Normally distributed values:")
10  print(rvalues)
11  print("Type of values:",type(values))
```

Listing 9.3 Writing Data to a File

Output

```
Normally distributed values:
[51.33 49.76 50.17 49.4  49.4  47.78 49.9  47.99 50.35 48.83 50.22 48.79 48.7
50.48 50.65 49.87 49.7  50.42 49.34 49.51 50.17 51.3  50.17 50.14 49.44 49.3
48.27 49.99 50.4  49.13 50.03 51.08 50.03 51.72 49.42 49.9 49.43 49.03 49.91
50.43 50.35 49.48 49.25 50.48 49.17 49.68 52.23 50.44 49.73 50.9]
Type of values: <class 'numpy.ndarray'>
```

Analysis

In line 08, the random numbers generated in line 06 are saved to the hard disk using the `np.savetxt("data.txt",values,fmt="%4.2f")` NumPy function. The first parameter is the freely selectable file name with the `txt` file extension. The numbers are therefore stored in text format. They can be displayed and also edited with any text editor. The second parameter is the `values` variable, in which the 50 elements of the array (filled with random numbers) are stored. The third parameter specifies that the numbers should be stored with two decimal places. For control purposes, line 10 outputs the rounded values.

9.1.4 Reading Measurement Values from a File

To make the simulated measurement values available for statistical analysis, they must be read from the file in which they were previously stored. Listing 9.4 reads all numbers from the `data.txt` file using the `loadtext()` NumPy function.

```
01  #04_read.py
02  import numpy as np
03  values = np.loadtxt("data.txt")
04  n=len(values)
05  print("Loaded values:")
06  print(values)
07  print("Number of values:",n)
08  print("Type of values:",type(values))
```

Listing 9.4 Reading Data from a File

Output

```
Loaded values:
[51.33 49.76 50.17 49.4  49.4  47.78 49.9  47.99 50.35 48.83 50.22 48.79 48.7
50.48 50.65 49.87 49.7  50.42 49.34 49.51 50.17 51.3  50.17 50.14 49.44 49.3
48.27 49.99 50.4  49.13 50.03 51.08 50.03 51.72 49.42 49.9 49.43 49.03 49.91
50.43 50.35 49.48 49.25 50.48 49.17 49.68 52.23 50.44 49.73 50.9]
Number of values: 50
Type of values: <class 'numpy.ndarray'>
```

Analysis

In line 03, the random numbers are read from the *data.txt* file and stored in the `values` variable. The `np.loadtxt("data.txt")` function expects only one parameter, namely, the name of the file from which the data is to be read. For control purposes, line 06 outputs the simulated measurement series. A comparison with the output from Listing 9.4 shows that the values from both programs match as expected.

9.2 Frequency Distribution

A *frequency distribution* provides information about how the measured values in a measurement series are distributed between the smallest value and the largest value. Thus, process quality can be roughly estimated from the shape of the distribution: Is the mean value close to the nominal value, do the measured values scatter strongly, or are they unevenly distributed? Frequency distributions can be determined using frequency tables (tally sheets) or visualized graphically by means of histograms.

9.2.1 Frequency Tables

You can use a frequency table to determine how often a measurement value occurs within a certain interval. This interval is referred to as *class interval* w. To determine the class interval, the number of classes k must first be determined. Actually, the number of classes is freely selectable. In real life, however, accepted convention is to calculate k from the square root of an n number of samples:

$$k = \sqrt{n}$$

The result for k is rounded up to whole numbers.

The class interval w is calculated from the quotient of the span R

$$R = x_{\max} - x_{\min}$$

and the number of classes:

$$w = \frac{R}{k}$$

A frequency table is created using the following NumPy function:

```
H,I=np.histogram(array, bins=k)
```

Listing 9.5 shows how to implement a frequency table by using the NumPy module.

```python
#05_tally_sheet.py
import numpy as np
values=np.loadtxt("data.txt")
n=len(values)
k=int(np.sqrt(n)+0.5)
minimum=np.amin(values)
maximum=np.amax(values)
R=round(maximum-minimum,2)
w=round(R/k,2)
svalues=np.sort(values)
H,I=np.histogram(values, bins=k)
h=100*H/n
print("Measurement values:\n",svalues)
print("Minimum value:", minimum)
print("Maximum value:", maximum)
print("Span:",R)
print("Number of classes:", k)
print("Class interval:", w)
print("Ranges:", np.around(I,decimals=2))
print("Absolute frequency:", H)
print("Relative frequency:", h,"%")
print("Number of measurement values:", sum(H))
```

Listing 9.5 Frequency Table

Output

```
Measurement values:
[47.78 47.99 48.27 48.7  48.79 48.83 49.03 49.13 49.17 49.25 49.3  49.34 49.4
49.4  49.42 49.43 49.44 49.48 49.51 49.68 49.7  49.73 49.76 49.87 49.9  49.9
49.91 49.99 50.03 50.03 50.14 50.17 50.17 50.17 50.22 50.35 50.35 50.4  50.42
50.43 50.44 50.48 50.48 50.65 50.9  51.08 51.3  51.33 51.72 52.23]
Minimum value: 47.78
Maximum value: 52.23
Span: 4.45
Number of classes: 7
Class interval: 0.64
Ranges: [47.78 48.42 49.05 49.69 50.32 50.96 51.59 52.23]
Absolute frequency: [ 3  4 13 15 10  3  2]
Relative frequency: [ 6.  8. 26. 30. 20.  6.  4.] %
Number of measurement values: 50
```

Analysis

The program loads the measurement values from the `data.txt` file (line 03), determines the length of the `values` array (line 04), and computes the number of classes `k` (line 05). The minimum and maximum are determined using NumPy functions `np.amin(values)` (line 06) and `np.amax(values)` (line 07), respectively. From the span `R` (line 08) and the number of classes `k`, the program computes the class interval `w` (line 09).

In line 11, the `np.histogramm(param1, param2)` NumPy function computes the absolute frequency H and the intervals I of the individual classes and assigns them to the `H,I` variables as tuples. This function expects two parameters: The `values` array is passed as the first parameter; the second parameter `bins=k` expects the number of classes.

Line 20 provides the absolute frequency, and line 21 gives the relative frequency. For control purposes, the sum of the absolute frequencies is output in line 22.

9.2.2 Histograms

Histograms visualize frequency tables as bar charts. The class interval corresponds to the width of a rectangle and the absolute frequency corresponds to the height of a rectangle. The number of rectangles corresponds to the number of classes. To display histograms, the `matplotlib.pyplot` module must first be imported. The `hist(param1, param2,param3,param4,...)` method displays a histogram as a bar chart. Listing 9.6 shows how to visualize the frequency table created earlier in Listing 9.5.

```
01  #06_histogram.py
02  import numpy as np
03  import matplotlib.pyplot as plt
04  values = np.loadtxt("data.txt")
```

```
05  n=len(values)
06  k=int(np.sqrt(n)+0.5)
07  fig, ax=plt.subplots()
08  ax.hist(values,bins=k,edgecolor="b",color="w")
09  ax.set(xlabel="Measurement values",ylabel="Absolute frequency")
10  plt.show()
```

Listing 9.6 Histogram

Output

The resulting histogram output from running the program is shown in Figure 9.2.

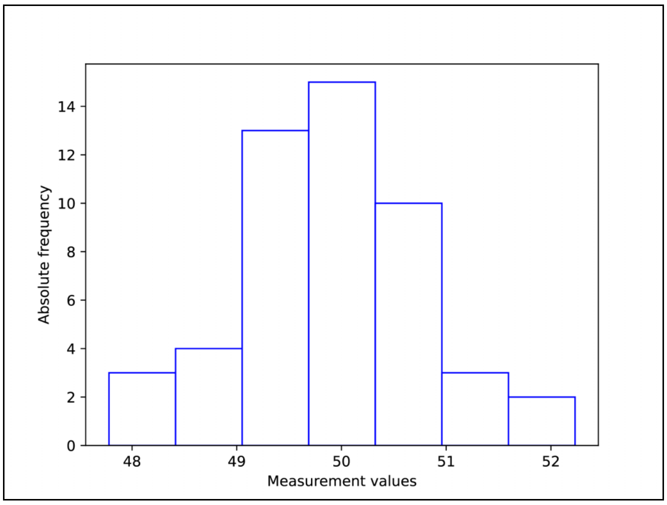


Figure 9.2 Histogram

Analysis

For the display of the histogram via the `hist(values,bins=k,edgecolor="b",color="w")` method, only four parameters are used (line 08). The first parameter is the `values` array. The second parameter (`bins=k`) specifies the number of classes. The third parameter (`edgecolor="b"`) sets the border color of the bars to blue. The fourth parameter (`color="w"`) specifies that white is the background color of the rectangles.

Line 09 defines the axis label. Line 10 is necessary so that the graphic is also output on the screen.

9.3 Location Parameters

In statistics, a *location parameter* is a measure that describes the clustering of values near a central value. Statistics uses the arithmetic, harmonic, and geometric mean values as well as the median and mode as location parameters.

9.3.1 Arithmetic Mean

The *arithmetic mean* $\bar{x}$ of a measurement series is the sum of the individual sample values divided by the number n of these values. The following equation therefore applies:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Both Python and the NumPy module have statistical functions that compute the arithmetic mean. Listing 9.7 calculates means using a custom function named `mean()`, the Python function `statistics.mean()`, and the NumPy function `numpy.mean()`. This program also compares the individual runtimes of these functions to find the most effective method.

```python
#07_mean_value.py
import numpy as np
import statistics as st
import time as t
n=100000
setpoint=50
s=1

def mean(values):
    n=len(values)
    sum=0
    for i in range(n):
        sum=sum+values[i]
    return sum/n

#values=[1,2,3,4,5,6]
values=np.random.normal(setpoint,s,size=n)
t1=t.time()
m1=mean(values)
t2=t.time()
m2=st.mean(values)
t3=t.time()
m3=np.mean(values)
t4=t.time()
print("\t\tArithmetic mean","Time",sep=3*("\t"))
print("Custom version:",m1,t2-t1,sep=("\t"))
print("Python version:",m2,t3-t2,sep=("\t"))
print("NumPy version:",m3,t4-t3,sep=("\t"))
```

Listing 9.7 Runtimes for Calculating Arithmetic Mean Values

Output

```
                  Arithmetic mean          Time
Custom version: 50.00002592181933   0.019836902618408203
Python version: 50.00002592181897   0.15338611602783203
NumPy version:  50.00002592181897   0.00013494491577148438
```

Analysis

The Python modules `statistics` and `time` are imported in lines 03 and 04. Line 05 specifies a particularly high number of `n=100000` random numbers so that meaningful runtimes can also be determined.

The `mean()` function in lines 09 to 14 computes the sum of all numbers and returns the arithmetic mean `sum/n`.

The time measurement is performed for each function call according to the same schema: Before a function is called, the system time `t1` is determined. After returning the value for the arithmetic mean, the current system time is stored in the `t2` variable. The runtime is then determined from the difference of `t2-t1`.

An interesting thing to note is that the custom `mean()` function is about 7.5 times faster than the Python function `st.mean()`. However, the custom function also provides a somewhat less accurate value. The NumPy function `np.mean()` is about 1,100 times faster than the Python function `st.mean()`. Thus, for future programs, we recommend using the NumPy function.

9.3.2 Mode, Median, Harmonic Mean, and Geometric Mean

The mode or *modal value* indicates which value of a measurement series occurs most frequently. The mode corresponds to the maximum of a frequency distribution.

The *median* (or central value) indicates the mean value of a sorted measurement series if the number of measurement values is odd. If the number of measurement values is even, the median is formed from the mean value of the two central measurement values.

To calculate the harmonic mean, the reciprocals of the individual measurement values must first be added up. The reciprocal value is then formed from this sum and then multiplied by the number of measurement values, as in the following:

$$\bar{x}_{\text{harmonic}} = n\left(\sum_{i=1}^{n} \frac{1}{x_i}\right)^{-1}$$

To calculate the geometric mean, all the individual measurement values of a sample must be multiplied with each other. Then, the nth root is taken from this product:

$$\bar{x}_{\text{geometric}} = \sqrt[n]{x_1 \cdot x_2 \, \cdots \, x_n}$$

Listing 9.8 calculates all four location values using NumPy and SciPy functions. The measurement values are read from the data.txt file.

```
01  #08_location_parameters.py
02  import numpy as np
03  import scipy.stats as sta
04  values=np.loadtxt("data.txt")
05  mw=np.mean(values)
06  md=np.median(values)
07  mode=sta.mode(values)
08  hm=sta.hmean(values)
09  gm=sta.gmean(values)
10  print(np.sort(values))
11  print("Mode:",mode)
12  print("Arithmetic mean:",mw)
13  print("Median:         ",md)
14  print("Harmonic mean:  ",hm)
15  print("Geometric mean: ",gm)
```

Listing 9.8 Mode, Median, and Mean Values

Output

```
[47.78 47.99 48.27 48.7  48.79 48.83 49.03 49.13 49.17 49.25 49.3  49.34 49.4
49.4  49.42 49.43 49.44 49.48 49.51 49.68 49.7  49.73 49.76 49.87 49.9  49.9
49.91 49.99 50.03 50.03 50.14 50.17 50.17 50.17 50.22 50.35 50.35 50.4  50.42
50.43 50.44 50.48 50.48 50.65 50.9  51.08 51.3  51.33 51.72 52.23]
Mode: ModeResult(mode=array([50.17]), count=array([3]))
Arithmetic mean: 49.8718
Median:          49.9
Harmonic mean:   49.85678962270494
Geometric mean:  49.864292035952936
```

Analysis

Line 03 imports the statistics module scipy.stats from SciPy. You can use the sta alias to access the statistics functions of this module. The arithmetic mean and the median are calculated using the NumPy functions np.mean() and np.median() in lines 05 and 06. The mode, harmonic mean, and geometric mean are computed using the SciPy functions sta.mode(), sta.hmean(), and sta.gmean() in lines 07 to 09. The return value of the sta.mode() function contains a tuple of two values. The first value indicates the mode; the second value indicates how often the mode occurs (line 11).

9.4 Dispersion Parameters

Dispersion parameters describe the distance of individual measurement values from their mean value. To ensure process quality, the dispersion of the population and individual samples should be as low as possible. Statistics uses two measures to quantify dispersion: the span and the standard deviation.

The *span* R is simply calculated from the difference between the largest and the smallest measurement value of a series of measurements, as in the following:

$$R = x_{\max} - x_{\min}$$

While this measure is easy to calculate, for more detailed statistical studies, this approach is not recommended because it is quite sensitive to outliers.

More precisely, the *standard deviation* s describes the dispersion of the measurement values around a mean value. This value is calculated by taking the difference between each individual measurement value and the arithmetic mean of the measurement series. This difference is squared to give negative differences a positive sign and then totaled. The totaled squares of the differences must then be divided by $n - 1$. The result obtained in this way is referred to as *variance*. If you calculate the square root of the variance, you obtain the standard deviation, as shown in the following equation:

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

Listing 9.9 computes the standard deviation from 100,000 normally distributed random numbers using the three functions: a custom function named `stdaw()`, Python's `statistics.std()` function, and the NumPy function `numpy.std()`. To determine the most powerful function, the runtimes of these three functions are compared with each other.

```python
#09_stdaw.py
import numpy as np
import statistics as st
import time as t
n=100000
setpoint=100
s=2
def stdaw(values):
    n=len(values)
    sum=0
    for i in range(n):
        sum=sum+values[i]
        mean=sum/n
    sum_rq=0
```

```
15      for i in range(n):
16          sum_rq=sum_rq+(values[i]-mean)**2
17      v=sum_rq/(n-1) #variance
18      return np.sqrt(v)
19
20  #values=[1,2,3,4,5,6]
21  values=np.random.normal(setpoint,s,size=n)
22  t1=t.time()
23  s1=stdaw(values)
24  t2=t.time()
25  s2=st.stdev(values)
26  t3=t.time()
27  s3=np.std(values,ddof=1)
28  t4=t.time()
29  print("\t\t  Standard deviation","Time",sep=2*("\t"))
30  print("Custom version  :",s1,t2-t1)
31  print("Python version  :",s2,t3-t2)
32  print("NumPy version   :",s3,t4-t3)
```

Listing 9.9 Computing the Standard Deviation

Output

```
                Standard deviation        Time
Custom version: 1.9953201231531883 0.07551097869873047
Python version: 1.9953201231531907 0.4775989055633545
NumPy version : 1.9953201231531905 0.0005953311920166016
```

Analysis

The custom function `staw()` consistently implements the algorithm (lines 08 to 18). In line 17, the variance is calculated. To ensure that the `staw()` function also returns the standard deviation, the square root of the variance must be determined, which happens in line 18.

The commented-out line 20 is supposed to provide test values. All three functions output the expected value of $s = 1.8708$.

The additional `ddof=1` parameter of the `std()` NumPy function in line 27 ensures that the total of the squares from the differences (`values[i]-mean`) is not divided by n, as given by default, but by $n - 1$. The `ddof` acronym stands for *Delta Degrees of Freedom*.

Surprisingly, the custom `staw()` function is about six times faster than Python's `stdev()` function. As expected, the NumPy version (`np.std()`) is much faster than the other two versions, about 800 times faster than the Python version.

Usage Example: Checking the Machine Capability

Before a complete series of workpieces can be manufactured, we need to check whether the machine delivers the desired process quality at all through a *machine capability study*. The term *machine capability* is understood by quality management to refer to the ability of a machine to produce defect-free workpieces under consistent conditions. For this purpose, the arithmetic mean and the standard deviation must be determined from a sample of at least 50 measurement values. Only if the manufacturing variation is within 99.73% of all workpieces (equivalent to 6s) and the required tolerance is greater than or equal to ten times the standard deviation (equivalent to 10s) is the machine capability criterion met. From these criteria, that the manufacturing scatter must be $\leq 6s$ and that the required tolerance must be $T \geq 10s$, the machine capability index can be defined in the following way:

$$C_m = \frac{T}{6s} \geq 1.67$$

The machine capability index C_m describes only the influence of the dispersion of measurement values on the manufacturing process. The distance of the sample mean from the tolerance center is not taken into account. This criterion, that is, the distance of the mean value to the tolerance center, is described by the machine capability characteristic value C_{mk} in the following equation:

$$C_{mk} = \frac{\Delta_{\text{krit}}}{3s} \geq 1.67$$

The critical distance Δ_{krit} is the smallest distance of the arithmetic mean value $\bar{x}$ of the measurement series to the tolerance limit. The tolerance limit can be near the lower limit or near the upper limit of the measured values, depending on the location of $\bar{x}$. Based on this requirement, it always follows that the machine capability index C_{mk} must be smaller than the machine capability index C_m.

Listing 9.10 computes the machine capability index and the machine capability characteristic value from a sample of 50 measurement values. The simulated measurement values are again read from the `data.txt` file using the `np.loadtext()` NumPy function. The tolerance limits are specified, and the arithmetic mean and the standard deviation are calculated by the program from the measurement series of the sample. By varying the tolerance limits, you can simulate whether the machine meets the process quality requirements.

