Wiley Finance Series

Financial Instrument Pricing Using C++

Second Edition

DANIEL J. DUFFY

WILEY

Financial Instrument Pricing Using C++ 2e

Founded in 1807, John Wiley & Sons is the oldest independent publishing company in the United States. With offices in North America, Europe, Australia and Asia, Wiley is globally committed to developing and marketing print and electronic products and services for our customers' professional and personal knowledge and understanding.

The Wiley Finance series contains books written specifically for finance and investment professionals as well as sophisticated individual investors and their financial advisors. Book topics range from portfolio management to e-commerce, risk management, financial engineering, valuation and financial instrument analysis, as well as much more.

For a list of available titles, visit our website at www.WileyFinance.com.

Financial Instrument Pricing Using C++ 2e

DANIEL J. DUFFY

WILEY

This edition first published 2018

Registered office
John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, United Kingdom

Library of Congress Cataloging-in-Publication Data
Names: Duffy, Daniel J., author.
Title: Financial instrument pricing using C++ / Daniel J. Duffy.
Description: Second Edition. | Hoboken : Wiley, [2018] | Series: Wiley finance series | Revised and updated edition of the author's Financial instrument pricing using C++, c2004. | Includes bibliographical references and index. |
Identifiers: LCCN 2018017672 (print) | LCCN 2018019643 (ebook) | ISBN 9781119170495 (Adobe PDF) | ISBN 9781119170488 (ePub) | ISBN 9780470971192 (hardcover) | ISBN 9781119170495 (ePDF) | ISBN 9781119170518 (Obook)
Subjects: LCSH: Investments–Mathematical models. | Financial engineering. | C++ (Computer program language)
Classification: LCC HG4515.2 (ebook) | LCC HG4515.2 .D85 2018 (print) | DDC 332.60285/5133–dc23
LC record available at https://lccn.loc.gov/2018017672

A catalogue record for this book is available from the British Library.

ISBN 978-0-470-97119-2 (hardback) ISBN 978-1-119-17049-5 (ePDF)
ISBN 978-1-119-17048-8 (ePub) ISBN 978-1-119-17051-8 (Obook)

10 9 8 7 6 5 4 3 2 1

Cover design: Wiley
Cover images: © whiteMocca/Shutterstock

Set in 10/12pt Times by Aptara Inc., New Delhi, India
Printed in Great Britain by TJ International Ltd, Padstow, Cornwall, UK

Contents

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 1

A Tour of C++ and Environs

riverrun, past Eve and Adam's, from swerve of shore to bend of bay, brings us by a commodius vicus of recirculation back to Howth Castle and Environs

—Joyce (1939)

1.1 INTRODUCTION AND OBJECTIVES

This book is the second edition of *Financial Instrument Pricing Using C++*, also written by the author (Duffy, 2004B). The most important reason for writing this hands-on book is to reflect the many changes and improvements to the C++ language, in particular due to the announcement of the new standard C++11 (and to a lesser extent C++14 and C++17). It feels like a new language compared to C++03 and in a sense it is. First, C++11 improves and extends the syntax of C++03. Second, it has become a programming language that supports the functional programming model in addition to the object-oriented and generic programming models.

We apply modern C++ to design and implement applications in computational finance, in particular option pricing problems using partial differential equation (PDE)/finite difference method (FDM), Monte Carlo and lattice models. We show the benefits of using C++11 compared to similar solutions in C++03. The resulting code tends to be more maintainable and extendible, especially if the software system has been properly designed. We recommend spending some time on designing the software system before jumping into code and to this end we include a *defined process* to take a problem description, design the problem and then implement it in such a way that it results in a product that satisfies the requirements and that is delivered on time and within budget.

This book is a detailed exposition of the language features in C++, how to use these features and how to design applications in computational finance. We discuss modern numerical methods to price plain and American options and the book is written in a hands-on, step-by-step fashion.

1.2 WHAT IS C++?

C++ is a general-purpose systems programming language that was originally designed as an extension to the C programming language. Its original name was 'C with classes' and its

object-oriented roots can be traced to the programming language *Simula* which was one of the first object-oriented languages. C++ was standardised by the *International Organization for Standardization* (ISO) in 1998 (called the C++03 standard) and C++14 is the standard at the moment of writing. It can be seen as a minor extension to C++11 which is a major update to the language.

C++ was designed primarily for applications in which performance, efficiency and flexibility play a vital role. In this sense it is a systems programming language and early applications in the 1990s were in telecommunications, embedded systems, medical devices and *Computer Aided Design* (CAD) as well as first-generation option pricing risk management systems in computational finance. The rise in popularity continued well into the late 1990s as major vendors such as Microsoft, Sun and IBM began to endorse object-oriented technology in general and C++ in particular. It was also in this period that the Java programming language appeared which in time became a competitor to C++.

C++ remains one of the most important programming languages at the moment of writing. It is evolving to support new hardware such as multicore processors, GPUs (graphics processing units) and heterogeneous computing environments. It also has a number of mathematical libraries that are useful in computational finance applications.

1.3 C++ AS A MULTIPARADIGM PROGRAMMING LANGUAGE

We give an overview of the programming paradigms that C++ supports. In general, a *programming paradigm* is a way to classify programming languages according to the style of computer programming. Features of various programming languages determine which programming paradigms they belong to. C++ is a multiparadigm programming language because it supports the following styles:

- *Procedural*: organises code around functions, as typically seen in programs written in C, FORTRAN and COBOL. The style is based on structured programming in which a function or program is decomposed into simpler functions.
- *Object-oriented*: organises code around classes. A *class* is an abstract entity that encapsulates functions and data into a logical unit. We instantiate a class to produce *objects*. Furthermore, classes can be grouped into hierarchies. It is probably safe to say that this style is the most popular one in the C++ community.
- *Generic/template*: templates are a feature of C++ that allow functions and classes to operate with generic types. A function or class can then work on different data types.
- *Functional*: treats computation as the evaluation of mathematical functions. It is a declarative programming paradigm; this means that programming is done with expressions and declarations instead of statements. The output value of a function depends only on its input arguments.

The generic programming style is becoming more important and pronounced in C++, possibly at the expense of the traditional object-oriented model which is based on class hierarchies and subtype (dynamic) polymorphism. Template code tends to perform better at run-time while many errors are caught at compile-time, in contrast to object-oriented code where the errors tend to be caught by the linker or even at run-time.

The most recent style that C++ has (some) support for is *functional programming*. This style predates both structured and object-oriented programming. Functional programming has its origins in *lambda calculus*, a formal system developed by Alonzo Church in the 1930s to investigate computability, function definition, function application and recursion. Many functional programming languages can be viewed as elaborations on the lambda calculus. C++ supports the notion of lambda functions. A *lambda function* in C++ is an unnamed function but it has all the characteristics of a normal function. Here is an example of defining a *stored lambda function* (which we can define in place in code) and we then call it as a normal function:

```
// TestLambda101.cpp
//
// Simple example of a lambda function
//
// (C) Datasim Education BV 2018
//
//

#include <iostream>
#include <string>

int main()
{
      // Captured variable
      std::string cVar("Hello");

      // Stored lambda function, with captured variable
      auto hello = [&cVar](const std::string& s)
      { // Return type automatically deduced

         std::cout << cVar << " " << s << '\n';
      };

      // Call the stored lambda function
      hello(std::string("C"));
      hello(std::string("C++"));

      return 0;
}
```

In this case we see that the lambda function has a formal input string argument and it uses a *captured variable* `cVar`. Lambda functions are simple but powerful and we shall show how they can be used in computational finance.

C++11 is a major improvement on C++03 and it has a number of features that facilitate the design of software systems based on a combination of *Structured Analysis* and object-oriented technology. In general, we have a *defined process* to decompose a system into loosely coupled subsystems (Duffy, 2004). We then implement each subsystem in C++11. We discuss this process in detail in this book.

1.4 THE STRUCTURE AND CONTENTS OF THIS BOOK: OVERVIEW

This book examines C++ from a number of perspectives. In this sense it differs from other C++ literature because it discusses the full software lifecycle, starting with the problem description and eventually producing a working C++ program. In this book the topics are based on numerical analysis and its applications to computational finance (in particular, option pricing). In order to design and implement maintainable and efficient software systems we discuss each of the following *building blocks* in detail:

- A1: The new and improved syntax and language features in C++.
- A2: Integrating object-oriented, generic and functional programming styles in C++ code.
- A3: Replacing and upgrading the traditional *Gang-of-Four* software design patterns to fit into a multiparadigm design methodology.
- A4: Analysing and designing large and complex software systems using a combination of top-down system decomposition and bottom-up object assembly.
- A5: When writing applications, determining how much of the features in A1, A2, A3 and A4 to use.

The chapters can be categorised into those that deal with modern C++ syntax and language features, those that focus on system design and finally those chapters that discuss applications. In general, the first ten chapters introduce new language features. Chapters 11 to 19 focus on using C++ to create numerical libraries, visualisation software in Excel and lattice option pricing code. Chapters 20 to 29 are devoted to the finite difference method on the one hand and to multithreading and parallel processing on the other hand. The last three chapters of the book deal with Monte Carlo methods. For easy reference, we give a one-line summary of each chapter in the book:

2. Smart pointers, move semantics, r-value references.
3. All kinds of function types; lambda functions, `std::bind`, functional programming fundamentals.
4. Advanced templates, variadic templates, `decltype`, template metaprogramming.
5. Tuples A–Z and their applications.
6. Type traits and compile-time introspection of template types.
7. Fundamental C++ syntax improvements.
8. IEEE 754 standard: operations on floating-point types.
9. A defined process to decompose systems into software components.
10. Useful data types: static and dynamic bitsets, fractions, date and time, fixed-sized arrays, matrices, matrix solvers.
11. Fundamental software design and data structures for lattice models.
12. Option pricing with lattice models. Both plain and early-exercise cases are considered.
13. Essential numerical linear algebra and cubic spline interpolation.
14. A C++ package to visualise data in Excel (for example, a matrix or array of option prices from a finite difference solver). This package also allows us to use Excel for simple data storage.

15. Univariate statistical distributions in C++ and Boost. We also discuss some applications.
16. The different ways to compute the bivariate cumulative normal (BVN) distribution accurately and efficiently using the Genz algorithm and by solving a hyperbolic PDE. Applications to computing the analytic solution of two-factor asset option pricing problems are given.
17. STL algorithms A–Z. Part I.
18. STL algorithms A–Z. Part II.
19. The solution of nonlinear equations and optimisation. The scope is restricted to the univariate case.
20. A mathematical background to convection–diffusion–reaction and Black–Scholes PDEs.
21. A software framework for the Black–Scholes PDE using the finite difference method.
22. Extending the functionality of the framework in Chapter 21; computing option sensitivities; an analysis of traditional software design patterns. We also discuss opportunities to upgrade software patterns to their multiparadigm extensions.
23. Path-dependent option problems using the finite difference method.
24. Ordinary differential equations (ODEs); theory and numerical approximations.
25. The method of lines (MOL) for PDEs.
26. Random number generation; some numerical linear algebra solvers.
27. Interoperability between ISO C++ and the Microsoft .NET software framework.
28. C++ Concurrency: threads.
29. C++ Concurrency: task.
30. Introduction to Parallel Patterns Library (PPL).
31. Single-threaded Monte Carlo simulation.
32. Multithreaded Monte Carlo simulation.

Appendix 1: Multiprecision data types in C++.

Appendix 2: Computing implied volatility.

This is quite a list of topics. The first ten chapters are essential reading as they lay the foundation for the rest of the book. In particular, Chapters 2, 3, 4, 5, 7 and 8 introduce the most important syntax and language features. Chapters 11 to 19 are more or less independent of each other and we recommend that you read Chapter 9 before embarking on Chapters 11, 12 and 19. Chapters 17 and 18 discuss STL algorithms in great detail. Chapters 20 to 25 are devoted to PDEs and their numerical approximation using the finite difference method. They should be read sequentially. The same advice holds for Chapters 28 to 30 and Chapters 31 to 32.

We have put some effort into creating exercises for each chapter. Reading them and understanding their intent is crucial in our opinion. Even better, actually programming these exercises is proof that you really understand the material.

1.5 A TOUR OF C++11: BLACK–SCHOLES AND ENVIRONS

Since this is a hands-on book we introduce a simple and relevant example to show some of the new features in C++. It is a kind of preview or *trailer*. In particular, we discuss the Black–Scholes option pricing formula and its sensitivities. We focus on the analytical solutions for stock options, futures contracts, futures options and currency options (see Haug, 2007). The approach that we take in this section is similar to how mathematicians solve problems. We quote the famous mathematician Paul Halmos:

> *…the source of all great mathematics is the special case, the concrete example. It is frequent in mathematics that every instance of a concept of generality is, in essence, the same as a small and concrete special case.*

We now describe a mini-system that mirrors many of the design techniques and C++ language features that we will discuss in the other 31 chapters of this book. Of course, it goes without saying that we could implement this problem in a few lines of C++ code, but the point of the exercise is to trace the system lifecycle from beginning to end by doing justice to each stage in the software process, no matter how small these stages are.

We use the following data type:

```
using value_type = double;
```

1.5.1 System Architecture

This is the first stage in which we scope the problem ('what are we trying to solve?') by defining the system scope and decomposing the system into loosely coupled subsystems each of which has a single major responsibility (Duffy, 2004). The subsystems cooperate to satisfy the system's core process, which is to compute plain call and put option prices and their sensitivities. The architecture is based on a *dataflow metaphor* in which each subsystem processes input data and produces output data. Data is transferred between subsystems using a *plug-and-socket architecture* (Leavens and Sitarman, 2000). In general, a system delivers a certain *service* to other systems. A service has a type and it can be connected to the service of another system if the other service is of *dual type*. We sometimes say that a service is a *plug* and the dual service is called a *socket*.

We represent the architectural model for this problem by the *UML* (Unified Modelling Language) *component diagram* in Figure 1.1. Each system does one job well and it interfaces with other systems by means of plugs and sockets. We first define the data that is exchanged between systems:

```
// Option data {K, T, r, sig/v} from Input system
template <typename T>
    using OptionData = std::tuple<T, T, T, T>;

// Return type of Algorithm system
// We compute V, delta and gamma
template <typename T>
    using ComputedData = std::tuple<T, T, T>;
```

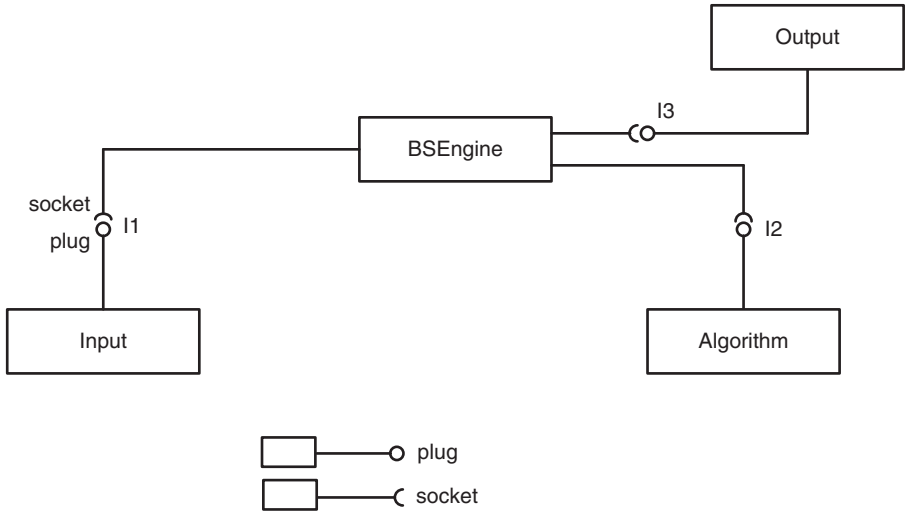


FIGURE 1.1 Context diagram

We also define the interface to compute option price and sensitivities based on option data. To this end, we use *type-safe function pointers*:

```
// The abstract interface to compute V, delta and gamma
template <typename T> using IAlgorithm
      = std::function<ComputedData<T> (const OptionData<T>&
      optData, const T& S)>;
```

Having defined the data structures we now need to design the classes in Figure 1.1 that use them. To this end, we describe how to design these classes.

1.5.2 Detailed Design

In this case we apply the *policy-based design* idiom (Alexandrescu, 2001) to model the classes in Figure 1.1. We are not necessarily endorsing this design as being the best one in general, but it does show some of the important design features that we wish to highlight. We model the input and output systems as template parameters of a template class. We use *private inheritance* and *template–template parameters* to model this class that we call SUD (*System Under Discussion*) while we use a *signature-based approach* to model the algorithms to compute option prices and sensitivities:

```
template <typename T, template <typename T> class Source,
             template <typename T> class Sink>

class SUD : private Source<T>, private Sink<T>
{ // System under discussion, in this case for Black Scholes equation
```

```
private:
   // Define 'provides'/'requires' interfaces of satellite systems
   using Source<T>::getData;                    // Get input
   using Sink<T>::SendData;                     // Produce output
   using Sink<T>::end;                          // End of program

   // Conversion
   IAlgorithm<T> convert;
public:
   SUD(const IAlgorithm<T>& conversion): convert(conversion) {}
   void run(const T& S)
   {
      // The main process in the application
      OptionData<T> t1 = getData();             // Source
      ComputedData<T> t2 = convert(t1, S);      // Processing
      SendData(t2);                             // Sink

      end();                                    // Notification to Sink
   }
};
```

Thus, this class inherits from its source and sink classes and it is composed of an algorithm.

The member function `run()` ties in the participating systems to produce the desired output. We should have a clear idea of the data flow in the system.

1.5.3 Libraries and Algorithms

Examining Figure 1.1 we see that the `Algorithm` subsystem computes option prices from option data. It has the same signature as the interface `IAlgorithm` and this means that we can configure these algorithms with any other *callable object* (for example, a free function, function object or lambda function) that has the same signature as `IAlgorithm`. This means that we do not need to create class hierarchies to achieve this level of flexibility. Furthermore, we can switch algorithms at run-time more easily than with traditional object-oriented technology.

C++ has support for a number of mathematical functions that are useful in computational finance. In this section we introduce the *error function* that allows us to compute the *univariate cumulative normal distribution*:

```
// Normal variates etc.
double n(double x)
{
    const double A = 1.0 / std::sqrt(2.0 * 3.14159265358979323846);
    return A * std::exp(-x*x*0.5);
}

// C++11 supports the error function
```

```
auto cndN = [](double x)
       { return 0.5 * (1.0 - std::erf(-x / std::sqrt(2.0))); };

double N(double x)
{ // The approximation to the cumulative normal distribution

    return cndN(x);
}
```

We now use these functions to compute the analytical solution of plain call and put option prices and their sensitivities. The aggregated value is placed in a *tuple* which is a new data type in C++11:

```
// Option Pricing; give price+delta+gamma
template <typename V>
    ComputedData<V> CallValues(const OptionData<V>& optData,
    const V& S)
{
    // Extract data
    V K = std::get<0>(optData); V T = std::get<1>(optData);
    V r = std::get<2>(optData); V v = std::get<3>(optData);
    V b = r; // Stock option

    // Common functionality
    V tmp = v * std::sqrt(T);
    V d1 = (std::log(S / K) + (b + (v*v)*0.5) * T) / tmp;
    V d2 = d1 - tmp;

    V t1 = std::exp((b - r)*T); V t2 = std::exp(-r * T);
    V Nd1 = N(d1); V Nd2 = N(d2);

    V price = (S * t1 * Nd1) - (K * t2* Nd2);
    V delta = t1*Nd1;
    V gamma = (n(d1) * t1) / (S * tmp);

    return std::make_tuple(price, delta, gamma);
}

// Option Pricing; give price+delta+gamma
template <typename V>
    ComputedData<V> PutValues(const OptionData<V>& optData,
    const V& S)
{
    // Extract data
    V K = std::get<0>(optData); V T = std::get<1>(optData);
    V r = std::get<2>(optData); V v = std::get<3>(optData);
    V b = r; // Stock option

    // Common functionality
    V tmp = v * std::sqrt(T);
```

```
    V d1 = (std::log(S / K) + (b + (v*v)*0.5) * T) / tmp;
    V d2 = d1 - tmp;

    V t1 = std::exp((b - r)*T); V t2 = std::exp(-r * T);
    V Nmd2 = N(-d2); V Nmd1 = N(-d1);

    V price = (K * t2 * Nmd2) - (S * t1* Nmd1);
    V delta = t1*(Nmd1 - 1.0);
    V gamma = (n(d1) * t1) / (S * tmp);

    return std::make_tuple(price, delta, gamma);
}
```

We see how useful tuples are as return types of functions. This is a more efficient solution than creating a separate function for each of an option's price, delta and gamma functions.

1.5.4 Configuration and Execution

We now discuss how to configure the objects and interfaces in Figure 1.1. This usually takes place by either creating the needed objects directly in the body of `main()` or by outsourcing this process to *creational design patterns* and *builders* (see GOF, 1995). In general, we need to choose what we want. As an example, we consider the following hard-coded *source* and *sink* classes:

```
template <typename T> class Input
{
public:

   static OptionData<T> getData ()
   { // Function object

      T K = 65.0; T expiration = 0.25;
      T r = 0.08; T v = 0.3;
      OptionData<T> optData(K, expiration, r, v);

      return optData;
   }
};

template <typename T> class Output
{
public:

   void SendData (const ComputedData<T>& tup) const
   {
      ThreadSafePrint(tup);
   }
```

```
    void end() const
    {
       std::cout << "end" << std::endl;
    }

};
```

Multiple threads can write to the console in a non-deterministic way. For this reason we create a lock on the console if and when we port the single-threaded code to a multithreaded program. The corresponding *thread-safe code* is:

```
template <typename T>
    void ThreadSafePrint(const ComputedData<T>& tup)
{ // Function to avoid garbled output on the console

    std::mutex my_mutex;
    std::lock_guard<std::mutex> guard(my_mutex);
    std::cout << "(" << std::get<0>(tup) << "," << std::get<1>(tup)
              << "," << std::get<2>(tup) << ")\n";
}
```

We create classes to model calls and puts as follows:

```
template <typename T> class Processing
{
public:

   ComputedData<T> convert(const OptionData<T>& optData,
                           const T& S) const
   {
      return CallValues(optData, S);
   }

   ComputedData<T> operator () (const OptionData<T>& optData,
                                const T& S) const
{
      return CallValues(optData, S);
   }

};

template <typename T> class ProcessingII
{
public:

   ComputedData<T> convert(const OptionData<T>& optData,
                           const T& S) const
   {
      return PutValues(optData, S);
   }
```

```
    ComputedData<T> operator () (const OptionData<T>& optData,
                                 const T& S) const
    {
       return PutValues(optData, S);
    }

};
```

Having created the objects that we need we are now in a position to run the application:

```
Processing<value_type> converter;

// Calls
SUD<value_type, Input, Output> callPricer(converter);
value_type S = 60.0;
callPricer.run(S);

// Puts
ProcessingII<value_type> converter2;
SUD<value_type, Input, Output> putPricer(converter2);
value_type S2 = 60.0;
putPricer.run(S2);
```

The output in this case is:

```
(2.13337,0.372483,0.0420428)
end
(5.84628,-0.372483,0.0420428)
end
```

1.6 PARALLEL PROGRAMMING IN C++ AND PARALLEL C++ LIBRARIES

The *C++ Concurrency* library supports both multithreading and multitasking, that is creating programs that are executed by multiple independent threads of control. Multithreading is *pre-emptive* in the sense that the scheduler allocates a fixed amount of time (a *quantum*) to each thread after which time the thread reverts to *sleep* or to *wait-to-join* mode. The library also supports the creation of programs and algorithms by decomposing them into components that can potentially run in parallel with little interaction between them. In general terms, *potential or exploitable concurrency* involves our being able to structure code to permit a problem's subproblems to run on multiple processors. Each subproblem is implemented by a task. A *task* (Quinn, 2004) is a program in local memory in combination with a collection of I/O ports. Tasks send data to other tasks through their output ports and they receive data from other tasks through their input ports.

We take a simple example. In this case we parallelise the code in Section 1.5.4. Both of the following solutions are special cases of a more general *fork-join idiom* in which a single

(main) thread creates two child threads. Each child thread executes code independently of the other child threads. Since the shared data is read-only in this special case there is no danger of non-deterministic behaviour. For both solutions the main thread or task must wait on its children to complete before it can proceed.

We now describe the solution using *C++ Concurrency*. We first encapsulate the algorithms in stored lambda functions:

```
// Parallel execution
auto fn1 = [&converter](value_type S)
{
   SUD<value_type, Input, Output> callPricer(converter);
   callPricer.run(S);
};

auto fn2 = [&converter2](value_type S)
{
   ProcessingII<value_type> converter2;
   SUD<value_type, Input, Output> putPricer(converter2);
   putPricer.run(S);
};
```

The stock value is:

```
value_type stock = 60.0;
```

The solution using C++ threads is:

```
// Threads
std::thread t1(fn1, stock);
std::thread t2(fn2, stock);

// Wait on threads to complete
t1.join(); t2.join();
```

For the task-based solution we use C++ *asynchronous futures*:

```
// Asynchronous Tasks
std::future<void> task1(std::async(fn1, stock));
std::future<void> task2(std::async(fn2, stock));

// Wait on threads to complete
task1.wait(); task2.wait();

// Get results from tasks
task1.get(); task2.get();
```

Our final example is to show how to parallelise this code using the *OpenMP* library (Chapman, Jost and Van der Pas, 2008). We do not discuss this library in this book, but we

recommend it as a good way of learning how to write multithreaded applications before moving to *C++ Concurrency*. In this case we create an array of threads and we execute them using *loop-level parallelism*:

```
// OMP solution
std::vector <std::function<void(value_type)>> tGroupFunctions
            = { fn1, fn2 };

value_type stock = 60.0;

#pragma omp parallel for
for (std::size_t i = 0; i < tGroupFunctions.size(); ++i)
{
   tGroupFunctions[i](stock);
}
```

The output produced from this code is:

```
(2.13337,0.372483,0.0420428)
end
(5.84628,-0.372483,0.0420428)
end
(2.13337,0.372483,0.0420428)
end
(5.84628,-0.372483,0.0420428)
end
(2.13337,0.372483,0.0420428)
end
(5.84628,-0.372483,0.0420428)
end
```

In general, writing parallel applications using tasks is easier than using threads because tasks hide many of the tricky *synchronisation* and *notification* use cases when using threads and it is possible to apply the system decomposition technique (Duffy, 2004) to help create *task-dependency graphs* that we then implement in C++.

1.7 WRITING C++ APPLICATIONS; WHERE AND HOW TO START?

This book centres around the modern multiparadigm language features in C++11 and C++14. We discuss most of the essential language features in the first ten chapters. Later chapters introduce a number of topics that build on these first ten chapters. These are shown in Figure 1.2 in the form of a *concept map*. The main goal of this book is to develop tools to show how to design and implement software systems for computational finance applications. Based on this remark we show in Figure 1.2 how we will achieve the goal by displaying the main concepts and

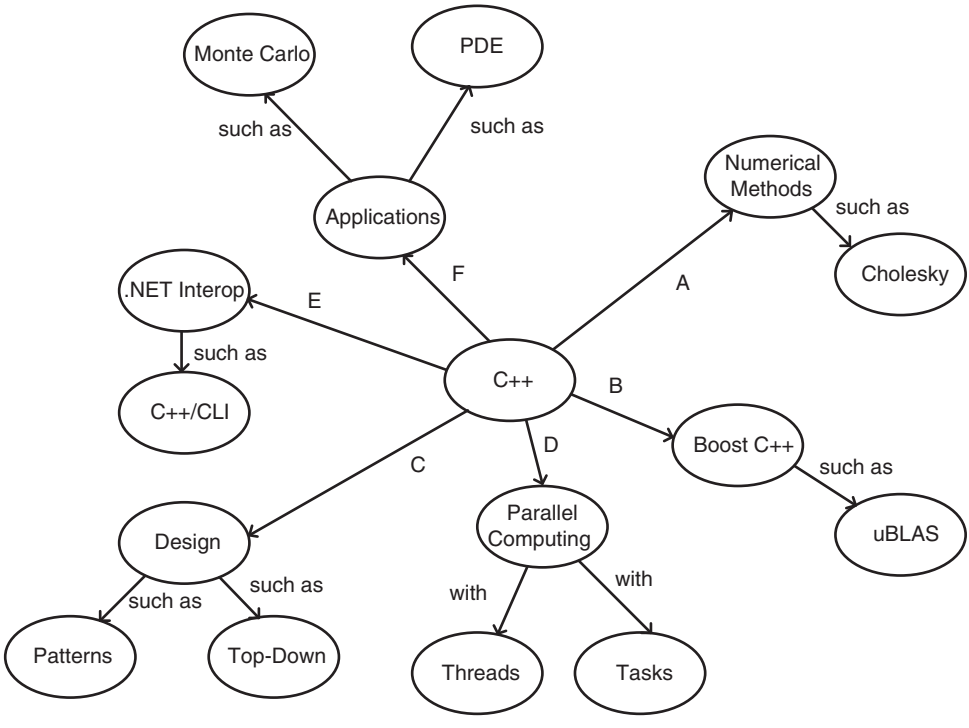


FIGURE 1.2 C++ and environs

their relationship with C++. In general, we use standard C++ in most of the examples unless otherwise mentioned. The book is self-contained in the sense that we prefer to use standard libraries (such as those in C++ as well as *Boost* and *Quantlib*) rather than proprietary libraries. The code for the numerical methods in the book is self-contained and has been developed by the author.

We give a short description of the concepts in Figure 1.2 and their relationship with the applications in this book:

- A: We develop code for well-known numerical methods such as numerical differentiation, interpolation, numerical quadrature, matrix algebra, finding the zeroes of nonlinear equations and optimisation problems.
- B: We use the *Boost* C++ libraries when we need functionality that is not (yet) in C++. Much of the new functionality in C++ had its roots in *Boost*. In general, the *Boost* libraries tend to be reasonably well documented. We recommend the *Boost Math Toolkit* for numerical applications. We also use the *Boost odeint* library to numerically solve systems of ordinary differential equations (ODEs).
- C: This book is unique in our opinion because it introduces a *defined process* to analyse, design and implement any kind of software system in a step-by-step fashion. The process is based on the author's experience as a requirements analyst and software architect in several application domains (see Duffy, 2004 where these domains have been documented).

The process is a fusion of *Structured Analysis* (De Marco, 1978) and the object-oriented paradigm. We apply the process to creating *design blueprints* for FDM and Monte Carlo applications.

- D: Modern laptop and desktop computers have multiple processors on board. This opens the door to developing multithreaded and parallel code to increase the *speedup* of programs. C++ offers support in the *C++ Concurrency* library.
- E: We decided to include a chapter on the .NET language C++/CLI that bridges the (native/ISO) C++ and .NET worlds. C++ is and remains a *systems programming language* which makes it suitable for certain kinds of applications. The C++/CLI language then allows us to create C++ applications that can call .NET functionality on the one hand while it is possible to create .NET wrappers for native C++ classes and call them from C# code. This approach promotes code reusability and helps C# developers because they do not have to learn C++. All they need is to wrap C++ code in .NET wrappers.
- F: In this book, we design option pricing software using lattice methods, Monte Carlo and PDE/FDM methods. We propose a range of numerical methods, design patterns and design styles. Furthermore, we use *C++ Concurrency* to parallelise the corresponding algorithms.

1.8 FOR WHOM IS THIS BOOK INTENDED?

C++ is probably one of the most difficult programming languages to master. The learning curve is much steeper than that of other object-oriented languages such as C# and Java, for example. The only real way to learn C++ is to program in C++ and it is for this reason we say that you need to have a number of years of solid C++ experience writing code and applications. **In other words, this is not a beginner's book to learn C++.**

The primary focus of this book is to design and implement maintainable and extendible applications and it is suitable for front-office and middle-office quant developers who use C++ in their daily work. The book is also useful for software architects and project managers who wish to understand and manage software projects.

This book is also useful for MSc students in finance. We would hope that the step-by-step approach will help them structure their theses.

The first ten chapters can be read by a range of C++ developers because the topics are application independent and to our knowledge this is the first book on the new features in C++11. Chapters 20 to 25 could also be of interest to mathematical physicists.

1.9 NEXT-GENERATION DESIGN AND DESIGN PATTERNS IN C++

The approach taken in this book is special in the sense that we find it important to have an idea of the software system that we wish to build before developing the code to implement it. This is in the interest of developer productivity. We need *design blueprints* that describe the system at a level higher than raw C++ code. Computer science is not yet an engineering discipline and there are few *standardised* design processes and standards to help developers analyse, design and implement C++ applications. One possible exception is the famous *software design*

patterns that were first published in GOF (1995) although, when used on their own, they do not ensure that the software system will be stable.

Design patterns have become very popular in the last 20 years as witnessed by the number of books devoted to them for object-oriented languages such as Java, C#, C++ and others. Neither the structure nor the number of patterns have changed much in the last 20 years as most of the literature seems to imitate the 23 patterns in GOF (1995). In a sense, these patterns were invented when C++ was still in its infancy and when it only supported the traditional object-oriented technology based on subtype polymorphism (`virtual` functions) and class hierarchies. Our basic premise is that the GOF patterns represent knowledge that has not adapted to improvements in software, hardware and development methods.

The GOF design patterns are based on the object model which means that the patterns are implemented using objects, classes and class hierarchies in combination with subtype polymorphism. This means that an abstract requirement regarding the flexibility of a software design must be turned into an explicit data model by introducing a *proxy* for a non-computable concept. This process is called *reification* and it allows any aspect of a programming language to be expressed in the language itself.

It is possible to *upgrade* the GOF patterns in a number of ways:

- S1: Keep and use the patterns in their current form without change.
- S2: Improve the patterns by using new improved C++ functionality such as shared pointers and the syntax that we discuss in the first ten chapters of this book.
- S3: Re-engineer those patterns that can be implemented more easily and correctly using the generic and functional programming models, for example.
- S4: Do not (yet) use design patterns but instead postpone their use in the design trajectory for as long as possible. Instead, we use the system decomposition techniques of Chapter 9 and we hope to achieve the same (and improved) levels of flexibility as with GOF patterns but then by other means, specifically by defined standardised interfaces between components.

1.10 SOME USEFUL GUIDELINES AND DEVELOPER FOLKLORE

We conclude this chapter with some guidelines on the estimation, planning and management of software projects. The size of a project can range from a one-person software endeavour lasting three months to a 30-person application with a lifetime of five years, for example. Some of the principles underlying our design approach can be summarised by the steps that György Pólya described when solving a mathematical problem (Pólya, 1990):

1. First, you have to *understand the problem*.
2. After understanding, then *make a plan*.
3. *Carry out the plan*.
4. *Look back* at your work. How could it be better?

We see these steps as being applicable to the software development process in general and to the creation of software systems for computational finance in particular. Getting each step right saves time and money. In short, we take the following tactic regarding software projects: *get it working, then get it right and only then get it optimised* (in that order). In the current

context we translate these steps into a defined software process (as explained in Chapter 9) in order to avoid some scary outcomes, for example:

> *A Big Ball of Mud is a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle. These systems show unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated. The overall structure of the system may never have been well defined. If it was, it may have eroded beyond recognition. Programmers with a shred of architectural sensibility shun these quagmires. Only those who are unconcerned about architecture, and, perhaps, are comfortable with the inertia of the day-to-day chore of patching the holes in these failing dikes, are content to work on such systems.*

We use the following general principles when developing software systems:

1. Understand the problem as soon as possible. Can you explain the problem to non-developers?
2. Can you develop a software prototype in a few days?
3. Scope the problem by identifying the boundaries of the software system.
4. Decompose the system into loosely coupled subsystems.
5. Use a suitable combination of the object-oriented, generic and functional programming styles.

Each developer has her own way to design software systems. There is no *silver bullet* (Brooks, 1995).

1.11 ABOUT THE AUTHOR

Daniel J. Duffy is the author of both the first and second editions of *Financial Instrument Pricing Using C++*. He started his company Datasim in 1987 to promote C++ as a new object-oriented language for developing applications. He played the roles of developer, architect and requirements analyst to help clients design and analyse software systems in areas such as CAD, process control and hardware–software systems, logistics, holography (optical technology) and computational finance. He used a combination of top-down functional decomposition and bottom-up object-oriented programming techniques to create stable and extendible applications (for a discussion, see Duffy, 2004 where we have grouped applications into domain categories). He also worked on engineering applications in oil and gas and semiconductor industries using a range of numerical methods (for example, the finite element method (FEM)).

Daniel Duffy has BA, MSc and PhD degrees in pure and applied mathematics (from Trinity College, the University of Dublin) and has been active in promoting PDE/FDM for applications in computational finance. He was responsible for the introduction of the *Fractional Step (Soviet Splitting)* method and the *Alternating Direction Explicit* (ADE) method in computational finance.

He is the originator of two popular C++ online courses on www.quantnet.com in cooperation with Quantnet LLC and Baruch College (CUNY), NYC. He also trains quant developers around the world. He can be contacted at: dduffy@datasim.nl. The official Datasim site is www.datasimfinancial.com.

1.12 THE SOURCE CODE AND GETTING THE SOURCE CODE

The C++ code in this book is based on the C++11 standard (and later versions). The only exception is the code in Chapter 14 where we introduce the Excel Driver library and in Chapter 27 where we discuss interfacing between C++ and Microsoft's .NET Framework. Furthermore, the code that is presented in each chapter is machine readable and has been tested beforehand. Our development environment is Visual Studio C++ which supports all the C++ functionality that we present in this book. We have not tested the code using other compilers but we would not expect major issues. It is the responsibility of the reader to know how to install the compiler, *Boost C++* libraries and *Quantlib*.

Regarding copyright, legitimate owners of the book are entitled to the source code which can be used for personal use provided you do not remove the copyright notice in the source code `(C)Datasim Education BV 2018`.

For further queries concerning training and support, please contact me directly at dduffy@datasim.nl. The corresponding website is www.datasim.nl.

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 2

New and Improved C++ Fundamentals

2.1 INTRODUCTION AND OBJECTIVES

In this chapter we introduce new syntax and functionality in C++ that adds value to the language as a worthy successor to C++03. We discuss the features that promote the run-time efficiency, reliability and usability of code, namely:

- *Automatic memory management*: avoiding memory disasters that the use of raw pointers can lead to.
- *Move semantics*: allowing the compiler to replace expensive copying operations with less expensive move operations. In some cases, move semantics are the only options as some classes do not support copy constructors. Typical examples are smart pointers and classes for multithreading and multitasking.
- New fundamental data types.

The features in this chapter are crucial as they represent *best practices* when writing C++ code. For this reason we introduce these features early on in the book. We also strongly recommend that you do the exercises in this chapter to become acquainted with the new features as soon as possible.

2.2 THE C++ SMART POINTERS

In this chapter we first introduce (as background) the *Boost Smart Pointer* library that makes object lifecycle management easier when compared to using the *new* and *delete* operators in C++. In particular, the responsibility for removing objects from memory is taken from the developer's shoulders. To this end, we discuss a number of classes that improve the reliability of C++ code. The two most important classes are:

- *Scoped pointer*: ensures proper deletion of dynamically allocated objects. These are objects having a short lifetime (for example, *factory objects*) and they are typically created and deleted in a single scope. In other words, these objects are not needed outside the code block in which they are defined and used.
- *Shared pointer*: ensures that a dynamically allocated object is deleted only when no other objects are referencing it. The shared pointer class eliminates the need to write code to explicitly control the lifetime of objects. It enables *shared ownership* of objects.

The four other classes in Boost are:

- *Scoped array*: ensures proper deletion of dynamically allocated arrays in a scope.
- *Shared array*: enables shared ownership of arrays. It is similar to the shared pointer class except that it is used with arrays instead of with single objects.
- *Weak pointer*: this is an *observer* of a shared pointer. It does not interfere with the ownership of the object that the shared pointer shares. Its main use is to avoid dangling pointers because a weak pointer cannot hold a dangling pointer. For completeness we note that a valid weak pointer can be promoted to a strong pointer (shared) and hence the lifetime of the original shared pointer might be affected/extended by that promotion.
- *Intrusive pointer*: this is a special kind of smart pointer and is used in code that has already been written with an internal reference counter. You can write your own smart pointer class when you are not happy with the performance of shared pointers or when the software that you are using denotes it.

2.2.1 An Introduction to Memory Management

In this section we give a short overview of object lifecycles. In particular, we are interested in *heap-based memory* allocation of objects. Such memory allocation and deallocation in C++ is the responsibility of the developer. But knowing when memory is no longer needed is not easy to determine and this uncertainty can lead to a number of problems:

- *Dangling pointers*: these occur when an object is deleted or deallocated without modifying the value of the pointer. In this case the pointer continues to point to the location of the deallocated memory.
- *Wild pointers*: these are pointers that are used before they are initialised.
- *Memory leak*: in this case the program is unable to release memory that it has acquired. This situation can be caused by a pointer when it goes out of scope. In other words, the dynamically allocated memory is unreachable and lost forever.
- *Double free bugs*: this refers to the case when we try to delete memory that has already been deleted.

In order to resolve (or avoid) these problems we have a number of options open to us. We can allow automatic memory management by using *garbage collection* (GC) that is supported by some languages such as C# and Java. The garbage collector attempts to reclaim memory used by objects that will never be accessed again in an application. Garbage collection is the opposite of *manual memory management*. There are various kinds of garbage collectors, the most common being called *trace garbage collectors.* These first determine which objects are reachable (or potentially reachable) and they then proceed to discard the remaining 'dead' objects.

There is also the *reference counting* technique (used in C++) that stores the number of pointers or handles to a dynamically allocated object. When an object is no longer referenced it will be deleted from memory. Reference counting is a form of garbage collection in which each object contains a count of the number of references to it that are held by other objects. Reference counting can entail frequent updates because the reference count needs to be incremented or decremented when an object is referenced or dereferenced.

Some of the advantages of reference counting are:

- Objects are reclaimed as soon as they are no longer referenced and then in an incremental fashion without incurring long waits on collection cycles.
- Reference counting is one of the simplest forms of garbage collection to implement.
- It is a useful technique for the management of non-memory resource objects (such as file and database handles).

Some disadvantages of reference counting are:

- Frequent updates are a source of inefficiency because objects are being continually accessed. Furthermore, each memory-managed object must reserve space for a reference count.
- Some reference-counting algorithms cannot resolve reference cycles (objects that refer directly or indirectly to themselves).

We now discuss the smart pointer library in C++11. We note that classical garbage collection is not implemented in C++. There is a need in C++ for some kind of automatic (or semi-automatic) memory management mechanism and to this end C++ provides template classes to help developers create reliable and robust code. In general, we use smart pointers in the following situations:

- Avoiding the errors that we discussed in this section.
- Creating objects with well-defined lifetimes.
- Shared ownership of resources.
- Resolving a number of exception-unsafe problems when using raw pointers.

2.3 USING SMART POINTERS IN CODE

We discuss three classes of smart pointers in C++. In the next sections we focus on the syntax of each class and we give some simple examples to show what they do.

2.3.1 Class `std::shared_ptr`

This smart pointer class implements the concept of *shared ownership*. A *resource* or object (a piece of memory on the heap or a file handle, for example) is shared among a number of shared pointers. Only when the resource is no longer needed is it deleted. This is when the *reference count* becomes zero.

We discuss shared pointers in some detail. First, we show how to create empty shared pointers and shared pointers that are coupled to resources. We also show how a shared pointer gives up ownership of one resource and how it becomes owner of another resource. We can see how many shared pointers *own* a resource by using the member function `use_count()`:

```
#include <memory>

// Handy alias
template <typename T>
   using SP = std::shared_ptr<T>;
using value_type = double;
```

```
// Creating shared pointers with default deleters
SP<value_type> sp1;                              // empty shared ptr
SP<value_type> sp2(nullptr);                     // empty shared ptr for
                                                 // C++11 nullptr

SP<value_type> sp3(new value_type(148.413));     // ptr owning raw ptr
SP<value_type> sp4(sp3);                         // share ownership with sp3
SP<value_type> sp5(sp4);                         // share ownership with sp4
                                                 // and sp3

// The number of shared owners
std::cout << "sp2 shared # " << sp2.use_count() << '\n';
std::cout << "sp3 shared # " << sp3.use_count() << '\n';
std::cout << "sp4 shared # " << sp4.use_count() << '\n';

sp3 = sp2;   // sp3 now shares ownership with sp2;
             // sp3 no longer has ownership of its previous resource
std::cout << "sp3 shared # " << sp3.use_count() << '\n';
std::cout << "sp4 shared # " << sp4.use_count() << '\n';
```

In the above cases the last owner of the resource is responsible for destroying the resource and by default this is achieved by a call of the operator `delete`. We now show how to create shared pointers in combination with a user-defined *deleter*. This is a useful option when you wish to execute a command just before destroying a resource, for example notifying clients or printing a message. We can implement a deleter using a function object, lambda function or stored lambda function (which we shall discuss in Chapter 3), in this case as a function object and as a stored lambda function:

```
// Memory deleters
template <typename T>
    struct Deleter
{
    void operator () (T* t) const
    {
       std::cout << "delete memory from function object\n";
       delete t;
    }
};

// Creating shared pointers with user-defined deleters

// Deleter as function object
SP<value_type> sp(new value_type(148.413), Deleter<value_type>());

// Deleter as lambda function
SP<value_type> sp2(new value_type(148.413), [](value_type* p)
                                              { std::cout << "bye\n";
                                                delete p; });
```

```
// Stored lambda function as deleter
auto deleter = [](value_type* p)
                { std::cout << "bye\n"; delete p; };
SP<value_type> sp32(new value_type(148.413), deleter);
```

Continuing, we now discuss more ways to construct shared pointers. They are:

- `std::make_shared<T>`: construct an instance of `T` and wrap it in a shared pointer using arguments as the parameter list for the constructor of `T`.
- `std::allocate_shared`: construct an instance of `T` and wrap it in a shared pointer using arguments as the parameter list for the constructor of `T`. An explicit memory allocator object is one of the arguments to this function.

In the second case above there is an option to give a memory allocator as one of the arguments (in this case we show examples of both C++ and Boost C++ allocators):

```
struct Point2d
{
    double x, y;
    Point2d() : x(0.0), y(0.0) {}
    Point2d(double xVal, double yVal) : x(xVal), y(yVal) {}
    void print() const {std::cout << "(" << x << "," << y << ")\n";}
    ~Point2d() { std::cout << " point destroyed\n"; }
};

// More efficient ways to construct shared pointers
auto sp = std::make_shared<int>(42);
(*sp)++;
std::cout << "sp: " << *sp << '\n'; // 43

auto sp2 = std::make_shared<Point2d>(-1.0, 2.0);
(*sp2).print(); // (-1, 2)

auto sp3 = std::make_shared<Point2d>();
(*sp3).print(); // (0,0)

// More efficient ways to construct shared pointers
auto sp = std::allocate_shared<int>(std::allocator<int>(),42);
(*sp)++;
std::cout << "sp: " << *sp << '\n'; // 43

auto sp2 = std::allocate_shared<Point2d>(std::allocator<int>(), -1.0, 2.0);
(*sp2).print(); // (-1, 2)

// Use a Boost pool allocator
auto sp3 = std::allocate_shared<Point2d> (boost::pool_allocator<Point2d>(),
14.45,28.45);
(*sp3).print(); // (14.45,28.45)
```

There are four overloaded versions of the function `reset()` that give up ownership by a shared pointer in some way:

- Give up ownership and reset to an empty shared pointer.
- Give up ownership and reinitialise the pointer (with default and user-defined deleters).
- Give up ownership and reinitialise the pointer using a memory allocator and a user-defined deleter.

We can determine if a shared pointer `sp` is the only owner of a resource by calling the predicate `unique()` (which is semantically equivalent to `sp.use_count() == 1`). We now give some examples on how to reset a shared pointer:

```
// Reset
std::cout << "Reset\n";
SP<value_type> sp1(new value_type(148.413));
SP<value_type> sp2(sp1);
SP<value_type> sp3(sp2);

std::cout << "sp3 shared # " << sp3.use_count() << '\n';  // 3

SP<value_type> sp4(new value_type(42.0));
SP<value_type> sp5(sp4);

std::cout << "sp5 shared # " << sp5.use_count() << '\n';  // 2

sp3.reset();
std::cout << "sp3 shared # " << sp3.use_count() << '\n';  // 0
std::cout << "sp2 shared # " << sp2.use_count() << '\n';  // 2

sp3.reset(new value_type(3.1415));
std::cout << "sp3 shared # " << sp3.use_count() << '\n';  // 1
std::cout << "sp2 shared # " << sp2.use_count() << '\n';  // 2

sp2.reset(new value_type(3.1415), Deleter<value_type>());
std::cout << "sp2 shared # " << sp2.use_count() << '\n';  // 1

std::cout << "sp2 sole owner? " << std::boolalpha << sp2.unique() << '\n';
// true
```

2.3.2 Class `std::unique_ptr`

Whereas `std::shared_ptr` allows a resource to be shared among several shared pointers, in the case of `std::unique_ptr` there is only one transferable owner of a resource. In this case we speak of *exclusive* or *strict ownership*. Its main added value is in avoiding *resource leaks* (for example, missing calls to `delete` when using raw pointers) and for this reason it can be called an *exception-safe* pointer. Its main member functions are:

- Constructors (similar to those in `std::shared_ptr`).
- Assign a unique pointer.

- Release; return a pointer to the resource and release ownership.
- Reset; replace the resource.
- Operator overloading (`==`, `!=`, `<` and other comparison operators).

The interface is similar to that of `std::shared_ptr` which means that most of the code will be easy to understand. Finally, `std::unique_ptr` succeeds `auto_ptr`, the latter being considered deprecated.

Our first example entails creating a unique pointer in a scope. Under normal circumstances when the pointer goes out of scope the corresponding resource is cleaned up but in this case we (artificially) throw an exception before the end of the scope. What happens? When we run the code we see that the resulting exception is caught *and* the resource is automatically destroyed:

```
template <typename T>
   using UP = std::unique_ptr<T>;

try
{
   // Unique pointers

   // Stored lambda function as deleter
   auto deleter = [](value_type* p)
            { std::cout << "bye, bye unique pointer\n"; delete p; };
   UP<value_type> sp32(new value_type(148.413), deleter);

   throw - 1;
}
catch(int& n)
{
   std::cout << "error but memory is cleaned up\n";
}
```

This code also works when we use shared pointers instead of unique pointers. In C++14 we can create unique pointers using `std::make_unique()` for non-array types:

```
// Other examples with unique pointers
// More efficient ways to construct unique pointers
auto up = std::make_unique<int>(42);
(*up)++;
std::cout << "up: " << *up << '\n';        // 43

auto up2 = std::make_unique<Point2d>(-1.0, 2.0);
(*up2).print();             // (-1, 2)

auto up3 = std::make_unique<Point2d>();
(*up3).print();             // (0,0)
```

Finally, we can *reset* a unique pointer (as we saw with shared pointers) and we can also *release ownership* and give it back to the caller. We show these functions by way of the following user-defined type that models two-dimensional points:

```
struct Point2d
{
   double x, y;
   Point2d() : x(0.0), y(0.0) {}
   Point2d(double xVal, double yVal) : x(xVal), y(yVal) {}
   void print() const {std::cout << "(" << x << "," << y << ")\n";}
   ~Point2d() { std::cout << " point destroyed\n"; }

};

// Reset of unique pointers
UP<value_type> up1(new value_type(148.413));
up1.reset();
assert(up1 == nullptr);
// std::cout << "reset: " << *up1 << '\n';

up1.reset(new value_type(3.1415));
std::cout << "reset: " << *up1 << '\n';
// Give ownership back to caller without calling deleter
std::cout << "Release unique pointer\n";
auto up = std::make_unique<Point2d>(42.0, 44.5);
Point2d* fp = up.release();

assert(up.get() == nullptr);
std::cout << "No longer owned by unique_ptr...\n";

(*fp).print();

delete fp; // Destructor of Point2d called
```

2.3.3 `std::weak_ptr`

The third smart pointer class holds a non-owning (*weak*) reference to an object (resource) that is managed by a shared pointer. It must be converted to a shared pointer in order to access the resource. It is a helper class to `std::shared_ptr` and it is needed when the latter's behaviour does not work as intended, namely:

- Resolving *cyclical dependencies* between shared pointers: dependencies that occur when two objects refer to each other using shared pointers. We cannot release the objects because each has a use count of 1. It is like a *deadlock*.
- Situations in which you wish to share an object but you do not own it. In this case we define a reference to a resource and that reference outlives the resource.

The class `std::weak_ptr` is used in both of the above cases. Sharing is allowed but ownership is not required. The operations in `std::weak_ptr` can be characterised as follows:

- Constructors (default, from a shared pointer, from a weak pointer).
- Assignment operators.
- Swap two weak pointers.
- Reset a weak pointer.
- Check if a managed object (resource) has expired.
- Create a shared pointer from a weak pointer by *locking* it. The shared pointer will share ownership if the weak pointer has not expired.

Some examples are:

```
// Create a default weak pointer
std::weak_ptr<double> wp;
std::cout << "Expired wp? " << wp.expired() << '\n';     // true

// Create a weak pointer from a shared pointer
std::shared_ptr<double> sp(new double (3.1415));
std::cout << "Reference count: " << sp.use_count() << std::endl;  //1

// Assign weak pointer to shared pointer
wp = sp;
std::cout << "Reference count: " << sp.use_count() << std::endl;  //1

std::weak_ptr<double> wp2(sp);
std::cout << "Reference count: " << sp.use_count() << std::endl;  //1

wp = sp;
std::shared_ptr<double> sp2(wp);
std::cout << "Reference count, sp2: " << sp2.use_count();         //2
std::cout << std::boolalpha << "Expired wp? " << wp.expired()     // false
std::shared_ptr<double> sp3 = wp.lock();
std::cout << "Reference count: " << sp3.use_count() << std::endl; //3
std::cout << "Reference count: " << sp.use_count() << std::endl;  //3

// Event notification (Observer) pattern and weak pointers
std::shared_ptr<double> spA(new double (3.1415));

std::weak_ptr<double> wA(spA);
std::weak_ptr<double> wB(spA);

spA.reset();
std::cout << "wA expired: " << wA.expired() << std::endl;
std::cout << "wB expired: " << wB.expired() << std::endl;
```

2.3.4 Should We Use Smart Pointers and When?

It is clear that smart pointers are a big improvement on raw pointers. Their use promotes the reliability of code in general but at what cost? For example, a shared pointer object is a wrapper for an ordinary pointer as it contains both this pointer and a reference counter that is shared by all shared pointers that refer to the same object. The situation becomes even more complicated

if we use weak pointers because they also need another counter. These features may hinder certain compiler optimisations.

For unique pointers, there is no run-time penalty when compared to raw pointers. Unique pointers are typically used for *factory* objects and object *engines* that create objects in a given scope. When application objects have been constructed, the factory objects are cleaned up by going out of scope. For problems that cannot be directly resolved in C++11 it is possible to resort to the *Boost Smart Pointer* library that has more smart pointer classes than in C++11 (see Demming and Duffy, 2010). We note that shared pointers are not thread-safe. This fact will have major consequences when porting single-threaded code to multithreaded code. There are functions in C++11 to perform *atomic* and *thread-safe* operations on shared pointers that we shall discuss in Chapter 28. In short, you need to determine what smart pointers can and cannot do and what the consequences are when you use them in code. Issues such as performance and maintainability are central. A special project would be to upgrade *legacy code* that uses raw pointers to code that uses smart pointers.

2.4 EXTENDED EXAMPLES OF SMART POINTERS USAGE

In the previous sections we gave a discussion of the new pointer classes in C++. The next question to answer is how they represent an improvement on raw pointers. To this end, we discuss how to design user-defined assemblies and aggregations containing embedded pointers and we re-engineer the popular *Factory Method* design pattern (GOF, 1995) that uses raw pointers.

2.4.1 Classes with Embedded Pointers

A common occurrence in software development is when we create *composite* and *whole–part* objects consisting of collections whose components are of built-in or user-defined types (a discussion can be found in POSA, 1996). Our interest here is the variant in which the components can be accessed by clients external to the container. This is the *shared parts* variant. They are pointers that have been initialised elsewhere, possibly in dedicated factories. In a sense, this property breaks encapsulation. Some of the issues and attention points when using raw pointers to implement the components are:

- Which component(s) are responsible for the lifetime management of the heap-based container components? In GOF (1995), for example, a tactic is that composites are responsible for deleting their children.
- A component's resource may be destroyed by one of its owners, thereby invalidating the component and thus delivering a null pointer.
- Implementing code in a composite to explicitly manage component lifetime leads to a class with more than one major responsibility, leading to potential maintenance problems.
- The code is not thread-safe (because shared pointers are not thread-safe). Non-deterministic behaviour and *race conditions* are bound to occur when multiple independent threads access shared resources.

Smart pointers in C++ resolve some of these issues. An example of use is (notice that we use the user-defined delete from Section 2.3.1):

```
template <typename T>
   using SP = std::shared_ptr<T>;

class CompositeContainer
{ // New style class for lists of embedded pointers
private:
   std::list<SP<int>> data;
public:
   CompositeContainer() : data(std::list<SP<int>>()) {}

   void add(int* elem)
   {
      data.push_back(SP<int>(elem, Deleter<int>()));
      //data.push_back(SP<int>(elem));
   }

   void add(const SP<int>& elem)
   {
      data.push_back(elem);
   }

};

// Composition
std::cout << "Composition\n";
CompositeContainer con;
int* i = new int(10);
auto e1 = SP<int>(new int(13), Deleter<int>());

con.add(i);
con.add(e1);
```

This code gives an indication of how to create robust whole–part collections from the designs in POSA (1996).

2.4.2 Re-engineering Object-Oriented Design Patterns

The design style of the object-oriented design patterns as described in GOF (1995) is based on a combination of the following tactics:

- Class hierarchies and subtype polymorphism.
- Use of raw pointers; no distinction is made between short-lived and long-lived objects.
- Using *Composition* to design client classes that access the interfaces of server objects using raw pointers to base class.

We show how to resolve these issues by using shared and unique pointers. To this end, we take the *Factory Method* pattern as exemplar. We take the quotation from Wikipedia:

> *In class-based programming, the factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. This is done by creating*

objects by calling a factory method—either specified in an interface and implemented by child classes, or implemented in a base class and optionally overridden by derived classes—rather than by calling a constructor.

We take an example of how to apply this pattern to the construction of one-factor option payoffs (we mention that this is not the only way to create this functionality) and the example is given mainly for motivation. We model the payoffs as a hierarchy of function objects in combination with subtype polymorphism:

```
// Traditional class hierarchy
class Payoff
{ // Function object
public:
   virtual double operator ()(double S) const = 0;
};


class CallPayoff : public Payoff
{
private:
   double K;
public:
   CallPayoff(double strike) { K = strike; }
   double  operator ()(double S) const
   {
      return std::max<double>(S - K, 0);
   }
};

class PutPayoff : public Payoff
{
private:
   double K;
public:
   PutPayoff(double strike) { K = strike; }
   double  operator ()(double S) const
   {
       return std::max<double>(K - S, 0);
   }
};
```

Other kinds of payoffs may need to be added in the future. For this reason we create clients that only refer to the base class `Payoff`. Furthermore, specialised factory objects create payoffs based on a certain choice. To this end, the factory class hierarchy is (notice that they deliver shared pointers to payoffs in their interface):

```
class PayoffCreator
{
public:
```

```
    // C++11 way to define 'nonusable'
    PayoffCreator(const PayoffCreator& orig) = delete;
    PayoffCreator& operator=(const PayoffCreator& orig )= delete;

public:
    PayoffCreator() = default;        // C++11 syntax
    virtual ~PayoffCreator()=default; // Destructor

    // Functions to create an object
    virtual std::shared_ptr<Payoff> CreatePayoff(double K) = 0;
};

class CallPayoffCreator : public PayoffCreator
{
public:

    // C++11 way to define 'nonusable'
    CallPayoffCreator(const CallPayoffCreator& orig) = delete;
    CallPayoffCreator& operator=(const CallPayoffCreator& orig) = delete;

public:
    CallPayoffCreator() = default;
    virtual ~CallPayoffCreator() = default; // Destructor

    // Functions to create an object
    virtual std::shared_ptr<Payoff> CreatePayoff(double K)
    {
        return std::shared_ptr<Payoff>(new CallPayoff(K));
    }
};

class PutPayoffCreator : public PayoffCreator
{
public:

    // C++11 way to define 'nonusable'
    PutPayoffCreator(const PutPayoffCreator& orig) = delete;
    PutPayoffCreator& operator=(const PutPayoffCreator& orig) = delete;

public:
    PutPayoffCreator()=default;
    virtual ~PutPayoffCreator()=default; // Destructor

    // Functions to create an object
    virtual std::shared_ptr<Payoff> CreatePayoff(double K)
    {
        return std::shared_ptr<Payoff>(new PutPayoff(K));
    }

};
```

In this case we have chosen (for convenience) a one-to-one correspondence between a payoff class and its corresponding factory class. Finally, we show how the above class hierarchies are used in configuration code:

```
int main()
{
    // Main steps:
    // 1. Choose the specific factory (unique pointer)
    // 2. Create the specific payoff (shared pointer)
    // 3. Use the payoff (for example, in an option pricer)

    // 1.
    int choice; std::cout << "1: Call, 2: Put: "; std::cin >> choice;

    std::unique_ptr<PayoffCreator> creator;
    if (1 == choice)
       creator.reset(new CallPayoffCreator());
    else
       creator.reset(new PutPayoffCreator());

    // 2.
    double K = 100.0;
    auto payoff = creator->CreatePayoff(K);

    // 3. Compute the payoff for a given S
    double S = 90.0;
    std::cout << "Payoff: " << (*payoff)(S) << '\n';

    return 0;
}
```

Here we see that the factory is a unique pointer which is probably a more nuanced solution than using raw or shared pointers. This is because we wish to have exclusive ownership and the factory is automatically destroyed when it is no longer needed even when an exception occurs.

This example is the tip of the iceberg as it were and the underlying concepts can be applied to other design patterns.

2.5 MOVE SEMANTICS AND RVALUE REFERENCES

Move semantics is a new feature since C++11. It allows the compiler to replace expensive copy operations and the creation of temporary objects by less expensive moves. In general, we can define *move constructors* and *move assignment operators* in addition to (or instead of) copy constructors and copy assignment operators.

2.5.1 A Quick Overview of Value Categories

In general, a *C++ expression* is a name that encompasses operators with their operands, literals and variable names. An expression is characterised by two orthogonal properties, namely its

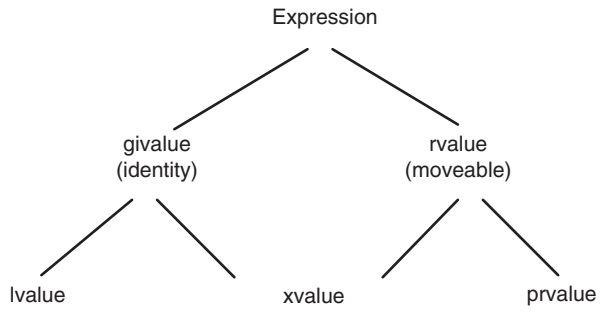


FIGURE 2.1 Expression categories

type and its *value category*. We also note that an expression may have identity which means that it is possible to compare expressions (for example, whether they refer to the same entity) or compare the addresses of the objects or the functions that they identify.

There are three primary value categories that represent the combination of identity and whether it is legal to move from the expression's value:

- *xvalue*: has identity and it can be moved from.
- *lvalue*: has identity and it cannot be moved from.
- *prvalue*: does not have identity and it can be moved from. It is a *pure value expression*.

Expressions that have identity are called *glvalue* expressions (generalised *lvalues*). Thus, *xvalue* and *lvalue* are *glvalues*. The expressions that can be moved from are called *rvalue* expressions. Both *prvalue* and *xvalue* are *rvalue* expressions. An *rvalue* is an anonymous temporary that can appear only on the right-hand side of an assignment. The category taxonomy is shown in Figure 2.1.

The operation `std::move` does not actually move anything but instead it converts its arguments into an *rvalue reference* which is a type that is declared with two ampersands `&&`. This is a new type and it stands for an *rvalue* that can be modified. The rationale is that it is a temporary object that is no longer needed. This implies that its contents or its resources can be stolen.

The following move operations are supported in C++:

- Move a range in a container to another container.
- Move a range in a container to another container, backwards.
- Move a general type to another type.

2.5.2 Why Some Classes Need Move Semantics

Move semantics offer the developer a choice: instead of expensive copies (using copy constructors and copy assignment operators) we can use the new move constructor and move assignment operator. In some cases we wish to create classes whose copy constructor cannot be invoked, for example. In C++03 this requirement is realised by defining the copy constructor to be private and giving it an empty body. This leads to various problems and C++11 resolves

these problems by defining the constructor as a *deleted* body function (which we shall discuss in detail in Chapter 7, Section 7.3.2). An example is:

```
class F
{
public:
   // Support default operations OK
   F() = default;

   // Move ctor and move assignment are defined
   F(F&&) = default;
   F& operator = (F&&) = default;

   F(const F&) =  delete;
   F& operator = (const F&) = delete;

   virtual ~F() = default;
};
```

In the above case we also see how to define a move constructor and a move assignment operator. In this case we use the keyword `default` to let the compiler generate the body of the given member function.

Examples of types that typically need move semantics and that must suppress copy operations are file descriptors, database connections, TCP sockets, I/O streams, running threads and asynchronous futures (in multitasking) and of `std::unique_ptr`.

This discussion brings us to a review of the *Rule of Three* which states that if a class defines one or more of the following member functions then it should implement all three:

- Destructor.
- Copy constructor.
- Copy assignment operator.

These functions are so-called *special member functions*. We can broaden the list to provide the *Rule of Five* consisting of the three above functions in addition to:

- Move constructor.
- Move assignment operator.

Concluding, we could explicitly specify the five members as default in a base class that is intended for polymorphic use. Then the destructor must be virtual.

```
class Base
{
public:
    Base(const Base&) = default;
    Base(Base&&) = default;
    Base& operator=(const Base&) = default;
    Base& operator=(Base&&) = default;
    virtual ~Base() = default;
};
```

2.5.3 Move Semantics and Performance

Many applications involve shovelling/copying vectors and arrays into other containers. In many cases, the input argument is no longer needed after the copy has taken place. In such cases we may prefer to move the input vector to the target vector (incidentally, the input vector is unspecified and it has size zero after the move). The approach can lead to significant performance gains because the elements are moved by switching some pointers instead of copying container elements. In short, it is optimal to use the move constructor when we no longer need to copy the contents of a container.

We take an example of creating a vector and comparing the copy operation with the three versions of move that we mentioned in Section 2.5.1. We monitor run-time performance using the stopwatch class that we shall introduce in Chapter 10 (in practice, you can use your own favourite timer if you prefer):

```
// Copy vs move an array performance
std::size_t N = 1'000'000;
std::vector<double> v(N);
std::iota(std::begin(v), std::end(v), 1);
std::vector<double> v2;
StopWatch<> sw;
sw.Start();

// Option A (copy elements, linear complexity)
std::copy(std::begin(v), std::end(v), std::back_inserter(v2));

sw.Stop();
std::cout << "Elapsed time copy : " << sw.GetTime() << '\n';
assert(std::equal(std::begin(v), std::end(v), std::begin(v2)));

v2.clear();
sw.Start();

// Option B: Move a range of elements
std::move(std::begin(v), std::end(v), std::back_inserter(v2));

// Option C: Move constructor
v2 = std::move(v);

sw.Stop();
std::cout << "Elapsed time move: " << sw.GetTime() << '\n';
assert(std::equal(std::begin(v), std::end(v), std::begin(v2)));
std::cout << "Size of input vector: " << v.size() << '\n';

v2.clear();
v = std::vector<double> (N);
std::iota(std::begin(v), std::end(v), 1);
v2.resize(v.size());
sw.Start();
```

```
// Option D: move elements to v2, starting at end(v2)
std::move_backward(std::begin(v), std::end(v), std::end(v2));

sw.Stop();
std::cout << "Elapsed time move backward: " << sw.GetTime() << '\n';

assert(std::equal(std::begin(v), std::end(v), std::begin(v2)));
```

We have executed a number of basic tests with large arrays (typically having a million to one hundred million elements). In general, option C is fastest (30 times faster than option A), then option D (10 times faster than option A), then option B (5 times faster than option A) based on the small set of experiments that we tested. We advise you to carry out your own experiments, not only with vectors but with other containers such as lists, for example. The numbers may differ in your applications.

We conclude this section with the remark that fixed-size arrays can also be move constructed:

```
const int N = 100;
std::array <std::string, N> arr1;

auto arr2 = std::move(arr1);
```

In later chapters we shall show how to improve the run-time performance of numerical algorithms (in particular, finite difference and Monte Carlo methods).

2.5.4 Move Semantics and Shared Pointers

There are several use cases in which smart pointers can be moved:

- Move a shared pointer to a shared pointer.
- Move a unique pointer to a shared pointer.
- Assign an `auto_ptr` to a shared pointer.
- Move a unique pointer to a unique pointer.

Some code examples are:

```
// A. From auto_ptr to shared ptr; upgrading legacy code
// Define auto_ptr pointers instead of raw pointers
std::auto_ptr <double> ap(new double(1.0));
// Dereference
*ap = 2.0;

std::shared_ptr<double> sp(std::move(ap));
// std::cout << "ap: " << *ap << '\n'; CRASH
std::cout << "sp: " << *sp << '\n'; // 2
```

2.6 OTHER BITS AND PIECES: USABILITY ENHANCEMENTS

We now give a discussion of a number of useful features in C++. They improve the readability and robustness of code. In other cases they represent structural solutions to workarounds that were used in previous versions of C++ to resolve some issues. You may need to use these new features in your code.

2.6.1 Type Alias and Alias Templates

Developers are familiar with the `typedef` specifier that allows aliases to be defined for existing types. The specifier cannot be used to change the meaning of an existing type name and hence it does not represent the declaration of a new type. A `typedef` has effect in the scope in which it is visible. An example is:

```
typedef std::map<std::string, double> Dictionary;
```

The use of an alias promotes code readability and maintainability. Unfortunately, `typedef` cannot be used for class templates and hence the following code will not compile:

```
/*
template <typename Key, typename Value>
   typename std::map<Key, Value> DictionaryII;

template <typename Value>
   typename std::map<std::string, Value> DictionaryIII;
*/
```

But there is hope! C++11 offers the possibility to define an alias for class templates and non-template classes alike. The term is sometimes called `alias template` or `template typedef`. The above example can now be formulated as follows:

```
using Dictionary = std::map<std::string, double>;

template <typename Key, typename Value>
   usage DictionaryII = std::map<Key, Value>;

template <typename Value>
   using DictionaryIII = std::map<std::string, Value>;
```

We can also use this technique in combination with user-defined types with long names and with types having many template parameters in order to make it easier to work with them. Let us take the simple example:

```
template <typename T1, typename T2>
   class C
```

```
{
private:
public:
   T1 t1;
   T2 t2;
public:
   C() : t1(T1()), t2(T2()) {}
   C(const T1& t1Val, const T2& t2Val) : t1(t1Val), t2(t2Val) {}

   void print() const
   {
      // T1 and T2 must support <<
      std::cout << t1 << ", " << t2 << '\n';
   }
   // other code here
};
```

We can imagine that client code might wish to use this class in different ways and to this end we can create various aliases, for example:

```
template <typename T1, typename T2>
   using C1 = C<T1, T2>;

// 'Nested' alias
template <typename T>
   using Vec = std::vector<T>;

template <typename T1, typename T2>
   using C2 = C<T1, Vec<T2>>;

template <typename T>
   using C3 = C<T, int>;

using C4 = C<double, int>;
// Using alias template
C4 c4; // 0,0
c4.print();

float f = 1.0F;
int j = 3;
C3<float> c3(f, j);
c3.print(); // 1,3

// Nested alias in class C2 for readability
Vec<double> v(100);
int n = 10;
C2<int, double> c2(n, v);
C<int, std::vector<double>> c2_1(n, v);

C1<int, int> c1;
C<int, int> c(1, 2);
```

Finally, we can create classes that are composed of these aliases:

```
template <typename T>
   using SP = std::shared_ptr<T>;

// Client class
template <typename T>
   class Client
{ // Composition
private:
   C <T, SP<T>> data;
public:
   Client(const T& t) : data(t, SP<T>(new T(t))) {}

   void print() const
   {
      std::cout << data.t1 << ", " << *(data.t2) << '\n';
   }
};

// Client class as a composition
Client<double> client(1.0);
client.print(); // 1,1
```

We have given a number of examples to show the usefulness of this technique. The alias template should not be confused with the well-known `using` declaration that introduces a name that is defined elsewhere.

2.6.2 Automatic Type Deduction and the `auto` Specifier

This feature allows us to declare a variable or an object without having to use its specific type. In other words, the variable is automatically deduced from its initialiser. Some examples are:

```
// auto variable
double d1 = 1.0; float f1 = 2.0f; int p1 = 10;
auto d2 = d1 + f1;
auto d3 = d1 + p1;
auto d4 = f1 + p1;
```

We could have used explicit return types but the compiler is clever and it knows what the correct type should be, as the following checks show using the *type traits* library (which we discuss in detail in Chapter 6):

```
// Example resulting type by type_traits (chapters 4 and 6)
static_assert(std::is_same<decltype(d1), double>::value, "ouch");
static_assert(std::is_same<decltype(d2), double>::value, "ouch");
static_assert(std::is_same<decltype(d3), double>::value, "ouch");
// FAIL static_assert(std::is_same<decltype(d4), double>::value, "ouch");
static_assert(std::is_same<decltype(d4), float>::value, "ouch"); / Yes
```

We can also use the `auto` specifier to make life easier when working with user-defined types:

```
// auto variable with user-defined types

using CP = std::complex <double>;
CP c(1.0, 2.0);
C<CP, CP> cA(c, c);
auto cB = cA;
static_assert(std::is_same<decltype(cB), C<CP, CP>>::value, "ouch");
```

We discuss `auto` in more detail and its relationship with other C++ features in later chapters, for example as return types of, and as input arguments to, lambda functions.

2.6.3 Range-Based `for` Loops

This feature can be used as a more readable alternative to the traditional `for` loop to iterate over a range of values. We take an example of printing the contents of a vector. The first version uses iterators and the second version uses a function object called `doit()`:

```
void printI(const std::vector<double>& arr)
{
   for (auto i = std::begin(arr); i != std::end(arr); ++i)
   {
      std::cout << *i << ",";
   }
   std::cout << '\n';
}

void doit(double d)

{ // Function object
   std::cout << d << ",";
}

void printII(const std::vector<double>& arr)
{
   // Apply the function object 'doit' to each element of container
   std::for_each(std::begin(arr), std::end(arr), doit);
   std::cout << '\n';
}
```

The range-based version is:

```
void printIII(const std::vector<double>& arr)
{
   // Range-based for loop
   for (const auto& elem : arr)    // access by const reference
```

```
    {
       std::cout << elem << ",";
    }
    std::cout << '\n';

    for (auto elem : arr)          // access by value
    {
       std::cout << elem << ",";
    }
    std::cout << '\n';

    for (auto&& elem : arr)          // access by reference
    {
       std::cout << elem << ",";
    }
    std::cout << '\n';
}
```

An example of use is (same output in all cases):

```
// For loops and vectors
const std::vector<double> arr(4, 148.413);
printI(arr);
printII(arr);
printIII(arr);
```

You may find this to be a useful feature. It is a minor C++ feature. Some developers prefer to use standard *for loops* instead of the new-fangled range-based `for` loops.