```
#10_mcapability.py
import numpy as np
setpoint=50
To=5
Tu=-5
T=To-Tu
values = np.loadtxt("data.txt")
```

```
08  m=np.mean(values)
09  s=np.std(values,ddof=1)
10  Cm=T/(6*s)
11  UCL=setpoint+To #Upper control limit
12  LCL=setpoint+Tu #Lower control limit
13  delta_o=UCL-m
14  delta_u=m-LCL
15  if delta_o > delta_u:
16      delta_k=delta_u
17  else:
18      delta_k=delta_o
19  Cmk=delta_k/(3*s)
20  print("Mean:",m)
21  print("Standard deviation:",s)
22  print("Machine capability index: ",Cm)
23  print("Machine capability characteristic value:",Cmk)
```

Listing 9.10 Checking the Machine Capability

Output

```
Mean: 49.8718
Standard deviation: 0.8745510568402549
Machine capability index:   1.905739697677936
Machine capability characteristic value: 1.8568765318294738
```

Analysis

The machine capability index and the machine capability characteristic value are greater than 1.67. The machine is therefore able to maintain the required process quality. However, this check passes due to the high tolerances of ± 10% (lines 04 and 05). If the tolerances were reduced, then it would become apparent that the machine cannot maintain the required process quality.

In line 10, the machine capability index `Cm` is calculated from the tolerance (line 06) and the standard deviation (line 09). The program calculates the machine capability parameter `Cmk` in lines 11 to 19. For this purpose, the upper limit value `UCL` (line 11) and the lower limit value `LCL` (line 12) must be determined. The `delta_o` (line 13) and `delta_u` (line 14) deviations are needed to compute the smallest distance from the arithmetic mean `delta_k`. This computation step is performed in lines 15 to 18 by an `if-else` query. The machine capability parameter is then calculated in line 19.

The output of the machine parameters is performed in lines 22 and 23. The arithmetic mean and the standard deviation are output in lines 20 and 21 to show the relationship between the statistical parameters and machine parameters: The greater the standard deviation and the greater the distance of the setpoint from the arithmetic mean of the measurement values, the worse the machine capability will be rated.

9.5 Normal Distribution

If we were to considerably increase the number of measurements, perhaps even theoretically to infinity, the frequency distribution of a histogram would approach the density function of the *normal distribution* (also called the Gaussian distribution). The density function $g(x)$ is an e-function with parameters μ and σ :

$$g(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

The μ parameter is referred to as the *expected value*. This value coincides with the arithmetic mean, the mode, and the median when the number of measurement values of a normally distributed measurement series approaches infinity. The σ^2 parameter is referred to as the *variance*. The square root of the variance corresponds to the standard deviation of a measurement series with a very large number of measurement values.

9.5.1 Graphical Representation of the Density Function

If we use $\mu = 0$ for the mean and $\sigma = 1$ for the standard deviation, then we speak of a *standard normal distribution*. Listing 9.11 represents the density of the standard normal distribution within the limits from -3σ to 3σ.

```
#11_gauss.py
import numpy as np
import matplotlib.pyplot as plt
#density function
def g(x,sigma,my):
    y=np.exp(-0.5*(x-my)**2/sigma**2)/(sigma*np.sqrt(2*np.pi))
    return y

s=1
m=0
x = np.arange(m-3*s, m+3*s, 0.01);
y = g(x,s,m)
fig, ax=plt.subplots()
ax.plot(x, y)
ax.plot(-1, g(-s,s,m),"ro")
ax.plot( 1, g( s,s,m),"ro")
ax.set(xlabel="x",ylabel="g(x)")
plt.show()
```

Listing 9.11 Density of the Standard Normal Distribution

Output

Figure 9.3 shows the output density of a standard normal distribution as a curve.

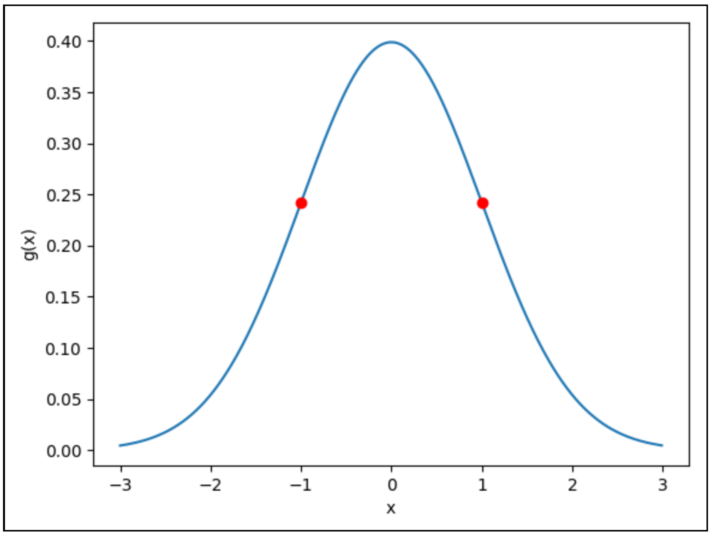


Figure 9.3 Standard Normal Distribution

Analysis

In lines 05 to 07, the function of the normal distribution is defined. The `g(x,sigma,my)` function must be passed the standard deviation and the arithmetic mean as arguments.

Lines 09 and 10 contain the standard deviation `s=1` and the mean `m=0`. These parameters specify that a normalized density function is represented.

Line 11 creates an array with the initial value `m-3*s` and the final value `m+3*s`. The third parameter of the `arange()` NumPy function sets the increment to `0.01`.

In line 12, the density function is called with the `x`, `s`, and `m` parameters. The function values are assigned to the `y` variable. This variable is also an array in which all function values of the `g(x,s,m)` function have been stored.

9.5.2 Probability Distribution

In real life, often, you need to know the probability that a certain dimension of a workpiece from a series occurs within a given range within a normal distribution. For this purpose, the area under the Gaussian curve must be determined within the selected limits using a numerical integration method.

With its `quad()` function, the `scipy.integrate` module provides a powerful method for numerical integration. The name *quad* comes from the integration method of the QUADPACK Fortran library.

Listing 9.12 calculates the probabilities within the bounds $\pm\sigma$, $\pm2\sigma$, and $\pm3\sigma$. The $\sigma = 1$ and $\mu = 0$ parameters specify that the program represents a standard normal distribution.

```
01  #12_probability.py
02  import numpy as np
03  from scipy.integrate import quad
04  def f(x,sigma,my):
05      y=np.exp(-0.5*(x-my)**2/sigma**2)/(sigma*np.sqrt(2*np.pi))
06      return y
07
08  s=1
09  m=0
10  w1=quad(f,  -s,  s,args=(s,m))
11  w2=quad(f,-2*s,2*s,args=(s,m))
12  w3=quad(f,-3*s,3*s,args=(s,m))
13  wp1,wp2,wp3=100*w1[0],100*w2[0],100*w3[0]
14  print("Expected value between  -\u03C3 and  +\u03C3: %2.2f%%"%wp1)
15  print("Expected value between -2\u03C3 and +2\u03C3: %2.2f%%"%wp2)
16  print("Expected value between -3\u03C3 and +3\u03C3: %2.2f%%"%wp3)
```

Listing 9.12 Probability Ranges

Output

```
Expected value between -σ and +σ: 68.27%
Expected value between -2σ and +2σ: 95.45%
Expected value between -3σ and +3σ: 99.73%
```

Analysis

The statement in line 03 imports the `scipy` module with the `integrate` package and the `quad()` function. This function, which is called in lines 10 to 12, expects three parameters. The name of the function is passed as the first parameter without parentheses. The second and third parameters set the lower and upper integration limits. The third parameter consists of a tuple (`args=(s,m)`) with the function parameters of the normal distribution, namely, standard deviation *s* and mean *m*.

The `quad()` SciPy function returns two values as a tuple. The first value is the computed area of the numerical integration. The second return value specifies an error estimate.

Since only the first return value is of interest in our case, only this values `w1[0]`, `w2[0]`, and `w3[0]` are assigned to the `wp1`, `wp2`, and `wp3` variables, respectively, in line 13. The computed areas are multiplied by a factor of 100 so that the output of the probabilities is represented in percentages.

The output in lines 14 to 16 uses the Unicode character U+03C3 for the formula character of the standard deviation. The 99.73% expected value between -3σ and $+3\sigma$ is of particular interest: This range specifies the limits of process quality. The determined measurement values should fall within in this interval; conversely, this value also means that only 0.27% scrap is allowed.

9.6 Skew

For SPC, you must know whether a measurement series is normally distributed, more or less. This criterion can be checked with the statistical measure of *skewness*. Skew is a statistical measure between –1 and +1 that describes whether and how strongly a frequency distribution is *skewed* to the right or to the left. If the frequency distribution is skewed to the right, it is called *right skewed*. The dimension number S is then negative. If the frequency distribution is skewed to the left, it is called *left skewed*. The dimension number S is then positive. The smaller S is, the more symmetrical the frequency distribution is. For the normal distribution, S = 0. Karl Pearson (1895–1980) gave a simple rule of thumb to describe the deviation of a frequency distribution from symmetry:

$$S = \frac{\bar{x} - \tilde{x}}{s}$$

If the median $\tilde{x}$ turns out to be smaller than the mean $\bar{x}$, then S is positive, and the distribution is left skewed. If the median $\tilde{x}$ is larger than the mean $\bar{x}$, then S is negative, and the distribution is right skewed.

For the left-skewed distribution, the following often holds true: *mode* < *median* < *mean*. The reverse is true for the right-skewed distribution: *mean* < *median* < *mode*.

However, these rules of thumb do not always determine the correct value of the skew. The stats package from the scipy module calculates the correct value of skew using the stats.skew(array) function.

Listing 9.13 computes the approximate value and the exact value of skewness from 50 random numbers.

```python
#13_skew.py
import numpy as np
from scipy import stats
n=50
setpoint=50
s=2
values=np.random.normal(setpoint,s,size=n)
rvalues=np.around(values,decimals=2)
mode=stats.mode(rvalues, axis=None)
mw=np.mean(rvalues)
```

```
11  md=np.median(rvalues)
12  stabw=np.std(rvalues,ddof=1)
13  S1=(mw-md)/stabw
14  S2=stats.skew(rvalues)
15  print(np.sort(rvalues))
16  print("Mode:        ",mode)
17  print("Mean:        ",mw)
18  print("Median:      ",md)
19  print("Standard deviation:",stabw)
20  print("Skew (approximation):",S1)
21  print("Skew (exact):   ",S2)
22  if S2<0:
23      print("right-skewed")
24  else:
25      print("left-skewed")
```

Listing 9.13 Computation of the Skew

Output

```
[45.41 46.2  46.39 46.5  47.14 47.24 47.25 47.38 47.59 47.73 48.04 48.21 48.3
48.33 48.56 48.76 49.31 49.4  49.45 49.73 49.75 49.78 49.89 49.91 49.93 49.95
49.98 50.05 50.15 50.17 50.34 50.42 50.43 50.74 50.89 50.94 51.02 51.09 51.14
51.32 51.58 51.92 51.97 52.17 52.19 52.37 52.64 53.34 53.63 54.01]
Mode: ModeResult(mode=array([45.41]), count=array([1]))
Mean:         49.81260000000001
Median:       49.94
Standard deviation: 2.0127043539294416
Skew (approximation): -0.06329792040806283
Skew (exact):         -0.09607672698739309
right-skewed
```

Analysis

Note that different values for skewness are output after each program start because the program also generates different random numbers in each case. In line 13, the `S1` skew is calculated according to the simple Pearson formula. The exact calculation of the `S2` skew is performed in line 14 using SciPy function `stats.skew(rvalues)`. By starting the program multiple times, you can determine that the calculation according to Pearson does not always match the result of the `stats.skew()` SciPy function. In lines 22 to 25, the sign of the `S2` skew is analyzed. If `S2<0`, `right-skewed` is output, and if `S2>0`, `left-skewed` is output.

9.7 Regression Analysis

A *regression analysis* is a statistical analysis method that examines the relationships between two variables: a dependent variable Y (also called the *outcome variable*) and an independent variable X (also called the *influencing variable*). Regression analyses are always used when correlations need to be described or predicted in terms of quantities. Mathematically, the influence of X on Y is symbolized by an arrow:

$X \rightarrow Y$

I want to use a simple example to describe the regression analysis method: In a storage room for synthetic fibers, there is a certain relative humidity. Over a period of 15 days, the relative humidity of the room (X) and the moisture content of the synthetic fiber (Y) are measured once a day. A regression analysis will be used to investigate whether there is a correlation between these two variables and, if so, how strong this correlation is. In Table 9.2, the measurement values are documented.

T	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
X	46	53	29	61	36	39	47	49	52	38	55	32	57	54	44
Y	12	15	7	17	10	11	11	12	14	9	16	8	18	14	12

Table 9.2 Measurement Values for the Relative Humidity X and Moisture Content of Material Y in Percent

In the simplest case, a linear relationship exists between the X and Y values, which can be described by a linear function with the slope m and the intercept a of the function line with the y axis:

$y = mx + a$

The m and a parameters are also referred to as regression parameters. The strength of the correlation between X and Y of the two measurement series is determined by the correlation coefficient r.

9.7.1 Computing the Regression Parameters

The correlation coefficient r is defined as the quotient of the covariance s_{xy} of two measurement series and the product of the standard deviations, $s_x s_y$:

$$r = \frac{s_{xy}}{s_x s_y}$$

The slope m of the regression line is the quotient of the covariance and the square of the standard deviation of the x-values:

$$m = \frac{s_{xy}}{s_x^2}$$

The slope can also be calculated using the correlation coefficient:

$$m = r\frac{s_y}{s_x}$$

The intercept a of the regression line is computed from the difference of the mean value $\bar{y}$ of all y-values and the product of the slope m with the mean value $\bar{x}$ of all x-values:

$$a = \bar{y} - m\bar{x}$$

All regression parameters can be computed directly using the following SciPy function:

```
m,a,r,p,e = stats.linregress(X,Y)
```

The return values p and e of the tuple are not needed for computing the regression parameters.

Listing 9.14 calculates the slope m, the y-axis intercept a of the regression line and the correlation coefficient r for the measurement values listed in Table 9.2. To show the different options of Python, the parameters are calculated and compared by using the corresponding NumPy and SciPy functions.

```
#14_correlation.py
import numpy as np
from scipy import stats
X=np.array([46,53,29,61,36,39,47,49,52,38,55,32,57,54,44])
Y=np.array([12,15,7,17,10,11,11,12,14,9,16,8,18,14,12])
xm=np.mean(X)
ym=np.mean(Y)
sx=np.std(X,ddof=1)
sy=np.std(Y,ddof=1)
sxy=np.cov(X,Y)
r1=sxy/(sx*sy)
r2=np.corrcoef(X,Y)
m1=sxy[0,1]/sx**2
m2=r2[0,1]*sy/sx
a1=ym-m2*xm
m3, a2, r3, p, e = stats.linregress(X,Y)
print("NumPy1 slope:",m1)
print("NumPy2 slope:",m2)
print("SciPy slope:",m3)
print("Intersection with the y-axis:",a1)
print("Intersection with the y-axis:",a2)
print("Def.  correlation coefficient:",r1[0,1])
print("NumPy correlation coefficient:",r2[0,1])
```

```
24  print("SciPy correlation coefficient:",r3)
25  print("Estimated error:",e)
```

Listing 9.14 Computing the Regression Parameters

Output

```
NumPy1 slope: 0.32320356181404014
NumPy2 slope: 0.3232035618140402
SciPy slope: 0.3232035618140402
Intersection with the y-axis: -2.5104576516877213
Intersection with the y-axis: -2.5104576516877213
Def  correlation coefficient: 0.9546538498757964
NumPy correlation coefficient: 0.9546538498757965
SciPy correlation coefficient: 0.9546538498757965
Estimated error: 0.027955268902524828
```

Analysis

Lines 04 and 05 contain the measurement values for the relative humidity `X` and the moisture content of the material `Y`. The program computes the regression parameters from these measurement values and uses the correlation coefficient to check whether a correlation exists between the influencing variable `X` and the outcome variable `Y` and to determine how strong this correlation is.

Line 10 computes the covariance `sxy` using the `np.cov(X,Y)` NumPy function. This function returns a 2 × 2 matrix. The value for `sxy` is either in the first row, second column, of the matrix or in the second row, first column, of the matrix. The correlation coefficient `r1` is then calculated with `sxy` in line 11.

A simpler way to calculate the correlation coefficient is to directly use NumPy function `np.corrcoef(X,Y)` (line 12). This function also returns a 2 × 2 matrix. The value for `r2` is either in the first row, second column, of the matrix or in the second row, first column of the matrix.

Line 13 calculates the slope `m1` from the covariance `sxy[0,1]` and the square of the standard deviation `sx` from the `X` measurement values. The slope `m2` is calculated in line 14 using the correlation coefficient `r2(0,1)` and the standard deviations `sx` and `sy`.

In line 15, the y-axis intercept `a1` is calculated from the mean values of the `X` and `Y` values using the conventional method. Instead of the slope `m2`, the slope `m1` could have been used as well.

The most effective method to calculate all three parameters with only one statement is shown in line 16. The slope `m3`, the y-axis intercept `a2`, and the correlation coefficient `r3` are returned as tuples by the SciPy function `stats.linregress(X,Y)`.

The `print()` function outputs the regression parameters in lines 17 to 25. All computation methods provide the same results. The slope is about 0.32, and the y-axis intercept has a value of about −2.51. Thus, the regression line adheres to the following equation:

$$y = 0.32x - 2.51$$

The correlation coefficient of `r = 0.95465` is close to 1. Thus, a strong correlation exists between the relative humidity (`X`) and the moisture content of the material (`Y`).

9.7.2 Representing the Scatter Plot and the Regression Line

When the discrete y_i values of the Y measurement series and the discrete x_i values of the X measurement series are plotted in an x-y coordinate system, this plot is referred to as a *scatter plot*. Listing 9.15 shows how to implement such a scatter plot with the values listed in Table 9.2 and the corresponding regression line.

```
01  #15_regeression_line.py
02  import numpy as np
03  import matplotlib.pyplot as plt
04  from scipy import stats
05  X=np.array([46,53,29,61,36,39,47,49,52,38,55,32,57,54,44])
06  Y=np.array([12,15,7,17,10,11,11,12,14,9,16,8,18,14,12])
07  m, a, r, p, e = stats.linregress(X,Y)
08  fig, ax=plt.subplots()
09  ax.plot(X, Y,'rx')
10  ax.plot(X, m*X+a)
11  ax.set_xlabel("Relative humidity in %")
12  ax.set_ylabel("Moisture content of the material")
13  plt.show()
```

Listing 9.15 Scatter Plot with Regression Line

Output

Figure 9.4 shows the output regression line after the calculation performed in Listing 9.15.

Analysis

The program determines the y-axis intercept `a` and the slope `m` in line 07 using the `stats.linregress(X,Y)` SciPy function. Line 09 plots the discrete x_i and y_i values as red crosses using the `plot(X, Y, 'rx')` method. The `ax.plot(X, m*X+a)` statement in line 10 causes the plot of the regression line. The crosses of the scatter plot clearly show that a strong correlation exists between the relative humidity and the moisture content of the material.

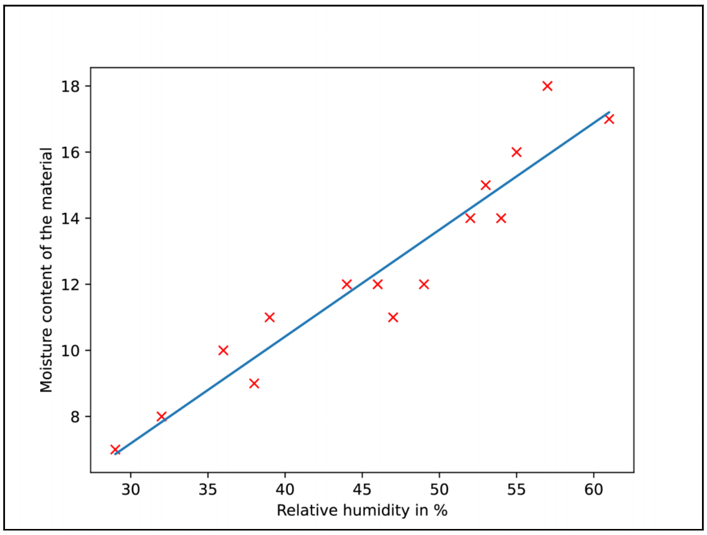


Figure 9.4 Regression Line

9.8 Project Task: Simulation of a Quality Control Chart

Quality control charts are used in manufacturing engineering to monitor process quality. The basic idea is to intervene in the process before rejects are produced. For this purpose, a random sample of $n = 5$ workpieces is taken from the running production every hour. Measuring machines record the quality-relevant variables and store the measured data in a file. Computer programs then calculate the arithmetic mean and standard deviation of the individual samples from these measurement results and display these quantities as a polyline in two graphic windows in each case. The upper graphic window represents the course of the mean value and the window below it represents the course of the standard deviation from the five measurement values. For both graphs, the program still needs to calculate upper and lower action limits. These intervention limits are calculated from the mean values of the means $\bar{\bar{x}}$ and the mean values of the standard deviations $\bar{s}$ of all samples.

The following equation applies to the upper control limit of the mean value chart:

$$UCL_{\bar{x}} = \bar{\bar{x}} + A_3\bar{s}$$

The following equation applies to the lower control limit of the mean value chart:

$$LCL_{\bar{x}} = \bar{\bar{x}} - A_3\bar{s}$$

The following equation applies to the upper control limit of the standard deviation chart:

$$UCL_s = B_4\bar{s}$$

The values of factors A_3 and B_4 can be found in relevant tables. Thus, for a sample of $n = 5$, we find for $A_3 = 1.152$ and for $B_4 = 1.669$.

The required process quality is not met if the following conditions are true:

- The lower or upper intervention limit is exceeded.
- Seven consecutive values are above or below the centerline (*run*).
- Seven consecutive values in an interval are ascending or descending (*trend*).
- More than 90% of the values lie within the middle third of the intervention limits (*middle third*).

Listing 9.16 computes the arithmetic mean, standard deviation, and intervention limits for each sample from ten samples with five measurement values each. The measurement data is read sequentially from a file and converted into a table with five rows and ten columns using NumPy function `reshape()`.

```
#16_qrt_table.py
import numpy as np
rows=5
columns=10
A3=1.152
B4=1.669
values = np.loadtxt("data.txt")
n=len(values)
table=np.reshape(values,(rows,columns),order='F')
mw=[]
staw=[]
for i in range(columns):
    sum1=0
    sum2=0
    for j in range(rows):
        sum1=sum1+table[j,i]
        mean=sum1/rows
    for j in range(rows):
        sum2=sum2+(mean-table[j,i])**2
        standardabw=np.sqrt(sum2/(rows-1))
    mw.append(round(mean,2))
    staw.append(round(standardabw,3))
mmw=np.mean(mw)
mws=np.mean(staw)
UCLm=mmw + A3*mws #Upper control limit
LCLm=mmw - A3*mws #Lower control limit
UCLs=B4*mws
print("Table of measurement values:\n",table)
print("Mean value of the samples:")
```

```
30  print(mw)
31  print("Standard deviation of the samples:")
32  print(staw)
33  print("Mean value chart")
34  print("Upper intervention limit:",UCLm)
35  print("Lower intervention limit:",LCLm)
36  print("Standard deviation chart")
37  print("Upper intervention limit:",UCLs)
```

Listing 9.16 Computing Intervention Limits

Output

```
Table of measurement values:
 [[51.33 47.78 50.22 49.87 50.17 49.3  50.03 49.9  50.35 49.68]
 [49.76 49.9  48.79 49.7  51.3  48.27 51.08 49.43 49.48 52.23]
 [50.17 47.99 48.7  50.42 50.17 49.99 50.03 49.03 49.25 50.44]
 [49.4  50.35 50.48 49.34 50.14 50.4  51.72 49.91 50.48 49.73]
 [49.4  48.83 50.65 49.51 49.44 49.13 49.42 50.43 49.17 50.9 ]]
Mean value of the samples
[50.01, 48.97, 49.77, 49.77, 50.24, 49.42, 50.46, 49.74, 49.75, 50.6]
Standard deviation of the samples:
[0.802, 1.136, 0.947, 0.415, 0.668, 0.823, 0.925, 0.532, 0.623, 1.046]
Mean value chart
Upper intervention limit : 50.785038400000005
Lower intervention limit: 48.960961600000005
Standard deviation chart
Upper intervention limit: 1.3213473000000002
```

Analysis

In lines 10 and 11, two empty `mw[]` and `staw[]` lists are defined for the means and standard deviations of the ten samples. These lists are passed as parameters to the `np.mean()` NumPy function in lines 23 and 24.

The `mw` means and standard deviations `staw` of each sample are calculated in lines 15 to 22. The sum algorithms and the nested loop construct used in this case are elaborate and, as will be shown in the next example, actually unnecessary because they do not exhaust the capabilities of Python.

The statements in lines 25 to 27 compute the upper and lower intervention limits according to the specifications with factors `A3` and `B4`. For samples with $n = 5$, the calculation of the lower intervention limits for the standard deviation is omitted.

The outputs are created in lines 28 through 37. The table of the 50 measurement values is displayed so that you can check whether the program correctly calculates the statistical characteristics of the individual samples. Each column of the table can be assigned the corresponding value for the mean and standard deviation.

Checking process quality on the basis of numerical values is too cumbersome in operational practice. For this reason, the courses of the mean values and standard deviations of the individual samples are visualized graphically as polylines. Listing 9.17 shows how such a graphic program can be implemented with the resources of Python (slicing).

```python
#17_qrt_graphics.py
import numpy as np
import matplotlib.pyplot as plt
rows=5
columns=10
A3=1.152
B4=1.669
values = np.loadtxt("data.txt")
n=len(values)
table=np.reshape(values,(rows,columns),order='F')
h=np.linspace(1,columns,columns)
mw=[]
staw=[]
for i in range(columns):
    mw.append(round(np.mean(table[0:rows,i]),2))
    staw.append(round(np.std(table[0:rows,i],ddof=1),3))
mmw=np.mean(mw)
mws=np.mean(staw)
UCLm=mmw+A3*mws #Upper control limit
LCLm=mmw-A3*mws #Lower control limit
UCLs=B4*mws
x=[1,columns]
y1=[UCLm,UCLm]
y2=[mmw,mmw]
y3=[LCLm,LCLm]
y4=[UCLs,UCLs]
fig, ax = plt.subplots(2, 1)
ax[0].set_title("Mean value chart")
ax[0].plot(x,y1,'r-')
ax[0].plot(x,y2,'g-')
ax[0].plot(x,y3,'r-')
ax[0].plot(h,mw,'bx-')
ax[0].set_ylabel("Mean")
ax[1].set_title("Standard deviation chart")
```

```
35  ax[1].plot(x,y4,'r-')
36  ax[1].plot(h,staw,'gx-')
37  ax[1].set_xlabel("Samples")
38  ax[1].set_ylabel("s")
39  fig.tight_layout()
40  plt.show()
```

Listing 9.17 Graphical Representation of a Two-Lane Quality Control Chart

Output

Figure 9.5 shows a graphical representation of the two-lane control chart output by the program.

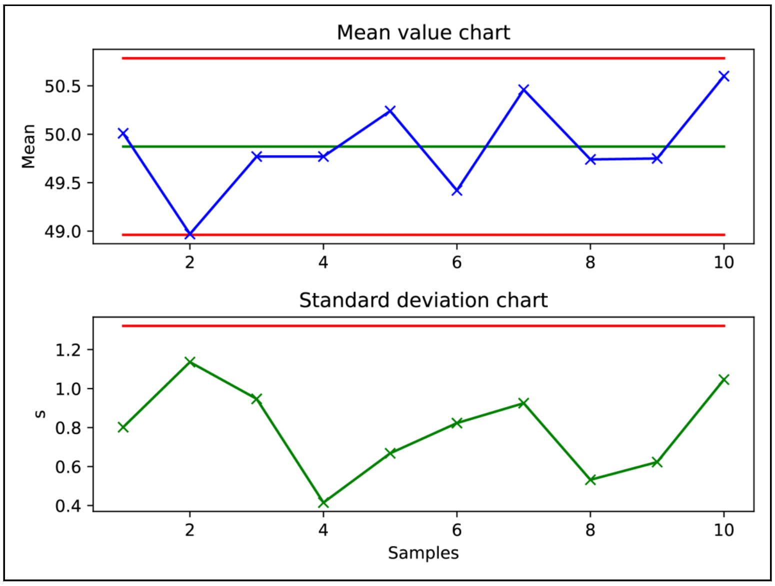


Figure 9.5 Two-Lane Quality Control Chart

Analysis

In lines 14 to 16, the program computes the mean `mw` and standard deviation `staw` of a sample within a `for` loop with column index `i`. The `for` loop iterates all table columns from 1 to 10. This improvement over the first version is achieved by what's called *slicing* with the slicing operator `[index1: index2]`. Both statistical characteristic values are computed for each column from the beginning of the row (index 0) to the end of the row (index 4). Thus, slicing does not require a `for` loop.

Lines 23, 25 and 26 define the y-coordinates of the intervention limits. The mean value of all 50 measurement values is determined by the y-coordinate in line 24.

In line 27, a graphical window for two *subplots* is created using the `subplots(2,1)` method. The first parameter sets the number of subplots, and the second parameter sets the number of columns. Thus, the standard deviation chart is drawn directly below the mean chart. The `subplots()` method creates the `fig` and `ax` objects. `ax[0]` and `ax[1]` can be used to access the methods of the Matplotlib module. The 0 index sets the properties of the first subplot, while index 1 sets the properties of the second subplot.

For further work on this program, a useful next step might be to assign normally distributed random numbers to the `values` variable in line 08 by using the `random.normal()` NumPy function. Thus, with each new program start, you can simulate whether an overrun of intervention limits has occurred in a trend, a run, or a middle third.

9.9 Tasks

1. Write a program to calculate the harmonic mean and compare the runtime with the runtime of SciPy function `stats.hmean(array)`.
2. Write a program to compute the geometric mean and compare the runtime with the runtime of SciPy function `stats.gmean(array)`.
3. The following formula is supposed to be used to compute the standard deviation:

   $$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n} {x_i}^2 - n\overline{x}^2}$$

   Write an appropriate program and compare the runtime with Python function `stdev(array)`, the NumPy function `numpy.std(array,ddof=1)`, and the SciPy function `stats.tstd(array)`.
4. Write a program that computes the absolute and relative probability of how often a given measure might occur in the limits from a to b in a normally distributed series of measurements with n workpieces. The population n of all measurement values and the standard deviation of a series are given.
5. The skew of a frequency distribution can also be computed using the following formula:

   $$S_3 = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{x_i - \bar{x}}{\sqrt{\mathrm{var}(x)}}\right)^3$$

Write a program that determines whether a frequency distribution is left skewed or right skewed. Compare the result with the `scipy.stats.skew(a)` function.

Chapter 10

Boolean Algebra

In this chapter, you'll learn how to create truth tables using Python and how to simplify logical functions by using the SymPy module.

When George Boole formulated the algebra named after him in 1854, no one yet thought of the importance this subfield called discrete mathematics and how it would impact the entire evolution of technology. *Boolean algebra* forms the theoretical foundation of all modern automation and computer systems. In Boolean algebra, variables have only two states: 0 (false) and 1 (true). Only three logical operations are applicable to these variables: the logical AND operation (*conjunction*), the logical OR operation (*disjunction*), and the *negation*.

The representation of functions $y = f(x)$ known from analysis is also applicable to Boolean algebra:

$$y \ = f(x_1, x_2 \ldots x_n) \text{ with } x_i, y \in \{0,1\}$$

Since the independent variables $x_1, x_2 \ldots x_n$ may only take the states 0 or 1, the dependent variables $y_1, y_2 \ldots y_n$ also have only these two states. This functional dependence can be illustrated by a technical system (blackbox) with inputs x and outputs y, as shown in Figure 10.1.

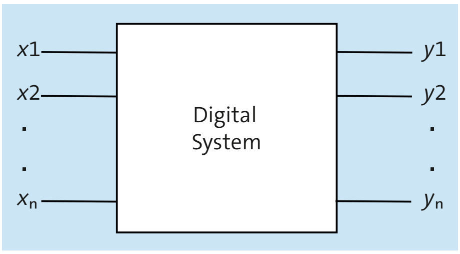


Figure 10.1 A General Representation of a Digital System

Boolean algebra helps you design and develop digital circuits and controls. In the development process, *truth tables* are first created to represent the function of the control task. Logic functions are then derived from these truth tables and simplified using Boolean algebra. The logic functions obtained in this way then form the basis for programming programmable logic controllers (PLCs) or for implementing electronic circuits, for example with transistor-transistor logic (TTL) integrated circuits (ICs) as hardware.

Python provides a powerful tool for simplifying logic functions: the `simplify_logic()` method from the SymPy module.

10.1 Logical Operations

In digital technology only these three different logical operations occur: the AND operation (*conjunction*), the OR operation (*disjunction*), and the *negation*. Using these three basic logical operations, you can implement all digital circuits and controls. Table 10.1 contains the keywords for the basic logical operations.

Operation	Python Keyword
AND	and
OR	or
NEGATION	not

Table 10.1 Keywords for Basic Logical Operations

10.1.1 Conjunction

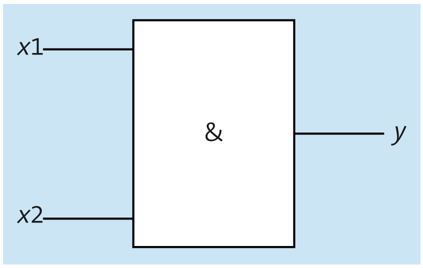


Figure 10.2 Circuit Symbol of the AND Operation

Three variants are common for the notation of the AND operation:

$y = x_1 \wedge x_2$ or $y = x_1 \cdot x_2$ or $y = x_1 \,\&\, x_2$

In Python, the AND operation is implemented using the `and` keyword or the `&` operator (bitwise AND). In your Python development environment, enter the source code provided in Listing 10.1 and run the program.

```
#01_and.py
print("x1\tx2\ty")
for x1 in False,True:
    for x2 in False,True:
```

```
05          y=x1 and x2
06          #y=x1 & x2
07          print(x1,x2,y,sep='\t')
```

Listing 10.1 Program for the Truth Table of an AND Operation

As expected, you'll obtain the following truth table of the AND operation:

```
x1      x2      y
False   False   False
False   True    False
True    False   False
True    True    True
```

For the program to output a table with all four possible combinations of the input variables, a double-nested `for` loop is implemented in lines 03 and 04. The commented-out line 06 contains the `&` operator for the bitwise AND operation. Both variants provide the same result. In line 07, the `sep='\t'` parameter inserts a tab between all outputs.

10.1.2 Disjunction

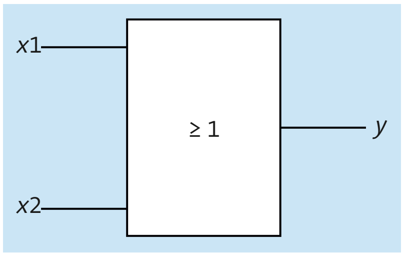


Figure 10.3 Circuit Symbol of the OR Operation

Three variants are common for the notation of the OR operation:

$y = x_1 \vee x_2$ or $y = x_1 + x_2$ or $y = x_1 \mid x_2$

In Python, the OR operation is implemented using the `or` keyword or the `|` operator (bitwise OR). In your Python development environment, enter the source code provided in Listing 10.2 and run the program.

```
#02_or.py
print("x1\tx2\ty")
for x1 in False,True:
    for x2 in False,True:
        y=x1 or x2
```

```
06          #y=x1 | x2
07          print(x1,x2,y,sep='\t')
```

Listing 10.2 Program for the Truth Table of an OR Operation

As expected, you'll obtain the following truth table of the OR operation:

```
x1      x2      y
False   False   False
False   True    True
True    False   True
True    True    True
```

In lines 05 or 06, the OR operation is implemented. The commented-out line 06 returns the same result as line 05 with the | operator (bitwise OR).

10.1.3 Negation

I now want to describe negation through examples using the NAND and NOR functions. The NAND function allows you to negate the output of the AND circuit. The NOR function allows you to negate the output of the OR circuit. The negation is symbolized either with an overbar (¯) or the negation sign (¬). In the circuit symbols for the NAND and NOR circuits, the negation is symbolized by a small circle at the output.

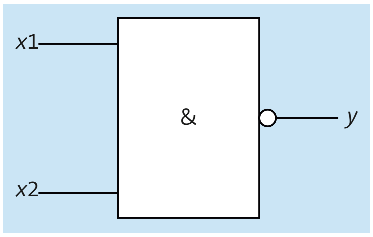


Figure 10.4 Circuit Symbol of the NAND Operation

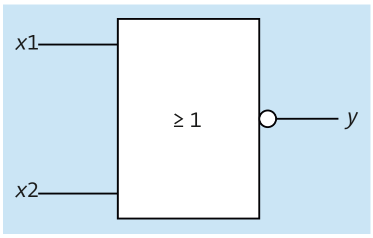


Figure 10.5 Circuit Symbol of the NOR Operation

Logic function for a NAND operation: $y = \overline{x_1 \wedge x_2}$

Logic function for a NOR operation: $y = \overline{x_1 \vee x_2}$

In Python, the negation can be implemented using the `not` keyword or the ~ operator (a tilde, for bitwise negation). Let's examine both functions using the Python program provided in Listing 10.3.