2.6.4 `nullptr`

This keyword denotes the *null pointer literal* and it can be seen as the safe alternative to the pre-C++11 macro NULL which is an implementation-dependent null pointer constant. A possible implementation is:

```
#define NULL 0
#define NULL nullptr; // since C++11
```

We take an example of a function that takes a raw pointer as input argument:

```
bool CheckNullPtr(int* i)
{
   return (i == nullptr);
}
```

We can call this function as follows:

```
// Null ptr
int* i = nullptr;
```

```
std::cout << std::boolalpha << CheckNullPtr(i) << '\n'; // true

auto i2 = std::unique_ptr<int> (new int);
std::cout << std::boolalpha << CheckNullPtr(i2.get()) << '\n';// false
```

In general, use of `nullptr` is recommended. It is a *type-safe* null pointer.

2.6.5 New Fundamental Data Types

C++11 supports a number of new character types for the *Universal Characters Set* and *Unicode* standards. Some examples are:

```
// Unicode characters
char16_t ch1;       // UTF - 16 characters
char32_t ch2;       // UTF - 32 characters
wchar_t ch3;        // Largest extended character set
```

Furthermore, new integral types are:

```
// long data type
long long v1 = 3;
unsigned long v2 = 4;

std::cout << std::numeric_limits<long long>::min() << ", "
          << std::numeric_limits<long long>::max() << '\n';
std::cout << std::numeric_limits<unsigned long long>::min() << ", "
          << std::numeric_limits<unsigned long long>::max() << '\n';
std::cout << std::numeric_limits<long>::min() << ", "
          << std::numeric_limits<long>::max() << '\n';
```

The output from this code is:

```
-9223372036854775808, 9223372036854775807
0, 18446744073709551615
-2147483648, 2147483647
```

2.6.6 Scoped and Strongly Typed Enumerations

Enumerations in C++03 are not type-safe. They are essentially integers to allow enum values of different enumeration types to be compared. In C++11 it is possible to define a *type-safe enumeration class* that does not have the shortcomings of the old-style enums. Type-safe enums cannot be compared to integers nor are they implicitly converted to integers. An example is:

```
enum class Colour
{
   RED,
   ORANGE,
```

```
    YELLOW,
    GREEN,
    BLUE = 100,
    INDIGO
};
```

How do we work with this enum class? First, we can create a colour by assigning it to a component of the enum class `Colour` and second, we must explicitly cast the component to an integer if we wish to print it:

```
Colour colour = Colour::RED;
//std::cout << colour;             // error, no implicit conversion to int
std::cout << static_cast<int>(colour) << '\n'; // 0

colour = Colour::INDIGO;
std::cout << static_cast<int>(colour) << '\n'; // 101
```

The following code will not compile:

```
// int g = GREEN; // GREEN unidentified
```

We remark that error condition values of standard exceptions are scoped enumerations.

2.6.7 The Attribute `[[deprecated]]`

In general, an *attribute* is a piece of *metadata* that is used to add custom information to code elements such as types, members, return values and functions. The concept is well developed in languages such as C# (Albahari and Albahari, 2016). In C++, attributes provide the unified standard syntax for implementation-defined language extensions such as Microsoft's `__declspec` keyword which is an extended attribute syntax for specifying storage-class information. It specifies that an instance of a given type is to be stored with a Microsoft-specific storage class and it is used when creating *dynamic link libraries* (DLLs) that are called from Excel, for example. The following code shows how it is used; we create a function `doit()`, export it into a DLL, then call the functions in the DLL from a main function:

```
#using <mscorlib.dll>

using namespace System;
using namespace System::Runtime::InteropServices;

[DllImport("C:\\VSProjects\\DLL101\\Release\\DLL101.dll")]
extern void doit();

int main()
{
    // Call the DLL function
    doit();
```

```
    return 0;
}
```

The function being called is:

```
#include <iostream>

__declspec(dllexport) void _stdcall doit()
{

    std::cout << "Hello ...";

}
```

Following this small digression we now discuss the attribute `[[deprecated[]` in C++. The description from www.cppreference.com is:

> *Indicates that the use of the name or entity declared with this attribute is allowed, but discouraged for some* reason. *This attribute is allowed in declarations of classes, typedef-names, variables, non-static data members, functions, enumerations and template specialisations. A name declared non-deprecated may be redeclared deprecated. A name declared deprecated cannot be un-deprecated by redeclaring it without this attribute.*

At the moment of writing not all cases mentioned in the above description seem to compile and for this reason we concentrate on those examples which do work for us. We give some examples of the use of this attribute which should be easy to understand:

```
// Deprecated a global funtion
[[deprecated("old stuff, use F() instead")]] void Func()
{
}

// Deprecate an enum
enum[[deprecated]] animals{
    CAT, DOG, MOUSE
};

// Deprecate a typedef
[[deprecated("Cannot use 'type'")]]
    typedef int type;

// Deprecate a template specialisation
template <typename T> class templ;

template <>
    class[[deprecated]] templ<int> {};
```

We see that if you call a deprecated function or try to access deprecated data then the code will not compile and it will issue an error message.

We conclude this section with a concocted example of a class with a mixture of deprecated and non-deprecated members:

```
struct[[deprecated("old skule class")]] C
{
   C() : data(1) {}

   [[deprecated]] int data;

   [[deprecated]] void print() { std::cout << "Deprecated\n"; }
   void NewPrint() const { std::cout << "Not deprecated\n"; }
};
```

An example shows which code compiles and which does not:

```
C c;
// c.data = 10;
//c.print();
c.NewPrint();
```

There are a number of other attributes defined for C++17 but a discussion is outside the scope of this book. We are not yet convinced of the usefulness of this feature.

2.6.8 Digit Separators

This functionality is a useful way to represent numbers in a more readable form. An example is:

```
int x = 1'000'000; // This is a digit separator
```

There is not much more that we can say about this feature. It can be seen as *syntactic sugaring*.

2.6.9 Unrestricted Unions

A *union* is a variable that may hold objects of different types and of different sizes at different times. The compiler keeps track of size and alignment requirements. Thus, unions provide a way to manipulate different kinds of data in a single area of storage (Kernighan and Ritchie, 1988).

C++03 placed restrictions on the types of objects that can be members of a union. For example, they may not contain objects that have a non-trivial constructor or destructor. C++11 does not have these restrictions. We take an example:

```
struct Point
{
   double x, y;
};
```

```
struct Line
{
   Point p1, p2;
};

struct Circle
{
   Point p1;
   double r;
};
```

We take an example of a class with an embedded *anonymous union*. The class is:

```
struct CADBucket
{ // This concept is used in AutoCAD

   // Tag to allow switching
   enum { PO, LI, CI } tag;

   union
   { // Anonymous, default access is public
      Point p;
      Line l;
      Circle c;
   };
};
```

How do we work with this class, for example how do we print its instances depending on the value of its tag?

```
void print(const CADBucket& cb)
{
   switch (cb.tag)
   {
      case CADBucket::PO: print(cb.p); std::cout << '\n'; break;
      case CADBucket::LI: print(cb.l); std::cout << '\n'; break;
      case CADBucket::CI: print(cb.c); std::cout << '\n'; break;
   }
}
```

The following print functions are used:

```
void print(const Point& pt)
{
   std::cout << "(" << pt.x << "," << pt.y << ") " ;
}

void print(const Line& l)
{
   print(l.p1); print(l.p2);
```

```
}

void print(const Circle& c)
{
   print(c.centre); std::cout << c.radius;
}
```

An example of use is:

```
   Point pA{ 1.0, 2.0 };
   Point pB{ 3.0, -4.9 };
   Line lA{ pA, pB };
   Circle cA{ pB, 2.0 };

CADBucket cb = {CADBucket::PO, pA};
print(cb);

// Switch to line segment
cb.tag = CADBucket::LI;
cb.l = lA;
print(cb);

// Switch to circle
cb.tag = CADBucket::CI;
cb.c = cA;
print(cb);
```

This code works but it is not very flexible; if we define a new shape (for example, a rectangle or a polyline (collection of `Point` instances)) then some changes are needed and the code must be edited and recompiled. We now discuss a more flexible solution.

2.6.10 `std::variant` (C++17) and `boost::variant`

We have seen that unions are not very flexible. However, they are used in many production applications. It seems that they will be superseded in C++17 by `std::variant` which can then replace unions and union-like classes. Since we do not have a C++17 compiler at the moment of writing, we resort to a discussion of how `boost::variant` works (Demming and Duffy, 2010). From the Boost online documentation:

> *The variant class template is a safe, generic, stack-based discriminated union container, offering a simple solution for manipulating an object from a heterogeneous set of types in a uniform manner. Whereas standard containers such as `std::vector` may be thought of as 'multi-value, single type,' variant is 'multi-type, single value.'*

The main uses of variants are:

- Type-safe storage and retrieval of user-specified sets of types.
- Storing heterogeneous types in STL containers.

- Compile-time checked visitation of variants (using the *Visitor* pattern).
- Efficient, stack-based storage for variants.

We give a short introduction using the previous CAD example. The two use cases that we consider here are:

- Creating a variant.
- *Type-safe visitation* of a variant; in this case we replace the user-defined switch (see example in Section 2.6.9 again) to decide which member of a struct to access by a class that implements the compile-time *Visitor* pattern (GOF, 1995).

We take an example of a variant consisting of three members:

```
#include <boost/variant.hpp> // Includes all other header files

using boost::variant;

// Some ways to construct a variant
variant<long, std::string, Point> myVariant;
// Give some values
myVariant = 24;

myVariant = std::string("It's amazing");
myVariant = Point(3.0, 4.0);
```

For completeness, we include the definition for class `Point`:

```
struct Point
{
   double x, y;
   Point() : x(0.0), y(0.0) {}
   Point(double xVal, double yVal) : x(xVal), y(yVal) {}

};

void print(const Point& pt)
{
   std::cout << "(" << pt.x << "," << pt.y << ") ";
}
```

We can now get the current value in a variant by calling the `get()` function:

```
// Try to get the value out of the variant
Point ptA;
try
{
   ptA = boost::get<Point>(myVariant);
}
catch (boost::bad_get& err)
```

```
{
   std::cout << "Error: " << err.what() << std::endl;
}

   std::cout << "Value got from Variant: "; print(ptA);
```

We see that this code is not type-safe. In order to make the code type-safe we use the `boost::apply_visitor::apply_visitor` function that allows us to apply compile-time checked type-safe application of `boost::static_visitor` to the contents of the variant, thus ensuring that all types are handled by the visitor. To this end, we consider how to use this functionality to print the fields of the variant in a type-safe manner using a visitor:

```
class Print_Visitor : public boost::static_visitor<void>
{ // A visitor with function object 'look-alike'

public:
   void operator () (long val) const
         {std::cout << "Long: " << val << '\n';}

   void operator () (std::string& val)
         {std::cout << "String: " << val << '\n';}

   void operator () (Point& val)
         {std::cout << "Point: "; print(val); }
};
```

An example of use is:

```
variant<long, std::string, Point> myVariant;  // default long

Print_Visitor vis;

// Give some values; visit each component
myVariant = 24;
boost::apply_visitor(vis, myVariant);

myVariant = std::string("It's amazing");
boost::apply_visitor(vis, myVariant);

myVariant = Point(3.0, 4.0);
boost::apply_visitor(vis, myVariant);
```

The output produced is:

```
Long: 24
String: It's amazing
Point: (3,4)
```

We remark that it is possible to create visitors with non-`void` return type in Boost, a feature not shared by the traditional *Visitor* design patterns in GOF (1995).

2.7 SUMMARY AND CONCLUSIONS

In this chapter we introduced fundamental syntax and new features in C++ that promote the robustness and reliability of code. We took a focused approach in order to give a compact description of each piece of syntax and functionality and then we gave a number of examples. The main topics were smart pointers, move semantics, unions and variants.

2.8 EXERCISES AND PROJECTS

1. (First Encounters with Smart Pointers: *Unique Pointers*)
Consider the following code that uses raw pointers:

```
{ // Block with raw pointer lifecycle

  double* d = new double (1.0);
  Point* pt = new Point(1.0, 2.0);    // Two-d Point class

  // Dereference and call member functions
  *d = 2.0;
  (*pt).X(3.0);         // Modify x coordinate
  (*pt).Y(3.0);         // Modify y coordinate

  // Can use operators for pointer operations
  pt->X(3.0);           // Modify x coordinate
  pt->Y(3.0);           // Modify y coordinate

  // Explicitly clean up memory (if you have not forgotten)
  delete d;
  delete pt;
}
```

Answer the following questions:

a) Type, run and test the above code. Introduce a *try-catch* block in which memory is allocated and directly afterwards an exception is thrown. You need to throw an exception in the body of the code. Since the code is not *re-entrant*, the memory is not reclaimed and hence it introduces a *memory leak* (in more general cases it would be a *resource leak*).

b) Now port the above code by replacing raw pointers by `std::unique_ptr`. Run the code. Are there memory leaks now?

c) Experiment with the following: initialising two unique pointers to the same pointer, assigning a unique pointer to another unique pointer and *transferring ownership* from one unique pointer to another unique pointer.

d) Use *alias template* (template typedef) to make the code more readable.

2. (Shared Pointers)

The objective of this exercise is to show *shared ownership* using smart pointers in C++.

Create two classes `C1` and `C2` that share a common heap-based object `d` as data member:

```
std::shared_ptr<double> d;
```

Answer the following questions:

a) Create the code for the classes `C1` and `C2` each of which contains the shared object `d` above, for example:

```
class C1
{
private:
      //double* d; OLD WAY
      std::shared_ptr<double> d;
public:
      C1(std::shared_ptr<double> value) : d(value) {}
      virtual ~C1() { cout << "\nC1 destructor\n";}
      void print() const { cout << "Value " << *d; }
};
```

The interface for class `C2` is similar to that of `C1`.

b) Create instances of these classes in scopes so that you can see that resources are being automatically released when no longer needed. To this end, employ the member function `use_count()` to keep track of the number of shared owners. This number should be equal to 0 when the program exits.

c) Carry out the same exercise as in steps a) and b) but with a user-defined type as shared data:

```
std::shared_ptr<Point> p;
```

d) Now extend the code in parts a) to c) to include the following operations on shared pointers: assigning, copying and comparing two shared pointers `sp1` and `sp2`. Furthermore, test the following features (some research needed here):

- Transfer ownership from `sp1` to `sp2`.
- Determine if a shared pointer is the only owner of a resource.
- Swap `sp1` and `sp2`.
- Give up ownership and reinitialise the shared pointer as being empty.

3. (The Smart Pointer `std::auto_ptr`)

This pointer type is deprecated in C++11. Consider the following code:

```
using std::auto_ptr;

// Define auto_ptr pointers instead of raw pointers
std::auto_ptr <double> d(new double (1.0));
```

```
// Dereference
*d = 2.0;

// Change ownership of auto_ptr
// (after assignment, d is undefined)
auto_ptr <double> d2 = d;
*d2 = 3.0;

cout << "Auto values: " << *d.get() << ", " << *d2.get();
```

Answer the following questions:

a) Type the above code and run it. Why does it give a run-time error?

b) Port the code to code that uses `std::unique_ptr`. Run and test the new code.

4. (Smart Pointers and STL Algorithms)

In this exercise we create a simple example of STL containers whose members are smart pointers.

To this end, consider the following class hierarchy:

```
class Base
{ // Base class
private:

public:
      Base() { }
      virtual void print() const = 0;

protected:
      virtual ~Base() { cout << "Base destructor\n\n"; }
};

class Derived : public Base
{ // Derived class
private:

public:
      Derived() : Base() { }
      ~Derived() { cout << "Derived destructor\n"; }
      void print() const { cout << "derived object\n";}
};
```

Answer the following questions:

a) Create a list whose elements are smart pointers to `Base`.

b) Create a factory function to create instances of class `Derived` and then add these instances to the list.

c) Test the functionality (try to break the code) and check that there are no memory leaks.

5. (Custom Deleter)

Shared and unique pointers support deleters. A *deleter* is a callable object that executes some code before an object goes out of scope. A deleter can be seen as a kind of

callback function. We first give a simple example to show what we mean: we create two-dimensional points as smart pointers. Just before a point goes out of scope a callback function will be called:

```
auto deleter = [](Point* pt)-> void
               { std::cout << "Bye:" << *pt; };
```

The corresponding code is:

```
using SmartPoint = std::shared_ptr<Point>;
SmartPoint p1(new Point(), deleter);
```

We now discuss the exercise. To this end, answer the following questions:

a) The goal of this exercise is to open a file, write some records to the file and then close it when we are finished. Under normal circumstances we are able to close the file. However, if an exception occurs before the file has been closed the file pointer will remain open and hence it cannot be accessed. In order to ensure *exception safety* we employ a shared pointer with a deleter in the constructor, for example by using a function object:

```
std::shared_ptr<FILE> mySharedFile(myFile, FileFinalizer());
```

where `FileFinalizer` is a function object. Similar to the deleter that we gave at the beginning of the exercise.

b) Create a free function and a stored lambda function that also play the role of custom deleters for this problem.

c) Test the code for the three kinds of deleter function (function object, free function, lambda function).

d) Create a small loop in which records are added to the file; throw an exception at some stage in the loop, catch the exception and then open the file again. Does it work?

6. (Weak Pointers)

A *weak pointer* is an observer of a shared pointer. It is useful as a way to avoid dangling pointers and also when we wish to use shared resources without assuming ownership.

Answer the following questions:

a) Create a shared pointer, print its use count and then create a weak pointer that observes it. Print the use count again. What are the values?

b) Assign a weak pointer to a shared pointer and check that the weak pointer is not empty.

c) Assign a weak pointer to another weak pointer; assign a weak pointer to a shared pointer. What is the use count in both cases?

7. (Alias Template and its Advantages Compared to `typedef`)

The keyword `typedef` does not work with templates (at least not directly) and we need to do a lot of contortions when working with template classes. To this end, we create a class that is composed of a container:

```
template <typename T>
   class Client
```

```
{ // An example of Composition
private:
   typename Storage<T>::type data; // typename mandatory
public:
   Client(int n, const T& val) : data(n, val) {}

   void print() const
   {
      std::for_each(data.begin(), data.end(), [](const T& n)
                    { std::cout << n << ","; });
      std::cout << '\n';
   }
};
```

where the storage ADT is:

```
// C++03 approach
// Data storage types
template <typename T> struct Storage
{

   // Possible ADTs and their memory allocators
   // typedef std::vector<T, CustomAllocator<T>> type;
   // typedef std::vector<T, std::allocator<T>> type;

   typedef std::list<T, std::allocator<T>> type;
};
```

An example of use is:

```
// Client of storage using typedef
int n = 10; int val = 2;
Client<int> myClient(n, val); myClient.print();
```

Answer the following questions:

a) Create the class using alias template instead of typedef.

b) Test your code. Do you get the same output as before? What are the advantages?

8. (Smart Pointers Review)

We include some easy exercises on the use of smart pointers.

Answer the following questions:

a) Use move semantics to create a shared pointer from another shared pointer and from a unique pointer.

b) Swap two shared pointers; swap two unique pointers.

c) Create a weak pointer from a shared pointer and from another weak pointer.

In all cases, determine what the reference count is and how resource ownership is addressed.

9. (Move Semantics '101')

The objective of this exercise is to get you used to *move semantics* and *rvalue* references.

Answer the following questions:

a) Create a string and move it to another string. Check the contents of the source and target strings before and after the move.

b) Create a vector and move it to another vector. Check the contents of the source and target vectors before and after the move. Compare the time it takes to move a vector compared to the time when using a copy constructor or a copy assignment statement.

c) Consider the following user-defined code to swap two objects:

```
template < typename T >
         void SwapCopyStyle(T& a, T& b)
{ // Swap a and b in copying way; temporary object needed

         T tmp(a);  // Copy constructor
         a = b;     // Copy all elements from b to a
         b = tmp;   // Copy all elements from tmp to b
}
```

How many temporary copies are created? Now create a function based on move semantics to swap two objects. Compare the relative performance of the two swap functions by timing the swap of two very large vectors. Use `std::chrono` or your favourite timer. (We discuss `std::chrono` in Chapter 10.)

10. (Creating Composite Objects)

In this exercise we create a variation and generalised version of the composite class that we introduced in Section 2.4.1 but we now use unique pointers (instead of shared pointers). The related pattern is called *Collection Members* (POSA, 1996) and it is described as follows:

> *In this case we aggregate similar objects into one whole; we speak of an organisation and its members. We make no distinction between the members of the collection. The whole provides functionality to clients for iterating over the members and performing operations on them. An example of a collection-members object from everyday life is when we model a house that consists of a number of rooms. Each room has the same interface. Another example is a portfolio of assets, with each C++ asset type having the same interface.*

Answer the following questions:

a) Create a class template that consists of a list of instances of a generic type `T`. These are wrapped in a unique pointer analogously to the class in Section 2.4.1.

b) Create methods to add member to and to remove members from the container. Determine how to use pointers as input arguments. Consider the option of move semantics to add members to the container. What would the advantages be? What are the dangers?

c) Create a member function that iterates over the members of the collection and that performs a *command* on each one. The command should be a universal function wrapper of the form:

```
template <typename T>
      using CommandFunctionType = std::function<void (T& t)>;
```

d) Test the code in parts a) to c) by considering a `Polyline` class that consists of a list of `Point` instances. Consider commands to execute geometric transformations on the polyline's members (which are points) such as scaling, rotation and translation. Make the command so generic that it works for a range of functions with the same signature. This will be a simple example of the *Command* pattern.

e) Can you find examples of the *Collection Members* pattern in computational finance? Choose a representative example and implement it using the same approach as in part d). You need to identify the members of the collection and the corresponding commands.

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 3

Modelling Functions in C++

3.1 INTRODUCTION AND OBJECTIVES

In this chapter we introduce various ways to define and use functions in C++. A *function* is a mathematical concept and the goal of this chapter is to show how to realise this concept in C++ from the viewpoint of the procedural, object-oriented and functional programming models. Having done that, we examine how to apply these capabilities to designing and implementing flexible software. Up to the present time it is safe to say that many C++ applications make extensive use of function pointers, overloaded functions and subtype polymorphism. In this chapter we discuss how similar functionality can be achieved by using the new functionality in C++. We shall show how to achieve the same (or better) level of flexibility using constructs based on generic and functional programming models.

Much of the material in this chapter is relatively new because it is part of C++11. Furthermore, it may require some effort to learn by developers who have become accustomed to writing applications based on the object-oriented and procedural programming models. For example, we shall see that it is possible to create loosely coupled software systems based on *signature-oriented function* definitions. In particular, the resulting functionality will allow us to implement a number of *software connection architectures* based on *object-oriented*, *interface* and *plug-and-socket architectural models*.

In one sense, this chapter could be seen as one of the most important in the book because we discuss function modelling in C++ and this functionality will be used in future chapters in areas such as:

- *Next-generation design patterns* which are patterns based on generic and functional programming models in addition to the object-oriented model.
- Creating applications based on *policy-based design* principles and module interface languages.
- Combining different programming paradigms; choosing the most appropriate design for the problem at hand.

We have introduced a number of these topics in Demming and Duffy (2010) in which we discussed multiparadigm programming based on the Boost C++ libraries.

In general, we describe the steps to define and use functions as follows:

a) Define the *signature* of the function that you wish to model, including its input arguments, return type and possibly a function name.
b) Determine if more data is needed to specify the function (this is called the *function closure*).
c) Choose how to declare the function (using function pointers, `std::function<>`, lambda function or function objects).
d) Define (implement) the function.
e) Use the function in client code, design patterns and applications. Investigate the amount of coupling between software components as a consequence of choosing a given implementation.

These steps are typical when developing applications. There are many options and the final choice will be determined by the desired level of flexibility. We discuss all these topics as we progress in the book.

3.2 ANALYSING AND CLASSIFYING FUNCTIONS

In this section we define what a function is and we then discuss the various ways to model functions in C++. This classification will be useful in later chapters and applications.

In general, a *function* is mathematically defined. It is a computational entity that produces output from input. It can also have a name, although this is not mandatory (for example, lambda functions have no name as we shall see). In particular, each function has zero or more input arguments. Each of its arguments can be built-in type or user-defined type. In order to avoid *side-effects* we normally assume that all input arguments are read-only, although there are situations in which we wish to modify these arguments on output. The return type of a function is usually a single entity.

We discuss a number of concepts from functional programming languages that will help in our understanding of new features in C++. These concepts have their origins in other programming languages (such as Lisp, F# and Haskell).

We take a fresh look at what functions are, how they are implemented in C++ and how they are used in applications.

3.2.1 An Introduction to Functional Programming

Functional programming is a style of programming that models computation as the evaluation of expressions. In other words, we execute programs by evaluating expressions. Functional programming requires that functions be treated as *first-class objects* so that they can be used just like an ordinary variable in a programming language, for example:

- They can be passed as arguments to other functions.
- They can be returned as the result of a function.
- We can define and manipulate functions from within other functions.

In general, functional programming languages support these features while historically *imperative programming languages* (such as C++) did not. In recent times a number of

extensions to C++ have been proposed that support some of the above features, in particular several Boost C++ libraries and C++11. An *imperative programming language* is one in which programs are composed of statements. These statements can change global state when executed while in *pure* functional programming languages program state tends to be *immutable*.

Some features of functional programming languages are:

- Support for *higher-order functions* (HOFs). A higher-order function can take other functions as arguments. An example of a higher-order function is a loop of the form (pseudocode):

  ```
  foreach(record.begin, record.end, func);
  ```

  where `record` is a data collection and `func` is a function that operates on each element of that collection. Higher-order functions are useful in code refactoring projects because their use reduces the amount of code duplication. We shall see how to simulate higher-order functions in C++ by the use of *function objects* (*functors*) and *lambda functions*.
- *Purity*: in general, some functional programming languages allow expressions to yield actions in addition to returning values. These actions are called *side-effects*. A *pure language* is one that does not allow side-effects.
- *Recursion*: this technique is pervasive in functional programming and it is sometimes the only way to iterate in collections. Use is often made of *tail call optimisation* which avoids recursion consuming too much memory.
- *Strict* and *non-strict evaluation*: these terms refer to how function arguments are processed when an expression is being evaluated. In the former case all arguments are instantiated and then the function returns a value. In the latter case some arguments have not yet been instantiated (these are called *delayed arguments*). It is then not possible to call the function in the traditional sense but what we get is a function with a given *arity* which can be called later by instantiating all the remaining delayed arguments. This is a useful feature because we can create new functions from existing ones by *binding* one or more of their arguments to specific values. We shall see how to do this when we discuss `std::function` and `std::bind`.

Functional programming languages provide better support for structured and top-down programming than imperative programming languages. This is mainly due to the fact that we can model abstractions and decompose them into *software components* using this style. It is possible to implement such components using higher-order functions. We shall see in this book how to define higher-order functions in C++ and use them to design and implement applications for computational finance.

3.2.2 Function Closure

In computer science, a *closure* (also known as *lexical closure* and *function closure*) is the combination of a function and its *referencing environment*, the latter being a table that stores a reference to each non-local (*free*) variable of the function. A closure allows a function to access these non-local variables when invoked outside of its immediate lexical scope. This is in contrast to plain function pointers where this is not possible unless we use static or global variables (whose use we generally do not recommend). The free variables referenced in a

closure survive the enclosing function's execution. To this end, we must allocate them so that they persist until they are no longer needed.

We shall see how the C++ lambda function supports closures in Section 3.4.

3.2.3 Currying

Currying is a technique to transform a function that takes multiple arguments (or an n-tuple of arguments) in such a way that it can be called a chain of functions each of which accepts a single parameter. *Uncurrying* is the dual transformation of currying. It accepts a function and returns another function as a result.

3.2.4 Partial Function Application

Partial application (or *partial function application*) refers to the process of fixing a number of arguments to a function, producing another function of smaller arity. Given a function $f(x, y)$ we might fix (or 'bind') the first argument, producing a function $g(y) \equiv f(1.0, y)$. Evaluation of this function might be represented as $g(2.0)$.

The C++ standard library provides `bind()` (in the namespace `std`) to return a functor that is the result of partial application to the given function. We shall discuss this topic in Section 3.6.

3.2.5 Lambda (Anonymous) Functions

Anonymous functions were originally studied by Alonzo Church in the 1930s when he invented the *lambda calculus*. In this book we shall see how C++ supports anonymous functions. The advantage of anonymous functions is that they do not need a name. Thus, instead of having to define a named function (that is possibly in a separate compilation unit or living in a header file) we can define the necessary code *in situ*, that is at the client site.

We give a number of examples of how to use lambda functions in combination with higher-order functions. We use *pseudocode* and stress that it is not yet C++.

- *Map*: the map function performs a function call on each element of a collection. For example, the following function squares the elements of an array using an anonymous function:

```
a = [1, 2, 3, 4, 5, 6]
print map(lambda x: x*x, a)
[1, 4, 9, 16, 25, 36].
```

  In this case the anonymous function accepts an argument and multiplies it by itself.
- *Filter*: the filter function returns all those elements from a collection that evaluates to `true` when they are passed to the anonymous function, in this case a function that tests if a number is even:

```
a = [1, 2, 3, 4, 5, 6]
print filter(lambda x: x % 2 == 0, a)
[2, 4, 6].
```

- *Fold*: a fold (*reduce* or *accumulation*) function iterates over all the elements of a collection and it accumulates a value as it progresses, for example:

```
a = [1, 2, 3, 4, 5]
print reduce(lambda x,y: x*y, a)
120.
```

  In this case the following evaluation takes place: $((((1 \times 2) \times 3) \times 4) \times 5 = 120$.

3.2.6 Eager and Lazy Evaluation

Lazy evaluation (or *call-by-need*) is an evaluation strategy that delays the evaluation of an expression until its value is needed (which is called *eager* or *non-strict* evaluation). For example, a lazy type is a *thunk* or *placeholder* for some computation. Once created, we can pass the lazy value around as though it has been evaluated. This value is evaluated only once, that is when forced to do so. Eager evaluation is the default behaviour in many programming languages, for example C++.

Lazy evaluation can lead to reduction in memory footprint because values are created only when needed. However, it is difficult to combine with features in imperative languages, for example input, output and exception handling due to the fact that operations can become indeterminate.

3.2.7 Fold

A *fold* (also known as *reduce*, *accumulate* or *compress*) is a family of higher-order functions that analyse a recursive data structure and that recombine the elements in the data structure in some way. This is achieved by using an operation to build a return value. In general, a fold consists of a combining function, a top node of a data structure and possibly some default values to be used under certain circumstances.

Folds can be seen as structural transformations because they consistently replace the structural components of a data structure with functions and values. For example, in Lisp we can build lists from two primitives, namely a list is either empty or it is constructed by prepending an element in front of another list. Another example is the folding of the list [1,2,3,4,5] with some kind of mathematical operator, for example addition (operator +). In this case the value will be 15. We shall see in Chapters 17 and 18 how STL supports fold operations, in particular when applied to numerical algorithms.

3.2.8 Continuation

A *continuation* is an abstract representation of the control state of a computer program. It *reifies* (the process by which an abstract idea about a computer program is turned into an explicit data model or other object created in a programming language) the program control state. In other words, we can access the created data structure directly in the programming language instead of it being hidden in the run-time environment. Continuations are useful for encoding other control mechanisms such as exceptions, generators and coroutines. A *coroutine* is a component in a computer program that generalises subroutines to allow multiple *entry points* for suspending and resuming execution at certain locations. Coroutines

are well suited for implementing more familiar program components such as cooperative tasks, iterators, infinite lists and pipes. In some languages (for example C#) use is made of continuations to allow sequential programs to be analysed in terms of functional programming semantics, as can be seen in LINQ (*Language Integrated Query*) for example. A disadvantage of using continuations is that code can be difficult to understand.

3.3 NEW FUNCTIONALITY IN C++: `std::function<>`

The class `std::function` (defined in `<functional>`) provides polymorphic wrappers that generalise the function pointer concept. Its origins can be traced to the *Boost C++ Function* library (as discussed in Demming and Duffy, 2010). The main advantage is that it can be used as a *bridge* or *virtual machine* to *callable objects* such as free functions, member functions, function objects and lambda functions. To this end, the class `std::function<>` allows us to model these callable objects as first-class objects. As an example, we show how to use `std::function<>` in combination with free functions. Let us first define three functions:

```
double Func(double x)
{
       return x*x;
}

double Func2(double x)
{
       return std::sqrt(x);
}

double Func3(double x)
{
       return std::exp(x);
}
```

We see that these functions accept a `double` as input argument and they return a `double`. In order to subsume these functions in a universal function wrapper as it were, we define:

```
std::function<double (double x)> f;
```

We can then assign an instance of `f` to a free function or to each of the above free functions, for example:

```
// Assign to a global function
f = Func;
std::cout << "First function: " << f(2.0) << std::endl;
```

In client code we work with instances of `std::function<>` which have been instantiated somewhere else. This approach promotes code reusability and we shall see more examples in this and later chapters, in particular how to use `std::function<>` with lambda functions, functions to objects and member functions.

The second example is to define an array of functions and then execute them. We add each of the above free functions to the array of `std::function<>` instances and we use a lambda function (discussed in detail in Section 3.4) to execute each one:

```
// Create an array of functions
typedef std::function<double (double sx)> Task;
typedef std::vector<Task> Tasks;
Tasks tasks;

tasks.push_back(Func);
tasks.push_back(Func2);
tasks.push_back(Func3);

double val = 10.0;              // Captured variable
std::for_each(tasks.begin(), tasks.end(), [&val] (const Task& t)
                        { std::cout << t(val) << std::endl;});
```

This example gives an indication of what is possible with `std::function<>`. For example, we could use it to implement *callback functions* in the *Observer* pattern (GOF, 1995). More generally, we can use the *Boost C++ signals2* library as implementation of *Observer* (Demming and Duffy, 2010).

3.4 NEW FUNCTIONALITY IN C++: LAMBDA FUNCTIONS AND LAMBDA EXPRESSIONS

We have already given an introduction to lambda functions in Section 3.2.5 from a language-independent viewpoint. These are unnamed functions that can be declared within the scope of other functions. They are alternatives to using global functions and *function objects* (*functors*) in C++.

3.4.1 Basic Syntax

The general syntax for a lambda function is:

```
[capture variables](input arguments) <mutable> -> return type
{
    function implementation
}
```

First, we see that the presence of the brackets `[]` announces a new lambda function having a given return type. Second, a lambda function has input arguments as well as *captured variables* that belong to the lambda function's closure. These two sets of variables correspond to the lambda function's state. It is possible to capture variables by value or by reference, depending on whether we wish to copy by value or by reference into the body of the lambda function. We can specify the choice by using the following choices:

```
[a]       // Capture local variable 'a' by value.
[a, b]    // Capture vars 'a' and 'b' by value
[&a]      // Capture var 'a' by reference
[a, &b]   // Capture 'a' by value, 'b' by reference
[=]       // Capture all used variables by value
[&]       // Capture all used vars by reference
[=, &a]   // 'a' by ref; other vars by value
```

We also note the presence of the optional keyword `mutable` in the above specification. This is sometimes needed because lambda functions are `const` and hence their state cannot be changed. Thus, in order to make the lambda function non-`const` we declare it as `mutable`. Finally, the body of the (lambda functions are compiled into function objects) lambda function is standard C++ and it may use both its input and captured variables.

The simplest lambda function has the structure:

```
[]
{
      function implementation
}
```

All lambda functions have a return type (which can be `void`). It is not necessary to explicitly specify what this return type is unless the body of the function contains embedded if–else logic. We say that the return type is *inferred* or *deduced* when it is absent.

3.4.2 Initial Examples

We now implement the examples from Section 3.2.5 using lambda functions. We shall see in Chapters 17 and 18 how to achieve these ends in STL but for the moment we shall use the `std::for_each()` algorithm that allows us to access, process and modify the elements in a container in different ways. We assume some knowledge of STL containers that implement vectors and lists. The three functions (*map*, *filter* and *fold*) all produce output from input and to this end we use captured variables in which to store this output. The complete code is given by:

```
#include <algorithm> // e.g. std::for_each
#include <vector>
#include <list>
#include <iostream>

typedef std::vector<int> Vector;
typedef std::list<int> List;

void print(const Vector& vec)
{
   std::for_each(vec.begin(), vec.end(), [] (double d)
      { std::cout << d << ",";});
   std::cout << "\n";
}
void print(const List& list)
```

```
{
   std::for_each(list.begin(), list.end(), [] (double d)
      { std::cout << d << ",";});
   std::cout << "\n";
}

int main()
{
    std::size_t N = 6;
    Vector vec(N);

    // Initialise the values of the vector
    int val = 1;
    std::for_each(vec.begin(), vec.end(), [&val] (int& j) {j = val++;});
    print(vec);

    /* a = [1, 2, 3, 4, 5, 6]
    print map(lambda x: x*x, a)
    [1, 4, 9, 16, 25, 36]*/

    Vector vec2(vec.size());
    int index = 0;
    std::for_each(vec.begin(), vec.end(), [&vec2, &index] (int j)
                    {vec2[index++] = j*j;});
    print(vec2);

    /*a = [1, 2, 3, 4, 5, 6]
    print filter(lambda x: x % 2 == 0, a)
    [2, 4, 6]*/

    List myList;
    std::for_each(vec.begin(), vec.end(), [&myList] (int j)
                            { if (j%2 == 0) myList.push_back(j);});
    print(myList);

    /* a = [1, 2, 3, 4, 5]
    print reduce(lambda x,y: x*y, a)
    120*/

    // Reinitialise vec
    std::size_t M = 5;
    vec.clear(); vec.resize(M); for (std::size_t n = 0; n < vec.size(); ++n)
                            { vec[n] = n+1;}

    // Incorrect; variable 'sum' call by value (and all captured
    // variables)
    int sum = 0;
    std::for_each(vec.begin(), vec.end(), [sum] (int j) mutable
    { sum += j;});
    std::cout << sum << std::endl;
```

```
    // Correct; variable 'sum' call by reference (and all captured
    // variables)
    val = 1;
    std::for_each(vec.begin(), vec.end(), [&val] (int& j) {j = val++;});

    sum = 1.0;
    std::for_each(vec.begin(), vec.end(), [&sum] (int j) { sum *= j;});
    std::cout << sum << std::endl;

    return 0;
}
```

We see that some boilerplate code is needed in order to realise our objectives. This is not a concern to us at the moment because we wish to show how to write lambda code. In Chapters 17 and 18 we shall see how STL implements these functions.

3.4.3 Lambda Functions and Classes: Capturing Member Data

It is possible to define lambda functions in a class and it is also possible to capture member data. The rules are:

`[this]`	captures the *this* pointer only.
`[=]`	captures the *this* pointer and local variables by value.
`[&]`	captures the *this* pointer and local variables by reference.

Inside the body of the lambda function we access the class members in the usual way. We take an example of a class to calculate a vector of sine values. To this end, we transform the input vector using a lambda function. The objective is to use a lambda function to effect the transformation and in this case we can access the member data in the lambda function:

```
class CalculateSine
{
private:
    double m_amplitude;

public:
    CalculateSine(): m_amplitude(0.0) {}
    CalculateSine(double amplitude): m_amplitude(amplitude) {}

    // Calculate sine.
    std::vector<double> Calculate(vector<double>& input)
    {
       // Create result vector.
       std::vector<double> result(input.size());

       // Use STL algorithm
       std::transform(input.begin(), input.end(), result.begin(),
                      [this] (double v)-> double
                          {return m_amplitude*sin(6.28319*v/360.0); } );
```

```
        // Return the result.
        return result;
    }
};
```

3.4.4 Storing Lambda Functions

We now discuss how to reuse lambda functions by defining them once and calling them from multiple clients. In general, lambda functions are compiled into function objects (a *function object* is a class that implements the *function call operator* `()`). Since a lambda function is an anonymous type we cannot create a variable to refer to it. However, a lambda function can be stored in a `std::function` (but not in a function pointer). To make things easier, we can use the keyword `auto` to declare the function without having to specify a type. For example, the following lines of code have the same effect:

```
#include <functional>

// Define two lambda functions and store them.
std::function<double (double)> f1=[](double v){ return v*v; };
auto f2=[](double v){ return std::sqrt(v); };
```

We can then use the new functions in code as follows:

```
// Create and fill vector.
std::vector<double> v;
for (double d=0.0; d<=360.0; d+=15) v.push_back(d);

// Print vector and calculations.
// Note that we use an auto variable here so we don't
// need to say: vector<double>::iterator it;
for (auto it = v.begin(); it < v.end(); it++)
{
   std::cout<<*it<<"\t"<<f1(*it)<<"\t"<<f2(*it)<<std::endl;
}
```

3.5 CALLABLE OBJECTS

A *callable object* is a general name given to different ways to define functions in C++ (Josuttis, 2012):

- A free function.
- A pointer to a member function.
- A function object, a class that overloads the function call operator `()`.
- A lambda function, a special kind of function object.
- A static member function.

From a design point of view we wish to write code once by using `std::function` as the most general way to declare callable objects. Since we create software that can be used with a variety of programming styles, it is clear that using the most general way to declare functions using `std::function` will be optimal because its instances can be assigned to any of the above specific function types. To this end, we distinguish between a number of roles in software development. We take the example of creating code (we call it the *supplier code*) that is policy free in the sense that it is independent of the above function types because it uses `std::function` and second *user code* (or *configurator code*) that specifies which specific function type to use. We shall extend this approach in later chapters when we design and implement applications for computational finance.

3.6 FUNCTION ADAPTERS AND BINDERS

A *function adapter* is a function object that enables the composition of function objects with each other. We concentrate on the `bind()` adapter that is supported in C++ and which is similar to *Boost C++ Bind* library (see Demming and Duffy, 2010). We use `bind()` for the following reasons (Josuttis, 2012):

- Adapt and compose new function objects from existing or predefined function objects.
- Call free functions.
- Call member functions for objects, pointers to objects and smart pointers to objects. In other words, the syntax (the free function `bind()` in namespace `std`) is uniform.

We see that the `bind()` adapter is a realisation of partial function application as already discussed in Section 3.2.4. In particular, it binds parameters for callable objects. This means that we can bind parameters for the function types that we described in Section 3.5. For passed arguments we use predefined *placeholders* `_1, _2, …` in the namespace `std::placeholders`.

We take a simple example to show how to use `bind()`. The goal is to write a function that takes an argument and then prints it on the console. Using a free function we would have:

```
template <typename T> void print (const T& t)
          { std::cout << "Function pointer: " << t << std::endl;}
```

More generally, the corresponding syntax is:

```
typedef std::function<void (const double& d)> FunType;
FunType fun;
```

We now define a class consisting of two member functions having the same signature as that of `FunType`:

```
template <typename T> class PrintClassII
{
private:
       T data;
```

```
public:
       PrintClassII (const T& t) : data(t) {}

       void print (const T& t)const
             { std::cout << "Bind member: " << data << ", " << t;}
       static void printStatic (const T& t)
             {std::cout << "Bind static member function: " << t;}
};
```

We cannot assign `fun` directly to the member functions in this class because it is incompatible with the member functions. Instead, we must bind:

```
// Bind to member functions
PrintClassII<double> pcII(33.0);

fun = std::bind<double>
       (&PrintClassII<double>::print,pcII, std::placeholders::_1);
fun(value);

fun = std::bind<double>
       (&PrintClassII<double>::printStatic,std::placeholders::_1);
fun(value);
```

We now consider another example of a class that has a member function with two input arguments. The class is:

```
template <typename T> class PrintClassIII
{ // Print function has two input parameters
private:
       // No state
public:
       PrintClassIII () {}

       void print (const T& t, const T& data)
          { std::cout << "Adapted member function: "<<data << ", " << t;}
};
```

In applications we may wish to fix the variable `data` and subsequently use `print()` as a function of one argument only. Again, we can then use `std::function`:

```
// Adapt an incompatible function because it has 2 parameters
PrintClassIII<double> pcIII;
double fixedValue = 999.0;
fun = std::bind<double>
       (&PrintClassIII<double>::print,pcIII, std::placeholders::_1,
       fixedValue);
fun(value);
```

Finally, we show how to define functions of arity two and one. Notice the use of the keyword `auto` that saves some cognitive overload and makes the code more readable:

```
// Functions of arity 2
typedef std::function<void (const double& d1,const double& d2)> FunType2;
FunType2 fun2 = std::bind<double> (&PrintClassIII<double>::print,pcIII,
          std::placeholders::_1, std::placeholders::_2);
fun2(1.0, 2.0);

// Arity 1
double value2 = 3.1415;
auto fun3 = std::bind<double>
          (&PrintClassIII<double>::print,pcIII, value2,
          std::placeholders::_1);
fun3(3.0);

auto fun4 = std::bind<double>
          (&PrintClassIII<double>::print,pcIII, std::placeholders::_1,
          value2);
fun4(3.0);
```

3.6.1 Binding and Function Objects

C++ has closure-like constructs in the form of function objects. A *function object* in C++ is a class that implements the *function call operator* `()`. It behaves like a function in functional programming languages. But it does not implicitly capture local variables as closures do since there are no free variables associated with it. However, it may contain state.

Again, we take an example of a function object containing state and that is able to print this state:

```
// Function object
template <typename T> class PrintClass
{
private:
      T data;
public:
      PrintClass (const T& t) : data(t) {}

      void operator () (const T& t)const
            {std::cout << "Function object: " << data << ", " << t;}
};
```

We can then call the function in the following ways:

```
typedef std::function<void (const double& d)> FunType;
FunType fun;

double value = 2.0;

// Function object
PrintClass<double> pc(3.0);
pc(value);
```

```
fun = pc;
fun(3.0);
```

In this case, creating a function object is more time consuming than creating a lambda function. And we usually save it in a separate file while a lambda function can be defined at the point where it is needed.

3.6.2 Binding and Free Functions

It is possible to use `bind()` with free functions. We give examples of free functions of arity two and one, respectively:

```
double func(double a, double b)
{
      return  a - b;
}

double func1d(double x)
{
      return exp(x);
}
```

For example, these functions could be defined in a library of algorithms (of course, the code would be more complex in real software systems) and we wish to experiment with these functions in client code. The code was ported from the *Boost C++ Bind* library and this was a relatively painless process. We only needed to change the namespace from `boost` to `std` and redefine placeholders. The code is:

```
// Port from Boost Bind

using namespace std::placeholders;

// Bind params 1 and 2, answer = -1
double result = (std::bind(&func, _1, _2))(1.0, 2.0);
std::cout << "_1, _2: " << result << std::endl;

// Bind params 2 and 1, answer = 1
result = (std::bind(&func, _2, _1))(1.0, 2.0);
std::cout << "_2, _1: " << result << std::endl;


// Binding to params chosen at run-time
std::cout << "Give a value 1: "; int v1; std::cin >> v1;
std::cout << "Give a value 2: "; int v2; std::cin >> v2;

result = (std::bind(&func, _1, _2))(v1, v2);
std::cout << "_1, _2: " << result << std::endl;

// Using placeholders in more than one place in an expression
result = (std::bind(&func, _1, _1))(v1, v2);
```

```
std::cout << "_1, _1: " << result << std::endl;

result = (std::bind(&func, _2, _2))(v1, v2);
std::cout << "_2, _2: " << result << std::endl;

std::function<double (double x)> fA; // Preferred form
fA = func1d;
std::cout << "Value: " << fA(0.0);
```

We recommend that you run this code and inspect the output in order to understand this somewhat cryptic syntax.

We need to understand the relationship between placeholders, formal function arguments and how placeholders are assigned to values. As an example, we have:

```
result = (std::bind(&func, _2, _1))(1.0, 2.0);

double func(double a, double b)
{
       return  a - b;
}
```

In this case placeholder `_2` is assigned the value 2.0 and it corresponds to formal variable `a` while placeholder `_1` is assigned value 1.0 and it corresponds to formal variable `b`. In this sense the placeholder is a mediator between a function's formal parameter and the value assigned to that parameter. Thus, the result of calling this function is 1.0.

3.6.3 Binding and Subtype Polymorphism

We investigate the possibility of using virtual member functions with `bind()`. In particular, we would like to bind to a base class's polymorphic member function and to change the base class pointer at run-time. We consider the following class hierarchy:

```
class Base
{
public:
      virtual void print() const { std::cout << "Base\n";}
};

class D : public Base
{
public:
      void print() const { std::cout << "Derived\n";}
};

class D2 : public Base
{
public:
      void print() const { std::cout << "Derived2\n";}
};
```

We wish to assign a function to any of the above member functions at run-time. The following code shows how to do it:

```
// Binding and polymorphism
typedef std::function <void ()> FunType3;
Base* b = new D;
FunType3 myF3 = std::bind(&Base::print,b);
myF3(); // Derived class D print will be called

delete b;
b = new D2;
myF3(); // Derived class D2 print will be called

delete b;
```

In this way we can achieve high levels of customisability using `bind` in our code. A separate issue is to measure the relative performance.

3.7 APPLICATION AREAS

We have now completed our discussion of some of the ways to model functions in C++. The new features are to be found in the C++ standard and are essentially based on the functional programming model, allowing us to write code that is more flexible than code based exclusively on subtype polymorphism. We shall see examples of these features as we progress in this book. Furthermore, we can integrate functions, data types and data containers to form generic and reusable structures for numerical applications. A preview of some examples is:

- Improvements to the STL, for example the new `bind()` functionality and the ability to use lambda functions with STL algorithms.
- New data types such as `std::tuple`, `std::array<>` and `boost::variant` that we can use as function arguments and return types.
- Re-engineering of the somewhat outdated GOF patterns (GOF, 1995) to subsume the object-oriented, generic and functional programming models. The resulting design patterns are very flexible and more reliable than the GOF patterns.
- Using `std::function` to model C++ classes for stochastic differential equations (SDEs) and partial differential equations (PDEs).

3.8 AN EXAMPLE: *STRATEGY* PATTERN NEW STYLE

We take an example that shows what we wish to achieve in later chapters. In order to scope the problem, we create a class called `Point` that implements points in two-dimensional space. What is special about the problem is that we create a customisable function that calculates the distance between two points. Furthermore, we have two major requirements concerning the flexibility of the proposed solution:

- It should use the `Strategy` design pattern (GOF, 1995) because this is based on the object-oriented model in conjunction with subtype polymorphism (using virtual functions).

- It should be possible to use the class `Point` using any callable object. In this way we do not enforce a specific programming paradigm on developers. For example, the algorithm to calculate the distance between two points could be a free function, a member function or a lambda function. This is possible because we have used `std::function<>` as member data of `Point` and it is instantiated in the constructors of `Point` to define a uniform syntax.
- The body of the code to compute the distance between two points should be customisable. For example, some applications may demand reduced or enhanced accuracy and this entails defining extra variables and/or help functions. The class `Point` should know nothing about these specific requirements.

We model two-dimensional points by embedding an `std::function<>` instance as member data of `Point`. This is not necessarily the most efficient solution but that is not our concern at this moment:

```
template <class X=double> class Point
{
private:

   // The two coordinates
   X m_x;
   X m_y;

   // The embedded algorithm (strategy) object; each object gets its
   // own copy
   std::function<double (const Point<X>& p1, const Point<X>& p2)> algo;

public:

   typedef std::function<double (const Point<X>& p1, const Point<X>& p2)>
   FunctionType;

   // Constructors & destructor
   Point(const FunctionType& algorithm);
   Point(const X& first, const X& second, const FunctionType& algorithm);
   Point(const Point<X>& source);
   virtual ~Point();

   // Selectors
   const X& First() const;
   const X& Second() const;

   // Modifiers
   void First(const X& val);
   void Second(const X& val);

   // Functions
   double Distance(const Point<X>& p2) const;
```

```
   // Assignment operator
   Point<X>& operator = (const Point<X>& source);

   template <class X>
      friend std::ostream& operator<<(std::ostream& os,const Point<X>& p);
};
```

We can create points by providing them with some implementation of `FunctionType`. We define two specific functions:

```
// Algorithms for calculating distances between points using
//'double' in this case
double ExactDistance(const Point<double>& p, const Point<double>& p2)
{ // Exact distance as a C function with no state

   // Get the length of the sides
   double a = p.First() - p2.First();
   double b = p.First() - p2.First();

   // Use Pythagoras' theorem to calculate distance
   return std::sqrt(a*a + b*b);
}

struct ApproxDistance
{ // Exact distance as a function object with extra state; this is
  //in essence a wrapper for an algorithm containing its own
  //private state

    // Stick on an extra 'margin' to the calculated distance
    double extra;

    ApproxDistance(double margin): extra(margin) {}

    double operator () (const Point<double>& p, const
    Point<double>& p2) const
    {
       std::cout << extra;
       return ExactDistance(p, p2) * (1.0 + extra);
    }
};
```

We can now use these algorithms in client code:

```
// The case of stateful delegation (Composition); N.B. each object is
// born with its own algorithm so that there is asymmetry in the calls
// on individual points. The distance from point p1 to point p2 is not
// necessarily the same as from p2 to p1.
Point<double> pA(1.0, 1.0, ExactDistance);

std::cout << "Give margin as fraction [0.0,1.0] please: "; double
myMargin;
```

```
std::cin >> myMargin;

ApproxDistance taxiObject(myMargin);
Point<double> pB(taxiObject);

std::cout << "Exact distance between A and B points: " << pA.Distance(pB);
std::cout << "Approximate distance: " << pB.Distance(pA);
```

Summarising, we have given a typical example of *composition-based delegation* which is one of the fundamental design techniques to implement many of the design patterns in GOF (1995). However, the GOF patterns are based on the traditional object model which is less flexible it seems than the combination of the object, generic and functional models that we introduce in this book.

3.9 MIGRATING FROM TRADITIONAL OBJECT-ORIENTED SOLUTIONS: NUMERICAL QUADRATURE

In Duffy (2004) and Duffy and Kienitz (2009) we discussed a number of classes and code to numerically integrate scalar-valued functions of one variable using the *midpoint rule* and the *Monte Carlo 'hit or miss' rule*. The approach used has the following characteristics and shortcomings:

a) The function to be integrated is a global function (function pointer).
b) We used the *Bridge* design pattern (GOF, 1995) using raw pointers, subtype polymorphism and object-level composition. We need to pay attention to performance and memory management issues.
c) Function names have not been standardised, making it difficult to use with other software.
d) Clients who wish to define their own numerical quadrature schemes must create classes that are derived from a common base class (called `IntegratorImp`).

We have used a combination of C (function pointers), C++ and OOP design patterns. We will rectify these problems by using `std::function<>` to implement the *integrand* (function to be integrated) and the numerical quadrature schemes. Furthermore, the current solution does not allow much flexibility regarding customisability of input and output. For example, some future requirements are:

e) Absolute and relative accuracy required.
f) The ability to define user break points in the interval of integration where local difficulties may occur, for example singularities and discontinuities in the integrand.
g) Output: estimate of the absolute error.
h) Output: number of function evaluations needed during the execution of the numerical quadrature scheme.

We address issues a) to d) in this chapter while we discuss the other topics in Chapter 19 in the context of the numerical solution of the nonlinear equations. The motivation for the new design is to first define the signature of the function that models the integrand and then to define the signature of the functions that implement numerical quadrature schemes. The input

arguments are the integrand, the interval of integration and the number of subdivisions of the interval:

```
// Define interfaces
typedef std::function<double (double x)> Function;
typedef std::function<double (const Function& fun,
                              const Range<double>& range, std::size_t
                              N)> QuadMethod;
```

An important remark is that all quadrature methods accept three arguments representing objects that are created elsewhere. This ensures that the code for numerical quadrature is not hampered by our having to construct objects in addition to implementing an algorithm (this is an example of the *Single Responsibility Principle* (SRP)). These construction activities are encapsulated in a dedicated class:

```
class NumericalIntegrator
{
public:     // Convenience only
   Function fun;
   Range<double> ran;
   QuadMethod quad;

   // Call the quad scheme as a function object
   double operator () (std::size_t N) { return quad(fun, ran, N);}
};
```

This is a generic and customisable class. Clients choose which functions to integrate and which numerical quadrature schemes to use. For example, we test the following functions:

```
// Test data
double func(double x)
{
       return x*x*x*x + 1;
}

double func2(double x)
{
       return exp(-x);
}
```

Furthermore, we have implemented two specific schemes (in this case we use random number generators from the *Boost Random* library):

```
double MidpointRule(const Function& fun, const Range<double>& range,
                    std::size_t N)
    { // Global C function (based on Duffy 2004)

    double A = range.low();
    double B = range.high();
```

```
    double res = 0.0;
    double h = range.spread() / double(N);
    for (double x = A + (0.5 * h); x < B; x += h)
    {
          res += fun(x);
    }

    return res*h;
}

struct HitorMissMonteCarlo
{ // Based on Duffy and Kienitz 2009

   double C; // Max value of fun in range

   double operator()(const Function& fun, const Range<double>& range,
   std::size_t N)
   {
      double A = range.low();
      double B = range.high();
      double d = B - A;
      double V = C * d;            // Volume of bounding box

      // A. Define the lagged Fibonacci and Normal objects
      boost::lagged_fibonacci607 rng;
      rng.seed(static_cast<boost::uint32_t> (std::time(0)));
      boost::uniform_real<> uni(A, B);
      boost::variate_generator<boost::lagged_fibonacci607&,
                  boost::uniform_real<> > uniRng(rng, uni);

      long NH = 0;  // Number of hits
      double tmp1;
      double tmp2;

      for (long j = 1; j <= N; ++j)
      {
         tmp1 = uniRng(); tmp2 = uniRng();
         if (fun(tmp1) > C*tmp2)
         { // Then the point falls inside the curve f(x)

            NH++;
         }
      }
      return V * double (NH) / double (N);
    }
};
```

Incidentally, we discuss random number generation and its applications in Chapter 16. Finally, the following code shows how to use the new design:

```
NumericalIntegrator myQuad;

myQuad.fun = func;
myQuad.ran = Range<double>(0.0, 1.0);

// Integrator is midpoint rule
myQuad.quad = MidpointRule;

std::size_t N = 1000;
std::cout << "Approx integral (Midpoint): " << myQuad(N) << std::endl;

// Integrator is Monte Carlo 'dart throwing'
HitorMissMonteCarlo mc;
mc.C = 2.0;
myQuad.quad = mc;

N = 1000000;
std::cout << "Approx integral (MC): " << myQuad(N) << std::endl;

// Nasty functions
myQuad.fun = func2;
mc.C = func2(1.0);
N = 1000000;
std::cout << "Approx integral (MC): " << myQuad(N) << ", exact: "
          << -exp(-1.0) + 1.0 << std::endl;

myQuad.quad = MidpointRule;
N = 1000;
std::cout << "Approx integral (Midpoint): " << myQuad(N) << ", ";
```

The advantages of this approach are:

a) We are free to implement the integrand in a number of ways and not just as a free function.
b) The GOF *Bridge* pattern is now redundant because we use `std::function<>` which is more flexible. The latter class plays the role of the *abstraction* in the *Bridge* pattern.
c) We overload the function call `operator()`. This approach promotes standardisation and uniformity.
d) Clients have freedom in choosing how they implement schemes. It is no longer necessary to create specialisations of a common base class containing polymorphic functions. We also note improved performance.

This example could serve as a test case for other kinds of applications involving functions and numerical algorithms. We expand on this topic in later chapters.

3.10 SUMMARY AND CONCLUSIONS

We have given an introduction to modelling functions in C++. We discussed some new features in C++ such as `std::function<>` and lambda functions as well as function objects and function pointers. We also gave an introduction to a number of concepts from computer science

and functional programming that are directly (or indirectly) supported in C++. Finally, we gave a number of simple and extended examples on how to use the functionality that we discussed in this chapter.

In this chapter lie the seeds for modern design methods in C++ that we shall apply to various applications in computational finance. The *supplier code* should be as flexible as possible and for this reason it interfaces with `std::function<>` and the *configurator* or *client code* that corresponds to the different programming styles in software teams.

Learning a functional programming language such as Haskell or F# is a good way to get insight into the functional programming model.

3.11 EXERCISES AND PROJECTS

1. (General Brainstorming Questions)

We wish to compare the relative advantages and disadvantages of using the object-oriented approach (classes, inheritance and subtype polymorphism) and the more general approach taken in this chapter. To keep the discussion concrete you can refer to the example on numerical quadrature in Section 3.9. We examine these solutions from the viewpoints of the *ISO 9126 product quality standard*. The characteristics of ISO 9126 that we address in this exercise are:

1. *Functionality*: a set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs.
2. *Reliability*: a set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time.
3. *Efficiency*: a set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions.
4. *Maintainability*: a set of attributes that bear on the effort needed to make specified modifications.
5. *Portability*: a set of attributes that bear on the ability of software to be transferred from one environment to another.

In general, all software products need to conform to one or more of the above characteristics (they may even be implicit and their absence in the software may (unexpectedly) surface at the later stages of a software development project). The objective in this exercise is to determine whether the object-oriented approach is a better fit to requirements 1 to 5 than the approach using `std::function<>` and lambda functions. As specific use cases consider the following requirements:

- The ability of the software to operate with different programming styles.
- Supporting `double` and `float` types.
- Integrating (possibly incompatible) libraries and code with the current software.
- Avoiding run-time memory allocation and deallocation.

Which ISO 9126 characteristics are related to the above points?

2. (Using Libraries and Code)

Let us assume that we have a library of modules for three-dimensional geometry. One module computes the distance between two points in 3D space:

```
// Library adapter
double Taximan3D(double x1, double y1, double z1,
                 double x2, double y2, double z2)
```

```
{ // Distance between two points in 3d taxi space

   return std::abs(x1 - x2) + std::abs(y1 - y2) + std::abs(z1 - z2);
}
```

We reuse this code to create an algorithm to define the taximan's distance in two dimensions. We now create a function that is suitable for two-dimensional geometry:

```
std::function<double (double x1,double y1,double x2,double y2)>
Taximan2D =
            std::bind(Taximan3D, std::placeholders::_1,
            std::placeholders::_2,0.0,
            std::placeholders::_3, std::placeholders::_4,0.0);
```

We explain this cryptic code. It is an example of partial function application (see Section 3.2.4) in which a function with six arguments is morphed into one with four arguments. In particular, the third and sixth arguments (that correspond to the z coordinates of the first and second points, respectively) are bound to the value 0.0.

We now use this function to create a new function object similar to the *Strategy* that we created in Section 3.8:

```
struct Taximan
{ // The scenic route(aka expensive way) from A to B.

   Taximan() {}

   double operator ()(const Point<double>& p,
                      const Point<double>& p2) const
   {
      // For readability
      double x1 = p.First(); double x2 = p2.First();
      double y1 = p.Second(); double y2 = p2.Second();

      double result = Taximan2D(x1, y1, x2, y2);

      return result;
   }
};
```

A test program is:

```
Taximan myTaxi;
Point<double> pC(1.0, 1.0, myTaxi);
Point<double> pD(2.0, 2.0, myTaxi);
std::cout << "Taximan C and D points: "  << pC.Distance(pD) << std::endl;
std::cout << "Taximan D and C points: "  << pD.Distance(pC) << std::endl;
```

Answer the following questions:

a) Write and compile the above code.

b) Compare the efficiency of this code that uses `std::bind` with (possibly) more straightforward code to compute the distance between two points using the two-dimensional taximan algorithm.

3. (Numerical Quadrature Scheme)

In Section 3.9 we developed some numerical quadrature schemes. In this exercise we extend the functionality by implementing the basic *trapezoidal rule*:

$$\int_a^b f(x)dx \approx \frac{(b-a)}{2}[f(a)+f(b)].$$

Answer the following questions:

a) Implement the code for this scheme by a free function.

b) Test the code by comparing the result with the results obtained by the other methods in Section 3.9.

4. (Exception Handling and Functions)

Exceptions can be generated when working with functions and they can be caused by not having defined a function target, for example. In these cases an exception of type `std::bad_function_call` will be thrown. We take an example of a function that takes a `double` as input. We forget to define the function, then we define it and we define it using a `std::string` as input. In the first case we get a run-time error and in the second case we get a compiler error:

Answer the following questions:

a) Create and run some code and see how this functionality works.

b) Create some test cases to determine how to use exception handling. Distinguish between compile-time and run-time errors.

5. (Re-engineering Legacy Code)

We discuss a specific example that is based on the *Bridge* pattern that we used to create a flexible framework for numerical quadrature in one dimension (see Duffy, 2004, pp. 290–292). The design is based on a number of assumptions and techniques:

- Traditional OOP (class hierarchies and subtype polymorphism).
- Modelling functions by function pointers.
- Raw memory allocation (`new` and `delete` operators).

We first discuss the original solution from Duffy (2004) and then we pose some questions on improving the flexibility of the design using the techniques in this chapter. We have already addressed this problem in Section 3.9.

The *Bridge* pattern.

This object structural pattern is very powerful. Its main intent is to separate a class into two distinct parts by using a separation mechanism. In precise terms, we divide a class into two other classes. The first class contains the code that is *implementation independent* and does not change, while the second class contains code that is *implementation dependent*. Since the classes are now disjoint we can switch between implementations at configuration-time or at run-time. There are very many situations where we can apply this pattern.

Some examples of the *Bridge* pattern in numerical analysis are:

- Structuring a matrix as a full matrix or as a sparse matrix.
- Solving the linear system AU = F by double sweep, LU decomposition or iterative techniques (see Chapter 13).
- Integrating functions using various quadrature techniques (e.g. Newton–Cotes).
- Various finite difference schemes for ordinary and partial differential equations.

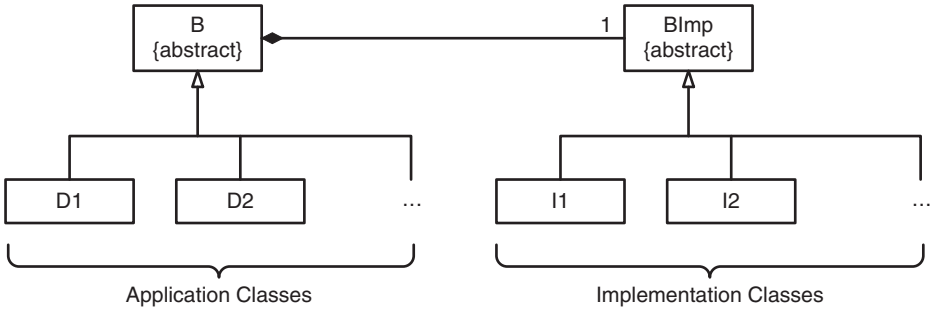


FIGURE 3.1 *Bridge* pattern

Some other examples in financial engineering are:

- Modelling stochastic differential equations by Wiener processes.
- Calculating option price and sensitivities by Monte Carlo, finite differences or finite element methods.
- Calculating option price and sensitivities depending on whether it is a call option or a put option.

Flexibility in switching from one implementation to another is the reason for using a *Bridge* pattern in the first place. Each of the above descriptions can be posed in the same general form and we discuss this form now. In general, we create two hierarchies: the first contains *invariant code* (implementation independent) while the second contains implementation-dependent code, as shown in Figure 3.1. In general, clients call member functions in the abstraction classes in the application hierarchy and these classes then forward the request to a specific implementation class in the implementation hierarchy. The UML sequence diagram that shows the flow of control is shown in Figure 3.2. In more specific cases, the message names will have particular significance in a given application.

We now give a concrete example of the *Bridge* pattern. In this case we integrate real-valued functions of a single variable using a variety of numerical integration rules. In some cases the functions may be well behaved but they may also have discontinuities or singularities either on the boundary or in the region where the function is to be integrated. To this end, we create numerical integrators that can switch between specific numerical integration solvers. The UML class structure is shown in Figure 3.3. In this case the class `NumIntegrator` plays the role of the abstraction and the class `TanhRule` and the midpoint rule defined by the class `MidpointRule` play the roles of the *Bridge* implementations. Mathematically, these integration rules are given by:

$$\int_a^b f(x)dx \approx 2\tanh\left(\frac{h}{2}f\left(\frac{a+b}{2}\right)\right), \quad h = b - a$$

$$\int_a^b f(x)dx \approx hf\left(\frac{a+b}{2}\right), \quad h = b - a.$$

It is possible to modify the code corresponding to Figure 3.3 in a sequence of steps. We get a running prototype after the execution of each step.

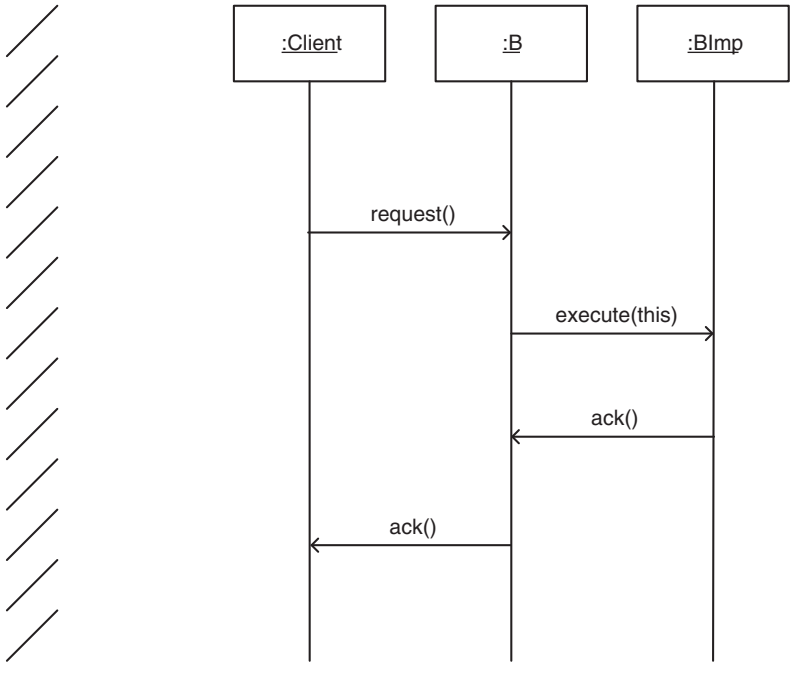


FIGURE 3.2 Information flow in *Bridge* pattern

Answer the following questions:

a) Instead of function pointers, use `std::function<>` to model the integrand in supplier code. Test the numerical quadrature scheme in the configurator code by testing it using the five callable object types that we introduced in Section 3.5.

b) Replace the class hierarchy in Figure 3.3 by a solution incorporating `std::function<>` and function objects. Test the new code using the same examples as in part a). Under which circumstances (if any) would you use lambda functions to implement numerical quadrature algorithms or algorithms in general?

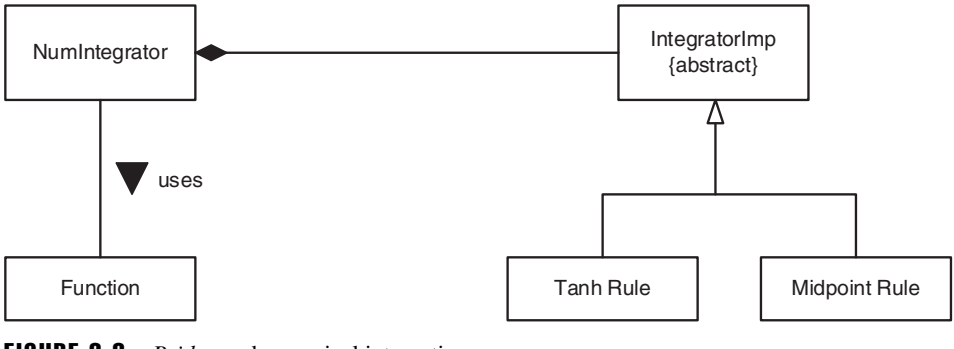


FIGURE 3.3 *Bridge* and numerical integration

c) Modify the class `NumIntegator` to reflect the changes in steps a) and b). In particular, design it as a function object (it computes the approximate value of an integral) with the following members:
- The integrand (this is a `std::function<>`).
- The interval of integration (model as an instance of `Range<>`).
- The specific numerical quadrature scheme (a function object).

Furthermore, where do you define the parameter that represents the number of subdivisions of the interval that the quadrature method needs?

d) Consider how to define and configure a small framework to allow a developer to compute an approximate integral for a range of scalar-valued functions of a single variable.

We shall discuss STL numerical algorithms in Chapters 17 and 18. In Chapter 19 we discuss the numerical solution of nonlinear equations which is similar in style to the design issues in this exercise.

6. (Becoming Acquainted with Callable Objects)

In Section 3.5 we introduced five kinds of callable objects that can be used as target methods of `std::function<>`. The objective here is to write code to compute the *inner product* of two vectors using each callable object type and then to generalise the notion of inner product to one that takes two vectors as input and that computes a scalar (*reduction variable*) from them. Answer the following questions:

a) Consider the inner product (also known as *scalar product* and *dot product*) of two n-dimensional vectors:

$$u = (u_1, \ldots, u_n)^T,\ v = (v_1, \ldots, v_n)^T, \quad u \cdot v = \sum_{j=1}^{n} u_j v_j.$$

Write a free function that accepts two `std::vector<>`instances as input and that produces the quantity described by the above mathematical formula.

b) Write code for the inner product using a function object and a lambda function.

c) Generalise the code so that it operates with other kinds of binary operators (products), for example the distance between two vectors. Three examples are:

$$l_1 \text{ distance: } \sum_{j=1}^{n} |u_j - v_j|$$

$$l_2 \text{ distance: } \left(\sum_{j=1}^{n} |u_j - v_j|^2 \right)^{1/2}$$

$$l_\infty \text{ distance: } \max_{1 \le j \le n} |u_j - v_j|.$$

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 4

Advanced C++ Template Programming

4.1 INTRODUCTION AND OBJECTIVES

This chapter introduces a number of new and advanced language features in C++. These features underlie much of *template metaprogramming* that is making its way into the standard (Abrahams and Gurtovoy, 2005). We give a detailed analysis of these features in this chapter and continue on the same theme in Chapter 6.

The goals of this chapter are:

- To acquaint the reader with template metaprogramming (TMP) in C++.
- To give illustrative examples of TMP to show its application to creating robust code.
- To discuss how to use TMP in your applications.

This chapter can be skipped on a first reading. It can be used mainly as reference for a specific syntax. As with Chapter 6, the topics in this chapter are probably more useful to library builders than to application programmers.

Central to metaprogramming is the concept of a metafunction. In general terms, a *metafunction* is the compile-time analogue of a run-time function. Traditional functions accept values/objects as parameters and return values or objects. However, metafunctions accept types and compile-time constants as parameters and return types and constants. Metafunctions can be used to implement the following functionality:

- Encapsulate a complex type computational algorithm.
- Generate a type using compile-time type selection techniques.

A metafunction is a template. In general, *metafunction implementation* is usually based on template specialisation. We take an example of a metafunction that computes *Fibonacci numbers* $\{F_n\}_n \geq 0$ given by the iterative algorithm:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2}, \quad n \geq 2. \end{aligned}$$

To give an example of what to expect in this chapter and in Chapter 6, we create the well-known recursive metafunction to implement the above algorithm:

```
// Fibonacci
template <int N>
      struct Fibonacci
{
         enum { value  = Fibonacci<N-1>::value + Fibonacci<N - 2>::value};
};

template <>
      struct Fibonacci<0>
{
         enum { value = 0 };
};

template <>
      struct Fibonacci<1>
{
         enum { value = 1 };
};

void TestFibonacci()
{
      // 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
      int x = Fibonacci<6>::value; // 8
      int y = Fibonacci<9>::value; // 34

      std::cout << x << ", " << y << '\n';
}
```

The C++ idiom used here is more general than the *type generator idiom* in which common (*invariant*) parts of a family of type definitions are collected in a structure whose only purpose is to generate another type.

The intent of the type generator idiom is to:

- Simplify the creation of complex template-based types.
- Synthesise a new type or types based on template argument(s).
- Localise default policies when using *policy-based design* (PBD) (we discuss PBD in Chapter 9).