```
#03_negation.py
print("x1\tx2\tNAND\tNOR")
for x1 in False,True:
    for x2 in False,True:
        y1=not (x1 and x2)
        y2=not (x1 or x2)
        print(x1,x2,y1,y2,sep='\t')
```

Listing 10.3 Negated AND and OR Function

The program provided in Listing 10.3 outputs the following truth table of negated AND and OR functions:

```
x1      x2      NAND    NOR
False   False   True    True
False   True    True    False
True    False   True    False
True    True    False   False
```

The negated AND function is `False` if both input variables have the value `True`. The negated OR function is `True` if both input variables have the value `False`.

10.2 Laws of Boolean Algebra

The variables of logic functions are linked conjunctively or disjunctively by logical operations. During the synthesis of digital circuits, often, the determined logic functions contain more terms than would be necessary for the required function of the control. Boolean algebra, also called *switching algebra* in the language of technicians, provides rules to simplify or transform logical functions. Let's now examine some of these rules directly using the Python shell.

10.2.1 Simple Postulates

We'll examine only a few postulates as examples. Enter the following statements into your Python shell:

```
>>> 0 and 1
0
>>> 0 & 1
0
```

```
>>> 0 or 1
1
>>> 0 | 1
1
```

Apparently, the following rule applies: If a 0 is conjunctively linked to a 1, then the result is a 0. If, on the other hand, a 0 is disjunctively combined with a 1, then the result is a 1. These relationships can be vividly described with a series circuit and a parallel circuit consisting of two switches. Current can never flow through a series connection of one open and one closed contact. In a parallel circuit consisting of one open and one closed contact, current always flows.

10.2.2 De Morgan's Laws

Using *De Morgan's laws*, a NAND function can be transformed into a NOR function with negated inputs with the following equation:

$$\overline{x_1 \wedge x_2} = \overline{x_1} \vee \overline{x_2}$$

Similarly, a NOR function can be transformed into a NAND function with negated inputs with the following equation:

$$\overline{x_1 \vee x_2} = \overline{x_1} \wedge \overline{x_2}$$

The program provided Listing 10.4 illustrates the validity of this law.

```
#04_demorgan.py
print("x1\tx2\ty1\ty2\ty3\ty4")
for x1 in False, True:
    for x2 in False, True:
        y1=not(x1 and x2)
        y2=not x1 or not x2
        y3=not(x1 or x2)
        y4=not x1 and not x2
        print(x1,x2,y1,y2,y3,y4,sep='\t')
```

Listing 10.4 Proof of De Morgan's Law

Output

```
x1      x2      y1      y2      y3      y4
False   False   True    True    True    True
False   True    True    True    False   False
True    False   True    True    False   False
True    True    False   False   False   False
```

The table of the program output confirms that the values of the third column (y1) match the values of the fourth column (y2). The same applies to the fifth and sixth columns (y3=y4).

10.2.3 Distributive Law

Using the two forms of the *distributive law*, a digital circuit with three gates with three inputs can be reduced to a circuit with two gates. This simplification is performed by factoring out equal variables, for instance, in the following way.

$$(x_1 \wedge x_2) \vee (x_1 \wedge x_3) = x_1 \wedge (x_1 \vee x_3)$$

$$(x_1 \vee x_2) \wedge (x_1 \vee x_3) = x_1 \vee (x_1 \wedge x_3)$$

Both forms of the distributive law can now be proven using a Python program. Enter the source code provided in Listing 10.5 into your development environment and start the program.

```
01  #05_distributive.py
02  print("x1 \t x2 \t y1 \t y2 \t y3 \t y4")
03  for x1 in False, True:
04      for x2 in False, True:
05          for x3 in False, True:
06              y1=(x1 and x2) or (x1 and x3)
07              y2=x1 and (x2 or x3)
08              y3=(x1 or x2) and (x1 or x3)
09              y4=x1 or (x2 and x3)
10              print(x1,x2,y1,y2,y3,y4,sep='\t')
```

Listing 10.5 Proof of the Distributive Law

Output

```
x1      x2      y1      y2      y3      y4
False   False   False   False   False   False
False   False   False   False   False   False
False   True    False   False   False   False
False   True    False   False   True    True
True    False   False   False   True    True
True    False   True    True    True    True
True    True    True    True    True    True
True    True    True    True    True    True
```

The truth table confirms both forms of the distributive law. In lines 03 to 05, a triple-nested `for` loop is implemented so that all eight possible combinations of input states can be captured.

10.3 Circuit Synthesis

The synthesis of digital circuits occurs in five steps:

1. Clearing a description of the task
2. Setting the input and output variables
3. Entering states of the outputs into the truth table
4. Setting up the logic functions according to the OR normal form or according to the AND normal form
5. Simplifying the logic functions

The simplification of logic functions is usually quite complex and prone to errors. For this reason, you should rely on software-based support for the development process. Python provides developers with a powerful tool for simplifying complex logic functions: the `simplify_logic(y)` method from the SymPy module. Through three examples, I will now demonstrate how to use this tool.

10.3.1 Simplifying Logic Functions by Factoring Out

Enter the following statements into your Python shell:

```
>>> x1=0
>>> x1 or not x1
True
>>> x1=1
>>> x1 or not x1
1
```

Obviously, regardless of the value of the variable x_1, the following rule applies: x_1 `or not` $x_1 = 1$. This law can be illustrated by connecting two switches in parallel. Regardless of whether a switch has the state 0 or 1, the circuit is always closed.

The following example of four input variables $x_1 \dots x_4$ and one output variable y illustrates how a logic function can be simplified. The input variables are conjunctively linked in the following ways:

$y_1 = x_1 \wedge x_2 \wedge \neg x_3 \wedge \neg x_4$

$y_2 = \neg x_1 \wedge x_2 \wedge \neg x_3 \wedge \neg x_4$

$y_3 = x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge \neg x_4$

$y_4 = \neg x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge \neg x_4$

All four logic functions of this example are linked disjunctively in the following way:

$y = y_1 \vee y_2 \vee y_3 \vee y_4$

By factoring out x_1 and x_2, we can obtain the following simplified logic function:

$y = \neg x_3 \land \neg x_4$

Listing 10.6 shows the results.

```
01 #06_simplify1.py
02 from sympy.logic import simplify_logic
03 from sympy import symbols
04 x1, x2, x3, x4 = symbols('x1 x2 x3 x4')
05 y1 = ( x1 &  x2 & ~x3 & ~x4)
06 y2 = (~x1 &  x2 & ~x3 & ~x4)
07 y3 = ( x1 & ~x2 & ~x3 & ~x4)
08 y4 = (~x1 & ~x2 & ~x3 & ~x4)
09 y = y1 | y2 | y3 | y4
10 V = simplify_logic(y)
11 print("y = ",V)
```

Listing 10.6 Simplification of Four Logic Functions with Four Variables Each

Output

```
y = ~x3 & ~x4
```

In line 02, Python's `sympy.logic` module is using the `simplify_logic` method to simplify logic functions. In line 03, the `sympy` module is imported with the `symbols` class for the symbolic processing of the logical operations. The statement in line 04 specifies the names of the logical input variables.

The lines 05 to 08 contain the logic functions `y1` to `y4`. Note that all variables must be linked via the binary operator `&` (bitwise AND). All negations require the unary operator `~` (a tilde, for bitwise negation). In line 0909, these four functions are linked disjunctively using the `|` operator (bitwise OR). The implementation of logic functions is not bound to a special form. You can implement logic functions as Python source code in conjunctive form, in the disjunctive normal form, or in any other valid form.

The `simplify_logic(y)` method in line 10 simplifies the logic function from line 09. The simplified function is then output in line 11. The result matches the theoretically determined value.

10.3.2 Simplification Using the Disjunctive Normal Form

When simplifying using the OR normal form, also called the *disjunctive normal form* (DNF), a logic function is written for each *minterm* (all outputs with $y = 1$).

No.	x_1	x_2	x_3	x_4	y
0	0	0	0	0	1
1	0	0	0	1	1
2	0	0	1	0	1
3	0	0	1	1	1
4	0	1	0	0	1
5	0	1	0	1	0
6	0	1	1	0	0
7	0	1	1	1	0
8	1	0	0	0	1
9	1	0	0	1	0
10	1	0	1	0	1
11	1	0	1	1	1
12	1	1	0	0	0
13	1	1	0	1	0
14	1	1	1	0	1
15	1	1	1	1	1

Table 10.2 Truth Table for a Digital Circuit with Four Inputs

The truth table provided in Table 10.2 contains ten minterms. For the first and the second lines, the minterms are provided. The other eight minterms can be written according to the same schema:

No. 0: $y1 = \neg x_1 \neg x_2 \neg x_3 \neg x_4$

No. 1: $y2 = \neg x_1 \neg x_2 \neg x_3 x_4$

and so on.

However, when you design a digital circuit, writing down the minterms is not necessary. Instead, can use a Karnaugh map (K-map) instead. You can then enter all circuit states of the output variables with $y = 1$ directly into the K-map shown in Table 10.3.

	x_1		$\neg x_1$		
x_2	1				x_4
	1		1		$\neg x_4$
$\neg x_2$	1	1	1	1	
	1		1	1	x_4
	x_3	$\neg x_3$		x_3	

Table 10.3 K-Map for Four Variables

All connected rectangular blocks of two, four, and eight with y = 1 can be combined into simplified terms. Based on the K-map in Table 10.3, you can then obtain the following simplified logic function:

$$y = (x_1 \wedge x_3) \vee (\neg x_1 \wedge \neg x_2) \vee (\neg x_2 \wedge \neg x_4) \ \vee (\neg x_1 \wedge \neg x_3 \wedge \neg x_4)$$

Listing 10.7 simplifies the minterms of the truth table from Table 10.2.

```python
#07_simplify2.py
from sympy.logic import simplify_logic
from sympy import symbols
x1, x2, x3, x4 = symbols('x1 x2 x3 x4')
y1 = (~x1 & ~x2 & ~x3 & ~x4)
y2 = (~x1 & ~x2 & ~x3 &  x4)
y3 = (~x1 & ~x2 &  x3 & ~x4)
y4 = (~x1 & ~x2 &  x3 &  x4)
y5 = (~x1 &  x2 & ~x3 & ~x4)
y6 = ( x1 & ~x2 & ~x3 & ~x4)
y7 = ( x1 & ~x2 &  x3 & ~x4)
y8 = ( x1 & ~x2 &  x3 &  x4)
y9 = ( x1 &  x2 &  x3 & ~x4)
y10 =( x1 &  x2 &  x3 &  x4)
y=y1 | y2 | y3 | y4 | y5 | y6 | y7 | y8 |y9 | y10
V=simplify_logic(y)
print("y = ",V)
```

Listing 10.7 Simplification for Ten Minterms

The output of the program from Listing 10.7 confirms the result obtained using the K-map from Table 10.3:

```
y = (x1 & x3) | (~x1 & ~x2) | (~x2 & ~x4) | (~x1 & ~x3 & ~x4)
```

10.3.3 Simplification Using the Conjunctive Normal Form

When simplifying using the conjunctive normal form (CNF), a logic function is written for each maxterm (all outputs with $y = 0$).

The truth table in Table 10.2 contains six maxterms. The input variables are disjunctively linked in the following ways:

$y_1 = x_1 \vee \neg x_2 \vee x_3 \vee \neg x_4$
$y_2 = x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4$
$y_3 = x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4$
$y_4 = \neg x_1 \vee x_2 \vee x_3 \vee \neg x_4$
$y_5 = \neg x_1 \vee \neg x_2 \vee x_3 \vee x_4$
$y_6 = \neg x_1 \vee \neg x_2 \vee x_3 \vee \neg x_4$

The output variables are conjunctively linked in the follow way:

$y = y_1 \wedge y_2 \wedge y_3 \wedge y_4 \wedge y_5 \wedge y_5$

The logic functions can be directly copied from the truth table in Table 10.2 into the Python source code, as shown in Listing 10.8.

```
#08_simplify3.py
from sympy.logic import simplify_logic
from sympy import symbols
x1, x2, x3, x4 = symbols('x1 x2 x3 x4')
y1 = ( x1 | ~x2 |  x3 | ~x4)
y2 = ( x1 | ~x2 | ~x3 |  x4)
y3 = ( x1 | ~x2 | ~x3 | ~x4)
y4 = (~x1 |  x2 |  x3 | ~x4)
y5 = (~x1 | ~x2 |  x3 |  x4)
y6 = (~x1 | ~x2 |  x3 | ~x4)
y=y1 & y2 & y3 & y4 & y5 & y6
V=simplify_logic(y)
print("y = ",V)
```

Listing 10.8 Simplification for Six Maxterms

Output

```
y = (x1 & x3) | (~x1 & ~x2) | (~x2 & ~x4) | (~x1 & ~x3 & ~x4)
```

The output from Listing 10.8 matches the result of the program from Listing 10.7.

10.4 Project Task: Seven-Segment Coding

In simple displays, for example, in pocket calculators or digital watches, numbers are represented by displays with seven segments, as shown in Figure 10.6.

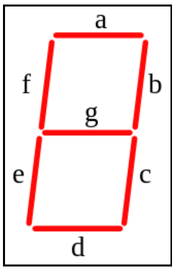


Figure 10.6 Seven-Segment Display

By using a logic function for each segment, we can control the segments in such a way that the desired number is always displayed.

The design of our digital circuit should have the four inputs A, B, C, and D (binary-coded decimal [BCD] code) and the seven outputs a through g. The circuit must control the seven segments so that the numbers from 0 to 9 are shown correctly in the display. For the representation of zero, for example, current must flow in all segments except g.

Table 10.4 defines in detail how the segments should be controlled.

	Inputs				Outputs						
T	D	C	B	A	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	0	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	1	0	1	1
10	1	0	1	0	x	x	x	x	x	x	x
11	1	0	1	1	x	x	x	x	x	x	x
12	1	1	0	0	x	x	x	x	x	x	x
13	1	1	0	1	x	x	x	x	x	x	x
14	1	1	1	0	x	x	x	x	x	x	x
15	1	1	1	1	x	x	x	x	x	x	x

Table 10.4 Truth Table for Seven-Segment Coding

Only the lines from 0 to 9 are needed to display the ten digits. The fields that are not required have been marked with an x. These fields are referred to as *don't-care terms*. Since it does not matter whether they take the logical state 0 or 1, they can be counted as logical 1 for the OR normal form or as logical 0 for the AND normal form, which simplifies the logic functions even further.

Experienced digital technicians can immediately recognize the solution for segment *c*, for example, from the conjunctive normal form:

$c = A \vee \neg B \vee C$

You do not need to consider input D because it can be either 0 or 1, so whether it is present does not matter.

You can transfer the logic functions for the segments a to g directly from the truth table as disjunctive or conjunctive normal form into the source code of the program. The don't-care terms (in lines 10 through 15 in Table 10.4) are not included in Listing 10.9.

```python
#09_seven_segment.py
from sympy.logic import simplify_logic
from sympy import symbols
D, C, B, A = symbols('D C B A')
a=( D|C|B|~A) & (D|~C|B|A) & (D|~C|~B|A)
b=( D|~C|B|~A)&(D|~C|~B|A)
c=( D|C|~B|A)
d=( D|C|B|~A) & (D|~C|B|A) & (D|~C|~B|~A)
e=(~D&~C&~B&~A)|(~D&~C&B&~A)|(~D&C&B&~A)|(D&~C&~B&~A)
f=(D|C|B|~A) & (D|C|~B|A) & (D|C|~B|~A) & (D|~C|~B|~A)
g=(D|C|B|A) & (D|C|B|~A) & (D|~C|~B|~A)
print("a = ",simplify_logic(a))
print("b = ",simplify_logic(b))
print("c = ",simplify_logic(c))
print("d = ",simplify_logic(d))
print("e = ",simplify_logic(e))
print("f = ",simplify_logic(f))
print("g = ",simplify_logic(g))
```

Listing 10.9 Simplification for Seven-Segment Coding

Output

```
a =  D | (A & C) | (B & ~C) | (~A & ~C)
b =  D | ~C | (A & B) | (~A & ~B)
c =  A | C | D | ~B
d =  D | (B & ~A) | (B & ~C) | (~A & ~C) | (A & C & ~B)
e =  ~A & (B | ~C) & (~B | ~D)
```

```
f =  D | (C & ~A) | (C & ~B) | (~A & ~B)
g =  D | (B & ~A) | (B & ~C) | (C & ~B)
```

Analysis

Lines 05 to 11 contain the logic functions of segments a through g. The functions are implemented as either min or max terms. The result for segment `c=A|C|D|~B` (line 07) was expected because nothing at all could be simplified. Individual logic functions could be simplified even further by considering don't-care terms (see task number 6).

10.5 Tasks

1. Justify the following:
   a) 2 | 2 = 2
   b) 3 | 4 = 7
   c) 3 & 8 = 0
   d) 2 & 2 = 2
   e) ~3 = –4
2. Write a program that outputs the truth table of an alternating circuit (antivalence function).
3. Write a program that outputs the truth table for the functions $y_1 = (\neg x_1 \vee x_2 \vee x_3) \wedge \neg(x_1 \vee x_4)$ and $y_2 = \neg x_1 \wedge \neg x_4$.
4. Write a program that simplifies the function $y = (\neg x_1 \wedge x_2 \wedge \neg x_3) \vee (x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_1 \wedge x_2 \wedge \neg x_3)$.
5. For a system with three motors, a warning (indicator light or horn) should be issued if the sum of the powers is greater than or equal to 4kW. The rated power of the motors is P_1 = 1kW, P_2 = 2kW und P_3 = 3kW. Develop a logic function for the controller and write a program for simplifying the logic function.
6. The logic functions of segment c need to be simplified considering don't-care terms. Write an appropriate Python program.
7. Write a program that checks the results of a seven-segment display.

Chapter 11
Interactive Programming Using Tkinter

In this chapter, you'll learn how to create user interfaces using Python's Tkinter module. The project task demonstrates the simulation of a control loop with a PID controller.

One disadvantage of console applications is how they severely limit interactions with the user. The Python console provides only the `input` and `print` methods for input and output but no other interactive controls. In addition, mathematical functions cannot be represented graphically.

To compensate for this shortcoming, several other *graphical user interfaces (GUIs)* for Python exist besides Tkinter, such as pyGTK, wxPython, or PyQt. Because of the sometimes pretty high programming effort and the high required resources, I won't go into these modules in more detail.

The name of the Tkinter module is an abbreviation of "Tk-interface." Tk is a cross-platform toolkit for programming GUI. This toolkit provides all the classes for implementing the controls (widgets), such as *labels (Label)*, single-line text fields (*Entry*) and command *buttons (Button)*. Tkinter is already part of Python, so it does not need to be installed like many other modules.

The particular advantage of Tkinter is that you can write functional prototypes for engineering applications with a GUI with little programming effort. Because Tkinter programs require few resources, they also run on Raspberry Pis.

Tkinter programs are developed in five steps:

1. Importing the Tkinter module
2. Creating an object for the main window of the user interface
3. Creating objects for the controls (widgets)
4. Arranging the controls (widgets) in the user interface
5. Implementing an event query

In all the following programs, prefixes represent the controls we use. This approach makes the source code more clear and easier to read. Based on the prefixes, the controls we use can be identified more quickly in source code analysis.

Table 11.1 contains an overview of the most important controls and their prefixes.

Control (Widgets)	Description	Prefix
Label	Label field and output of text (string)	`lbl`
Entry	Single-line textbox for entering text (string)	`txt`
Button	Command button for triggering events (e.g., a mouse click)	`cmd`
Frame	Area for positioning controls	`frm`
Scale	Slider	`sld`
Canvas	Drawing area for displaying lines, circles, etc.	`can`
Radiobutton	Single selection	`opt`
Checkbutton	Multiple selection	`chk`

Table 11.1 Essential Controls and Their Prefixes

For a description of the Tkinter module, including an overview of all classes and methods, enter the following statements in the Python shell:

```
>>> import tkinter
>>> help(tkinter)
```

Most of the following sample programs refer to the project task described in Section 11.6. The project is divided into individual subtasks, each of which determines self-contained partial solutions: implementing inputs and outputs for numerical calculations, designing user interfaces, graphical representations of functions, querying mouse coordinates to determine function values, and the developing and processing of algorithms for the numerical solutions of the differential equations of the controller.

11.1 Interactions with Command Buttons, Textboxes, and Labels

A minimal version of an interactive program consists of label (*Label*), textboxes (*Entry*), and command buttons (*Button*). The textboxes are needed to enter numerical values for calculations.

The input is always performed using strings. These strings must be converted into floats by the program. A label has two functions: They can be used either to label the text fields (inputs) or to output the results. In the second case, the program must convert the floats of the result into strings. The task of command buttons consists of performing specific actions. Once you have entered the values into the text fields, computing operations must be performed (**Start** button) or the program is supposed to be terminated (**Exit** button).

11.1.1 Labels

You can create a label by using the constructor of the `Label` class:

```
lblE=Label(window, text="Result output"...)
```

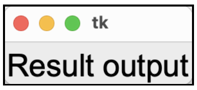
You must pass at least two arguments to the constructor of the `Label` class. The first parameter is the object of the main window, `window`. The second parameter is the `text` property, which determines the text that will be displayed on the user interface. The remaining parameters are optional. By calling the `Label` constructor, you can display the `lblE` object.

The first sample program demonstrates how to use Tkinter to implement a window for simple text output. In your Python development environment, enter the source code provided in Listing 11.1.

```
01  #01_gui_window.py
02  import tkinter as tk
03  window=tk.Tk()
04  lblResult=tk.Label(window,text="Result output",font=("Arial 24"))
05  lblResult.pack()
06  window.mainloop()
```

Listing 11.1 Window with Text Output

When you start the program, the window shown in Figure 11.1 should appear on the screen.



```
Result output
```

Figure 11.1 Text Output Window

Analysis

In line 02, the `tkinter` module is imported, and the `tk` alias is set. This alias enables you to access all methods of the classes of `Tk()`.

In line 03, an object of the `Tk` class named `window` is created. In the further course of the program, this object is needed as a parameter for other controls.

In line 04, an object named `lblResult` of the `Label` class is created. This object is useful to prefix all the names of the label fields. Here and in all sample programs that follow, the `lbl` prefix is used for all label fields. This convention makes reading Tkinter source code much easier when applied to other controls as well (see Table 11.1). The `lblResult` variable can also be used to output the result of a computer operation if required. Three parameters are passed to the `Label()` constructor (method of the same name of the

Label class): The first parameter is the name of the window window created as an object in line 03. This parameter determines in which window the lblResult label will be displayed. The second parameter (text) is passed the label's caption as a string. The third parameter sets the font type and size of the output.

The pack() method in line 05 is needed to display and arrange controls in the main window. If no parameters are passed to this method, then all the controls are arranged one below the other in the order in which they are listed, line by line.

The mainloop() method in line 06 is an infinite loop that is responsible for retrieving all user activities (events), processing them, and executing the requested actions, such as function calls or exiting the program.

All methods are always accessed using the dot operator according to the convention ObjectName.Method(param1, param2, ...).

11.1.2 Textboxes and Command Buttons

You can create a single-line text field (Entry) using the constructor of the Entry class:

```
txtE=Entry(window, width=5,justify="right")
```

The first parameter (window) is again the object of the main window. The second parameter sets the width of the text box, and the third parameter sets the alignment of the text. After calling the constructor the txtE object is created.

You can create a command button via the constructor of the Button class:

```
cmdB=Button(window, text="Compute", command=function)
```

Again, the first parameter is the object of the main window, window. The second parameter sets the label of the command button. To the third parameter (command) must be assigned the name of the function function, which is to be executed when you click the command button. You must omit the usual parentheses in a function call.

11.2 The Layout Manager of Tkinter

Layout managers support the application developer in arranging the controls in the user interface (window). For arranging and designing user interfaces, the following design criteria are suggested in the literature:

- **Proportions**
  Windows should have an aspect ratio from 1:1 to 2:1 (the ratio of width to height), which improves the user orientation. The window shown in Figure 11.2 meets this requirement.

- **Balance**
  A window is divided by a vertical line in the center. The information density (number of controls) should be approximately equally distributed on both sides, left and right. The window shown in Figure 11.2 does not meet this requirement because the information has not been spread across two columns. The label fields should be on the left, and the text fields should be on the right.
- **Symmetry**
  A window is divided by a horizontal line in the middle. The horizontally opposed controls should be of the same type. The labels and the corresponding text fields should therefore each be arranged in one line. A new line must be set up for each new input. On the left-hand side of the window, above and below the horizontal line, there are the labels. On the right-hand side of the window, above and below the horizontal line, there are the text fields. The window shown in Figure 11.2 does not meet this requirement because the labels have not been placed in front of the text fields on one level. However, the requirement for symmetry cannot always be met in real life.
- **Sequence**
  The user's perception should be guided sequentially through the window. Unnecessary jumps should be avoided. Because the user first looks at the upper left-hand area, the most important information should also be at the top left.
- **Simplicity**
  Different fonts or colors should be used with restraint. Each window should be designed to be as simple as possible.
- **Minimizing virtual lines**
  Users unconsciously form virtual lines when viewing user interfaces. In other words, if the controls were not placed vertically at the same distance from the window frame, as shown in Figure 11.2, then the user will find the window design awkward: The harmony of the user's observation will be disturbed.

Tkinter's layout manager provides the following methods: `pack()`, `frame()`, `place()`, and `grid()`. The `frame` method allows you to divide the user interface into separate areas (frames). The layout manager then inserts the controls into the desired frame. The `place` method allows you to place the individual controls anywhere in the user interface (window) by specifying their x-y coordinates. The coordinate origin is located in the upper-left corner of the window. Since the `frame` and `place` methods do not provide satisfactory results, and at a relatively high cost, I won't discuss them further. The sample programs for these two methods can be found in the download area of *https://www.rheinwerk-computing.com/5852/* or *https://drsteinkamp.de*.

The `pack` method is particularly well suited for testing a prototype. If this prototype meets all technical requirements, its design can be optimized using the `grid` layout manager.

11.2.1 The pack Method

The following program shows how to implement the computation of the moment of inertia *J* of a solid steel cylinder with density ρ = 7.85kg/dm3, length *l*, and diameter *d* using the `pack` method.

The moment of inertia of a solid cylinder is calculated in the following way:

$$J = \frac{\varrho l \pi d^4}{32}$$

The formula shows that the program requires two single-line text fields (`Entry`) for the input of the cylinder length *l* and the cylinder diameter *d* as well as one label each for the labeling of the text fields and the result output. You can use the following example as a pattern for the calculation of any complex formula. You only need to add the necessary text fields for the inputs and adjust the function according to the calculation rule.

To keep the source code as simple and clear as possible, this example deliberately avoids an appealing and functional design. In your development environment, enter the source code provided in Listing 11.2 and start the program.

```python
#02_gui_pack.py
import tkinter as tk
#calculation of the moment of inertia
def moment_of_inertia():
    pi=3.14159
    rho=7.85 #kg/dm^3
    d = float(txtDiameter.get())#dm
    l = float(txtLength.get())     #dm
    try:
        J = rho*l*pi*d**4/32
        J = ('{0:6.2f}'.format(1e-2*J)) #kgm^2
        lblResult["text"] = "J=" + str(J) + " kgm^2"
    except:
        lblResult["text"]= "Enter numbers"
#graphics area
main = tk.Tk()
main.minsize(400,200)
main.title("Moment of inertia of a cylinder")
#create label
lblDiameter=tk.Label(main, text="Enter diameter in dm")
lblLength=tk.Label(main, text="Enter length in dm")
lblResult = tk.Label(main,text="")
#create text fields
txtDiameter=tk.Entry(main, width=5,justify="right")
```

```
25 txtDiameter.insert(5, "0.8")
26 txtLength = tk.Entry(main,width=5, justify="right")
27 txtLength.insert(5,"10")
28 #create command buttons
29 cmdCompute=tk.Button(main, text="Compute", command=moment_of_inertia)
30 cmdExit=tk.Button(main, text="Exit", command=main.destroy)
31 #insert controls
32 lblDiameter.pack()
33 txtDiameter.pack()
34 lblLength.pack()
35 txtLength.pack()
36 cmdCompute.pack()
37 lblResult.pack()
38 cmdExit.pack()
39 main.mainloop()
```

Listing 11.2 Program with Label Fields, Text Fields, and Command Buttons

After starting the program, the user interface appears on the desktop roughly as shown in Figure 11.2.

The actual appearance of the window depends on the operating system used (i.e., macOS, Linux, or Windows).

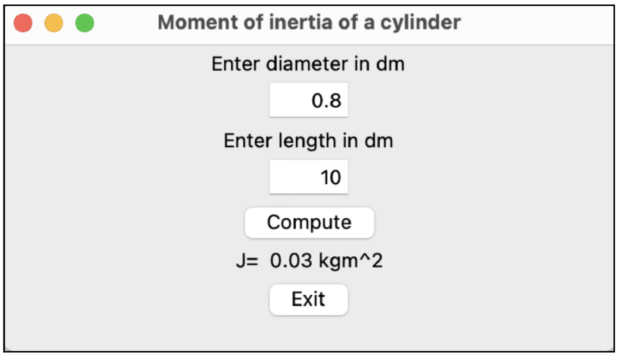


Figure 11.2 Computing the Moment of Inertia

Analysis

From line 04, the `moment_of_inertia()` function is defined. The `get()` method reads the *strings* from the `txtDiameter` and `txtLength` text fields, converts them to floats, and assigns them to the `d` and `l` variables. An exception handling process ensures that input errors (when letters instead of numbers are entered) will be caught (lines 09 to 14).

In line 17, the minimum window size of 400 pixels width and 200 pixels height is set. Thus, the aspect ratio is 2:1.

In line 24, the `txtDiameter` object is created by calling the `Entry()` constructor of the `Entry` class. Three parameters are passed to the constructor: the name of the main window, the width of the text field, and the alignment of the string (right aligned in this case).

The `insert(param1, param2)` method is used to read the content of the text field and store it in the `txtDiameter` object (line 25). The first parameter sets the length of the string. The second parameter contains default values.

In lines 29 and 30, the constructors of the `Button(param1, param2, param3)` class are called. The name of the main window is passed as the first parameter. The second parameter specifies the label of the command button. The third parameter (`command=function name`) is used to call and run a function. Note that the function call omits the parentheses. The `command=main.destroy` method call terminates the program.

The `pack()` method does not yet contain any parameters. The controls are displayed line by line and centered in the order of the `pack` statements.

If you insert the `side="left"` parameter in the `pack` method, the layout manager will place the controls in one row from left to right.

The layout of the program is visually unappealing and dysfunctional for now. Using the `grid` method, you can design a user-friendly interface with little effort.

11.2.2 The grid Method

The `grid()` method can be used to insert all controls into the cells of a table. This method is passed at least two parameters: the row number (`row`) and the column number (`column`). The count starts with zero for the first row and the first column.

In real life, proven advantages come with designing user interfaces using tables, as shown in Table 11.2.

`lblDiameter`	`txtDiameter`
`lblLength`	`txtLength`
`lblMoment of inertia`	`lblResult`
`cmdCompute`	`cmdExit`

Table 11.2 Designing the User Interface

Add the source code provided Listing 11.3 to our example from Listing 11.2 and start the program.

```python
#03_gui_grid.py
import tkinter as tk
#function for moment of inertia
def moment_of_inertia():
    pi=3.14159
    rho=7.85  #kg/dm^3
    d = float(txtDiameter.get()) #dm
    l = float(txtLength.get())   #dm
    try:
        J = rho*l*pi*d**4/32
        J = ('{0:6.2f}'.format(1e-2*J)) #kgm^2
        lblResult["text"] = str(J) + " kgm^2"
    except:
        lblResult["text"]= "Enter numbers"
#graphics area
main = tk.Tk()
main.minsize(400,115)
main.title("Moment of inertia of a cylinder")
lblDiameter=tk.Label(main, text="Enter diameter in dm")
lblLength=tk.Label(main, text="Enter length in dm")
lblMoment_of_inertia = tk.Label(main,text="Moment of inertia")
lblResult = tk.Label(main,text="")
txtDiameter=tk.Entry(main, justify="right")
txtDiameter.insert(2,"0.8")
txtLength = tk.Entry(main, justify="right")
txtLength.insert(2,"10")
cmdCompute=tk.Button(main, text="Compute",command=moment_of_inertia)
cmdExit=tk.Button(main, text="Exit",command=main.destroy)
lblDiameter.grid(row=0,column=0,sticky="w")
txtDiameter.grid(row=0,column=1,sticky="e")
lblLength.grid(row=1,column=0,sticky="w")
txtLength.grid(row=1,column=1,sticky="e")
lblMoment_of_inertia.grid(row=2,column=0,sticky="w")
lblResult.grid(row=2,column=1,sticky="e")
cmdCompute.grid(row=3,column=0)
cmdExit.grid(row=3,column=1)
main.mainloop()
```

Listing 11.3 Arranging Controls Using the Grid Method

The user interface created using the `grid` method is shown in Figure 11.3.

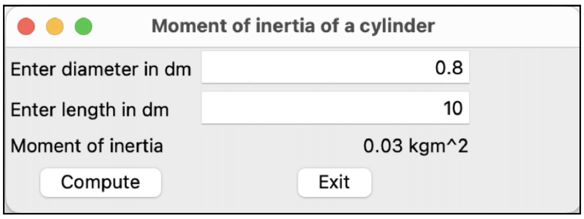


Figure 11.3 User Interface Created Using the Grid Method

Analysis

The label fields are located in the left column (`column=0`), just like the **Compute** command button. The text fields for the inputs are located in the right column (`column=1`), just like the **Exit** command button. The label fields are left aligned (`sticky=w`), whereas the text fields and the result output are right aligned (`sticky=e`). Since the `grid` method of the command buttons does not contain any other properties, they are centered in the cells.

Listing 11.3 shows that the grid layout meets all design criteria for user interfaces with relatively little effort. The following sample programs show further layout options for the `grid` method.

11.2.3 Summary

Let's briefly summarize what we've learned so far: The constructors of the `Label` and `Entry` classes require the name of the main window or the name of the `frame` as the first parameter. Other parameters determine the text of the labels and the appearance of the controls, for instance:

```
lblName = Label(window, properties)
txtName = Entry(windows, properties)
```

For better identification, the names of the controls are given unique prefixes (e.g., `lbl`..., `txt`..., `cmd`..., etc.).

The `Insert()` method of the `Entry` class requires the string length of the variable as the first parameter. It makes sense to specify a realistic default value as the second parameter in the following way:

```
txtName.Insert(stringlength, defaultvalue)
```

The constructor of command buttons expects a third parameter for the function call, in the following way:

```
cmdName = Button(window, label, command=function)
```

The minimum size of a window can be set using the following statement:

```
window.minsize(width, height)
```

Tkinter's layout manager uses the `grid` method to specify the arrangement of all controls as a table grid:

```
objName.grid(row,column)
```

All methods of the `Tk()` class are accessed using the dot operator (.), as in `objName.Method()`.

11.3 Selection with Radio Button

A radio button, also referred to as an option button, is a control that allows only one option out of a number of options to be selected. This control can be created using the following method:

```
Radiobutton(main,text="Sphere",variable=select,value="sp")
```

The first parameter (`main`) refers to the window in which the radio button is to be inserted. The second parameter sets the label. The third parameter (`variable`) must be assigned a global variable that manages and summarizes the options of the selection. The fourth parameter activates the selected option. Listing 11.4 shows how the moment of inertia is calculated for the three bodies (point mass, solid cylinder, and sphere), after the user makes a selection with radio buttons.

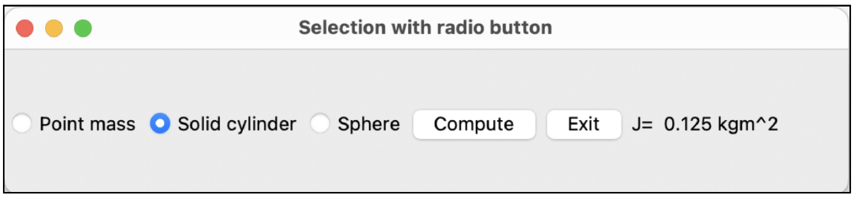
```
#04_radiobutton.py
import tkinter as tk

def compute():
    m=1   #Mass in kg
    r=0.5 #Radius in m
    Jp = m*r**2     #Point mass
    Jz = 0.5*m*r**2 #Solid cylinder
    Jk = 2./5.*m*r**2 #Sphere
    if select.get()=="pm":
        lblResult["text"]=str(Jp)+" kgm^2"
    elif select.get()=="sc":
        lblResult["text"]=str(Jz)+" kgm^2"
    elif select.get()=="sp":
        lblResult["text"]=str(Jk)+" kgm^2"
#graphics area
main = tk.Tk()
```

```
18 main.minsize(580,100)
19 main.title("Selection with radio button")
20 select=tk.StringVar()
21 select.set("sc")
22 #create controls
23 optPm=tk.Radiobutton(main,text="Point mass",variable=select,value="pm")
24 optSc=tk.Radiobutton(main,text="Solid cylinder",variable=select,value="sc")
25 optSp=tk.Radiobutton(main,text="Sphere",variable=select,value="sp")
26 lblJ=tk.Label(main, text="J=")
27 lblResult = tk.Label(main,text="")
28 cmdStart = tk.Button(main, text="Compute",command=compute)
29 cmdExit=tk.Button(main,text="Exit",command=main.destroy)
30 #display controls
31 optPm.pack(side="left")
32 optSc.pack(side="left")
33 optSp.pack(side="left")
34 cmdStart.pack(side="left")
35 cmdExit.pack(side="left")
36 lblJ.pack(side="left")
37 lblResult.pack(side="left")
38 main.mainloop()
```

Listing 11.4 Selection with Radio Buttons

Output

Figure 11.4 shows how selection with radio buttons is enabled and displayed in the user interface when executing the program provided in Listing 11.4.