Let us take an example. Consider introducing an extra level of indirection by creating an *adapter class* that captures an *invariant part* (a Boost memory allocator in this case) and a customisable *variant part* (underlying element type in a list):

```
// Type generators
template <class Value>
    struct List
```

```
    {
       // Pool library(fast pool; allocate one chunk at a time)
       typedef std::list<Value, boost::fast_pool_allocator<Value>> type;
    };
```

We can now customise this class for various specialisations of the template parameter, for example:

```
// Type generators
List<int>::type  con1;

struct X
{
   double d1;
   double d2;

   X() : d1(1.0), d2 (3.14) {}
};

List<X>::type  con2;

for (std::size_t j = 0; j < 10; ++j) con2.push_back(X());

for (auto elem : con2)
{
   std::cout << elem.d1 << ", " << elem.d2 << '\n';
}
```

In general, the type generator concept has been replaced by the concept of a metafunction.

We note that the *Boost.MPL* library provides a collection of metafunctions and compile-time data structures to simplify C++ template metaprogramming. A further discussion is outside the scope of this book.

4.2 PRELIMINARIES

We discuss a number of important language features in C++ that will be needed and used throughout the chapters of this book:

- Mixed-mode arithmetic conversions.
- Functions that take a variable number of arguments.
- Value categories and expressions.

4.2.1 Arithmetic Operators and Implicit Conversions

C++ is a language that is used for numerical computation in science, engineering and computational finance. In particular, we are interested in *mixed-mode arithmetic* and *implicit conversions* from one data type to another data type as well as the result of specific *arithmetic*

operations. We give a short introduction to these topics in order to prepare for more advanced topics and applications in later chapters.

In general, a conversion may occur in an expression where a value of a different type is expected, for example:

- A1: In the assignment operator; the value of the right-hand operand is converted to the unqualified type of the left-hand operand.
- A2: Scalar initialisation; the value of the initialiser expression is converted to the unqualified type of the left-hand operand.
- A3: Calling a function with argument types different from those in the function prototype declaration.
- A4: The value of the operand in a function's return statement is converted to an object having the return type of the function.

Some simple code to show what we mean by these examples is:

```
double func(double d)
{
   return d*d;
}

double func()
{ // Return type (an int) convert to double

   int result = 1'000'000;
   return result;
}

void ImplicitConversions()
{ // Simple examples of conversions

   // Assignment
   int i = 1L;                    // Convert long to int
   int j = 3.14;                  // Truncate a double

   // Scalar initialisation
   double d2 = 2.4;
   int i2 = 3;
   double d3 = i2*d2;

   // Function call expression
   int i3 = 5;
   std::cout << func(i3) << '\n';

   // Function return type
   int i4 = func(3.14);
   std::cout << i4 << '\n';

   std::cout << func() << '\n';
}
```

Warnings will be generated when this code is compiled due to the conversions taking place.

We now discuss the topic of defining binary operators whose operands are of different types. It is clear that one of the operands must be *promoted* to the same type as that of the other operand using *arithmetic conversions.* The goal is to produce a *common type* (as also supported by the `std::common_type` type trait that we shall discuss in Chapter 6). We do not go into the details here, but it is important to know which rules are being applied. Some guidelines are:

- If either operand is `long double` then the other operand is converted to `long double`.
- If either operand is `double` then the other operand is converted to `double`.
- If either operand is `float` then the other operand is converted to `float`.
- The types `bool`, `char`, `char16_t`, `char32_t`, `wchar_t` and unscoped enumerations are promoted to integer.
- The *integer conversion rank* increases in the following order: `bool`, `signed char`, `short`, `int`, `long`, `long long`.

We discuss integral and floating-point types in more detail in Chapters 6 and 8.

4.2.2 A Primer on Variadic Functions

Variadic functions are functions that accept a variable number of arguments. A famous example is the function `std::printf`. In order to define a variadic function we use the object type `va_list` that holds information about a function's arguments. This object can be accessed by the following function macros:

- `va_start`: enable access to variadic function arguments.
- `va_arg`: access next variadic function argument.
- `va_copy`: make a copy of variadic function arguments.
- `va_end`: end traversal of variadic function arguments.

We take an example; in this case we define a variadic function to sum a list of integers:

```
long AddIntegers(long count, ...)
{ // Variadic function

   long result = 0;

   // Hold information needed by other macros
   va_list args;

   // Enable access to variadic function arguments
   va_start(args, count);

   // Iterate over the arguments
   for (int n = 0; n < count; ++n)
   {
      // Access next variadic function argument
      result += va_arg(args, long);
   }
```

```
    // End traversal of variadic function arguments
    va_end(args);

    return result;
}
```

An example is:

```
std::cout << AddIntegers(4, 10, 10, 10, 20) << '\n';          // 50
std::cout << AddIntegers(4, 10, 'A', true, false) << '\n';    // 76
```

4.2.3 Value Categories

In general, a *C++ expression* is a name that encompasses operators with their operands, literals and variable names. An expression is characterised by two orthogonal properties, namely its *type* and its *value category*. We also note that an expression has *identity* which means that it is possible to compare expressions (for example, do they refer to the same entity?) and we can compare the addresses of the objects or the functions that they identify.

There are three primary value categories:

- *xvalue*: has identity and it can be moved from.
- *lvalue*: has identity and it cannot be moved from.
- *prvalue*: has no identity and it can be moved from. It is a *pure value expression*.

Expressions that have identity are called *glvalue* expressions (generalised *lvalues*). Thus, *xvalues* and *lvalues* are *glvalues*. The expressions that can be moved from are called *rvalue* expressions. Both *prvalues* and *xvalues* are *rvalue* expressions.

Examples of *lvalue* expressions are: `a = b` and string literals. Examples of *prvalue* expressions are: `42`, `true`, a cast expression to a non-reference type (for example, `static_cast<double>(x)`), `&a` (address of variable `a`) and a lambda expression. Examples of *xvalue* expressions are: a cast expression to an *rvalue* reference type (for example, `static_cast<double&&>(x)`) and a function call.

We shall give some concrete examples of these types and categories in Section 4.3.4.

4.3 `decltype` SPECIFIER

The `decltype` specifier inspects the declare type of an entity or the type and value category of an expression. It can be seen as a metafunction in the sense that it accepts an entity or expression as input and produces the type of that entity or expression as output. The specifier is useful when declaring types that are difficult or impossible to declare using standard notation, for example lambda-related types or types that depend on template parameters. We expect `decltype` to be useful when creating code, classes and libraries involving mixed-mode arithmetic.

4.3.1 Initial Examples

We take some examples of mixed-mode arithmetic using built-in types:

```
int i=20;                  // Create an int.
double d=3.16;             // Create a double.

decltype(i) j=i;           // j is an int.
decltype(d) k=d;           // k is a double.

// Promote float * int to double
float f = 3.3F;
int n= 4;
decltype(d) d2 = f*n;
```

In the last example we are stating that variable `d2` should be of type `double`. We can check the validity of this statement by calling `std:: is_same` type trait (that we discuss in more detail in Chapter 6):

```
std::cout << "Check decltype:  " << std::is_same<double,
decltype(d)>::value; // true
```

(using the same variables as above).

Another example using the same variables as above is:

```
// What is the result of int*double. With decltype we
// don't need to know anymore.
decltype(j*k) l=j*k; // Result of int*double is double.
std::cout<<j<<"*"<<k<<"="<<l<<std::endl;

std::cout << "Check decltype:  "
          << std::is_same<double, decltype(j*k)>::value;   // true
```

Finally, `decltype` can be applied to lambda functions as the following code shows:

```
// decltype and lambda functions
auto f2 = [](double x, double y) -> int
{
   int result = std::floor(x + y);
   return result;
};

auto f3 = [](double x, double y) -> double
{
   double result = x + y;
   return result;
};


// Lambda function types are unique and unnamed
std::cout << std::boolalpha
           << "Type of lambda function, not its return type:  "
           << std::is_same<int, decltype(f2)>::value;   // false
```

```
decltype(f2) g = f2;
std::cout << std::boolalpha
          << "Type of lambda functions g and f2:  "
          << std::is_same<decltype(g),decltype(f2)>::value;  // true
std::cout << "g, f2 at (2.5, 3.6): "
          << g(2.5, 3.6) << ", " << f2(2.5, 3.6) << "\n";

std::cout << std::boolalpha
          << "Type of lambda functions f3 and f2:  "  // false
          << std::is_same<decltype(f3), decltype(f2)>::value;
```

Thus, in the case of lambda functions we are referring to their return type.

4.3.2 Extended Examples

Having given some illustrative examples of using `decltype` we now discuss its application to user-defined types. To this end, we create a number of classes to model units for ampere `A` (SI unit for current), volt `V` (SI unit for electric potential) and watt `W` (SI unit for power) and in particular the determination of the correct type when we multiply voltage and current, that is `W = V*A = A*V` (Wildi, 1995). The code is easy to understand. However, the ideas can be generalised; a good example is the *Boost Units* library. According to the library's online documentation:

> *This library is a C++ implementation of dimensional analysis in a general and extensible manner, treating it as a generic compile-time metaprogramming problem. Support for units and quantities (defined as a unit and associated value) for arbitrary unit system models and arbitrary value types is provided. Complete SI and CGS unit systems are provided along with systems for angles measured in degrees and radians, as well as systems for temperatures measured in the degrees Kelvin, Celsius and Fahrenheit.*

A discussion of this library is outside the scope of this book.
The user-defined types in the current discussion are (for motivational purposes):

```
class Unit
{

protected:
   // Protected default constructor.
   Unit(){}

public:

   // Get the name of the unit. Implemented in derived class.
   virtual const std::string& Name() const=0;

   // Output the unit to the given ostream.
   friend std::ostream& operator << (std::ostream& os, const Unit& u)
   {
     os<<u.Name();
```

```
      return os;
   }
};

// Forward class declarations.
class Watt;
class Volt;
class Ampere;

// Watt unit class.
class Watt: public Unit
{
private:
   static const std::string s_name;

public:
   Watt(): Unit() {}
   const std::string& Name() const { return s_name; }
};

// The name of the "Watt" unit.
const std::string Watt::s_name="Watt";


// Volt unit class.
class Volt: public Unit
{
private:
   static const std::string s_name;

public:
   Volt(): Unit() {}
   const std::string& Name() const { return s_name; }

   // Volt*Ampere=Watt
   Watt operator * (const Ampere& a) const { return Watt(); }
};

// The name of the "Volt" unit.
const std::string Volt::s_name="Volt";


// Ampere unit class.
class Ampere: public Unit
{
private:
   static const std::string s_name;

public:
   Ampere(): Unit() {}
   const std::string& Name() const { return s_name; }
```

```
    // Ampere*Volt=Watt
    Watt operator * (const Volt& v) const { return Watt(); }
};

// The name of the "Ampere" unit.
const std::string Ampere::s_name="Ampere";
```

In this case we have used operator overloading of the multiplication operator to multiply volt and ampere (in any order) to produce a unit of watt.

An example of use is:

```
// Volt*Ampere=Watt.
std::cout << "Watt, volt and ampere\n";
Volt v;
Ampere a;
decltype(v*a) result=v*a;
std::cout<<v<<"*"<<a<<"="<<result<<std::endl;
decltype(a*v) result2 = a*v;
std::cout << a << "*" << v << "=" << result2 << std::endl;
```

4.3.3 The Auxiliary Trait `std::declval`

We have seen how to use `decltype` to discover the type of a class instance, for example one that has been default created. But what if we cannot (or do not want to) create an instance? In that case we use `std::declval` to convert any type to a call reference type, thus making it possible to use member functions in `decltype` expressions without having to call constructors.

The definition is:

```
template<class T>
    typename std::add_rvalue_reference<T>::type declval();
```

and we use type traits rules as follows:

```
std::add_lvalue_reference<T&>::type is T&
std::add_lvalue_reference<T&&>::type is T&
std::add_rvalue_reference<T&>::type is T&
std::add_rvalue_reference<T&&>::type is T&&
```

Let us take an example of two classes, one of which has a default constructor and the other not. In one case (class `C1`) we cannot use `decltype` because its default constructor is not defined and can't be used and we must use `std::declval` but in the second case (class `C2`) we can use `decltype`.

The class and code in the first case are as follows:

```
struct C1
{ // Default ctor cannot be used by clients

    C1() = delete;
```

```
   int compute() { return 1; }
};

// No default constructor
// decltype(C1().compute()) n1 = 1;       // Error

//  Side-track ctor
decltype(std::declval<C1>().compute()) n1 = 1;
std::cout << std::boolalpha << "Same as int?:  "
          << std::is_same<decltype(n1),int>::value << "\n"; // true
```

The code in the second case is:

```
struct C2
{ // Default ctor can be used by clients

   C2() = default;

   int compute() { return 1; }
};

// Now a class with a default ctor
decltype(C2().compute()) n2 = 1;

//  Side-track ctor
decltype(std::declval<C2>().compute()) n3 = 1;
std::cout << std::boolalpha << "Same as int?:  "
          << std::is_same<decltype(n3),int>::value << "\n";  // true
```

Finally, for completeness we give an example of adding references to a type as this functionality is used in the definition of `std::declval`. In this case we define a non-reference type and we add references to it:

```
using NonRef = int;
using lref = std::add_lvalue_reference<NonRef>::type;
using rref = std::add_rvalue_reference<NonRef>::type;

// Answers: false, true, true
std::cout << std::is_lvalue_reference<NonRef>::value << '\n';
std::cout << std::is_lvalue_reference<lref>::value << '\n';
std::cout << std::is_rvalue_reference<rref>::value << '\n';
```

We discuss type traits in more detail in Chapter 6.

4.3.4 Expressions, *lvalues*, *rvalues* and *xvalues*

We give some examples of code to make the general discussion of Section 4.2.3 more concrete. Some of the examples use type traits:

```
void funcR(int&& z)
{ // r values and  related issues
```

```
   std::cout << "\nfunc with rvalue input\n";
   std::cout << std::boolalpha << "Check value ref:  "
      << std::is_reference<decltype(z)>::value << "\n";   // true
   std::cout << std::boolalpha << "Check l-value ref:  "
             << std::is_lvalue_reference<decltype(z)>::value << "\n";
   std::cout << std::boolalpha << "Check r-value ref:  "
             << std::is_rvalue_reference<decltype(z)>::value << "\n";

   std::cout << "\nMoved variable 2\n";
   auto&& v = std::move(z);
   std::cout << std::boolalpha << "Check value ref:  "
             << std::is_reference<decltype(v)>::value << "\n";
   std::cout << std::boolalpha << "Check l-value ref:  "
             << std::is_lvalue_reference<decltype(v)>::value << "\n";
   std::cout << std::boolalpha << "Check r-value ref:  "
             << std::is_rvalue_reference<decltype(v)>::value << "\n";
   std::cout << std::boolalpha << "Same as int?:  "
             << std::is_same<decltype(v), int>::value << "\n";
   std::cout << std::boolalpha << "Same as int?:  "
             << std::is_same<decltype(v), int&&>::value << "\n";
}
```

An example of use is:

```
int j = 19;
funcR(std::move(j));
```

Another set of examples is:

```
struct C
{
   C(double val, int valInt) : x(val) {}
   double x;
};

void ValueCategories()
{  // Value categories of expressions in C++11, decltype and type traits

   const C* c = new C{ 0.0, 1 };

   decltype(c -> x) y;            // type of y is double
   decltype((c->x)) z = y;        // type of z is const double&

   std::cout << std::boolalpha << "Check value ref:  "
             << std::is_reference<decltype(c->x)>::value << "\n";
   std::cout << std::boolalpha << "Check l-value ref:  "
             << std::is_lvalue_reference<decltype(c->x)>::value << "\n";
   std::cout << std::boolalpha << "Check r-value ref:  "
             << std::is_rvalue_reference<decltype(c->x)>::value << "\n";
```

```
    // Non-reference type
    std::cout << std::boolalpha << "Check value ref:  "
              << std::is_reference<decltype(y)>::value << "\n";
    std::cout << std::boolalpha << "Check l-value ref:  "
              << std::is_lvalue_reference<decltype(y)>::value << "\n";
    std::cout << std::boolalpha << "Check r-value ref:  "
              << std::is_rvalue_reference<decltype(y)>::value << "\n";

    // Reference type
    std::cout << std::boolalpha << "Check value ref:  "
              << std::is_reference<decltype(z)>::value << "\n"; // true
    std::cout << std::boolalpha << "Check l-value ref:  "
              << std::is_lvalue_reference<decltype(z)>::value << "\n";
    std::cout << std::boolalpha << "Check r-value ref:  "
              << std::is_rvalue_reference<decltype(z)>::value << "\n";
}
```

You can run this code and check the output.

4.4 LIFE BEFORE AND AFTER `decltype`

Many applications use heterogeneous data types that are combined and manipulated in various ways. In particular, we are interested in defining what the conversion rules are in mixed-mode floating-point operations and conversions. This discussion is of interest when developing libraries. It promotes the robustness of the code.

Some examples are:

- Matrix linear algebra operations on vectors and matrices and BLAS (*Basic Linear Algebra Subprograms*). The underlying data types can be a mixture of integers, floating-point types and complex numbers.
- Common types for *durations* and common type manipulation in the C++ *chrono* library.
- *Boost Units*, a C++ library for *dimensional analysis* that is an important technique in many branches of science and engineering (Hughes and Brighton, 1967; Wildi, 1995).

We begin with a simple example and define an operation (in this case multiplication) that accepts two generic input arguments, combines them in some way and then produces a generic return type. To this end, we consider the following alternatives:

- S1: Function object with one template parameter.
- S2: Function object with different template parameters for input arguments and return type.
- S3: Function object using `decltype` to *deduce* the return type.
- S4: Using `std::common_type<>`.

The corresponding code is easy to understand; we provide four different function objects to multiply generic types. It is recommended to read the corresponding code comments:

```
// S1: Function object class using only one template parameter for
// operator().
```

```
class MultiplyV1
{
public:
   template <typename T> T operator () (const T& v1, const T& v2)
   {
      return v1*v2;
   }
};

// S2: Function object class using different template parameters
// for inputs and output. The output template parameter is a
// class template instead of the function because then we do not
// specify the type when used as callback functor.
template <typename TReturn>
   class MultiplyV2
{
public:
   template <typename T1, typename T2> TReturn operator ()
   (const T1& v1,const T2& v2)
    {
      return v1*v2;
    }
};

// S3: Function object class using decltype to deduce the return
// type. No template argument needed for return type.Note that we
// use "auto" as return type. The decltype is at the end because
// only then v1 * v2 is declared. In this case "auto" says there
// is a trailing return type.
class MultiplyV3
{
public:
   template <typename T1, typename T2> auto operator ()
      (const T1& v1, const T2& v2) -> decltype(v1*v2)
   {
      return v1*v2;
   }
};

// S4: Free function to implement the binary operator *. Returns
// the common type of T1 and T2.
   template <typename T1, typename T2>
   typename std::common_type<T1, T2>::type Multiply4 (const T1& v1,
   const T2& v2)
{
      return{ v1 * v2 };
};
```

The first two classes represent how C++03 would implement the problem while the third and fourth solutions represent the preferred C++11 functionality. In the fourth case we give examples of how the compiler handles heterogeneous data types:

```
double d1 = 2.0; float f = -3.13F;
std::cout << "Product: " << Multiply4<double, float>(d1, f);

std::complex<double> c1(1, 1); std::complex<double> c2(2, 2);
std::cout << "Complex Product: " << Multiply4(c1,c2) << '\n';
std::cout << "Complex Product: " << Multiply4(c1, d1) << '\n';
std::cout << "Complex Product: " << Multiply4(c1, static_cast
                                  <double>(f)) << '\n';
```

We see that the responsibility for type checking is done by the compiler.

4.4.1 Extending the STL to Support Heterogeneous Data Types

We conclude this section with a discussion and examples to show how to extend the functionality of STL data types and algorithms. In general, STL algorithms use a single data type and hence calling them with different types will result in a compiler error, as the following examples show:

```
// STL algorithms and extensions
std::cout << "STL algos, scary conversions\n";
double a = 2.0; double b = 3.0; float c = 3.9F;
int d = -13;
std::cout << std::min(a, b) << '\n';                        // OK
std::cout << std::min<double>(a, b) << '\n';                // OK
//  std::cout << std::min(a, c) << '\n';                    // NOT OK
std::cout << std::min<double>(a, c) << '\n';                // OK
std::cout << std::min<float>(a, c) << '\n';                 // OK
std::cout << "int? " << std::min<int>(a, d) << '\n';        // OK
```

The issue that we see here is that STL has difficulties with heterogeneous data types in some cases and some data conversion seems to be taking place. We now use C++11 syntax to define a new function in two different ways, as follows:

```
// Algorithm type functions
template <typename T1, typename T2> auto minNew (const T1& v1,
                    const T2& v2) -> decltype(v1*v2)
{
   if (v1 < v2)
      return v1;
   return v2;
};

template <typename T1, typename T2> auto maxNew(const T1& v1, const T2& v2)
{
   using Common = typename std::common_type<T1, T2>::type; // for
                                                           // readability
   return std::max<Common>(v1, v2);
};
```

Some test code is:

```
double d1 = 1.0; double d2 = 20.4;
double f = +1.6F;
std::cout << "Min\n";
std::cout << minNew(d1, d2) << '\n';
std::cout << minNew(d1, f) << '\n';
std::cout << minNew(f, d2) << '\n';
int i = 1; double d3 = -1.0001;
std::cout << minNew(i, d3) << '\n';
```

The examples in this section were taken for pedagogical reasons. The situation could change however, for more complicated algorithms and cases in which pre- and post-processing needs to be carried out on the input arguments and on the algorithm's return value, respectively.

We now take a more extended example to compute the inner product of two STL containers, for example:

```
// Sums of products + initVal
std::vector<double> vec{ 1.0, 2.0, 3.0, 4.0 };
std::vector<double> vec2{ 4.0, 3.0, 2.0, 1.0 };

double initVal = 0.0;
double ip1 = std::inner_product(std::begin(vec), std::end(vec),
std::begin(vec2), initVal);
std::cout << "Inner product STL: " << ip1 << std::endl;
```

We now wish to create a user-defined function that can handle heterogeneous data types and that uses `std::inner_product` as internal algorithm. To this end, we create two static member functions, each one delivering a different implementation of the same functionality (note that we use *template–template parameters*):

```
template < typename T1, typename T2,
   template <typename TC1, typename TAlloc1> class Container1 =
   std::vector,
   template <typename TC2, typename TAlloc2> class Container2 =
   std::vector,

   typename TAlloc1 = std::allocator<T1>, typename TAlloc2 =
   std::allocator<T2>>

class Algorithms
{ // Wrappers/Composition for STL algorithms

public:
   static auto InnerProduct(const Container1<T1, TAlloc1>& con1,
   const Container2<T2, TAlloc2>& con2, const T1& initVal)
           -> typename std::common_type<T1, T2>::type
   {

      // Possible preprocessing on T1 and T2

      // Body
```

```
        return std::inner_product (std::cbegin(con1),
              std::cend(con1), std::cbegin(con2), initVal);

        // Possible postprocessing on T1 and T2
    }

     static auto InnerProduct_II(const Container1<T1, TAlloc1>& con1,
     const Container2<T2, TAlloc2>& con2, const T1& initVal)
             -> decltype(std::declval<T1>()*std::declval<T2>())
     { // Using decltype and declval

        // Possible preprocessing on T1 and T2

        // Body

         return std::inner_product (std::begin(con1), std::end(con1),
                                    std::begin(con2), initVal);

        // Possible postprocessing on T1 and T2
     }
};
```

This is a generic class and we can call it with a range of arithmetic data types and containers, for example:

```
std::cout << "Inner product extended STL: "
   << Algorithms<double,double>::InnerProduct(vec,vec2, initVal);

std::vector<int> vec3{ 1, 2, 3, 4 };
std::vector<float> vec4{ 4.0F, 3.0F, 2.0F, 1.0F };
std::cout << "Inner product extended STL: "
           << Algorithms<int,float>::InnerProduct(vec3, vec4, initVal);

std::list<int> myList1{ 1, 2, 3, 4 };
myList1.reverse();
initVal = 0;
std::cout << "Inner product extended STL: "
           << Algorithms<int, int, std::vector, std::list>::InnerProduct
            (vec3, myList1, initVal) << std::endl;

std::list<int> myList2{ 4, 3, 2, 1};
myList2.reverse();
initVal = 0;
std::cout << "Inner product extended STL: "
           << Algorithms<int, int, std::list, std::list>::InnerProduct
            (myList2, myList1, initVal) << std::endl;

std::cout << "Inner product extended STL: "
           << Algorithms<int, double, std::list>::InnerProduct
              (myList2, vec2, initVal) << std::endl;
```

```
// Using decltype syntax
std::cout << "decltype\n";
std::cout << "Inner product extended STL II: "
           << Algorithms<int, int, std::list, std::list>::InnerProduct_II
            (myList2, myList1, initVal) << std::endl;
std::cout << "Inner product extended STL II: "
           << Algorithms<int, double, std::list>::InnerProduct_II
            (myList2, vec2, initVal) << std::endl;
```

See also Exercise 6 in this chapter where we elaborate on this and related topics.

4.5 `std::result_of` AND SFINAE

In general terms, a *callable* type is one for which the INVOKE operation as defined by `std::function`, `std::bind` or `std::thread::thread` is applicable and it may be performed explicitly using the library function `std::invoke` (C++17). In general, a type `T` is *callable* if we have an object `f` of type `T`, a suitable list of argument types `ArgTypes` and a suitable return type `R`:

```
INVOKE(f, std::declval<ArgTypes>()..., R)
```

This syntax is easy to interpret if we consider it as invoking a function `f` with a variable number of arguments `ArgTypes` and returning a type `R`. It is also possible to determine if a function is callable by using `std::is_callable` (C++17).

The type trait `std::result_of<f,ArgTypes>` represents the type of calling `f` with argument types `ArgTypes`:

```
template< class T >
    using result_of_t = typename result_of<T>::type;
```

Behaviour is undefined if the corresponding INVOKE expression is ill-formed, in other words if `f` is not a callable type. The rationale is to determine the result of invoking a *callable*, in particular if its return type is different for different sets of arguments.

We take some examples. First, we define an empty class `X` and a callable object `S` and two free functions `Func` and `Func2`:

```
struct X {};

struct S
{
   double operator () (int n, int m) { return 3.14; }
   float operator () (float f) { return f; }
   int operator () (double x) { return std::floor(x); }

   X func(int n) { return X(); }
};
```

```
int Func()
{
   return 42;
}

double Func2(int n)
{
   return static_cast<double>(n);
}
```

We now ascertain the return types by using type traits as follows:

```
std::result_of<S(int, int)>::type d = 2.71;
static_assert(std::is_same<decltype(d), double>::value, "ouch");

// Compile error, wrong types
// static_assert(std::is_same<decltype(d), float>::value, "ouch");
// Compile error


// Member function return type
std::result_of<decltype(&S::func)(S, int)>::type  x = X();
static_assert(std::is_same<decltype(x), X>::value, "ouch");

// Free function, need a reference
std::result_of<decltype(&Func)()>::type  n = 3;
static_assert(std::is_same<decltype(n), int>::value, "ouch");

std::result_of<decltype(&Func2)(int)>::type  d2 = 3.14;
static_assert(std::is_same<decltype(n), int>::value, "ouch");
```

SFINAE (*Substitution Failure is Not an Error*) is a feature used in template metaprogramming and it is applied during overload resolution of function templates. Specifically, if substituting the deduced type for the template parameter fails then the specialisation is discarded from the overload set instead of causing a compiler error. In simpler terms, it refers to a situation in C++ where an invalid substitution of template parameters *is not in itself an error*. If an error occurs during the substitution of a set of arguments for any given template then the compiler removes the potential overload from the candidate set instead of producing a compiler error.

We give a simple example of how SFINAE works. First, consider the following class and free function:

```
struct Wrapper
{
   typedef std::vector<double> Data;
};

template <typename T>
   void f(typename T::Data)
{ // #1
```

```
   std::cout << "Has wrapped data\n";
}
```

The following code will then compile:

```
f<Wrapper>(std::vector<double>(20)); // Call #1.
```

However, the following code will not compile because `int` type does not have an embedded vector as does `Wrapper`:

```
f<int>(10);
```

The error produced is something similar to the following:

```
Error C2770 invalid explicit template argument(s) for 'void f(T::foo)'
```

We can avoid the compilation error by creating the following function:

```
template <typename T> void f(T)
{ // #2

   std::cout << "Has NOT wrapped data\n";
}
```

SFINAE was introduced to avoid creating ill-formed programs and in general you will probably not need to use it directly in your code. Many type traits (discussed in Chapter 6) are implemented using SFINAE. It can be seen as an example of *compile-time introspection*.

4.6 `std::enable_if`

We now discuss `std::enable_if`; it is a metafunction and its specification is:

```
template< bool B, class T = void > struct enable_if;
```

This yields type `T` only if `bool B` is true.

This piece of exotic syntax has its roots in Boost and it is now also supported in C++. The `enable_if` family of templates is a set of tools to allow a function template or a class template specialisation to include or exclude itself from a set of matching functions or specialisations based on properties of its template arguments. For example, one can define function templates that are only enabled for, and thus only match, an arbitrary set of types defined by a traits class. The `enable_if` templates can also be applied to enable class template specialisations. In short, `enable_if` allows us to manage the appropriateness of argument types in a non-intrusive way at compile-time (Abrahams and Gurtovoy, 2005).

The first example is code to execute some kind of *command* depending on the data type, in this case simple print functions for built-in types representing integers and floating-point numbers. The two implementations are:

```
// Compile-time Command pattern

template <typename T>
   void print(typename std::enable_if <std::is_integral<T>::value,
   T>::type i)
{
   std::cout << "Integral: " << i << '\n';
}

template <typename T>
   void print(typename std::enable_if <std::is_floating_point<T>::value,
   T>::type f)
{
   std::cout << "Floating point: " << f << '\n';
}
```

We can call this function with any integral or floating-point argument; any other argument type will result in a compiler error:

```
// Commands
print<short>(1);
print<long>(2);
print<double>(3.14);
//print<std::string>(std::string("abcd")); // Error
```

It is possible to extend this function to user-defined types:

```
struct SomeClass {};

template <typename T>
   void print(typename std::enable_if <std::is_same<T,
   SomeClass>::value, T>::type p)
{
   std::cout << "SomeClass type: " << '\n';
}

// User-defined type
SomeClass c;
print<SomeClass>(c);
```

Another application is to write code to create algorithms whose implementations depend on the specific data types in use. This is similar to the popular *Strategy* design pattern (GOF, 1995). To this end, we take the toy example of computing the distance between instances of some class. The focus in this case is on the encapsulation of an algorithm that computes a quantity of interest:

```
// Strategy
struct P1
{
   P1() {}
   int distance(int i, int j) { return i + j; }
};

struct P2
{
   P2() {}
   int distance(int i, int j) { return  std::max<int>(i,j); }
};

struct P3
{
   P3() {}
   int distance(int i, int j) { return  std::min<int>(i, j); }
};

template <typename T>
   int distance(typename std::enable_if
      <std::is_same<T, P1>::value, T>::type p, int i, int j)
{
   return p.distance(i, j);
}

template <typename T>
   int distance(typename std::enable_if
      <std::is_same<T, P2>::value, T>::type p, int i, int j)
{
   return p.distance(i, j);
}
```

We can call these functions as follows (note that the code does not compile for the case of class `P3` because the free function `distance` is not defined for class `P3`):

```
// Algorithm style
P1 p1;
std::cout << "Distance 1: " << distance<P1>(p1, 1, 2) << '\n';

P2 p2;
std::cout << "Distance 2: " << distance<P2>(p2, 1, 2) << '\n';

P3 p3;
//std::cout << "Distance 3: " << distance<P3>(p3, 1, 2) << '\n';
```

Finally, we give the example of the compile-time equivalent of the *Prototype* creational design pattern (GOF, 1995). It is used when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects.

This pattern is used to:

- Avoid subclasses of an object creator in the client application.

- Avoid the inherent cost of creating a new object in the standard way (e.g. using the `'new'` keyword) when it is prohibitively expensive to do so.

Again, we can use `std::enable_if` to achieve the same results. We take an example from both Boost and C++ to create objects depending on a given data type:

```
// A. Boost version
template <typename T>
   typename boost::enable_if <std::is_same<T, int>, T>::type create()

{
   return 1;
}

template <typename T>
   typename boost::enable_if <std::is_same<T, std::string>,
   T>::type create()
{
   return "A string is born";
}

// B. C++ version
template <typename T>
   typename std::enable_if <std::is_same<T, double>::value,
   T>::type create()
{
   return 2.71;
}


struct C
{
   C() { }
   friend std::ostream& operator << (std::ostream& os,const C& c)
         { os << "A new C";  return os; }
};

template <typename T>
   typename std::enable_if <std::is_same<T, C>::value, T>::type create()

{
   return C();
}
```

An example of use is:

```
std::cout << create<std::string>() << '\n';
std::cout << create<double>() << '\n';
std::cout << create<C>() << '\n';
C myC = std::move(create<C>());
std::cout << myC << '\n';
```

4.7 BOOST enable_if

Boost has an extended version of `std::enable_if`. The header `<boost/utility/enable_if.hpp>` contains the following classes:

```
namespace boost
{
   template <class Cond, class T = void> struct enable_if;
   template <class Cond, class T = void> struct disable_if;
   template <class Cond, class T> struct lazy_enable_if;
   template <class Cond, class T> struct lazy_disable_if;

   template <bool B, class T = void> struct enable_if_c;
   template <bool B, class T = void> struct disable_if_c;
   template <bool B, class T> struct lazy_enable_if_c;
   template <bool B, class T> struct lazy_disable_if_c;
}
```

A full discussion of these classes is outside the scope of this chapter and for more information we refer the reader to the Boost C++ online documentation. However, we try to give an overview of the main functionality.

The names of the `enable_if` templates have three parts:

- An optional `lazy_` tag (avoids instantiating part of a function signature unless an enabling condition is true).
- The mutually exclusive options `enable_if` or `disable_if` that indicate that the `true` condition should enable or disable the current overload, respectively.
- An optional `_c` tag that indicates if the condition argument is a `bool` value (`_c` suffix) or a type containing a static `bool` constant names `value` (no suffix).

The definitions of `enable_if_c` and `enable_if` are:

```
template <bool B, class T = void>
   struct enable_if_c
{
      typedef T type;
};

template <class T>
    struct enable_if_c<false, T> {};

template <class Cond, class T = void>
    struct enable_if : public enable_if_c<Cond::value, T> {};
```

We revisit the final example in Section 4.6 to show how the code is implemented in Boost:

```
// A. Boost version
template <typename T>
   typename boost::enable_if_c <std::is_same<T, double>::value,
   T>::type create()
{ // _c form
```

```
   return 1;
}

template <typename T>
   typename boost::enable_if_c <std::is_same<T, std::string>::value,
   T>::type create()
{ // _c form

   return "A string is born";
}

template <typename T>
   typename boost::enable_if <std::is_same<T, C>, T>::type create()
{ // not _c form

   return C();
}
```

The syntax has two forms, the latter one being similar to the C++ solution.

Finally, in some cases (for example, when creating a C++ library) we may wish to enable or disable functions by including or excluding template specialisations of a given type, for example arithmetic types:

```
// Enable and disable for certain data types
template <typename T>
   typename boost::enable_if_c <std::is_arithmetic<T>::value, T>::type
ArithmeticCompute(T t)
{ // Only for arithmetic types

   return t*t;
}

template <typename T>
   typename boost::disable_if_c <std::is_arithmetic<T>::value, T>::type
NonArithmeticCompute(T t)
{ // Only for non-arithmetic types

   return t;
}
```

In other words, the first function enables arithmetic types and the second function disables arithmetic types.

An example of use shows how to ensure that functions only compile with objects of the 'correct' type:

```
// Enable and disable for certain data types
double d = 1.3;
std::cout << ArithmeticCompute(d) << '\n';        // OK
```

```
std::string s("1.3");
//std::cout << ArithmeticCompute(s) << '\n';      // Error

//std::cout << NonArithmeticCompute(d) << '\n';   // Error

std::cout << NonArithmeticCompute(s) << '\n';     // OK
```

A final example: many numerical processes need to support both floating-point and complex numbers. Let us suppose that we develop an algorithm and that we wish to restrict the data types to `double` and `std::complex<double>`. Then the following code satisfies this requirement:

```
template <typename T> typename boost::enable_if_c
   <std::is_same<T, double>::value
   || std::is_same<T, std::complex<double>>::value, T>::type
Exponential(T t)
{ // Only for double or complex<double>

   return std::exp(t);
}
```

An example of use is:

```
// Exponential for double or complex<double>
double x = 3.2;
std::complex<double> c(1.0, 2.0);
float f = 3.0F;
std::cout << Exponential<double>(x) << '\n';                    // OK
std::cout << Exponential<std::complex<double>>(c) << '\n';      // OK
//std::cout << Exponential<float>(f) << '\n';                   // Error
```

Here we also see that `float` is not allowed and a compiler error will result if we try to use it. More generally, the function `Exponential` only accepts doubles and complex numbers.

As a concluding remark, we note that many of the language features in C++ started life as one or more Boost libraries. In general, Boost tends to offer more extensive functionality, examples and documentation than C++ at the moment of writing. For this reason we recommend keeping up to date with developments in Boost and its online documentation.

4.8 `std::decay()` TRAIT

This is another piece of exotic syntax in C++. It can best be understood as a function that transforms a given type into a simpler type. In general terms, it removes *extraneous* or other secondary information from a type. The transformed type (called the *decay type*) is aliased as `std::decay::type`. The conversion resembles implicit conversions that take place when an argument is passed by value to a function.

The rules are:

- If `T` is a function type, a function-to-pointer conversion is applied and the decay type is the same as `add_pointer<T>::type`.
- If `T` is an array type, an array-to-pointer conversion is applied and the decay type is the same as `add_pointer<remove_extent<remove_reference<T>::type>::type>::type`. In plain language we remove references and array indexes from the type and we add a pointer to the type as a sequence of actions.
- Otherwise, a regular *lvalue-to-rvalue* conversion is applied and the decay type is the same as `remove_cv<remove_reference<T>::type>::type`. Incidentally, we shall discuss these 'remove' functions in more detail in Chapter 6.

We remark that `std::decay()` obtains a type using another type as model but it does not transform values nor objects between those types. It is a *higher-order function* because it operates on a function and it also returns a function. It can be seen as the C++ equivalent of *paint stripper* (in the author's opinion).

We take an example. We first define some type aliases:

```
using A = std::decay<int>::type;                    // int
using B = std::decay<int&>::type;                   // int
using C = std::decay<int&&>::type;                  // int
using D = std::decay<const int&>::type;             // int
using E = std::decay<int[2]>::type;                 // int*
using F = std::decay<int (int)>::type;              // int(*)(int)

using G = std::decay<volatile int>::type;           // int
using H = std::decay<const volatile int&>::type;    // int
```

Second, we compare the resulting decayed types against the type `int`. You should run the code and compare the output with your expectations and experiment with user-defined types.

4.9 A SMALL APPLICATION: QUANTITIES AND UNITS

We conclude this chapter with a small application to model dimensions, units and quantities. For example, we would like to define the value 100 as a length (possibly miles) or as a temperature (for example, degrees Celsius). How do we achieve this? To this end, we take an example from Abrahams and Gurtovoy (2005). Before we do so we cite an ill-fated project that failed because one contactor used SI units while the other contactor used non-SI units:

> *The Mars Climate Orbiter was a 338-kilogram (745 lbs.) robotic space probe launched by NASA on December 11, 1998 to study the Martian climate, Martian atmosphere, and surface changes and to act as the communications relay in the Mars Surveyor '98 program for Mars Polar Lander. However, on September 23, 1999, communication with the spacecraft was lost as the spacecraft went into orbital insertion, due to ground-based computer software which produced output in non-SI units of pound (force)-seconds (lbf·s) instead of the SI units of newton-seconds (N·s)*

specified in the contract between NASA and Lockheed. The spacecraft encountered Mars on a trajectory that brought it too close to the planet, causing it to pass through the upper atmosphere and disintegrate.

We give a short introduction to the topic of *dimensional analysis*. This theory is concerned with the analysis of the relationships between different physical quantities by identifying their fundamental *dimensions* (for example, length, mass and time) and *units of measure* (for example, miles versus kilometres and pounds versus kilograms). We also wish to convert between dimensional units and track dimensions as calculations and comparisons are performed.

A *base dimension* is some entity that we wish to measure, for example, mass, length and time. Base dimensions are essentially tags and they do not provide dimensional analysis functionality as such. In C++ we can model both base and composite dimensions using the functionality from *Boost.MPL*:

```
#include <boost/mpl/vector_c.hpp>

// Base dimensions
using mass =
   boost::mpl::vector_c<int, 1, 0, 0, 0, 0, 0, 0>;       // M
using length =
   boost::mpl::vector_c<int, 0, 1, 0, 0, 0, 0, 0>;       // L
using time =
   boost::mpl::vector_c<int, 0, 0, 1, 0, 0, 0, 0>;       // T
using charge =
   boost::mpl::vector_c<int, 0, 0, 0, 1, 0, 0, 0>;
using temperature =
   boost::mpl::vector_c<int, 0, 0, 0, 0, 1, 0, 0>;
using intensity =
   boost::mpl::vector_c<int, 0, 0, 0, 0, 0, 1, 0>;
using angle =
   boost::mpl::vector_c<int, 0, 0, 0, 0, 0, 0, 1>;

// (Composite) dimensions
using force =
   boost::mpl::vector_c<int, 1, 1, -2, 0, 0, 0, 0>;      // ML/T^2
using velocity =
   boost::mpl::vector_c<int, 0, 1, -1, 0, 0, 0, 0>;      // L/T
using acceleration =
   boost::mpl::vector_c<int, 0, 1, -2, 0, 0, 0, 0>;      // L/T^2
using dynamicViscosity =
   boost::mpl::vector_c<int, 1, -1, -1, 0, 0, 0, 0>;     // M/(LT)
```

These types are pure metadata. In order to be able to perform computations we create a class to model *quantities*. The dimensions of the argument to be added or subtracted must always match, otherwise we will get a compiler error:

```
// TMP Abrahams and Gurtovoy page 40
template <class T, class Dimensions>
   struct Quantity
```

```
{ // Simple class to show the application of dimensions

   explicit Quantity(T x): m_value(x) {}

   T value() const { return m_value; }

private:
   T    m_value;
};


template <class T, class D> Quantity<T, D>
   operator+(const Quantity<T, D>& x, const Quantity<T, D>& y)
{
   return Quantity<T, D>(x.value() + y.value());
}

template <class T, class D> Quantity<T, D>
   operator-(const Quantity<T, D>& x, const Quantity<T, D>& y)
{
   return Quantity<T, D>(x.value() - y.value());
}
```

Some compilable and non-compilable examples are:

```
// Quantity
Quantity<float, length> q1(1.0f);
Quantity<float, length> q2(2.0f);
Quantity<double, length> q3(1.0f);
Quantity<double, length> q4(2.0f);
Quantity<double, velocity> q5(2.0f);

q1 = q1 + q2;   // OK
std::cout << "Q1: " << q1.value() << '\n';

// q1 = q2 + Quantity<float, mass>(3.7f);   // Error
q4 = q3 + q3;                               // OK
// q5 = q4 + q3;                            // Error
```

We see that semantically incorrect computations will result in a compiler error and no unexpected implicit conversions can take place.

Dimensional analysis is not discussed much in the financial literature; some examples however are given in Kyle and Obizhaeva (2016) and Lewis (2016). We take a final example from Lewis (2016) for the case of the *variance gamma* (VG) process that is a Lévy process determined by a random time change. The process has finite moments distinguishing it from many Lévy processes. There is no diffusion component in the VG process and it is thus a *pure jump process*. In this case we quote the dimensional analysis result that computes the spectrum

of the VG process that is associated with a certain *partial integro-differential equation* (PIDE). The spectral value is:

$$\lambda_n = \frac{1}{\vartheta} \times \Lambda_n \left(\frac{\sigma^2 \vartheta}{L^2} \right), \quad n = 1, 2, \ldots$$

where the parameters and their dimensions are given by:

(ϑ, σ) = parameters of VG process

$[\vartheta] = [T],\ [\sigma^2] = [X^2/T]$

$[X]$ = dimension of the spatial coordinate

L = upper boundary of PIDE problem

$[L] = [X]$

λ_n = eigenvalue of PIDE spectral problem, $[\lambda_n] = [1/T]$

Λ_n = scaling function

Summarising again, we see that the scaling function is dimensionless.

4.10 CONCLUSIONS AND SUMMARY

We have given a detailed exposition of a number of new and advanced topics in C++ that fall under various names such as *template metaprogramming* and *generic programming*. An understanding of these techniques is crucial if we wish to write flexible and robust code.

We shall discuss how to apply generic programming techniques to numerical analysis and computational finance in later chapters of this book.

4.11 EXERCISES AND PROJECTS

1. (Implicit Conversions '101')

Review and run the code that implements the cases A1, A2, A3 and A4 in Section 4.2.1.

Answer the following questions:

a) Determine which compiler errors and/or warnings are generated by this code. Determine the seriousness of these errors.

b) Determine how to modify the code in cases in which these compiler errors and/or warnings should be removed.

c) Explain the output from this code:

```
std::cout << AddIntegers(4, 10, 'A', true, false) << '\n';   // 76
```

2. (Variadic Functions)

The goal of this exercise is to compute the standard deviation of the arguments of a variadic function using the functionality in Section 4.2.2. We propose two phases; first create a `va_list` called `args1` and use it to compute the mean, then copy `args1` using `va_copy` to compute the standard deviation of the input arguments.

3. (Value Categories)

Determine by inspection if the following expressions are *xvalue*, *lvalue* or *prvalue*:

a) `a ? b : c` (ternary conditional expression for some `a`, `b` and `c`).

b) `a+b`, `a%b`, `&a`.

c) "Hello world".

d) `nullptr`.

e) `++ a`, `--a`.

f) `a++`, `a--`.

Can you use type traits to answer this question as well?

4. (Code Inspection)

The objective of this exercise is to inspect the code in Section 4.3.4 in order to understand it. The questions are:

a) Inspect each line and determine if the result is `true` or `false`; for example:

```
std::cout << std::boolalpha << "Check value ref:  "
          << std::is_reference<decltype(z)>::value << "\n";  // true
```

b) Run the code and check the output with the answers in part a). Did you get the correct answers?

5. (References)

Let `A` be an empty class. Determine which of the following types are references, *lvalue* references or *rvalue* references: `A`, `A&`. What about `A&&`, `int`, `int&`, `int&&`?

6. (C++11 and STL Algorithms)

In Section 4.4.1 we discussed how to extend STL algorithms using a number of features in C++. The objective of this exercise is to analyse and review the code to compute the inner product of two containers containing arithmetic data.

Answer the following questions:

a) Modify the code to check that the instantiated data is arithmetic. Use `static_assert` and the trait `std::is_arithmetic` (we discuss type traits in detail in Chapter 6).

b) We used the *template–template parameter* trick in Section 4.4.1 to allow us to define an algorithm that works with a range of generic containers. The STL specification on the other hand is:

```
template< class InputIt1, class InputIt2, class T >
      T inner_product(InputIt1 first1, InputIt1 last1,
      InputIt2 first2, T value);
```

Compare these contrasting approaches with regard to understandability, robustness and extendibility. For example, we may wish to parallelise the code or to introduce code to reduce round-off errors.

c) STL implements a generalised algorithm to compute the inner product of two vectors; an example is:

```
// Using two predefined function objects using inner and outer
// operations
std::vector<int> vecInt{ 1, 2, 3, 4};
```

```
int ip2 = std::inner_product(
            std::begin(vecInt), std::end(vecInt),
            std::begin(vecInt), 1, std::multiplies<int>(),   // inner
            std::plus<int>());                               // outer
std::cout << "Inner product2: " << ip2 <<std::endl;
```

The objective of this part is to generalise this code to support heterogeneous data types as explained in Section 4.4.

d) Test the code in Section 4.4 with a variety of data types (`int`, `float`, `double`), containers (`std::vector`, `std::list`) and default input arguments to the function `InnerProduct`. What happens if you try to compute the inner product of two string containers?

7. (Advantages of `std::enable_if`)

Which of the following statements can be considered useful features of `std::enable_if`?:

a) More user-friendly error messages than when using 'raw' (unrestricted) template parameters.

b) Its use can lead to more robust code.

c) It restricts templates to types that have certain properties.

d) Its use reduces the amount of boilerplate code that needs to be written.

8. (Overlapping Enabler Conditions)

Function templates with enabling conditions that are not mutually exclusive can lead to ambiguities. Once the compiler has examined the enabling conditions and included the function into the overload resolution set, normal C++ overload resolution rules are then used to select the best matching function. Examine how this can occur based on the following example and how to resolve the ambiguity:

```
template <class T>
    typename enable_if<boost::is_integral<T>, void>::type
       foo(T t) {}

template <class T>
    typename enable_if<boost::is_arithmetic<T>, void>::type
       foo(T t) {}
```

(*hint:* all integral types are also arithmetic).

9. (Ackermann Function)

Consider the Ackermann function:

$$A(m,\ n) = \begin{cases} n+1 \text{ if } m = 0 \\ A(m-1, 1) \text{ if } m > 0 \text{ and } n = 0 \\ A(m-1,\ A(m,\ n-1)) \text{ if } m > 0 \text{ and } n > 0 \end{cases}$$

Implement this definition in C++ and experiment with the following examples:

$$A(1,2) = 4$$
$$A(4,3) = 2^{2^{65536}} - 3$$
$$A(1,3) = 5.$$

Compute $A(1, 3)$ by hand. How many steps are there?
Investigate what happens in the following code and why:

```
std::cout << Ackermann(1,3) << std::endl;
std::cout << Ackermann(1,2) << std::endl;

// Stack overflow and program 'silently' stops
try
{
    std::cout << Ackermann(4,3) << std::endl;
}
catch (std::exception& e)
{
    std::cout << e.what() << std::endl;
}
```

Now implement the Ackermann function as a metafunction.

10. The *Reynolds number* (Re) is an important dimensionless quantity in fluid mechanics that is used to help predict flow patterns in different fluid flow situations. It is widely used in applications ranging from liquid flow in a pipe to the passage of air over an aircraft wing. It is also related to approximating the solutions of partial differential equations in finance using the finite difference method, as we shall see in later chapters (correction-dominated problems). The number is defined by:

$$\frac{\rho v L}{\mu}$$

where:

- ρ is the density of the fluid (kg/m^3)
- v is a characteristic velocity of the fluid with respect to the object (m/s)
- L is a characteristic linear dimension (m)
- μ is the dynamic viscosity of the fluid (kg/(m·s)).

Prove that the Reynolds number is a dimensionless quantity.

11. (Factorials)
Write a recursive metafunction to compute factorials:

$$n! = 1 \text{ if } n = 0$$
$$n! = n(n-1)! \text{ if } n \geq 1.$$

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 5

Tuples in C++ and their Applications

5.1 INTRODUCTION AND OBJECTIVES

This chapter is devoted to a data type called tuple. A *tuple* (or *n-tuple*) is a fixed-size collection of elements. Pairs, triples and quadruples are tuples. In a programming language, a tuple is a data object containing other objects as elements or members. These elements may be of different types. In other words, tuples can hold *heterogeneous data types*.

We shall see some useful applications of tuples in later chapters.

5.2 AN `std::pair` REFRESHER AND NEW EXTENSIONS

In this section we pave the way for a discussion of `std::tuple` by reviewing the functionality in `std::pair` (which is a 2-tuple) and by giving examples of the new features in C++11. Much of the syntax translates directly to tuples which means that we only have to learn the syntax once and that we can later pay more attention to the design of tuples and to their applications in computational finance.

The features to be considered here are:

1. Create pairs in two ways.
2. Copy a pair by value and copy by reference.
3. Accessing elements of a pair.
4. Swapping and modifying pairs.
5. Pair comparison.
6. Piecewise construction and forwarding.
7. Compile-time access to a pair's elements.
8. *Tie* variables and tuple elements.
9. Pairs and nesting.

We now discuss each of these features by describing them in general terms, by giving easy examples and by placing strategic comments in the code. Of course, the best way to learn is to program the examples yourself!

In the following discussion we create a number of variables that will be used to illustrate the above nine use cases. In order to create pairs, we can choose between constructors and the

convenience function `std::make_pair`. We recall that `std::pair` is a struct with (public) member data `first` and `second`:

```
using value_type = double;
// Create pairs by calling constructors
std::pair<int, float> p1;
std::cout << p1.first << "," << p1.second << '\n';      // 0,0

std::pair<int, value_type> p2(65, 2.01f);
std::cout << p2.first << "," << p2.second << '\n';      // 65, 2.01
value_type d = 2.2;
std::complex<value_type> c(1.1, -1.7);

// Using convenience function, call by value
auto p1 = std::make_pair(d, c);
std::cout << p1.first << "," << p1.second << '\n';
d = 0.0;
std::cout << p1.first << "," << p1.second << '\n';
```

Instead of copying values into a pair it is possible to use references to avoid creating expensive copies:

```
auto p2 = std::make_pair(std::ref(d), std::ref(c));
std::cout << p2.first << ","
          << p2.second << '\n'; // 0, (1.1, -1.7)
c = std::complex<value_type>  (100.0, -200.0);
std::cout << "Reference: " << p2.first
          << "," << p2.second << '\n';      // (0, (100.0, -200.0)
```

Implicit conversions can occur when creating pairs with types other than those of its elements. In these cases we can experience truncation or promotion effects:

```
// Implicit conversions
std::pair<char, int> p3(p2);
std::cout << p3.first << "," << p3.second << '\n';

std::pair<value_type, std::complex<value_type>> p4(p2);
std::cout << p4.first << "," << p4.second << '\n';
```

We now discuss how to read and modify the elements in a pair. We can use an indexing function with indexes 0 and 1 (a pair has two elements) or we can access the elements based on their type:

```
std::cout << '(' << std::get<0>(p1) << ", " << std::get<1>(p1) << ")\n";
std::cout << '(' << std::get<value_type>(p1) << ", "
          << std::get<std::complex<value_type>>(p1) << ")\n";
```

The elements of a pair can be modified and the contents of pairs can be swapped, as the following examples show:

```
int n = 1;
std::array<int, 4> arr{ 1,2,3,4 };
std::array<int, 4> arr2{ -1,-2,-3,-4 };
auto p3 = std::make_pair(n, arr);
auto p4 = std::make_pair(-n, arr2);
std::swap(p3, p4);
std::cout << p3.first << "," << p3.second[1] << '\n';

// 4. Modify individual elements of pair
std::get<0>(p3) = 200;
std::get<1>(p3) = std::array<int,4>(arr);
std::cout << p3.first << "," << p3.second[1] << '\n';
```

Having created a pair we might wish to compare the values of its elements with those of another pair. To this end, the usual binary predicates are applicable. We focus on the *inequality operator* as an exemplar:

```
using IntPair = std::pair<int,int>;

// Comparison operators < etc. In general, 1st value
// has higher value
IntPair pi1 = std::make_pair(1, 2);
IntPair pi2 = std::make_pair(1, 1);
IntPair pi3 = std::make_pair(2, 1);
IntPair pi4 = std::make_pair(2, 2);
std::cout << std::boolalpha << (pi1 < pi2) << '\n';    // false
std::cout << std::boolalpha << (pi2 < pi1) << '\n';    // true
std::cout << std::boolalpha << (pi1 < pi3) << '\n';    // true
std::cout << std::boolalpha << (pi3 < pi1) << '\n';    // false
std::cout << std::boolalpha << (pi3 < pi4) << '\n';    // true
std::cout << std::boolalpha << (pi4 < pi3) << '\n';    // false
std::cout << std::boolalpha << (pi4 == pi3) << '\n';   // false
std::cout << std::boolalpha << (pi4 != pi3) << '\n';  // true
```

The fact that binary operators are defined for pairs implies that they can be sorted and used with STL algorithms.

We now come to the topic of constructing a pair consisting of user-defined types and we switch the discussion from `std::pair` to `std::tuple`. We can force calling a constructor rather than using a tuple as a whole. We can also *forward* arguments to construct a tuple. To this end, let us first consider a class whose instances will be elements of a pair:

```
struct C
{
   C(const std::tuple<int, double>& tup)
   {
      std::cout << "\tConstructed a C from a tuple\n";
   }
```

```
    C(int, double)
    {
       std::cout << "\tConstructed a C from an int and a double\n";
    }
};
```

We first create a tuple as follows:

```
// Create a pair using 'whole' tuples
std::tuple<int, double> t1(1, 3.14), t2(-3.0, 2.71);
std::pair<C, C> p1(t1, t2);
```

The output in this case is:

```
Constructed a C from a tuple
Constructed a C from a tuple
```

We now wish to side-track the explicit creation of tuples and instead we *forward* the tuples `t1` and `t2` to the elements of class `C`:

```
// Now force the ctor to take the elements of the tuple
// rather than the tuple as a whole.
std::pair<C, C> p2(std::piecewise_construct, t1, t2);
```

The output in this case is:

```
Constructed a C from an int and a double
Constructed a C from an int and a double
```

We can *forward* the arguments to the pair's constructor:

```
std::cout << "\nForward of C: \n";
std::pair<C, C> p2A(std::piecewise_construct,
                    std::forward_as_tuple(1, -1.7),
                    std::forward_as_tuple(2, 110.3));
std::cout << "End, forward of C:\n";

// Forwarding the arguments
int charCount = 10; char c = 'A';
std::pair<std::complex<value_type>, std::string>
pZ(std::piecewise_construct,
       std::forward_as_tuple(1.0, -1.7),
       std::forward_as_tuple(charCount, c));
std::cout << "\nForward: " << std::get<0>(pZ) << ", " <<
std::get<1>(pZ) << ")\n";
```

The output in this case is:

```
Forward of C:
   Constructed a C from an int and a double
   Constructed a C from an int and a double
End, forward of C:

Forward: (1,-1.7), AAAAAAAAAA)
```

In general, forwarding is more efficient.

We discuss the *type traits* library in Chapter 6 and we can use two relevant functions from that library to examine the types of the elements comprising a tuple, for example in the case of a P1 consisting of a `value-type` and a complex number:

```
// 7. Compile-time access to pair's element types.
// Use type traits to check types corresponding to element types
// (chapters 4 and 6)
using TypeFirst = std::tuple_element<0, decltype(p1)>::type;
static_assert(std::is_same<TypeFirst, value_type>::value, "ouch");

using TypeSecond = std::tuple_element<1, decltype(p1)>::type;
static_assert(std::is_same<TypeSecond, std::complex<value_type>>
::value, "ouch");
```

We now introduce `std::tie`. In general, this function is a way to associate variables with elements of a pair or of a tuple. We take the example in which the input arguments are copied into a pair:

```
using IntPair = std::pair<int,int>;

// a. Values
int n = 1; int m = 2;
IntPair pA = std::make_pair(n, m);
std::tie(n, m) = pA;

n = 10; m = 40;
std::cout << pA.first << "," << pA.second << '\n';      // 1,2
```

We can create a tuple in which the input arguments are *references*:

```
// b. References
IntPair pB = std::make_pair(std::ref(n), std::ref(m));
std::cout << pB.first << "," << pB.second << '\n'; // 10, 40
n = -100; m = -1'000'000;
std::cout << pB.first << "," << pB.second << '\n'; // 10, 40 !
```

We note that changes to the values of the variables have no effect on the tuple's element values, nor vice versa. If we wish to have this feature we could use *shared pointers*:

```
template <typename T>
       using SP = std::shared_ptr<T>;
       using SharedIntPair = std::pair<SP<int>, SP<int>>;
```

```
// c. Shared pointers
SP<int> a = SP<int>(new int); *a = 2;
SP<int> b = SP<int>(new int); *b = 3;

SharedIntPair pC = std::make_pair(a,b);
std::cout << *pC.first << "," << *pC.second << '\n'; // 2,3

*a = 3; *b = 2;
std::cout << *pC.first << "," << *pC.second << '\n'; // 3,2
```

The syntax `SP<T>` means `std::shared_ptr<T>`. It can be seen as a handy shorthand notation.

Finally, we discuss how to create *nested pairs*; in this case we create a pair consisting of two other pairs:

```
// 9. Pairs and nesting.
using A = std::pair<int, int>;
using B = std::pair<std::array<int,4>, std::array<int, 6>>;
using C = std::pair<A, B>;

std::array<int, 4> a1 = { 1,2,3,4 };
std::array<int, 6> a2 = { 1,2,-3,4,5,6 };
C pY(std::piecewise_construct,
     std::forward_as_tuple(1, 2), std::forward_as_tuple(a1,a2));
std::cout << "\nNested (1, -3): " << (std::get<0>(pY)).first
           << ", " << (std::get<1>(pY)).second[2] << "\n";
```

We shall extend this idea in later sections when we discuss how to create tuples and nested tuples and apply them as a means to group logically related data.

5.3 MATHEMATICAL AND COMPUTER SCIENCE BACKGROUND

We now give a mathematical introduction to the concept of a tuple (or *n-tuple*) which is an ordered set of n elements or components of heterogeneous type where n is a non-negative integer. There are several ways to define tuples and access their elements. In C++, tuples are generalisations of the pair struct (a pair is a 2-tuple) as the following example shows:

```
#include <tuple>
using Tuple = std::tuple<int, std::string>;

Tuple tup1(3, std::string("three"));
Tuple tup2 = std::make_tuple(4, std::string("four"));

// Accessing the tuple elements in two different ways
std::cout << std::get<0>(tup1) << "," << std::get<1>(tup1) << '\n';
// 3, "three"
std::cout << std::get<int>(tup1) << ","
           << std::get<std::string>(tup1) << '\n'; // 3, "three"
```

We see that elements can be accessed by integer indices or by element type.
Mathematically it is possible to define tuples in various ways, for example:

- As functions.
- As nested ordered pairs.
- Using tuple-oriented relational calculus.

A tuple can be visualised as a function F whose *domain* is an implicit set X of element indices and whose *range* (*codomain*) Y is the tuple's set of elements:

$$\left(a_1, a_2, \ldots, a_n\right) \equiv (X, Y, F)$$

where:

$$\begin{aligned} X &= \{1, 2, \ldots, n\} \\ Y &= \left\{a_1, a_2, \ldots, a_n\right\} \\ F &= \left\{(1, a_1), (2, a_2), \ldots, (n, a_n)\right\}. \end{aligned}$$

Alternatively, we can write the representation in the more readable form:

$$\left(a_1, \ldots, a_n\right) := (F(1), F(2), \ldots, F(n)).$$

Here it becomes clear how F is a mapping from the set of positive integers to a collection of elements.

We can also define tuples as nested ordered pairs. Our initial assumption is that we already know what an ordered pair is. We first define the *0-tuple* (*empty tuple*) consisting of zero elements. This is the empty set denoted by $\emptyset$. More generally, an *n-tuple* can be defined as an ordered pair consisting of its first element and the remaining $(n - 1)$ elements:

$$\left(a_1, a_2, a_3, \ldots, a_n\right) = \left(a_1, \left(a_2, a_3, \ldots, a_n\right)\right).$$

We apply this definition recursively:

$$\left(a_1, a_2, a_3, \ldots, a_n\right) = (a_1, (a_2, (a_3, (\ldots, (a_n, \emptyset) \ldots)))).$$

An example is:

$$\begin{aligned} (1, 2, 3) &= (1, (2, (3, \emptyset))) \\ (1, 2, 3, 4) &= (1, (2, (3, (4, \emptyset)))). \end{aligned}$$

A variant of this approach is to *peel* the elements of a tuple from its end:

$$\left(a_1, a_2, a_3, \ldots, a_n\right) = ((a_1, a_2, a_3, \ldots, a_{n-1}), a_n).$$

An example is:

$$\begin{aligned} (1, 2, 3) &= (((\emptyset, 1), 2), 3) \\ (1, 2, 3, 4) &= ((((\emptyset, 1), 2), 3), 4). \end{aligned}$$

It is possible to reformulate this second representation in terms of set theory but a discussion is outside the current scope. Finally, we discuss the *relational model* (Date, 1981). It uses a tuple definition that is similar to the function definition above but now each element is identified by a distinct name (rather than by a numerical index) called an *attribute*.

An example of a database table with five attributes is:

```
(name: "widget", ID: 324, city: London, delivery: 2016-12-12)
```

One of the advantages of this approach is that *attribute–value* pairs may appear in any order. A tuple is usually implemented as a row in a database table.

5.4 TUPLE FUNDAMENTALS AND SIMPLE EXAMPLES

We have discussed pairs in detail. Tuples are generalisations of pairs in the sense that the number of elements in the latter type is greater than or equal to two. We have done the hard work in Section 5.2 and much of the code also applies to tuples. We give an initial example that is similar to the syntax in Section 5.2. We present the code and it should be easy to understand as it is similar to the corresponding code for `std::pair<>` :

```
using TupleType = std::tuple<std::string, int>;

// Creating tuples
TupleType myTuple(std::string("A"), 1);
TupleType myTuple2 = std::make_tuple("B", 0);
TupleType myTuple3(myTuple);
auto myTuple4 = myTuple2;

// Accessing the elements of a tuple
std::cout << std::get<0>(myTuple) << std::get<1>(myTuple) << '\n';  // A,1
std::get<0>(myTuple) = std::string("C");
std::get<1>(myTuple) = 3;
std::cout << std::get<0>(myTuple) << std::get<1>(myTuple) << '\n';  // C,3
```

We thus see that the elements of a tuple can be modified. Extending the code to tuples with more than two elements is straightforward.

5.5 ADVANCED TUPLES

Having discussed the main language features of `std::tuple` that C++11 offers we now describe how to create tuples that can contain other tuples as well as tuples with a variable number of elements (called *variadic tuples*).

5.5.1 Tuple Nesting

A useful feature in C++ is that we can define fixed-size recursive and composite structures. In other words, we can define tuples that contain other tuples as well as other types of elements. This feature can be applied to creating *configuration data* in an application. This data can be

created by dedicated factory objects and the data can then be processed in an application at a suitable *entry point* in the code.

Some typical applications are:

- Modelling data for rainbow and multi-asset options by a tuple with three elements. The first and second elements are themselves tuples containing the data (such as strike and expiration) pertaining to the first and second assets respectively, while the third element contains the value of the correlation between the assets.
- Mathematical properties of differential equations encapsulated in reusable tuples.
- Essential parameters pertaining to numerical processes, for example step size, tolerances and maximum number of iterations.
- Any non-trivial application that needs to be configured from various sources. To this end, we use tuples that client code reads without having to know where the data comes from.

We give a simple example of how to create nested tuples that model people and their whereabouts (it can be seen as a simple database):

```
// Nested tuples
// First and last names
typedef std::tuple<std::string, std::string> Name;

// Home address and city and code
typedef std::tuple<std::string, std::string, long> HomeAddress;
typedef std::tuple<std::string, std::string> Location;

// Name + Address + Location
typedef std::tuple<Name, HomeAddress, Location> NHL;

Name pName;
std::get<0>(pName) = std::string("Frankie");
std::get<1>(pName) = std::string("Carbone");

HomeAddress ha;
std::get<0>(ha) = std::string("Sunnyside");
std::get<1>(ha) = 199;

Location loc;
std::get<0>(ha) = std::string("Queens");

std::string code("AK");
std::get<1>(ha) = code;

// Create the top-level (Whole)tuple
NHL nhl;
std::get<0>(nhl) = pName;
std::get<1>(nhl) = ha;
std::get<2>(nhl) = loc;
```

We shall discuss composite and nested tuples in later chapters.

5.5.2 Variadic Tuples

In general, a *variadic template* is one that takes a variable number of input arguments. In particular, tuples can be defined as having a variable number of elements:

```
template<class... Types>
    class tuple;
```

This declaration signifies that this class can take any number of typenames (zero or more) as its template parameters. Incidentally, the number of arguments can be zero. If we wish to disable the definition of a template with zero arguments, we can modify the definition above as follows (in which case the tuple must have at least one element):

```
template<typename First, typename... Rest>
    class tuple;
```

Variadic templates also apply to functions, for example free functions:

```
template<class... Args>
    void print(const std::tuple<Args...>& t)
{

}
```

The advantages of variadic tuples are:

- They can take an arbitrary number of arguments of any type.
- They reduce code explosion: you (*supplier*) can define code once that uses variadic templates and *clients* can customise the code by choosing the types and number of arguments that they wish to use.
- It is possible to design and implement generic code for some cases in which traditional object-oriented technology (using subtype polymorphism, for example) would lead to code that is difficult to understand or to maintain.
- Variadic templates are a compile-time feature which should lead to efficient and robust code.

We take an initial example of using variadic templates. In particular, we write a function to print a tuple with any number of elements. The code is based on the discussion in Section 5.3 in which we peel the elements of a tuple from its end, print the last value and then call the same function again on a tuple with one element less than the original one. The code is:

```
// General case; recursive call from <N> to <N-1>
template<class Tuple, std::size_t N>
     struct TuplePrinter
{
      static void print(const Tuple& t)
      {
```

```
        TuplePrinter<Tuple, N - 1>::print(t);
        std::cout << ", " << std::get<N - 1>(t);
    }
};
```

Since the function is recursive we know that we need to define its very last action (this is the equivalent of *tail recursion*). To this end, we employ template specialisation:

```
// Tail recursion case when N = 1 (stopping criteria)
template<class Tuple>
   struct TuplePrinter<Tuple, 1>

{
    static void print(const Tuple& t)
    {
        // Assume << is overloaded
        std::cout << std::get<0>(t);
    }
};
```

We can now print the contents of the tuple example from Section 5.5.1 (using the same variable name `pName`):

```
auto nhl2 = std::tuple_cat(pName, ha, loc);
Print(nhl2);

// Concatenate the sub-tuples int one flat tuple and print it
const std::size_t N = std::tuple_size <decltype(nhl2)>::value;
std::cout << "Number of elements in tuple: " << N << '\n'; // 7
TuplePrinter<decltype(nhl2),N>::print(nhl2);
```

You can run the code and examine the output.

5.6 USING TUPLES IN CODE

We are interested in investigating the consequences of using tuples as return types of, and as input arguments to, functions.

5.6.1 Function Return Types

An interesting application of tuples is to use them as return types of functions. Typical use cases are:

- UC1: A function that returns a value of interest and some *condition code* or flag relating to that return value, for example a flag giving SUCCESS or FAILURE (see a discussion of the COM HRESULT 32-bit code in Rogerson, 1997). This approach could be an alternative to exception-handling mechanisms.

- UC2: Merging the results of several computationally intensive algorithms (each of which calculates a given property of a container with arithmetic elements) into a single algorithm that computes all the properties of the container in one 'sweep' as it were.
- UC3: Using tuples to hold elements representing objects that are needed when assembling and configuring an application using *creational design patterns* (GOF, 1995). We shall discuss this important *next-generation design pattern* trajectory in later chapters, in particular Chapters 9, 19, 23, 31 and 32.

We take an example of use case UC2 where we wish to compute the sum and product of the elements of a vector using a single loop:

```
template <typename V>
   std::tuple<V,V> SumAndProduct(const std::vector<V>& x)
{
   V sum = x[0];
   V prod = x[0];


   for (std::size_t j = 1; j < x.size(); ++j)
   { // Only one loop iteration needed.

     sum += x[j];
     prod *= x[j];
   }

   return std::tuple<V,V>(sum, prod);
}
```

An example of use is:

```
std::vector<double> v1(4); // 1, 2, 3, 4
for (std::size_t k = 0; k < v1.size(); ++k)
{
     v1[k] = double(k) + 1.0;
}

std::tuple<double, double> t1 = SumAndProduct(v1);
std::cout << "Sum and product (10, 24): " << std::get<0>(t1)
           << ", " << std::get<1>(t1) << std::endl;
```

Let us now assume that we wish to associate these computed values with variables. Then we can use `std::tie` as we have already seen in Section 5.2:

```
 // We tie tuple element to variables
double sum; double product;
std::tie(sum, product) = SumAndProduct(v1);
std::cout << "Sum: " << sum << ",Product: " << product;
```

```
// Ignoring some tuple elements
double sum2;
std::tie(sum2, std::ignore) = SumAndProduct(v1);
std::cout << "Sum and ignoring product: " << sum2 << std::endl;
```

In the second example we used `std::ignore` if we are not interested in certain variables. This may be a useful trick when working with tuples having elements that you are not interested in, for example.

We now give an example of use case UC3. In particular, we give an example of the *Abstract Factory* design pattern: the essence of this pattern is to 'provide an interface for creating families of related or dependent objects without specifying their concrete classes'. The objective is to select compatible objects and related products from several class hierarchies and present them to clients. The basic class hierarchies are:

```
// Creational patterns (Abstract Factory)
struct B1 {}; struct B2 {}; struct B3 {};
struct D1 : public B1 { D1() { std::cout << "D1 born\n"; } };
struct D2 : public B2 { D2() { std::cout << "D2 born\n"; } };
struct D3 : public B3 { D3() { std::cout << "D3 born\n"; } };
```

The traditional pattern has the abstract interface below that we extend by providing a member function to return the individual derived class instances in a tuple:

```
template <typename T>
   using SP = std::shared_ptr <T>

class AbstractFactory
{ // Classic GOF pattern meeting std::tuple

public:
   AbstractFactory() {}

   // Derived classes implement these functions
   virtual SP<B1> Part1() = 0;
   virtual SP<B2> Part2() = 0;

   virtual std::tuple < SP<B1>, SP<B2>> Product()
   { // An example of the Template Method Pattern.

     return std::make_tuple(Part1(), Part2());
   }
};
```

Derived classes of `AbstractFactory` must implement the above two pure virtual member functions. An alternative to this factory class is to create a factory class in which there is only one virtual function that returns a tuple to be overridden:

```
class Builder
{
public:
```

```
        Builder() {}
        virtual std::tuple < SP<B1>, SP<B2>> Product() = 0;
};

// Specific builder
class RealBuilder : public Builder
{
public:
   RealBuilder() {}
   std::tuple <SP<B1>, SP<B2>> Product()
   { // Create a product family consisting of C1 and C2

      SP<B1> c1 = SP<B1>(new D1);
      SP<B2> c2 = SP<B2>(new D2);

      return std::make_tuple(c1, c2);
   }
};
```

An example of use is:

```
// Builder example
RealBuilder builder;
auto prod = builder.Product();
auto element1 = std::get<0>(prod); // A D1 instance
auto element2 = std::get<1>(prod); // A D2 instance

// Dynamic cast
static_assert(std::is_same<decltype(element1), SP<B1>>::value, "ouch");
static_assert(std::is_same<decltype(element2), SP<B2>>::value, "ouch");
```

More extended examples of creational patterns will be discussed in later chapters in the context of Monte Carlo option pricing applications and PDE models, for example. In particular, using tuples leads to code that is easier to understand and to maintain than code based on the more traditional approach in GOF (1995).

5.6.2 Function Input Arguments

In many cases we are accustomed to dealing with tuples whose elements are a combination of built-in and user-defined data types. The ability to group related data into a single logical entity is seen as an advantage. Furthermore, tuple elements can also contain universal function pointers and function objects!

When is it useful to 'combine' tuples and functions and why would we need such functionality? Some applications could be:

- Many numerical processes and methods use related functions; for example, the Newton–Raphson method which computes the root of a scalar nonlinear function requires that we give the function and its derivative.

- Functions to transform a tuple in a given frame of reference into a tuple in another frame of reference. A typical category is when approximating the solution of ordinary, stochastic and partial differential equations in a continuous domain by numerical schemes defined in a discrete domain.
- Using tuples in combination with *creational design patterns* (GOF, 1995). When configuring an application we create the objects that are needed and we package them in a tuple that clients use. This is a *next-generation Builder* pattern because it uses C++11 language features that have not yet found their way into the object-oriented patterns in GOF (1995).