```
J= 0.125 kgm^2
```

Figure 11.4 Selection with Radio Buttons

Analysis

In line 20, the global `select` variable is created as a copy of the `StringVar` class. In line 21, this variable is initialized using `select.set("sc")`. The `sc` designator is supposed to stand for "solid cylinder." So, the preselection is on the **Solid cylinder** option.

In lines 23 to 25, the `Radiobutton()` method creates the three objects `optPm`, `optSc`, and `optSp`. By assigning the common `select` variable to the `value` property in all three methods, all three radio buttons are combined into one unit.

In lines 10 to 15, the selection is implemented using the `select.get()=="pm"` method. The value of the string variable `select` determines which moment of inertia is supposed to be computed. Note that the case query requires a double equal sign.

In lines 31 to 33, the `pack(side="left")` method causes all radio buttons to be arranged in a row from left to right.

11.4 Slider

The slider control enables you to use the mouse pointer to continuously adjust values for variables. The `Scale(main, parmeterlist)` method creates a slider object.

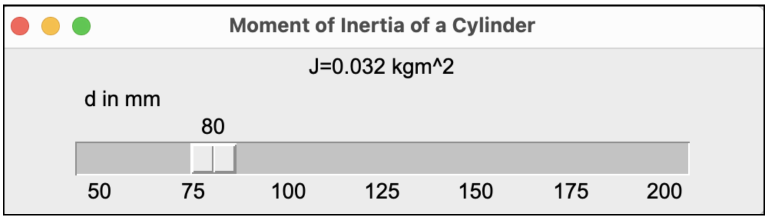
Listing 11.5 calculates the moment of inertia for a solid steel cylinder with a length of 1 meter (m) for cylinder diameters from 50 to 200 millimeters (mm).

```
#05_slider.py
import tkinter as tk
main=tk.Tk()
#moment of inertia
def moment_of_inertia(self):
    pi=3.14159
    rho=7.85 #Density of steel kg/dm^3
    l=10     #Length of the cylinder in dm
    d=sldD.get() #Diameter in mm
    d=1e-2*d #Conversion to dm
    J = rho*l*pi*d**4/32 #in kgdm^2
    J=('{0:5.3f}'.format(1e-2*J))  #Conversion to m
    lblD["text"] = "J=" + str(J) + " kgm^2"
#generate slider object
sldD=tk.Scale(main, width=20, length=400,
              from_= 50, to= 200,  #range in mm
              orient='horizontal',
              resolution=1,        #resolution in mm
              tickinterval=25,
              label="d in mm",
              command=moment_of_inertia, #function call
              font=("Arial 14"))
#graphics area
main.minsize(500,110)
main.title("Moment of inertia of a cylinder")
lblD=tk.Label(main,text="J=",font=("Arial 14"))
sldD.set(80) #set initial value
lblD.pack()
```

```
29  sldD.pack()
30  main.mainloop()
```

Listing 11.5 Computations Using the Slider Control

Output



```
J=0.032 kgm^2
```

Figure 11.5 Computing the Moment of Inertia Using a Slider

Analysis

In line 09, the `get()` method determines the current value of the diameter set by the slider.

In line 15, the slider object, `sldD`, is created for the cylinder diameter. In line 21, the `moment_of_inertia` function is called. Note that you must omit the parentheses. In line 27, you can set an initial value for the cylinder diameter using the `sldD.set(80)` statement.

11.5 The Canvas Drawing Area

The `Canvas` class reserves an area of a window in which objects such as lines, rectangles, and circles can be displayed.

The constructor of the `Canvas` class provides a surface with a width of `xmax` and a height of `ymax`:

```
Canvas(width=xmax, height=ymax, bg='white')
```

The `bg` property sets the background color of the drawing area.

11.5.1 Representing Lines

The following method creates lines in the drawing area:

```
create_line(x1,y1,x2,y2,fill='black',width=3)
```

For the representation of a line, two coordinates must be specified: top left `x1,y1` and bottom right `x2,y2`. The `fill` and `width` properties set the color and width of the line.

Listing 11.6 draws two lines: a horizontal black line in the center of the drawing area and a blue line running from top left to bottom right.

```
01  #06_canvas_line.py
02  import tkinter as tk
03  main = tk.Tk()
04  xmax,ymax=500,500
05  canZ = tk.Canvas(width=xmax, height=ymax, bg='white')
06  canZ.create_line(0,ymax/2,xmax, ymax/2, fill='black', width=3)
07  canZ.create_line(0,0,xmax, ymax, fill='blue', width=3)
08  canZ.pack()
09  main.mainloop()
```

Listing 11.6 Two Lines in the Canvas Drawing Area

Output

Figure 11.6 shows how the lines are displayed in the user interface.

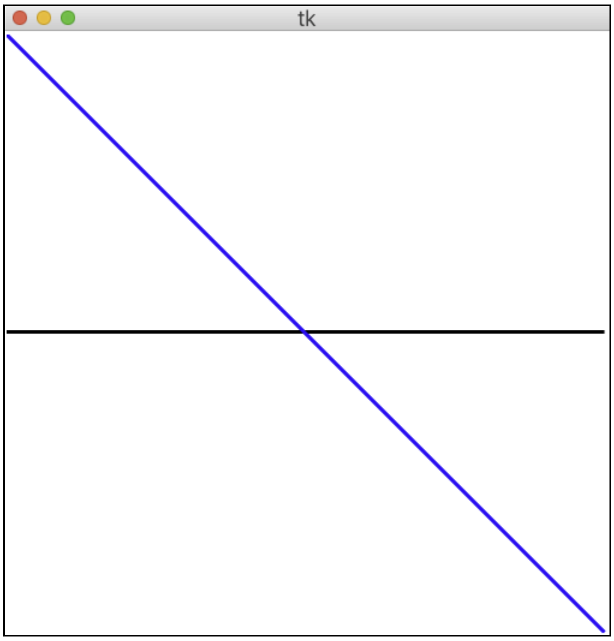


Figure 11.6 Representation of Lines in the Canvas Drawing Area

Analysis

In line 04, the width and the height of the drawing area (canvas) are set to 500 pixels each.

In line 05, the `Canvas()` constructor of the `Canvas` class creates the `canZ` object. This object is needed to access the methods of the `Canvas` class.

In line 06, the `create_line()` method draws a black horizontal line in the center of the drawing area. In line 07, the `create_line()` method draws a blue line from the top-left corner to the bottom-right corner of the drawing area. The lines have a width of 3 pixels.

In line 08, the `pack()` method ensures that the two lines are displayed in the `canZ` drawing area.

11.5.2 Function Plots

The

`create_line(x,-sy*f(sx*x)+ym,(x+1),-sy*f(sx*(x+1))+ym,...)`

method represents a mathematical function—$y = f(x)$—in the canvas drawing area as lines with a “length” of 1 pixel. In this case, `sx` and `sy` are scaling factors that adjust the display range of the drawing area function on the *x* and *y* axes. For the scaling factors, the following applies:

$$s_x = \frac{x_2 - x_1}{x_{\max}}$$

$$s_y = \frac{y_{\max}}{y_2 - y_1}$$

For example, if a sinusoidal function $y = \sin(x)$ with the amplitude of 8 is to be represented in a drawing area with 500×500 pixels on the x-axis in the range from 0 to 10 and the y-axis from –10 to +10, then the values for the arguments of the x-axis must be multiplied by the factor 10/500 = 0.02 and the function values for the y-axis must be multiplied by the factor 20/8 = 2.5. Listing 11.7 represents a sinusoidal function with the amplitude 8 for a value range from 0 to 10.

```python
#07_scaling.py
import math as m
import tkinter as tk
main = tk.Tk()
xmax, ymax = 500, 500
x1,x2 = 0, 10
y1,y2 =-10, 10
ym=ymax/2
sx=(x2-x1)/xmax
sy=ymax/(y2-y1)
canZ = tk.Canvas(width=xmax, height=ymax, bg='white')
canZ.create_line(0,ym,xmax,ym,fill='black',width=2)

def f(x):
    return 8*m.sin(x)

```

```
17  def draw():
18      dx=1
19      x=0
20      while x<=xmax:
21          canZ.create_line(x,-sy*f(sx*x)+ym,(x+1),\
22          -sy*f(sx*(x+1))+ym,fill='blue',width=2)
23          x=x+dx
24
25  cmdStart=tk.Button(main, text="Draw", command=draw)
26  canZ.pack()
27  cmdStart.pack()
28  main.mainloop()
```

Listing 11.7 Plotting a Sine Curve

Output

Figure 11.7 shows the sine curve defined in Listing 11.7 inside the canvas.

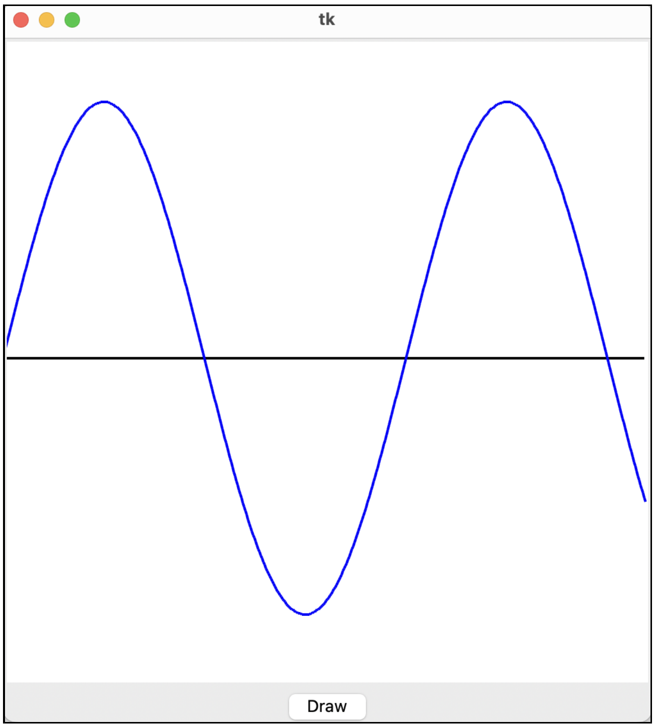


Figure 11.7 Scaling the Drawing Area

Analysis

The drawing area has a size of 500×500 pixels (line 05). The scaling factors sx and sy are computed in lines 09 and 10.

In lines 14 and 15, the mathematical function f(x)=8*m.sin(x) is defined. For further testing purposes, you can also enter other functions here.

The draw() function defined in lines 17 to 23 is called by the Button() constructor of the Button class in line 25.

In lines 21 and 22, the create_line() method draws the sine function as a blue polyline with a width of 2 pixels.

11.5.3 Querying Mouse Coordinates

If you want to read the function value y at position x, you can query the coordinates using the mouse pointer.

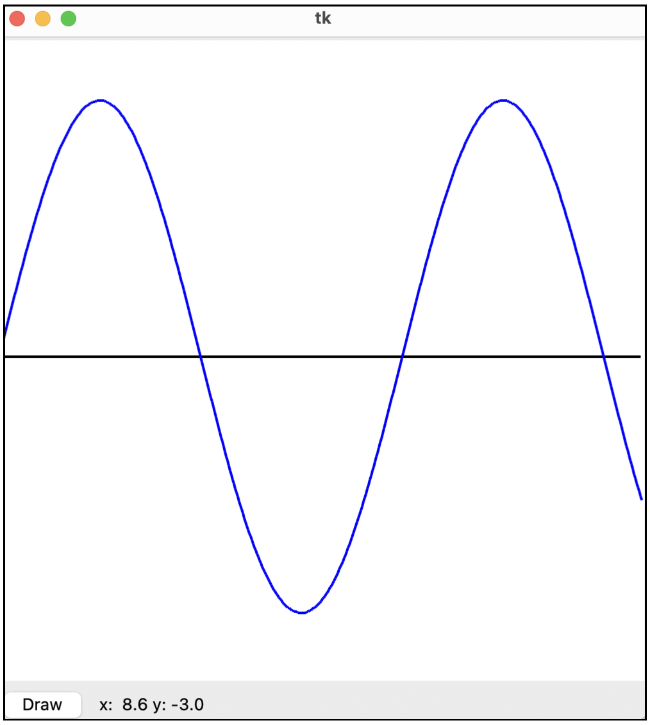
In a custom function, such as coordinate(event), the current event parameter x,y = event.x, event.y is assigned to a tuple (x,y). The bind("<Motion>", coordinate) method determines the current position of the mouse pointer. Listing 11.8 shows how to determine the function values *f*(*x*) of a sine function at position x using the mouse pointer.

```python
#08_mouse_coordinates.py
import tkinter as tk
import math as m
main = tk.Tk()
xmax, ymax = 500, 500
x1,x2 = 0, 10
y1,y2 =-10, 10
ym=ymax/2
sx=(x2-x1)/xmax
sy=ymax/(y2-y1)
canZ = tk.Canvas(width=xmax, height=ymax, bg='white')
canZ.create_line(0,ym,xmax,ym,fill='black',width=2)

def f(x):
    return 8*m.sin(x)

def coordinate(e):
    x, y = e.x, e.y
    x, y = sx*x, (0.5*ymax-y)/sy
    x, y = ('{0:4.1f}'.format(x)), ('{0:4.1f}'.format(y))
    lblCoordinate["text"]='  x: '+str(x)+' y: '+str(y)

def draw():
    dx=1
    x=0
    while x<=xmax:
        canZ.create_line(x,-sy*f(sx*x)+ym,(x+1),\
28          -sy*f(sx*(x+1))+ym,fill='blue',width=2)
29          x=x+dx
30
31  lblCoordinate=tk.Label(main, text="")
32  cmdStart=tk.Button(main, text="Draw", command=draw)
33  canZ.pack()
34  cmdStart.pack(side="left")
35  lblCoordinate.pack(side="left")
36  canZ.bind("<Motion>", coordinate)
37  main.mainloop()
```

Listing 11.8 Querying Mouse Coordinates

Output

How the query of coordinates via mouse pointer is displayed to the user is shown in Figure 11.8.



```
x:  8.6 y: -3.0
```

Figure 11.8 Querying the Position of the Mouse Pointer Coordinates

Analysis

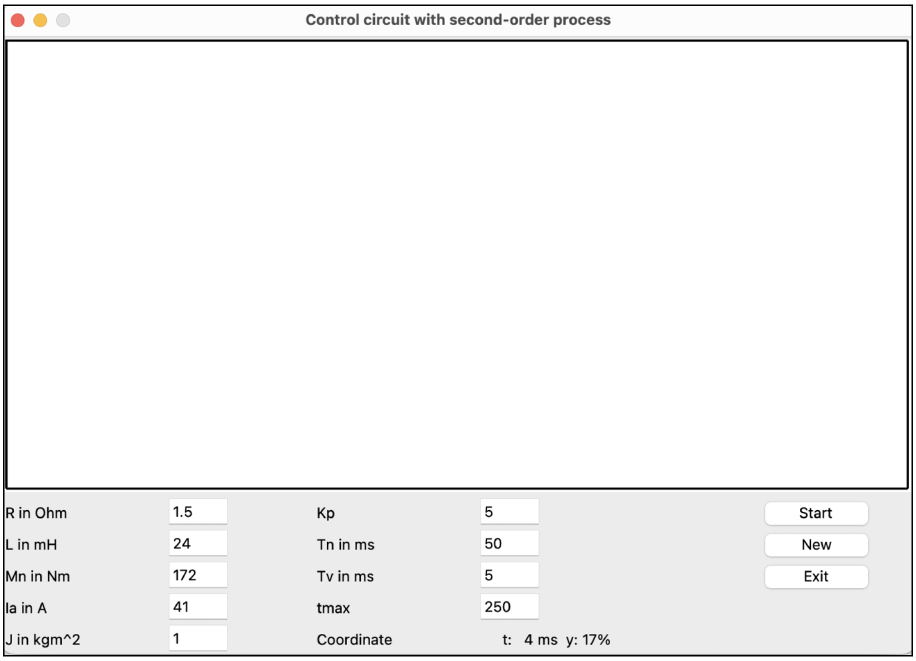
In lines 17 to 21, the `coordinate(e)` function is defined. In line 36, this function is called by the `bind()` method. The `x, y = e.x, e.y` assignment of the current mouse position is performed in line 18. The position of the mouse pointer still needs to be scaled (line 19).

The coordinates of the mouse pointer are converted to a string in line 21 and assigned to the `lblCoordinate` label. The current mouse position is displayed with the `lblCoordinate` label in line 31.

11.6 Project Task: Rotational Frequency Control of an Externally Excited DC Motor

This program is supposed to simulate controlling the rotational frequency of an externally excited DC motor. The data of the process and the PID controller is given.

The planning of the program starts with the design of the user interface, as shown in Figure 11.9.


Control circuit with second-order process

R in Ohm	1.5	Kp	5	Start
L in mH	24	Tn in ms	50	New
Mn in Nm	172	Tv in ms	5	Exit
Ia in A	41	tmax	250	
J in kgm^2	1	Coordinate	t: 4 ms y: 17%	

Figure 11.9 Designing the User Interface

The canvas is located in the upper area. Below the *canvas*, the data of the process must be entered in the left-hand column: armature resistance (*R*), armature inductance (*L*), rated torque (*Mn*), rated armature current (*Ia*), and the moment of inertia (*J*). The middle column is for the gain (*Kp*), reset time (*Tn*), and derivative time (*Tv*) parameters. The command buttons are located in the right-hand column. Except for symmetry, almost all of the criteria we've listed, according to which a user interface should be designed, are fulfilled.

In the next step, you'll analyze the active circuit diagram of a standardized control loop, as shown in Figure 11.10.

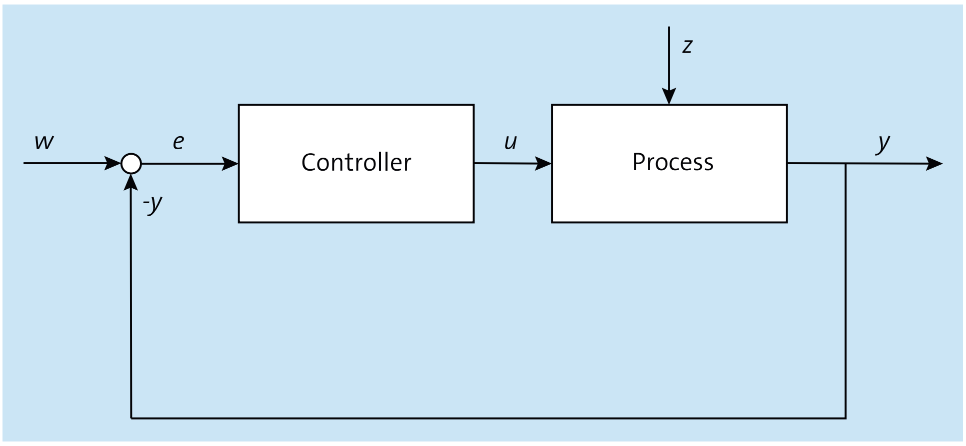


Figure 11.10 Active Circuit Diagram of a Control Loop

In this diagram, *w* represents the reference variable, *e* the control error, *u* the manipulated variable, *z* the disturbance variable, and *y* the controlled variable. For the algorithm of the control loop simulation, you only need to add the equations for the PID controller and the process into a loop. The loop is run until the transient state is completed.

In the third and last step, you must set up the differential equations for the process. This task can be achieved with the help of a substitute circuit of the externally excited DC motor, as shown in Figure 11.11.

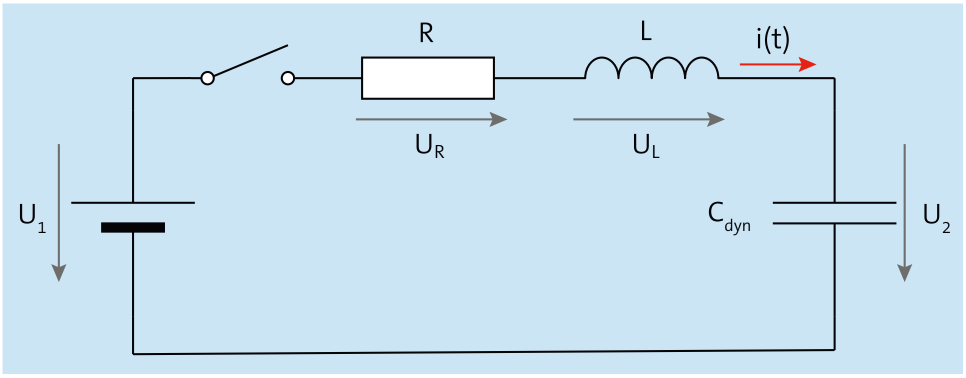


Figure 11.11 Substitute Circuit for an Externally Excited DC Motor

The differential equation system must be developed from the substitute circuit in the following way:

$$\mathrm{d}i = \frac{1}{L}(u_1 - Ri - u_2)\mathrm{d}t$$

$$\mathrm{d}u_2 = \frac{1}{C}i\mathrm{d}t$$

Based on this system of equations, the Euler algorithm can be developed:

```
i = i + (U1-R*i-u2)*dt/L
u2 = u2 + i*dt/C
```

The dynamic capacity is set using the following equation:

$$C = J\left(\frac{I_a}{M_n}\right)^2$$

Listing 11.9 is used to simulate the step response of the process.

```python
#09_process.py
import tkinter as tk
main = tk.Tk()
main.title("Second-order process")
xmax, ymax = 800, 400
canZ = tk.Canvas(width=xmax, height=ymax, bg='white')
#label fields
lblR=tk.Label(main, text="R")
lblL = tk.Label(main, text="L")
lblMn=tk.Label(main, text="Mn")
lblIa=tk.Label(main, text="Ia")
lblJ=tk.Label(main, text="J")
lblTmax=tk.Label(main, text="tmax")
#text fields
txtR=tk.Entry(main, width=5)
txtR.insert(5,"1.5")
txtL=tk.Entry(main, width=5)
txtL.insert(5,"24")
txtMn=tk.Entry(main, width=5)
txtMn.insert(5,"170")
txtIa=tk.Entry(main, width=5)
txtIa.insert(5,"40")
txtJ=tk.Entry(main, width=5)
txtJ.insert(5,"0.22")
txtTmax=tk.Entry(main, width=5)
txtTmax.insert(5,"300")

def delete():
    return canZ.delete("all")

def process():
    w = ymax/2
    U1 = w
```

```
        R = float(txtR.get())
        L = float(txtL.get())
        J = float(txtJ.get())
        Ia = float(txtIa.get())
        Mn = float(txtMn.get())
        tmax=float(txtTmax.get())
        C=1e3*J*(Ia/Mn)**2
        u2=i=t = 0
        dt = 0.5
        sx=xmax/tmax
        u2_t = []
        canZ.create_line(0, w, xmax, w, fill='green', width=2)
        while t<=tmax:
            i = i + (U1-R*i-u2)*dt/L
            u2 = u2 + i*dt/C
            t=t+dt
            u2_t.append(int(sx*t))
            u2_t.append(int(-u2) + ymax)
        canZ.create_line(u2_t, fill='blue', width=2)
#command buttons
cmdStart=tk.Button(main, text="Start", command=process)
cmdNew = tk.Button(main, text="New", command=delete)
cmdExit=tk.Button(main, text="Exit", command=main.destroy)
#arrange controls
canZ.pack()
lblR.pack(side="left")
txtR.pack(side="left")
lblL.pack(side="left")
txtL.pack(side="left")
lblMn.pack(side="left")
txtMn.pack(side="left")
lblIa.pack(side="left")
txtIa.pack(side="left")
lblJ.pack(side="left")
txtJ.pack(side="left")
lblTmax.pack(side="left")
txtTmax.pack(side="left")
cmdStart.pack(side="left")
cmdNew.pack(side="left")
cmdExit.pack(side="left")
main.mainloop()
```

Listing 11.9 Simulation of the Process

Output

Figure 11.12 shows the step response in the user interface.

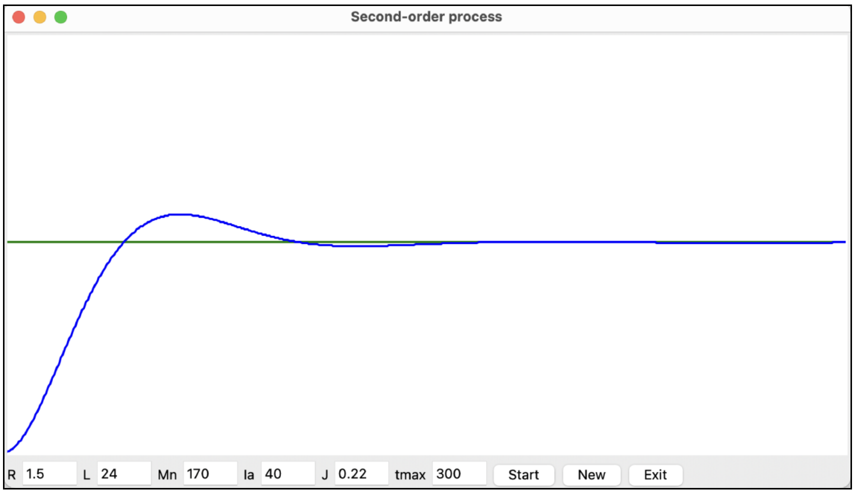


Figure 11.12 Step Response of the Process

Analysis

First, all label fields are defined (lines 08 to 13), followed by the definition of the text fields (lines 15 to 26). The `delete()` function in lines 28 and 29 enables you to delete the drawing area for new simulations.

In lines 31 to 51, the `process()` function is defined. The reference variable `w` in lines 32 and 33 is used to set half of the drawing area. The entries for the process data are made in lines 34 to 38. The dynamic capacity `C` is multiplied by the factor `1e3` in line 40. This operation causes the time axis to be scaled to milliseconds. The initial values `u2`, `i`, and `t` must be initialized with zero (line 41). The scaling of the t-axis is performed in line 43, as described earlier. The empty list `u2_t` (line 44) is needed to store the simulation result. Within the `while` loop (lines 46 to 51), the solution of the differential equation system is performed by using the Euler algorithm. In lines 50 and 51, the time values and the values of the output voltage `u2` are inserted into the empty list, `u2_t`.

In line 54, the `Button` method calls the custom function `process`. When you click the **Start** button, the simulation will be executed.

In lines 58 to 73, the `pack(side="left")` method inserts all controls from left to right in the main window.

The PID Controller

The step response of a PID controller is computed using the following equations for the proportional, differential, and integral components:

$$e = w - y$$

$$u = K_p e + K_p T_v \frac{\mathrm{de}}{\mathrm{dt}} + \frac{K_p}{T_n} \int e\mathrm{d}t$$

The following algorithm can be set up for these equations:

```
up = Kp*e            #P controller
ui = ui + Kp*e*dt/Tn #PI controller
ud = Kp*Tv*(e-e0)/dt #PD controller
e0=e
```

In the final line, the last value of the control error `e` is assigned to the `e0` variable, so that the PD controller can be computed using the difference quotient.

Listing 11.10 simulates either the step response of a P, a PI, or a PID controller.

```
#14_pid_controller.py
import tkinter as tk
main = tk.Tk()
main.title("Step response of P, PI and PID controllers")
xmax, ymax = 800,400
select=tk.StringVar()
select.set("PID")
canZ = tk.Canvas(width=xmax, height=ymax, bg='white')

def delete():
    return canZ.delete("all")

def controller():
    tmax=400
    Kp=50
    Tn=50
    Tv=50
    e=1
    t=ui=e0=ur=0
    dt=2
    sx=xmax/tmax
    u2_t = []
    while t<=tmax:
        up = Kp*e            #P controller
        ui = ui + Kp*e*dt/Tn #PI controller
        ud = Kp*Tv*(e-e0)/dt #PD controller
        e0=e
        t=t+dt
        if select.get()=="P":     ur=up
        elif select.get()=="PI":  ur= up+ui
        elif select.get()=="PID": ur=up+ui+ud
        u2_t.append(sx*t)
        u2_t.append(int(-ur) + ymax)
    canZ.create_line(u2_t, fill='blue', width=2)
optP=tk.Radiobutton(main,text="P controller",variable=select,value="P")
optPI=tk.Radiobutton(main,text="PI controller",variable=select,value="PI")
optPID=tk.Radiobutton(main,text="PID controller",variable=select,value=
"PID")
cmdStart = tk.Button(main,text="Run",command=controller)
cmdNeu=tk.Button(main,text="New",command=delete)
canZ.pack()
optP.pack(side="left")
optPI.pack(side="left")
optPID.pack(side="left")
cmdStart.pack(side="left")
cmdNeu.pack(side="left")
main.mainloop()
```

Listing 11.10 Simulation of the Controller Types

Output

How the step response of the P, PI, or PID controllers is displayed in the user interface is shown in Figure 11.13.

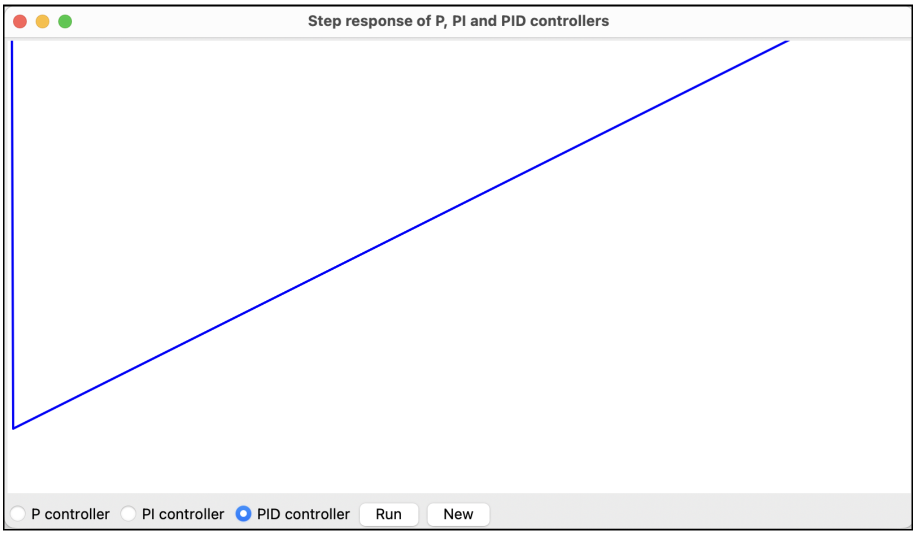


Figure 11.13 Step Response of the PID Controller

Analysis

In line 06, the string variable `select` is defined and set to the default value `PID` in line 07.

In lines 13 to 34, the `controller()` function is defined. For the control parameters, values were selected that provide the most meaningful result possible.

The important statements are in lines 24 to 27, where the algorithm described earlier is implemented. The P controller has the output variable (manipulated variable) `up`; the PI controller has the output variable `ui`; and the PD controller has the output variable `ud`.

The controller is selected in lines 29 to 31. Note that you must use the equality operator `==` in the `if-elif` query.

Simulation Prototype

For complex programs, a useful step is to first program a prototype without text fields and command buttons.

Listing 11.11 shows a simple test version for the control circuit simulation.

```
01  #11_control_circuit1.py
02  import tkinter as tk
03  main = tk.Tk()
04  main.title("PID controller with second-order process")
05  xmax, ymax = 800,400
06  canZ = tk.Canvas(width=xmax, height=ymax, bg='white')
07  U1=100.0
08  R,L=1,25
09  I,M,J = 40,170,1
10  C=1e3*J*(I/M)**2
11  tmax=250
12  Kp=5
13  Tn=50
14  Tv=10
15  t=i=y=ui=e=e0=0.0
16  w=ymax/2
17  dt=0.25
18  sx=xmax/tmax
19  u2_t = []
20  while t<=tmax:
21      e=w-y
22      up = Kp*e
23      ud = Kp*Tv*(e-e0)/dt
24      e0=e
25      ui = ui + Kp*e*dt/Tn
26      U1 = up + ui + ud
27      i = i + (U1-R*i-y)*dt/L
```

```
28        y = y + i*dt/C
29        t=t+dt
30        u2_t.append(sx*t)
31        u2_t.append(-y + ymax)
32    canZ.create_line(u2_t, fill='blue', width=2)
33    canZ.create_line(0, w, xmax, w, fill='red', width=2)
34    canZ.pack()
35    main.mainloop()
```

Listing 11.11 Simulation Prototype

Output

Figure 11.14 shows the control circuit simulation for the test version.

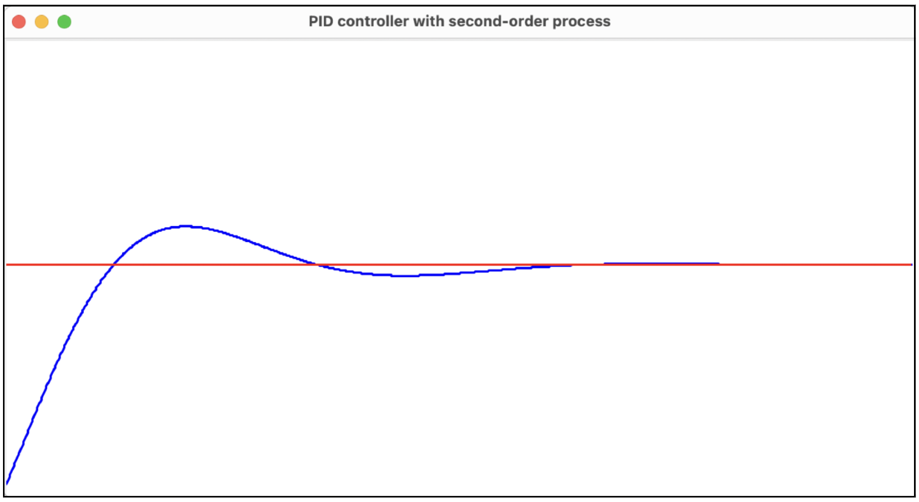


Figure 11.14 Step Response for the Test Version

Analysis

In line 07, the value of the input voltage `U1=100` of the manipulated variable is set. The value 100 was chosen so that the output is a percentage. A value of 100% indicates that the rotational frequency has reached exactly its setpoint. The values for the motor data (lines 08 and 09) were taken from the technical literature.

In line 20, the `while` loop starts, and it ends in line 31. In line 21, the control error `e=w-y` is calculated. Lines 22 to 26 contain the algorithm of the PID controller. Lines 27 and 28 contain the algorithm of the process. The loop is run until the end time `tmax` of 250ms set in line 11 is reached.

Final Version of the Simulation Program

Listing 11.12 shows the final version with all controls.

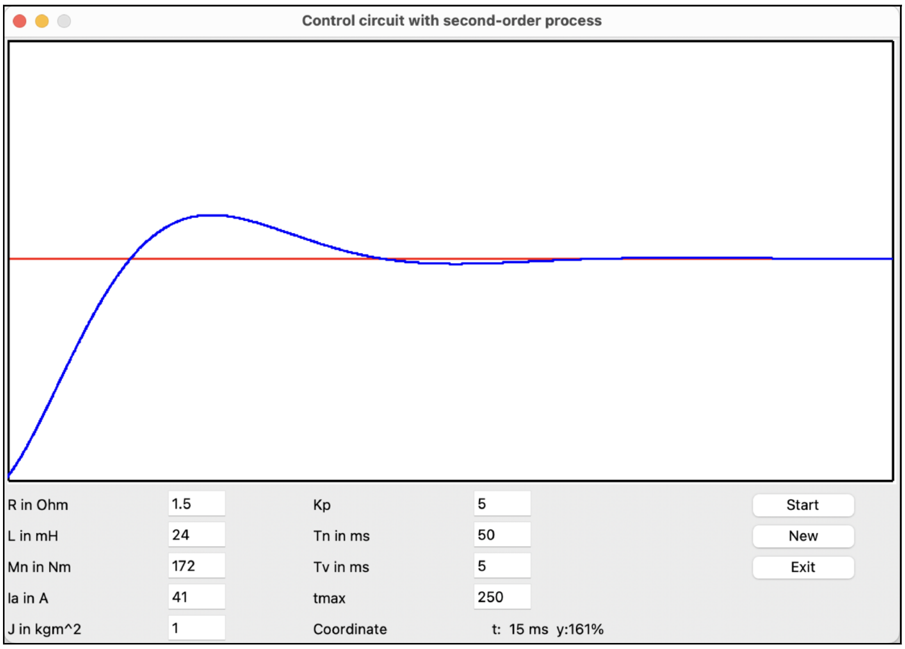
```
#12_control_circuit2.py
import tkinter as tk
main = tk.Tk()
main.title("Control circuit with second-order process")
xmax, ymax = 800,400
xy=0     #global variable
U1=100.0 #setpoint
w=ymax/2.#reference variable
rd=4     #edge
main.minsize(xmax,ymax)
main.resizable(False,False)
canZ = tk.Canvas(width=xmax,height=ymax,bg='white')
canZ.grid(row=0,column=0,columnspan=5)
def frame():
    canZ.create_line(0,w,xmax,w,fill='red',width=2)
    canZ.create_line(rd,rd,xmax,rd,fill="black",width=2)
    canZ.create_line(rd,ymax,xmax,ymax,fill="black",width=2)
    canZ.create_line(rd,0,rd,ymax,fill="black",width=2)
    canZ.create_line(xmax, rd, xmax, ymax,fill="black",width=2)

def delete():
    return canZ.delete(xy)

def coordinate(event):
    tmax=float(txtTmax.get())
    x, y = event.x, event.y
    x, y = tmax*x/xmax-2, U1*(ymax-y)/w
    x, y = ('{0:4.0f}'.format(x)), ('{0:3.0f}'.format(y))
    lblKoordinate["text"]="t:"+str(x)+" ms  y:"+str(y)+"%"

def control_circuit():
    global xy
    tmax=float(txtTmax.get())
    Kp = float(txtKp.get())
    Tn = float(txtTn.get())
    Tv = float(txtTv.get())
    R = float(txtR.get())
    L=float(txtL.get())
    J = float(txtJ.get())
    Ia= float(txtIa.get())
    Mn = float(txtMn.get())
    C=1.0e3*J*(Ia/Mn)**2
    t=y=ui=ud=i=e=e0=0
    dt=0.05
    sx=xmax/tmax
    u2_t = []
    while t<=tmax:
        e=w-y
        up = Kp*e
        ui = ui + Kp*e*dt/Tn
        ud = Kp*Tv*(e-e0)/dt
        e0=e
        U1 = up + ui + ud
        i = i + (U1-R*i-y)*dt/L
        y = y + i*dt/C
        t=t+dt
        u2_t.append(sx*t)
        u2_t.append(-y+ymax)
    xy=canZ.create_line(u2_t,fill='blue',width=2)

frame()
txtTmax=tk.Entry(main, width=5)
txtTmax.insert(5,"250")
#armature resistance
tk.Label(main,text="R in Ohm").grid(row=1,column=0,sticky="w")
txtR=tk.Entry(main, width=5)
txtR.insert(5,"1.5")
txtR.grid(row=1,column=1,sticky="w")
#armature inductance
tk.Label(main,text="L in mH").grid(row=2,column=0,sticky="w")
txtL=tk.Entry(main, width=5)
txtL.insert(5,"24")
txtL.grid(row=2,column=1,sticky="w")
#rated torque
tk.Label(main, text="Mn in Nm").grid(row=3,column=0,sticky="w")
txtMn=tk.Entry(main, width=5)
txtMn.insert(5,"172")
txtMn.grid(row=3,column=1,sticky="w")
#rated current
tk.Label(main, text="Ia in A").grid(row=4,column=0,sticky="w")
txtIa=tk.Entry(main, width=5)
txtIa.insert(5,"41")
txtIa.grid(row=4,column=1,sticky="w")
```