We shall see examples of these topics as we progress in the book. We conclude this section with some examples. First, we can define relevant data structures as follows:

```
// Map a (single) type to a type
template <typename T>
    using FunctionType = std::function<T (const T& t)>;

// Tuple whose elements are functions
template <typename T>
     using TupleFunctionType
                = std::tuple<FunctionType<T>,
                                FunctionType<T>>;
```

We can use these data types to define tuples consisting of functions:

```
// Second approach, need f(x) and df/dx for many cases.
auto f = [](double x) { return std::exp(x) - 148.413; };
auto fd = [](double x) { return std::exp(x); };

TupleFunctionType<double> fun = std::make_tuple(f, fd);
auto funVal = std::make_tuple(f(2.0),fd(3.0));
std::cout << "Tuple values: " << std::get<0>(funVal) << ", "
          << std::get<1>(funVal) << '\n';
```

The final example in this section is to discuss the use of the Newton–Raphson method as presented in the *Boost Math Toolkit*. To this end, the algorithm needs two functions that we implement as a function object returning a tuple consisting of two computed values:

```
// Newton-Raphson, exp(x) = 148.4131591, x = 5
template <typename T> class NRFunctions
{ // Functions needed for input to NR method
private:

public:
    NRFunctions()
    {
    }

    std::tuple<T, T> operator()(const T& x)
    { // x is estimate so far.
```

```
        // Return the function and its dervivative.
        return std::make_tuple(std::exp(x) - 148.4131591, std::exp(x));
    }
};
```

We can now use this function object as one of the input parameters to the Newton–Raphson algorithm in Boost:

```
template <typename T>
   std::tuple<T, boost::uintmax_t> NewtonRaphson (const T& L, const T& R)
{
   using namespace std; // for frexp, ldexp, numeric_limits.
   using namespace boost::math::tools;

   T min = L;;
   T max = R;
   T guess = (L + R)/ 2; // Rough guess
   int digits = std::numeric_limits<T>::digits;
   boost::uintmax_t maxIter = 10;
   T val = boost::math::tools::newton_raphson_iterate
            (NRFunctions<T>(), guess,min, max, digits, maxIter);

   return std::make_tuple(val, maxIter);
}
```

Finally, an example of use is:

```
double leftBracket = 0.0; double rightBracket = 10.0;
auto result = NewtonRaphson(leftBracket, rightBracket);
std::cout << "NR results: " <<  std::setprecision(8)
          << std::get<0>(result) << ", " << std::get<1>(result);
```

Typically, the value 5.0 is the answer and the algorithm converges after two iterations. We discuss univariate nonlinear solvers in more detail in Chapter 19.

5.7 OTHER RELATED LIBRARIES

We discuss two libraries that offer similar functionality to `std::tuple` as well as functionality that is not offered by the latter.

5.7.1 Boost Tuple

Before C++11 became available we used *Boost Tuple* (see Demming and Duffy, 2010). We ported examples written in this library to C++11 using `std::tuple`. For this reason we do not need the library anymore, preferring `std::tuple` indeed or *Boost Fusion*.

5.7.2 Boost Fusion

We thought that it might be useful to introduce the *Fusion* library which allows developers to work with heterogeneous collections of data that we know at this stage to be tuples. It offers more functionality than `std::tuple`:

- A set of containers (vector, list, set and map).
- Views that provide a transformed presentation of the underlying data.
- Algorithms that operate on various sequence types (a *sequence* is an entity comprising a container and a view). Basically, these are tuple manipulation and traversal routines.

The architecture of *Fusion* is based on MPL (which focuses on type manipulation only and which in turn is based on STL) and it received its name because it uses a combination of compile-time metaprogramming and run-time programming techniques. One of the reasons for developing *Fusion* was to be able to support heterogeneous containers in C++ similar to the functionality that languages such as Python and Haskell provide.

Let us take an example. We first create a fusion tuple:

```
#include <boost/fusion/sequence.hpp>
#include <boost/fusion/include/sequence.hpp>
#include <boost/fusion/include/algorithm.hpp>

namespace fusion = boost::fusion;

// Create a fusion tuple
fusion::vector<int, char, std::string, C> t(1, 'x',
std::string("101"), C());

// Examine its elements
int i = fusion::at_c<0>(t);
char ch = fusion::at_c<1>(t);
std::string s = fusion::at_c<2>(t);
C c = fusion::at_c<3>(t);
```

where `C` is a user-defined type:

```
class C
{
public:
   C() {}
};
```

This syntax is not so shocking but the added advantage is that we can iterate over the elements of the tuple and apply a print function to each one (this is the magic of *Fusion*):

```
// Now call a fusion generic algorithm to iterate over
// the sequence and call a user-defined function.
// std::tuple does not have this.
fusion::for_each(t, Print());
```

The user-defined function is:

```
struct Print
{
template <typename T>
   void operator()(T const& x) const
   {
      // Platform-specific, but that's not the point
      std::cout << "Name: " << typeid(x).name() << '\n';
   }
};
```

This functionality can be seen as a form of polymorphic behaviour. A detailed discussion on *Fusion* is outside the scope of this book. We refer the reader to the Boost online documentation.

5.8 TUPLES AND RUN-TIME EFFICIENCY

We give a discussion on how to avoid possible run-time performance bottlenecks when using `std::tuple` in the wrong way. We recall that tuples group logically related data into one entity. We need to consider the main use cases:

- Packing values into a tuple (using a constructor or `std::make_tuple`).
- Unpacking/extracting the elements from a tuple.

In general, tuples are used to hold data. The data can be large grained in many cases, although this is not a requirement. It is possible to use tuples to hold data that can be modified at run-time but we should ask ourselves if it is not more efficient to use a `std::vector<>`, `std::array<>` or even a struct. The performance impact of a single call to a function that has a tuple as return type and/or as input argument may not be noticeable, but it may be noticeable if it is called many times during program execution.

We encountered an example of performance degradation when stress-testing Monte Carlo option pricing code. We created a million paths and each path used 500 time steps when approximating the associated stochastic differential equation (SDE) with the Euler method. We used the Box–Muller method which returns two standard normal variates that we encapsulated in a tuple. The Monte Carlo path evolver subsequently unpacked the tuple in order to access these variates 500 million times!

The code for Box–Muller is:

```
std::tuple<double,double> BoxMuller::GenerateRn()
{
   double BoxMuller::GenerateRn() {
   // Generate independent uniform random variates
   double u1 = randu(0, 1);
   double u2 = randu(0, 1);
   // Generate two standard normal variates
```

```
    double norm1 =
    std::sqrt(-2.0*std::log(u1))*std::cos(2.0*Pi*u2); double norm2 =
    std::sqrt(-2.0*std::log(u1))*std::sin(2.0*Pi*u2);

    std::tuple<double, double> result(norm1, norm2);
    return result;
}
```

We see that a temporary tuple instance is created and a copy is created and returned to the caller. This code does not satisfy the *Single Responsibility Principle* (SRP) (that we discuss in Chapter 7) because it implements an algorithm and creates a temporary object. A more efficient solution is to use the *Mersenne Twister* algorithm and to return a single automatic variable:

```
// Abstract Base(Interface) RNG class
class IRNG
{
protected:
public:
    //Default Constructor
    IRNG() {}
    virtual double GenerateRng() = 0;
};

class MTNormalRNG : public IRNG
{ // Sample code

private:
    double m_mean;          // Mean of Normal Distribution
    double m_variance;      // Variance of Normal Distribution
    std::mt19937 mt;        // random engine to be used
    std::normal_distribution<double> normal;
public:
//Constructor
   MTNormalRNG(double v1, double v2) : IRNG(), m_mean(v1),
   m_variance(v2), mt(std::mt19937()),
   normal(std::normal_distribution<double>()) {}

// Random Number Generation function
   virtual double GenerateRng() override
   {

      return normal(mt);   // Return random number
   }
};
```

Evidence showed that the Monte Carlo pricer using tuples was 50 times slower than the Monte Carlo pricer that did not use tuples in an experiment that we conducted.

5.9 ADVANTAGES AND APPLICATIONS OF TUPLES

We find it surprising that tuples were not supported before C++11. They represent a key concept in computer science in a range of programming languages, database theory and parallel/distributed computing (see, for example, Halter, 2002). We are primarily interested in using the functionality offered by `std::tuple` to help us create compact and manageable entities that are used as configuration data in applications.

Some advantages are:

- A1: An alternative to, and an improvement on, other approaches such as arrays of pointers to data (or even `void*`). The resulting code is usually more robust and efficient than previous solutions based on subtype polymorphism, for example.
- A2: Supplier code can use variadic tuples that can be customised by clients. This reduces the amount of code to be created in general. Roughly speaking, we create supplier code once and let the compiler generate code for us.
- A3: Code using tuples tends to be readable and maintainable. This has major consequences for the reliability of software because our human ability to process information decreases dramatically beyond a critical size of the input information (see Miller, 1956). In particular, the number of pieces that human memory can hold at any one moment in time is seven, plus or minus two. A solution to this problem is to use *data chunking*. *Chunking* in psychology is a process by which individual pieces of information are bound together into a meaningful whole. A chunk is defined as a familiar collection of more elementary units that have been inter-associated and stored in memory repeatedly and act as a coherent, integrated group when retrieved. For us, tuples are chunks in the current context.
- A4: Standardisation: a useful project is to investigate how to group configuration data into logical entities that we implement as tuples and to investigate in how many applications these tuples can be (re)used. A typical example is the tuple that encapsulates the data corresponding to a one-factor option. This tuple can be used in lattice, PDE and Monte Carlo applications, for example.

There are many applications of tuples. We can categorise these applications as discussed in the *Layers* pattern (see POSA, 1996 and Chapter 11 of the current book):

- Layer 0: `std::tuple`. Its members have already been discussed in this book.
- Layer 1: User-defined ADTs based on `std::tuple` as building blocks.
- Layer 2: *Mechanisms* (functions) to create user-defined tuples in some way or transform a tuple to another data structure, possibly also a tuple. For example, we can create a tuple consisting of a date and a collection of commodity prices based on input from a text file:

```
Date,Open,High,Low,Close,Volume,Adj Close

2013-02-01,54.87,55.20,54.67,54.92,2347600,54.92

2013-01-31,54.74,54.97,53.99,54.29,3343300,54.29

2013-01-30,54.84,55.35,54.68,54.68,2472800,54.68
```

- Layer 3: Using tuples as components in classes and complex objects. In particular, this layer could make extensive use of variadic tuples in conjunction with universal function wrappers.
- Layer 4: System-level code that uses tuples, for example using the next-generation *Builder* pattern to configure an application based on domain architectures as we shall discuss in Chapters 9, 19 and 31.

We expand on these topics as we progress in the book.

5.10 SUMMARY AND CONCLUSIONS

In this chapter we gave an introduction to the class template `std::tuple` and its application to software development in C++. The approach was hands-on using compact and extended code examples.

The major topics presented were:

- A review of `std::pair` (including its new features) and much of its functionality is also supported by `std::tuple`.
- A mathematical introduction to tuples.
- Recursive aggregated tuples; variadic tuples.
- Tuples and their relationship with functions.
- Suitability of tuples for applications. The advantages of using tuples.

Finally, we recommend that you do the exercises in this chapter because they contain designs and features that will be helpful in larger applications in later chapters.

5.11 EXERCISES AND PROJECTS

1. What are the top two advantages of using tuples?
   a) As return types of functions.
   b) As input arguments to functions.
   c) To align data on word boundaries.
   d) To avoid copying data.

2. In which of the following situations would we use tuples instead of structs or classes?
   a) When we wish to create lists of logically related data in client code.
   b) When we are not concerned with nested data structures.
   c) When the tuple data is just a repository.
   d) A tuple is similar to a class but with no member functions.

3. Which of the following functionalities can be used to construct tuples?
   a) `std::make_tuple`.
   b) Initializer lists.
   c) Tuple concatenation.
   d) Using `std::tie` to create a tuple of *lvalue* references.

4. Which of the following features are supported by C++ tuples?
   a) Associating C++ variables with a tuple's elements.
   b) The ability to forward elements of a tuple to a function taking tuple arguments.

c) Concatenating tuples.
d) Tuples, some of whose elements can be universal function wrappers.

5. What are the main differences between tuples and sets in C++?
a) Sets and tuples may have elements of heterogeneous types.
b) A tuple may have multiple instances of the same element.
c) A tuple has a finite set of elements.
d) Tuple elements are ordered.

6. (Reviewing the Code, Small Project)

The goal of this exercise is to understand the code examples in Section 5.2 and to apply them in your own programs.

Answer the following questions:

a) Since a `std::pair` is a `std::tuple` with just two elements then the code in Section 5.2 should be easy to compile using tuples (instead of `std::make_pair` you should use `std::make_tuple`). Run and test the modified code.
b) Create two instances of `std::tuple` consisting of element type `int` and `std::string`. Compare them using all the standard Boolean binary operators.
c) Create a simple class that accepts a pair in its constructor. The class has two-member data initialised from the pair's elements. In how many ways can this requirement be realised? Test the code and review the efficiency and understandability of the solution.
d) Determine the output of the following code snippet:

```
// Sort an array of pairs
std::vector<std::pair<int, int>> v = {{ 2, 100 },{ 2, 90 },{ 1, 1000}};
std::sort(v.begin(), v.end());

for (auto p : v)
{
   std::cout << "(" << p.first << "," << p.second << ")\n";
}
```

e) Is it possible to create a tuple with two elements from a pair or vice versa? Is it possible to create a tuple with three elements from a pair or vice versa?
f) Investigate how to concatenate tuples, including concatenation of nested tuples.

7. (Generalising the Code for Newton–Raphson)
a) In Section 5.6.2 we applied the Newton–Raphson method to find the solution of $e^x = 148.4131591$. Generalise the code so that it works with any function of the correct signature:

```
// Function that maps a type to a tuple
template <typename T> using TupleFunctionTypeII
         = std::function<std::tuple<T,T> (const T& t)>;
```

b) Test the new code with several nonlinear functions.

8. (Performance)

Consider the following code to generate arrays containing uniform random values:

```
std::vector<double> compute()
{   // Re-seed uniform generator when requesting a new path

    uniformGenerator.seed(seeder());
```

```
        std::vector<double> v(nSteps);
        std::transform(v.begin(), v.end(), v.begin(), normalGenerator);

        return v;
    }
```

Answer the following questions (assuming that `nSteps`, `uniformGenerator` and `normalGenerator` have been defined):

a) Determine how many temporary vectors are created when this function is called from client code. Measure the run-time performance when this function is called one million times.

b) Now consider the use case in which a *single* vector is created and modified one million times. How would you modify the code of `compute()` and improve the run-time performance?

c) Generalise the results in parts a) and b) to situations in which tuples are used instead of vectors. Consider the code in this chapter to determine the presence of potential performance bottlenecks in the style of discussion in Section 5.8.

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 6

Type Traits, Advanced Lambdas and Multiparadigm Design in C++

6.1 INTRODUCTION AND OBJECTIVES

In this chapter we introduce a number of advanced concepts and language features in C++. First, we discuss the C++ *type traits* library. For example, we can provide answers to the following kinds of questions:

- Is a type a floating-point type?
- Is a type an arithmetic type?
- Does a type have a defined constructor?
- Are two types the same?
- Remove const and volatile specifiers from a given type.
- Remove a reference/pointer from a given type.
- Make an integral type signed or unsigned.
- And many more.

These advanced topics will be useful for library builders. Application developers can browse the topics here although they should become familiar with Sections 6.5, 6.6.1 and 6.8 on a first reading.

A simple example is to determine whether a type is arithmetic, in other words whether it supports the usual arithmetic operators for addition, subtraction, multiplication and division. For example, we may need to exclude non-arithmetic types in a generic matrix algebra library. A typical code snippet is:

```
#include <iostream>
#include <complex>
#include <string>
#include <ratio>
#include <type-traits>
```

```
// #include <boost/type_traits.hpp>        // Not needed anymore,
                                           // now in C++

template <typename T> void IsArithmetic()
{ // First example of type_traits to check if T supports +,-,*,/

    if (std::is_arithmetic<T>::value)
    {
       std::cout << "This is an arithmetic type argument\n";
    }
    else
    {
       std::cout << "This is _not_ an arithmetic type argument\n";
    }
}

int main()
{

    IsArithmetic<int>();                          // true
    IsArithmetic<long double>();                  // true
    IsArithmetic<std::complex<double>>();         // false
    IsArithmetic<std::string>();                  // false

    // Compile-time fractional arithmetic
    const intmax_t numerator = 1;
    const intmax_t denominator = 2;
    using Half = std::ratio<numerator, denominator>;

    IsArithmetic<Half>();                         // false

    // Multiple precision types in Boost, chapter 8 of book
    using Multiprecision = boost::multiprecision::cpp_dec_float_100;
    IsArithmetic<Multiprecision>();               // false

    return 0;
}
```

Even in this simple example we can see the advantages because we can check whether a given type satisfies certain requirements and if not we can terminate the program by using a *static assert* (for example).

We also give an introduction to the concept of *variadic templates*. These are functions that accept a variable number of arguments. A *template parameter pack* is a template parameter that accepts zero or more template arguments. A *function parameter pack* is a function parameter that accepts zero or more function arguments. A template with at least one parameter pack is called a *variadic template*.

We take an example to show the use of variadic templates. To this end, we define two function types, both of which return a value. However, the first type expects one or more input arguments while the second type expects zero or more input arguments:

```
// Functions with given return type with at least one argument
template <typename T, typename... Types>
    using FunctionTypeI = std::function<T (const T& arg, const
    Types... args)>;

// Functions with given return type with zero or more arguments
template <typename T, typename... Types>
    using FunctionTypeII = std::function<T (const Types... args)>;
```

We can now create functions of different *arities* (namely one, two and three) and we define them as special cases of the above function types:

```
// Variadic function class
double x = 2.0;
FunctionTypeI<double> g1 = [=](double d)-> double { return x*d*d; };
FunctionTypeI<double, double> g1A = [=](double d1, double d2)->
double { return x*d1*d2; };
std::cout << g1A(3.0, -4.0) << '\n';

double y = 3.0;
FunctionTypeII<double, double, double> g2
                  = [=](double d1, double d2)-> double
                          { return y*d1*d2; };

y = 4.0;
using Alias = FunctionTypeII<double, double, double, double>;
Alias g3 = [=](double d1, double d2, double d3)-> double { return
y*d1*d2*d3; };
```

The next question would be to ask how variadic templates can be applied in order to reduce code bloat in applications. A general answer is that variadic templates can be used in *supplier code* (which makes it highly reusable) and then instantiated to suit the requirements of *clients*, as seen in the above simple examples. This removes the need to declare separate functions for each specific arity, for example.

Closely related to the topics in this chapter is the concept of *template metaprogramming* (Abrahams and Gurtovoy, 2005; Alexandrescu, 2001). To understand this concept, we define a *metafunction* as a program that manipulates code. In fact, it is a function that operates on metadata that can be invoked at compile-time.

In most of your development work you will probably not need to write code based on metaprogramming principles. However, there are applications where it is useful (Abrahams and Gurtovoy, 2005), in particular *domain-specific embedded languages* (DSELs) as a way to exploit templates. A discussion of these topics is outside the scope of this book.

6.2 SOME BUILDING BLOCKS

We introduce `std::integral_constant<>` and its aliases in this section. This class wraps a static constant of a specified type:

```
template< class T, T v > struct integral_constant;
```

It has a member that contains the static wrapped constant `value` which can be accessed as follows:

```
using TWO = std::integral_constant<int, 2>;
using CHARTWO = std::integral_constant<char, '2'>; // char OK
//using DECTWO = std::integral_constant<double, 2.0>; // illegal

std::cout << "TWO: " << TWO::value << '\n';
std::cout << "CHAR TWO: " << CHARTWO::value << '\n';

using FOUR = std::integral_constant<int, 4>;
static_assert(TWO::value*FOUR::value == 8, "2*4 != 8");
```

We note that the specified type must be of integral type. The standard also defines a number of aliases:

```
template <bool B>
    using bool_constant = integral_constant<bool, B>;

true_type -> std::integral_constant<bool, true>
false_type -> std::integral_constant<bool, false>.
```

An example of use is:

```
// C++14
std::bool_constant<true> bTrue;
std::bool_constant<false> bFalse;
static_assert(bTrue == true, "Ouch,true is not true");       // OK
static_assert(bFalse == false, "Ouch,false is not false");   // OK
//static_assert(bFalse == true, "What's up");           // Compile
                                                        // error
```

6.3 C++ TYPE TRAITS

Type traits can be considered as template utilities based on metaprogramming techniques and they provide a mechanism to define behaviour based on type. They can also be used to optimise code for types that provide special functionality. These utilities help both *application programmers* and *library implementors*.

We discuss the different type traits categories and we give some examples. The important thing to remember is that we are working with types such as enums, classes, integral and floating-point types, *lvalue* and *rvalue* references and pointers as well as their underlying types and members.

Most of the examples are self-documenting and easy to understand. We now discuss the different categories in the type traits library.

6.3.1 Primary Type Categories

The functions in this category check whether a generic type is of a given type, for example whether a type is a function type. Intuitively, this category examines the essence of a data type (for example, is it a class or a floating-point type?) as it were while ignoring its members. We take some user-defined classes and functions:

```
class Shape
{
public:
    virtual void draw() = 0;
};

class Base
{
private:
    int y;
public:
    Base() {}
    void draw() {}
};

class Point : public Shape
{
public:
    Point() {}
    virtual void draw() override {}
};

void func()
{ // For test purposes

}
```

We encapsulate the 'yes or no' tests in a function:

```
#include <type_traits>

template <typename T>
    void TypeInformation(const T& t)
{ // Primary type categories

    // Numeric type
    std::cout << "Integral type? " << std::boolalpha << std::is_
    integral<T>::value;
    std::cout << "Floating-point type? " << std::boolalpha
              << std::is_floating_point<T>::value;
    std::cout << "Is null pointer? " << std::boolalpha << std::is_
    null_pointer<T>::value;
```

```
    std::cout << "Is pointer? " << std::boolalpha << std::is_
    pointer<T>::value;
    std::cout << "Is array? " << std::boolalpha << std::is_array
    <T>::value;
    std::cout << "Is class? " << std::boolalpha << std::is_class
    <T>::value;

    std::cout << "End ********************\n";
}
```

We test these functions as follows:

```
// Primary type categories
std::cout << "** Primary type categories **\n";
int j = 1;
TypeInformation(j);

Shape* s;
TypeInformation(s);

Point pt;
TypeInformation(pt);

TypeInformation(nullptr);

auto f = std::bind(&Point::draw, pt);
TypeInformation(f);
TypeInformation(&Point::draw);

TypeInformation(func);
```

The output is:

```
** Primary type categories **
Integral type? true
Floating-point type? false
Is null pointer? false
Is pointer? false
Is array? false
Is class? false
End ********************
Integral type? false
Floating-point type? false
Is null pointer? false
Is pointer? true
Is array? false
Is class? false
End ********************
Integral type? false
Floating-point type? false
```

```
Is null pointer? false
Is pointer? false
Is array? false
Is class? true
End ********************
Integral type? false
Floating-point type? false
Is null pointer? true
Is pointer? false
Is array? false
Is class? false
End ********************
Integral type? false
Floating-point type? false
Is null pointer? false
Is pointer? false
Is array? false
Is class? true
End ********************
Integral type? false
Floating-point type? false
Is null pointer? false
Is pointer? false
Is array? false
Is class? false
End ********************
Integral type? false
Floating-point type? false
Is null pointer? false
Is pointer? false
Is array? false
Is class? false
End ********************
```

6.3.2 Composite Type Categories

The functions in this category check if a type is scalar, compound or object, for example. In a sense, these functions provide information regarding the structure of a type. We encapsulate the tests in a function:

```
template <typename T>
    void TypeInformationII(const T& t)
{ // Composite type categories

    // arithmetic type, void, or nullptr_t
    std::cout << "Fundamental type? " << std::boolalpha
              << std::is_fundamental<T>::value << std::endl;

    // an integral type or a floating-point type
    std::cout << "Arithmetic type? " << std::boolalpha
              << std::is_arithmetic<T>::value << std::endl;
```

```
    // arithmetic type, enumeration type, pointer, pointer to
    // member,
    // or std::nullptr_t, including any cv - qualified variants),
    std::cout << "Is scalar? "
              << std::boolalpha << std::is_scalar<T>::value
              << std::endl;
    std::cout << "Is object? "
              << std::boolalpha << std::is_object<T>::value
              << std::endl;

    // array, function, object pointer, function pointer, member
    // object pointer, member function pointer, reference,
    // class,union,enumeration, including cv - qualified variants),
    std::cout << "Is compound? "
              << std::boolalpha << std::is_compound<T>::value
              << std::endl;
    std::cout << "Is reference? " << std::boolalpha
              << std::is_reference<T>::value << std::endl;

    std::cout << "Is non-static member function pointer? "
              << std::boolalpha << std::is_member_pointer<T>::
              value;
    std::cout << "End ********************\n";
}
```

The test code is:

```
// Composite type categories
std::cout << "** Composite type categories **\n";
int n = 10;
TypeInformationII(n);

std::vector<std::complex<double>> arr(10);
TypeInformationII(arr);
```

The output is:

```
** Composite type categories **
Fundamental type? true
Arithmetic type? true
Is scalar? true
Is object? true
Is compound? false
Is reference? false
Is non-static member function pointer? false
End ********************
Fundamental type? false
Arithmetic type? false
Is scalar? false
Is object? true
```

```
Is compound? true
Is reference? false
Is non-static member function pointer? false
End ********************
```

6.3.3 Type Properties

The functions in this category can be seen as providing more detailed information concerning the functionality in Section 6.3.1. For example, we can determine if a class is abstract or if it has at least one virtual function:

```
template <typename T>
    void TypeInformationIII(const T& t)
{ // Type properties

    // A class with no data and no virtual functions
    std::cout << "Is class with no data? " << std::boolalpha
              << std::is_empty<T>::value << std::endl;
    std::cout << "Polymorphic class type? " << std::boolalpha
              << std::is_polymorphic<T>::value << std::endl;
    std::cout << "Abstract class type? " << std::boolalpha
              << std::is_abstract<T>::value << std::endl;


    std::cout << "End ********************\n";
}
```

The test code is:

```
// Type properties
std::cout << "** Type properties **\n";
Shape* sh;
TypeInformationIII(sh);

Shape* shape = new Point;
TypeInformationIII(*shape);
delete shape;

Base b;
TypeInformationIII(b);

Point p;
TypeInformationIII(p);
```

The output is:

```
** Type properties **
Is class with no data? false
Polymorphic class type? false
```

```
Abstract class type? false
End ********************
Is class with no data? false
Polymorphic class type? true
Abstract class type? true
End ********************
Is class with no data? false
Polymorphic class type? false
Abstract class type? false
End ********************
Is class with no data? false
Polymorphic class type? true
Abstract class type? false
End ********************
```

6.3.4 Type Relationships

The functions in this category examine relationships between types, for example whether a class is derived from another class:

```
template <typename T1, typename T2>
    void TypeInformationIV()
{ // Type relationships

    // A class with no data and no virtual functions
    std::cout << "Is base of? " << std::boolalpha
              << std::is_base_of<T1, T2>::value << std::endl;
    std::cout << "Is same class? " << std::boolalpha
              << std::is_same<T1, T2>::value << std::endl;
    std::cout << "Is convertible from/to? " << std::boolalpha
              << std::is_convertible<T1,T2>::value << std::endl;

    std::cout << "End ********************\n";
}
```

The test code is:

```
// Type relationships
std::cout << "** Type relationships **\n";
TypeInformationIV<Shape, Point>();
TypeInformationIV<Point, Shape>();
TypeInformationIV<Shape*, Point>();
TypeInformationIV<Point*, Shape>();
TypeInformationIV<Shape*, Point*>();
TypeInformationIV<Point*, Shape*>();
```

The output is:

```
** Type relationships **
Is base of? true
```

```
Is same class? false
Is convertible from/to? false
End ********************
Is base of? false
Is same class? false
Is convertible from/to? true
End ********************
Is base of? false
Is same class? false
Is convertible from/to? false
End ********************
Is base of? false
Is same class? false
Is convertible from/to? false
End ********************
Is base of? false
Is same class? false
Is convertible from/to? false
End ********************
Is base of? false
Is same class? false
Is convertible from/to? true
End ********************
```

6.3.5 'Internal Properties' of Types

We show how to examine several properties of member functions of a type, in particular constructors and destructors:

```
template <typename T>
    void MethodInformation(const T& t)
{ //

    // Numeric type
    std::cout << "Virtual destructor? " << std::boolalpha
              << std::has_virtual_destructor<T>::value << std::endl;
    std::cout << "Default constructible? " << std::boolalpha
              << std::is_default_constructible<T>::value << std::endl;
    std::cout << "Copy constructible? " << std::boolalpha
              << std::is_copy_constructible<T>::value << std::endl;
    std::cout << "Move constructible? " << std::boolalpha
              << std::is_move_constructible<T>::value << std::endl;
    std::cout << "Copy assignable? " << std::boolalpha
              << std::is_copy_assignable<T>::value << std::endl;
    std::cout << "Move assignable? " << std::boolalpha
              << std::is_move_assignable<T>::value << std::endl;
    std::cout << "Destructible? " << std::boolalpha
              << std::is_destructible<T>::value << std::endl;

    std::cout << "End ********************\n";
}
```

The test code is:

```
// Type 'internals
std::cout << "**Type 'internals **\n";
int j = 1;
MethodInformation(j);

Point pt;
MethodInformation(pt);

Shape* s;
MethodInformation(s);
```

The output is:

```
**Type 'internals'**
Virtual destructor? false
Default constructible? true
Copy constructible? true
Move constructible? true
Copy assignable? true
Move assignable? true
Destructible? true
End ********************
Virtual destructor? false
Default constructible? true
Copy constructible? false
Move constructible? false
Copy assignable? true
Move assignable? true
Destructible? true
End ********************
Virtual destructor? false
Default constructible? true
Copy constructible? true
Move constructible? true
Copy assignable? true
Move assignable? true
Destructible? true
End ********************
```

6.3.6 Other Type Traits

There are a number of other categories that we do not discuss as they are easy to understand and to apply. See Exercise 6.

6.4 INITIAL EXAMPLES OF TYPE TRAITS

Having discussed the functionality in the type traits library we may ask ourselves what this functionality is good for and how it can be used to improve code quality in general.

6.4.1 Simple Bridge Pattern

In Section 6.2 we introduced the aliases:

```
true_type -> std::integral_constant<bool, true>
false_type -> std::integral_constant<bool, false>
```

To take an example, let us consider writing code to model *vector* (or *linear*) *spaces* (Shilov, 1977). In order to motivate the following topics we introduce some definitions. A *field* is a set of elements that can be added and multiplied together. Multiplication is *commutative* ($a * b = b * a$), there exists a unique unit element for multiplication and each non-zero element has a multiplicative inverse. Examples of fields are: the set of all (non-zero) real numbers, the set of all rational numbers, the set of all complex numbers and the set of all 2×2 matrices with real coefficients and having non-zero determinant. These fields can be modelled in C++. A *vector space V over a field K* generalises the well-known concepts of vectors in two- and three-dimensional space and we can define vector addition and multiplication by scalar field values:

- A1: $(x + y) + z = x + (y + z), \quad x, y, z \in V$.
- A2: $x + y = y + x$.
- A3: There exists a unique 0 in V such that $0 + x = x + 0 = x$.
- A4: For each x in V there exists a unique y such that $x + y = 0$ (the negative of x), called $-x$.

And:

- B1: $a(x + y) = ax + ay$.
- B2: $(a + b)x = ax + bx$.
- B3: $(ab)x = a(bx)$.
- B4: $1x = x$ (1 is the unit element).

In the above axioms we assume $x, y, z \in V, a, b \in K$.
We mention two important examples of vector spaces:

- n-Dimensional vectors over a field:

$$x = (x_1, \ldots, x_n) \quad (x_j \in K, j = 1, \ldots, n)$$
$$y = (y_1, \ldots, y_n) \quad (y_j \in K, j = 1, \ldots, n)$$

- $x, y \in K^n$

$$(x + y) = (x_1 + y_1, \ldots, x_n + y_n)$$
$$\lambda x = (\lambda x_1, \ldots, \lambda x_n), \lambda \in K$$

and

- Set of $m \times n$ matrices with values in K:

$$M = M(K; m, n)$$
$$m_1 = (a_{ij}), m_2 = (b_{ij})$$
$$m_1 + m_2 = (a_{ij} + b_{ij})$$
$$\lambda m_1 = (\lambda a_{ij})$$

where: $a_{ij}, b_{ij}, \lambda \in K, \ i = 1, \ldots, m, \ j = 1, \ldots, n$.

As an example of implementing axioms A1–A4 and B1–B4 in C++ we create code that is applicable to *both* scalar and vector types and we wish to provide a single interface to clients.

We use type traits to decide which implementation to apply depending on the data type of the input argument provided. The public interface that clients see is:

```
template <typename T>
    T Addition(const T& t1, const T& t2)
{ // Vector space addition

       // Best(?) approx for an array, open issue in C++11?
       return Addition_impl(t1, t2, std::is_compound<T>());
}
```

When calling this function the code will test whether the instantiated type `T` is a compound type. The two mutually exclusive options are:

```
// Scalar and Vector addition
template <typename T>
    T Addition_impl(const T& t1, const T& t2, std::true_type)
{ // Addition of arrays

    T result(t1);
    for (auto i = 0; i < t1.size(); ++i)
    {
       result[i] = t1[i] + t2[i];
    }
    return result;
}

template <typename T>
    T Addition_impl(const T& t1, const T& t2, std::false_type)
{ // Addition of scalars

       return t1 + t2;
}
```

If we implement this functionality in a class then the last two member functions (the '*implementations*') would be private while the member function `Addition()` would be public. We now take some examples of how to use this functionality:

```
std::cout << "\n\nAddition of arrays...\n";

std::vector<int> arr = { 1,2,3,4 };
std::vector<int> arr2 = { 1,2,3,4 };

auto sum = Addition<std::vector<int>>(arr, arr2);
print<std::vector<int>>(sum);

double x = 1.0; double y = 2.0;
auto scalarSum = Addition<double>(x, y);
print<double>(scalarSum);
```

Note the presence of the function to print scalars and vectors. Again, we use type traits to implement two versions depending on the type of the input argument:

```
template <typename T>
    void print(const T& t)
{
    print_impl(t, std::is_compound<T>());
}

template <typename T>
    void print_impl(const T& t, std::true_type)
{
    std::cout << "\nPrint a vector: ";
    for (auto it = std::begin(t); it != std::end(t); ++it)
    {
        std::cout << *it << ",";
    }
    std::cout << '\n';
}

template <typename T>
    void print_impl(const T& t, std::false_type)
{
    std::cout << "\nPrint a scalar value: " << t << '\n';
}
```

We see the above *mini-design* as a simple example of the *Bridge* design pattern because the function to add two types has two implementations, one for scalar types and one for vector types. The same technique was used to print scalar and array types.

6.5 GENERIC LAMBDAS

An interesting question that many developers ask is how to combine `auto` and templates. Well, you can't in the current version of C++! For example, one might expect the compiler to be clever enough to understand the following code:

```
template <typename T>
    auto yourLambda = [] (const T& v) const { return v[0]; }
```

Unfortunately, we get a compile error such as:

```
'auto yourLambda': cannot be a template definition'
```

But there is hope! A solution could be to wrap the code in a simple function object that does compile and that can be used by clients:

```
struct CGL2
{ // Generic lambda lookalike
```

```
template<typename T>
    auto operator () (const T& vec) const
    {
       return vec[0];              // Test case only

    };
};

// Simulating generic lambda
std::vector<int> v{ -132,2,3,4 };
std::vector<int> dq{ 42,2,3,4 };

CGL2 gL;
std::cout << gL(dq) << '\n';      // -132
std::cout << gL(v) << '\n';       // 42
```

The code compiles and runs but we have been forced to create a function object as wrapper in order to achieve this end. This approach results in so-called *code bloat*. Fortunately, this is not necessary and we can use the following *generic lambda*:

```
// Using generic lambda
auto f= [] (auto vec) { return vec[0]; };
std::cout << f(v) << '\n';
std::cout << f(dq) << '\n';
```

This is much easier!

Some other simple examples are:

```
// Generic lambda
auto glambda = [](auto a) { return a*2; };
auto glambdaPrint= [](auto a) { std::cout << a << '\n'; };

auto d = glambda(-3.0); glambdaPrint(d);

// Function return type deduction
auto f = [](auto x) { return x - 1; };
glambdaPrint(f(3.0));
```

We now think about how generic lambdas can be used as an alternative to templates. To this end, we take the example of creating wrappers for mathematical functions and algorithms, for example when we define preconditions such as input data being in a given range or that only floating-point types are allowed as input arguments. A function object solution could be:

```
template<typename T>
    struct AltSqrtGeneric
{ // Simulating a lambda function

    std::function<void (T&)> f;
```

```
    AltSqrtGeneric(const std::function<void (T&)>& func) : f(func) {}

    auto operator () (T x) -> decltype(x) const
    {
       f(x); // Usually a precondition check on input x

       // Restrict to floating point types
       static_assert(std::is_floating_point<decltype(x)>::value, "ouch");

       return std::sqrt(x);
    };
};
```

This code looks respectable enough but it is somewhat long winded. An example of use is:

```
double A = 10.0; double B = 20.0;
auto clamp = [=](double& x)->void { if (x <= A) x = A;if (x >= B) x = B; };

double z = 16.0;
AltSqrtGeneric<double> sqrt2(clamp);
auto val2 = sqrt2(z);
std::cout << "\nAlt sqrt of " << z << ", is " << val2 << '\n';
```

Using generic lambdas obviates the need to create a function object. The new code is:

```
auto MySqrtGenericLambda = [](auto x, auto f)
{ // Multiple checks

    f(x); // Usually a precondition check on input x

    return std::sqrt(x);

};
```

An example of use is:

```
double y = 90.0;
auto val = MySqrtGenericLambda(y, clamp);
std::cout << "\nsqrt of " << y << ", is " << val << '\n';
```

As a final example, we consider the problem of how to *compose* two functions f and g. This is possible only if the range of g is the same space as the domain of f. To start, here is the generic lambda function that realises this functionality:

```
auto compose = [](auto f, auto g)
             { return [=](auto&& ...x) { return f(g(x...)); }; };
```

We can now compose functions of arbitrary arity, as the following example shows:

```
// Scalar-valued functions of arity 1
auto f = [](double x) { return x*x;};
auto g = [](double x) { return std::sqrt(x);};
auto h = [](double x) { return std::exp(x);};

auto fg = compose(f, g);
std::cout << fg(2.0) << std::endl;
std::cout << f(g(2.0)) << std::endl;
auto hv = compose(h, fg);

// Scalar-valued functions functions of arity 2
auto f2 = [](double x) { return x; };
auto g2 = [](double x, double y) { return std::sqrt(x) + y; };
auto h2 = [](double x, double y) { return std::exp(x+y); };

auto fg2 = compose(f2, g2);
std::cout << fg2(2.0, 3.0) << std::endl;
std::cout << f2(g2(2.0, 3.0)) << std::endl;
auto hv2 = compose(h2, fg2);
std::cout << h(f2(g2(2.0,3.0))) << std::endl;
```

This is interesting syntax and it opens the door to defining *higher-order functions* in C++. A discussion is outside the scope of this book.

6.6 HOW USEFUL WILL GENERIC LAMBDA FUNCTIONS BE IN THE FUTURE?

We give a short discussion on applications of generic lambdas and provide motivating examples. Some of the questions that we can ask at this stage are:

- Q1: Which problems can be solved using templates and not with generic lambdas?
- Q2: Which problems can be solved using generic lambdas and not with templates?
- Q3: How do we use generic lambdas in combination with variadics?
- Q4: What are the consequences of using generic lambdas (readability, maintainability)?

We discuss these topics in the following sections.

6.6.1 Duck Typing and Avoiding Class Hierarchies

Duck typing is concerned with establishing the suitability of an object for some purpose. For example, let us consider an algorithm that calls methods and properties of objects whose instantiated types are specified somewhere else in client code. In other words, the emphasis is on methods and not on the actual type of the object. Duck typing is distinct from subtype polymorphism in that no explicit interface is defined in the former case. It works in a manner similar to subtype polymorphism, except that inheritance is not needed. *The run-time system only checks if arguments implement the appropriate methods*.

We take an initial example. In this case we add two objects, and we are implicitly assuming that instantiated types implement the operator '+'. The first solution applies duck typing in a static typing context. Only the methods that are called at run-time need to be implemented:

```
// Adding numbers
template<class T, class U>
    auto add(T a, U b) -> decltype(a + b)
{
    return a + b;
}
```

The second solution is a generic lambda function and it uses the `auto` specifier which implies that the variable being declared is automatically deduced from its initialiser:

```
auto addII = [] (auto a, auto b)
{
    return a + b;
};
```

Some compilable and non-compilable examples are:

```
// No operator "+" defined
class C {};

// Adding numbers
double a = 10.0; double b = 12.0;
std::cout << "Add I: " << add(a, b) << '\n';
std::cout << "Add II: " << addII(a, b) << '\n';

// Concatenation of strings, OK then, we got lucky
std::cout << "Add III: " << addII(std::string("dd"), std::string("ih"));

// Class not implementing +, won't compile
C c1; C c2;
// std::cout << "Add IV: " << add(c1, c2) << '\n';
// std::cout << "Add V: " << addII(c1, c2) << '\n';

std::complex<double> com1(1.0, 2.0);
std::complex<double> com2(2.0, 1.0);
std::cout << "Add IV: " << add(com1, com2) << '\n';
std::cout << "Add V: " << addII(com1, com2) << '\n';
```

Summarising, we can use both C++ templates and the `auto` specifier to implement duck typing at compile-time without the need to create a class hierarchy. We now extend the discussion by taking an example that is a simple model problem when pricing financial derivatives. We exclude non-relevant details at this stage. The main focus is on showing how polymorphism is realised by creating types and classes from *disparate hierarchies*. This promotes maintainability and avoids monolithic class libraries. The toy classes are:

```
class Bond
{ // Simplified class
private:
    // data
public:
    Bond() {}

    double price(double u) const { return u; }
    void print(double p) const { std::cout << "a bond: " << p; }
};

class Option
{ // Simplified class
private:
    // data
public:
    Option() {}

    double price(double u) const { return u; }
    void print(double p) const { std::cout << "an option: " << p; }
    double delta(double u) { return u; }
};

class Widget
{ // Something completely different
private:
    // data
public:
    Widget() {}

    double price(double u) const { return u; }
    void print(double p) const { std::cout << "a widget: " << p; }
};
```

The next stage is to use these classes, in particular calling their methods in the steps of an algorithm. For example, we compute the price of some object and then we print the price:

```
// Polymorphism
template <typename Derivative>
    void ComputeAndDispatch(const Derivative& der, double u)
{ // Data flow algorithm, similar to a continuation

    double p = der.price(u);
    der.print(p);
}

// Need a new name for ComputeAndDispatch (otherwise
// it's a redefinition)
```

```
auto ComputeAndDispatchII = [](auto der, double u)->void
{
    double p = der.price(u);
    der.print(p);
};
```

An example of use is:

```
{
    double u = 60.0;
    Bond bond; ComputeAndDispatch(bond, u);
    Option option; ComputeAndDispatch(option, u);
    Widget widget; ComputeAndDispatch(widget, u);
}

{
    double u = 60.0;
    Bond bond; ComputeAndDispatchII(bond, u);
    Option option; ComputeAndDispatchII(option, u);
    Widget widget; ComputeAndDispatchII(widget, u);
}
```

You can compare the solutions and decide which one you prefer.

In general, duck typing gets its name from the well-known cliché 'if it quacks like a duck, walks like a duck then it is a duck'. In other words, your class does not have to be derived from a duck; it only needs to have the same methods as a duck. This approach can be seen as a kind of *compile-time design pattern*.

6.6.2 Something Completely Different: Homotopy Theory

We give a short introduction to a branch of topology called *homotopy theory* (Hocking and Young, 1961) and in particular homotopic mappings. We chose this example to show the power of generic lambdas. We consider a *parametrised family of mappings* of a space X to a space Y. A *homotopic mapping* is a continuous function $h : X \times C \to Y$ where C is called the *parameter space*. In the following we shall assume that C is the *closed unit interval* $I = [0, 1]$.

Let f and g be two mappings of a space X onto a space Y. They are called *homotopic* (we write $f \approx g$) if there is a mapping $h : X \times I \to Y$ such that for each point x in X:

$$h(x, t) = h_1(t)f(x) + h_2(t)g(x) \tag{6.1}$$

where $h_1(t)$ and $h_2(t)$ are functions satisfying:

$$h_1(0) = 1, \quad h_2(0) = 0$$
$$h_1(1) = 0, \quad h_2(1) = 1$$

resulting in:

$$h(x,0) = h_1(0)f(x) + h_2(0)g(x) = f(x)$$
$$h(x,1) = h_1(1)f(x) + h_2(1)g(x) = g(x).$$

The mapping h is called a *homotopy* between f and g and the product space $X \times I$ is called the *homotopy cylinder.* How would we design a homotopy in C++? Given two mappings f and g we need to construct a homotopy that satisfies equation (6.1) . We assume that the functions f and g map n-dimensional space into one-dimensional space (they are the so-called *real-valued* functions). A specific example to show how to define a homotopy in C++ is:

```
auto homotopy = [=](auto f, auto g)
{ // Homotopy of functions

    return [=](auto&& t, auto&&... x)
    {
        return (1-t)*f(x...) + t*g(x...);
    };
};
```

We see that the return type is in fact a *higher-order function*. In client code we work with concrete function spaces and associated operators, such as:

```
// Function maps Domain to Range
template <typename R, typename D>
    using FunctionType = std::function<R (const D x)>;

FunctionType<double, double> f = [](double x) {return x*x; };
FunctionType<double, double> g = [=](double x) { return x; };

auto hom = homotopy(f, g);
double x = 3.0;
double t = 0.5;
std::cout << "Homotopy: " << hom(x, t) << '\n';

// Extreme cases
t = 0.0;
std::cout << "Homotopy (t=0): " << f(x) << ", " << hom(t, x) << '\n';
t = 1.0;
std::cout << "Homotopy (t=1): " << g(x) << ", " << hom(t, x) << '\n';
```

We can make the homotopy more flexible as follows:

```
auto homotopyII = [=](auto f, auto g, auto h1, auto h2)
{ // Homotopy of functions

    // t is usually a scalar while x can be a list of arguments

    return [=](auto&& t, auto&&... x)
```

```
    {
        return h1(t)*f(x...) + h2(t)*g(x...);
    };
};
```

An example of use is:

```
FunctionType<double, double> h1 = [](double t) {return 1.0 -t; };
FunctionType<double, double> h2 = [=](double t) { return t; };

auto hom2 = homotopyII(f, g, h1, h2);

t = 0.5;
x = 3.0;
std::cout << "Homotopy II: " << hom2(x, t) << '\n';
```

It is possible to simplify the homotopy to one that only needs the mappings f and g as input:

```
auto convexHomotopy = [=](auto f, auto g)
{ // Homotopy of functions

    // t is usually a scalar while x can be a list of arguments

    auto h1 = [](double t) { return 1.0 - t; };
    auto h2 = [](double t) { return t; };

    return [=](double x, auto&&... t)
    {
       return h1(t)*f(x...) + h2(t)*g(x...);
    };
};
```

In this case we have 'hidden' lambda functions h_1 and h_2 inside this generic lambda. An example of use is:

```
auto hom3 = convexHomotopy(f, g);
t = 0.25;
x = 4.0;
std::cout << "Homotopy III: " << hom3(t, x) << '\n'; // 13
```

We note that we have changed the relative order of the arguments in the definition of the homotopy in equation (6.1) and its implementation in C++. This is because the time variable `t` is always a scalar and the state variable is a point in n-dimensional space in general and it will be implemented as a variadic argument. In C++ the compiler complains if the variadic argument is placed before the fixed parameter, hence the reason why the relative order is different between the mathematical and C++ representations.

Finally, we give some examples of homotopy mappings that are used in numerical analysis (Allgower and Georg, 2003):

```
auto homotopyII = [=](auto f, auto g, auto h1, auto h2)
{ // Homotopy of functions

    return [=](auto&& t, auto&&... x)
    {
       return h1(t)*f(x...) + h2(t)*g(x...);
    };
};

auto convexHomotopy = [=](auto f, auto g)
{ // Homotopy of functions

    auto h1 = [](double t) { return 1.0 - t; };
    auto h2 = [](double t) { return t; };

    return [=](auto&& t, auto&&... x)
    {
       return h1(t)*f(x...) + h2(t)*g(x...);
    };
};

auto canonicalHomotopy = [=](auto f, auto&&... x0)
{ // Global homotopy of functions

    return [=](auto&& t, auto&&... x)
    {
       return f(x...) + (t - 1.0)*f(x0...);
    };
};
```

We conclude this section with a very brief discussion on how to apply homotopy methods to find the solution to nonlinear systems of equations (Allgower and Georg, 2003):

$$F(x) = 0, \quad F : \mathbb{R}^n \to \mathbb{R}^n.$$

In many cases it is difficult to find a solution, mainly because we need a good initial guess at the true solution. Homotopy methods are less sensitive to this problem and to this end we embed the above nonlinear system in a homotopy:

$$h(x,t) = (1-t)\{F(x) - F(x_0)\} + tF(x) = F(x) - (1-t)F(x_0)$$

where x_0 is an arbitrary initial guess.

We solve this modified system (see Allgower and Georg, 2003) from $t = 0$ to $t = 1$. At $t = 0$, the solution is $x = x_0$ (which we already know because x_0 is given) and we must come

up with a scheme that drives a solution to $t = 1$ in which case we have found the solution of the original problem. The C++ code to model the homotopy is:

```
auto canonicalHomotopy = [=](auto f, auto&&... x0)
{ // Homotopy of functions

    // t is usually a scalar while x can be a list of arguments

    return [=](auto&& t, auto&&... x)
    {
        return f(x...) + (t - 1.0)*f(x0...);
    };
};
```

An example of use is:

```
t = 0.25;
x = 6.0;

double x0 = 2.0;
auto hom5 = canonicalHomotopy(f, x0);
std::cout << "Homotopy V: " << hom5(t, x) << '\n'; // 33
```

The numerical details of solving nonlinear systems using *homotopy/continuation* methods are given in Allgower and Georg (2003).

6.7 GENERALISED LAMBDA CAPTURE

We have already discussed captured variables in the context of lambda functions in Chapter 3. In this section we discuss several *capture modes*:

- S1: By value: the value is copied into the body of the lambda function.
- S2: By reference: the lambda function uses the 'live' variable that is defined outside of the function.
- S3: Generalised lambda capture (init capture) using local (stack) variables: the variable is only defined in the body of the lambda function (this is similar to the use of local variables in the *Boost Phoenix* library (Demming and Duffy, 2010).
- S4: Using move semantics: the encompassing variable is moved into a variable in the lambda function.

The main challenge is to determine which of these options to use in a given context. Each option has its own pitfalls. For example, option S1 may not work for types for which the copy constructor is not defined. Reference capture (case S2) can lead to *dangling references*, for example when a variable goes out of scope before the lambda function that uses it is called. Thus, when the lambda function has been called the variable no longer exists!

In this section we focus on cases S3 and S4. To this end, an example of S3 is:

```
void CaptureLocalVariable()
{
    // C++14 init capture with local variables
    int x = 1'000; // This is a digit separator
    auto fun4 = [&r = x]()
    {
       r *= 2;
       std::cout << "\nr: " << r << '\n';
    };

    // Undefined r
    // std::cout << r << '\n';

    fun4();  // Prints the value 2000
}
```

In this case the lifetime of the *local variable* `r` is the scope of the lambda function itself.

We now take an example of a `mutable` lambda function which processes a vector that has already been initialised outside the scope of the lambda function and that is modified in the body of the function. We present four alternatives.

First, capture by reference is:

```
int size = 4; double val = 2.71;
std::vector<double> data(size, val);

// By reference capture mode
auto fun = [&data]() mutable
{
    std::cout << "Vector by reference, values being changed\n";
    for (std::size_t i = 0; i < data.size(); ++i)
    {
       data[i] = 3.14; std::cout << data[i] << ",";
    }

    std::cout << '\n';
};
fun();

// data still exists and has been modified
for (std::size_t i = 0; i < data.size(); ++i)
std::cout << data[i] << ":"; }
```

Second, capture by value is:

```
std::cout << "Capture by value:\n";
int size = 4; double val = -99.99;
std::vector<double> data(size, val);
```

```
// all captured variables are copied
auto fun4 = [=]() mutable
{
    for (std::size_t i = 0; i < data.size(); ++i)
       { data[i] = 42.00; std::cout << data[i] << ","; }
    std::cout << '\n';
};
fun4();

// What has changed? No, we get the old ones i.e -99.99!
for (auto elem : data) { std::cout << elem << ","; }
```

Third, capture by using local variables in combination with a move is:

```
// Using local variable for capture
int size2 = 6; double val2 = 5.0;
std::vector<double> newData(size2, val2);

auto fun3 = [&localData = std::move(newData)]() mutable
{
    for (std::size_t i = 0; i < localData.size(); ++i)
{ localData[i] = 148.413; std::cout << localData[i] << ","; }
    std::cout << '\n';
};
fun3();
```

Finally, the solution using move semantics is:

```
// C++14 init capture
int sz = 5; double d = 1.414;
std::vector<double> arr(sz, d);
auto fun2 = [myData = std::move(arr)]() mutable
{
    std::cout << '\n';
    for (std::size_t i = 0; i < myData.size(); ++i)
    { myData[i] = 93.8344; std::cout << myData[i] << ","; }
    std::cout << '\n';
};
fun2();
```

It is instructive to run the code. For more complex cases, it might be an option to use function objects instead of lambda functions.

6.7.1 Living Without Generalised Lambda Capture

Generalised lambda capture is a C++14 feature. Before C++14 we had to depend on C++11 in combination with `std::bind` in order to emulate the feature. You may prefer

to use the C++14 option instead due to the fact that `std::bind` syntax can be cryptic at times.

The example that we take is as before: we create a lambda function that accepts a vector as input and prints it. The point to note is that `std::bind` accepts a lambda function as input argument which is bound to a moved vector `std::move(data2)`, as the following code shows:

```
void CaptureEmulation()
{ // Emulating generalized lambda capture with C++11

    std::cout << "\nCapture in C++11 and std::bind\n";
    int size2 = 3; double val2 = 1.41;
    std::vector<double> data2(size2, val2);
    auto fun3 = std::bind([](std::vector<double>& array)
    {
        for (std::size_t i = 0; i < array.size(); ++i)
        {
           std::cout << array[i] << "/";
        }


    }, std::move(data2));
    fun3();

    if (0 == data2.size())
    {
       std::cout << "\nDouble array has no elements, OK\n";
    }
    else
    {
       std::cout << "\n Ouch!\n";
       for (std::size_t i = 0; i < data2.size(); ++i)
       {
           std::cout << data2[i];
       }
    }
}
```

The above code is quite clumsy in our opinion. C++14 syntax is preferable, hence we do not use the above code style in this book.

6.8 APPLICATION TO STOCHASTIC DIFFERENTIAL EQUATIONS

Object-oriented developers are familiar with subtype polymorphism and creating class hierarchies in applications. In general, we can create a base class consisting of pure or default `virtual` functions which are then overridden or can be overridden by derived classes. For example, this was the approach taken in Duffy and Kienitz (2009) to model classes for SDEs. In particular, we model drift and diffusion as functions that can be customised in derived classes.

While this is an accepted technique we do run the risk of creating redundant code by promoting objects to the level of a class (we discuss this topic in detail in Chapter 7).

In this section we create a single class that models an SDE. The class is composed of two universal functions that describe the drift and diffusion functions. It is up to clients to decide how to implement these functions when creating an instance of the SDE. We know from Chapter 3 that this can be realised by the following target methods:

- A free function (traditional function pointer).
- A stored lambda function or a lambda function.
- A function object.
- A binded member function or static member function of a class.

This approach reduces code bloat, especially at the server (supplier) side because only a single class is needed.

We first describe some supporting syntax and shorthand notation to make the resulting code more readable:

```
// Functions of arity 2 (two input arguments)
template <typename T>
    using FunctionType = std::function<T (const T& arg1, const T& arg2)>;

// Interface to simulate any SDE with drift/diffusion
// Use a tuple to aggregate two functions into what is similar to an
// interface in C# or Java.
template <typename T>
    using ISde = std::tuple<FunctionType<T>, FunctionType<T>>;
```

We are now in a position to post the class that models the SDE; we note that it makes use of universal function wrappers and tuples:

```
template <typename T = double>
    class Sde
{
private:
    FunctionType<T> dr_;
    FunctionType<T> diff_;

    T ic; // Initial condition
public:
    Sde() = default;
    Sde(const Sde<T>& sde2, const T& initialcCondition)
       : dr_(sde2.dr_), diff_(sde2.diff_), ic(initialcCondition) {}
    Sde(const ISde<T>& functions, const T& initialcCondition)
       : dr_(std::get<0>(functions)),
       diff_(std::get<1>(functions)), ic(initialcCondition) {}

    T drift(const T& S, const T& t) { return dr_(S, t); }
    T diffusion(const T& S, const T& t) { return diff_(S, t); }
};
```

We instantiate this class when creating SDEs for *geometric Brownian motion* (GBM) and *constant elasticity of variance* (CEV) models:

```
// Second test case; GBM dS = a S dt + sig S dW
double r = 0.08; double sig = 0.3;
auto drift = [&r](double t, double S) { return r * S; };
auto gbmDiffusion = [&sig](double t, double S) { return sig * S; };

double beta = 0.5;
auto cevDiffusion = [&sig, &beta](double t, double S)
                    { return sig * std::pow(S, beta); };
auto iFace = std::make_tuple(drift, gbmDiffusion);
ISde<double> gbmFunctions = std::make_tuple(drift, gbmDiffusion);
ISde<double> cevFunctions = std::make_tuple(drift, cevDiffusion);

// Create Sdes
double ic = 60.0;
Sde<> sdeGbm(gbmFunctions, ic);
```

We see that we can create as many instances of `SDE<>` as we want by defining a single lightweight class and using *in-place* lambda functions.

6.8.1 SDE Factories

From the above code sample we see how to create instances of `SDE<>`. In larger applications this ad-hoc approach breaks down in the sense that the code to initialise the data and functions that are used to create instances of `SDE<>` tend to become unmaintainable and unreadable. For this reason we apply the *separation of concerns* approach by creating code blocks with each block having a single responsibility. A typical example is to create loosely coupled modules for the following actions that we describe in a top-down manner:

1. Create multiple instances of `SDE<>`; create a specific instance of `SDE<>` based on some user choice.
2. Create the functions that are the components of the `SDE<>` instances in step 1.
3. Create the data that the functions in step 2 need.

Each of these steps can be executed by dedicated *factory objects* that are instances of *creational design patterns* as described in GOF (1995). We give an initial example of code in which steps 1, 2 and 3 are amalgamated into one code block. In later chapters we can refine this process even more when we use the *Builder* design pattern (GOF, 1995) to configure a complete application. We take a first example of a function that creates an *arithmetic SDE* (notice that the data is hard coded but the code can be generalised later). We use an alias for the factory class:

```
// Interface for a factor to create SDE<T> interfaces
    template <typename T>
             using SdeFactory = std::function<std::shared_ptr
             <Sde<T>> () >;
```

The factory function to create an SDE instance and an example of use is:

```
template <typename T> std::shared_ptr<Sde<T>> ArithmeticSde()
{ // dx = nu dt + sig dW, after transformation x = log(S)

    std::cout << "Arithmetic\n";

    // Initial condition for SDE
    double S0 = 60.0;
    // Necessary data
    T r = 0.08; T sig = 0.3;
    double nu = r - 0.5 * sig * sig;

    // Create the components of the SDE
    auto drift = [=](T t, T S) { return nu; };
    auto diffusion = [=](T t, T S) { return sig; };

    ISde<T> functions = std::make_tuple(drift, diffusion);

    // Create Sde
    return std::shared_ptr<Sde<T>>(new Sde<T>(functions, S0));
}

// Switch to a new factory and generate a sde
SdeFactory<float> arithmeticSdeFactory = ArithmeticSde<float>;
auto sde3 = arithmeticSdeFactory();
std::cout << "Drift, diffusion: " << sde3->drift(t, S) << ", "
          << sde3->diffusion(t, S) << '\n';
```

Instead of a new factory, we could have recycled as it were:

```
sdeFactory = ArithmeticSde<float>;
auto sde3 = sdeFactory();
std::cout << "Drift, diffusion: " << sde3->drift(t, S) << ", "
          << sde3->diffusion(t, S) << '\n';
```

Finally, we can simulate the *Factory Method* pattern (GOF, 1995) by creating a function with an internal switch allowing us to choose the type of SDE that we are interested in:

```
template <typename T>
    std::shared_ptr<Sde<T>> ChooseSde(int choice)
{ // Simple factory method

    // Initial condition for SDE
    double S0 = 60.0;

    if (1 == choice)
    {
       std::cout << "GBM\n";
       // Second test case; GBM dS = a S dt + sig S dW
```

```
        T r = 0.08; T sig = 0.3;
        auto drift = [=](T t, T S) { return r * S; };
        auto gbmDiffusion = [=](T t, T S) { return sig * S; };

        ISde<T> gbmFunctions = std::make_tuple(drift, gbmDiffusion);


        // Create Sde
        return std::shared_ptr<Sde<T>>(new Sde<T>(gbmFunctions, S0));
    }
    else
    {
        std::cout << "CEV\n";
        double r = 0.08; double sig = 0.3;
        auto drift = [=](double t, double S) { return r * S; };
        auto gbmDiffusion = [=](double t, double S) { return sig * S; };

        double beta = 0.5;
        auto cevDiffusion = [=](double t, double S) { return sig * std::
        pow(S, beta); };

        ISde<double> cevFunctions = std::make_tuple(drift, cevDiffusion);

        // Create Sde
        return std::shared_ptr<Sde<T>>(new Sde<T>(cevFunctions, S0));
    }
}
```

An example of use is (notice the clever application of `std::bind`):

```
// Higher-level factory, can switch
SdeFactory<float> sdeFactory = std::bind(ChooseSde<float>, 1);
auto sde2 = sdeFactory();

std::cout << "Drift, diffusion: " << sde2->drift(t, S) << ", "
          << sde2->diffusion(t, S) << '\n';
```

6.9 EMERGING MULTIPARADIGM DESIGN PATTERNS: SUMMARY

In Section 6.8 we applied modern C++ syntax to design a small framework to model stochastic differential equations that we shall generalise in Chapters 31 and 32. We have avoided subtype polymorphism and instead we took an approach that is based on creating a single class composed of universal function wrappers. This is similar to *interface programming* in languages that support interfaces (such as C# and Java, for example). In general, an *interface* is a *pure specification* or protocol and it describes a collection of abstract methods. An interface may not contain non-abstract methods nor may it contain member data. In this sense it is very different from an abstract class in C++ (even one without non-abstract member functions and

member data). Superficially, abstract classes and interfaces are similar. C++ does not support interfaces; however, we can emulate them by creating (variadic) tuples whose elements are universal function wrappers. In this case we define our interface emulator as:

```
template <typename T>
    using ISde = std::tuple<FunctionType<T>, FunctionType<T>>;
```

We create an instance of `ISde` as follows, for example:

```
auto drift = [=](T t, T S) { return nu; };
auto diffusion = [=](T t, T S) { return sig; };

ISde<T> functions = std::make_tuple(drift, diffusion);
```

We then use functions in a constructor of `Sde<>` as already discussed. Another way to emulate interfaces is to create a struct that is composed of one or more universal function wrappers.

6.10 SUMMARY AND CONCLUSIONS

In this chapter we introduced several advanced language features in C++ that can be used when creating generic libraries, generic lambda functions and when moving to multiparadigm design methods. We also show how to define *higher-order functions* that can be used in numerical analysis and computational finance. We shall extend these functional programming features in later chapters.

In Section 6.9 we included a summary of the essential features of what a multiprogramming design approach entails in the context of classes that model stochastic differential equations. A relevant set of questions is given in Exercise 12, in which we discuss design issues from a number of viewpoints.

6.11 EXERCISES AND PROJECTS

1. Give the top two advantages of *type traits*:
   a) Creating type-independent code.
   b) 'Compile-time' *reflection*.
   c) It is a replacement for subtype polymorphism.
   d) It is used to add properties to C++ types.
2. Consider the code:

```
// Testing arithmetic types
std::cout << "int*: " << std::boolalpha << std::is_arithmetic<int*>::value;

std::cout << "std::complex<double>: " << std::boolalpha
          << std::is_arithmetic<std::complex<double>>::value;

std::cout << "char: " << std::boolalpha << std::is_arithmetic<char>::value;
```

```
std::cout << "std::bitset<8>: " << std::boolalpha
          << std::is_arithmetic<std::bitset<8>>::value;
```

Which option is the correct output?

a) `false,false,false,false`.
b) `false,false,true,false`.
c) `false,true,false,true`.
d) `false,true,true,true`.

3. (Arithmetic Types)
a) Which of the following types are of arithmetic type? `int`, `int const`, `int &`, `int *`, `float`, `float const`, `float &`, `float *`.
b) Consider `std::complex<double>`. Does it fit into the following *composite type categories*? `is_{fundamental, arithmetic, scalar, object, compound}`.
c) In which composite type categories does `std::array<>`fit? Is it a non-union class type?

4. Which statements describe a *variadic function*?
a) Arguments whose types are variant.
b) Takes a variable number of template arguments.
c) Takes a variable number of arguments of specific types.
d) C++11 does not support variadic functions; they are supported in C++14.

5. Which of the following statements are true regarding a *variadic function*?
a) It is declared using ellipses.
b) It is not possible to make a copy of the variadic function arguments.
c) It accepts any number of arguments.
d) A template with at least one *parameter pack* is called a *variadic template.*

6. (Other Type Traits, Investigation)

This exercise entails determining which type traits functions to use for the following functionality. Having discovered the appropriate functions then create some code with two specific types to show how it works:

a) Is a type either a signed or an unsigned arithmetic type?
b) Make a given integral type signed/unsigned.
c) Obtain the number of dimensions of an array type.
d) Remove/add a pointer from or to a given type.
e) Remove/add a reference from or to a given type.

7. (Pointers and Non-pointers)

This exercise consists of calling some functions from the *Primary type* category. Answer the following questions:

a) Write a function to determine if a type is a pointer, null pointer, *lvalue* reference or *rvalue* reference.
b) Determine if a type is a member function pointer or if it is a pointer to a non-static member object.
c) Is a shared pointer a pointer type? Is it a pointer type when converted to a raw pointer?

Typical code is:

```
template <typename T>
    void IsPointer(const T& t)
{ // First example of type_traits;check if t is a pointer
```

```
    // Return type is std::true_type or std::false_type
    if (std::is_pointer<T>::value)
    {
       std::cout << "This is a pointer type argument\n";
    }
    else
    {
       std::cout << "_not_ a pointer type argument\n";
    }
}
```

8. (Simple Switchable *Bridge* Functionality)

We create a template function that supports both pointers and reference types. If it is a pointer it is dereferenced and then printed while if it is not a pointer type and if it is a scalar reference type then it is printed directly. Use the `is_pointer()` function in conjunction with `std::true_type` and `std::false_type` to determine which implementation to call.

9. (Array Categories)

We discuss some functions that work with array types. The *rank* of an array type is equal to the number of dimensions of the array. The *extent* of an array type is the number of elements along the *N*th dimension of the array if *N* is in the closed interval `[0, std::rank<T>::value]`. For any other type, the value is 0.

Answer the following questions:

a) Test `std::is_array()` on a range of fundamental, scalar, object, arithmetic and compound types.

b) Create an array `int [][3][4][5]`. Find its rank and extent.

c) Call `std::remove_extent()` and `std::remove_all_extent()` on the array in question b). What is happening?

10. (Conversions)

Sometimes you may wish to structurally change fundamental properties of types in an application. Examples are:

- Mapping integers to unsigned integers and vice versa.
- Adding/removing the `const` specifier to or from a type.
- Adding/removing a pointer to or from a type.
- Adding/removing the `volatile` specifier to or from a type.

Answer the following questions:

a) Write a separate function for each of the above requirements.

b) Test the functions on a range of fundamental, scalar, object, arithmetic and compound types.

11. (Capture Modes, Brainstorming Question)

In Sections 6.5, 6.6 and 6.7 we introduced generic lambda functions, some examples and how to deal with captured variables. It is natural to ask which solution is most appropriate in a given context. We have given code examples in this chapter to help you answer the following questions. Of course, a definitive answer can only be given when you actually develop your own applications and make your own conclusions:

a) Compare generic lambdas and template functions with regard to reliability, maintainability and understandability.

b) We can capture by value, by reference, by generalised capture, by local variables and by emulation using `std::bind`. Give the two best candidates from this list that satisfy the following requirements: maintainability, efficiency, understandability and interoperability with client code.

c) How do the techniques in part b) scale to larger and more complex code blocks? In which sense are encapsulation and information hiding potentially compromised (if at all) by the introduction of captured variables?

d) As a follow-on to question c), in which circumstances would we prefer to use function objects instead of the other techniques? Would they improve maintainability or efficiency?

e) Is there a need to emulate generalised lambda capture using C++11 and `std::bind`? Are there potential benefits?

f) How would you reimplement the code in Section 6.7.1 using a function object? Would it be a better solution?

12. (Creating Stochastic Differential Equations, Brainstorming)

The approach in Section 6.9 seems to be relatively new and innovative. We have asked several quant developers for their opinions on how useful it is as a competitor to their current design style.

We reproduce the answer from one user:

Question: Is this approach to polymorphism 'better' than e.g. subtype polymorphism?

Answer: I would say that the benefit of this kind of scheme is that there is only one class (`OneFactorProcess`) instead of one base class (which is nothing more than abstract interface) and several implementation classes. The drift and diffusion coefficients can be built outside the `OneFactorProcess` class. I see that by using this scheme we can increase flexibility on how the actual SDE (drift and diffusion coefficients) will be built. I think this is one of the strongest arguments for using this kind of scheme. Finally, it feels like a modern way to do things – at least for myself.

Incidentally, we use the alias:

```
template <typename T>
         using OneFactorProcess = Sde<T>;
```

Question: Does this approach lead to more maintainable code?

Answer: I would say yes.

Question: Does the approach scale to larger systems?

Answer: Well, if you consider my own path generator as a larger system, then yes. I see no reason why this scheme would not be implementable for any system, despite its complexity, *as long as all interfaces between components are well designed*.

Answer the following questions:

a) What would your answers be to these questions?

b) Read Section 6.9 again. Based on that short discussion is the approach a viable competitor to the traditional object-oriented programming style?

c) Consider an application that you created in the past. How would the design approach in Section 6.9 resolve possible maintenance issues that you had with the code?

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 7

Multiparadigm Design in C++

7.1 INTRODUCTION AND OBJECTIVES

In this chapter we discuss the integration of modern features in C++ with object-oriented design techniques. The goal is to design robust and flexible classes in C++. We strive to design classes that have a single responsibility and that also have minimal coupling with other classes. In other words, a class should depend on a small set of classes to which it delegates. Furthermore, we discuss how to implement class hierarchies using subtype polymorphism, *Curiously Recurring Template Pattern* (CRTP), *Strategy* design pattern and by using a mix of object-oriented and functional programming styles. Instead of having to create class hierarchies with possibly many derived classes we can create a single class that is composed of one or more universal function wrappers (instances of `std::function`). These functions can be assigned to a range of target methods as described in Chapter 3. We have already seen an example in Chapter 6 (Section 6.8) when we discussed stochastic differential equations. In this sense we note that function wrappers are competitors to `virtual` functions.

We also give an overview of object-oriented software metrics that measure *code quality* by providing metrics for classes, their members as well as class hierarchies.

In general, we see objects and classes as important during the detailed design and implementation phases of software development. We do not use them as *drivers* to decompose a software system into subsystems. We discuss this topic in Chapter 9. We introduce a number of high-level design techniques in this chapter and their realisation in C++.

Summarising, we integrate modern C++ syntax with design patterns. Later chapters of this book discuss these topics in more detail.

7.2 MODELLING AND DESIGN

We discuss two modelling techniques in this section; first, the *Liskov Substitution Principle* (LSP) and the *Single Responsibility Principle* (SRP). As an application of these design techniques we discuss the initial top-level design diagram for a one-factor Monte Carlo option price as preparation for Chapters 31 and 32. We offer a defined process to analyse, design and implement a range of applications

7.2.1 Liskov Substitution Principle

This principle concerns *strong behavioural subtyping* and it guarantees semantic interoperability of types in a hierarchy, in particular of object types. The subtype requirement can be stated as follows:

> Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.
>
> In other words: if S is a subtype of type T then objects of type T may be replaced by objects of type S in a program without altering any of the properties or correctness of that program.

We give an example of LSP. Consider the simple class hierarchy:

```
class B
{
public:
   B() {}
   B(const B& b2) {}
   virtual void sayHello() const
      { std::cout << "Base Hello\n"; }};

class D1: public B
{
public:
   D1() {}
   D1(const D1& d1) : B(d1) {}     // Substitution in action
   void sayHello() const override
      {std::cout << "D1 Hello\n"; }};

class D2: public B
{
public:
   D2() {}
   D2(const D2& d2) : B(d2) {}     // Substitution in action
   virtual void sayHello() const override
      { std::cout << "D2 Hello\n"; }};
```

We see an example of the substitution principle when implementing copy constructors in derived classes `D1` and `D2`. The interesting part is when client code takes references to the base class as input argument, for example:

```
void Print(const B& b)
{
   // This function can be called with any instance of a derived class of B
   b.sayHello();
}
```

This means that this function can be called with any object of a derived class as input argument:

```
template <typename T>
   using SP = std::shared_ptr<T>;

int main()
{
   SP<B> b;

   b = SP<B>(new D1());
   Print(*b);
   b = SP<B>(new D2());
   Print(*b);

   D1 d1;
   Print(d1);

   // Copy-ctors, apply substitution
   D1 d11(d1);

   D2 d2;
   D2 d21(d2);

   return 0;
}
```

Another form of subtyping is *structural subtyping*. Then type A is a subtype of type B if and only if A has at least as much state and functionality as B. Furthermore, its invariants are at least as strong as those of A. We recall that a *class invariant* (Meyer, 1997, p. 364) is a global property that must be preserved by all the member functions in a class. Invariants capture deep semantic properties.

Examples of invariants are:

- The size of a stack ADT is always non-negative.
- The width and height of a square are always equal.
- Stock price in the Black–Scholes formula is non-negative.

Many class hierarchies have been designed based on the structural subtyping principle. The real danger is that invariants may be violated by derived classes. An example is when publicly deriving a `Square` from a `Rectangle`, the latter having public methods to modify height and width. What happens to a square when its width has been modified? This example shows the potential dangers of using *implementation* that leads to incorrect semantic relationships between classes.

7.2.2 Single Responsibility Principle

This principle states that each module and class should have responsibility for a single part of the functionality in a software system. This responsibility should be encapsulated by the

module or class. In other words, a class should have one reason to change. We avoid merging two or more separate pieces of functionality in a class. Classes should focus on a *single concern*.

We are careful to ensure that classes do not violate SRP as it is very enticing to keep adding functionality to a class as time goes on. A fitting remark regarding an example from C++ is:

> *Herb Sutter already uses* `std::string` *as a prime example of a class with too many responsibilities (in the form of many member functions). Like it was said before,* `std::string` *should be a dynamic container of characters, nothing more.*

Closely related to SRP is the *Separation of Concerns* principle that separates a computer program into distinct sections or blocks in such a way that each block addresses a separate concern. In fact, each block should have well-defined interfaces and satisfy SRP. We shall see how to formalise these ideas in later chapters using techniques such as:

- Modular decomposition and *Information Hiding* (Parnas, 1972).
- Creating system context diagrams (De Marco, 1978 and Chapter 9 of this book).
- Layered designs (see POSA, 1996 and Chapter 11 of this book).

Compared to traditional object-oriented software development (which tends to be bottom-up and which uses classes as the driver of the software process) the approach taken in this book adopts a top-down approach by decomposing a system into orthogonal subsystems with well-defined interfaces. Eventually (during detailed design) each subsystem can be implemented by C++ classes.

A common design error is when a class has functionality for both semantic processing and presentation on a given media. For example, early versions of HTML performed both duties of describing the presentation of a document and its style. Furthermore, it is well known that multiple inheritance leads to maintenance problems and it can be argued that this is caused by the fact that its use violates SRP. In general, classes should not have functionality for input–output operations. A class with two major responsibilities is a design error.

7.2.3 An Example: Separation of Concerns for Monte Carlo Simulation

We take an example of a computer program to price one-factor plain options using the Monte Carlo method. We discuss the design of this problem in detail in Chapters 31 and 32 but for the moment we list the classes participating in the design (each one satisfies SRP). A decomposition based on the author's *Domain Architectures* approach (Duffy, 2004) is shown in Figure 7.1.

The *context diagram* is shown in Figure 7.1. It is a special case of the abstract context diagram for applications that belong to the *Resource Allocation and Tracking* (RAT) category. In general, RAT systems track requests in time and space and they produce corresponding reports relating to the status of these requests. In this case the goal is to compute the price of one-factor plain, barrier, lookback and Asian options using the Monte Carlo method. The

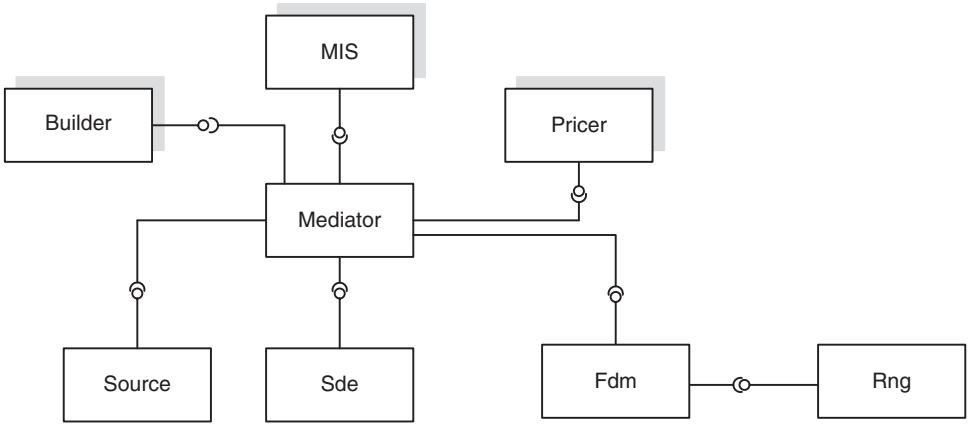


FIGURE 7.1 Context diagram for Monte Carlo engine

request is processed in a series of steps to produce the final output by the modules in the UML component diagram in Figure 7.1:

- *Source*: The system containing the data relating to the request, for example market and model data. It contains data that is needed by other modules in Figure 7.1.
- *Sde*: The system that models stochastic differential equations. In this case we model GBM and its variants. In particular, we are interested in modelling the drift and diffusion of some underlying variables such as the stock price or interest rate, for example.
- *Fdm*: The family of finite difference schemes that approximate the SDEs in the Sde system. In this case we use one-step difference schemes to advance the approximate solution from one time level to the next time level until we reach the desired solution at expiration. The finite difference schemes require the services of a module that computes random numbers.
- *Pricer*: This system contains classes to price one-factor options using Monte Carlo. The classes process path information from the Mediator and each class processes this path information in its own way. For example, for a plain option the pricer uses the path data at expiration, uses it to compute the payoff, adds the result to a running total and then discounts the result to compute the option price.
- *MIS*: This is the statistics-gathering system that receives status information concerning the progress of computation. For example, this system could display how many paths have been processed at any given time.
- *Builder*: This system implements a configuration/creational pattern (based on GOF, 1995) that creates and initialises the systems and their structural relationships in Figure 7.1. The newly created objects are encapsulated in a single tuple which adds to the overall maintainability of the system.
- *Mediator*: This is the central coordinating entity that manages the data flow and control flow in the system. It contains the state machine that computes the paths of the Sde. It also informs the other systems of changes that they need to know about. It plays the role of client in the *Builder* pattern (GOF, 1995).
- *Rng*: A system to generate random numbers. This feature is supported in C++11.

We explain this design in more detail in Chapters 9, 19 and 31. The current discussion is meant as a preview.

7.3 LOW-LEVEL C++ DESIGN OF CLASSES

In this section we introduce a number of language features that improve the robustness and reliability of C++ classes and member functions. We also pay attention to the different ways to construct objects and to initialise data. The features are *tactical* in the sense that they help to improve code quality at class level. A follow-on question is how to quantify code quality (software metrics) for object-oriented object networks and we give an introduction to this topic in Section 7.6. In Chapter 9 we discuss the design of large and complex software systems based on system decomposition principles.

7.3.1 Explicit Specifier

The `explicit` specifier specifies that a constructor or conversion function does not allow implicit conversions or copy initialisation. Its use avoids unexpected conversions. For example, an `explicit` default constructor can be used to perform both default initialisation and value initialisation. This means that attempting to create objects other than prescribed by the `explicit` constructor will result in a compiler error.

We take the following example:

```
struct B
{
   explicit B(int) {}
   explicit B(int, int) {}
   explicit operator int() const { return 0; }
};


class C
{
public:
   explicit C() {}
   C(int a, int b) {}
   explicit C(int a, int b, int c) {}

   // User-defined conversion
   operator int() const { return 0; }
   operator double() const { return 3.14; }
   operator B() const { return B(1); }    // Another class
};
```

We show what compiles and what does not compile as follows:

```
// Use of explicit
//C p = { 1,2,3 };                          // Not OK
```

```
C x(1, 2);                                  // OK
C y{ 1,2,3 };                               // OK
C y2(1, 2, 3);                              // OK
C z = { 1,2 };                              // OK
//C c = 1;                                  // Not OK
C c2;                                       // OK
int n = c2;                                 // OK
int m = static_cast<int>(c2);               // OK
if (n != m) std::cout << "oops\n"; else std::cout << "ok\n";
B b = c2;                                   // OK
```

We recommend that you inspect and run this code to make sure that you understand it.

7.3.2 Deleted and Defaulted Member Functions

The specifiers in this section are primarily concerned with C++03's *special member functions*, namely:

- Default constructor.
- Copy constructor that corresponds to member-wise construction of non-static data members. It is generated only if the class does not have a user-defined copy constructor.
- Copy assignment operator. Generation of this function in a class with an already user-defined copy constructor is deprecated.

These functions are generated only if they are needed. For example, a default constructor is generated only if a class declares no constructors whatsoever. Generated special member functions are implicitly public and inline. In C++11 the above set of special member functions has been extended to include the *move constructor* and the *move assignment operator*.

We use the keyword `default` to let the compiler generate the body of a constructor. At the other extreme we may wish to suppress the use of special member functions. The standard approach in C++03 is to declare them as `private` and not to define them. C++11 uses the specifier `delete` to mark these functions as *deleted* functions, which is an improvement on C++03 for a number of reasons:

- Deleted functions may not be used, even by friends.
- Deleted functions are `public`; C++ checks accessibility before deleted status.
- Errors are caught at compile-time rather than at link-time in contrast to the case with C++03.
- More user-friendly error messages.
- Any function (and not just member functions) can be given the `delete` status. This is useful when we wish to avoid implicit conversions.

We take an initial example of a class with deleted and defaulted functions:

```
// Explicitly defaulted and deleted functions
class F
{
public:
```

```
    // Support default operations OK
    F() = default;

    // Move ctor and move assignment are defined
    F(F&&) = default;
    F& operator = (F&&) = default;

    F(const F&) =  delete;
    F& operator = (const F&) = delete;

    virtual ~F() = default;
};
```

Some test code shows what does and does not compile (commented code does not compile):

```
// Explicitly defaulted and deleted functions
F f;

// Both give error message like 'F::F(const F &)': attempting to
// reference a deleted function
//F f2 = f;
//F f3(f);

F f4 = std::move(f);
```

Finally, we give an example of a deleted non-member function. In this case we define some overloaded print functions that are deleted when the input argument is a `double` or a `bool`. Hence they will not compile:

```
// Overloaded non-member function
void print(float f)
{
   std::cout << f << '\n';
}

void print(long n)
{
   std::cout << n << '\n';
}
// Deleted for these types

void print(bool b) = delete;
void print(double d) = delete;
```

An example of use (commented lines do not compile):

```
// Deleted overloaded non-member function
float f5 = 4.0F;
```

```
double d = 5.0;
long p = 3;
print(f5);
print(p);

// void print(bool)': attempting to reference a deleted function
// print(true);
// print(d);
```

7.3.3 The `constexpr` Keyword

From www.cppreference.com we have the description:

> *The `constexpr` specifier declares that it is possible to evaluate the value of a function or variable at compile time. Such variables and functions can then be used where only compile-time constant expressions are allowed (provided that appropriate function arguments are given). A `constexpr` specifier used in an object declaration implies `const`. A `constexpr` specifier used in a function declaration implies inline.*

We first discuss various kinds of constant objects. A simple example is:

```
// Macro, C++03 and C++11 solutions to const
#define MEANING_OF_LIFE 42
const int MeaningOfLifeI = 42;
constexpr int MeaningOfLifeII() { return 42; }

std::cout << MEANING_OF_LIFE << ", " << MeaningOfLifeI << ", "
          << MeaningOfLifeII() << '\n';
```

In general, `constexpr` objects are `const` and are initialised with values that are known at compile-time. More generally, their values are determined during translation which consists of both compilation and linking phases.

We now take a look at creating classes, some of whose member functions are `constexpr`. To this end, we wish to produce compile-time results whenever possible. A good example is a class to model two-dimensional points in CAD and computer graphics applications. Points are the lowest common denominator among two-dimensional shapes as it were and designing them for optimal performance is worth the effort. The class is defined as:

```
class Point
{
private:
   double x, y;

public:
   constexpr Point() noexcept : x(0.0), y(0.0) {}
   constexpr Point(double xVal, double yVal) noexcept
      : x(xVal), y(yVal) {}
```

```
    constexpr Point(const Point& pt2) noexcept
       : x(pt2.x), y(pt2.x) {}

    constexpr double X() const noexcept { return x; }
    constexpr double Y() const noexcept { return y; }

    void X(double xNew)  { x = xNew; }
    void Y(double yNew) noexcept { y = yNew; }

    // Not supported on VS2015
    //constexpr void X(double xNew) noexcept { x = xNew; }
    //constexpr void Y(double yNew) noexcept { y = yNew; }

    double Distance(const Point& pt2) const
    {
       return std::sqrt((x - pt2.x)*(x - pt2.x) + (y - pt2.y)*(y - pt2.y));
    }

    void print() const
    {

       std::cout << "(" << x << "," << y << ")\n";
    }
};
```

We can now create non-member functions, for example as follows:

```
constexpr Point MidPoint(const Point& p1, const Point& p2)
{
   return{ (p1.X() + p2.X()) / 2, (p1.Y() + p2.Y()) / 2 };
}
```

Furthermore, we can call these functions as follows:

```
double x = 1.0; double y = 1.0;

Point pt(x, y); Point pt2(pt);
Point pt3(pt.X() + 1.0, pt.Y() + 1.0);

auto midPoint = MidPoint(pt, pt3);
midPoint.print();

auto d = pt.Distance(pt3);
std::cout << d << "\n";
```

Concluding, we should strive to use `constexpr` objects and functions whenever possible, especially for cases in which values can be computed at compile-time.

7.3.4 The `override` and `final` Keywords

The `override` specifier avoids some of the pitfalls in C++98 when overriding a base class's pure and default `virtual` functions in the presence of implicit conversions. Furthermore, when inspecting a derived class it was not possible to see which functions are overrides of its base class functions. Second, the `final` specifier is used in a `virtual` function declaration and it ensures that the function is `virtual` and specifies that it may not be overridden by derived classes. In other words, it allows us to demand that a class is *sealed* (cannot be inherited from) or that a given member function in a given class cannot be overridden in a derived class. An example of a `final` class and an overridden function is:

```
struct Base
{
   virtual void draw() { std::cout << "print a base\n"; }
   void print() {}
   ~Base() { std::cout << "bye base\n"; }
};

struct Derived final : public Base
{
   Derived() {}
   void draw() override { std::cout << "print a derived\n"; }
   // void draw() const override {}
   // void print() override {}
   ~Derived() { std::cout << "bye derived\n"; }
};
```

Here we see how the member function `draw() const` is overridden, why the function must be `virtual` and why function signature is important. As a second example we consider the example of a sealed class:

```
// final specifier
class SealedClass final
{ // Derivation not possible


};

/* gives compiler error
class DerivedSeal : public SealedClass
{
};
*/
```

We now give an important example (from a design viewpoint, see GOF, 1995) of a `final` member function. We implement the *Template Method Pattern* in C++11 which is a behavioural design pattern that defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. It lets us redefine certain steps of an algorithm without changing the algorithm's overall structure. We take an example of a class called `DataFlow` that accepts a

numeric value from a customisable data source and applies a function to it that transforms it to another value:

```
using FunctionType = std::function<double (double x)>;

class DataFlow
{
private:
   double data;
   FunctionType _fun;

public:
   DataFlow(const FunctionType& fun) : data(0), _fun(fun) {}

   // Derived classes must override Input()
   virtual double Input() = 0;
   double Compute(double d) { return _fun(d); }

   // Template member function, cannot be overridden
   virtual double Algorithm() final
   { // This algorithm has a fixed structure

      // The steps in the algorithm
      double val = Input();
      return Compute(val);
   }
};
```

We now create a derived class that accepts its input from the console:

```
class ConsoleDataFlow : public DataFlow
{
public:
   ConsoleDataFlow(const FunctionType& fun) : DataFlow(fun) {}

   double Input() override
   {
      std::cout << "Give input value: ";
      double val; std::cin >> val;
      return val;
   }
};
```

An example of use is:

```
// Design pattern
auto fun = [](double d)->double {return std::exp(d);};
auto fun2 = [](double d)->double {return d*d;};

ConsoleDataFlow cdf(fun);
std::cout << cdf.Algorithm() << std::endl;
```

```
ConsoleDataFlow cdf2(fun2);
std::cout << cdf2.Algorithm() << std::endl;
```

Summarising, the member function `Algorithm()` cannot be overridden but its variant parts (for example, `Input()` and `Compute()`) can be overridden. Finally, we note that the above design combines subtype polymorphism with uniform function wrappers. In this sense the solution represents a mixture of object-oriented and functional programming styles.

7.3.5 Uniform Initialisation

In previous versions of C++ there was some confusion concerning how to initialise variables and objects. It was not always clear when to use braces, parentheses or assignment operators. In C++11 we can avail of a single common syntax called *uniform (braced) initialisation* by placing the values between braces. Some simple examples are:

```
// Initializer list (they are recursive)
print<int>({ 3, 4, 5, 6, 99 });

// Uniform initialisation
double arr[] {1.0, 2.0, 3.0};
std::list<int> arr2{ 1, 3, 5, 7 };

std::tuple<double, double> tup(1.0, 2.0);
std::map<int, std::tuple<double, double>> database
                        { { 1, tup }, { 2, tup } };

for (auto it=std::begin(database);it != std::end(database); ++it)
{
   std::cout << it->first << ": "
             << std::get<0>(it->second) << ", " << std::get<1>
             (it->second) << std::endl;
}
```

We can use this technique to initialise the members of an aggregate object. Consider the classes:

```
// Struct that holds data together
struct Person
{

   std::string name_;
   std::string address_;
   int age_;

};

struct Person2
```

```
{
   std::string name_;
   std::string address_;
   int age_;

   Person2(int age, const std::string name, const std::string address)
      : age_(age), name_(name), address_(address) {}
};
```

We use uniform initialisation as follows:

```
// Aggregate initialisation; initialise the data in a
// struct without needing a constructor
Person p1{ "Joe", "Denver", 23 };

// ERROR! Person p1_a( "Joe", "Denver", 23 );
// (No function talking three arguments!!)

// Using constructors to initialise data
Person2 p2{ 55, "Piet", "Zurich" };
Person2 p3( 55, "Piet", "Zurich" );
```

We see from class `Person` that we can initialise a class's member data using braced initialisation even though the class does not have the appropriate constructors.

7.3.6 Initialiser Lists

An object of type `std::initializer_list<T>` is a lightweight proxy object that provides access to an array of objects of type `const T`:

```
// Initialiser lists
template <typename T>
   class ILClass
{ // Class using initialiser lists
private:
   std::vector<T> v;
public:
   ILClass(std::initializer_list<T> myList) : v(myList) {}

   void append(std::initializer_list<T> myList)
   { // Extend vector v before element at specified position

      v.insert(std::end(v), std::begin(myList),
                  std::end(myList));
   }

   void print() const
   {
```

```
        std::cout << '\n';
        for (auto elem : v) { std::cout << elem << ","; }
        std::cout << '\n';
    }
};
```

An example of use and corresponding output is:

```
// Initializer lists
ILClass<int> c = { 1,2,3 };
c.print();                              // 1,2,3
c.append({ 4,5,6 });
c.print();                              // 1,2,3,4,5,6
```

7.3.7 Keyword `noexcept`

This keyword specifies that a function cannot throw or is not prepared to throw an exception. If a function does throw an exception then the program will terminate by calling `std::terminate()` which by default calls `std::abort()`.

In general, the C++03 exception handling mechanism had a number of issues (that we do not discuss here) with the end result that many programmers just avoided using it. In C++11, however, the situation is much more clear-cut: a function throws an exception or it does not throw an exception. The keyword `noexcept` is part of a function and callers may depend on it.

We say that a function is *exception-neutral* if it does not throw an exception itself but functions that it calls might emit an exception. If an exception is emitted then the exception-neutral function allows the emitted exception to pass through the *calling chain* until a handler has been found. Such functions are never `noexcept`.

7.4 SHADES OF POLYMORPHISM

Polymorphism is defined as the ability to take several forms. In object-oriented technology we achieve polymorphic behaviour by being able to attach behaviour to different objects at run-time and calling their member functions. In short, we support polymorphism by declaring *pure* or *default* `virtual` member functions in a base class and then implementing these functions in derived classes. We also call this *subtype* (or *dynamic*) *polymorphism*.

We discuss four different ways to achieve polymorphic behaviour in classes and class hierarchies. We take the popular example of modelling payoff functions. We restrict the scope to one-factor payoffs and we consider just two cases, namely call and bull spread payoff. The goal is that clients should not have to know directly about specific payoffs or their defining data. This is a form of *information hiding*. This makes client code more extendible and reusable.

We propose the following solutions:

- S1: Traditional C++ hierarchy using subtype polymorphism.
- S2: Using the *Strategy* design pattern (GOF, 1995) with shared pointers. This is dynamic polymorphism.

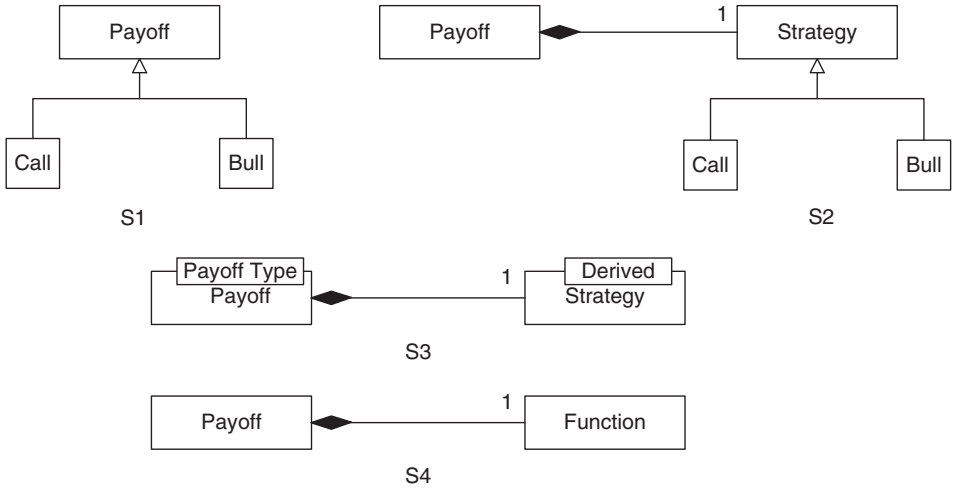


FIGURE 7.2 Modelling classes and class hierarchies

- S3: Using the *Strategy* design pattern in combination with the CRTP. This is static polymorphism.
- S4: Eliminate the class hierarchies in S1, S2 and S3 altogether and instead use a single class with an (abstract) embedded function pointer that can be customised by clients.

At this stage we depict each of these options by a UML class diagram as shown in Figure 7.2 (the triangle symbols depict inheritance between the respective classes). In all cases we prefer to use shared pointers instead of raw pointers.

Case S1 is implemented by the following code (we have not used the `override` specifier):

```
// Traditional class hierarchy
class Payoff
{
public:
   virtual double operator ()(double S) const = 0;
};


class CallPayoff : public Payoff
{
private:
   double K;
public:
   CallPayoff(double strike) { K = strike;}
   double  operator ()(double S) const
   {
      return std::max<double>(S - K, 0);
   }
};
```

```
class BullSpreadPayoff : public Payoff
{
private:
   double K1;
   double K2;
public:
   BullSpreadPayoff(double strike1, double strike2)
   { K1 = strike1; K2 = strike2;}

   double operator () (double S) const
   {

      if (S >= K2)
         return K2 - K1;
      if (S <= K1)
         return 0.0;

      // In the interval [K1, K2]
      return S - K1;
   }
};
```

A test case is:

```
template <typename T>
using SP = std::shared_ptr<T>;

double K = 20.0;
SP<Payoff> call (new CallPayoff (K));

std::cout << "Give a stock price (plain Call): ";
double S;
std::cin >> S;

std::cout << "Call Payoff is: " << (*call)(S) << '\n';


double K1 = 30.0;           // Strike price of bought call
double K2 = 35.0;           // Strike price of sell call

SP<Payoff> bs (new BullSpreadPayoff(K1, K2));     // Hull example

std::cout << "Give a stock price (BullSpread): ";
std::cin >> S;

std::cout << "Bull Spread Payoff is: " << (*bs)(S) << std::endl;
```

This is a popular approach and no doubt many legacy C++ applications (possibly in combination with raw pointers) use it to achieve polymorphism. The main disadvantages of this approach are:

- D1: Once a payoff has been selected it is not possible to switch to another one without creating a new object. In this case we say that there is a *permanent binding* between the parts of the code.
- D2: A potential explosion in the number of payoff classes that must be created, instantiated and loaded into an application. For example, this occurs with the *Command* design pattern (GOF, 1995) in which tens and even hundreds of *tiny classes* are created in order to satisfy application requirements. In general, these objects are usually retained in memory for the duration of the program.
- D3: Possible performance penalty due to the presence of *virtual* functions. There is evidence to show that the run-time efficiency of calling these functions can be seven to ten times slower than that of non-virtual functions. You should investigate this for your own applications.

We now turn our attention to solution S2 (for space reasons we only show the code for call payoffs as the other cases are similar). Using the *Strategy* design pattern alleviates problem D1 somewhat because we can now decouple the `Payoff` class from its various implementations and this allows us to switch from one kind of payoff to another one at run-time:

```
template <typename T>
   using SP = std::shared_ptr<T>;

#include "PayoffStrategy.hpp"

class Payoff
{
private:
   // PayoffStrategy* ps; // OLD
   SP<PayoffStrategy> ps;
public:
   // Constructors and destructor
   explicit Payoff(const SP<PayoffStrategy>& pstrat);
   Payoff(const Payoff& source) = delete;
   virtual ~Payoff();

   // Operator overloading
   Payoff& operator = (const Payoff& source) = delete;

   // Not a pure virtual payoff function anymore
   virtual double payoff(double S) const; // For spot price
};
```

We see that the function `payoff()` is no longer *pure* `virtual` but instead it delegates to its embedded strategy:

```
// Not a pure virtual payoff function anymore
double Payoff::payoff(double S) const
{ // Call function in strategy object for a given spot price

      return ps->payoff(S);        // DELEGATE to strategy
}
```

The algorithms now form their own independent class hierarchy:

```
class PayoffStrategy
{
public:
   virtual double payoff(double S) const = 0;
};


class CallStrategy : public PayoffStrategy
{
private:
   double K;
public:
   CallStrategy(double strike) { K = strike;}
   double payoff(double S) const
   {
      return std::max<double>(S - K, 0);
   }
};
```

An example of use is:

```
double K = 20.0;
SP<PayoffStrategy> call (new CallStrategy (K));
Payoff pay1(call);

std::cout << "Give a stock price (plain Call): ";
double S;
std::cin >> S;

std::cout << "Call Payoff is: " << pay1.payoff(S) << '\n';
```

We now apply CRTP to create a compile-time version of solution S2. In this new solution S3 we create template classes as shown in Figure 7.2:

```
template <typename PayoffType>
   class Payoff
{
private:
   // PayoffStrategy* ps; // OLD
   SP<PayoffStrategy<PayoffType>> ps;

public:
   // Constructors and destructor
   explicit Payoff(const SP<PayoffStrategy<PayoffType>>& p);
   Payoff(const Payoff& source) = delete;
   virtual ~Payoff() {}
```

```
    // Operator overloading
    Payoff& operator = (const Payoff& source) = delete;

    // Not a pure virtual payoff function anymore
    double payoff(double S) const; // For a given spot price
};
```

The implementations form a class hierarchy and we see a direct application of CRTP:

```
template <typename DerivedClass>
    class PayoffStrategy
{
public:
    //virtual double payoff(double S) const = 0;
    double payoff(double S)
    {
        return static_cast<DerivedClass*>(this) -> payoff(S);
    }
};

class CallStrategy : public PayoffStrategy<CallStrategy>
{
private:
    double K;
public:
    CallStrategy(double strike) { K = strike;}
    double payoff(double S) const
    {
        return std::max<double>(S - K, 0);
    }
};
```

An example of use is:

```
double K = 20.0;
SP<PayoffStrategy<CallStrategy>> call (new CallStrategy (K));
double S = 30.0;
std::cout << "Call payoff: " << call->payoff(30.0) << '\n';
```

Finally, we discuss solution S4 which can be seen as a hybrid object-oriented/functional solution. Instead of a class hierarchy we create a single class with an embedded (not yet assigned to a target function) function that can be assigned to a range of target function types in client code, as we have seen in Chapter 3:

```
using Function = std::function<double (double)>;

// #include "PayoffStrategy.hpp"

class Payoff
```

```
{
private:
     // PayoffStrategy* ps;    // OLD
     Function ps;

public:
     explicit Payoff(const Function& pstrat);

     // Code same as solution S3

     // ...

};
```

This means that `Payoff` can be instantiated by any function or callable object having the appropriate signature. We give some code examples to show how flexible this solution is:

```
using namespace std::placeholders;

double K = 20.0;
CallStrategy call(K);
Function payoffCall = std::bind(&CallStrategy::payoff, call, _1);
Payoff pay1(payoffCall);

std::cout << "Give a stock price (plain Call): ";
double S;
std::cin >> S;

//std::cout << "Call Payoff is: " << pay1.payoff(S) << '\n';
std::cout << "Call Payoff is: " << payoffCall(S) << '\n';

double K1 = 30.0;           // Strike price of bought call
double K2 = 35.0;           // Strike price of sell call

BullSpreadStrategy bull(K1, K2);     // Hull example
auto payoffBull=std::bind(&BullSpreadStrategy::payoff, bull, _1);
Payoff pay2(payoffBull);

std::cout << "Give a stock price (BullSpread): ";
std::cin >> S;

std::cout << "Bull spread Payoff: " << payoffCall(S) << '\n';

// Create a payoff on the fly, captured variables
double p = 2.0;
double C = 50.0;
double strike = 50.0;
```

```
auto payoff3 = [=](double S)
{
   return std::min<double>(std::max<double>(std::pow(S - strike, p),
   0.0), C);
};

std::cout << "Symmetric capped power call: " << payoff3(S);
Payoff pay3(payoff3);
std::cout << "Symmetric capped power call: " << pay3.payoff(S);

auto payoff4 = [=](double S)
{
   return std::min<double> (std::max<double>(std::pow(S, p), 0.0), C);
};

std::cout << "Asymmetric capped power call: " << payoff4(S);
Payoff pay4(payoff4);
std::cout << "Asymmetric capped power call: " << pay4.payoff(S);
```

We see that it is possible to create payoff functions at run-time ('on the fly') and it is possible to reuse existing code by use of `std::bind`. This code is an example of how to configure code with the help of lambda functions and this style will be used in later chapters when we apply the *Builder* design pattern to configure applications.

7.5 IS THERE MORE TO LIFE THAN INHERITANCE?

Looking back, developers probably went overboard with the creation of deep and complex class hierarchies in the 1990s. This was conventional wisdom at the time. In fairness, the author's CAD and computer graphics applications avoided this pitfall. We used no multiple inheritance, the class hierarchies were at most one deep and many of the classes were created using *composition*. The code was reasonably maintainable and well documented and knowledge of the problem domain was also advantageous.

In our experience subtype polymorphism is neither necessary nor sufficient for successful software projects. On the other hand, class hierarchies do have their uses. But we do need to be careful:

- Creating classes that are essentially objects or instances; this manifests itself in the explosion of *tiny* classes with little extra behaviour when compared to the base class from which they are derived.
- The relationship between the classes is not an ISA (Gen/Spec) relationship but something else. This is a common error and it can have serious consequences for the integrity of a software system.
- Confusing HASA and ISA relationships.
- Developers tend to use data and structure rather than behaviour to determine if a relationship is a Gen/Spec one, the iconic example being deriving a `Square` from a `Rectangle`. Another example is to create two classes that model call and put options both of which are derived from a common base class.

7.6 AN INTRODUCTION TO OBJECT-ORIENTED SOFTWARE METRICS

We conclude this chapter with a reminder to do an inspection of our code to determine its software quality. In particular, we focus on several properties. We measure them by computing a numerical value from a collection of data. We define *software metrics* that deal with the measurement of a software product or a process. The goal is to obtain objective, reproducible and quantifiable measurements that give an indication of the quality of object-oriented code.

We reduce the scope of the discussion by focusing on the quality of classes, their members and the amount of coupling between classes, including guidelines on what constitutes a good class hierarchy or object-oriented design.

The following discussion is meant to make things more explicit and to pinpoint some common problems. For more details, see Lorenz and Kidd (1994).

In general, large and small values for the metrics might be suspect.

7.6.1 Class Size

This metric can be measured in a number of ways, for example:

- M1: Number of public member functions in a class.
- M2: Total number of member functions (public and private) in a class.
- M3: Average number of member functions per class.
- M4: Number of data members in a class.

For each of these metrics we associate a numerical value. Although highly context sensitive, we should strive to define upper and lower threshold bounds for the value of each metric. For example, for metric M1 a rule of thumb is to have the number of public member functions in the closed range [4,20]. Smaller and larger values are suspect and could be a sign of a bad design. On the one hand, a class may be a tiny class in which case we doubt whether it is a real class in the first place and a class with more than 20 member functions may be violating the SRP. Some early-warning signals concerning the other metrics are:

- M2: Classes with a large number of private member functions might suggest that the class is reinventing the wheel (for example, as a *nested class*) that has already been written somewhere else, possibly in a C++ library. Does the class satisfy SRP?
- M4: This might indicate that we have designed an incorrect data structure.
- M3: A system whose classes contain many or few member functions might indicate a bad design. It is clear that a class with many member functions will be more difficult to maintain than a class with a smaller number of member functions.

In general, we try to reuse as much functionality as possible from STL (containers, algorithms), Boost and other libraries rather than developing our own code. Hopefully this approach will reduce the amount of code to write and maintain. The metric being measured in this section is closely related to SRP.

7.6.2 Class Internals

In computer programming the term *cohesion* (De Marco, 1978) refers to the degree to which the elements of a module (or class) belong together. It describes the strength of the relationship

between pieces of functionality within a given module. For classes, *cohesion metrics* measure how well the member functions of a class are related to each other. In general, a cohesive class performs one function. A non-cohesive class performs two or more unrelated functions. Cohesion is a *qualitative measure* (we can speak of *high cohesion* and *low cohesion*, for example) and we can identify different forms of cohesion (McConnell, 2004, pp. 168–171), from worst to best. McConnell describes seven types but we restrict our attention to the three best ones. In the following the term module can refer to a class:

- *Communication/informational cohesion*: this occurs when parts of a module are grouped because they operate on the same data.
- *Sequential cohesion*: this occurs when parts of a module are grouped because the output from one part is the input to another part as in an assembly line.
- *Functional cohesion*: this occurs when parts of a module are grouped because they contribute to a single well-defined task of the module. This measure has a high correlation with SRP.

Functional cohesion is most desirable but it may not always be achievable. In general, modules with high cohesion have higher reliability and have fewer faults than low-cohesion modules.

We now give some low-level (but important) metrics related to class internals from Lorenz and Kidd (1994):

- M5: How many global and static variables used in a class.
- M6: Average number of arguments per member function.
- M7: Number of friend functions per class.
- M8: Average number of comment lines per member function.
- M9: Average number of commented methods.

How many applications have the value zero for the metrics M8 and M9?

7.6.3 Class Coupling

This metric examines how classes relate to other classes, to subsystems and to other code. In particular, we are interested in the *dependencies* between classes. The more dependencies there are the more difficult it is to understand and maintain the code.

Classes fulfil their responsibilities by collaborating with other classes. A *collaboration* represents a request from a client to a server in fulfilment of a *client responsibility* (Wirfs-Brock, Wilkerson and Wiener, 1990). For example, in a Monte Carlo simulation a client class that models a finite difference scheme collaborates with a server class that delivers random numbers. Identifying *collaborations* in the early stages of class design is important for a number of reasons:

- Information and control flow in an application is revealed.
- We are able to design an application as a graph of collaborating classes. This helps in distributing responsibilities among the collaborators.
- We can discover missing responsibilities by identifying a collaboration.
- We can identify a collaboration without an associated responsibility.

In order to identify collaborations we ask the following questions for each responsibility of each class:

- Is the class capable of fulfilling a given responsibility by itself?
- If not, what does the class need?
- From which other classes can the class acquire what it needs?

Knowing what the inter-class relationships are will be useful in identifying collaborations. The major relationships are:

- R1: *Composition* ('is part of'). These are also called *Whole–Part* and aggregation relationships (see POSA, 1996).
- R2: *Has knowledge of*. This relationship is a more loosely coupled version of composition. In general, class *X* has knowledge of class *Y* if *Y* can provide information to *X*.
- R3: *Depends on*. This is similar to relationship R2 but the classes forming the relationship do not necessarily have to know about each other. A third-party *mediator* may know about the dependency, for example.

Having described what coupling is we now define the corresponding metrics:

- M10: The number of other classes that a given class collaborates with.
- M11: The amount of collaboration with other classes.

The metrics M10 and M11 will take on another form when we discuss software development based on system decomposition and interface specifications in Chapter 9. At that stage object-oriented technology fades into the background somewhat.

7.6.4 Class and Member Function Inheritance

This group of metrics revolves around the quality when we use inheritance:

- M12: Class hierarchy nesting level. This refers to the depth of the inheritance tree.
- M13: Number of abstract classes.
- M14: Use of multiple inheritance.
- M15: Number of member functions overridden by a derived class.
- M16: Number of member functions inherited by a derived class.
- M17: Number of member functions added by a derived class.

In general (as we discussed in Section 7.5) inheritance is a mixed blessing and we tend to avoid using it too much for the reasons alluded to in that section. We also have a number of alternative designs to reduce the numerical values corresponding to the metrics M12 to M17. It is clear that the higher the numerical value, the more difficult it is to maintain the class hierarchy and the less likely that it retains its *semantic integrity*. Some ways to reduce the value are:

1. Use static polymorphism with CRTP instead of subtype/dynamic polymorphism.
2. We could pretend that we live in an inheritance-free world and use composition throughout. But this approach is probably not realistic.

3. Create a single class C that contains one or more universal function wrappers. Then class C has constructors that accept functions as arguments. These functions replace `virtual` member functions and obviate the need for a class hierarchy to a certain extent.

We shall see more examples of these techniques as we progress in this book. In particular, they can be used to replace and upgrade the object-oriented design patterns in GOF (1995). Some of the concerns in Section 7.6 become non-issues in the new paradigm of system decomposition that we introduce in Chapter 9.

7.7 SUMMARY AND CONCLUSIONS

In this chapter we have discussed how to design classes using well-established modelling techniques and C++ language features. We design classes that are self-contained, robust and have loose coupling with other classes. This approach will hopefully be seen as an improvement on how classes are designed using class hierarchies and the approach will be generalised to the design of large systems in later chapters.

A special feature of this chapter is that we use the functional programming style (in particular `std::function`) to replace the `virtual` function by universal function wrappers. In particular, this allows us to reduce the potential explosion in the number of classes that need to be created in an application. One caveat is that we must pay attention to issues of efficiency.

7.8 EXERCISES AND PROJECTS

1. (Explicit Constructors)
   Consider the following class:

```
struct B
{
   explicit B(int) {}
   explicit B(int, int) {}
   explicit operator int() const { return 0; }
};
```

   Create a program to test the class. Which lines of code below compile and which ones do not? How would you modify the class to ensure that they do compile? Otherwise, how can these lines be modified so that the code compiles error free without modifying the class?

```
B b1 = 1;
B b2(3);
B b3{ 7,3 };
B b4 = (B)42;
```

2. Which of the following statements are advantages of using *deleted functions* compared to declaring these functions as `private` as in the C++98 standard?
   a) Calling a deleted function will cause a compile-time error rather than a link-error with C++98.
   b) Use of a deleted function in an unevaluated expression leads to a link-error.

c) Friends accessing a private function with missing definition experience a compile-time error.
d) Making deleted functions `public` results in better error messages in general.

3. In this exercise we investigate the copy constructor and copy assignment operator as default and deleted forms (there are four choices of `default`/`delete` for these two functions). One choice is:

```
// Copy ctor and copy assignment operator
F(const F&) =  delete;
F& operator = (const F&) = delete;
```

Test this and the *three other options* and determine which ones compile and which ones do not.

4. Which of the following statements are true?
a) `constexpr` applies to both objects and functions.
b) All `const` objects are also `constexpr` objects.
c) The output from a `constexpr` function will be known at compile-time if its input arguments are known at compile-time.
d) `constexpr` functions need not necessarily produce values that are known at compile-time.

5. Explain why the following member function of class `Point` does not compile:

```
constexpr double Distance(const Point& pt2) const
{
   return std::sqrt((x - pt2.x)*(x - pt2.x) + (y - pt2.y)*(y - pt2.y));
}
```

Furthermore, does the following code compile?

```
constexpr Point p2(1.0, 2.0);
double newVal = 3.0;
p2.X(newVal);

Point p3(1.0, 2.0);
double newVal2 = 3.0;
p3.X(newVal2);
```

6. Which two (most important) symptoms in your opinion can manifest themselves if a class does not satisfy SRP?
a) The number of methods in a class increases (threshold value $\sim$ 12).
b) Class hierarchy nesting level (depth of inheritance hierarchy) (threshold value $\sim$ 6).
c) Number of data members in a class (threshold value $\sim$ 3).
d) Methods with many lines of code (threshold value $\sim$ [10,20]).

7. What is the *Fragile Base Class Problem*?
a) A base class which when modified can cause derived classes to malfunction.
b) A base class containing many abstract methods that must be overridden in derived classes.

c) A class that is derived from two base classes.
d) A base class with too many protected methods.

8. How can the *Fragile Base Class Problem* be resolved?
a) Using interfaces (if they are supported) instead of base classes.
b) Using more composition and less inheritance.
c) Making some base class methods `final` in order to preserve integrity.
d) Creating base classes with only abstract methods and no data.

9. Give the top three risks associated with deep inheritance hierarchies:
a) They become increasingly difficult to maintain.
b) There is a probability that the inherited functionality is semantically incorrect.
c) Many developers create class hierarchies based on structure and not on behaviour.
d) Performance degradation as the depth of the hierarchy increases.

10. Which choice for modelling functions results in the most flexible software?
a) Dynamic polymorphism.
b) Static polymorphism (CRTP pattern).
c) Universal function wrapper.
d) Free functions (C function).

11. How can you identify code (in general terms) that is not using *Whole–Part*?
a) The participating objects tend to fail SRP, in other words weak separation of concerns.
b) Classes tend to have relatively many member functions.
c) Classes tend to have relatively many data members.
d) There tends to be no single mediating object in the object network.

12. (Designing a Monte Carlo Framework)

In Section 7.2.3 we produced a context diagram that shows the modules/subsystems that participate in a Monte Carlo option pricer that we shall discuss in more detail in Chapter 31. The objective of the current exercise is to answer some questions relating to the topics of Chapter 7. In a sense you have to know what the different modules do and how they cooperate to satisfy the goals of the system.

Answer the following questions:

a) Read the description of each module and determine what its responsibilities are. Does each module satisfy SRP?
b) Describe in words or as a *data flow diagram* (DFD) the process of computing the option price based on certain inputs such as the type of options whose prices should be computed, market data, model data and so on.
c) Based on the answers in parts a) and b) determine which steps/activities have no corresponding modules and which modules have less than or more than one major responsibility. Determine what to do if there is a discrepancy, for example designing new modules and documenting them in the context diagram.
d) For each module determine the services that it *provides to* and the services that it *requires from* other modules.
e) Create a small software prototype in C++ to implement the context diagram in Figure 7.1. The requirements are:
1. Support for call and put payoffs. Choose one of the solutions from Section 7.4 that implement C++ payoff classes.
2. Support for plain options and arithmetic average-rate Asian options.
3. Support for GBM and CEV stochastic differential equations.
4. Euler and Milstein finite difference schemes to approximate the SDEs in part 3.

5. Use the C++ library `<random>` for random number generation.
6. Design a flexible *Mediator* (see GOF, 1995) class that connects the different modules.

Determine which programming style you will use to implement each module. We implement this problem in Chapter 31.

13. (Class Hierarchies, Brainstorming)

Consider the following class hierarchies and Gen/Spec relationships. Which ones are semantically correct, which ones are semantically incorrect and can you model them without the need for class hierarchies (general answers are sufficient)?

a) Base class `Lattice`, derived classes `BinomialLattice` and `TrinomialLattice`.
b) `CallOption` and `PutOption` as derived classes of `Option`.
c) A `Square` class that is derived from a `Rectangle`.
d) PDE class hierarchies. How would you design them?
e) The wisdom of inheriting from template classes.

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 8

C++ Numerics, IEEE 754 and Boost C++ Multiprecision

8.1 INTRODUCTION AND OBJECTIVES

In this chapter we introduce a number of concepts, methods and C++ language features to promote the robustness and reliability of code. Specifically, *robustness* is the ability of a computer system to cope with errors during execution and cope with erroneous input while *reliability* is concerned with how a software system maintains a given level of performance over some given period of time. We discuss a number of techniques to promote *robust programming.* This can be defined as 'a style of programming that focuses on handling unexpected termination and unexpected actions. It requires code to handle these terminations and actions gracefully by displaying accurate and unambiguous error messages. These error messages allow the user to more easily debug the program'. Reliability describes avoiding abnormal program termination. It has three sub-characteristics:

- *Maturity*: has to do with the frequency of failure in the system. Most failures are caused by *faults*.
- *Fault tolerance*: refers to the ability of the system to maintain a specified level of performance. We must specify the duration of time for which that level is to be maintained.
- *Recoverability*: refers to the capability to re-establish previous levels of performance.

These are major concerns when writing C++ code that implements the numerical methods and algorithms in computational finance. In particular, we discuss a number of methods to promote code robustness and reliability with emphasis on integral and floating-point data types. We pay particular attention to a number of scenarios:

- S1: Rounding rules when using operations on arithmetic types.
- S2: Normal, subnormal and infinite numbers. NaN (not-a-number).
- S3: Exception handling due to division by zero, overflow, underflow and invalid operations.
- S4: What is machine precision?
- S5: Rounding and cancellation errors.

We discuss these scenarios and how they are supported in C++. Many errors are caused by the fact that numbers have a fixed word length and information is lost when the data does not fit into this range. In Appendix 1 we introduce multiple precision data types that are able to hold a wider range of values and digits than is possible using standard C++ data types.

IEEE 754 is a technical standard for floating-point computation. Many hardware floating-point units now support IEEE 754. We give an overview of the standard before discussing related topics in C++. The standard addresses the following topics:

- *Arithmetic formats*: support for *binary* (base 2) and *decimal* (base 10) floating-point data consisting of finite numbers.
- Special cases of finite numbers are *signed zeros*, *subnormal numbers*, *infinities* and special 'not-a-numbers' (*NaNs*).
- *Interchange formats*: encodings (bit strings) that can be used to exchange floating-point data in an efficient and compact form.
- *Operations*: arithmetic (for example, addition, multiplication and trigonometric functions) and other operations on arithmetic formats.
- *Rounding rules*: these are properties to be satisfied when rounding numbers during arithmetic operations and conversions.
- *Exception handling*: indications of exceptional conditions, for example *division by zero*, *overflow*, *underflow* and *inexact results* (due to rounding).

We discuss these topics in the coming sections. They are of particular importance in applications of numerical analysis. We also try to realise their requirements in C++.

8.1.1 Formats

A *format* is a set of representations of numerical values and symbols. It can be a finite number in either base 2 or base 10 in single or double-precision format. The numerical value of a finite number is $(-1)^s \times c \times b^q$ where:

b = base (2 or 10), also called the radix
s = sign (0 or 1) (0 for positive, 1 for negative)
c = significand (and number of digits == p)
q = exponent (upper limit emax) parameter.

An example is:

$$c = 12345,\ b = 10,\ q = -3,\ s = +1 \text{ value} = 12.345.$$

There are constraints in the values of the significand and the exponent parameter:

$c \in [0, b^p - 1]$
q is such that $1 - emax \leq q + p - 1$.

We thus see that some numbers have several representations but for most arithmetic operations the result (value) does not depend on the representation of the inputs.

A format also comprises positive and negative infinities $+\infty$ and $-\infty$, respectively. Finally, IEEE 754 describes two kinds of NaNs. A *quiet NaN* (qNaN) does not raise any additional exceptions as it propagates through a chain of operations. A *signalling NaN* (sNaN) is a special

form of NaN that raises an exception when consumed in an operation. These could be used in various cases:

- Filling uninitialised memory with a sNaN would produce an invalid operation exception of the data that is used before it is initialised.
- A representation of a number that has underflowed or overflowed.
- A number in a higher-precision format.
- A complex number (a number with real and imaginary parts).

Operations that generate NaNs are:

- Operations in which at least one operand is a NaN.
- Multiplications $0 \times \pm\infty,\ \pm\infty \times 0$.
- Additions $\infty + (-\infty),\ (-\infty) + \infty$ and equivalent subtractions.
- Real operations with complex arguments, for example the square root or logarithm of a negative number. Another example is the inverse sine or cosine whose argument is not in the open interval $(-1,1)$.

8.1.2 Rounding Rules

Rounding a numerical value entails replacing the value by a value that is approximately equal to it. The approximate value is usually shorter or simpler than the original value or it may have a more explicit representation. There are many ways to round values in IEEE 754; we reduce the scope by examining five rounding rules. We first discuss rounding to the *nearest value*:

- Round to nearest, ties to even: round to the nearest value.
- Round to nearest, ties away from even: round to the nearest value.

An example of how C++ supports these features is:

```
void CelingFloorTruncRound(double d)
{
   std::cout << "*Value: " << d << '\n';
   std::cout << "\tCeiling: " << std::ceil(d) << '\n';
   std::cout << "\tFloor: " << std::floor(d) << '\n';
   std::cout << "\tTruncate: " << std::trunc(d) << '\n';
   std::cout << "\tRound: " << std::round(d) << '\n';
   std::cout << "\tNearest int using current rounding mode: "
             << std::round(d);
   std::cout << '\n';
}
```

Directed roundings are:

- Round to 0: this is directed rounding towards zero. This is known as *truncation*.
- Round to $+\infty$: directed rounding towards positive infinity. This is known as *rounding up* or *ceiling*.

- Round to $-\infty$: directed rounding towards negative infinity. This is known as *rounding down* or *floor*.

We now describe some of the features in C++ to support these concepts.

- `FE_DOWNWARD`: rounding towards negative infinity.
- `FE_TONEAREST`: rounding towards nearest representable value.
- `FE_TOWARDZERO`: rounding towards zero.
- `FE_UPWARD`: rounding towards positive infinity.

An example of use is:

```
void RoundingInAllDirections(float x)
{
   std::cout << std::setprecision(50);
   std::cout << "*Value: " << x << '\n';

#pragma STDC FENV_ACCESS ON
   // Set rounding directions
   std::fesetround(FE_DOWNWARD);
   // Get nearest integer based on current mode
   std::cout << "\tDownward: " << x << '\n';

   std::fesetround(FE_UPWARD);
   std::cout << "\tUpward          : " << x << '\n';

   std::fesetround(FE_TONEAREST);
   std::cout << "\tNearest         : " << x << '\n';

   std::fesetround(FE_TOWARDZERO);
   std::cout << "\tToward zero     : " << x << '\n';
}
```

8.1.3 Exception Handling

The standard defines a number of exceptions. Each one returns a default value and it has a corresponding status flag that is raised when the exception is raised:

- Invalid operation, for example the square root of a negative number. This returns a qNaN by default.
- Division by zero. Returns plus or minus infinity by default.
- Overflow (result too large to be represented correctly). Returns plus or minus infinity by default.
- Underflow (a result is very small). Returns a denormalised value infinity by default. A denormal (also called *subnormal*) number is one where the usual representation in terms of significand and exponent would result in an exponent that is below the minimum allowed exponent.

8.1.4 Extended and Extendible Precision Formats

The standard specifies extended and extendible precision formats that are recommended when greater precision is needed. An *extended precision format* extends a basic format by using more precision and a larger exponent range. For example, *quadruple* (*quad*) *precision* is a binary floating-point-based computer number format that occupies 16 bytes (128 bits). Such precision is needed in applications that require results in higher than double precision and to allow more accurate and reliable computation by minimising overflow and round-off errors. IEEE 754 specifies *binary128* as having the following elements:

- Sign bit (one bit).
- An exponent width of 15 bits.
- A significand precision of 113 bits.

There are a number of software libraries that support quadruple precision but direct hardware support is less common. In this chapter we restrict our attention to the *Boost Multiprecision* library. It is introduced in the online documentation as follows:

> Multiprecision *consists of a generic interface to the mathematics of large numbers as well as a selection of big number back ends, with support for integer, rational and floating-point types.* Boost.Multiprecision *provides a selection of back ends including interfaces to GMP, MPFR, MPIR, TomMath as well as its own collection of Boost-licensed, header-only back ends for integers, rationals and floats. In addition, user-defined back ends can be created and used with the interface of* Multiprecision*, provided the class implementation adheres to the necessary concepts.*

The *Multiprecision* library provides integer, rational and floating-point types in C++ that have more range and precision than C++'s ordinary built-in types. The *big number* types can be used with a wide selection of basic mathematical operations, elementary transcendental functions as well as the functions in *Boost.Math*. The *Multiprecision* types can also interoperate with the built-in types in C++ using clearly defined conversion rules. This allows *Boost.Multiprecision* to be used for all kinds of mathematical calculations involving integer, rational and floating-point types requiring extended range and precision.

We introduce the *Boost.Multiprecision* library in Appendix 1.

8.2 FLOATING-POINT DECOMPOSITION FUNCTIONS IN C++

- D1: Decompose a floating-point value into a normalised fraction and an integral power of two.
- D2: Multiply a floating-point value by the number 2 raised to an exponential power.
- D3: Decompose a floating-point value into a normalised fraction and an integral power of two.
- D4: Extract the value of the unbiased radix-independent exponent from a floating-point argument and return as a signed floating-point value or as a signed integer value.
- D5: Return the next representable value of a floating-point value in a given direction.

Function D2 is the dual of function D1. Both can be used to manipulate the representation of a floating-point number without direct bit manipulations. Some code to show the use of these functions is:

```
template <typename T>
   void FloatPointManipulation(const T& t)
{
   // Decompose t into normalised fraction + integral power of 2
   int i;
   T t2 = std::frexp(t, &i);
   std::cout << "Fraction, power of 2: " << t2 << "*2^" << i << '\n';

   // Multiply a floating-point value by 2 raised to an exponential power
   // "Load exponent"
   int n = -4;
   std::cout << "Load exponent x * 2^exp: " << std::ldexp(t, n) << '\n';

   // Decompose t into integral and fractional parts, each having the same
   // type and sign as x;
   T iptr;
   T t3 = std::modf(t, &iptr);
   std::cout << "Decompose into integral and fractional parts: "
             << iptr << "," << t3 << '\n';

   // Extract the value of the unbiased radix-independent exponent
   // from a floating-point argument.
   std::cout << "Extracted exponent (float) : " << std::logb(t) << '\n';
   std::cout << "Extracted exponent (integer) : " << std::ilogb(t);

   // Next representable floating-point values
   T upper = std::numeric_limits<T>::max();
   T lower = std::numeric_limits<T>::min();
   std::cout << "Next going forward: " << std::nextafter(t, upper);
   std::cout << "Next going backward: " << std::nextafter(t, lower);
}
```

Finally, it is possible to determine the 'size' of a value given by the function `fpclassify`:

```
template <typename T>
   const char* Classify(T t) // T = float, double, long double, Integral
{
   switch (std::fpclassify(t))
   {
      case FP_INFINITE:    return "Inf";
      case FP_NAN:         return "NaN";
      case FP_NORMAL:      return "normal";
      case FP_SUBNORMAL:   return "subnormal";
      case FP_ZERO:        return "zero";
      default:             return "unknown";
   }
}
```

```
template <typename T>
   void classify(const T& t)
{
   std::cout << "*Value: " << t << '\n';
   std::cout << "\tFinite: " << std::isfinite(t) << '\n';
   std::cout << "\tInfinite: " << std::isinf(t) << '\n';
   std::cout << "\tIs NaN: " << std::isnan(t) << '\n';
   std::cout << "\tIs Normal: "<< std::isnormal(t) << '\n';
   std::cout << "\tIs Negative: " << std::signbit(t) << '\n';
}
```

8.3 A TOUR OF `std::numeric_limits<T>`

This class template provides a standardised way to query various properties of arithmetic types. It offers a number of member constants and member functions that can be used in applications. A specialisation exists for each cv (`const volatile`) qualified version of each arithmetic type in addition to each unqualified specialisation. We note that non-arithmetic standard types (for example, `std::complex<T>`) do not have specialisations.

```
template <typename T>
   void NumericConstants(const std::string& arithType)
{
   std::cout << '\n' << "**Type: " << arithType
         << "\n, #representable digits: "
         << std::numeric_limits<T>::digits << "\n";

   // Infinity, NaN and detection of denormalisation/subnormal numbers
   std::cout << "Has a special value for infinity: " << std::boolalpha
             << std::numeric_limits<T>::has_infinity << "\n";
   std::cout << "Has a special value for quiet NaN: " << std::boolalpha
             << std::numeric_limits<T>::has_quiet_NaN << "\n";
   std::cout << "Has a special value for signaling NaN: "<< std::boolalpha
             << std::numeric_limits<T>::has_signaling_NaN << "\n";

   std::cout << "Float subnormal styles: -1 == indeterminate, 0 == absent,
            1 == present: " << "\n";
   std::cout << "Supports subnormal values: "
             << std::numeric_limits<T>::has_denorm << "\n";
   std::cout << "Detects subnormal loss of precision: " << std::boolalpha
             << std::numeric_limits<T>::has_denorm_loss << "\n";

   std::cout << "Float rounding style (see std::float_round_style): "
             << std::numeric_limits<T>::round_style << "\n";

   // Digits etc.
   std::cout << "For int/floats, number of essential bits or digits in
            significand: " << std::numeric_limits<T>::digits << "\n";
```

```
    std::cout << "Number of decimal digits representable without change:"
              << std::numeric_limits<T>::digits10 << "\n";
    std::cout << "Smallest neg power of radix valid normalised number: "
              << std::numeric_limits<T>::min_exponent << "\n";
    std::cout << "One more than largest power of radix that gives a valid
             floating point value: "
              << std::numeric_limits<T>::max_exponent << "\n";

}

template <typename T>
    void NumericFunctions(const std::string& arithType)
{
    std::cout << '\n' << "**Type: " << arithType << '\n';
    std::cout << "Largest finite value: " << std::numeric_limits<T>::max();
    std::cout << "Smallest finite value: "<< std::numeric_limits<T>::min();
    std::cout << "Lowest finite value: " << std::numeric_limits<T>::min();
    std::cout << "Postive infinity value: "
              << std::numeric_limits<T>::infinity() << "\n";
    std::cout << "Difference between 1.0 and next representable value: "
              << std::numeric_limits<T>::epsilon() << "\n";

    std::cout << "Fl rounding: 0.5 to nearest digit, 1.0 to 0 or inf";
    std::cout << "Maximum rounding rounding error: " <<
    std::numeric_limits<T>::round_error() << "\n";

    std::cout << "Return value for quiet NaN: " << std::boolalpha
              << std::numeric_limits<T>::quiet_NaN() << "\n";
    std::cout << "Return value for signaling NaN: "
              << std::numeric_limits<T>::signaling_NaN() << "\n";

    std::cout << "Smallest positive subnormal value: " << std::hex
              << std::numeric_limits<T>::denorm_min << "\n";

}


int main()
{
    NumericConstants<float>("float");
    NumericConstants<double>("double");

    NumericFunctions<float>("float");
    NumericFunctions<double>("double");

    // 100-digit precision type from Boost multiprecision
    // Discussed in Appendix C in more detail
    namespace mp = boost::multiprecision;
    NumericConstants<mp::cpp_dec_float_100>("100-digit type");
    NumericFunctions<mp::cpp_dec_float_100>("100-digit type");
```

```
    NumericConstants<mp::cpp_dec_float_50>("50-digit type");
    NumericFunctions<mp::cpp_dec_float_50>("50-digit typet");

    return 0;
}
```

A subset of the functionality consisting of public static member constants and public static member functions in the class template is:

```
enum float_round_style
{

      round_indeterminate = -1,
      round_toward_zero = 0,
      round_to_nearest = 1,
      round_toward_infinity = 2,
      round_toward_neg_infinity = 3
};
```

8.4 AN INTRODUCTION TO ERROR ANALYSIS

A major topic in numerical analysis and its applications is the study of approximation errors and how errors are propagated in numerical processes and algorithms. In particular, we need some way of quantifying the concept of error because the measurement of data is not precise in general and floating-point data types have a fixed precision. Approximations are used instead of the real or exact data. In other words, we need some way to measure the difference between a floating-point number and the real number that it approximates. To this end, let the value a be an approximation to the value v. Some examples are:

- Absolute error:

$$\varepsilon = |v - a|. \tag{8.1}$$

- Relative error:

$$\eta = \varepsilon/|v| = \left|\frac{v-a}{v}\right| = \left|1 - \frac{a}{v}\right|. \tag{8.2}$$

- Percentage error:

$$\delta = 100\% \times \eta = 100\% \times \varepsilon/|v| = 100\% \times \left|\frac{v-a}{v}\right|. \tag{8.3}$$

The relative error is often used when comparing approximations of numbers of widely different sizes. Second, it makes sense only when the denominator in equations (8.2) and (8.3) is not zero. As an example, if $v = 50$ and $a = 49.9$ then the absolute error is 0.1, the relative error is $0.1/50 = 0.002$ and the percentage error is 2%.

A *round-off error* (*rounding error*) is the difference between the calculated approximation of a number and its exact mathematical value due to rounding. It is important to take this into consideration in numerical analysis and its applications where it is essential to estimate errors in calculations (including round-off errors) when using approximation equations and algorithms, especially when using finitely many digits to represent real numbers (which in theory have infinitely many digits). When a sequence of calculations subject to rounding error is made, errors may accumulate, sometimes dominating the calculation. In *ill-conditioned* problems, significant errors may accumulate.

We note that these definitions can be extended to cases in which the variables are vectors and matrices. In this case, the concept of the absolute value of a number is replaced by the concepts of *vector norm* and *matrix norm*, respectively. We discuss these concepts in more detail in later chapters when we need them in applying the finite difference method, for example.

8.4.1 Loss of Significance

This is an undesirable effect in calculations using finite-precision arithmetic. It occurs when an operation on two numbers increases relative error substantially more than it increases absolute error, for example in subtracting two nearly equal numbers (known as *catastrophic cancellation*). The effect is that the number of significant digits in the result is unacceptably reduced.

8.5 EXAMPLE: NUMERICAL QUADRATURE

We consider the common case of computing approximate values of one-dimensional integrals. In order to reduce the scope we focus on *Gauss quadrature*. This is a popular scheme in computational finance and it is a weighted sum of function values at specified points in the domain of integration. An n-point Gaussian quadrature rule yields a result that is exact for polynomials of degree $2n-1$ or less by a suitable choice of integration points and weights. In general, the domain of integration is $[-1,1]$:

$$\int_{-1}^{1} f(x)dx = \sum_{j=1}^{n} w_j f(x_j)$$

where:

$$w_j = \text{weights}, x_j = \text{integration points } j = 1, \ldots, n.$$

This formula only produces good results if the function $f(x)$ can be accurately approximated by a polynomial. For example, it is not suitable for functions with singularities. For a general interval (a, b) the Gauss integration rule becomes:

$$\int_{a}^{b} f(x)dx \approx \frac{b-a}{2} \sum_{j=1}^{n} w_j f\left(\frac{b-a}{2}x_j + \frac{a+b}{2}\right).$$

In this case the related polynomials are *Legendre polynomials* (Abramowitz and Stegun, 1965, formula 25.4.29).

We now discuss a small C++ software framework for *Gauss–Legendre* quadrature. We model the following entities:

- The *integrand* (the function to be integrated).
- The domain of integration.
- The numerical quadrature method.

We model these entities as classes. The class to model the integrand and domain is:

```
#include <functional>
#include <tuple>

template <typename T>
   class IIntegration
{
private:
   std::function<T(T x)> f;        // Function be integrated
   std::tuple<T, T> range;         // Interval of integration

public:
   IIntegration(const std::function<T(T x)>& function,
                const std::tuple<T, T>& interval)
         : f(function), range(interval) {}

   // Getter functions (V2 A = A(x), B = B(x))
   T A() const
   { // Lower bound

      return std::get<0>(range);
   }

   T B() const
   { // Upper bound

      return std::get<1>(range);
   }

   T operator() (T x)
   {
      return f(x);
   }
};
```

The class that implements Gauss–Legendre quadrature is:

```
template <typename T>
   class GaussLegendre
{
private:
   std::vector<T> w;               // Weights
   std::vector<T> x;               // Nodes
   IIntegration<T> fun;            // Function + interval
```

```
public:
   GaussLegendre(const IIntegration<T>& integrand,
      const std::vector<T>& weights, const std::vector<T>& nodes)
      : fun(integrand), w(weights), x(nodes) {}

   T operator () ()
   {

      // Using Gauss Legendre n-point method
      long N = x.size();

      T res = 0.0;

      T a = fun.A(); T b = fun.B();
      T alpha = (b-a) / 2.0;
      T beta = (b+a) / 2.0;

      T xl;
      for (long i = 0; i < N; ++i)
      {
         //xl = alpha*x[i] + beta;
         // infinite precision
         xl = std::fma(alpha, x[i, beta);
         res += w[i] * fun(xl);
      }

      return alpha*res;
   }

};
```

We support the 10-point and 2-point Gauss–Legendre rules. The corresponding weights and integration points are:

```
// Gauss Legendre for nodes and weights
const std::vector<double> wGL10 = {0.2955242247147529,
0.2955242247147529, 0.2692667193099963, 0.2692667193099963,
0.2190863625159820, 0.2190863625159820, 0.1494513491505806,
0.1494513491505806, 0.0666713443086881, 0.0666713443086881};

const std::vector<double> xGL10 = { -0.1488743389816312,
0.1488743389816312, - 0.4333953941292472, 0.4333953941292472,
-0.6794095682990244, 0.6794095682990244, -0.8650633666889845,
0.8650633666889845, -0.9739065285171717, 0.9739065285171717 };

const std::vector<double> xGL20 = { -0.0765265211334973,
0.0765265211334973, -0.2277858511416451, 0.2277858511416451,
-0.3737060887154195,0.3737060887154195, -0.5108670019508271,
0.5108670019508271, -0.6360536807265150, 0.6360536807265150,
-0.7463319064601508, 0.7463319064601508, -0.8391169718222188,
```

```
0.8391169718222188, -0.9122344282513259, 0.9122344282513259,
-0.9639719272779138, 0.9639719272779138, -0.9931285991850949,
0.9931285991850949 };

const std::vector<double> wGL20 = { 0.1527533871307258,
0.1527533871307258, 0.1491729864726037, 0.1491729864726037,
0.1420961093183820, 0.1420961093183820, 0.1316886384491766,
0.1316886384491766, 0.1181945319615184, 0.1181945319615184,
0.1019301198172404,0.1019301198172404, 0.0832767415767048,
0.0832767415767048, 0.0626720483341091, 0.0626720483341091,
0.0406014298003869, 0.0406014298003869, 0.0176140071391521,
0.0176140071391521 };
```

Finally, a test program is:

```
int main()
{
   auto f = [](double x) { return std::exp(x); };
   auto range = std::make_tuple(0.0, 5.0);

   IIntegration<double> integrandInfo(f, range);


   GaussLegendre<double> glIntegrator10(integrandInfo,
                                        wGL10, xGL10);
   std::cout << "Integral 10 nodes: " << std::setprecision(20)
             << glIntegrator10() << '\n';

   GaussLegendre<double> glIntegrator20(integrandInfo,
                                        wGL20, xGL20);
   std::cout << "Integral 20 nodes: " << std::setprecision(20)
             << glIntegrator20() << '\n';

   std::cout << "Exact e^5: " << std::setprecision(20)
             << std::exp(5.0) << '\n';

   return 0;
}
```

The output from this program is:

```
Integral 10 nodes: 147.41315910257662836
Integral 20 nodes: 147.41315910257657151
Exact e^5: 148.41315910257659993
```

This approach can be generalised to numerical quadrature schemes.

8.6 OTHER USEFUL MATHEMATICAL FUNCTIONS IN C++

In this section we assemble some of the C++ common mathematical functions for easy reference. We emphasise those functions that are related to floating-point types, numerical algorithms and functions. The categories of functions and a description of the main functions in each group (relating to floating-point variables) are:

- Basic operations: absolute value, minimum and maximum, remainder after division.
- Exponential functions: $e^x, 2^x, e^x - 1, \log x, \log_{10} x, \log_2 x, \log(1 + x)$ (some authors use *ln* instead of log).
- Power functions: $x^y, \sqrt{x}, \sqrt[3]{x}, \sqrt{x^2 + y^2}$.
- Trigonometric functions: sin, cos, tan and their inverses.
- Hyperbolic functions: sinh, cosh, tanh and their inverses.
- Error and gamma functions: error and complementary error function, gamma function and natural logarithm of the gamma function.
- Nearest integer floating-point operations.
- Classification and comparison.
- Types: implementation-independent rounding direction of quotient and remainder with floating-point division; what are the fundamental floating-point types which are at least as wide as `float` and `double`?

Most of the above functions are easy to use, which is why we do not need to give examples with the exception of error functions, comparison functions and a couple of basic functions. We focus on `double` as type for convenience. We show some examples on how to use the code:

```
void Remainder(double x, double y)
{
   std::cout << "Remainder\n";
   // Compute floating-point remainder of x/y
   std::cout << "fmod: " << std::fmod(x, y) << '\n';

   // Compose a float with magnitude of x and sign of y
   std::cout << "copysign: " << std::copysign(x, y) << '\n';

   std::cout << "remainder: " << std::remainder(x, y) << '\n';

   // remquo = remainder+sign of x/y +at least 3 oflast bits of x/y
   int* quadrant = nullptr;
   std::cout << "remquo: " << std::remquo(x, y, quadrant) << '\n';

}

void BasicOperations(double x, double y, double z)
{
   std::cout << "Basic Operations\n";
   // Fused multiply-add (mostly hardware)
   // Compute (x*y) + z to infinite precision and rounded once
   std::cout << "fma: " << std::fma(x, y, z) << '\n';
```

```
    std::cout << "fmax: " << std::fmax(x, y) << '\n';
    std::cout << "fmin: " << std::fmin(x, y) << '\n';
    std::cout << "fdim == max(0, x-y): " << std::fdim(x, y) << '\n';

    // NaN stuff
    auto f = NAN;
    std::cout << std::boolalpha << "is NaN?: " << std::isnan(f);
    std::cout << "fmax: " << std::fmax(f, y) << '\n'; // y
    std::cout << "fmin: " << std::fmin(f, y) << '\n'; // y
}


void Comparison(double x, double y)
{ // Comparing floating point numbers
  // If at least one of the arguments is NaN then result is NaN

    std::cout << "Comparison\n";
    std::cout << std::boolalpha << "is greater?: "
              << std::isgreater(x,y) << '\n';
    std::cout << std::boolalpha << "is greaterequal?: "
              << std::isgreaterequal(x, y) << '\n';

    std::cout << std::boolalpha << "is less?: " << std::isless(x, y);
    std::cout << std::boolalpha << "is lessequal?: "
              << std::islessequal(x, y) << '\n';

    // True if x < y || x > y
    std::cout << std::boolalpha << "is lessgreater?: "
              << std::islessgreater(x, y) << '\n';
    std::cout << std::boolalpha << "is lessgreater?: "
              << std::islessgreater(x, x) << '\n';

    double eps = std::numeric_limits<double>::epsilon();
    std::cout << std::boolalpha << "is lessgreater?: "
              << std::islessgreater(x, x + eps) << '\n';
    std::cout << std::boolalpha << "is lessgreater?: "
              << std::islessgreater(x, x + 10*eps) << '\n';

    // NaN
    auto f = NAN;
    std::cout << std::boolalpha << "is lessgreater NaN?: "
          << std::islessgreater(x, f) << '\n';

    // Can we compare NaNs withh normal numbers?
    std::cout << std::boolalpha << "is unordered?: "
              << std::isunordered(x, f) << '\n';
    std::cout << std::boolalpha << "is unordered?: "
              << std::isunordered(x, y) << '\n';
}
```

```
void ErrorGammaFunctions(double x)
{ // Error and gamma functions

   std::cout << "Error and Gamma functions\n";
   std::cout << "erf: " << std::erf(x) << '\n';
   std::cout << "erfc: " << std::erfc(x) << '\n';

   std::cout << "tgamma: " << std::tgamma(x) << '\n';
   std::cout << "lgamma: " << std::lgamma(x) << '\n';
}
```

A test program is:

```
int main()
{
   double x = 5.1; double y = 3.0; double z = 5.3;

   Remainder(x, y);
   BasicOperations(x, y, z);
   Comparison(x, y);
   ErrorGammaFunctions(x);

   return 0;
}
```

The output from this program is:

```
Remainder

fmod: 2.1
copysign: 5.1
remainder: -0.9
remquo: -0.9

Basic Operations
fma: 20.6
fmax: 5.1
fmin: 3
fdim == max(0, x-y): 2.1
is NaN?: true
fmax: 3
fmin: 3

Comparison
is greater?: true
is greaterequal?: true
is less?: false
is lessequal?: false
is lessgreater?: true
is lessgreater?: false
```

```
is lessgreater?: false
is lessgreater?: true
is lessgreater NaN?: false
is unordered?: true
is unordered?: false

Error and Gamma functions
erf: 1
erfc: 5.49382e-13
tgamma: 27.9318
lgamma: 3.32976
```

You may find these functions to be of use in your applications, especially when you wish to increase the latter's robustness.

8.7 CREATING C++ LIBRARIES

In this section we discuss a number of ways to encapsulate compiled source code as libraries.

8.7.1 Creating Static C++ Libraries

At a certain stage a developer may wish to package code into units or *libraries* that are written once and reused in many applications. The developer may prefer not to distribute source code for security reasons and because the developer's clients can then link the static library as part of their application without having to include very many source files.

In this section we discuss how to create a *static library* in Microsoft's Visual Studio (a discussion of how to do this for other compilers is outside the current scope). The main steps are: (1) create a static library project; (2) add classes to the static library; (3) build the project to produce a .lib file; (4) create a Console executable project, reference the static library and use its functionality in the main method. For more specific details, see the Microsoft online documentation.

We take a simple example to show the mechanics of creating a static library. We take a class containing a default constructor and a number of static member functions:

```
// StaticLib.hpp

namespace StaticFuncs
{
   class MyStaticFuncs
   {
   public:

      MyStaticFuncs() {}

      // Returns a + b
      static double Add(double a, double b);
```

```
        // Returns a - b
        static double Subtract(double a, double b);

        // Returns a * b
        static double Multiply(double a, double b);

        // Returns a / b
        static double Divide(double a, double b);
    };
}

// StaticLib.cpp
#include "StaticLib.hpp"

#include <stdexcept>

namespace StaticFuncs
{
    double MyStaticFuncs::Add(double a, double b)
    {
        return a + b;
    }

    double MyStaticFuncs::Subtract(double a, double b)
    {
        return a - b;
    }

    double MyStaticFuncs::Multiply(double a, double b)
    {
        return a * b;
    }

    double MyStaticFuncs::Divide(double a, double b)
    {
        return a / b;
    }
}
```

We add the .cpp file containing this code and build the project. A file called `StaticLib.lib` will be created that is added as a reference in the executable Console project. The test code is:

```
// TestStaticLib.cpp

#include <iostream>
#include "StaticLib.hpp"

int main()
{
```

```
    // Test
    StaticFuncs::MyStaticFuncs f;

    double a = 7.4;
    int b = 99;

    std::cout << "a + b = "
              << StaticFuncs::MyStaticFuncs::Add(a, b) << std::endl;
    std::cout << "a - b = "
              << StaticFuncs::MyStaticFuncs::Subtract(a, b) << std::endl;
    std::cout << "a * b = "
              << StaticFuncs::MyStaticFuncs::Multiply(a, b) << std::endl;
    std::cout << "a / b = "
              << StaticFuncs::MyStaticFuncs::Divide(a, b) << std::endl;

    return 0;
}
```

Summarising, clients receive the files `StaticLib.lib` and `StaticLib.hpp` but not `StaticLib.cpp`. We must tell the compiler where to find the library file as well as telling it in which directory to look.