```python
#moment of inertia
tk.Label(main, text="J in kgm^2").grid(row=5,column=0,sticky="w")
txtJ=tk.Entry(main, width=5)
txtJ.insert(5,"1")
txtJ.grid(row=5,column=1,sticky="w")
#controller gain
tk.Label(main, text="Kp").grid(row=1,column=2,sticky="w")
txtKp=tk.Entry(main, width=5)
txtKp.insert(5,"5")
txtKp.grid(row=1,column=3,sticky="w")
#reset time
tk.Label(main, text="Tn in ms").grid(row=2,column=2,sticky="w")
txtTn=tk.Entry(main, width=5)
txtTn.insert(5,"50")
txtTn.grid(row=2,column=3,sticky="w")
#derivative time
tk.Label(main, text="Tv in ms").grid(row=3,column=2,sticky="w")
txtTv=tk.Entry(main, width=5)
txtTv.insert(5,"5")
txtTv.grid(row=3,column=3,sticky="w")
#coordinates
tk.Label(main, text="tmax").grid(row=4,column=2,sticky="w")
txtTmax.grid(row=4,column=3,sticky="w")
tk.Label(main, text="Coordinate").grid(row=5,column=2,sticky="w")
lblKoordinate=tk.Label(main,width=15)
lblKoordinate.grid(row=5,column=3,sticky="w")
#command buttons
tk.Button(main,text="Start",command=control_circuit,
width=7).grid(row=1,column=4)
tk.Button(main,text="New",command=delete,
width=7).grid(row=2,column=4)
tk.Button(main,text="Exit",command=main.destroy,
width=7).grid(row=3,column=4)
canZ.bind('<Motion>', coordinate)
main.mainloop()
```

Listing 11.12 Final Version of the Simulation Program

Output

The output of the final version of the rotation frequency control simulation program is shown in Figure 11.15.



R in Ohm	1.5	Kp	5
L in mH	24	Tn in ms	50
Mn in Nm	172	Tv in ms	5
Ia in A	41	tmax	250
J in kgm^2	1	Coordinate	t: 15 ms y:161%

Figure 11.15 Simulation of the Rotation Speed Curve (Final Version)

Analysis

In line 06, the global `xy` variable is defined, and in the `control_circuit()` function in line 32, the variable is marked as `global`. It occurs again in line 59 where it is assigned all the coordinate data of the simulation. Now, the `delete()` function in line 22 can delete all data stored in the `xy` variable if it is called in line 112 by pressing **New**. Otherwise, the program does not contain any other unknown program elements. For a better orientation, the individual program parts have been commented.

11.7 Tasks

1. Write a Tkinter program that computes the volume, mass, and surface area of a steel cylinder.
2. Write a Tkinter program that calculates the current $I = U/R$ for a simple circuit. Both the voltage and the current are each to be changed using a slider.
3. Write a Tkinter program that computes the current and voltage drops in a series circuit of three resistors. The voltage and the resistances should be changed using a slider in each case.
4. Write a Tkinter program that simulates a control loop with a third-order process.
5. Complement the simulation program for rotational frequency control in such a way that you can select the individual controllers P, PI, or PID via a radio button.

Appendix

A.1 Glossary: Basic Applied Computer Science Terminology

Term	Description
Algorithm	Precise description of how to solve a problem.
Statement	Code section that describes a command or operation.
Argument	Value that is passed when a function is called. This value is assigned to the assigned parameters within the function.
Term	Combination of variables, operations, and values.
Identifier	Name for an object.
Data encapsulation	Controlled access from outside.
Data structure	Objects for which specified operations are defined; its short formula is object + operations.
Function	A self-contained sequence of statements.
Information hiding principle	Under this principle, internal information is hidden.
Interpret	Executing a program by translating it line by line.
Class	Construction plan for objects; its components are properties + methods.
Constructor	Routines for creating and initializing new objects. All objects belonging to the class were created by constructors of this class.
Method	A function definition within a class.
Module	Several classes that have been grouped into a self-contained unit.
Object	An instance of a class, an abstract data type.
Parameter	Name used within the function.
Program	Sequence of statements describing a computation operation.
Return value	Result of a function.
Keyword	Reserved word in a programming language.

Term	Description
Syntax	Structure of a program.
Variable	Name referring to a value (symbolic memory address). A distinction is made between local variables and global variables.
Inheritance	An extended class adopts properties and methods of the base class (superclass).
Assignment	A statement that assigns a value to a variable.

A.2 Derivatives of Elementary Functions

$f(x)$	$f'(x)$	$f(x)$	$f'(x)$
C	0	x^x	$x^x(1 + ln\ x)$
ax^n	nax^{n-1}	$ln\frac{1+x}{1-x}$	$\frac{2}{1-x^2}$
$(a+bx)^n$	$nb(a+bx)^{n-1}$	$\sin x$	$\cos x$
$\sqrt{a^2-x^2}$	$-\frac{x}{\sqrt{a^2-x^2}}$	$\sin ax$	$a\cos ax$
$\sqrt{a^2+bx+x^2}$	$\frac{b+2x}{2\sqrt{a^2+bx+x^2}}$	$\cos x$	$-\sin x$
$\frac{1}{x}\sqrt{a^2-x^2}$	$-\frac{a^2}{x^2\sqrt{a^2-x^2}}$	$\sin^n x$	$n\sin^{n-1}x\cos x$
$\frac{a-x^n}{a+x^n}$	$-\frac{2\,a\,n\,x^{n-1}}{(a+x^n)^2}$	$\sin(\omega x+\varphi)$	$\omega\cos(\omega x+\varphi)$
$\log_a x$	$\frac{1}{x\ln a}$	$x\sin ax$	$ax\cos ax + \sin ax$
$\ln x$	$\frac{1}{x}$	$\tan x$	$\frac{1}{\cos^2 x}$
a^x	$a^x\ ln\ a$	$\cot x$	$\frac{-1}{\sin^2 x}$
e^x	e^x	$\mathrm{arc}\sin x$	$\frac{1}{\sqrt{1-x^2}}$
e^{ax}	$a\,e^{ax}$	$\mathrm{arc}\cos x$	$\frac{-1}{\sqrt{1-x^2}}$
$e^x\,x^n$	$e^x\,x^{n-1}(n+x)$	$\mathrm{arc}\tan x$	$\frac{1}{1+x^2}$

A.3 Antiderivative of Elementary Functions

$f(x)$	$F(x)$	$f(x)$	$F(x)$
1	x	$\sin x$	$-\cos x$
a	ax	$\cos x$	$\sin x$
x^n	$\frac{x^{n+1}}{n+1}$	$\tan x$	$-\ln\cos x$
$\frac{1}{x}$	$ln\ x$	$\cot x$	$\ln\sin x$
a^x	$\frac{a^x}{ln\ a}$	$\frac{1}{\sin x}$	$\ln\tan\frac{x}{2}$
e^x	e^x	$\frac{1}{\cos x}$	$\ln\ \tan\left(\frac{\pi}{4}+\frac{x}{2}\right)$
e^{mx}	$\frac{e^{mx}}{m}$	$\frac{1}{1+\sin x}$	$-\ \tan\left(\frac{\pi}{4}-\frac{x}{2}\right)$
$(ax+b)^n$	$\frac{(ax+b)^{n+1}}{a(n+1)}$	$\frac{1}{1-\sin x}$	$\tan\left(\frac{\pi}{4}+\frac{x}{2}\right)$
$\frac{1}{(ax+b)^2}$	$-\frac{1}{a(ax+b)}$	$\frac{1}{1+\cos x}$	$\tan\frac{x}{2}$
$\frac{1}{ax+b}$	$\frac{1}{a}\ln(ax+b)$	$\frac{1}{1-\cos x}$	$-\cot\frac{x}{2}$
$\frac{1}{\sqrt{a^2-x^2}}$	$\arcsin\frac{x}{a}$	$\frac{1}{\sin^2 x}$	$-\cot x$
$\frac{1}{\sqrt{x^2+a^2}}$	$ln\left(x+\sqrt{x^2+a^2}\right)$	$\frac{1}{\cos^2 x}$	$\tan x$
$\frac{x}{\sqrt{a^2-x^2}}$	$-\sqrt{a^2-x^2}$	$\frac{\sin x}{\cos^2 x}$	$\frac{1}{\cos x}$
$\frac{x}{\sqrt{x^2+a^2}}$	$\sqrt{x^2+a^2}$	$\frac{\cos x}{\sin^2 x}$	$-\frac{1}{\sin x}$
$\sqrt{ax+b}$	$\frac{2}{3a}\left(\sqrt{ax+b}\right)^3$	$\sin x\cos x$	$\frac{1}{2}\sin^2 x$
$\sqrt{a^2-x^2}$	$\frac{x}{2}\sqrt{a^2-x^2}+\frac{a^2}{2}\arcsin\frac{x}{a}$	$\sin^2 x$	$\frac{1}{2}(x-\sin x\cos x)$

A.4 Fourier Series of Important Electrotechnical Voltage Characteristics

$$u(t) = U_0 + \sum_{k=1}^{\infty} a_k \cos k\omega t + b_k \sin k\omega t$$

No.	Function	Fourier Coefficients
1	Rectangle	$U_0 = 0, \quad a_k = 0, \quad b_k = \frac{4\hat{u}}{\pi k} k = 1, 3, 5, \ldots$
2	Rectangle with pulse width control	$U_0 = 0, \quad a_k = 0, \quad b_k = \frac{4\hat{u}}{\pi k} \cos(k\omega t_{an}) k = 1, 3, 5, \ldots$ $t_{an} = \frac{1}{2}\left(\frac{T}{2} - T_p\right)$ T_p : Pulse width
3	Sawtooth	$U_0 = 0, \quad a_k = 0, \quad b_k = \frac{2\hat{u}}{\pi k}(-1)^{k+1} k = 1, 2, 3, \ldots$
4	Isosceles triangle	$U_0 = 0, \quad a_k = 0, \quad b_k = \frac{8\hat{u}}{\pi^2 k^2} k = 1, 3, 5, \ldots$
5	Half-wave rectification	$U_0 = \frac{\hat{u}}{\pi}, \quad a_k = \frac{2\hat{u}}{\pi(1-k^2)}, \quad b_1 = \frac{\hat{u}}{2} k = 2, 4, 6, \ldots$
6	Full-wave rectification	$U_0 = \frac{2\hat{u}}{\pi}, \quad a_k = \frac{4\hat{u}}{\pi(1-k^2)}, \quad b_k = 0 \, k = 2, 4,6, \ldots$
7	Phase angle control	$U_0 = \frac{2\hat{u}}{\pi} \cos^2 \frac{\alpha}{2}$ $a_k = \frac{2\hat{u}(1 + k \sin\alpha \sin k\alpha + \cos\alpha \cos k\alpha)}{\pi(1-k^2)}$ $b_k = \frac{2\hat{u}(\cos\alpha \sin k\alpha - k \sin\alpha \cos k\alpha)}{\pi(1-k^2)}$ $k = 2, 4, 6, \ldots$ $\alpha = \omega t_{an}$: Phase angle

A.5 Correspondence Table of Important Inverse Laplace Transforms

No.	Image Function *F*(*s*)	Original Function *f*(*t*)
1	$\frac{a}{s}$	a
2	$\frac{a}{s^2}$	$a \cdot t$

No.	Image Function $F(s)$	Original Function $f(t)$
3	$\frac{1}{s+a}$	e^{-at}
4	$\frac{a}{s(s+a)}$	$1-e^{-at}$
5	$\frac{a}{s^2+a^2}$	$\sin at$
6	$\frac{s}{s^2+a^2}$	$\cos at$
7	$\frac{b}{(s+a)^2+b^2}$	$e^{-at}\sin bt$
8	$\frac{s+a}{(s+a)^2+b^2}$	$e^{-at}\cos bt$
9	$\frac{\omega_0^2}{s(s^2+2D\omega_0+\omega_0^2)}$	$1-\frac{e^{-\delta t}}{\omega_e}(\delta\sin\omega_e t+\omega_e\cos\omega_e t)$ $D<1, \delta=D\omega_0, \omega_e=\omega_0\sqrt{1-D^2}$

A.6 Bibliography

The following contains the bibliography for the original German-language edition of this book. Where available, English-language resources have been provided:

- Arens, Tilo et al: *Mathematik*. Berlin, Heidelberg 2015.
- Bartsch, Hans-Jochen: *Taschenbuch mathematischer Formeln*. Munich, Vienna 1997.
- Beuth, Klaus: *Digitaltechnik*. Würzburg 2006.
- Czichos, Horst (ed.): *Hütte: Die Grundlagen der Ingenieurwissenschaften*. Berlin, Heidelberg, New York 2000.
- Downey, Allen B.: *Think Python*. O'Reilly 2012.
- Führer, Arnold; Heidemann, Klaus; Nerreter, Wolfgang: *Grundgebiete der Elektrotechnik 2. Zeitabhängige Vorgänge*. Munich 2011.
- Halliday, David; Resnick, Robert; Walker, Jearl: *Fundamentals of Physics. Extended 6th Edition*. John Wiley & Sons 2001.
- Herter, Eberhard; Lörcher, Wolfgang: *Nachrichtentechnik. Übertragung – Vermittlung – Verarbeitung*. Munich, Vienna 2004.
- Holbrook, James G.: *Laplace Transforms for Electronic Engineers*. Pergamon Press Ltd. 1966.
- Höwing, Marika: *Einführung in die Elektrotechnik*. Bonn 2019.

- Johansson, Robert: *Numerical Python: Scientific Computing and Data Science Applications with Numpy, SciPy and Matplotlib*. New York 2019.
- Klein, Bernd: *Einführung in Python. Für Ein- und Umsteiger*. Munich 2021.
- Klein, Bernd: *Numerisches Python. Arbeiten mit NumPy, Matplotlib und Pandas*. Munich 2019.
- Kofler, Michael: *Python*. Der Grundkurs. Bonn 2020.
- Kuchling, Horst: *Taschenbuch der Physik*. Munich 2014.
- Küpfmüller, Karl: *Einstieg in die theoretische Elektrotechnik*. Berlin, Heidelberg, New York 1973.
- Natt, Oliver: *Physik mit Python. Simulationen, Visualisierungen und Animationen von Anfang an*. Berlin 2020.
- Nerreter, Wolfgang: *Grundlagen der Elektrotechnik*. Munich 2006.
- Philippsen, Hans-Werner: *Einstieg in die Regelungstechnik*. Munich 2015.
- Rupprecht, Werner: *Netzwerksynthese. Entwurfstheorie linearer passiver und aktiver Zweipole und Vierpole*. Berlin, Heidelberg, New York 1972.
- Rybach, Johannes: *Physik für Bachelors*. Munich 2008.
- Skolaut, Werner (ed.): *Maschinenbau. Ein Lehrbuch für das ganze Bachelor-Studium*. Berlin, Heidelberg 2014.
- Theis, Thomas: *Einstieg in Python*. Bonn 2019.
- Tietze, Ulrich; Schenk, Christoph; Gamm, Eberhard: *Electronic Circuits --- Handbook for Design and Applications*. Springer 2008.
- Tipler, Paul A.; Mosca, Gene*: Physics for Scientists and Engineers*. With Modern Physics. Sixth Edition. Freeman and Company 2008.
- Vöth, Stefan*: Dynamik schwingungsfähiger Systeme. Von der Modellbildung bis zur Betriebsfestigkeitsrechnung mit MATLAB/SIMULINK*. Wiesbaden 2006.
- Weltner, Klaus: *Mathematik für Physiker und Ingenieure*. Vols. 1 and 2. Berlin, Heidelberg 2013.
- Woyand, Hans-Bernhard: *Python für Ingenieure und Naturwissenschaftler. Einführung in die Programmierung, mathematische Anwendungen und Visualisierungen*. Munich 2021.
- Zimmermann, Klaus: *Übungsaufgaben Technische Mechanik*. Cologne 1994.

The Author

Dr. Veit Steinkamp studied electrical engineering and German to become a teacher and pass on his knowledge at vocational schools and technical colleges. He teaches electrical engineering, application development, and mechanical engineering technology. He has also taught theoretical electrical engineering and the fundamentals of electrical engineering.

Index

Service Pages

The following sections contain notes on how you can contact us.

Praise and Criticism

We hope that you enjoyed reading this book. If it met your expectations, please do recommend it. If you think there is room for improvement, please get in touch with the editor of the book: meaganw@rheinwerk-publishing.com. We welcome every suggestion for improvement but, of course, also any praise!

You can also share your reading experience via Twitter, Facebook, or email.

Supplements

If there are supplements available (sample code, exercise materials, lists, and so on), they will be provided in your online library and on the web catalog page for this book. You can directly navigate to this page using the following link: http://www.rheinwerk-computing.com/5852. Should we learn about typos that alter the meaning or content errors, we will provide a list with corrections there, too.

Technical Issues

If you experience technical issues with your e-book or e-book account at Rheinwerk Computing, please feel free to contact our reader service: support@rheinwerk-publishing.com.

About Us and Our Program

The website http://www.rheinwerk-computing.com provides detailed and first-hand information on our current publishing program. Here, you can also easily order all of our books and e-books. Information on Rheinwerk Publishing Inc. and additional contact options can also be found at http://www.rheinwerk-computing.com.

Legal Notes