More generally, it is beneficial to encapsulate reusable code in libraries. We can group related code into easy-to-use functions in order to reduce cognitive overload. Furthermore, we can introduce checks and exception-handling mechanisms to make the code more robust. We take an example that uses C++11 code and that is closely related to computational finance applications. In this case we encapsulate functionality relating to the univariate normal distribution function. We shall discuss the code in more detail in later chapters:

```
// Normal.hpp
//
// Useful normal distribution functionality in one place
//
// (C) Datasim Education BV 2018
//

#include <cstdlib>
#include <random>

namespace Distributions
{
    double PdfNormal(double x); // n(x)

    double CdfNormal(double x); // N(x)

    class NormalDistribution
    { // Test case
    private:
        std::default_random_engine eng;
        std::normal_distribution<double> nor;
```

```
   public:
      NormalDistribution(double a=0.0, double b=1.0);

      // Generate normal(a,b) normal variate
      double rng();

      // Create an array of normal variates
      std::vector<double> rngArray(long n);

      // Create an array of normal variates and shuffle them
      std::vector<double> rngShuffledArray(long n);

      // Create an array by Mersenne Twister
      std::vector<double> rngMTArray(long n);
   };
}

// Normal.cpp
//
// Useful normal distribution functionality in one place
//
// (C) Datasim Education BV 2018
//

#include <cmath>
#include "Normal.hpp"
#include <algorithm>


const double PI = 3.141592653589793;

namespace Distributions
{
   double PdfNormal(double x)
   { // aka n(x)

      double factor = 1.0 / std::sqrt(2.0*PI);

      return factor * std::exp(-0.5*x*x);
   }


   double CdfNormal(double x)
   { // aka N(x)

      const double sqrt2Inv = 1.0 / std::sqrt(2.0);
      return 0.5*(1.0 + std::erf(x * sqrt2Inv));
   }
```

```
    NormalDistribution::NormalDistribution(double a, double b)
          : eng(std::default_random_engine()),
          nor(std::normal_distribution<double>(a, b))
    {
       std::random_device rd;
       eng.seed(rd());
    }

    double NormalDistribution::rng()
    {
       return nor(eng);
    }

    std::vector<double> NormalDistribution::rngArray(long n)
    { // Create an array of normal variates

       std::vector<double> v;
       for (long i = 0; i < n; ++i)
       {
          v.emplace_back(rng());
       }

       return v;
    }

    std::vector<double> NormalDistribution::rngShuffledArray(long n)
    { // Create an array of normal variates and shuffle them

       std::random_device rd;
       eng.seed(rd());
       std::vector<double> v = std::move(rngArray(n));
       std::shuffle(std::begin(v), std::end(v), eng);

       return v;
    }

    std::vector<double> NormalDistribution::rngMTArray(long n)
    { // Create an array by Mersenne Twister

       std::vector<double> v;
       std::random_device rd;
       std::mt19937 engMT(rd());

       for (long i = 0; i < n; ++i)
       {
          v.emplace_back(nor(engMT));
       }

       return v;
    }
}
```

A test program is:

```
// TestNormalDistribution.cpp
//
// Calling functionality from a static library.
//
// (C) Datasim Education BV 2018
//


#include "Normal.hpp"
#include <iostream>
#include <vector>

void Print(const std::vector<double>& v)

{
   for (auto e : v)
   {
      std::cout << e << ",";
   }
   std::cout << '\n';
}

void TestNormal(double a, double b, long n)
{
   using namespace Distributions;
   double x = 0.0;
   std::cout << "pdf: " << PdfNormal(x) << '\n';
   std::cout << "cdf: " << CdfNormal(x) << '\n';

   NormalDistribution nd(a,b);

   std::cout << "random number: " << nd.rng() << '\n';

   auto arr1 = std::move(nd.rngArray(n));
   Print(arr1);

   auto arr2 = std::move(nd.rngShuffledArray(n));
   Print(arr2);

   auto arr3 = std::move(nd.rngMTArray(n));
   Print(arr3);

}

int main()
{
   double a = 0.0; double b = 1.0;
   long n = 6;
```

```
    TestNormal(a, b, n);

    return 0;
}
```

8.7.2 Dynamic Link Libraries

We now discuss how to create *dynamic link libraries* (DLLs) in Visual C++. A DLL is an executable file that acts as a shared library of functions and resources. An executable can then call functions or use resources that are stored in the DLL file. These functions and resources are compiled and deployed separately from the executables that use them. The operating system can load the DLL into the executable's memory space when the executable is loaded, or on demand at run-time. DLLs also make it easy to share functions and resources across executables. Multiple applications can access the contents of a single copy of a DLL in memory at the same time. Some of the advantages (compared to static link libraries) and features of DLLs are:

- They save memory and reduce swapping. Many processes can use a single DLL simultaneously. Compare to static link libraries for which *Windows* needs to load a copy of the library code for each application that is built with a static link library.
- Saving of disk space; multiple applications can share a single copy of the DLL on disk.
- DLLs are easier to upgrade. Applications that use DLLs do not need to be recompiled or relinked when the functions in a DLL change, provided their signatures (return types and input parameters) do not change.
- Multilanguage program support. A DLL can be called by programs in various programming languages provided that they adhere to the `_stdcall` or *Pascal Calling Convention*. In this case functions remove the parameters from the stack before they return to the caller. VBA, for example, uses the standard calling convention. In normal C/C++ calling convention the caller cleans up the stack.

A potential disadvantage of using DLLs is that the application is not self-contained because it depends on the existence of a separate DLL module.

We take the same example as in Section 8.7.1. We only show the code that differs from the above code, namely the header file:

```
// Normal_DLL.hpp
//
// Useful normal distribution functionality in one place.
// We put the functionality in a DLL.
//
// (C) Datasim Education BV 2018
//

// Tell the compiler and linker to export the functions
// from the DLL
#ifdef DISTRIBUTIONLIBRARY_EXPORTS
   #define DISTRIBUTIONLIBRARY_API __declspec(dllexport)
#else
   #define DISTRIBUTIONLIBRARY_API __declspec(dllimport)
#endif
```

```
#include <cstdlib>
#include <random>

namespace Distributions
{
   double DISTRIBUTIONLIBRARY_API PdfNormal(double x);   // n(x)

   double DISTRIBUTIONLIBRARY_API CdfNormal(double x);   // N(x)

   class NormalDistribution
   { // Test case
      private:

      std::default_random_engine eng;
      std::normal_distribution<double> nor;

      public:

      DISTRIBUTIONLIBRARY_API NormalDistribution(double a=0.0,
      double b=1.0);

      // Generate normal(a,b) normal variate
      DISTRIBUTIONLIBRARY_API double rng();

      // Create an array of normal variates
      DISTRIBUTIONLIBRARY_API std::vector<double> rngArray(long n);

      // Create an array of normal variates and shuffle them
      DISTRIBUTIONLIBRARY_API std::vector<double> rngShuffledArray
                                          (long n);

      // Create an array by Mersenne Twister
      DISTRIBUTIONLIBRARY_API std::vector<double> rngMTArray(long n);
   };
}
```

The keyword `__declspec(dllexport)` provides a means to *export* functions from an .exe or .dll file whereas programs that use public symbols defined by a DLL use the keyword `__declspec(dllimport)` to import them. The code file and test program are the same as in Section 8.7.1.

Finally, we discuss how to create a DLL library in Visual Studio (a discussion of how to do this for other compilers is outside the current scope). The main steps are: (1) create a DLL project; (2) add classes to the DLL library; (3) build the project to produce a .dll file; (4) create a Console executable project, reference the DLL library and use its functionality in the main method. For more specific details, see the Microsoft online documentation.

On a technical note, we make the .dll file visible to the .exe by copying the .dll to the same directory where the .exe file resides or by placing the DLL project in the .exe's solution project.

It is possible to create DLLs using *module-definition files* but a discussion is outside the current scope. See Rogerson (1997).

8.7.3 Boost C++ DLLs

Standard C++ does not yet support compiler-independent libraries although there are efforts afoot to redress this state of affairs using C++ modules. *Modules* are a mechanism to package libraries and encapsulate their implementations. They differ from the C approach of translation units and header files primarily in that all entities are defined in just one place (even classes, templates, etc.). There are three primary goals:

- Significantly improve build times of large projects.
- Enable a better separation between interface and implementation.
- Provide a viable transition path for existing libraries.

A possible workaround to lack of support for modules is to use the *Boost DLL* library. The goal of this *header-only* library is to simplify plug-in development using C++ in a portable cross-platform manner. Some features are:

- Load libraries.
- Import any native functions and variables.
- Make alias names for C++ mangled functions and symbols.
- Query libraries for sections and exported symbols.
- Self-loading and self-querying.
- Getting program and module location by exported symbol.

To use the library we include the `<boost/dll.hpp>` header. After that we can import and export functions and variables. A further discussion is outside the scope of this book. The Boost.DLL online documentation has several tutorials.

8.8 SUMMARY AND CONCLUSIONS

In this chapter we gave an introduction to the IEEE 754 technical standard for floating-point computation. Many hardware floating-point units now support IEEE 754. We discussed how a number of the recommendations in that standard are implemented in C++. The related functionality promotes the robustness and reliability of applications. We also give a compact summary of how to create static and dynamic libraries.

8.9 EXERCISES AND PROJECTS

1. Which three of the following statements are true?
   a) `std::isinf` determines if a number is positive infinity or negative infinity.
   b) `std::fpclassify` categorises floating-point numbers.
   c) `std::fpclassify` has functionality to process floating-point status flags.
   d) `std::isfinite` is true for NaN numbers.

2. Which one of the following is not a NaN?
   a) `std::exp(800).`
   b) `INFINITY - INFINITY.`
   c) `0/0.`
   d) `std::log(0.0/1.0);.`

3. Which of the following statements are true?
- **a)** `DBL_MIN/2` is subnormal.
- **b)** `- 0.0` is zero.
- **c)** `1.0/0.0` is `Inf`.
- **d)** `1.0` is normal.

4. Explain what the output of the following code is:

```
double S = 2.0; double K = 65.0; double h = 1.0;

double sum = 0.0;
for (int i = 1; i <= 100; ++i)
{
   sum += std::sqrt(S / K); S -= h;
   std::cout << i << ",";
}

std::cout << std::boolalpha << "Sum: " << sum << '\n';
```

5. (Gauss–Legendre Quadrature)

In this exercise we discuss some topics related to Section 8.5.

Answer the following questions:

- **a)** Implement the 6-point formula with the following pairs of weights and abscissae:

```
{ 0.3607615730481386   0.6612093864662645}
{ 0.3607615730481386   -0.6612093864662645}
{ 0.4679139345726910   -0.2386191860831969}
{ 0.4679139345726910   0.2386191860831969}
{ 0.1713244923791704   -0.9324695142031521}
{ 0.1713244923791704   0.9324695142031521}
```

Test the accuracy of this scheme against the 10-point and 20-point rules.

- **b)** Test the Gauss–Legendre formulae on the simple integrands:

$$\int_1^2 \frac{dx}{x} = \log 2$$

$$\int_0^{10} (1+x^2)^1 dx \approx 1.471127674.$$

- **c)** We examine the oscillatory integrands (*Fresnel integrals*) defined by:

$$S(x) = \int_0^x \sin\left(\frac{\pi}{2}t^2\right)dt$$

$$C(x) = \int_0^x \cos\left(\frac{\pi}{2}t^2\right)dt.$$

Focus on $S(x)$ and approximate it using the Gauss–Legendre formulae. Test against $S(6.0) = 0.44696$ and $S(6.40) = 0.49646$.

d) Implement the series representation for $S(x)$ (Wijngaarden and Scheen, 1949):

$$S(t) = \sum_{k=0}^{\infty} S_{4k+3} t^{4k+3}$$

where:

$$S_{4k+3} = (-1)^k \frac{(\pi/2)^{2k+1}}{(2k+1)!(4k+3)}.$$

Use the features in this chapter to implement this sum while taking underflow, overflow and other possible computational difficulties into account.

e) It is possible to transform the current problem to an ordinary differential equation:

$$\frac{dS}{dx} = \sin\left(\frac{\pi}{2}x^2\right),\ x > 0$$
$$S(0) = 0.$$

Solve this problem numerically using the implicit Euler scheme. How many steps do you need to take in order to get the same accuracy as in parts a) and b)?

6. (Exception Handling, Small Research Project)

This chapter seems to be a suitable place to discuss how C++ supports exception handling, error codes and error conditions (see Josuttis, 2012 for an introduction). All exceptions generated by the standard library inherit from `std::exception`. We reduce the scope by focusing on the following kinds of exceptions:

```
logic_error
   invalid_argument
   domain_error
   length_error
   out_of_range
   future_error(C++11)

runtime_error
   range_error
   overflow_error
   underflow_error

bad_weak_ptr(C++11)
bad_function_call(C++11)
```

Answer the following questions:

a) Determine the kinds of run-time errors that can be handled by these exception classes. Think of simple examples to generate and catch these kinds of exceptions.

b) Determine how to discover and catch underflow, overflow and NaN errors.

c) Can any of these exception classes be used with the code in Section 8.5?

7. Consider the following code:

```
// Exception handling
{
   std::cout << "STL exception time\n";
   double factor = 2.0;
   double val
         = factor*std::numeric_limits<double>::max(); // infinity
   val -= val; // NaN

   try
   {
      double b = std::exp(val);
   }
   catch (std::domain_error& e)
   {
      // 1.
      std::cout << "Error " << e.what() << std::endl;
   }
   catch (...)
   {
      // 2.
      std::cout << "Unexpected error " << std::endl;
   }
   std::cout << "end " << std::endl;
}
```

When this code is run, what happens?

a) A domain error occurs.
b) An unexpected error occurs.
c) The program crashes.
d) The program runs to completion without throwing an exception.

8. (Machine Epsilon)

Machine epsilon gives an upper bound on the relative error due to rounding in floating-point arithmetic. In C++11, epsilon is the difference between 1.0 and the next representable value of the given floating-point type.

Answer the following questions:

a) Write a small C function to compute epsilon:

```
double epsilon = 1.0;

while ((1.0 + 0.5*epsilon) != 1.0)
{
   epsilon *= 0.5;
}
```

Generalise this function for other types.

b) Compare the value from part a) with the value produced by:

```
std::numeric_limits<double>::epsilon().
```

9. (Machine Precision Issues)

This exercise requires that you do some research into how C++11 and *Boost C++ Math Toolkit* support the following functionality (*x* is a given value):

- The next representable value that is greater than *x*. An `overflow_error` is thrown if no such value exists (`float_next`).
- The next representable value that is less than *x*. An `overflow_error` is thrown if no such value exists (`float_prior`).
- Advance a floating-point number by a specified number of ULPs (*Units in Last Place*) (`float_advance`).
- Find the number of gaps/bits/ULPs between two floating-point values (`float_distance`).
- (C++11 and Boost) Return the next representable value of *x* in the direction *y* (`std/boost::nextafter(x,y)`).

The Boost header file to include is:

```
#include <boost/math/special_functions/next.hpp>
```

Take some specific values and experiment with these functions. Some examples are:

```
// Number gaps/bits/ULP between 2 floating-point values a and b
// Returns a signed value indicating whether a < b
double a = 0.1;
double b = a + std::numeric_limits<double>::min();
std::cout << boost::math::float_distance(a, b) << std::endl;
a = 1.0; b = 0.0;
std::cout << boost::math::float_distance(a, b) << std::endl;
```

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 9

An Introduction to Unified Software Design

No two languages are ever sufficiently similar to be considered as representing the same social reality. The worlds in which different societies live are distinct worlds, not merely the same world with different labels attached.

—Edward Sapir

The diversity of languages is not a diversity of signs and sounds but a diversity of views of the world.

—Wilhelm von Humboldt

9.1 INTRODUCTION AND OBJECTIVES

The first eight chapters of this book were devoted to the syntax and multiparadigm language features in C++. But syntax alone is not sufficient if we wish to create flexible and maintainable code. To this end, we introduce a design approach that subsumes and integrates a number of popular design methodologies to give us a *defined process* that we apply to the creation of small to complex libraries, frameworks and applications. We call it *Unified Software Design* (USD) for convenience and we shall apply it in a number of the remaining chapters of this book to applications such as Monte Carlo simulation, option pricing using PDE methods and to the creation of libraries and frameworks. Our goal is to have a standardised design approach to software development and this approach can also be applied by the reader when creating her own applications in computational finance.

The main goal of this chapter is to describe the underlying principles of USD and how it complements, subsumes and extends other design methodologies. Furthermore, we show how to analyse, design and implement small, medium-sized and complex applications. We give simplified (but not simple) generic examples to show the usefulness of the approach. Understanding the fundamental principles and the examples will help you make the transition to larger problems.

Some of the principles underlying our design approach can be summarised by the steps that György Pólya describes when solving a mathematical problem (Pólya, 1990):

1. First, you have to *understand the problem*.
2. After understanding what you are trying to solve, *make a plan*.
3. *Carry out the plan*.
4. *Look back* at your work. How could it be better?

We see these steps as being applicable to the software development process in general and to the creation of software for computational finance in particular. Getting each step right saves time and money. In short, we adopt the following tactic: *get it working, then get it right and only then get it optimised* (in that order).

We have applied USD to problems in a number of domains in the past (Duffy, 2004). In this book we use USD for computational finance applications. The examples in this chapter are generic and have been chosen to suit a wide audience.

9.1.1 Future Predictions and Expectations

We think that the approach taken in this chapter will be useful to applications in computational finance. We have used the techniques in this book and with clients in various application domains. Until now we have seen little published work on how to design stable designs in finance. In the future we expect maintainability to play a bigger role in this area.

It is now generally accepted that deep and badly designed C++ class libraries are difficult to maintain. In general, they tend to collapse under their own weight as new features are introduced into software systems that use these classes. One of the underlying issues with traditional object-oriented technology is that objects are too fine grained to model large and changeable software systems, especially during the early stages of software projects when the requirements are changing and the initial software architecture has stabilised. Objects and classes are useful during the detailed design phase of the software lifecycle. In this book we bridge the gap between requirements analysis and detailed design by decomposing the software system into loosely coupled subsystems (each of which has a well-defined single responsibility) that together satisfy the system's *core process*, which consists of a number of steps to transform external input to external output. This is one way to decompose a system. Another possible approach is discussed in Parnas (1972):

> *We have tried to demonstrate by these examples that it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such decisions from the others.*

This approach takes maintainability into account up-front and it immediately allows us to think about the possible application of popular *software design patterns* such as *Façade*, *Bridge* and *Adapter* (GOF, 1995) before we write a single line of code. These patterns tend to

be discovered at a much later stage using traditional object technology, when the software is more difficult to modify.

One of the goals of this book is to design and implement software systems that remain stable and maintainable as new features are added to them or as software bugs are fixed. We wish to avoid the following all-too-common nightmare:

> *A Big Ball of Mud is a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle. These systems show unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated. The overall structure of the system may never have been well defined. If it was, it may have eroded beyond recognition. Programmers with a shred of architectural sensibility shun these quagmires. Only those who are unconcerned about architecture, and, perhaps, are comfortable with the inertia of the day-to-day chore of patching the holes in these failing dikes, are content to work on such systems.*

In this book we propose a defined software process to avoid or mitigate this potential disaster. Even if you do not use our methods it is still vital that you take the time to construct stable software design blueprints before jumping into C++ code.

9.2 BACKGROUND

In this section we discuss the methods that have shaped and influenced USD. It is a multi-paradigm method to analyse, design and implement applications in a range of domains, for example process control, logistics and computer aided design (CAD). A discussion of these kinds of applications is outside the scope of this book. See Duffy (2004) for a discussion of this approach. Instead, we focus on applications related to computational finance and how they are implemented in C++.

USD can best be described as a process to analyse and design software systems. It uses features from a number of well-known system design approaches while at the same time it tries to avoid features that are more difficult to apply. The main phases in USD reflect Pólya's steps in Section 9.1:

- Phase I: System Scoping and Initial System Decomposition.
- Phases II: Identify System Components and Interfaces.
- Phase III: Detailed Design.
- Phase IV: Implementation, Review and Maintenance.

The execution of each phase can be seen as a process that maps input to output and we describe the *core process* as a sequence of (computable) *activities* that tie the process's output to its input. Ideally, we define a low-risk and seamless process to take user requirements and map them to code. To this end, we have used a number of established methods to analyse and design software systems. We now give a short description of each method to provide the reader with background information.

9.2.1 Jackson Problem Frames

Michael Jackson (not the singer) introduced the idea of a problem frame (see Jackson, 2001). A *problem frame* is defined as:

> *a kind of pattern. It defines an intuitively identifiable problem class in terms of its context and the characteristics of its domains, interfaces and requirements.*

Some of the reasons to discuss problem frames are:

- They provide us with ideas on how to document software architectures and software systems.
- They are based on a small set of very generic concepts.
- They provide reference models for classes of applications.
- It is a better starting point than using traditional object-oriented analysis.

We say that the *concern* of a problem frame captures the fundamental criteria of successful analysis for problems that fit that frame. Quoting Jackson (2001), the frame 'specifies what descriptions are needed, and how they must fit together to give a *convincing argument* that the problem has been fully understood and analysed'. Jackson identifies five basic reusable frames and these can be considered as being instances of a *problem frame model*.

The five basic forms in Jackson (2001) are:

- *Required behaviour*: some part of the physical world whose behaviour must be controlled until some conditions are satisfied. An example of such a system is a home heating system or an environmental controller.
- *Commanded behaviour*: the physical world is controlled by the actions of an operator. An example is a sluice gate problem; this problem is a model of a simple irrigation system.
- *Information display*: systems where we need constant information about the real world. The information must be presented in the required place in the required form. An example is a one-way traffic light system.
- *Simple workpiece*: a tool to allow users to control and edit text or graphics objects on a screen. The objects can subsequently be copied, printed or transformed in some way. An example is a CAD application.
- *Transformation*: computer-readable input files whose data must be transformed to give certain required output files. The output data must be in a particular format. An example of such a frame is a management information system (MIS).

One of the reasons for these forms is that we can relate our own systems to them by analogical *reasoning* techniques.

9.2.2 The Hatley-Pirbhai Method

This is a method for analysing *real-time systems*. It employs data flow decomposition in which model components are constructed in control and information space (see Hatley and Pirbhai, 1988). The authors take a hierarchical and iterative view of system development. In

particular, the method captures system requirements by viewing a system from three major perspectives:

- The process (functional) model.
- The control (state) model.
- The information (data) model.

The *data flow diagram* (DFD) view is the primary tool for depicting functional requirements in the Hatley–Pirbhai method. It partitions the requirements into processes or functions. These functions are connected by data flows to form a network. The second view is called the *control model* and it describes the circumstances under which the processes from the process model are performed. The control model examines the external events in a system and it is documented using finite state machines (FSMs). Finally, *information modelling* is the third perspective and it is not documented in Hatley and Pirbhai (1988).

We have been influenced by the Hatley–Pirbhai approach in a number of ways. First, it discusses the *context process* consisting of a single process, *terminators* (entities outside the context of the system) and *data flows*. Second, the main process is decomposed or levelled into subprocesses, thus promoting *separation of concerns*. Third, the usefulness of this method lies in the realisation that the context diagram, data flow and architecture are indispensable when analysing software systems. These concerns are missing from traditional object-oriented analysis.

9.2.3 Domain Architectures

A *domain architecture* (Duffy, 2004) is a reference model for a range of applications that share similar structure, functionality and behaviour. A domain architecture is a kind of *metamodel* that can be seen as a template for more specific systems.

We discuss five basic forms and one 'composite' form:

- *MIS* (management information system): produces high-level and consolidated decision-support data and reports based on transaction data from various sources.
- *PCS* (process control system): monitors and controls values of certain variables that must satisfy certain constraints. We are primarily interested in *exceptional events* and events that must be handled in the software system.
- *RAT* (resource allocation and tracking) system: monitors a request or some other entity in a system. The request is registered, resources are assigned to it, and its status in time and space is monitored. This is probably the most common model that we encounter in applications.
- *MAN* (manufacturing) system: creates finished products and services from raw materials and data.
- *ACS* (access control system): allows access to passive objects from active subjects. These systems are similar to security systems and the *reference model* in large computer systems.
- *LCM* (lifecycle model): a 'composite' model that describes the full lifecycle of an entity; it is a composition of MAN, RAT and MIS models.

9.2.4 Garlan-Shaw Architecture

An *architectural style* is a description of an architecture of a specific system as a collection of computational components together with a description of the interactions among these components, known as the *connectors* (see Shaw and Garlan, 1996). Examples of components are databases, layers, filters and clients. Examples of connectors are procedure calls, event broadcasts and database protocols. Shaw and Garlan propose several common architectural styles:

- *Dataflow systems*: for example, batch sequential, pipes and filters.
- *Call-and-return systems*: object-oriented systems, hierarchical layers, main program and subroutine.
- *Data-centred systems* (repositories): databases, hypertext systems, blackboards.
- *Independent components*: communicating processes, event-driven systems.
- *Virtual machines*: interpreters, rule-based systems.

These styles are *reference models* that we can apply in the detailed design stage of the software lifecycle. They are discussed in Shaw and Garlan (1996) and a more detailed account of a number of these styles can be found in POSA (1996) where they are documented in handbook form. A detailed discussion of architectural styles is outside the scope of this book.

9.2.5 System and Design Patterns

We discuss the system and design patterns of POSA and GOF respectively (see GOF, 1995; POSA, 1996). The patterns in POSA are concerned with large-scale system design while the GOF design patterns are more fine grained. For example, POSA describes how to design large systems in terms of patterns such as:

- Presentation–Abstraction–Control (PAC).
- Layers (to be discussed in Chapters 11 and 12).
- Blackboard.
- Model–View–Controller.
- Pipes and Filters.
- Microkernel.
- Proxy.
- Publisher–Subscriber.

The GOF design patterns are concerned with the lifecycle of objects (instances of classes):

- *Creational patterns*: flexible ways to create objects:
  - Abstract Factory (creating related instances of classes in a class hierarchy).
  - Factory method (creating instances of a given class).
  - Prototype (creating objects as 'clones' of some typical or representative object).
  - Builder (creating a complex object in a step-by-step fashion).
- *Structural patterns*: defining relationships between objects:
  - Composite: recursive aggregates and tree structures.
  - Proxy: indirect access to a resource via a 'go-between' object.
  - Bridge: separate a class from its various implementations.

- Facade: create a unified interface to a logical grouping of objects or subsystems.
- Adapter: convert the interface of one class into the interface of another otherwise incompatible class.

- *Behavioural patterns*: how objects send messages to each other:
  - Visitor: extend the functionality of a class hierarchy (non-intrusively).
  - State: implement a Harel/UML statechart (Harel and Politi, 2000).
  - Strategy: create flexible, interchangeable algorithms for object methods.
  - Observer: define synchronising procedures between objects.
  - Mediator: define a single communication 'hub' in a star of objects.
  - Command: encapsulate a request.

We remark that the GOF design patterns can be viewed as a special case of an LCM because we are interested in object lifetime; the main phases are the creation of an object (MAN), placing the object in some structure (RAT) and then monitoring how the object interacts with other objects in the object network (MIS).

9.3 SYSTEM SCOPING AND INITIAL DECOMPOSITION

In general, we try to understand as much of the problem as possible before delving into detailed design and coding work. To this end, our goal is to reduce *project risk* by locating the boundaries of the problem. In this way we must know what needs to be developed. Second, we decompose the initial amorphous ('ball of mud') and monolithic system into loosely coupled and autonomous systems with well-defined interfaces.

We need to be careful not to fall into the trap of *premature design and implementation*; our concern here is to understand the main data flow in the system and how the different systems partaking in the context diagram cooperate to realise that data flow.

9.3.1 System Context Diagram

The first step in understanding a software system is to describe the system (henceforth called SUD (*System Under Discussion*)) in terms of external systems where the inputs come from and where the outputs from the system go to, respectively.

We define the crucial concepts of *source* and *sink boxes* (DeMarco, 1978):

> *A* source *or* sink *is a person, organisation or software system that lies outside the context of the SUD system and that is a net originator or receiver of system data, respectively.*

It is obvious that we need to understand what a system's major sources and sinks are before proceeding. To this end, we represent the system context diagram by a drawing in which each system is depicted as a rectangular box with lines connecting the SUD with its sources and sinks. Even at this stage we would like (if possible) to distinguish between those external systems that must be included in the design (that is, those that must be implemented by the developer) and those that we do not design and for which we are only interested in the services that they provide or require with respect to the SUD. To this end, we add a single vertical line to those boxes representing the former category and no line for the latter category. The SUD is

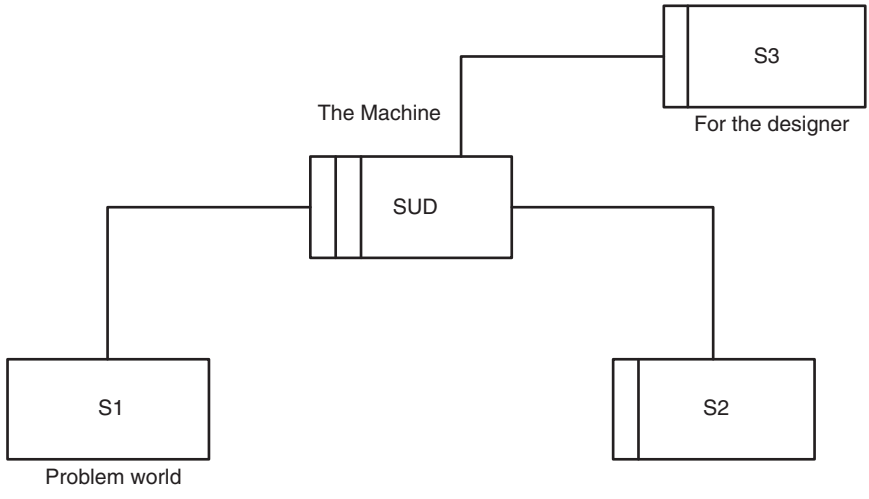


FIGURE 9.1 Example of context diagram (Jackson approach)

a special case by definition and its box will be annotated by two vertical lines. In this way we are able to distinguish between the *problem world* (the world outside the computer system) and the *solution world* (which is located in the computer). A generic example of a system context diagram is shown in Figure 9.1 (using the notation from Jackson, 2001).

Summarising, we describe the SUD as a black box that is surrounded by other systems that cooperate with the SUD to ensure that system goals are achieved. The discovery of the context diagram in applications is very important because it is a foundation for the discovery of other artefacts such as stakeholders, viewpoints, requirements and contractual interfaces between the SUD and its *satellite systems*. Furthermore, it is an indispensable tool for project managers who must determine project size and risk, not to mention monitoring system evolution once the development team has started on the software design.

Arriving at a stable context diagram is an iterative process in general and it can be created by analysing system requirements, interviewing domain experts and creating software prototypes. It is important to discover the most critical external systems in the early stages of software design because they are sources of requirements. See Exercise 9 where we simulate this requirement elicitation process.

Having created the system context diagram we now find the data and information exchange between the SUD and its external systems. This is a well-known technique in *Structured Analysis* (DeMarco, 1978; Hatley and Pirbhai, 1988) and it is described as a DFD as follows:

> *A network representation of a system. The system may be automated, manual or mixed. The DFD portrays the system in terms of its component pieces, with all interfaces among the components indicated.*

This is a practical definition and we adapt it in this book. At this stage we can use DFDs to model the high-level data flow between the SUD and its external systems. It is possible to check whether we understand what is going on and whether we have discovered the required systems

and their requirements. We shall see that they will allow us to determine the *responsibilities* of systems.

We take a well-known example to motivate the use of DFDs, namely the *Drink Vending Machine* (DVM) (see Hatley and Pirbhai, 1988). The problem is understandable to most readers while it is also challenging to analyse. We can generalise the DVM to other applications and domains. In fact, we already know that it is an instance system of the ACS domain architecture type (see Duffy, 2004). Those readers who develop interactive applications will hopefully find the solution of the DVM to be useful because there are many similarities between the features of the DVM and those of other applications.

We include the list of *features* and requirements from Hatley and Pirbhai (1988) (the pre-credit card era!):

- F1: Accept objects (candidate coins) from the customer as payment.
- F2: Check for slugs (not real coins), for example by validating size, weight, thickness and serrated edges.
- F3: Accept Euros only.
- F4: The system cannot be tricked by conniving people.
- F5: The customer should be able to select a product.
- F6: Check product availability; if not available, return coins to customer.
- F7: Products and their prices may change from time to time.
- F8: Return the customer's coins on request if she decides not to go through with the transaction.
- F9: Dispense product if it is available and enough coins have been inserted.
- F10: Return the correct change if the amount deposited is greater than the product price.
- F11: Disable the product selection after the product has been dispensed and until the next validated coin has been received.
- F12: Make deposited coins available for change.

The traditional DFD is shown in Figure 9.2. In this diagram we note that sources and sinks (*Products*, *Price Table* and *Coins*) are known as *data stores* and furthermore the processes that consume or produce data (or both) are shown as circles. This informal diagram adds to our understanding of the problem but it can be difficult to map to modern tools and languages. Furthermore, it does not scale well. Instead, we re-engineer this problem as a special case of an ACS as discussed in Duffy (2004). The canonical context diagram for all ACS systems is shown in Figure 9.3. The processes in Figure 9.2 will then be incorporated into the systems in Figure 9.3.

- *ACS:* the central system (we call it DVM).
- *Source*: the system corresponding to active users that initiate all requests.
- *Authentication*: the system for coin insertion and validation.
- *Resources*: the system containing drinks.
- *Sink*: recipient of status reports on transaction completion (there may be multiple sink systems).
- *MIS*: (possibly) remote management systems that communicate with the DVM.

DFDs are useful brainstorming tools when we wish to discover the information flow between the SUD and its external systems but further decomposition of the top-level DFD

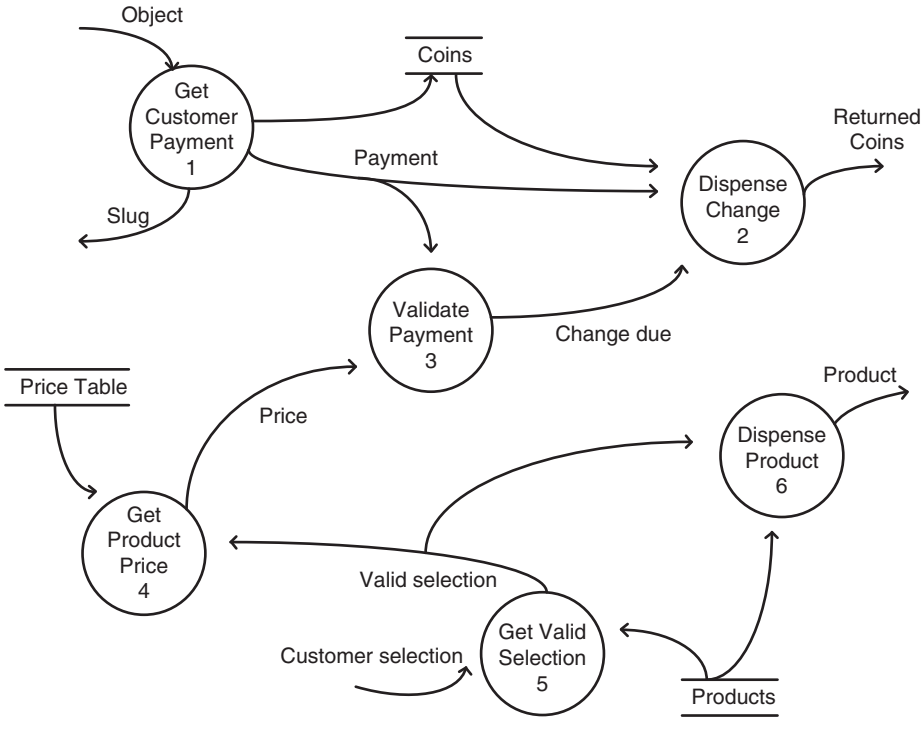


FIGURE 9.2 Traditional DFD

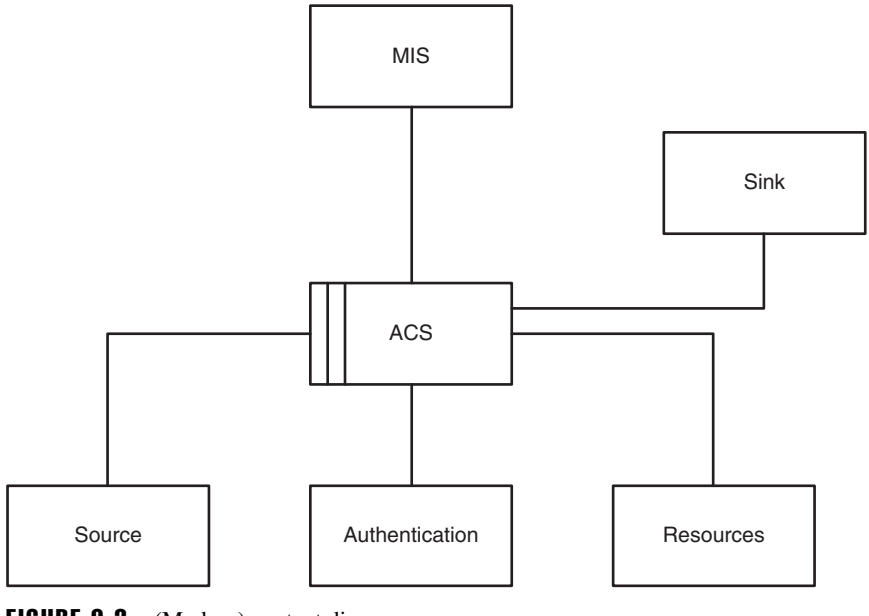


FIGURE 9.3 (Modern) context diagram

(for example, the case in Figure 9.2) is not yet desirable for a number of reasons. Instead, we try to formalise the top-level DFD by focusing on the data produced and consumed by the systems in the context diagram. In particular, analysing a problem by first determining what its *core process* is leads us to a more robust solution:

> *A core process is one whose deliverables are visible to external customers and it usually spans the whole organisation because several organisational units are involved in its execution.*

This means that we now start to concentrate on the major data flow, in particular the data that the critical sink systems receive. In other words, we adopt Pólya's heuristic H6 (*work backward*, see Exercise 1) as it is a good test of how well we understand the problem before proceeding. In fact, one of the first questions is to ask the customer what the system to be developed should deliver.

9.3.2 System Responsibilities and Services

At this stage we assume that we have a stable context diagram and that we are able to trace the high-level data flow through the system. We now address the issue of determining what the *responsibilities* of each system are. In other words, what does each system need from other systems and what does it deliver to other systems? More precisely, we define a *system service* as a discrete capability or behaviour that a system exhibits. A service can be an operation, function or transformation. It could also be an algorithm, transaction or a function to monitor external systems and devices. The characteristics of a service are:

- Its name. We can use a verb–noun combination, for example *CreateTransaction*.
- Its input and output parameters. We define a complete and consistent set of arguments that are syntactically correct and that can be refined during detailed design.
- *Preconditions*: these are the conditions that must be true in order for the service to execute. If a precondition evaluates to false then the service will not execute.
- *Postconditions*: these are the exit criteria for the service. In general, a postcondition is the state or condition that must be achieved or be valid when the service has finished execution.

We can consider this part of the project to be complete when each service has been scoped, documented and has also been assigned to a system. Ideally, each service should be assigned to a single system.

9.3.3 Optimisation: System Context and Domain Architectures

It is possible to design software systems using an iterative approach, especially if the requirements have not yet been pinned down when a project begins. To complicate things, *emerging requirements* can force a rewrite of the software system or they can even cause project cancellation due to budget overruns. One way to mitigate project risk is to draw on one's experience (or on the experience of others). To this end, the approach that we take here is to determine if

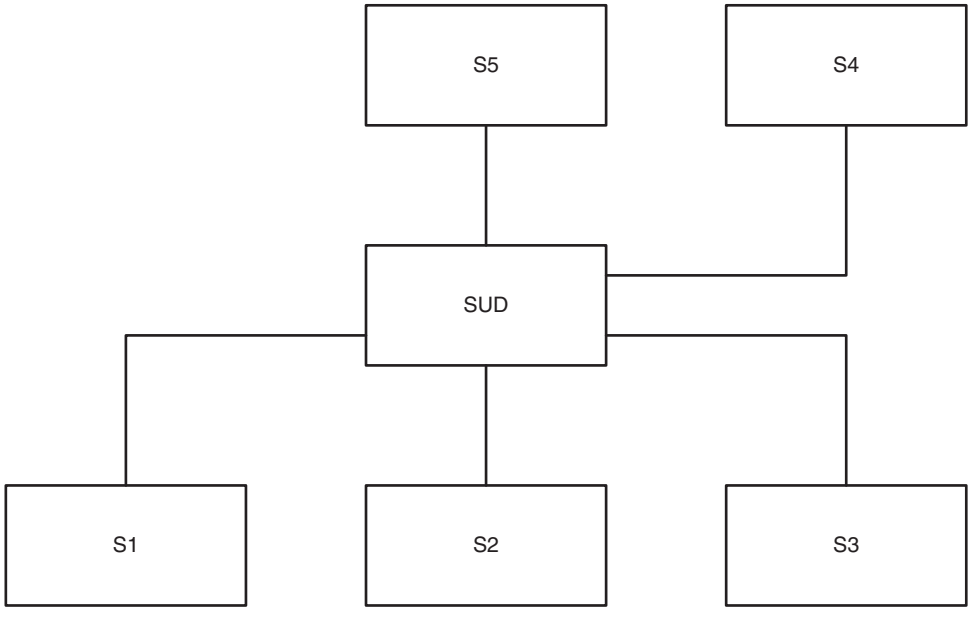


FIGURE 9.4 Canonical context diagram

the current system is a special case of one or more of the domain architectures that we introduced in Section 9.2.3. The styles have their differences but they have one thing in common, namely they all tend to have the same context diagram in their most general form. We show this in Figure 9.4 and it can be used as a template for any medium-sized application. In particular, the generic names SUD as well as the names S1 to S5 need to be renamed and their responsibilities must be found (as in Figure 9.3 for the case of the DVM system, for example). As motivation, we describe the general responsibilities of each system:

- S1: The system without which there would be no input data. All major events and data originate in this system.
- S2: This is a kind of help/classifier system that transforms the data from S1 to entities and objects that can be used (mainly) by system S3. In a sense, it is a kind of *object factory*.
- S3: The system that transforms the modified input data to the desired output form.
- S4: The external system that receives the output data. S4 could be a large system in its own right (with its own context diagram) and it can be designed in the same way as the current SUD.
- S5: There are (possibly optional) systems that monitor and control the SUD by exchanging information with it, for example downloading settings or performing *watchdog* duties.
- SUD: The central *mediator* system that coordinates the dataflow and control flow among the systems in the context diagram.

The essential systems to design as soon as possible are S1, S3, S4 and of course SUD (the other systems could be ignored or modelled as hard-coded *stubs* in an initial version). This is the minimal set to help kick-start the design process as it were. If you can identify these

systems then you will have considerably reduced project risk. We mention that each of the systems in Figure 9.4 can be multiple systems having their own context diagrams.

It is important to distinguish between the different domain categories. The distinguishing feature or 'separator' for a domain category is the type of data output that it produces. The reader can use the following initial separation mechanism to classify applications:

- MIS: produce decision-support information from transaction data. This could take the form of multidimensional data that is processed by *reporting systems*.
- MAN: create products and services from raw materials.
- RAT: track requests in time and space. Produce a status report concerning the request.
- PCS: satisfy certain conditions at all times. Monitor exceptional events.
- ACS: provide active authorised subjects with access to passive objects and resources.

We now take a toy example in C++ of a system based on the context diagram in Figure 9.4. This choice is for motivational purposes. We design the system using systems S1 (Input), S3 (Processing), S4 (Output) and SUD (Mediator). The core process is to create a string, trim it (remove leading and trailing blanks using functionality from the *Boost C++ String Algorithm* library), convert it to upper case and then finally display it on the console. The system can be seen as a very simple RAT system. It tracks a string.

We implement the system in C++ in which the SUD is a template class whose parameters correspond to the systems S1, S3 and S4. We agree on certain interfaces that these systems should implement so that the SUD can communicate with them. We use *private inheritance* as a trick to implement interfaces.

It is a simple *push model* in the sense that data (in this case a string) is pushed from one system to another one in a kind of pipeline model.

The class definition is:

```
#include <string>
#include <iostream>
#include <boost/algorithm/string.hpp>

template <typename I, typename O, typename Processing>
       class SUD : private I, private O, private Processing
{ // System under discussion

private:
       // Define the requires interfaces that SUD needs
       using I::message;           // Get input
       using Processing::convert;  // Convert input to output
       using O::print;             // Produce output
       using O::end;               // End of program

public:
       void run()
       {
          // Core process, showing mediating role of SUD
          auto s = message();      // I, input
          convert(s);              // P, processing
          print(s);                // O, output
```

```
            end();                    // 0, signals end of program
        }
};
```

We now specialise this class as follows; we create single input and output objects and two converter objects:

```
// Instance Systems
class MyInput
{
public:
      std::string message() const
      {
         return std::string(" Good morning ");
      }

};

class MyConverter
{
public:

      void convert(std::string& s) const
      {
         // Process string using Boost String Algorithm library
         boost::trim(s);
         boost::to_upper(s);
      }
};

class YourConverter
{
public:

      void convert(std::string& s) const
      {
         s = std::string("Sorry, it's a secret");
      }
};


class MyOutput
{
public:

      void print(std::string& s) const
      {
         std::cout << s << std::endl;
      }
```

```
        void end() const
        {
           std::cout << "end" << std::endl;
        }

};
```

A test program is:

```
// First instantiation
SUD<MyInput, MyOutput, MyConverter> sud;
sud.run();

// Second instantiation
SUD<MyInput, MyOutput, YourConverter> sud2;
sud2.run();
```

Summarising, this code is an example of *policy-based design* (PBD) that we shall introduce in Section 9.5. Properties of this solution are that specialisations of `SUD<I,O,P>` are classes and we cannot switch between different converters at run-time. Exercise 2 of this chapter discusses some extensions and modifications to the code.

9.4 CHECKLIST AND LOOKING BACK

Before proceeding to detailed design we should look back at what we have achieved. We reconsider the result and the steps that led to it. Some checklist questions are:

- Can we check the result? Can we see it at a glance?
- Can we write a C++ prototype (*proof of concept*) to motivate the system's core process?
- Can you use the result for other problems?
- Can you check system responsibilities by working backward from the output to the input (Pólya's heuristic H6) (see Exercise 1)?

Of course, we cannot hope to discover every detail and *what-if scenarios* at this early stage but we should feel confident that we are not in for some unpleasant surprises, budget overruns and code rewrites in the later phases of the software development lifecycle. This is consistent with Jackson's observations in Section 9.2.1.

9.4.1 A Special Case: Defining the System's Operating Environment

We conclude this section with a discussion of some topics that are relevant to systems that must fit into existing software and hardware environments. Not only do we need to find system interfaces but we may also need to add an extra level of indirection between the SUD and these external systems by introducing *virtual machines* that can be realised by design patterns such as *Proxy* and *Bridge* (see GOF, 1995; POSA, 1996). The sooner we know what these dependencies are, the more maintainable the resulting code will be because we know that the

operating environment will change over the lifetime of the system. In particular, the boundary between specification (the 'what') and design (the 'how') deserves attention and then *system modelling* will become part of the design process.

We summarise this section by listing the kinds of systems that emerge in practice. This activity is part of *system scoping* (Sommerville and Sawyer, 1997):

- Systems that directly interface with the SUD. This includes systems that already exist as well as planned systems that are being developed simultaneously with the SUD.
- Other systems (where known) that may coexist with the current system and defining the interactions between them.
- Related *business processes* that will eventually be integrated with systems.

It is worthwhile to examine the problem from these perspectives before moving to detailed design. We may discover functionality and requirements that would otherwise be difficult to realise in later stages of the software lifecycle. We should design with *change in mind*.

9.5 VARIANTS OF THE SOFTWARE PROCESS: POLICY-BASED DESIGN

PBD is a compile-time programming idiom in C++ based on policies. A *policy* defines a class interface or a class template interface. A policy can contain member functions, member variables and inner type definitions. Policies focus on behaviour and somewhat less on structure. In other words a policy is *syntax oriented*, by which we mean that it specifies the names and input/output parameters that conforming classes must implement. Universal function wrappers, on the other hand, are *signature oriented*. A common remark concerning policies is that some developers see them as the compile-time variant of the *Strategy* design pattern (GOF, 1995). They are closely related to *duck typing*. More precisely, policies employ parametric polymorphism while GOF design patterns employ subtype polymorphism.

Before going into detail on how to design a PBD system, we give an initial example of a GOF *Strategy* pattern, its implementation using subtype polymorphism and the corresponding implementation as a policy. We recall the definition of this design pattern:

> *Strategy defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.*

To this end, we create a function and a class that generate arrays of random numbers based on a number of generators, for example the C `rand()` function, the lagged Fibonacci generator in *Boost* and the Mersenne Twister algorithm in the C++ `<random>` library. The approach taken when applying the traditional *Strategy* pattern is to create an abstract base class containing a pure virtual member function that defines the contract between the strategy and its clients. Concrete-derived classes implement this member function.

The base class representing the strategy interface is:

```
class Rng
{
public:
    virtual double rng() = 0;
```

```
    // Function call operator; Template Method pattern
    double operator () ()
    {
         return rng();
    }
};
```

Some implementations are:

```
#include <random>
#include <cstdlib> // rand()
#include <memory>
#include <vector>
#include <algorithm>
#include <iostream>
#include <ctime>
#include <boost/random.hpp>

class InfamousRng : public Rng
{
public:
    InfamousRng()
    {
       srand(std::time(0));
    }

    double rng() override
    {
       return rand();
    }
};


class MTRng : public Rng
{ // Mersenne Twister
private:
    std::mt19937 eng;
    std::uniform_real_distribution<double> d;

public:
    MTRng() : eng(std::mt19937(std::time(0))),
              d(std::uniform_real_distribution<double>(0.0, RAND_MAX))
    {}

    double rng() override
    {
       return d(eng);
    }
};
```

```
class LFRng : public Rng
{ // Lagged Fibonacci
private:
    boost::lagged_fibonacci607 eng;
    boost::uniform_real<double> d;

    boost::variate_generator<boost::lagged_fibonacci607,
          boost::uniform_real<double>> gen;

public:
    LFRng() :
       eng(boost::lagged_fibonacci607(std::time(0))),
       d(boost::uniform_real<double>(0.0, RAND_MAX)),
       gen(boost::variate_generator<boost::lagged_fibonacci607,

       boost::uniform_real<double>>(eng, d))
       {

       }

    double rng() override
    {
       return gen();
    }
};
```

This class hierarchy uses subtype polymorphism and new generators will be derived classes of `Rng`.

The next step is to define client code that uses this hierarchy. To this end we create a class and a free function that generate arrays of random numbers. These two entities can be seen both as clients of the strategy algorithms:

```
class ArrayGenerator
{
private:
    std::shared_ptr<Rng> gen;
public:
    ArrayGenerator(const std::shared_ptr<Rng>& generator)
                          : gen(generator) {}

    std::vector<double> randomArray(unsigned N) const
    {
       std::vector<double> result(N);
       for (std::size_t i = 0; i < result.size(); ++i)
       {
          result[i] = gen->rng();
       }

       return result;
    }
```

```
    void generator(const std::shared_ptr<Rng>& generator)
    { // Choose another rn generator

        gen = generator;
    }
};

// C-style function
std::vector<double> CreateRandomArray(unsigned N, Rng& rng)
{ // Using Principle of Substitution

    std::vector<double> result(N);
    for (std::size_t i = 0; i < result.size(); ++i)
    {
        result[i] = rng();
    }

    return result;
}
```

Test code (including the ability to change the generator at run-time) is:

```
std::shared_ptr<Rng> rng(new MTRng);
ArrayGenerator gen(rng);

auto rnArr = gen.randomArray(20);

// Now switch to another generator
std::shared_ptr<Rng> rng2(new LFRng);
gen.generator(rng2);
rnArr = gen.randomArray(20);


// C function as client of GOF Strategy
std::size_t sz = 20;
InfamousRng rng3;
auto rnArr3 = CreateRandomArray(sz, rng3);
```

We now look back and examine the consequences of using this design pattern (see GOF, 1995). These consequences can be seen as an advantage or as a disadvantage depending on how the developer views the world:

1. We create families of algorithms that can be reused in multiple contexts. It avoids inheritance of classes that generate random numbers because classes use *composition* in which client context classes contain a reference to a base *Strategy* class using a shared pointer. We encapsulate algorithms in separate classes and we can vary each algorithm independently of its context.

2. Strategies provide different implementations for the *same* behaviour. We can thus use the strategy that best suits our efficiency and accuracy requirements. In this sense we call *Strategy* a *service-variation* pattern.
3. Clients must be aware of strategies. This introduces coupling because the client needs to know the name of the base class but not the name of the derived classes (in the above case the base class is called `Rng`). Each new algorithm must be implemented as a derived class of `Rng` which precludes functionality from other sources (unless we use adapters). This can be a disadvantage as it can lead to our having to use (multiple) inheritance which will lead to major maintenance problems.
4. Strategies increase the number of objects in an application. In many cases these objects must be created on the heap and then the application needs to define an object lifetime policy for them. This is an added complication and will add to development time.
5. Performance; the above code uses virtual member functions to vary behaviour which can have a performance impact in an application.

We now take the above code and adapt it to produce a solution based on the principles of PBD. In a sense it is a variation of the OOP approach. In general it is a compile-time trick (based on *parametric polymorphism*) while the above solution is based on *subtype polymorphism*.

The central idiom in PBD is to create a class template (called the *host class*) that in the most general case takes one or more type parameters as arguments. These types are specialised by *policy classes* and each one implements a particular implicit interface called a *policy*.

We have now defined enough terms to design the above problem using PBD. First, the host class is:

```
template <typename RngPolicy>
      class RandomGenerator : public RngPolicy
{ // Policy-based design random number generator.
  // This is a host class.

public:
    // Function call operator, Template Method pattern
    double operator () ()
    {
       return rng();
    }

    std::vector<double> randomArray(unsigned N)
    {
       std::vector<double> result(N);
       for (std::size_t i = 0; i < result.size(); ++i)
       {
          result[i] = rng();
       }

       return result;
    }

};
```

In this case we see that the host class includes no hard-coded class names, just a template parameter that implements an *implicit interface*, in this case the implementation of the *function call operator* (this entails that the class becomes a function object). *Any* class that implements this operator can be used as a policy. We do not force clients to inherit from some base class. This adds to the reusability and customisability of the host class. In fact, a library can be created consisting of the host class and a set of already *predefined implementations* for each policy. For example, we can define a policy that implements the `knuth_b` engine adaptor which returns shuffled sequences generated with the pseudo-random number generator engine `minstd_rand0`:

```
class KnuthRng
{ // knuth_b algorithm as a policy class
private:
    std::knuth_b eng;
    std::uniform_real_distribution<double> d;

public:
    KnuthRng() : eng(std::knuth_b(std::time(0))),
                 d(std::uniform_real_distribution<double>(0.0, RAND_MAX))
    {
    }

    double rng()
    {
       return d(eng);
    }

    // Function call operator,Template Method pattern
    double operator () ()
    {
       return rng();
    }

};
```

An example of using PBD in the current context is:

```
// Policy-based classes
RandomGenerator<KnuthRng> policyRng;

auto rnArr4 = policyRng.randomArray(100);


// Use OOP classes with policy RNG
RandomGenerator<LFRng> policyRng2;

auto rnArr5 = policyRng2.randomArray(100);
```

We note that in this it is *not* possible to change a policy of the host class at run-time, in contrast to the design based on the GOF *Strategy*. Incidentally, the code example in Section 9.3.3

was an example of PDB with three policy classes for input, processing and output. We recall that the host class in this case is defined as:

```
template <typename I, typename O, typename Processing>
    class SUD : private I, private O, private Processing
```

9.5.1 Advantages and Limitations of PBD

The term *policy-based design idiom* is discussed in Alexandrescu (2001) where it is defined and a number of examples are given to show how to use it. In a sense it has reached almost mythical status in the C++ community. The summary of PBD from Wikipedia reads:

> *A key feature of the policy idiom is that, usually (though it is not strictly necessary), the host class will derive from each of its policy classes using (public) multiple inheritance. (Alternatives are for the host class to merely contain a member variable of each policy class type, or else to inherit the policy classes privately; however inheriting the policy classes publicly has the major advantage that a policy class can add new methods, inherited by the instantiated host class and accessible to its users, which the host class itself need not even know about.) A notable feature of this aspect of the policy idiom is that, relative to object-oriented programming, policies invert the relationship between base class and derived class – whereas in OOP interfaces are traditionally represented by (abstract) base classes and implementations of interfaces by derived classes. In policy-based design the derived (host) class represents the interfaces and the policy classes implement them. It should also be noted that in the case of policies, the public inheritance does not represent ISA relationship between the host and the policy classes. While this would traditionally be considered evidence of a design defect in OOP contexts, this does not apply in the context of the policy idiom.*

We can check this description against examples. We take a critical look at PBD from both a language and a design process point of view.

- The policy interface has a direct, explicit representation in C++ code. This interface is defined implicitly (via *duck typing*); hence it is documented separately and manually using comments. Duck typing is a layer of the programming language on top of typing and it is concerned with establishing the suitability of an object for some purpose. This suitability is determined by the presence of certain methods and properties (with the appropriate semantic meaning). In other words, a programmer is only concerned with ensuring that *objects behave as demanded of them in a given context* rather than ensuring that they are of a specific class. This is a crucial observation. It implies a form of polymorphism without the need to create class hierarchies. We note that duck typing can be defined for both run-time and compile-time scenarios but a discussion is outside the scope of this book.

  Templates apply duck typing in a static typing context.
- Policy-based design may not be the best approach for all applications. It seems to be closely identified with C++ and with C++ templates in particular. Other languages, for

example C# generic interfaces with *constraints*, provide a possibly more elegant solution. For example, a simple system in C# with two policies, representing input and output, is:

```
interface IInput
{
    string message();
}

interface IOutput
{
    void print(string s);
}

// I/O stuff
class SUD<I,O>
    where I : IInput
    where O : IOutput
{

    private I i_;
    private O o_;

    public SUD(I i, O o)
    {
          i_ = i;
          o_ = o;
    }

    public void run()
    {
       o_.print(i_.message());
    }
}
```

- At the moment of writing, C++ supports neither interfaces nor constraints (or *concepts* as they are known in C++). Hence the above desirable property is not yet possible in the current version of C++. Consequently, the only solution seems to be to use public or private inheritance instead of composition.
- It is not easy to ensure in our opinion (beyond trial-and-error and iteration) how to create a good set of orthogonal policies.
- No support for *required interfaces*. This problem is caused by the fact that C++ does not have support for *events* and *delegates* (C# does support these software entities). A workaround is to use the GOF *Observe*r design pattern that implements *callbacks*. We have not tried to implement this pattern from a PBD perspective, preferring to use the Boost C++ *signals2* library solution. We shall see some examples in later chapters on Monte Carlo simulation.
- It is not clear to the author how to apply PBD to the design of parallel software systems.

9.5.2 A Defined Process for PBD

The UDP contains PBD as a special case. This means that we can resolve the issues with PBD by first analysing and designing a problem. We now give some useful guidelines:

1. Find host class and policies. Our starting point is to create the system context diagram. In general, the host class corresponds to the SUD system while policies correspond to the SUD's satellite systems.
2. Does each system in the context diagram satisfy SRP? Is there overlap in responsibilities between the systems? Determine how to remedy these problems, for example by ensuring that each system processes its own kind of data or by decomposing it into dedicated systems using the *Whole–Part design pattern* (POSA, 1996). This becomes easier if you can determine which domain category your application is an instance of because the systems have orthogonal responsibilities in those cases.
3. Determine the *provides–requires* interfaces of each system in the context diagram. Referring to Figure 9.4 we can say that systems S1, S2 and S3 provide services to the SUD while the SUD provides services to systems S4 and S5 (it is also possible that S4 and S5 require services from the SUD). In some cases system S5 could provide services to the SUD, for example the downloading of reference data that is needed by the SUD.
4. Decide how you will design the system as a prototype in C++. Review the prototype and determine if it satisfies the requirements. Asking these questions will hopefully help you to design stable software systems.

9.6 USING POLICY-BASED DESIGN FOR THE DVM PROBLEM

We now discuss how to create a simple prototype in C++ for the problem that we introduced in Section 9.3.2. It is easy to create the initial context diagram in Figure 9.3 because the DVM is a special case of an ACS whose context diagram is known. From a project management perspective we have hopefully removed a major source of risk because we have created a stable design that we can extend to satisfy current and emerging requirements.

We choose to implement the system in Figure 9.3 using C++ templates as follows:

```
// Notation: GSys == Generic system, not instantiated
template <typename GSource, typename GAuth, typename GResource,
          typename GSink, typename GMIS>

class DVM : private GSource, private GAuth, private GResource,
private GSink, private GMIS
{ // System under discussion
private:

    // C++ does not support interfaces NOR concepts (constraints)
    using GSource::message;

    using GAuth::amount;
    using GAuth::increment;
    using GAuth::decrement;
```

```
    using GResource::displayProducts;
    using GResource::getProduct;
    using GResource::updateProduct;

    using GSink::notify;

    // Data, the amount left to spend
    int amt;

    // Other members

};
```

We see that policy interfaces have been formalised and these are implemented by policy classes. We design the SUD (*mediator)* that coordinates interactions with the policy classes, as the following code shows:

```
void run()
{

   // 1. Get customer payment, Auth; Idle State,wait for coins
   std::cout << "Amount to insert: "; std::cin >> amt;
   GAuth::increment(amt);
   std::cout << "Amount in machine: " << amount();

   // 2. Choose product, Resource; Waiting for Selection State
   GResource::displayProducts();
   auto prod = getChoice();

   int change;
   try
   {
      auto info = GResource::getProduct(prod);
      std::cout << prod << " costs " << std::get<0>(info);
      change = amt - std::get<0>(info);
   }
   catch (int ex)
   {
      std::cout << "Product not found, bye\n";
      return;
   }

   // 3. Commit transaction (dispense product) and update
   // database, Resource
   GResource::updateProduct(prod, 1);
   GResource::displayProducts();

   // 4. Inform Sinks and MIS
   GSink::notify(prod, true);
```

```
    // 4A. Give change
    std::cout << "The amount of change is: " << change;
}
```

This simple *get it working* code realises the core process in the system. One approach is to model this code as a state transition table or as a UML statechart (Harel and Politi, 2000; Hatley and Pirbhai, 1988).

This concludes the design of the generic framework for the DVM application. In order to test it we need to create a number of specific policy classes. The first group is given by the following policy classes that implement the required interfaces:

```
class Auth
{ // This is where money is placed

private:
    int money_;
public:
    Auth(int money=0) : money_(money) {}

    int amount() const { return money_; }

    void increment(int amount) { money_ += amount; }
    void decrement(int amount) { money_ -= amount; }

};  // Tuple consisting of price + supply

using Product = std::tuple<int, int>;

class Resource
{ // (Simple) product database
private:
    // Name, price and supply
    std::map<std::string, Product> db;

public:
    Resource()
    {
       std::tuple<int, int> tup(2, 10);
       db.insert(std::pair<std::string, Product> ("COLA", tup));

       std::tuple<int, int> tup2(2, 20);
       db.insert(std::pair<std::string, Product>("ORANGE", tup2));
    }

void displayProducts() const
{
    std::cout << "**** Product Database **********\n";
    for (auto i = std::begin(db); i != std::end(db); i++)
    {
```

```
        std::cout << "Product: " << i->first << ", price: "
                  << std::get<0>(i->second)
                  << ", supply: " << std::get<1>(i->second);
    }
    std::cout << "*************\n";
}

std::tuple<int, int> getProduct(const std::string& product)
{
    auto prod = db.find(product);
    if (prod != db.end())
    {
       return prod->second;
    }

    throw - 1;
}

void updateProduct(const std::string& product, int quantity)
{ // Decrement supply by a given amount

    auto prod = db.find(product);
    if (prod != db.end())
    {
       std::get<1>(prod->second) -= quantity;
    }
}
};

class Sink
{ // Final recipient of end of transaction
private:

public:
    void notify(std::string prod, bool status) const
    {
       std::cout << "Transaction " << prod << ", status "
                 << std::boolalpha << status << std::endl;
    }

};


class MIS
{
public:

       // TBD $$

};
```

In this case we see that `Resource` is a very simple database and that there is only a single sink class, namely `Sink`.

An example of use is:

```
// Single sink class
DVM<Source, Auth, Resource, Sink, MIS> dvm;
dvm.run();
```

9.6.1 Introducing Events and Delegates

In some cases we wish to broadcast the end of a transaction to several sinks. C++ does not support this functionality directly and we propose alternative solutions:

a) The GOF *Observer* pattern (see GOF, 1995).
b) Using collections of universal function wrappers (class `std::function`).
c) Using Boost C++ *signals2* library (Demming and Duffy, 2010).

We discuss how to apply option c) to the current problem. To this end, we send information to the console, to a simulated local database and to a simulated remote database (for motivation only). The corresponding *slot functions* are:

```
void Print(const std::string& prod, bool status)
{ // Free function

    std::cout << "I am a printer " << prod << std::endl;
}


// Lambda function
auto database = [](const std::string& prod, bool status)-> void
                { std::cout << "In due time,I save " << prod << "to
                  database"; };

auto databaseBackup = [](const std::string& prod, bool status)-> void
                      { std::cout << "Just backuped..."; database
                        (prod,status); };
```

We now define a sink function object that plays the role of a *signal*. It triggers its connected slots when it is called:

```
class Sink2
{      // Final recipient(s) of end of transaction, now a signal;
       // configure beforehand
private:
    boost::signals2::signal<void(std::string& prod, bool status)> sig;
```

```
public:
    Sink2()
    {
       sig.connect(2,Print);
       sig.connect(0,database);
       sig.connect(1,databaseBackup);
       // add

    }

    void notify(std::string prod, bool status) const
    {
       std::cout << "\n*** Notifications ***\n";

       // Emit signal and broadcast to slots
       sig(prod, status);
       std::cout << "*** Notifications, end ***\n\n";
    }

};
```

An example of use is:

```
// Multiple sinks in Boost signals2
DVM<Source, Auth, Resource, Sink2, MIS> dvm2;
dvm2.run();
```

An important remark is that a `signal` and its slots must have the same signature.

9.7 ADVANTAGES OF UNIFORM DESIGN APPROACH

We summarise the findings and results. Some of the major advantages as we see them are:

- Separation of concerns: we avoid unnecessary and premature coding by decomposing the problem into *loosely coupled cohesive systems*. In other words, each system satisfies the SRP and it has narrow interfaces with other systems.
- We perform up-front analysis to create maintainable and reusable software systems and code.
- A *common vocabulary* is created that all team members can (hopefully) understand. Improved communication between team members during system evolution can be ensured.
- Using the design artefacts as input to project management and software risk management processes.
- Adopting an assembly mindset to glue software components to form larger systems. Deliver software incrementally as a sequence of stable prototypes.

9.8 SUMMARY AND CONCLUSIONS

In this chapter we have introduced a process to analyse, design and implement software systems of any size and any complexity. This process has been influenced by methods from *Structured Analysis*, *analogical reasoning* (techniques using domain architectures and problem frames) as well as developer experience. It allows us to design software as a sequence of stable prototypes. As a special case we formalise the policy-based design approach in C++ and we showed how our approach subsumes it. We also gave some concrete examples.

Finally, we summarise the main concepts and processes that we introduced in this chapter. The goal is to provide some guidelines and tips when creating applications. To this end, we tackle problems by the following plan:

1. Understanding the problem. In this phase we determine what the system should deliver (the output), the core input to the system and a description of the steps/operations that connect input to output. Being able to construct a data flow diagram (as in Figure 9.2) is an advantage.
2. System identification. We discover the systems and modules that realise the data flows in step 1. In particular, we create a context diagram (as in Figure 9.3). Each module has a single major responsibility and provide services to, and requires services from, other modules. It must be possible to describe the major data exchange between modules.
3. Logical interface specification. In this phase we describe the implementation-independent interfaces (an interface is a collection of abstract methods and each method has input arguments and a return type). Ideally, you should be able to describe both the *provided* and the *required* interfaces for each module.
4. Decide how to design and implement the logical interfaces. Since we are using C++ in this case (C# would offer us a different set of choices, for example) we can choose from:

   P1: Subtype polymorphism, class hierarchies and virtual member functions. This is a traditional object-oriented solution.

   P2: Modelling an interface's functions using `std::function`. This is a multiparadigm software approach because the interface's target methods can be free functions, lambda functions, function objects or object and static member functions.

   P3: Using the *policy-based design* approach based on C++ templates in combination with the *using* directive as described by the initial example in Section 9.3.3. This solution is rather limited in its scope in our opinion but it can be used when creating software prototypes.

   P4: Creating a template mediator class whose parameters implement the logical interfaces from step 3. We use *duck typing* (see Section 9.5.1) to implicitly define these interfaces when the template parameters are instantiated. This approach is basically another variant of policy-based design.

   P5: Create a single *monolithic* mediator class that implements the system core process. This option is useful for systems whose requirements are not clear or when we wish to quickly create a proof-of-concept to show to customers and clients.

5. Start designing, developing and testing the code that implements step 4. We may need to create a new system from scratch but in most cases we can use and reuse existing code from previous projects and standard libraries such as STL and Boost. The discussion in Section 9.4.1 might be relevant here.

Steps 1–5 sketch an ideal situation and in practice we see both backtracking and forward tracking between the steps taking place. It is an art as much as a science.

In later chapters we examine a fully worked-out test case based on option P2 in step 4. See also the exercises in Chapter 19 in which we discuss various design variants. In Chapter 30, Section 30.7 we show how to augment steps 1–5 above when designing multitasking and message-passing applications.

9.9 EXERCISES AND PROJECTS

1. (Brainstorming Question)

   Consider steps 1, 2, 3 and 4 in Section 9.1. Answer the following questions from the perspective of software development (preciseness is not a concern at the moment; the questions are meant to get you thinking about how to analyse and solve problems):

   a) Which steps would you execute (and how much time would you spend on each step) for the following kinds of projects?
   - Adding/modifying functionality in an existing application.
   - A project to test the feasibility of a new algorithm or model.
   - A new project that has no *precedence* (has never been done before) in your organisation.
   - Integrating third-party software into your application.

   b) When working on software projects whose functional requirements are missing, incomplete or even wrong you should determine how the following *heuristics* are useful in getting a better understanding of what needs to be designed (see Pólya, 1990, 1990A for more discussions and examples):
   - H1 *Analogy*: find a problem that is similar to the current problem and solve it.
   - H2 *Variation of the Problem*: vary or change the problem to create a new problem whose solution helps you to solve the original problem.
   - H3 *Auxiliary Problem*: reduce the scope by finding a subproblem or side problem whose solution helps you solve the original problem.
   - H4 *Precedence*: find a problem that is related to the current problem and that has been solved before.
   - H5 *Diagrammatic Reasoning*: can you draw a picture of the problem?
   - H6 *Working Backward*: start with the goal and work backward to something that is already known.
   - H7 *Decomposing and Recombining*: decompose the problem and recombine its elements in some new way.
   - H8 *Auxiliary Elements*: add a new element to the problem in order to get closer to a solution.
   - H9 *Generalisation*: find a problem that is more general than the current problem.
   - H10 *Specialisaton*: find a problem that is more specific than the current problem.

- H11 *Induction*: solve the problem by deriving a generalisation from some specific examples.

  The goal of applying these heuristics is to reduce risk and gain a clearer understanding of the problem that we are attempting to design. In general, you oscillate as it were between more general and more specialised versions of a problem. This is what can also happen in real-life software projects.

c) Consider the last software project that you worked on. Which of the heuristics in part b) did you use – either consciously or unconsciously? Which single heuristic should you have used but did not?

d) Which of the heuristics in part b) are addressed by the *Gamma* (GOF) design patterns? We recall the definition of a design pattern (see GOF, 1995):

> *In software engineering, a design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. A design pattern is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations.*

Which of the above heuristics a) are not addressed or subsumed by design patterns? As a guideline, take a pattern that you are familiar with (for example, *Strategy* as discussed in Section 9.5) and compare it with each heuristic in turn. Can you see any relationships?

2. (Extending the PBD Design of Section 9.3.3)

We ask some questions regarding the code in Section 9.3.3. The objective is to extend and modify the code in order to get hands-on experience with a small system before moving to more complex applications in computational finance that we discuss in later chapters.

Answer the following questions:

a) Generalise the code so that it works with *any data type* and *not just with strings*. Furthermore, the input data type `T1` is not necessarily the same as the output data type `T2` and hence the *converter becomes a class that maps instances of* `T1` *into instances of* `T2`. You will need to use *template–template parameters*:

```
template <typename T1, typename T2,
          template <typename T1> class Source,
          template <typename T2> class Sink,
          template <typename T1, typename T2> class Processing>

class SUD : private Source<T1>,private Sink<T2>, private
Processing<T1, T2>
{ // System under discussion

        // TBD
}
```

Implement the code and test it with scalar and composite data types.

b) In this part we implement the converter using a *universal function wrapper* that will be a data member of the SUD, namely an instance of `std::function<T2 (const T1& t)>`.

Implement the code. Is this a more flexible solution than in part a) and if so in what respect? In particular, implement the converter functionality using free functions, lambda functions, static member functions, function objects and binded member functions.

c) Modify the original code to support a new requirement: system *S1 opens a file* and processes each of its records, system S3 converts the record's strings to trimmed uppercase strings and system *S4 writes the strings into a new file*. (Refer to Figure 9.4; you can ignore systems S2 and S5 for the moment.)

3. (Extending the DVM Code of Section 9.6)

The initial code for the DVM problem needs to be extended to suit a wider range of customers. Some new features and requirements are:

a) Remote management of multiple DVM machines, for example monitoring and control of malfunctioning machines.

b) The ability to pay using credit cards.

c) Different kinds of user input panels to suit different styles.

d) The facility to choose a range of products and then pay for them as a single transaction, for example a lunch packet consisting of a drink, sandwich and chocolate bar.

Answer the following questions:

a) Determine the changes that need to be made to the context diagram in Figure 9.3 in order to accommodate these new features. For example, you may need to design new satellite systems or existing systems may need to be modified in some way. Ideally, interfaces will remain stable but this is not guaranteed. In that case you may need to create adapters (GOF, 1995).

b) Which of the following design patterns can be used to implement the required features? Proxy, Bridge, Remote Proxy, State Machine, Observer, Mediator.

c) Give a rough estimate (upper bound) of the number of classes that you will need to create in order to realise these features. Estimate how long it will take you to write and test the code.

4. (Alternative Implementation of the DVM Problem)

The solution of the DVM system in this chapter was based on C++ templates and PBD. This may not be the most suitable approach in all situations. More importantly, developers may feel more comfortable with class hierarchies and subtype polymorphism (which many C++ legacy systems use). For this reason, *implement DVM using class hierarchies and* `virtual` *member functions*. To this end, we have implemented DVM in C# using interfaces and generics and the code is almost in the form that we wish to see in C++. The interfaces (these will be *pure abstract classes* in C++) and SUD interface are:

```
// Define the interfaces/contracts
public interface ISource
{
        string message();
}

public interface IAuth
{
       int amount();
       void increment(int amount);
```

```
        void decrement(int amount);
}

public interface IResource
{
        void displayProducts();
        Tuple<int, int> getProduct(string product);
        void updateProduct(string product, int quantity);
}

public interface ISink
{
        void notify(string prod, bool status);
}

public interface IMIS
{
        void notify();
}
// ** end of interfaces

// Notation: G{Sys} == Generic system, not instantiated
public class DVM<GSource, GAuth, GResource, GSink, GMIS>
        where GSource : ISource
        where GAuth : IAuth
        where GResource : IResource
        where GSink : ISink
        where GMIS : IMIS
{

        private ISource iso_;
        private IAuth ia_;
        private IResource ir_;
        private ISink isi_;
        private IMIS im_;

        // More
}
```

Answer the following questions:

a) Implement DVM in C++ using C# as baseline requirements. Emulate C# interfaces using C++ pure abstract base classes. Test the program and check that the output is the same as before.

b) Compare all solutions in terms of efficiency, maintainability, extendibility and replaceability of components at run-time. The answers will help you decide which option is most suitable in a given context.

c) Consider the option of using C++ universal function wrappers and the Boost C++ *signals2* library as a means to allow the SUD to communicate with the external world. What are the advantages and disadvantages when compared to the other solutions?

Consider issues such as shared/non-shared state, event-driven programming, maintainability and whether data is pushed or pulled.

d) Discuss the feasibility of implementing the SUD using variadic template parameters.

5. (Which Category Does an Application Belong to?)

In this book we discuss a number of applications in computational finance. In this exercise we wish to gain *deep insights* into understanding problems even before we write a single line of code. Consider the following applications:

- A1: Using the binomial method to price one-factor options.
- A2: Creating an interpolation library.
- A3: Affine pricing models for interest-rate derivatives.
- A4: Optimisation algorithms.
- A5: Parameter estimation algorithms.
- A6: One-factor and two-factor Monte Carlo option pricers.
- A7: Computing the mean error of a finite difference approximation of an SDE (see Kloeden, Platen and Schurz, 1997, p. 118).
- A8: Option pricing using PDE models.
- A9: Postprocessing routines to test the accuracy of a numerical scheme by comparison with a method that produces reference and exact values.
- A10: Creating data and modules to be used by other systems.
- A11: Configuring a complete application.

Answer the following questions:

a) Determine the category (or categories) that these applications are instances of. In general, the most important categories are MAN, RAT and MIS.

b) Create a basic system context diagram for these applications by identifying the systems and their responsibilities (use Figure 9.4 as reference model).

c) Identify the similarities and differences between the system context diagrams in the case of PDE, binomial and Monte Carlo option pricers.

d) Can you identify reusable artefacts such as code, libraries, data types and containers that can be used in multiple applications, for example the applications in part c)?

6. (Brainstorming, Starting a Software Project)

The next three exercises are meant to stimulate discussion on scoping and designing a software system and then packaging the code so that it can be used by clients. The intent will hopefully become even more clear as we progress through the book as many of the applications and examples adopt the same strategies. We are assuming in all cases that the five steps summarised in Section 9.8 will be adhered to for the purposes of the three exercises. The answers in all cases will tell us how to use and/or customise these steps to fulfil the given requirements.

The goal in this exercise is to scope the project and remove major ambiguities and risks before jumping into implementation details.

Answer the following questions by stating how to use the bespoke steps for the following requirements:

a) How to scope a project whose requirements are new to you (no *precedence*).

b) Create a *proof-of-concept* program to demonstrate to clients. You have a week in which to write the prototype.

c) *Project estimation*: estimate the numbers of hours to design the modules in the system context diagram.

d) Prioritising the modules: which ones to design now and which ones to design later?

7. (Brainstorming, Design and Coding)

The focus in this exercise is to decide which of the design approaches 1, …, 5 from Section 9.8 in combination with modern C++ features to use for the following cases in which certain quality characteristics are hard requirements:

a) Efficiency and run-time performance. This includes computing times and the amount of resources used.

b) Interoperability (interfacing with existing and future software systems). These systems are not necessarily based on the object-oriented paradigm.

c) Portability (adaptability of software to work in different specified environments and the ability to install software in a specified environment).

d) Maintainability (for example, the effort that is needed to modify the code, remove faults or adapt to environmental change).

e) You may find that your initial software does not satisfy the above requirements. Then changes need to be made to the modules in the design. What are the software entities (interfaces, data types, algorithms, classes) that might have to be changed?

8. (Brainstorming, Configuration and Packaging)

Once code has been written we need to test it and deliver it to clients. Determine how to deliver the following functionality:

a) Separation of configuration code (for example, object/data creation and initiation) from application code.

b) Testing the full program.

c) Creating a directory structure to contain the code files. Standardise file names.

9. (Determining the Requirements)

This exercise attempts to simulate a discussion on how to discover and document system requirements for a new software product (see Sommerville and Sawyer, 1997 on the skills that are needed for requirements elicitation, analysis, negotiation and documentation). In order to make the discussion relevant to computational finance we consider the problem of creating a software framework for a one-factor Black–Scholes PDE and its generalisations. We are interested in approximating these PDEs by the FDM although this exercise could be extended to any application in principle.

We simulate an initial roundtable discussion with the following possible stakeholders attending:

- Architect: responsible for specifying the system to be designed.
- Domain expert: in-depth knowledge of the material and how the system will be used in the organisation.
- Project manager: responsible for the successful execution of the software project.
- Sponsor: This could be an internal or external stakeholder and is often the stakeholder who pays for the costs of the project as well as developers' salaries.
- User of the system: This is the person or group who actually works with the computer system on a daily basis.

The main goal of this *simulated discussion* is to reach consensus among the stakeholders on how to initiate the software project and bring it to a successful conclusion. We are also interested in flagging the assumptions and potential risks inherent in this kind of work. At some stage we would hope that the system decomposition methodology in this chapter will help us focus and avoid getting bogged down in endless discussions.

Answer the following questions:

a) What are the major risks and challenges that must be resolved before proceeding? (See Sommerville and Sawyer, 1997 for a discussion of some topics to help discover these risks.)

b) For high-risk projects what would be your approach to create a proof-of-concept software prototype? Have you made it clear what the objective of the prototype is and what the next phase is?

c) Can we use some of Pólya's heuristics (see Exercise 1) to help the stakeholders view the project from different perspectives? Can you use domain architecture categories to improve communication among the stakeholders?

d) Assuming consensus has been reached, determine how the project manager will produce a project plan including an estimate of how long each phase of the project will take.

e) Based on the discussions and findings, determine which design style, patterns, language features and libraries the architect and her team will use to produce a series of software prototypes on time and within budget.

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 10

New Data Types, Containers and Algorithms in C++ and Boost C++ Libraries

10.1 INTRODUCTION AND OBJECTIVES

In this chapter we introduce a number of features and libraries in C++ (and Boost) that we shall use throughout this book. In general, we use these libraries as utilities and building blocks in a range of applications. We discuss the essential functionality that is relevant to the topics in this book while functionality that we do not need will not be included. For more information on the Boost C++ libraries, see Demming and Duffy (2010, 2012).

In general, we complete our discussion of C++ syntax and functionality in this chapter before proceeding to applications beginning in Chapter 11.

10.2 OVERVIEW OF NEW FEATURES

In this chapter we assemble a number of data types and containers in one place. They are useful building blocks that we can embed and use in code using modelling techniques such as *composition* and *inheritance*. In particular, we focus on compile-time and run-time matrices that play a fundamental role in scientific applications. We shall need them in later chapters:

- The binomial and trinomial methods.
- Partial differential equations and the finite difference method.
- Creating lookup tables for statistical distributions.
- Using 'bit matrices' as components in the solution of nonlinear systems of equations, for example using *Differential Evolution* (see Price, Storn and Lampinen, 1996).

In most cases we are interested in working with matrices whose elements are of floating-point type. An important use case is to determine the run-time efficiency of matrix operations. To this end, we use the functions in the C++ `<chrono>` to measure processing time.

An important addition to C++ is the ability to model sets of bits using the `std::bitset<>` library. This library has many applications. See also Exercise 2 in the current chapter.

10.3 C++ std::bitset<N> AND BOOST DYNAMIC BITSET LIBRARY

We discuss bitsets in this section. They model *fixed-size arrays* (in the case of STL) and *dynamic arrays* (in the case of Boost). Bitsets are useful when we wish to manage sets of flags and when we use bitwise operations (such as AND, OR) on these sets.

We first discuss the STL class `bitset<N>`. This class models fixed-size arrays of bits. The number of bits `N` is defined at compile-time and this value cannot be changed thereafter. The function categories in `bitset` are:

- Constructors: default constructors and constructors based on integer and string input arguments.
- Querying if one, some or all bits in a bitset have been set; comparing bitsets for equality or inequality.
- Setting, resetting and flipping the bits in a bitset.
- Bitwise operations on bitsets (and, or, nor, left and right shift, toggle bits).
- Type conversions: converting bitsets to `string` and to `long`.
- Counting the number of bits that have been set in a bitset.

We give some examples of how to use class `bitset`. We first create a default bitset, a bitset that we initialise using an integral value and then we create a bitset from a string:

```
#include <bitset>

// Constructors.
const int N = 10;
bitset<N> bsA;           // All bits set to 0.
cout << bsA << endl;     // Prints 10 zeroes.

int val = 7;
bitset<N> bsB(val);      // Init bitset with val's bit representation.
cout << bsB << endl;     // The binary form of val, i.e. 0000...111.

// Constructing bitsets based on strings.
bitset<N> bsC(string("0101010101"));     // String of length 10.
cout << bsC << endl;
```

Next, we create bitsets by extracting certain bits from input strings. In the first case we create a bitset of length 7 by extracting information starting at the third bit (index = 2) and continuing to the end of the string:

```
// Bitsets with start pos and/or size.
int index = 2;                              // 3rd bit, indexing
                                            // starts at 0.
bitset<7> bsD(string("1111000"), index);    // From index to end.
cout << "Bitset with start index: " << bsD << endl;
```

The output in this case is `'0011000'`. We note that extra bits are *padded* from the left. The next example extracts 4 bits from the beginning of the input string:

```
int num_bits = 4;
index = 0;
bitset<7> bsE(string("10111000"), index, num_bits);
cout << "Bitset with start index and number bits: " << bsE << endl;
```

The output in this case is `'0001011'`. In both of the above cases we note that the bitset's elements are constructed *from right to left*.

We now discuss the operations that modify the bits in a bitset in some way. For example, we can set all the bits to `true`, set all the bits to a given value, set all the bits to `false`, flip all the bits and flip at a certain position:

```
const int SZ = 4;
bitset<SZ> bsX(string("1010"));

bsX.set();                  // Set all bits to true

// Set all the bits to bitValue.
bool bitValue = 1;   // True.
for (size_t j = 0; j < bsX.size(); ++j)
{
   bsX.set(j, bitValue);
}

bsX.reset();                // Set all bits to 0 (false).
bsX.flip();                 // Set unset bits and vice versa.
bsX.flip(2);                // Toggle bit at position 2.
```

We have also created two utility functions to print the contents of a bitset and to compare two bitsets:

```
template <int N>
   void ExamineBitset(const bitset<N>& bs)
{
   cout << "Number of total bits: " << bs.size() << endl;
   cout << "Number of set bits: " << bs.count() << endl;
   cout << "Is any bit set?: " << boolalpha << bs.any() << endl;
   cout << "Is no bit set?: " << boolalpha << bs.none() << endl;
}

template <int N>
   void CompareBitsets(const bitset<N>& bs1, const bitset<N>& bs2)
{
   cout << "Bitsets equal?: " << boolalpha << (bs1 == bs2) << endl;
   cout << "BitSets not equal?: " << boolalpha << (bs1 != bs2) << endl;
}
```

You can test these functions and examine the output.

10.3.1 Boolean Operations

We can perform bitwise binary and unary operations on bitsets. There are two binary forms, namely when the left-hand operand is modified and the second form is when two bitsets combine to form a new bitset. We take some examples of both forms:

```
const int M = 4;
bitset<M> bs1(string("1010")); cout<<"bs1: "<<bs1<<endl;
bitset<M> bs2(string("0011")); cout<<"bs2: "<<bs2<<endl;

// Boolean operators, non-modifying.
cout<<"~bs1: "<<~bs1<<endl;                        // Toggle all bits.
cout<<"bs1 ^ bs2: "<<(bs1 ^ bs2)<<endl;            // Bitwise XOR.
cout<<"bs1 | bs2: "<<(bs1 | bs2)<<endl;            // Bitwise OR.
cout<<"bs1 & bs2: "<<(bs1 & bs2)<<endl;            // Bitwise AND.

// Boolean operators, modifying.
bs1 ^= bs2;  cout<<"bs1 ^= bs2: "<<bs1<<endl;      // XOR .
bs1 |= bs2;  cout<<"bs1 |= bs2: "<<bs1<<endl;      // Bitwise OR.
bs1 &= bs2;  cout<<"bs1 &= bs2: "<<bs1<<endl;      // Bitwise AND.
```

Which option to choose depends on the application, but in general the 'modifier' form is more efficient than the non-modifying form.

We can also perform shift operations on bitsets. Again, there are two forms as in the previous case:

```
// Shift operators, non-modifying. Always zero's are shifted in.
int n=1;
bs1=bitset<M>(string("1001")); cout<<"bs1: "<<bs1<<endl;
cout<<"bs1>>n: "<<(bs1 >> n)<<endl;          // Shift n bits to the right.
cout<<"bs1<<n: "<<(bs1 << n)<<endl;          // Shift n bits to the left.

// Shift operators, modifying. Always zero's are shifted in.
bs1=bitset<M>(string("1001")); cout<<"bs1: "<<bs1<<endl;
bs1 >>= n; cout<<"bs1 >>= n: "<<bs1<<endl;   // Shift n bits to the right.
bs1 <<= n; cout<<"bs1 <<= n: "<<bs1<<endl;   // Shift n bits to the left.
```

10.3.2 Type Conversions

We conclude our discussion of `std::bitset` by showing how to return the integral and string values that a bitset represents. To this end, we use the member functions `to_ulong()` and `to_string()`. Here is an example:

```
// Type conversions.
const int P = 4;
bitset<P> bs(string("1011"));
```

```
// The integral value corresponding to the bits.
unsigned long value = bs.to_ulong();
string stringValue = bs.to_string();

cout << "Numeric value: " << value << ", String value: "
     << stringValue << endl;
```

10.3.3 Boost `dynamic_bitset`

This class is almost identical to `std::bitset` and it has an interface similar to that of `std::bitset`. There is, however, one major difference; in the current case we can create bitsets of dynamic length at run-time. This is achieved in the constructors of `dynamic_bitset<Block, Allocator>`. It has two template parameters:

- *Block*: the integer type in which the bits are stored (default is `unsigned long`).
- *Allocator*: the allocator type used for all internal memory management (default is `std::allocator<Block>`).

An example of use is:

```
#include <iostream>
#include <boost/dynamic_bitset.hpp>

using namespace std;

int main()
{
    boost::dynamic_bitset<unsigned int> x(4);  // All 0's by default.
    x[0] = 1;
    x[1] = 1;
    x[3] = 1;
    cout << x  << endl; // '1011'.

    boost::dynamic_bitset<unsigned int> y(4);
    cout << y << endl;  // '0000'.
    y = x;
    cout << y << endl;  // '1011'.

    return 0;
}
```

The other member functions are the same as those for `std::bitset`.

10.3.4 Applications of Dynamic Bitsets

There are many applications in which dynamic bitsets can be used. In general, their use results in efficient and compact code. Some applications are:

- Creating efficient and compact data structures.
- Memory allocation managers.

- Huffman coding, Bloom filters.
- Information retrieval.
- Raster graphics when we model pixels and bitmaps.
- Genetic algorithms, especially when modelling *selection*, *crossover* and *mutation* operations.
- When implementing security and access control policies.
- It can be an alternative to, and improvement on, `std::vector<bool>`.

Discussion of these topics is outside the scope of this book. See Demming and Duffy (2012) for more examples.

10.4 CHRONO LIBRARY

The *chrono* library provides a means to represent points in time, time durations (for example, days, minutes, seconds or nanoseconds) and several kinds of clocks (for example, system clock and high-resolution clock). It is a precision-neutral library and it is an improvement (more portable) on previous attempts to create flexible and accurate libraries to measure time.

What do we mean by 'time'? There are various definitions and perspectives:

- *Elapsed real time* (real time): the time taken from the start of a computer program to its end. It includes I/O time and other kinds of wait. Real time is the time measured by an ordinary clock.
- *CPU* (process) time: the amount of time that a CPU uses for processing the instructions of a computer program or of an operating system as opposed to measuring I/O operations. CPU time is usually measured in *clock ticks* or seconds. CPU time can further be divided into *user time*, which is the amount of time that the CPU was busy executing code in *user space* and *system time*, which is the amount of time that the CPU was busy executing code in *kernel space*. We can also measure CPU time as a percentage of CPU capacity and this is called *CPU usage*. In a multiprocessor machine the total CPU time is the sum of the CPU times consumed by all of the CPUs utilised by a computer program.
- *Wall-clock time* (wall time): the human perception of the passage of time from the start of a task to completion. It is the sum of CPU time, I/O time and the communication channel delay.
- *System time*: the computer system's notion of the passing of time. It also includes the passing of days on the calendar.

We now discuss in some detail how the chrono library measures time. We first introduce the supporting class for fractional arithmetic.

10.4.1 Compile-Time Fractional Arithmetic with `std::ratio<>`

The class template `std::ratio<>` supports compile-time rational arithmetic. A rational number is a pair consisting of a *numerator* and a non-zero *denominator*, both of which must

be compile-time constants of type `std::intmax_t`. Both numerator and denominator are reduced to the lowest terms. For example, the fraction 2/10 is reduced to the fraction 1/5.

Creating a compile-time fraction entails instantiating the template parameters of `std::ratio<>`. Some examples are:

```
// Ratio 101; ratio<N> == ratio<N,1>
std::ratio<10> r1; print(r1);              // 10/1
std::ratio<10,2> r2; print(r2);            // 5/1
std::ratio<2,10> r3; print(r3);            // 1/5

template <std::intmax_t Num, std::intmax_t Den>
   void print(const std::ratio <Num, Den>& r)
{
   std::cout << r.num << "/" << r.den << '\n';
}
```

Ratio types can be added, subtracted, multiplied and divided:

```
// Arithmetic operations on ratio
std::ratio<6,21> rA; print(rA);           // 2/7
std::ratio<7, 21> rB; print(rB);          // 1/3

using Add = std::ratio_add <std::ratio<6, 21>, std::ratio<7, 21>>;
Add rC; print(rC);                        // 13/21

using Multiply = std::ratio_multiply
   <std::ratio<6, 21>, std::ratio<7, 21>>;
Multiply rM; print(rM);                   // 2/21

using Subtract = std::ratio_subtract
   <std::ratio<6, 21>, std::ratio<7, 21>>;
Subtract rS; print(rS);                   // -1/21

using Divide = std::ratio_divide
   <std::ratio<6, 21>, std::ratio<7, 21>>;
Divide rD; print(rD);                     // 6/7
```

Furthermore, standard comparison operators are defined for ratio types, for example:

```
// Comparisons
if (std::ratio_equal<std::ratio<2, 3>, std::ratio<4, 6>>::value)
{
   std::cout << "2/3 == 4/6\n";
}
else
{
   std::cout << "2/3 != 4/6\n";
}
```

Other operators are:

```
ratio_not_equal
ratio_less
ratio_less_equal
ratio_greater
ratio_greater_equal.
```

Finally, the standard library supports SI (*International System of Units*) ratios (all in the namespace std):

```
typedef ratio<1, 1000000000000000000> atto;
typedef ratio<1, 1000000000000000> femto;
typedef ratio<1, 1000000000000> pico;
typedef ratio<1, 1000000000> nano;
typedef ratio<1, 1000000> micro;
typedef ratio<1, 1000> milli;
typedef ratio<1, 100> centi;
typedef ratio<1, 10> deci;
typedef ratio<10, 1> deca;
typedef ratio<100, 1> hecto;
typedef ratio<1000, 1> kilo;
typedef ratio<1000000, 1> mega;
typedef ratio<1000000000, 1> giga;
typedef ratio<1000000000000, 1> tera;
typedef ratio<1000000000000000, 1> peta;
typedef ratio<1000000000000000000, 1> exa;
```

Some examples are:

```
template <std::intmax_t Num, std::intmax_t Den>
    void print(const std::ratio <Num, Den>& r)
{
    std::cout << r.num << "/" << r.den << '\n';
}

// SI units
std::kilo myKilo; print(myKilo);           // 1000/1
std::milli myMilli; print(myMilli);        // 1/1000

std::nano myNano; print(myNano);           // 1/1'000'000'000
std::giga myGiga; print(myGiga);           // 1'000'000'000/1
```

The chrono library uses `std::ratio<>`.

10.4.2 Duration

A *duration* of time is defined as a number of ticks over a certain time unit. For example, 10 minutes is 10 ticks of a minute or 86,400 seconds, which is 86,400 ticks of a second (otherwise known as one day). Some initial examples of durations are:

```
namespace ch = std::chrono;

// 5 ticks of 1/10 second
ch::duration<int, std::ratio<1, 10>> d1(5);
// 4 ticks of 1/20 second
ch::duration<int, std::ratio<1, 20>> d2(4);
std::cout << d1.count() << ", " << d2.count() << '\n';  // 5,4
```

Other examples using SI units are:

```
// SI units
std::kilo myKilo; print(myKilo);            // 1000/1
std::milli myMilli; print(myMilli);         // 1/1000

std::nano myNano; print(myNano);            // 1/1'000'000'000
std::giga myGiga; print(myGiga);            // 1'000'000'000/1
```

Other helper predefined duration types are:

```
std::chrono::nanoseconds
    duration</*signed integer type of at least 64 bits*/, std::nano>
std::chrono::microseconds
    duration</*signed integer type of at least 55 bits*/, std::micro>
std::chrono::milliseconds
    duration</*signed integer type of at least 45 bits*/, std::milli>
std::chrono::seconds
    duration</*signed integer type of at least 35 bits*/>
std::chrono::minutes
    duration</*signed integer type of at least 29 bits*/, std::ratio<60>>
std::chrono::hours
    duration</*signed integer type of at least 23 bits*/, std::ratio<3600>>
```

Using these types is more user-friendly than using raw fractions:

```
// Duration typedefs
ch::duration<double, std::milli> mySec(30);
std::cout << mySec.count() << '\n';             // 30

std::chrono::hours h(1);                        // one hour
std::chrono::milliseconds ms{ 3 };              // 3 milliseconds
std::chrono::duration<int, std::kilo> ks(3);    // 3000 seconds
```

We now come to the topic of arithmetic duration operations and properties:

- Sum, difference, product and division of two durations using operator notation.
- Add and subtract ticks and other durations.
- Compare two durations (==, < , <=, >, >=).
- Convert a duration into another type.
- Minimum and maximum possible durations for a type.

Some typical examples are:

```
// Operations
ch::seconds s(100);                                          // 100 seconds
ch::milliseconds mills(10);     // 10 miliseconds
auto diff = s - mills;
std::cout << diff.count() << '\n';                           // 99990
diff++;
std::cout << diff.count() << '\n';                           // 99991
auto sum = s + mills;
std::cout << sum.count() << '\n';                            // 100010
std::cout << std::boolalpha << (sum == diff) << '\n';        // false

ch::hours aDay(24);
ch::milliseconds zeroMS(0);
zeroMS += aDay;
std::cout << zeroMS.count() << '\n';                         // 86.400,00
```

Finally, `duration_cast` converts a duration to a duration of a different type. Computation is done in the widest type available and converted to the final result. Here is an example of converting from milliseconds to seconds and between different ratios:

```
// Duration casting
ch::duration<double, std::milli> dA(3000);          // 3000 ms
auto dB = ch::duration_cast<ch::seconds>(dA);
std::cout << dB.count() << '\n';                    // 3

ch::duration<int, std::ratio<1, 10>> dC(5);
std::cout << dC.count() << '\n';                    // 5
auto dD = ch::duration_cast<ch::duration<int, std::ratio<1, 20>>> (dC);
std::cout << dD.count() << '\n';                    // 10
auto dE = ch::duration_cast<ch::duration<int, std::ratio<1, 2>>> (dC);
std::cout << dE.count() << '\n';                    // 1
```

10.4.3 Timepoint and Clocks

A *timepoint* represents a specific point in time. A *clock* defines an *epoch* (starting point of the clock). A timepoint associates a positive or negative duration with a given clock. For example, if the epoch is January 1, 1970 for a duration of 10 days then the corresponding timepoint will be January 11, 1970.

The C++ standard library supports three clocks:

- `system_clock`: this is a system-wide real-time wall clock. It also has convenience functions `to_time_t()` and `from_time_t()` to convert between timepoints and the C system type `std::time_t` (that in most cases is the number of seconds since 00:00, January 1, 1970 UTC).
- `steady_clock`: this represents a *monotonic clock*. The timepoints of this clock cannot decrease as physical time moves forward. It is suitable for measuring intervals.
- `high_resolution_clock`: this is the clock with the smallest tick period.

In this book we focus on `system_clock` because we are mainly interested in evaluating algorithm performance. We take an example involving timepoints, a clock and the C system type `std::time_t`:

```
void SystemClockExample()
{
   // System clock and timepoints
   std::chrono::system_clock::time_point start;
   std::chrono::system_clock::time_point end;

   start = std::chrono::system_clock::now();
   end = start - std::chrono::hours(24);

   // Convert and display C time
   std::time_t cTime = std::chrono::system_clock::to_time_t(end);
   std::cout << "24 hours ago, time was: "
             << std::put_time(std::localtime(&cTime), "%F %T") << '\n';
   std::cout << "Yesterday: " << ctime(&cTime) << '\n';
}
```

10.4.4 A Simple Stopwatch

After having introduced the main features of the `<chrono>` library we now discuss how and where to use it in code. In this book we shall need them to measure the run-time performance of algorithms, for example. Instead of cluttering up code we decide to create a class that does the same thing as the above code but whose interface is easier to use. The specification is:

```
template <typename TickType = double,
typename UnitType = std::ratio<1,1>>
   class StopWatch {
private:
   std::chrono::system_clock::time_point start;
   std::chrono::system_clock::time_point end;

   bool isStart;
   bool isEnd;
```

```
public:
   StopWatch() :isStart(false), isEnd(false) {}
   StopWatch(const StopWatch<TickType, UnitType>& src) = delete;
   StopWatch<TickType, UnitType>& operator
      = (const StopWatch<TickType, UnitType>& src) = delete;

   void Start();
   void Stop();
   void Reset();   // Reset the time to NOW

   // Duration between start and stop in seconds; reset
   TickType GetTime();
};
```

The code body is:

```
template <typename TickType,typename UnitType>
   void StopWatch<TickType,UnitType>::Start()
{
   start = std::chrono::system_clock::now();
   isStart = true;
}

template <typename TickType,typename UnitType>
   void StopWatch<TickType,UnitType>::Stop()
{
   end = std::chrono::system_clock::now();
   isEnd = true;
}

template <typename TickType,typename UnitType>
   void StopWatch<TickType,UnitType>::Reset()
{
   start = std::chrono::system_clock::now();
   isStart = false;
   end = std::chrono::system_clock::now();
   isEnd = false;
}

template <typename TickType,typename UnitType>
   TickType StopWatch<TickType,UnitType>::GetTime()
{
   TickType count;
   if (isStart && isEnd)
   {
      // Seconds as unit type (default)
      std::chrono::duration<TickType, UnitType> D = end - start;
      count = D.count();
   }
   else
```

```
    {
       std::cout
          << "Start time point/end time point not recorded\n";
       count = -1;
    }

    // Reset all counters
    Reset();

       return count;
}
```

An example of use is:

```
// Stopwatch
StopWatch<> sw;
StopWatch<double, std::ratio<1,1000>> sw2;
sw.Start();
sw.Stop();
std::cout << "Elapsed time: " << sw.GetTime() << '\n';

sw.Start();
sw2.Start();
long N = 1'000'000'000;
for (long n = 1; n <= N; ++n) { double d = static_cast<double>(n); }
sw.Stop();
sw2.Stop();
std::cout << "Elapsed time: " << sw.GetTime() << '\n';
std::cout << "Elapsed time: " << sw2.GetTime() << '\n';
```

We shall use this stopwatch class in later chapters when we wish to measure the run-time performance of applications.

10.4.5 Examples and Applications

We now give a small application: we use our freshly minted stopwatch class to measure the run-time performance of algorithms. As an example, we measure the running time of operations related to two-dimensional square matrices and we focus on the use cases:

- U1: Creating and initialising a matrix.
- U2: Matrix multiplication.

Next, there are many ways to model matrices and use matrix libraries in C++; in this section we focus on three options:

- M1: Using raw C++ pointers.
- M2: Using nested vectors.
- M3: Using the Boost uBLAS matrix library (Demming and Duffy, 2012).

Furthermore, the processor and library options are:

- C1: Single-threaded code.
- C2: Multithreaded code using OpenMP (Chapman, Jost and Van der Pas, 2008).
- C3: Using the Microsoft Parallel Patterns Library (PPL) (Campbell and Miller, 2011).

We concentrate on options C1, C2 and C3. In later chapters we shall discuss *C++ Concurrency* but it has no direct support for *loop parallelisation* at the moment of writing (while options C2 and C3 have this support). We use matrices as containers for holding data and we can define matrix algorithms such as solving tridiagonal systems and performing Cholesky decomposition. In order to avoid an explosion in the amount of code to be shown in this section we focus on option M3, as the code for the other two cases is similar.

For each matrix type we define a data structure to store the data in. The choices are:

```
// ADT: Nested SQUARE matrix
using NestedMatrix = std::vector<std::vector<double>>;

// ADT: Boost SQUARE matrix
using BoostMatrix = boost::numeric::ublas::matrix<double>;
using BoostMatrixR
   = boost::numeric::ublas::matrix<double, boost::numeric::ublas::
   row_major>;
using BoostMatrixC
   = boost::numeric::ublas::matrix<double, boost::numeric::ublas::
   column_major>;
```

Apart from showing how the `<chrono>` library works to measure the run-time efficiency of algorithms it is important to determine how to design data structures for matrices. Critical questions are:

- Memory usage.
- Run-time efficiency of basic BLAS operations, for example matrix multiplication.
- Numerical linear algebra.

We test the run-time performance of the various options discussed above. There is much common code and it is for this reason that we focus on the Boost case, in particular sequential, PPL and OpenMP:

```
void BoostMatrixMultiply(const BoostMatrix& m1, const BoostMatrix& m2,
BoostMatrix& m3)
{  // Matrix multiplication in Boost

   // Assume 'compatibility' for multiplication of two matrices; OK since
   // matrices are square.

   double temp;
```

```
    for (std::size_t i = 0; i < m3.size1(); ++i)
    {
       for (std::size_t j = 0; j < m3.size2(); ++j)
       {
          temp = 0.0;
          for (std::size_t k = 0; k < m1.size2(); ++k)
          {
             temp += m1(i,k) * m2(k,j);
          }
             m3(i,j) = temp;
       }
    }
}

void BoostParallelMatrixMultiply(const BoostMatrix& m1,
                            const BoostMatrix& m2, BoostMatrix& m3)
{  // Matrix multiplication in Boost

   // Assume 'compatibility' for multiplication of two matrices; OK since
   // matrices are square.

   double temp;

   concurrency::parallel_for(std::size_t(0), m1.size1(),[&](std::size_t i)
   {
      for (std::size_t j = 0; j < m3.size2(); ++j)
      {
         temp = 0.0;
         for (std::size_t k = 0; k < m1.size2(); ++k)
         {
            temp += m1(i, k) * m2(k, j);
         }
         m3(i, j) = temp;
      }
   });
}


void omp_BoostParallelMatrixMultiply(const BoostMatrix& m1,
   const BoostMatrix& m2, BoostMatrix& m3)
{ // Matrix multiplication in Boost

  // Assume 'compatibility' for multiplication of two matrices; OK since
  // matrices are square.

   double temp;

#pragma omp parallel for
    for (long i = 0; i < m1.size1(); ++i)
```

```
    {
       for (long j = 0; j < m3.size2(); ++j)
       {
          temp = 0.0;
          for (long k = 0; k < m1.size2(); ++k)
          {
             temp += m1(i, k) * m2(k, j);
          }
          m3(i, j) = temp;
       }
    };
}
```

A test case using the stopwatch class is:

```
    const size_t N = 500; // and 1000, 2000

    {
       StopWatch<> sw;
       sw.Start();
           BoostMatrix A(N,N);
           BoostMatrix B(N, N);
           BoostMatrix C(N, N);
       sw.Stop();
       std::cout << "Elapsed time, Boost creaton: "
                 << sw.GetTime() << '\n';

       sw.Reset();
       sw.Start();
       BoostMatrixMultiply(A, B, C);

       sw.Stop();
       std::cout << "Elapsed time, Boost sequential multiplication: "
                 << sw.GetTime() << '\n';

       sw.Reset();
       sw.Start();
       BoostParallelMatrixMultiply(A, B, C);
       sw.Stop();
       std::cout << "Elapsed time, Boost parallel multiplication PPL: "
                 << sw.GetTime() << '\n';

       sw.Reset();
       sw.Start();
       omp_BoostParallelMatrixMultiply(A, B, C);
       sw.Stop();
       std::cout << "Elapsed time, Boost parallel multiplication OMP: "
                 << sw.GetTime() << '\n';
    }
```

We tested the options on square matrices of size 500, 1000 and 2000 (values are in seconds):

```
Elapsed time, Raw creation: 0.0110011
Elapsed time, Raw sequential multiply: 0.629063
Elapsed time, Raw parallel multiply PPL: 0.487049
Elapsed time, Raw parallel multiply OpenMP: 0.79708
Elapsed time, Nested creation: 0.0030003
Elapsed time, Nested sequential multiplication: 0.390039
Elapsed time, Nested parallel multiplication PPL: 0.876088
Elapsed time, Nested parallel multiplication OpenMP: 0.648065
Elapsed time, Boost creation: 0.0010001
Elapsed time, Boost sequential multiplication: 0.20302
Elapsed time, Boost parallel multiplication PPL: 1.43914
Elapsed time, Boost parallel multiplication OMP: 0.546055
Elapsed time, Duffy creation: 0.0040004
Elapsed time, Duffy sequential multiplication: 2.11721
Elapsed time, Duffy parallel multiplication OMP: 1.89219
Elapsed time, Raw creation: 0.0360036
Elapsed time, Raw sequential multiply: 15.4175
Elapsed time, Raw parallel multiply PPL: 4.64546
Elapsed time, Raw parallel multiply OpenMP: 4.88649
Elapsed time, Nested creation: 0.0070007
Elapsed time, Nested sequential multiplication: 18.1538
Elapsed time, Nested parallel multiplication PPL: 8.56186
Elapsed time, Nested parallel multiplication OpenMP: 5.45755
Elapsed time, Boost creation: 0
Elapsed time, Boost sequential multiplication: 13.0693
Elapsed time, Boost parallel multiplication PPL: 12.5833
Elapsed time, Boost parallel multiplication OMP: 5.09251
Elapsed time, Duffy creation: 0.0210021
Elapsed time, Duffy sequential multiplication: 34.8985
Elapsed time, Duffy parallel multiplication PPL: 0
Elapsed time, Duffy parallel multiplication OMP: 21.8812
Elapsed time, Raw creation: 0.169017
Elapsed time, Raw sequential multiply: 96.8327
Elapsed time, Raw parallel multiply PPL: 47.2557
Elapsed time, Raw parallel multiply OpenMP: 43.8634
Elapsed time, Nested creation: 0.0890089
Elapsed time, Nested sequential multiplication: 107.685
Elapsed time, Nested parallel multiplication PPL: 57.9658
Elapsed time, Nested parallel multiplication OpenMP: 44.6464
Elapsed time, Boost creation: 0
Elapsed time, Boost sequential multiplication: 143.671
Elapsed time, Boost parallel multiplication PPL: 106.551
Elapsed time, Boost parallel multiplication OMP: 46.0605
Elapsed time, Duffy creation: 0.0900009
Elapsed time, Duffy sequential multiplication: 221.928
Elapsed time, Duffy parallel multiplication OMP: 156.694
```

We run this code on a 2-core processor with hyperthreading. We can compare and contrast the different outputs. We see in particular that the complexity of the class structure in

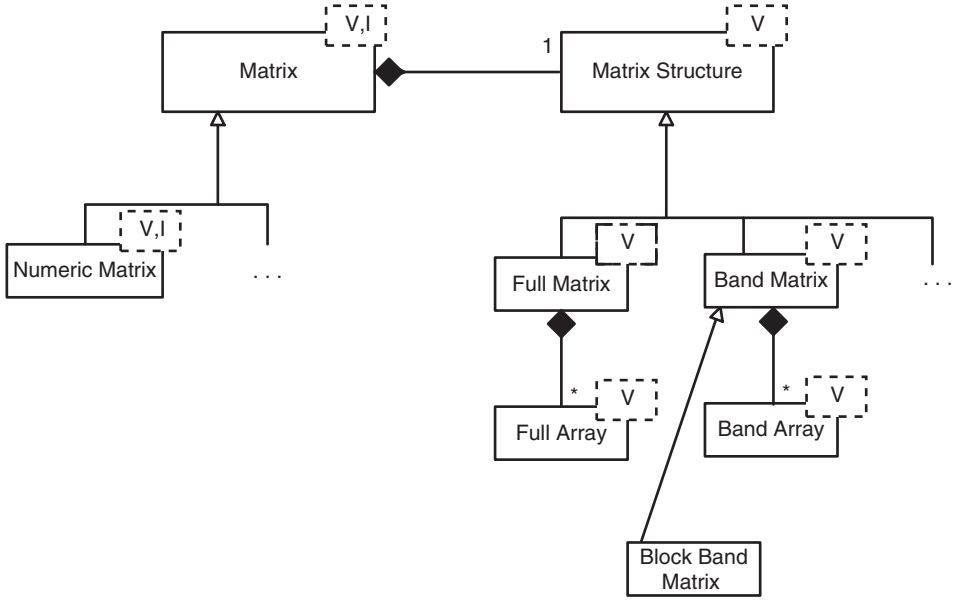


FIGURE 10.1 General matrix and numeric matrix classes

Figure 10.1 negatively affects performance when compared to the other solutions. Finally, the matrix classes' interfaces in Duffy (2004B) are incompatible with those of PPL and hence results for this case are not included. We note that we no longer support the matrix classes from Duffy (2004B).

We shall return to performance measurement, multithreading and multitasking in later chapters.

10.4.6 Boost Chrono Library

This library contains the same functionality as `<chrono>` in addition to a number of clocks not found in C++. These clocks are thin wrappers around the operating system's time APIs, thereby allowing the extraction of wall clock time, user CPU time and system CPU time spent by the process:

- `process_real_cpu_clock`: captures wall clock CPU time spent by the current process.
- `process_user_cpu_clock`: captures user CPU time spent by the current process.
- `process_system_cpu_clock`: captures system CPU time spent by the current process.
- A tuple-like class `process_cpu_clock` that captures real, user CPU and system CPU process times together.
- A `thread_clock` clock giving the time spent by the current thread (when supported by a platform).

We do not discuss these topics in this book but you may need to use them in certain circumstances. We recommend that you consult the Boost online documentation.

10.5 BOOST DATE AND TIME

We now introduce the Boost library that supports date and time entities. It is an understatement to say that having access to data structures and C++ classes that model *temporal entities* is important in many kinds of scheduling, management information and calendar-driven applications. To this end, the *Boost date–time* library has extensive support for many use cases, for example:

- Creating dates and times based on Gregorian, Posix and local time regimes.
- Date iterators; for example, iterating over the days in a week.
- Date generators and algorithms.
- Local and UTC (*Coordinated Universal Time*) support.
- Date and time input/output and serialisation.

The approach that we take in this book is to discuss the main features in the library and to give a number of '101' examples. We then continue by examining several extended examples and applications in which dates and times are used. For example, one project is to migrate and extend legacy code for computational finance applications to the current library (see Duffy, 2004B).

This chapter takes a hands-on approach and we discuss numerous examples to show the usefulness of the *Boost date–time* library. It is a large library.

10.5.1 Overview of Concepts and Functionality

Date and time abstractions are of fundamental importance in many kinds of applications. In particular, systems that incorporate *temporal logic* are numerous. Instead of defining our own libraries and functions to process dates and times we now use the *Boost date–time* library and extend it to suit the needs of specific application areas. To this end, we discuss the most important classes and functionality in the library. We can view time as a one-dimensional line or continuum.

There are three basic *temporal types*:

- *Time Point*: this is a specifier for a location in the time continuum.
- *Time Duration*: this is a length of time that is unattached to any point in the time continuum. It is a 'floating' quantity with a magnitude.
- *Time Interval* (*Time Period*): this is a duration of time that is attached to a specific point in the time continuum.

Corresponding to each of these concepts is a so-called *resolution* that is defined as the smallest representable duration. We define a *Time System* as a set of temporal types in combination with rules and operations to label and compute with these temporal types. A *Calendar System* is a time system with a resolution of one day. In this chapter we focus on the *Gregorian Calendar System* because it is the most widely used system. We also discuss the UTC calendar system that is used in civil time applications. *Local time systems* are based on UTC and are adjusted for the rotation of the earth so that daylight hours are similar everywhere. Finally, a *Clock Device* is a software component that is tied to hardware and that provides the current date or time in a time system.

10.5.2 Gregorian Time

The Boost implementation of the Gregorian system consists of the following types:

- `date`: this is the primary interface when working with dates.
- `date_duration`: this is a day-count class which we use when calculating with Gregorian dates.
- `date_period`: this is a class that represents a range or interval between two dates.

Furthermore, the library has support for iterating in different ways over a date period. It also has a number of functions for generating dates and schedules of dates.

In order to use the library, we recommend that you use the following namespace and include files in your code:

```
using namespace boost::gregorian;

#include <boost/date_time/gregorian/gregorian.hpp>       // Types and I/O.
#include <boost/date_time/gregorian/gregorian_types.hpp> // Types only.
```

We now discuss the Gregorian types in some detail.

10.5.3 Date

The `date` class has functionality for:

- Creating dates from date parts (day, month and year), from strings and from the system clock.
- Date accessors, for example retrieving date parts; queries on dates.
- Converting dates to strings.
- Operators, for example adding a duration to a date to produce another date.

We first consider date constructors. We note that the default constructor produces an invalid date. If we wish to create a date based on today's date we should call a constructor using the hardware clock. We can also create dates based on the Gregorian day, month and year. Finally, in some cases it can be advantageous to create dates having 'special' values, such as:

- Negative and positive infinity.
- Maximum and minimum dates.

We now give some examples to show how to create instances of `date`:

```
// Some basic constructors.
try
{
   date d1; Print(d1);         // Produces not_a_date_time.
}
catch (bad_year& e)
{
   cout << e.what() << endl;    // Year not valid range [1400, 10000].
}
```

```
// Create date from Gregorian year, month and day.
date myDate2(2011, May, 16); Print(myDate2);    // Print is author-defined.

// Copy constructor.
date myDate3(myDate2); Print(myDate3);

// Constructor at the 'extremes'.
try
{
   date myDate4(pos_infin); Print(myDate4, "Positive infinity");
}
catch (bad_year& e)
{
   cout << e.what() << endl;
}

try
{
   date myDate5(neg_infin); Print(myDate5, "Negative infinity");
}
catch (bad_year& e)
{
   cout << e.what() << endl;
}

try
{
   date myDate6(max_date_time); Print(myDate6, "Max Date Time");
}
catch (bad_year& e)
{
   cout << e.what() << endl;
}

try
{
   date myDate7(min_date_time); Print(myDate7, "Min Date Time");
}
catch (bad_year& e)
{
   cout << e.what() << endl;
}
```

The output from this code is:

```
Date Information...
not-a-date-time
Year is out of valid range: 1400..10000

Date Information...
2011-May-16
```

```
Year: 2011, Month: May, Day: 16
Day of week: Mon, Day of year: 136
End of month: 2011-May-31
ISO 8601 week number: 20, Day of year: 2455698

Date Information...
2011-May-16
Year: 2011, Month: May, Day: 16
Day of week: Mon, Day of year: 136
End of month: 2011-May-31
ISO 8601 week number: 20, Day of year: 2455698

Date Information...Positive infinity
+infinity
Year is out of valid range: 1400..10000

Date Information...Negative infinity
-infinity
Year is out of valid range: 1400..10000

Date Information...Max Date Time
9999-Dec-31
Year: 9999, Month: Dec, Day: 31
Day of week: Fri, Day of year: 365
End of month: 9999-Dec-31
ISO 8601 week number: 52, Day of year: 5373484

Date Information...Min Date Time
1400-Jan-01
Year: 1400, Month: Jan, Day: 1
Day of week: Wed, Day of year: 1
End of month: 1400-Jan-31
ISO 8601 week number: 1, Day of year: 2232400
```

Please note that we have created a function to print dates:

```
void Print(const date& myDate, const string& type = "")
{
   cout << "\nDate Information..." << type << endl;
   cout << myDate << endl;
   cout << "Year: " << myDate.year() << ", Month: "
         << myDate.month() << ", Day: " << myDate.day() << endl;

   // Extra stuff.
   cout << "Day of week: " << myDate.day_of_week()
         << ", Day of year: " << myDate.day_of_year() << endl;
   cout << "End of month: " << myDate.end_of_month() << endl;
   cout << "ISO 8601 week number: " << myDate.week_number()
         << ", Day of year: " << myDate.julian_day() << endl;
}
```

Continuing, we note that it is possible to create dates both from strings and from information based on the hardware clock, as the following example shows:

```
// Constructing dates from strings.
string s("2009/1/9");                          // 9 January 2009.
date myDate8(from_simple_string(s));
Print(myDate8,"from simple string");

// ISO 8601 extended format CCYY-MM-DD.
string s2("2009-10-9");                        // 9 October 2009.
date myDate9(from_simple_string(s2));
Print(myDate9,"from delimited string");

// Now convert to a string
string converted = to_simple_string(myDate9);
cout << "String: " << converted << endl;       // OUTPUT is 2009-Oct-09.

string s3("2009109");                          // 10 October 2009.
date myDate10(from_undelimited_string(s3));
Print(myDate10,"from UNdelimited string");

// Create dates from the clock.
date myDate11(day_clock::local_day());
Print(myDate11,"Local day based on time zone settings of computer");
date myDate12(day_clock::universal_day());
Print(myDate12,"Coordinated UniversalTime (UTC)");
```

The output from this code is (it depends on the current time):

```
Date Information...from simple string
2009-Jan-09
Year: 2009, Month: Jan, Day: 9
Day of week: Fri, Day of year: 9
End of month: 2009-Jan-31
ISO 8601 week number: 2, Day of year: 2454841

Date Information...from delimited string
2009-Oct-09
Year: 2009, Month: Oct, Day: 9
Day of week: Fri, Day of year: 282
End of month: 2009-Oct-31
ISO 8601 week number: 41, Day of year: 2455114
String: 2009-Oct-09

Date Information...from UNdelimited string
2009-Oct-09
Year: 2009, Month: Oct, Day: 9
Day of week: Fri, Day of year: 282
End of month: 2009-Oct-31
ISO 8601 week number: 41, Day of year: 2455114
```

```
Date Information...Local day based on time zone settings of computer
2011-May-18
Year: 2011, Month: May, Day: 18
Day of week: Wed, Day of year: 138
End of month: 2011-May-31
ISO 8601 week number: 20, Day of year: 2455700

Date Information...Coordinated Universal Time (UTC)
2011-May-18
Year: 2011, Month: May, Day: 18
Day of week: Wed, Day of year: 138
End of month: 2011-May-31
ISO 8601 week number: 20, Day of year: 2455700
```

We have now completed our discussion of the main functionality in class `date`.

10.6 FORWARDS LISTS AND COMPILE-TIME ARRAYS

In this section we discuss two recent data structures in C++.

10.6.1 `std::forward_list<>`

In STL, the container class `std::list<>` is usually implemented as a *doubly linked list*. This means that each element in the list has two pointers, one referring to the *predecessor* element and the other one referring to the *successor* element. This allows us to navigate in the list in both forward and backward directions. However, this level of flexibility comes at a cost:

- Increased memory consumption (we need to store pointers).
- Some applications only need to navigate in the forward direction.
- Deletion and insertion of elements in a doubly linked list demands destroying and creating resources.
- Possible performance issues when using doubly linked lists.

For these reasons we may consider using the container `std::forward_list<>` that manages its elements as a *singly linked list*. It supports efficient insertion and deletion of elements from anywhere in the container. Random access iterators are not supported. In short, a forward list provides a space-efficient alternative to `std::list<>` when bidirectional iteration is not needed.

The categories of member functions are (Josuttis, 2012):

- Creating and destroying lists.
- Non-modifying operations.
- Assignments.
- Element access.
- Inserting and removing elements.
- Splicing and merging lists.

The syntax is similar to that of `std::list<>` (which we assume to be known), which implies a friendly learning curve. We take some simple examples for motivation and the code

is easy to understand. First, we create some lists:

```
using List = std::forward_list<double>;

void CreateForwardLists()
{
   // Default empty list
   List c;

   // List with a size and a value
   int sz = 5; double val = 3.14;
   List c2(sz, val);
   print(c2);       // 3.14, 3.14, 3.14, 3.14, 3.14

   // List with initialiser list (2 forms)
   List c3{ 1.0, -4.6, 8.9 };
   print(c3); //  1.0, -4.6, 8.9

   List c4 = { 10.0, -40.6, 80.9 };
   print(c4);       //  10.0, -40.6, 80.9
}

void print(const List& list)
{
   // Single list has no size(), so we calculate distance
   // between first and last elements of list
   std::cout << "\nList size: "
             << std::distance(list.begin(), list.end())
             << ", contents are: ";

   for(auto& elem: list)
   {
      std::cout << elem << ",";
   }
}
```

We now give some code to find a given value in a list:

```
List::const_iterator Find(const List& list, double val)
{ // Find a value in a list

    for (auto pos = std::begin(list); pos != std::end(list); ++pos)
    {
       if (*pos == val)
       {
          return pos;  // value found
       }
    }

    return list.cend();
}
```

We take an example to show how to insert values at a certain position in a list and how to remove elements from a list. We notice that forward lists behave differently to doubly linked lists in this regard because the former do not have backward iterators.

```
void ModifyForwardList()
{ // Do some things with lists

    List list;
    list.push_front(-1.0);

    // Insert
    list.insert_after(list.begin(), 2.71);
    print(list);     // -1, 2.71

    int count = 2;
    list.insert_after(list.begin(), count, 148.33);
    print(list);     // -1, 148.33, 148.33, 2.71

    list.insert_after(list.begin(), { 150.1, 262.9, 56.3 });
    print(list);     // -1, 150.1, 262.9, 56.3, 148.33, 148.33, 2.71

    // Erase, remove the element AFTER the value
    double val = 262.9;
    auto pos = Find(list, val);
    if (pos != list.cend())
    {
       list.erase_after(pos);
    }
    else
    {
       std::cout << "\nValue " << val << " not found \n";
    }

    print(list);     // -1, 150.1, 262.9, 148.33, 148.33, 2.71
    val = 56.3;
    pos = Find(list, val);
    if (pos != list.cend())
    {
       list.erase_after(pos);
    }
    else
    {
       std::cout << "\nValue " << val << " not found \n";
    }

    print(list);     // -1, 150.1, 262.9, 148.33, 148.33, 2.71
}
```

Forward lists may be a viable alternative to doubly linked lists in certain cases. See Exercise 3.

10.6.2 `boost::array<>` and `std::array<>`

The STL `vector<T>` container provides the semantics of dynamic arrays. The number of elements in this class is variable and it is possible to append elements to it at run-time. In some applications, however, we may wish to work with arrays whose size is known at compile-time. To this end, the *Boost.Array* library is one solution. It consists of a class template `array<V,N>` having two template parameters, one parameter `N` for the size of the array and the other parameter `V` for the underlying data type. The interface has the following functionality:

- Array assignment.
- Accessing the elements of an array using the operator `[]` and the `at()` member function.
- Iterators.
- Member functions to retrieve the first and last elements of an array.
- We can compare arrays using the binary operators `==`, `!=`, `<`, `>`, `<=`, `>=`.
- We can exchange elements of two arrays using the `swap()` member function.

We take some simple examples. First, we create an array of `long` and we initialise its elements:

```
const int N = 4;
boost::array<long, N> myArr = {1, 2, 3, 4};
```

We can now print the elements of the array by using an iterator and the indexing operator:

```
// Using iterators
boost::array<long, N>::const_iterator it;
for (it = myArr.begin(); it != myArr.end(); ++it)
{
    cout << *it << endl;
}

// Indexing operator
for (long i = 0; i < myArr.size(); ++i)
{
    cout << myArr[i] << endl;
}
```

It is also possible to initialise an array as follows:

```
const int M = 2;
boost::array<string, M> myStringArr;
myStringArr[0] = "Hello";
myStringArr[1] = "World";
```

We now take an array that models a simple fixed-size polyline consisting of points. The elements of the array consist of instances of class `Point` whose interface is:

```
class Point
{
private:
   double x;                                  // X coordinate
   double y;                                  // Y coordinate

public:
   // Constructors
   Point();                                   // Default constructor
   Point(double xval, double yval);           // Initialise with x
   Point();                                   // and y value

   // Accessing functions
   double X() const;                          // The x-coordinate
   double Y() const;                          // The y-coordinate

   // Modifiers
   void X(double newX);
   void Y(double newY);

   Point operator + (const Point& p2) const;
   double distance (const Point& p2) const; // Distance between 2 points

   Point& operator = (const Point& pt);
};
```

We now create an array of points:

```
const int N = 4;
boost::array<Point, N> myPointArr;
myPointArr[0] = Point(1.0, 2.0);
myPointArr[1] = Point();
myPointArr[2] = Point(-1.0, 9.8);
myPointArr[3] = Point(3.4, 6.32);
```

There are two ways to access the elements of this array but they differ in the way they handle exceptional situations, in particular *out-of-bounds indexing*. First, an assert is executed in the case of operator overloading:

```
// What happens when the index is out-of-range?
// myPointArr[N] = Point();        // Assertion 'index < N'
// myPointArr[N].X(1.0);           // Assertion 'index < N'
```

The program stops executing in this case. Second, the `at()` member function throws an exception when an index is out-of-bounds and we catch the exception in the usual way:

```
try
{
   cout << "Give an index: "; int index; cin >> index;
```

```
    Point pt = myPointArr.at(index);
    cout << "(" << pt.X() << "," << pt.Y() << ")" << endl;
}
catch(const exception& e)
{
    cout << e.what() << endl;
}
```

Our final remark is that we cannot use arrays of different sizes together. This increases the robustness of applications that use them. We take an example of two arrays:

```
// Assignment of arrays with incompatible dimensions
const int N1 = 3;
const int N2 = 4;

boost::array<double, N1> arr1;
boost::array<double, N2> arr2;
```

Then the following code will not compile:

```
arr1 = arr2;           // Gives compiler error
```

10.7 APPLICATIONS OF BOOST.ARRAY

There are many applications of *Boost.Array*. Some examples are:

- *n*-Dimensional geometry: we can create a template class that represents points and other objects in *n* dimensions. In this case we implement the class `NPoint` by using an embedded `boost::array` instance:

```
template <typename Type, int n> class NPoint
{
private:

    // Originally: Type arr[n]
    boost::array<Type, n> arr;

public:

    // member functions here..
};
```

We compute the distance between two points as follows:

```
const int N = 3;
NPoint<double, N> p1(0.0);
NPoint<double, N> p2(1.0);

cout << "Distance: " << p1.distance(p2) << endl;
```

- Creating multidimensional, compile-time data structures: a good example is a template matrix with *N* rows and *M* columns:

```
template <typename T, std::size_t NRows, std::size_t NColumns>
class Matrix
{ // Basic class, can be extended

private:
   boost::array<boost::array<T, NColumns>, NRows> mat;

public:
   Matrix();
   std::size_t Rows() const;
   std::size_t Columns() const;

   T& operator () (std::size_t r, std::size_t c);
   const T& operator () (std::size_t r, std::size_t c) const;
};
```

An example of use is:

```
const int NROWS = 3; const int NCOLUMNS = 3;
Matrix<double, NROWS, NCOLUMNS> myMatrix;
```

- Using *Boost.Array* with *Boost.MultiArray*: part of the code for initialising a multiarray involves defining its extents. The following code shows how to define these extents using `boost::array`:

```
int main()
{
   const int N = 3;
   typedef boost::multi_array<int, N> Array;

   // Define the extents using boost::array
   boost::array<Array::size_type, N> arrayDims;
   arrayDims[0] = 2;
   arrayDims[1] = 3;
   arrayDims[2] = 4;

   Array myArray(arrayDims);

   int num = 0;
   for (Array::index i=0; i!=arrayDims[0]; ++i)
      for (Array::index j=0; j!= arrayDims[1]; ++j)
         for (Array::index k=0; k!=arrayDims[2]; ++k)
            myArray[i][j][k] = num++;

   return 0;
}
```

10.8 BOOST uBLAS (MATRIX LIBRARY)

10.8.1 Introduction and Objectives

The *Boost uBLAS* library supports vector and matrix data structures and basic linear operations on these structures. The syntax closely reflects mathematical notation because operator overloading is used. Furthermore, the library uses *expression templates* to generate efficient code. The library has been influenced by a number of other libraries such as ATLAS, BLAS, Blitz++, POOMA and MTL. The main design goals are:

- Use mathematical notation whenever appropriate.
- Efficiency (time and resource management).
- Functionality (provide features that appeal to a wide range of application areas).
- Compatibility (array-like indexing and use of STL allocators for storage allocation).

The two most important data structures represent vectors and matrices. A *vector* is a one-dimensional structure while a *matrix* is a two-dimensional structure. We can define various *vector and matrix patterns* that describe how matrices are arranged in memory; examples are *dense*, *sparse*, *banded*, *triangular*, *symmetric* and *Hermitian* matrices. These patterned structures are used in many kinds of applications. Furthermore, we can define primitive operations on vectors and matrices, for example:

- Addition of vectors and matrices.
- Scalar multiplication.
- Computed assignments.
- Transformations.
- Norms of vectors and matrices.
- Inner and outer products.

We can use these operations in code and applications. Finally, we can define *subvectors* and *submatrices* as well as *ranges* and *slices* of vectors and matrices.

Vectors and matrices are fundamental to scientific and engineering applications and having a well-developed library such as *Boost uBLAS* with ready-to-use modules will free up developer time. Seeing that matrix algebra consumes much of the effort in an application we expect that the productivity gains will be appreciable in general. A discussion of applications of matrices is outside the scope of this book. However, we do discuss LU decomposition, Cholesky decomposition and the solution of tridiagonal matrix systems in this book.

10.8.2 BLAS (Basic Linear Algebra Subprograms)

BLAS is a *de facto* application programming interface standard for libraries that perform basic linear algebra operations on vectors and matrices. They promote the readability, modularity and maintainability of software. BLAS promotes *modularity* by identifying frequently occurring operations of linear algebra and by specifying a standard interface to these operations. *Efficiency* is achieved by optimising the code within BLAS. It is important to identify and

define a set of basic operations that is rich enough to allow us to model high-level algorithms on the one hand and simple enough to allow optimisation on a range of computers on the other hand.

The advantages of using BLAS are its robustness, portability and readability. The BLAS functionality consists of three levels that we now discuss.

10.8.3 BLAS Level 1

This level consists of low-level operations such as *inner products* (also known as *dot products*) of vectors and the addition of a multiple of one vector to another vector. These *vector–vector operations* are referred to as Level 1 BLAS, such as:

$$y \leftarrow \alpha x + y \tag{10.1}$$

where x and y are vectors and α is a scalar.

These operations involve $O(n)$ floating-point operations in general, where n is the length of the vectors. In equation (10.1) we see that the vector y is being updated and the operation is sometimes called *saxpy* ('scalar ax plus y').

10.8.4 BLAS Level 2

This level contains matrix–vector operations of the form:

$$y \leftarrow \alpha Ax + \beta y \tag{10.2}$$

where x and y are vectors, A is a matrix, α and β are scalars.

These *matrix–vector operations* occur during the implementation of many of the most common algorithms in linear algebra. The Level 2 BLAS involve $O(mn)$ scalar operations where m and n are the dimensions of the matrix involved in the operations.

Other operations in BLAS Level 2 are:

- Rank-one and rank-two updates:

$$\begin{aligned} A &= \alpha xy^{\mathrm{T}} + A \text{ (rank 1 update)} \\ A &= \alpha xy^{\mathrm{T}} + \alpha yx^{\mathrm{T}} + A \text{ (rank 2 update)} \end{aligned} \tag{10.3}$$

where y^{T} is the transpose of vector y.

- Solution of triangular equations of the form:

$$Ty = x \tag{10.4}$$

where T is a non-singular *upper-triangular matrix* (all elements below the main diagonal are zero) or *lower-triangular matrix* (all elements above the main diagonal are zero).

In general the operations apply to general band, Hermitian, Hermitian band and triangular band matrices with real and complex coefficients in both single and double precision.

10.8.5 BLAS Level 3

This level concerns *matrix–matrix operations* and they are similar to Level 2 operations. In general, we replace the vectors x and y in equations (10.3) by matrices B and C, for example. This approach keeps the design of the software as consistent as possible with the software of the Level 2 BLAS. It also helps users remember the calling sequences and parameter conventions. Some examples are:

- Matrix–matrix products:

$$\begin{aligned} C &= \alpha AB + \beta C \\ C &= \alpha A^{\mathrm{T}}B + \beta C \\ C &= \alpha AB^{\mathrm{T}} + \beta C \\ C &= \alpha A^{\mathrm{T}}B^{\mathrm{T}} + \beta C. \end{aligned} \tag{10.5}$$

- Rank-k updates of a symmetric matrix C:

Here A, B and C are matrices, α and β are scalars.

$$\begin{aligned} C &= \alpha AA^{\mathrm{T}} + \beta C \ (A^{\mathrm{T}} \text{ is the transpose of } A) \\ C &= \alpha A^{\mathrm{T}}A + \beta C \\ C &= \alpha A^{\mathrm{T}}B + \alpha B^{\mathrm{T}}A + \beta C \\ C &= \alpha AB^{\mathrm{T}} + \alpha BA^{\mathrm{T}} + \beta C. \end{aligned} \tag{10.6}$$

- Multiplying a matrix B by a triangular matrix T:

$$\begin{aligned} B &= \alpha TB \\ B &= \alpha T^{\mathrm{T}}B \\ B &= \alpha BT \\ B &= \alpha BT^{\mathrm{T}}. \end{aligned} \tag{10.7}$$

- Solving triangular systems of equations with multiple right-hand sides:

$$\begin{aligned} B &= \alpha T^{-1}B \ (T^{-1} \text{ is the inverse of } T) \\ B &= \alpha BT^{-1} \end{aligned} \tag{10.8}$$

where α and β are scalars; A, B and C are rectangular matrices; T is an upper- or lower-triangular matrix. Boost uBLAS supports the above BLAS operations and they are mapped to appropriate C++ function calls. We discuss these mappings in detail in this and the next chapter.

10.9 VECTORS

10.9.1 Dense Vectors

A *dense vector* of length n is one all of whose elements are explicitly given a value. In other words, there are no 'gaps' (missing values) in these vectors. To this end, the class `vector<T, Alloc>` has two parameters:

- The type `T` of the object stored in the vector.
- The type `Alloc` of storage array (dynamic, run-time storage, fixed-size storage or compile-time storage).

We show the conceptual structure of this class in a UML class diagram in Figure 10.2. To use this class we instantiate its template parameters (the default allocator type is `unbounded_array<T>`).

The categories of member functions are:

- Constructors.
- Accessing the elements of the vector using the operators `()` and `[]`.
- Adding and subtracting two vectors.
- Multiplying and dividing a vector by a scalar.
- Constant and non-constant forward and reverse iterators.
- Resizing a vector (*reallocating* a vector to hold a given number of elements).

Rather than giving extended examples of all the different member functions (many of which are easy if you are familiar with STL) we prefer to concentrate on critical member functions. We also mention that many STL algorithms can be used on uBLAS vectors.

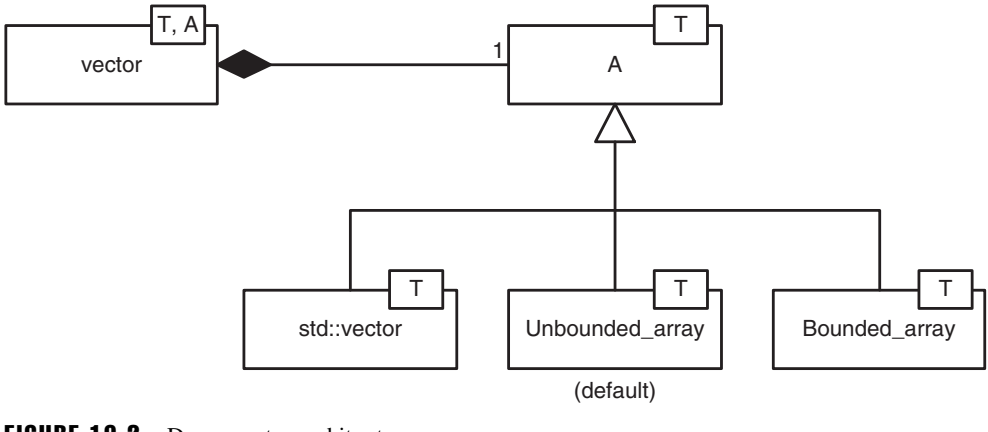


FIGURE 10.2 Dense vector and its storage

10.9.2 Creating and Accessing Dense Vectors

We first create a number of dense vectors based on the options in Figure 10.1. We can create vectors having both numeric and non-numeric underlying data types and given storage arrays. Note that we use an alias for the Boost uBLAS namespace:

```
#include <boost/numeric/ublas/vector.hpp>       // The vector class.
#include <boost/numeric/ublas/io.hpp>           // Sending to IO stream.

namespace ublas=boost::numeric::ublas;

ublas::vector<double> v1(10);
ublas::vector<double, ublas::unbounded_array<double> > v2(10);
ublas::vector<double, ublas::bounded_array<double, 20> > v3(10);
ublas::vector<double> v4(5, 3.14);
ublas::vector<double, vector<double> > v5(10, 4.5);
cout<<"v5: "<<v5<<endl;

ublas::vector<char, vector<char> > v6(10, 'd');
cout<<"v6: "<<v6<<endl;
```

We can access and modify the elements of a vector using overloaded operators:

```
// Fill the vectors.
for (int i=0; i<10; i++)
{
   // Use the [] operator.
   v1[i]=i;

   // Use the () operator.
   v2(i)=i+10;
   v3(i)=i+20;
}
```

We now show the use of composite operators and the support for iterators:

```
// Operators.
// When adding/subtracting two vectors, they should have the same size.
cout<<endl<<"*** Operators ***"<<endl;
cout<<"v1+=v3:  "<<(v1+=v3)<<endl;
cout<<"v1-=v3:  "<<(v1-=v3)<<endl;
cout<<"v1*=2.5: "<<(v1*=2.5)<<endl;
cout<<"v1/=2.5: "<<(v1/=2.5)<<endl;

// Iterators.
cout<<endl<<"*** Iterators ***"<<endl;
transform(v1.begin(), v1.end(), v2.begin(), v3.begin(),
          multiplies<double>());
cout<<"Multiply v1 & v2 to v3 using transform algorithm ";
cout<<"and multiplies function object: "<<v3<<endl;
```

Finally, we can resize a vector (notice the option to preserve the existing data in the vector):

```
v1.resize(8, false); cout<<".. to 8 (no data preserved): "<<v1<<endl;
v2.resize(8, true); cout<<".. to 8 (data preserved): "<<v2<<endl;
```

10.9.3 Special Dense Vectors

The uBLAS has support for special kinds of dense vectors:

- *Unit vector*: for a given vector size n, the kth *canonical vector* has value zero for all elements except for index k which has value 1. It is clear that there are n canonical *unit vectors* for vectors of size n and they form an *orthonormal basis* in n-dimensional Euclidean space. An example in three-dimensional space is:

```
// Create unit vectors. The first argument is the size,
// the second argument is the element that is 1.
ublas::unit_vector<double> uv1(3, 0);  // 1, 0, 0
ublas::unit_vector<double> uv2(3, 1);  // 0, 1, 0
ublas::unit_vector<double> uv3(3, 2);  // 0, 0, 1

cout<<"uv1: "<<uv1<<", index of 1: "<<uv1.index()<<endl;
cout<<"uv2: "<<uv2<<", index of 1: "<<uv2.index()<<endl;
cout<<"uv3: "<<uv3<<", index of 1: "<<uv3.index()<<endl;
```

- *Zero vector*: the n-dimensional zero vector has all its values equal to zero:

```
// All elements of a zero vector are always zero.
// Constructor accepts size.
ublas::zero_vector<double> zv(10);
cout<<"Zero vector (always contains all zeros): "<<zv<<endl;
```

- *Scalar vector*: this is a generalisation of the zero vector. A scalar vector of size n is one all of whose elements have the same value. An example of use is:

```
// A scalar vector has elements with the same value.
ublas::scalar_vector<double> sv1(5, 3.5);    // Size 5, contents 3.5.
ublas::scalar_vector<double> sv2(10, 9.0);   // Size 10, contents 9.0.
cout<<"Scalar vector with 3.5: "<<sv1<<endl;
cout<<"Scalar vector with 9.0: "<<sv1<<endl;
```

It is useful to have these special vectors at our disposal because they are needed in numerical linear algebra applications. Note that the elements of these special vectors are read-only.

10.10 MATRICES

We give an introduction to dense matrices in Boost uBlas.

10.10.1 Dense Matrices

A *dense matrix* having n rows and m columns is one all of whose $n \times m$ elements are stored in memory. These types of matrices are used in many kinds of applications. The class `matrix<T, F, Alloc>` has three template parameters:

- `T`: the type of object stored in the matrix.
- `F`: the function object that describes the storage organisation, for example row order (the default is `row_order`).
- `Alloc`: the type of storage array (the default is `unbounded_array<T>`).

The conceptual UML class diagram is shown in Figure 10.3. The categories of member functions in this class are:

- Constructors.
- Accessing and modifying the elements of a matrix.
- Arithmetic operations, such as addition and multiplication of matrices.
- Multiplication and division by scalars.
- Iterators.

We now discuss dense matrices from a C++ perspective and we also give some examples of dense matrices.

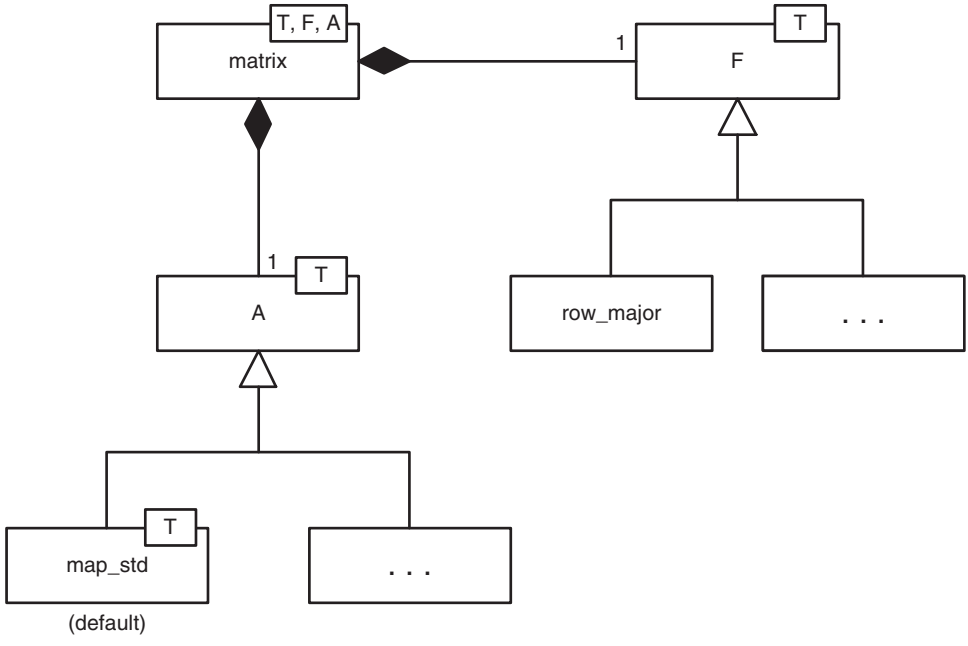


FIGURE 10.3 Dense matrix structure

10.10.2 Creating and Accessing Dense Matrices

We give some initial examples of creating dense matrices, accessing and modifying their elements and finally printing them:

```
#include <boost/numeric/ublas/matrix.hpp>          // The matrix class.
namespace ublas=boost::numeric::ublas;

// Default matrix which stores data in the unbounded_array<T> class
// by default and has row major storage order.
ublas::matrix<double> m1(2, 3);

// Matrix which stores data in a bounded array and has column major
// storage order.
ublas::matrix<double, ublas::column_major,
                  ublas::bounded_array<double, 6> > m2(2, 3);

// Matrix with all elements initialized to 3.14.
ublas::matrix<double> m3(2, 3, 3.14);

// Fill the matrices.
for (int row=0; row<2; row++)
{
   for (int column=0; column<3; column++)
   {
      m1(row, column)=10*row+column;
      m2(row, column)=m1(row, column)+5;
   }
}

// Display the matrices.
cout<<"m1: "<<m1<<endl;
cout<<"m2: "<<m2<<endl;
cout<<"m3: "<<m3<<endl;

// Sizes.
cout<<endl<<"*** Sizes ***"<<endl;
cout<<"m1 rows: "<<m1.size1()<<", columns: "<<m1.size2()<<endl;
cout<<"m2 rows: "<<m2.size1()<<", columns: "<<m2.size2()<<endl;
cout<<"m3 rows: "<<m3.size1()<<", columns: "<<m3.size2()<<endl;
```

The member functions `size1()` and `size2()` give the number of rows and columns in the matrix respectively.

We can use arithmetic operations on dense matrices, as the following example shows:

```
// Operators.
// Adding/subtracting two matrices, they should have the same size.
cout<<endl<<"*** Operators ***"<<endl;
cout<<"m1+=m2:  "<<(m1+=m3)<<endl;
cout<<"m1-=m2:  "<<(m1-=m3)<<endl;
cout<<"m1*=2.5: "<<(m1*=2.5)<<endl;
cout<<"m1/=2.5: "<<(m1/=2.5)<<endl;
```

Finally, it is possible to resize dense matrices. We can modify the number of rows and columns by providing the appropriate integer parameters as well as a Boolean parameter that determines whether the values from the 'old' matrix should be preserved in the 'new' matrix:

```
// Resize the matrix.
// Matrices which use the bounded_array as storage always preserve data
// regardless of the "preserve" argument.
cout<<endl<<"*** Resize ***"<<endl;
m1.resize(2, 2, false);
cout<<"m1 resized to 2,2 (no data preserved): "<<m1<<endl;
m2.resize(2, 2, false);
cout<<"m2 resized to 2,2 (bounded_array always preserves): "<<m2<<endl;
m3.resize(2, 2, true);
cout<<"m3 resized to 2,2 (data preserved): "<<m3<<endl;
cout<<endl;
m1.resize(2, 3, false);
cout<<"v1 resized back to 2,3 (no data preserved): "<<m1<<endl;
m2.resize(2, 3, false);
cout<<"v2 resized back to 2,3 (bounded always preserves): "<<m2<<endl;
m3.resize(2, 3, true);
cout<<"v3 resized back to 2,3 (data preserved): "<<m3<<endl;
```

10.10.3 Special Dense Matrices

We now discuss three special kinds of matrices that are used in numerical linear algebra applications:

- `Identity matrix`: this is a *square matrix* (the number of rows is the same as the number of columns) all of whose elements are zero except on the main diagonal of the matrix where the value is 1. An example of use is:

```
// Create identity matrices.
ublas::identity_matrix<double> m1(2);
ublas::identity_matrix<double> m2(3);

// Print the matrices.
cout<<"m1 (2x2): "<<m1<<endl;
cout<<"m2 (3x3): "<<m2<<endl;
cout<<"m1 rows: "<<m1.size1()<<", columns: "<<m1.size2()<<endl;
cout<<"m2 rows: "<<m2.size1()<<", columns: "<<m2.size2()<<endl;

// Resize the identity matrix. The data is always preserved even
// when preservation argument is false.
m2.resize(2, false);
cout<<"m2 resized to 2 (data always preserved)"<<m2<<endl;
m2.resize(3, false);
cout<<"m2 resized back to 3 (data always preserved)"<<m2<<endl;
```

- `Zero matrix`: this is a *rectangular matrix* (the number of rows is not necessarily equal to the number of columns) all of whose elements are zero. An example of use is:

```
// Create zero matrices.
ublas::zero_matrix<double> m1(2, 3);
ublas::zero_matrix<double> m2(3);

// Print the matrices.
cout<<"m1 (2x3): "<<m1<<endl;
cout<<"m2 (3x3): "<<m2<<endl;
cout<<"m1 rows: "<<m1.size1()<<", columns: "<<m1.size2()<<endl;
cout<<"m2 rows: "<<m2.size1()<<", columns: "<<m2.size2()<<endl;

// Resize the zero matrix. The data is always preserved even
// when preservation argument is false.
m1.resize(3, 2, false);
cout<<"m1 resized to 3x2 (data always preserved)"<<m1<<endl;
m1.resize(2, 3, false);
cout<<"m1 resized back to 2x3 (data always preserved)"<<m1<<endl;

cout<<endl<<"Swap the matrices"<<endl;
m1.swap(m2);
cout<<"m1: "<<m1<<endl;
cout<<"m2: "<<m2<<endl;
```

- *Scalar matrices*: a scalar matrix is a rectangular matrix all of whose elements are equal to a given (single) value. An example of use is:

```
// Create scalar matrices.
ublas::scalar_matrix<double> m1(2, 3, 4);
ublas::scalar_matrix<double> m2(3, 2, 5);

// Print the matrices.
cout<<"m1 (2x3)=4: "<<m1<<endl;
cout<<"m2 (3x2)=5: "<<m2<<endl;
cout<<"m1 rows: "<<m1.size1()<<", columns: "<<m1.size2()<<endl;
cout<<"m2 rows: "<<m2.size1()<<", columns: "<<m2.size2()<<endl;
```

Note that the elements of these special vectors are read-only.

10.11 APPLYING uBLAS: SOLVING LINEAR SYSTEMS OF EQUATIONS

We have completed our discussion of uBLAS functionality, including data structures for vectors and matrices, views of these data structures and finally operations on these data structures. We now show how to bring this functionality together. We focus on three important algorithms:

- The conjugate gradient method (CGM) for symmetric and general matrices.
- LU decomposition of a general matrix.
- Cholesky decomposition.

There are many other applications that we could have taken but we feel that these three algorithms motivate the usefulness of uBLAS. The following *features* in uBLAS will be used in the ensuing discussion:

- Symmetric, lower-triangular, upper-triangular and general matrices.
- Inner product of two vectors.
- Matrix–vector multiplication.
- Sum and difference of two vectors.
- Vector norms (needed when testing for convergence of iterative methods).
- The transpose of a matrix.

We discuss these matrix algorithms and their implementation using uBLAS.

10.11.1 Conjugate Gradient Method

This method is used for solving a system of simultaneous linear algebraic equations:

$$AU = F$$

where A is a given $n \times n$ matrix, U is an $n \times 1$ vector of unknowns and F is a known $n \times 1$ vector. In this section we first discuss the CGM algorithm and corresponding C++ code when A is a symmetric matrix and we then discuss the steps of the algorithm when A is a general (not necessarily symmetric) matrix. The algorithm in the case of a symmetric matrix is given by:

$$r_0 = F - AU_0$$
$$p_0 = r_0$$

for $j = 1,\ldots, n$ compute:

$$a_j = \|r_j\|_2^2 / p_i^T A p_i$$
$$U_{j+1} = U_j + a_j p_j$$
$$r_{j+1} = r_j - a_j A p_j$$
$$b_j = \frac{\|r_{j+1}\|_2^2}{\|r_i\|_2^2}$$
$$p_{j+1} = r_{j+1} + b_j p_j$$

where r_j and p_j are vectors of length n for $j \geq 0$.

The code for this algorithm is:

```
// Solve Ax = b; 'x' is initial estimate.
template <typename TMatrix>
void ConjugateGradient(const TMatrix& A, const ublas::vector<double>& b,
                       ublas::vector<double>& x, double tol)
{
   // The size.
   const size_t n = b.size();

   // Step 0: init.
   ublas::vector<double> r(n);
```

```
    r = b - ublas::prod(A,x);
    ublas::vector<double> p(r);

    double a;
    double beta;
    double norm, norm2;
    ublas::vector<double> tmpVec(n);

    for (size_t j=1; j<=n; ++j)
    {
       // Step 1.
       norm = ublas::norm_2(r);
       norm *= norm;

       // Exit loop is convergence has been reached.
       if (norm <= tol) break;

       tmpVec = ublas::prod(A, p);
       a = norm / ublas::inner_prod(p, tmpVec);

       // Step 2.
       x += a*p;

       // Step 3.
       r -= a*tmpVec;

       // Step 4.
       norm2 = ublas::norm_2(r); norm2 *= norm2;
       beta = norm2 / norm;
       p = beta*p + r;
    }
}
```

The CGM algorithm in the case of a general matrix A is:

$$r_0 = F - AU_0$$
$$p_0 = r_0$$

for $j = 1, 2, \ldots, n$:

$$a_j = \|A^T r_j\|^2 / \|Ap_j\|^2$$
$$U_{j+1} = U_j + a_j p_j$$
$$r_{j+1} = r_j - a_j A p_j$$
$$b_j = \|A^T r_{j+1}\|^2 / \|A^T r_j\|^2$$
$$p_{j+1} = A^T r_{j+1} + b_j p_j.$$

We leave it as an exercise to modify the CGM code for symmetric matrices so that it works in the case (non-symmetric) just given.

10.11.2 LU Decomposition

This algorithm partitions a general matrix A into the product of a lower-triangular matrix L and an upper-triangular matrix U. The algorithm is documented in Dahlquist and Björck (1974) and Isaacson and Keller (1966), for example. The C++ code in combination with uBLAS given as a free function is:

```
// LU decomposition: A -> L*U
void InitLU(const ublas::matrix<double>& A,
         ublas::triangular_matrix<double, ublas::lower>& L,
         ublas::triangular_matrix<double, ublas::upper>& U)
{
   double sum;
   unsigned N = A.size1();

   // Common to make all diagonal elements == 1.0
   for (size_t k = 0; k < N; ++k)
   {
      L(k,k) = 1.0;
   }

   for (size_t j = 0; j < N; ++j)      // Loop over columns.
   {
      // U
      for (size_t i= 0; i <= j; ++i)   // Columns.
      {
         sum = 0.0;
         for (size_t k = 0; k < i; ++k)
         {
            sum += L(i,k)*U(k,j);
         }
         U(i,j) = A(i,j) - sum;
       }
       // L
       for (size_t i = j+1; i < N; ++i)      // Rows.
       {
          sum = 0.0;
          for (size_t k = 0; k < j; ++k)
          {
             sum += L(i,k)*U(k,j);
          }
          L(i,j) = (A(i,j) - sum) / U(j,j);
       }
   }
}
```

Having computed the matrices L and U we can then solve the following problem in two steps:

$$\begin{aligned} &Ax = b \\ &\Leftrightarrow (LU)x = b \Leftrightarrow Ly = b \\ &ux = y. \end{aligned}$$

The code for this sequence of steps is:

```
// Solve Ax = b
ublas::vector<double> SolveLU(const ublas::vector<double>& b,
                              const ublas::triangular_matrix<double,
                              ublas::lower>& L,
                              const ublas::triangular_matrix<double,
                              ublas::upper>& U)
{
   size_t N = b.size();
   ublas::vector<double> result(N);

   double sum;

   // Forward sweep Ly = b
   result[0] = b[0] / L(0,0);
   for (size_t i = 1; i < N; ++i)
   {
      sum = 0.0;
      for (size_t k = 0; k < i; ++k)
      {
         sum += L(i,k)*result[k];
      }
      result[i] = (b[i] - sum) / L(i,i);
   }

   // Backward sweep Ux = y
   result[N-1] = result[N-1]/U(N-1, N-1);
   for (size_t i = N-2; i >= 0 && i < N; --i)
   {
      sum = 0.0;
      for (size_t k = i+1; k < N; ++k)
      {
         sum += U(i,k)*result[k];
      }
         result[i] = (result[i] - sum) / U(i,i);
   }

   return result;
}
```

A simple example (null test) of use is:

```
unsigned N = 4;

// Input for Ax = b
ublas::matrix<double> A(N, N);
ublas::vector<double> b(N);
for (size_t i = 0; i < A.size1(); ++i)
{
   b[i] = 1.0;
```

```
    for (size_t j = 0; j < A.size2(); ++j)
    {
       A(i, j) = 0.0;
    }
    A(i,i) = 1.0;
}
std::cout << "Initial matrix A: " << A << std::endl;
cout << "Initial vector b: " << b << endl;

// Determine the L and U matrices.
ublas::triangular_matrix<double, ublas::lower> L(N,N); Init(L, 0.0);
ublas::triangular_matrix<double, ublas::upper> U(N,N); Init(U, 0.0);
InitLU(A, L, U);
cout << "Matrix L: " << L << endl;
cout << "Matrix U: " << U << endl;

// Solve 'Ax=b'.
ublas::vector<double> result = SolveLU(b, L, U);
cout << "Result vector Ax=b: " << result << endl;
```

You can experiment with different choices of `A` and `b` to test the accuracy of the method.

10.11.3 Cholesky Decomposition

For symmetric positive-definite matrices the LU decomposition algorithm takes on a particularly simple form, namely when the matrix U is the transpose of the matrix L:

$$A = LU = LL^{\mathrm{T}}.$$

The algorithm to compute the matrix L is given by (see Dahlquist and Björck, 1974):

$$l_{jj} = \left(a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2 \right)^{1/2}, \quad 1 \le j \le n$$

$$l_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk} \right) \Big/ l_{jj}, \quad i = j+1, \ldots, n.$$

The C++ code that implements this algorithm is:

```
// Cholesky decomposition.
void Cholesky(const ublas::matrix<double>& A,
    ublas::triangular_matrix<double, ublas::lower>& L)
{
       double sum;
   size_t N = A.size1();

   for (size_t j = 0; j < N; ++j)   // Loop over columns.
   {
```

```
        sum = 0.0;
        for (size_t k = 0; k < j; ++k)
        {
           sum += L(j,k)*L(j,k);
        }
        L(j,j) = sqrt(A(j,j) - sum);

        for (size_t i = j+1; i < N; ++i) // Rows.
        {
           sum = 0.0;
           for (size_t k = 0; k < j; ++k)
           {
              sum += L(i,k)*L(j,k);
           }
           L(i,j) = (A(i,j) - sum) / L(j,j);
        }
     }
  }
```

Finally, an example of how to use the Cholesky algorithm is given by (notice we used a dense matrix but using a symmetric matrix would possibly have been more appropriate):

```
// Correlation matrix.
unsigned D = 4;
u::matrix<double> Corr(D,D);
Corr(0,0) = Corr(1,1) = Corr(2,2) = Corr(3,3) = 1.0;
Corr(0,1) = Corr(1,0) = 0.5;
Corr(0,2) = Corr(2,0) = 0.2;
Corr(0,3) = Corr(3,0) = 0.01;
Corr(1,2) = Corr(2,1) = 0.01;
Corr(1,3) = Corr(3,1) = 0.30;
Corr(3,2) = Corr(2,3) = 0.30;
cout << "Original matrix:" << endl << Corr << endl << endl;
```

A test case is:

```
// Cholesky decomposition.
cout << endl << "--- Do Cholesky decomposition ---" << endl << endl;
ublas::triangular_matrix<double, ublas::lower> L(D,D);
Cholesky(Corr, L);
cout << "L:" << endl << L << endl << endl;
```

Now we remark on stress testing the Cholesky method. In particular, we wish to double-check our algorithm while at the same time using some of the functionality of uBLAS. We try to *recover* the original matrix that we used in the algorithm by multiplying the constructed lower-triangular matrix by its transpose:

```
// Recover the original matrix; a kind of check.
ublas::triangular_matrix<double, ublas::upper> U = LowerToUpper(L);
```

```
cout << "U:" << endl << U << endl << endl;
cout << "L*U (same as orignal matrix):" << endl << L*U << endl << endl;
```

where `LowerToUpper()` is a free function that converts a lower-triangular matrix to an upper-triangular matrix:

```
// Transpose a lower triangular matrix to form an upper triangular matrix.
ublas::triangular_matrix<double, ublas::upper> LowerToUpper(
   const ublas::triangular_matrix<double, ublas::lower>& L)
{
   ublas::triangular_matrix<double, ublas::upper> U(L.size2(), L.size1());

   for (size_t i = 0; i < U.size1(); ++i)
   {
      for (size_t j = i; j < U.size2(); ++j)
      {
         U(i,j) = L(j,i);
      }
   }

   return U;
}
```

We finish this section by recovering the original matrix, multiplying the upper- and lower-triangular matrix that was the result of LU decomposition:

```
// Create LU matrices.
cout << "--- Recover original matrix from LD & LD ---" << endl << endl;
ublas::triangular_matrix<double, ublas::lower> LD(D,D);
ublas::triangular_matrix<double, ublas::upper> UD(D,D);
InitLU(Corr, LD, UD);
cout << "LD: " << endl << LD << endl << endl;
cout << "UD: " << endl << UD << endl << endl;

// Recover original matrix in different ways.
cout << "LD*UD (same as orignal matrix): " << endl << LD*UD << endl;
cout << "UD*LD (same as orignal matrix): " << endl << UD*LD << endl;
```

where we have defined operators for multiplying lower- and upper-triangular matrices:

```
// Compute Lower*Upper.
ublas::matrix<double> operator * (
                        const ublas::triangular_matrix<double,
                        ublas::lower>& L,
                        const ublas::triangular_matrix<double,
                        ublas::upper>& U)
{
   ublas::matrix<double> result(L.size1(), L.size2());

   double sum; size_t r;
```

```
    for (size_t i = 0; i < result.size1(); ++i)
    {
       for (size_t j = 0; j < result.size2(); ++j)
       {
          sum = 0.0;
          r = std::min(i,j);
          for (size_t p = 0; p <= r; ++p)
          {
             sum += L(i,p)*U(p,j);
          }

          result(i,j) = sum;
       }
    }

    return result;
}

// Compute Upper*Lower.
ublas::matrix<double> operator * (
    const ublas::triangular_matrix<double, ublas::upper>& U,
    const ublas::triangular_matrix<double, ublas::lower>& L)
{
    // Rough and Ready, but it works; it is a test of uBLAS::transpose.
    ublas::triangular_matrix<double, ublas::upper> Upper = trans(L);
    ublas::triangular_matrix<double, ublas::lower> Lower = trans(U);

    return Lower*Upper;
}
```

We can run this code and see that the results are the same in all cases. Finally, another test is to subtract the original matrix from both products and then take the corresponding max norm:

```
// Look at matrix norms.
ublas::matrix<double> m1 = Corr - UD*LD;
ublas::matrix<double> m2 = Corr - LD*UD;
cout << "A - U*L max error: " << ublas::norm_inf(m1) << endl;
cout << "A - L*U max error: " << ublas::norm_inf(m2) << endl << endl;
```

These values should be very small.

10.12 SUMMARY AND CONCLUSIONS

This is a special chapter in the sense that it is a discussion of functionality and libraries that support the applications in later chapters. It is meant as a reference that you can consult when reusing these resources. The focus was on matrices and matrix operations.

10.13 EXERCISES AND PROJECTS

1. (`std::vector<bool>` versus `std::bitset<>`)

An alternative to bitsets is to employ the class `std::vector<bool>`. There has been much discussion about the shortcomings of this class (for example, it does not necessarily store its elements as a contiguous array).

Answer the following questions:

a) Determine which functionality it supports compared to the two bitset classes discussed here.

b) Create a function to compute the intersection of two instances `std::vector<bool>`.

Having completed the exercise will probably convince you that it is better to use bitset classes instead of `std::vector<bool>`.

2. (Creating Object Adapters for Bitset, Compile-Time (Composition))

In this exercise we create a compile-time *bit matrix* (call it `BitMatrix<N, M>`) consisting of `N` rows and `M` columns all of whose elements are bits. Some requirements are:

- The chosen data structure must be efficient (for example, accessing the elements).
- Its interface must have the same look and feel as that of `std::bitset<>`.
- We wish to reuse as much code as possible.
- It must be generic enough to support a range of applications in different domains (for example, computer graphics and its many applications).

Answer the following questions:

a) Determine which data structure to use in order to implement `BitMatrix<N, M>`, for example as a nested array `std::array<std::bitset<M>, N>` or a one-dimensional array `std::bitset<N*M>`. Which choice is 'optimal' is for you to decide. You need to determine which criteria to use (for example, performance and maintainability).

b) Constructors need to be created. Use the same defaults as with `std::bitset<M>`.

c) Implement the following operators for all rows in the matrix and for a given row in the matrix:

- Set/reset all bits.
- Flip the bits.
- Test if none, all or any bits are set.
- Access the elements.
- Count the number of set bits.

d) Create member functions for OR, XOR and AND Boolean operations on bit matrices.

e) Consider creating the matrix as a derived class of `bitset`.

3. (Comparing Singly and Doubly Linked Lists)

In this exercise we carry out some operations on `std::list<double>` (call it A for convenience) and `std::forward_list<double>` (call it B).

Answer the following questions:

a) Create instances of A and B with *n* elements, where *n* is typically a large number (for example, at least a million).

b) Insert an element at every alternate position in the lists A and B.

c) Remove all even elements from the lists A and B.

d) Sort and reverse the lists A and B.

e) Create an instance of B with *n* elements all of whose values are the same value `val`. Compare the run-time efficiency of using a single call to remove all the elements with value `val` and removing elements one by one.

Use the stopwatch class to measure the relative run-time performance in all cases.

CHAPTER 11

Lattice Models Fundamental Data Structures and Algorithms

11.1 INTRODUCTION AND OBJECTIVES

The next two chapters are concerned with developing numerical methods and code to price options using *lattice methods*, in particular the *binomial* and *trinomial* methods. We first discuss the underlying data structures that are used in the pricing models. We then describe the pricing of one-factor European and American options using the binomial and trinomial methods.

In this chapter we concentrate on *software design* issues. It is important to decide how to design lattice data structures and to determine which C++ features to use in order to promote code reusability and maintainability, for example:

- Promoting code reusability and adaptability using the *Layers* system pattern (POSA, 1996).
- Using functionality from STL (for example, algorithms and `std::function<>` and new data structures such as `std::tuple<>`) to promote code reusability. Furthermore, we can use lambda functions to configure applications.
- Comparing the quality and accuracy of the solutions in this chapter with previous efforts to price options using the binomial method (for example, Clewlow and Strickland, 1998; Duffy, 2006; Hull, 2006). This can be seen as one step in a process of *continuous improvement*. It is easy to write software that produces a result, but it is more difficult to adapt that software to support new requirements.
- Some initial examples of use. Many of the design principles presented here are applicable to other kinds of C++ applications. In particular, we can run the option pricer code and use the option prices as baseline examples for PDE/FDM solvers. For example, when testing a new PDE/FDM solver we can use the option prices from a lattice model to check the corresponding prices.
- Using popular libraries such as Boost C++ *uBLAS* and *Eigen*.

We emphasise that the exercises and projects form an integral part of the learning process. We recommend that you at least read the exercises. Ideally, implementing the exercises in C++ will be even more beneficial.

This chapter is less concerned with the mathematical background to lattice models and more with how to design a flexible software framework (in particular by applying the *Layers* pattern). For background information on lattice models and their applications to option pricing, see Clewlow and Strickland (1998) and Hull (2006). In later chapters we shall apply parallel programming libraries and C++11 concurrency to option pricing using the binomial method.

We feel that this chapter is particularly useful for novice programmers with limited experience of *software maintenance*.

11.2 BACKGROUND AND CURRENT APPROACHES TO LATTICE MODELLING

Lattice models are well known and popular in computational finance. They have been documented in the literature and accompanying code in languages such as C, C++, Matlab and C# is available.

Some initial remarks concerning flexibility of previous software versions are:

- Tight coupling between data, models and algorithms, for example designing a lattice structure that uses the drift and diffusion functions of an SDE.
- Overuse of inheritance and subtype polymorphism to create difficult-to-maintain class hierarchies. We mitigate the ensuing problems by using *parametric polymorphism* and *lambda functions*.
- Using classes that implement lattice parameters, for example Jarrow–Rudd or CRR parameters. These classes are too top heavy for the job at hand. An alternative is to define these parameters using a lambda function that returns a tuple or a fixed-size array.
- Less-than-optimal use of the GOF design patterns. See Figure 11.1 that shows the UML design of a binomial pricer (Duffy, 2006A). We use the *Strategy* pattern to model the different ways to generate lattice parameters. We are now interested in next-generation design patterns by examining the problem in a different way.
- The software was not created with change and maintainability in mind.

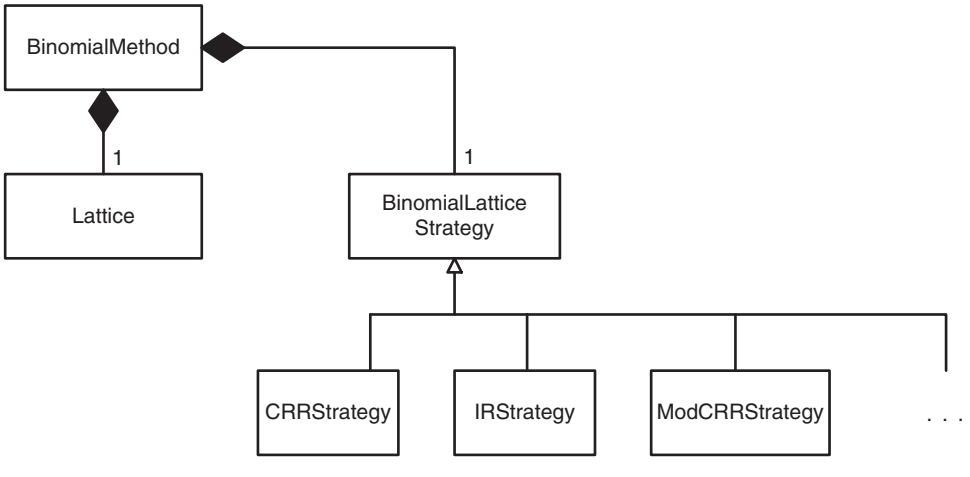


FIGURE 11.1 Models and strategies

For details, see Duffy (2006A).

11.3 NEW REQUIREMENTS AND USE CASES

We describe the main functionality that we can expect in the following two chapters on the application of lattice methods to problems in computational finance. In general, we focus on the applications in Clewlow and Strickland (1998) and from related literature. The goal is to group functionality into several categories:

- U1: Price a range of one-factor option pricing problems.
- U2: Support for binomial and trinomial lattices.
- U3: Computing hedge sensitivities with lattices.
- U4: Constructing binomial trees for the short rate.

The above list provides us with a foundation upon which to design a software framework in C++. The main goals of studying lattice models in this way are:

- Lattice models are well known and we implement them using design patterns and new features in C++. The examples that we discuss in the following two chapters have been studied for at least 20 years. The focus in this book is on designing flexible software frameworks rather than (necessarily) discussing state-of-the-art pricing models or the mathematical foundations of the binomial method.
- Chapters 11 and 12 are the first chapters in this book that discuss option pricing models. In this sense they represent a bridge between the first ten chapters and the topics in later chapters. In particular, lambda functions, tuples and functional programming models will play an important role as we progress in this book.
- We produce code to price a range of one-factor options and the results can be used to test the accuracy of the results based on other numerical approaches such as Monte Carlo and finite difference methods. Some code that we create here can also be reused in these applications.
- We make a concerted effort to show how to design a software framework that is easy to customise. We can also use the software in other applications not directly related to computational finance, for example polynomial and rational function interpolation in numerical analysis.

11.4 A NEW DESIGN APPROACH: A LAYERED APPROACH

In this section we show how we designed a software framework to price options using lattice models. The approach is based on a number of fundamental design principles:

a) An application is built from loosely coupled and cohesive components (and objects).
b) Each component has one major responsibility (it satisfies the SRP).
c) We build functionality in increasing levels of abstraction.

To this end, we shall see how the *Layers* design pattern (POSA, 1996) satisfies the above three design requirements, thus allowing us to create flexible software systems. Before we do so we discuss a specific motivational example and its implementation in C++ (based on the pseudo-code in Clewlow and Strickland, 1998, p. 41). In other words, we price an *American*

down-and-out call option. This is an American call option with the requirement that if the asset price falls below a predetermined barrier level H then the option ceases to exist (it *knocks out*) and it pays nothing. In some cases the option can pay a predetermined *cash rebate* but we do not discuss this topic here. We price this option using an additive binomial tree as described in Clewlow and Strickland (1998).

The following steps need to be followed:

1. Create the asset price mesh/lattice.
2. Initialise the option value at expiration.
3. Step back in the lattice and apply the barrier check and early-exercise conditions.
4. Compute the option price.

The code for this problem is:

```
#include <cmath>
#include <algorithm>
#include <vector>
#include <iostream>

double price(double K, double T, double S, double sig, double r,
             double H,int N)
{

    // Initialise coefficients based on the Trigeorgis approach
    // N = Number of intervals

    double dt = T/N;
    double nu = r - 0.5*sig*sig;

    // Up and down jumps
    double dxu = std::sqrt(sig*sig*dt + (nu*dt)*(nu*dt));
    double dxd = -dxu;

    // Corresponding probabilities
    double pu = 0.5 + 0.5*(nu*dt/dxu);
    double pd = 1.0 - pu;

    // Precompute constants
    double disc = std::exp(-r*dt);
    double dpu = disc*pu;
    double dpd = disc*pd;
    double edxud = std::exp(dxu - dxd);
    double edxd = std::exp(dxd);

    // Initialise asset prices
    std::vector<double> St(N+1);
    St[0] = S*std::exp(N*dxd);
    for (std::size_t j = 1; j < St.size(); ++j)
```

```
    {
        St[j] = edxud*St[j-1];
    }

    // Option value at maturity (t = N)
    std::vector<double> C(N+1);
    for (std::size_t j = 0; j < C.size(); ++j)
    {
    if (St[j] > H)
        {
           C[j] = std::max<double>(St[j] - K, 0.0);
        }
        else
        {
          C[j] = 0.0;
        }
    }

    // Backwards induction phase
    for (int i = N-1; i >= 0; --i) // Can't use std::size_t as index
    {
       for (std::size_t j = 0; j <= i; ++j)
       {
          St[j] /= edxd;

       if (St[j] > H)
       {
          C[j] = dpd*C[j] + dpu*C[j+1];

          // Early exercise condition
          C[j] = std::max<double>(C[j], St[j] - K);
       }
       else
       {
          C[j] = 0.0;
       }
    }

  }

  // Early exercise down and out call
  return C[0];
}
```

A simple test program is:

```
int main()
{

    double K = 100.0;
    double S = 100.0;
```

```
    double T = 1.0;
    double r = 0.06;
    double q = 0.0;
    double sig = 0.2;

    int N = 3;


    double H = 95;
    double optionPrice = price(K,T,S,sig,r,H,N);

    std::cout << "Price:" << optionPrice << std::endl;

    return 0;
}
```

The above code is not very flexible and we shall see how to generalise it. It was not created with maintainability in mind. See also Exercise 1.

11.4.1 Layers System Pattern

It is clear from the code example in Section 11.4 that we need some kind of strategy in order to create a flexible software framework that can support a range of option pricing problems that we implement using lattice methods. We have already seen that the above code does not satisfy our requirements and thus a new approach is needed. In Duffy (2006A) we used the object-oriented model in combination with the GOF design patterns and this approach offered a certain amount of flexibility. However, the design had a number of limitations, such as:

- It was object based and it used subtype polymorphism and class hierarchies. Using object technology is not always needed.
- Tight coupling between the classes.
- It was not always clear how to configure and initialise the application due to the lack of suitable patterns and possible limitations in the C++ language itself.
- Proliferation in the number of classes as new requirements is added to the software framework. We created classes and class hierarchies but function objects and lambda functions would possibly have been more suitable.

Notwithstanding, the design approach in Duffy (2006A) is an improvement to the hard-wired solution presented in Section 11.4. The code can be used as requirements for a more flexible design.

We now propose a design based on the *Layers* system pattern (for a full discussion, see POSA, 1996). The current problem can be modelled as a set of loosely coupled and cohesive components arranged in the form of a *uses hierarchy*; components at a given level use the services of components at the same or a lower level but a component at a given level has no knowledge of the components at higher levels. This approach promotes loose coupling and

hence makes it easier to extend the functionality. In the current case we have decided to design the system using a *three-layer regime*:

- *Layer 1*: this is the lowest layer and it contains the structures and containers that hold the data needed by higher-level components. We implement the containers as *adapters* of standard STL containers such as vectors, sets, unordered containers, containers from other libraries such as *Boost uBLAS*, *Eigen* and user-defined data structures. We define the interface of the adapter containers to meet the needs of higher-level components. The advantage of this approach is that we can change the internal representation of Layer 1 containers without breaking other interfaces at higher levels. This is an example of *information hiding*.

  The containers in Layer 1 are generic, hence they can be instantiated to suit various user requirements, for example lattices that can be used as the underlying data structure in numerical quadrature and numerical interpolation.
- *Layer 2*: the components in this layer use the services of the components in Layer 1. They model behaviour, for example the well-known *forward and backward induction algorithms* to iterate in lattices. In general, we say that the generic functionality in this layer implements algorithms that apply to the containers in Layer 1. This functionality can be implemented using GOF patterns (for example, *Strategy* or *Visitor* as shown in Figure 11.1) or by using free functions in combination with function objects or STL function wrappers.
- *Layer 3*: this is the layer that contains code to initialise and configure the components in Layers 1 and 2. We have found that the use of lambda functions leads to readable and maintainable code. It is also in this layer that we can use the GOF *Builder* pattern to configure an application from its constituent components.

We now discuss the specifics of the functionality in each of these layers.

11.4.2 Layer 1: Basic Lattice Data Structures

We have decided to model lattices as first-class abstract data types (ADTs) instead of the more ad-hoc solution discussed in Section 11.4. In other words, we create a generic class that models two-dimensional *jagged arrays*, a concept that is supported in C#, for example. C++ does not support such data structures which means that we will need to use a matrix library that supports upper- and lower-triangular matrices (such as *Eigen* or *Boost uBLAS*) or we can choose to create our own lattice classes in C++. In the current case we use *nested vector classes*. We discuss this latter solution in this section.

Since we wish to create lattice ADTs that are as reusable as possible we need to determine which of their defining attributes should be generic:

- A1: The row index space (models *time*).
- A2: The column index space (models *underlying asset*).
- A3: The data type that the lattice ADT holds.

To reduce the scope, we concentrate on binomial and trinomial lattices and in particular we discuss the former type in more detail. We have two lattice ADTs, the first of which can be

used in a number of option pricing problems (attribute A1 uses integral indexes) and a more generalised lattice in which the index space for attribute A1 can be a more general data type, such as a Boost date, for example. This level of flexibility is needed when we wish to work directly with dates and with Bermudan option pricing problems, for example. Using dates as the row index space is not discussed in this book.

The *simplest lattice* ADT uses integers as indexes. We use nested STL vectors to implement it:

```
// The Node class contains the values of interest.
// LatticeType == 2 for binomial,3 for trinomial.

template <class Node, int LatticeType> class Lattice
{ // Generic lattice class

private:

    // Implement as a full nested vector class
    std::vector<std::vector<Node> > tree;

    // ...
};
```

The *generalised lattice* ADT has an extra template parameter that supports non-integral indexes for attribute A1:

```
template <typename TimeAxis, typename Node, int LatticeType>
          class GeneralisedLattice
          { // Generic lattice class

    // Implement as an STL map
    std::map<TimeAxis, std::vector<Node> > tree;

    // ...
};
```

Here we implement the index space as a map.

These two lattice classes have similar interfaces that we summarise here:

- Constructors, for example to create a lattice with a given number of rows (time steps).
- Accessing data at a given row or accessing data at a given row and at a given column.
- Useful member functions when iterating in a lattice.

We now give some initial examples. We first create a binomial lattice and a trinomial lattice:

```
 // Create basic lattices
const int TYPEB = 2;    // BinomialLatticeType;
```

```
const int TYPET = 3;    // Trinomial Type

int depth = 10;         // Number of rows == depth + 1

typedef double Node;
Lattice<Node,TYPEB> lattice1(depth, 0.0);
Lattice<Node,TYPET> lattice2(depth, 1.0);

LatticeMechanisms::print(lattice1);
LatticeMechanisms::print(lattice2);
```

For convenience, we have created a simple function to print a lattice:

```
namespace LatticeMechanisms
{

    template <class Node, int LatticeType>
       void print(const Lattice<Node,LatticeType>& source)
    {
       std::cout << "\n";
       for (std::size_t j = source.MinIndex(); j <= source.MaxIndex(); ++j)
       {
           std::cout << "\nBranch Number " << j << ": [";
           for (std::size_t i = 0; i<source[j].size(); ++i)
           {
              std::cout << source[j][i] << ", ";
           }
           std::cout << "]";
       }
       std::cout << "\n";
    }
    // ...
}
```

Here we see how to access the individual elements of the lattice using the C++ indexing operator. We have also overloaded the operator `()` to reflect standard usage with matrix libraries. As an example, we create code to show how to generate the values of the famous *Pascal triangle*, which is a triangular array of the *binomial coefficients*. You can check that the sum of the entries in the nth row of the triangle is indeed 2^n:

```
// In this case we build the Pascal Triangle
std::cout << "\nPascal Triangle\n";

// Apex value of triangle
lattice1[0][0] = 1.0;

// Generate the triangle
for (std::size_t j = lattice1.MinIndex()+1; j <= lattice1.MaxIndex(); ++j)
{
```

```
        // Generate edge values
        lattice1(j,0) = lattice1(j-1,0);
        lattice1(j,lattice1.MaxIndex(j)) = lattice1(j-1,lattice1.
        MaxIndex(j-1));

        // Generate the interior values
        for (std::size_t i = 1; i < lattice1[j].size()-1; ++i)
        {
           lattice1(j,i)=lattice1(j-1,i)+lattice1(j-1,i-1);
        }
    }
    LatticeMechanisms::print(lattice1);
```

These examples show how to create a lattice and access and modify its values. You can run the code and examine the output.

We now create a generalised lattice whose row index set is a Boost *date* type. We first create a set of *fixing dates* that will form the row axis:

```
#include <boost/date_time/gregorian/gregorian.hpp>

// Dates as axis
namespace Dates = boost::gregorian;

// Simple dates for illustrative purposes
std::set<Dates::date> fixingDates;
Dates::date myDate(2014, Dates::May, 16);
fixingDates.insert(myDate);
Dates::date myDate2(2014, Dates::May, 26);
fixingDates.insert(myDate2);
Dates::date myDate3(2014, Dates::Jun, 2);
fixingDates.insert(myDate3);
Dates::date myDate4(2014, Dates::Jun, 12);
fixingDates.insert(myDate4);
```

We then create a generalised lattice based on these fixing dates:

```
GeneralisedLattice<Dates::date,double, 3> lattice2(fixingDates);
GeneralisedLatticeMechanisms::print(lattice2);
```

A useful project would be to extend the functionability to support dates.

11.4.3 Layer 2: Operations on Lattices

The functionality in Layer 1 is mainly concerned with generic classes that model lattice structures. In general, the focus lies on allocating memory for these structures and using operator overloading to access the elements of the lattices. This functionality is consistent with the SRP. These ADTs can be used in many kinds of applications (in this book we are mainly interested in option pricing). We develop code to iterate in lattices, use lattices

in applications and access the elements in lattices. To this end, we create generic code in Layer 2 that realises these features. In the past (Duffy, 2006A) we used an object-oriented approach that led to code that was somewhat inflexible and that introduced coupling into the framework. With the advent of C++11 we now have the opportunity to improve the genericity and maintainability of the framework. In this chapter we discuss the following features:

- Tuples that we use as return types of functions or as input arguments to functions.
- The function type wrapper `std::function<>`. This is a *signature-based feature* and it reduces the coupling between components because it is a specification and its implementations have been postponed to code in Layer 3. We have already seen in Chapters 3 and 4 that function type wrappers can be implemented as *callable objects* such as free functions, static member functions, object member functions (in combination with `std::bind<>`), function objects and lambda functions.
- Lambda functions that are used in Layer 3 of the framework.
- Using the keyword `auto` for automatic type deduction. This feature helps to make the code more readable and it takes the drudgery out of programming.

We also remark that it is possible to create function wrappers whose input arguments and return type are tuples. It is also possible to create tuples whose elements are function wrappers.

Having created a lattice, we then discuss algorithms that use or modify its elements. In order to reduce the scope, we focus on the well-known *forward* and *backward induction algorithms* (see Clewlow and Strickland, 1998; Duffy, 2006A; Duffy and Germani, 2013). In general, we access and modify binomial and trinomial lattices in different ways, for example:

- U1: Forward induction (create the lattice using *up* and *down* parameters).
- U2: Compute the payoff at each final node (maturity/expiration).
- U3: Backward induction (compute the option value at earlier nodes starting from expiration).

Some other use cases are:

- U4: Computing hedge sensitivities.
- U5: Lattice models for dividend-paying assets.
- U6: Pricing path-dependent options.
- U7: Computing the price of American put options.
- U8: Two-factor lattice methods.

To this end, we have created four free functions to realise forward and backward induction, one set of functions for plain options and another set for American options. We also make a distinction between binomial and trinomial lattices. There are opportunities to make the code more elegant. We also make use of function wrappers and tuples.

The code that implements forward induction is:

```
const int TYPEB = 2; // BinomialLatticeType

template <class Node>
   void ForwardInduction(Lattice<Node, TYPEB>& lattice,
   const std::function<std::tuple<Node, Node>,(const Node& node)>&
   generator, const Node& rootValue)
{
      // Initialise the root value lattice[0][0]
      std::size_t si = lattice.MinIndex();
      lattice[si][0] = rootValue;

      std::tuple<Node, Node> tuple;

      // Loop from min index to max, i.e. (min, end]
      for (std::size_t n = lattice.MinIndex() + 1; n <=
      lattice.MaxIndex(); n++)
      {
         for (std::size_t i=0;i<lattice[n-1].size(); i++)
         {
            tuple = generator(lattice[n-1][i]);
            lattice[n][i] = std::get<0>(tuple);
            lattice[n][i+1] = std::get<1>(tuple);
         }
      }
}
```

We have also coded the backward induction algorithm that computes a value at the apex (time = 0) of the lattice starting from its base (time = T). The code in the case of a binomial lattice is:

```
template <class Node>
    Node BackwardInduction(Lattice<Node, TYPEB>& lattice,
    const std::function<Node(const Node& upper, const Node& lower)>&
    generator, const std::function<Node (const Node& node)>&
    endCondition)
{

        std::size_t ei = lattice.MaxIndex();

        // Apply the end condition function to the existing values
        // of the base of the tree. In finance, this would be the
        // payoff at maturity.
        for (std::size_t i = 0; i < lattice[ei].size(); ++i)
        {
           lattice[ei][i] = endCondition(lattice[ei][i]);
        }
```

```
        // Loop from the max index to the start (min) index. Open
        //interval (max, min)
        for (std::size_t n = lattice.MaxIndex() - 1;n > 0;--n)
        {

        for (int i = 0; i < lattice[n].size(); ++i)
        {
           lattice[n][i] = generator (lattice[n+1][i+1],
           lattice[n+1][i]);
        }
    }

    // Kind of bug in C++ when size_t hits 0 (case n = 0)
    for (std::size_t i = 0; i < lattice[0].size(); ++i)
    {
       lattice[0][i] = generator (lattice[1][i+1],
       lattice[1][i]);
    }

    std::size_t si = lattice.MinIndex();
    return lattice[si][0];
}
```

As a simple case, we consider creating a *toy lattice* and specific algorithms for forward and backward induction as well as the payoff condition at the base of the lattice:

```
// Simple forward and backward induction processes
std::tuple<double, double> SimpleForwardInductionStep(const double& value)
{
   // Parameters to ensure recombining lattice (test purposes)
   double u = 2.0;
   double d= 1.0/u;

   return std::make_tuple(u*value, d*value);
}

double SimpleBackwardInductionStep(double upper, double lower)
{
   // Simple backward induction algorithm
   double p = 0.5;

   return p*upper + (1.0 - p)*lower;
}

double SimpleEndCondition(double S)
{
   return S;
}
```

A test program is:

```
// Binomial lattice, simple case
const int typeB = 2;             // BinomialLatticeType
int depth = 2;

// Create basic structure
Lattice<double,typeB> lattice(depth, 0.0);
LatticeMechanisms::print(lattice);

// Fill node values of lattice
double rootValue = 1.0;
LatticeMechanisms::ForwardInduction<double>
                    (lattice, SimpleForwardInductionStep, rootValue);
LatticeMechanisms::print(lattice);

std::cout << "Backward: " <<
LatticeMechanisms::BackwardInduction<double>
                    (lattice, SimpleBackwardInductionStep,
                    SimpleEndCondition);
LatticeMechanisms::print(lattice);
```

It is easy to check (for example, using pencil and paper) that the lattice has the following entries after having called the forward induction algorithm:

```
0: [1]
1: [2, 0.5]
2: [4,1,0.25]
```

and

```
0: [1.5625]
1: [2.5, 0.625]
2: [4,1,0.25]
```

after having called the backward induction algorithm. We shall see more financially relevant examples in Chapter 12. The current example is for motivation.

11.4.4 Layer 3: Application Configuration

The first two layers of the framework are generic. In general, we have attempted to postpone specific cases for as long as possible. In this book we are interested in binomial option pricers and it is in Layer 3 that we shall use some well-known algorithms and models (for example, Clewlow and Strickland, 1998; Hull, 2006).

In Layer 3 we configure and initialise the parameters and functions for an application, for example:

- The option parameters.
- The kind of payoff.
- Whether to use the binomial method or trinomial method.
- Accommodating extra features such as early exercise and barriers, for example.

From a software viewpoint it is possible to configure an application in many ways. An important requirement is to determine how maintainable the code in Layer 3 should be. In Duffy (2006A) we used class hierarchies to add flexibility into the configuration process. A more flexible solution is to use the *Builder* pattern (see GOF, 1995) that configures the application's object network. In this section, however, we avoid creating class hierarchies and instead we use function objects, lambda functions and function wrappers to configure applications. This approach leads to readable and understandable code in our opinion. We now give an initial example to price European and American options.

11.5 INITIAL '101' EXAMPLES OF OPTION PRICING

We take a complete example of pricing European and American options using the Cox–Rubinstein–Ross (CRR) method (see Clewlow and Strickland, 1998). We compute the up and down jump size and probability parameters and furthermore we encapsulate these parameters in a class called `CRRLatticeAlgorithms`. This class is a *function object* because it has two operators that are compatible with the function wrappers needed by the algorithms in Layer 2 of the framework:

```
class CRRLatticeAlgorithms
{ // The function object that implements forward and backward
  // induction algorithms based on CRR

private:
   double u;
   double d;
   double p;
   double discounting;

public:
   CRRLatticeAlgorithms(const OptionData& option, double dt)
   {

      double s = option.sig;
      double r = option.r;

      double R1 = (r - 0.5 * s * s) * dt;
      double R2 = s * std::sqrt(dt);
```

```
        u = std::exp(R1 + R2);
        d = std::exp(R1 - R2);

        discounting = std::exp(- r*dt);

        p = 0.5;

    }

    // Forward induction algorithm
    // Function object
    std::tuple<double, double> operator () (double value)
    {
        return std::make_tuple(u*value, d*value);
    }

    // Backward induction algorithm
    // Function object
    double operator()(double upper, double lower)
    {

        return discounting * (p*upper + (1.0 - p)*lower);
    }

};
```

We now create option data and a CRR lattice:

```
// Option data
OptionData opt;

opt.K = 65.0;
opt.T = 0.25;
opt.r = 0.08;
opt.q = 0.0;
opt.sig = 0.3;

int N = 101;
double dt = opt.T / double(N);

// Implements forward and backward induction
LatticeMechanisms::CRRLatticeAlgorithms algorithm(opt, dt);

// Create basic structure for plain options
Lattice<double,2> lattice(N, 0.0);
```

We now build the lattice and compute the plain option price as follows:

```
double rootVal = 60.0; // S(0)
LatticeMechanisms::ForwardInduction<double>(lattice, algorithm, rootVal);
```

```
// Payoff as a lambda function
double K = opt.K;
auto Payoff = [&K] (double S)-> double {return std::max<double>
(K - S, 0.0);};

// Price a plain option
double res = LatticeMechanisms::BackwardInduction<double>(lattice,
algorithm, Payoff);
std::cout << "Plain Option price, classic #1: ";
```

We now price the corresponding American option price as follows:

```
// Create basic structure for early exercise options
Lattice<double,2> lattice2(lattice);

// The early exercise constraint
auto AmericanAdjuster = [&Payoff](double& V, double S)->void
{ // e.g. early exercise

   V = std::max<double>(V, Payoff(S));
};

// Price an early exercise option
double res2 = LatticeMechanisms::BackwardInduction<double>
             (lattice2, algorithm, Payoff,AmericanAdjuster);
             std::cout << "Early exercise option price";
```

Notice that we use a lambda function to implement the early-exercise constraint.

You can run the code from the distribution medium and check the answers for a range of parameter values. We recommend that you experiment with the code, modify it and try to get a hands-on feeling for how the framework works.

11.6 ADVANTAGES OF SOFTWARE LAYERING

The three-layer design approach offers a number of advantages as far as software maintainability, understanding and extendibility are concerned. First, we have a well-known and defined process (the *Layers* pattern) on which to build the C++ lattice framework. It also gives developers a vocabulary that allows them to communicate with each other (and with themselves!) at a higher level than just (possibly undocumented) blocks of code. Second, it reduces the coupling between software components and modules. Third, it improves the cohesion of the components in each layer because their operations are closely related and hence do not depend on the components in other layers. Finally, it is relatively easy to add new modules to the framework. We now discuss these topics in more detail from the viewpoint of the quality characteristics of the ISO 9126 standard. The guidelines can also be applied to other problems.

11.6.1 Maintainability

Modifications to the software should be relatively easy to diagnose, analyse and test. Furthermore, the amount of effort to modify the code or remove faults is easier than with ad-hoc coding techniques. Finally, the framework tends to remain stable when modifications are made to it. In general, we can attribute these advantages to a number of factors, such as:

- The software objects and algorithms in Layers 1 and 2 are domain independent and can be reused by the applications in Layer 3. This means that only the software in this layer needs to be written and tested. In practice, new applications may demand modifications to existing code or we may need to create new modules to support the applications.
- The modules in each layer have well-defined interfaces and their internals are hidden from modules in higher layers. For example, the lattice classes have operations to access the elements of two-dimensional jagged array structures that are used by various algorithms.
- *The multiparadigm programming model* (templates, object oriented, functional) adds to the flexibility of the framework because we can choose the most appropriate model to use at a given level of abstraction. For example, template classes are used as ADTs in Layer 1 while lambda functions are very useful when configuring applications and objects in Layer 3.
- A number of features in C++11 add to the maintainability of the framework, in particular tuples, lambda functions and `std::function<>`.

  Ad-hoc solutions are easy to program but are difficult to maintain while a layered approach demands more up-front thinking. However, the resulting code tends to be more maintainable.

11.6.2 Functionality

This characteristic encompasses issues relating to the capability of the framework to satisfy user needs. In general, this characteristic is concerned with how suitable the framework is for a range of application types and problems. For example, issues such as *accuracy* (producing correct and agreed results) and the ability to interoperate with other systems are important.

One of the main problems with an ad-hoc approach is that adding new functionality or modifying existing functionality becomes more difficult as a system evolves. A layered approach tends to be more adaptable due to loose coupling. In particular, other benefits are (POSA, 1996):

- *Reuse of layers*: the components in a given layer can be reused in other contexts provided they have well-defined interfaces. In many cases *black-box reuse* of layers can reduce development time and also decrease the number of defects. In the current framework we see that the components in Layers 1 and 2 are candidates for reuse.
- *Support for standardisation*: in this case, we recommend designing data structures and algorithms based on the STL standard, for example STL-compatible containers. Having done that we can then use STL algorithms with the new containers without modification.

- *Dependencies are kept local*: a layer is a *black box* for the layers above it. For example, the algorithms in Layer 2 use the lattice structures from Layer 1 without knowing (or having to know) how the internals of these structures work.
- *Exchangeability*: with some effort it is possible to replace implementations in one layer by other semantically equivalent implementations. For example, we can use the *Adapter* pattern to adapt the interface of a component at design-time. For applications that need dynamic exchange of interfaces we can use the *Bridge* pattern (see GOF, 1995).

An example in the current framework is the case of the lattice data structures that we can implement using STL nested vectors or Boost uBLAS triangular matrices, for example.

The *Layers* design pattern does have a number of potential drawbacks relating to code maintainability and run-time performance (POSA, 1996):

- *Lower efficiency*: higher-level layers use the services and data from lower-level layers. All relevant data must be transferred up the calling chain through a number of intermediate layers. The data may be transformed several times. This is particularly true of error messages that can originate in lower (hardware) layers and then get transformed to error messages that are understandable to clients in higher layers.
- *Cascades of changing behaviour*: this problem is caused when the behaviour in a layer changes. For example, an apparently local change in a layer may demand having to change code in higher layers. An example is the modifications that need to be executed when we decide to use a new data structure to represent the lattice data in the application. In the worst case, it may demand a rewrite of the application.
- *Unnecessary work*: several layers may perform the same operations on data. This leads to code that may impact run-time performance, for example code that checks the validity of input data more than once.
- *Difficulty of establishing the correct granularity of layers*: having too few or too many layers can be disadvantageous. In the former case the potential for reusability, changeability and portability cannot be exploited while in the latter case much overhead and unnecessary complexity is introduced into the design. In this chapter we introduced a three-layer model but four-layer and five-layer models are possible.

Finally, the *Layers* design pattern can be applied to many kinds of applications in which functionality is built in increasing levels of abstraction. You need to analyse its feasibility by examining the advantages and disadvantages of this pattern for your own applications. A three-layer design is probably the easiest solution but for larger production systems you may need to add one or more extra layers.

11.6.3 Efficiency

We have designed the data structures in the framework so that they only need to allocate memory that will actually be used by algorithms. Thus, instead of employing redundant *dense matrices* to store the data in a lattice we employ *jagged arrays* or *triangular matrices*, for example:

```
// Implement as a full nested vector class
 std::vector<std::vector<Node> > tree;
```

or

```
namespace u = boost::numeric::ublas;
u::triangular_matrix<double, u::lower> lattice(N,N);
```

We notice that some literature uses dense matrices to model lattices, which is wasteful of memory and can result in degraded performance. A sub-characteristic of efficiency is concerned with the *resource behaviour* of algorithms.

The second sub-characteristic is concerned with the *time behaviour* of algorithms. These algorithms usually access the elements of lattice structures that we can implement using vectors, maps, sets or unordered sets. We can use *Complexity Analysis*, as discussed in Chapter 17, to predict the performance of these algorithms.

11.7 IMPROVING EFFICIENCY AND RELIABILITY

In Section 11.4 we produced code to price an American down-and-out call option. One of the bottlenecks is that the code used to create the lattice for the underlying (asset) is hard-wired in the algorithm. We recall:

```
// Initialise asset prices
std::vector<double> St(N+1);
St[0] = S*std::exp(N*dxd);
for (std::size_t j = 1; j < St.size(); ++j)
{
    St[j] = edxud*St[j-1];
}
```

We have resolved this problem by creating a lattice ADT that holds the asset values. However, the backward induction algorithm overwrites the values in the asset lattice. In some cases we would like to create one lattice for the underlying asset and then price several options based on the data in the lattice. Realising this use case implies that we must modify the code in the framework. In this section we develop a new backward induction algorithm that creates a lattice from an existing read-only lattice. The code is:

```
// Backward induction to update a lattice L2 based on an input
// lattice and a generator.
template <typename Node>
      Node BackwardInduction(const Lattice<Node, TYPEB>& lattice,
      Lattice<Node, TYPEB>& L2,
           const std::function<Node (const Node& upper,
           const Node& lower)>& generator,
           const std::function<Node (const Node& node)>& endCondition,
           const std::function<void (Node& node, const Node& val)>&
           constraintAdjuster)
{

      modifyBaseValues(L2, endCondition);
```

```
        double t1 = 0.0;
        // Loop from the max index to the start (min) index
        for (std::size_t n = L2.MaxIndex() - 1; n > 0; n--)
        {

           for (int i = 0; i < L2[n].size(); i++)
           {
              t1 = lattice[n][i];
              L2[n][i] = generator(L2[n+1][i+1], L2[n+1][i]);
              constraintAdjuster(L2[n][i], t1);
           }
        }

        // Case n = 0 treated separately for compiler reasons
        for (std::size_t i = 0; i < L2[0].size(); ++i)
        {
           t1 = lattice[0][i];
           L2[0][i] = generator(L2[1][i+1], L2[1][i]);
           constraintAdjuster(L2[0][i],t1);
        }

        std::size_t si = L2.MinIndex();
        return L2[si][0];
  }
```

To show how to use this algorithm we create an asset lattice and four lattices that will hold computed option prices. First, the CRR-based asset lattice is:

```
// Option data
OptionData opt;

opt.K = 65.0;
opt.T = 0.25;
opt.r = 0.08;
opt.q = 0.0;
opt.sig = 0.3;

int N = 100;
double dt = opt.T / static-cast<double>(N);

// Implements forward and backward induction
LatticeMechanisms::CRRLatticeAlgorithms algo(opt, dt);

// Create basic asset lattice
Lattice<double,2> asset(N, 0.0);    // init
double rootVal = 60.0;              // S(0)
LatticeMechanisms::ForwardInduction<double> (asset, algo, rootVal);
// build the lattice
```

The four option types are plain put and call, American put and call. We define the payoffs and constraint checkers as lambda functions. In order to keep the code consistent, we create a special constraint checker for plain options (it has an empty body):

```
// Kinds of payoff
double K = opt.K;
auto PutPayoff = [&K] (double S)-> double {return std::max<double>
(K - S, 0.0);};
auto CallPayoff = [&K] (double S)-> double {return std::max<double>
(S - K, 0.0);};

// American early exercise constraint
auto AmericanPutAdjuster = [&PutPayoff](double& V, double S)->void
{ // e.g. early exercise

   V = std::max<double>(V, PutPayoff(S));
};

auto AmericanCallAdjuster = [&CallPayoff](double& V, double S)->void
{ // e.g. early exercise

   V = std::max<double>(V, CallPayoff(S));
};

auto EmptyAdjuster = [](double& V, double S)->void
{ // No action executed at a node

   // Do nothing
};
```

We now create four lattices (one for each option type):

```
Lattice<double,2> euroPut(N, 0.0);
LatticeMechanisms::ForwardInduction<double>(euroPut, algo, rootVal);
Lattice<double,2> euroCall(N, 0.0);
LatticeMechanisms::ForwardInduction<double>(euroCall, algo, rootVal);
Lattice<double,2> earlyPut(N, 0.0);
LatticeMechanisms::ForwardInduction<double>(earlyPut, algo, rootVal);
Lattice<double,2> earlyCall(N, 0.0);
LatticeMechanisms::ForwardInduction<double>(earlyCall, algo, rootVal);
```

We now can call the backward induction algorithm to compute the option prices:

```
// Compute option prices
using namespace LatticeMechanisms;
double euroPutPrice = BackwardInduction<double>
                      (asset, euroPut, algo, PutPayoff, EmptyAdjuster);
std::cout << "Euro put: " << euroPutPrice;

double euroCallPrice = BackwardInduction<double>
                      (asset, euroCall, algo, CallPayoff, EmptyAdjuster);
```

```
std::cout << "Euro call: " << euroCallPrice;

// Price an early exercise option
double earlyPutPrice = BackwardInduction<double>
                       (asset,earlyPut, algo, PutPayoff,
                        AmericanPutAdjuster);
std::cout << "Early put: " << earlyPutPrice;

double earlyCallPrice = BackwardInduction<double>
                       (asset,earlyCall,algo,CallPayoff,
                        AmericanCallAdjuster);
std::cout << "Early call: " << earlyCallPrice;
```

You can run this code and check the output. The above algorithms can run in parallel because the asset lattice is *read-only shareable* and each algorithm runs in its own thread. We discuss this case in detail in a later chapter. We have discovered a pattern here in the sense that several algorithms in Layer 2 can use the same data structures as in Layer 1. We shall see more examples of the same patterns of use when we discuss Monte Carlo simulation and the FDM in later chapters.

11.8 MERGING LATTICES

In some cases we may wish to produce a lattice, each of whose nodes are tuples consisting of the asset price and option price. To this end, one possibility is to compute the values in the asset and option lattices and then merge these two lattices to form a new lattice. This choice is useful because we may wish to print both asset and option prices in Excel or on the console, for example. To this end, we have created a function to merge two lattices having the same size. The function creates a lattice from two input lattices:

```
template <class Node, int LatticeType>
      Lattice<std::tuple<Node,Node>, LatticeType> merge
      (const Lattice<Node,LatticeType>& latticeA,const Lattice<Node,
      LatticeType>& latticeB)
{ // Merge elements of two lattices to form a lattice of tuple

   Lattice<std::tuple<Node,Node>, LatticeType> result(latticeA.Depth());

   for (std::size_t n = latticeA.MinIndex(); n <= latticeA.MaxIndex();
   ++n)
   {
      for (std::size_t i = 0; i < latticeA[n].size(); ++i)
      {
         result[n][i] = std::make_tuple(latticeA[n][i],latticeB[n][i]);
      }
   }

   return result;
}
```

As an example, we create two lattices and merge them to form a new lattice:

```
// Option data
OptionData opt;

opt.K = 65.0;
opt.T = 0.25;
opt.r = 0.08;
opt.q = 0.00;
opt.sig = 0.30;

int N = 3;
double dt = opt.T / double(N);

LatticeMechanisms::CRRLatticeAlgorithms algo(opt, dt);

// Create basic asset lattice
Lattice<double,2> asset(N, 0.0);   // init
double rootVal = 60.0;             // Index level
LatticeMechanisms::ForwardInduction<double> (asset, algo, rootVal);
LatticeMechanisms::print(asset, "asset");

double K = opt.K;
auto PutPayoff = [&K] (double S)-> double {return std::max<double>
(K - S, 0.0);};

// American early exercise constraint
auto AmericanPutAdjuster = [&PutPayoff](double& V, double S)->void
{ // e.g. early exercise

   V = std::max<double>(V, PutPayoff(S));
};

Lattice<double,2> earlyPut(N, 0.0);

// Price an early exercise option
double earlyPutPrice = LatticeMechanisms::BackwardInduction<double>
                       (asset, earlyPut, algo, PutPayoff,
                       AmericanPutAdjuster);

// Merge the asset and derivative lattices
Lattice<std::tuple<double, double>, 2> combinedLattice
      = LatticeMechanisms::merge(asset, earlyPut);
LatticeMechanisms::print(combinedLattice);
```

The output from this code is:

```
asset

Branch Number 0: [60, ]
Branch Number 1: [65.6189, 55.1832, ]
```

```
Branch Number 2: [71.764, 60.351, 50.7531, ]
Branch Number 3: [78.4846, 66.0028, 55.506, 46.6787, ]
Early put: 6.03903

asset and price

Branch Number 0: [(60,6.03903)]
Branch Number 1: [(65.6189,2.34205)(55.1832,9.81679)]
Branch Number 2: [(71.764,0)(60.351,4.71543)(50.7531,14.2469)]
Branch Number 3: [(78.4846,0)(66.0028,0)(55.506,9.49395)(46.6787,18.3213)]
Branch Number 4: [(0,0)(0,0)(0,0)(0,0)(0,0)]
```

11.9 SUMMARY AND CONCLUSIONS

In this chapter we have designed a small software framework based on the *Layers* pattern to price options using lattice methods. We concentrated on the binomial method because of its popularity and it is also a good test case to show how to design a software framework with adaptability and maintainability in mind. In particular, we shall see that the framework can be used to create more sophisticated applications in Chapter 12. As a follow-on, we shall see in later chapters that the same software design principles will be applied to the creation of software frameworks for PDE/FDM and Monte Carlo applications.

As a guideline, we summarise the main steps when pricing one-factor options using lattice models:

1. Choose the lattice type (binomial, trinomial) and the number of time steps. Call the appropriate lattice constructor.
2. Choose the algorithm type (CRR, JR) and execute the *forward induction* algorithm.
3. Define *constraint functions* such as early exercise and barriers.
4. Define the payoff function.
5. Execute the *backward induction* algorithm to compute option price.

11.10 EXERCISES AND PROJECTS

1. (Analysing Code)

   The C++ code in Section 11.4 was not written with maintainability in mind. In fact, it was written in a hurry because the manager wanted to see results for a quick prototype, a perfectly acceptable reason. In this exercise we carry out a forensic investigation as it were to determine the level of flexibility of that code and how it can be adapted to suit new requirements. It would be unwise to use this prototype as the basis for future work.

   Answer the following questions:

   a) Determine the parts of the code that are hard-wired (*hint:* for example, the code only caters for call options but we may also like to support put options).

   b) Which C++ functionality can you use to make the code more flexible, for example subtype polymorphism and class hierarchies, function objects and lambda functions?

c) Adapt the code so that it can support a barrier option family (there are eight choices): down-and-out call, down-and-in call, up-and-out call, up-and-in call, down-and-out put, down-and-in put, up-and-out put, up-and-in put. What is the most flexible way to model these without introducing hard-wired conditional logic or code duplication (the infamous copy-and-paste syndrome)?

d) Test the code by comparing the answer obtained for large expiration against the exact solution for a perpetual American call option (see Haug, 2007 for the formula or Section 12.4.3 of the current book).

e) Modify the code so that it can support rebates. Test the accuracy of your results.

f) Consider an American call option with barrier $H = 0$. How many steps are needed in order to achieve the same accuracy as we get using the Bjerksund and Stensland (2002) or Barone–Adesi–Whaley closed-form approximations, for example? (See Haug, 2007.)

2. (Efficiency of ADT Implementations)

We have implemented simple and generalised lattices using nested STL vectors and STL maps, respectively. The performance of these data structures may not be good enough for all applications and in some cases it may be necessary to implement them in a different way.

Answer the following questions:

a) Are there advantages in using Boost uBLAS upper- and lower-triangular matrices instead of nested STL vectors?

b) For generalised lattices, consider using C++ (or Boost) unordered maps instead of STL maps because they have constant complexity instead of logarithmic complexity:

```
// boost or std namespace
unordered_map<TimeAxis, std::vector<Node> > tree;
```

c) Create a new generalised lattice class whose implementation is an unordered map (you can copy and modify the code from the original class). Test the performance of the new class by creating a large lattice whose row set is a Boost date.

d) Why would the use of nested STL lists be an inefficient implementation of a lattice? What is the resulting complexity? For example, how efficient is searching in this new structure?

3. (Software Quality Characteristics in a Lattice Framework)

We discuss some advantages associated with lattice ADTs. Discuss how they deliver the following benefits (McConnell, 2004):

a) *They hide implementation details*. We can change the representation of the ADT from one data type to another data type without affecting clients. It is also possible to store the data in external storage (for example, Excel or a relational database) or memory without affecting clients.

b) *Changes do not affect the whole program*. This benefit is even more pronounced in the current framework because it is based on the *Layers* pattern. Discuss why this statement is true.

c) *It is easier to improve performance*. Discuss how to improve the algorithms in Layer 2 and the ADTs in Layer 1. Are there opportunities to parallelise some of the critical algorithms, for example the code in Section 11.7?

d) *The software is more obviously correct*. For example, compare the general quality of the original algorithms in Section 11.4 to price an American down-and-out call option with similar code based on the current framework.

Can these general design guidelines be applied to other application areas, for example Monte Carlo simulation and PDE models?

4. (Code Rewrite as a Useful Exercise in *Code Versioning*)

Consider the code in Section 11.4 to price an American down-and-out call option. The objective of the current exercise is to manually re-engineer that code in a series of prototypes by using the lattice ADTS and Layer 2 algorithms. Test each prototype to make sure that it produces the same results as its predecessors.

What are the main advantages of the new code when compared with the original code?

This exercise can be seen as a follow-on from Exercise 1.

5. (C++ 11 Combinations)

Based on the discussion in Section 11.4.3 consider the following general scenarios:

- Using function wrappers with embedded tuples.
- Using function wrappers with embedded functions (these are called *functionals*).
- Using tuples containing function wrappers.
- Using tuples containing other tuples (nested *tuples*).

Answer the following questions:

a) Create simple examples of each of the above scenarios to gain some understanding of the syntax.

b) How does this functionality support the ability to create and maintain loosely coupled systems?

c) Consider the problem of modelling real-valued functions of a single variable and their integrals as given by the formula:

$$I(f) = \int_a^b f(x)dx. \tag{11.1}$$

In fact, we are defining a *higher-order function* (or *functional* in this case) that models the integral of the function f on the interval (a, b). Determine how to model equation (11.1) using C++ syntax.

d) We generalise equation (11.1) to the case of a *bilinear mapping*:

$$I(f, g) = \int_a^b f(x)g(x)dx. \tag{11.2}$$

Determine how to model equation (11.2) using C++ syntax (*hint:* use lambda functions).

6. (Pascal's Triangle)

Use the code in Section 11.4.2 to compute Pascal's triangle. Use a function object and a lambda function to implement the generator that computes up and down jumps.

We shall discuss Pascal's triangle in more detail in Chapter 12.

7. (Lattice Models)

In Section 11.5 we priced an option using the CRR model to compute lattice parameters. In this section we discuss a number of alternatives to the CRR method:

- Modified CRR:

$$\begin{aligned} u &= e^{K_N+V_N} \\ d &= e^{K_N-V_N} \\ p_u &= (e^{r\Delta t} - d)/(u-d) \end{aligned} \tag{11.3}$$

where:

$$K_N = \log(K/S)/N,\ V_N = \sigma\sqrt{\Delta t}.$$

- Trigeorgis (TRG):

$$\begin{aligned} \Delta x_u = \Delta x_d &= \sqrt{\sigma^2\Delta t + \nu^2\Delta t^2} \\ p_u &= \frac{1}{2} + \frac{1}{2}\frac{\nu\Delta t}{\Delta x} = \frac{1}{2}\left(1 + \frac{\nu\Delta t}{\Delta x}\right),\ \nu = r - \frac{1}{2}\sigma^2. \end{aligned} \tag{11.4}$$

- Equal probability (EQP):

$$\begin{aligned} a = \nu\Delta t, \quad b &= \frac{1}{2}\sqrt{4\sigma^2\Delta t - 3\nu^2\Delta t^2} \\ \Delta x_u &= \frac{1}{2}a + b \\ \Delta x_d &= \frac{3}{2}a - b \\ p_u = p_d &= \frac{1}{2}. \end{aligned} \tag{11.5}$$

Hint: In all cases $\nu = r - \frac{1}{2}\sigma^2$.
Answer the following questions:

a) Implement the above models using the function object approach taken with the class `CRRLatticeAlgorithms`.

b) Test the models using the data from Section 11.5. What can you say about the relative accuracy of the models for various values of the mesh size?

8. Investigate the accuracy of the CRR model as a function of the number of time steps. Do you get better approximations when you increase the number of time steps? Test the accuracy for both even and odd numbers of time steps.

9. (The Leisen–Reimer Method)

It is well known that convergence of the solution of the binomial method to the true solution is not *monotone* in general and it tends to oscillate around the exact value. Decreasing the step size (or equivalently, increasing the number of time steps NT) does not necessarily give better accuracy when using the CRR method, for example.

We consider a method to price one-factor options (Leisen and Reimer, 1996). The method involves the computation of the up and down parameters u and d in such a way

that the tree centres around the strike price. The method is an improvement on other methods such as CRR because of the less jagged convergence properties. The up and down parameters are defined by (Haug, 2007, pp. 290–292):

$$u = e^{b\Delta t}\frac{h(d_1)}{h(d_2)},\ d = \frac{e^{b\Delta t} - pu}{1-p},\ p = h(d_2) \tag{11.6}$$

where:

$$\begin{aligned} b &= \text{cost-of-carry} \\ \Delta t &= T/NT \\ d_1 &= \frac{\log(S/K) + (b + \sigma^2/2)T}{\sigma\sqrt{T}} \\ d_2 &= d_1 - \sigma\sqrt{T}. \end{aligned}$$

Finally, $h = h(x)$ is a specially chosen function, for example the Preitzer–Pratt inversion formula. Details can be found in Haug (2007).

10. (Code Reduction Using Variadic Parameters)

The algorithms for forward and backward induction in Layer 2 of the software framework use tuples. In particular, binomial lattices use tuples with two components while trinomial lattices use tuples with three components. The current design leads to duplicate code and we would like to create a single algorithm for forward induction and a single algorithm for backward induction. This goal should be achievable by using the fact that the second template parameter in the Level 1 lattice class tells us how many components to use as output (in the case of forward induction) or as input (in the case of backward induction). For example, we could use a fixed-size array `std::array<T,N>` where $N = 2$ for binomial lattices and $N = 3$ for trinomial lattices or the more exotic case of variadic templates in a later version.

We implement the algorithms for forward and backward induction for plain options using `std::array<T,N>` instead of `std::tuple<>`. To motivate what we mean, we give the function prototype and code of the forward induction algorithm:

```
template <typename Node, int LatticeType> void ForwardInduction
    (Lattice<Node, LatticeType>& lattice, const std::function
    <std::array<Node, LatticeType>
    (const Node& node)>& generator, const Node& rootValue)

{
       // Initialise the root value lattice[0][0]
       std::size_t si = lattice.MinIndex();
       lattice[si][0] = rootValue;

       std::array<Node, LatticeType> tuple;

       // Loop from min index to end index, i.e. (min, end]
       for (std::size_t n = lattice.MinIndex() + 1;
```

```
        n <= lattice.MaxIndex(); n++)
        {
           for (std::size_t i=0;i<lattice[n-1].size(); ++i)
           {
              tuple = generator(lattice[n-1][i]);
              // lattice[n][i] = std::get<0>(tuple);
              // lattice[n][i+1] = std::get<1>(tuple);

              for (std::size_t j=0;j<=tuple.size()-1; ++j)
              {
                      lattice[n][i+j] = tuple[j];
              }

           }
        }
}
```

Here we see the emergence of the lattice type as a template parameter.

Answer the following questions:

a) Create the function to perform backward induction based on the above code style.

b) Test the new forward and backward induction algorithms by pricing plain options using both binomial and trinomial lattices.

c) Extend the code to accommodate early-exercise features.

d) We suspect that the use of tuples will be less efficient than the use of arrays. We may need to re-engineer the code. Do you agree?

11. (STL Compatibility and Emulation, Project)

Using the design philosophy underlying the STL algorithms will improve the quality of the code for lattice ADTs and make the code less redundant.

Answer the following questions:

a) Modify the lattice ADTs to make them STL compatible.

b) Test the new functionality by writing a function (similar to `std::transform()`) that applies a real-valued function of one variable to each node of a source lattice to either modify that lattice or initialise a destination lattice. We use iterators in the style of STL by modifying the following code:

```
template <typename Node, int LatticeType>
   void transform(Lattice<Node,LatticeType>& lattice,
   const std::function<Node (const Node&)>& f)
{ // Apply a function to each node of lattice, modifying it
   for (std::size_t n = lattice.MinIndex(); n <= lattice.
   MaxIndex(); ++n)
   {
      for (std::size_t i = 0; i < lattice[n].size(); ++i)
      {
         lattice[n][i] = f(lattice[n][i]);
      }
   }
}
```

c) What are the advantages of having STL-compatible containers?

12. (Sawtooth Phenomenon and Smoothing)

One of the issues with some implementations of the binomial method is that the approximate solution oscillates around the true option price as the number of time steps increases. This *nonmonotone* behaviour is caused by the position of the nodes in relation to the strike price. This is the so-called *sawtooth* pattern. The solution is to *peg* the strike price to be at the centre of the final nodes for even values of the number of mesh points. Improved monotonic convergence can be achieved by using the Leisen–Reimer and modified CRR methods.

Answer the following questions:

a) Create a C++ algorithm to show the sawtooth phenomenon: the output is an array of option prices and the input is a sequence of time steps $N_1 < N_2 < \cdots < N_p$ where$p \geq 2$.

b) Determine how to code the algorithm in step a) using the Level 1 data structures and Level 2 mechanisms that we introduced in this chapter. All p option prices use the same underlying, thus in the interest of efficiency you might like to consider how to optimise this procedure.

c) Do you see possibilities to improve speedup by using multithreading and parallel programming patterns? *Hint:* Separate data structures for assets and options as discussed in Section 11.7 to make parallelisation easier. We discuss these issues in more detail in Chapters 28, 29 and 30.

d) Test the algorithm using CRR, modified CRR and Leisen–Reimer methods. What conclusions can you draw?

13. (Richardson Extrapolation)

The binomial method is first-order accurate and it is possible to expand the binomial price as an asymptotic expansion depending on the number of time steps n as follows:

$$X_n = BS + \frac{A}{n} + 0(n^{-2})$$
$$X_{2n+1} = BS + \frac{A}{2n+1} + 0(n^{-2})$$

where:

X_n = binomial price with n steps
A = constant independent of n
BS = exact Black–Scholes price (independent of n)
$O(n^{-2})$ = big oh notation as discussed in Chapter 17.

We wish to remove the first-order terms in the above asymptotic expansions to get a new second-order approximation of the following form after doing some simple algebra:

$$\left(1 - \frac{n}{2n+1}\right)^{-1} \left(X_{2n+1} - \frac{n}{2n+1} X_n\right) BS + 0(n^{-2}). \tag{11.7}$$

Answer the following questions:

a) Design and implement the code to realise Richardson extrapolation.

b) Test the code on plain options with different values of the number of time steps. Do you notice erratic behaviour in convergence?

c) Is convergence monotone or do you notice sawtooth behaviour as when using the CRR method? A discussion of this topic is given in Müller (2009).

(*Caveat:* the author is not convinced that using Richardson extrapolation is a good approach for this class of problems.)

14. (Project: Using C++11 and C++14 to Improve Readability of Code)

In previous chapters we introduced advanced C++ syntax to improve the readability of code, in particular:

- Scoped enumerators to model specific lattice types such as binomial and trinomial data.
- Variadic templates and variadic tuples to model lattice state changes (jumps).
- `std::array<T, int>` as an alternative to `std::tuple<>`.
- Using *alias template* and thereby phasing out the use of `typedef`, for example handy shortcut names for lattice ADTs, function wrappers and tuples that these ADTs use.

What are the advantages of using this syntax?

15. (Software Framework (New Version); Medium-Sized Project)

The goal of this exercise is to extend the initial software framework that we introduced in this chapter to help model a larger range of problems.

a) In this chapter we did not create a separate entity or class to model the SDE the lattice method is approximating. At this stage the method is only applicable to GBM but we need to determine if it can be adapted to other kinds of SDEs, for example by a change of variables as discussed in Nelson and Ramaswamy (1990). To this end, use the results from Section 6.8 to model an SDE class using the functional programming style. We repeat the code for convenience:

```
// Functions of arity 2 (two input arguments)
template <typename T>
   using FunctionType
      = std::function<T (const T& arg1, const T& arg2)>;

template <typename T>
   using interface
      = std::tuple<FunctionType<T>, FunctionType<T>>;

// Interface to simulate any SDE with drift/diffusion
// Use a tuple to aggregate two functions into what is similar to
// an interface in C# or Java.
template <typename T>
   using ISde = std::tuple<FunctionType<T>, FunctionType<T>>;

// Boost Functional Factory
template <typename T>
   using Factory = boost::factory<T>;

template <typename T = double> class Sde
{
private:
   FunctionType<T> dr_;
   FunctionType<T> diff_;

   T ic; // Initial condition
public:
Sde() = default;
Sde(const Sde<T>& sde2, const T& initialcCondition)
```

```
   : dr_(sde2.dr_),diff_(sde2.diff_),ic(initialcCondition) {}
Sde(const ISde<T>& functions, const T& initialcCondition)
   :dr_(std::get<0>(functions)),diff_(std::get<1>(functions)),
      ic(initialcCondition) {}

T drift(const T& S, const T& t) { return dr_(S, t); }
T diffusion(const T& S, const T& t) { return diff_(S, t); }
;
```

Integrate this class into the framework and test for plain and American puts and calls. Choose an example from Nelson and Ramaswamy (1990) and test it using the modified framework. In which layer will you place the SDE classes and their *adapters*?

b) Consider how to create *factory* and *builder* classes to initialise the application objects in the framework. In which layers will you place these factories?

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 12

Lattice Models Applications to Computational Finance

12.1 INTRODUCTION AND OBJECTIVES

In this chapter we apply the data structures and mechanism functions from Chapter 11 to write code to price European and American options for a range of payoff functions. As in Chapter 11, the focus is on producing accurate results as efficiently as possible. Furthermore, we use features in C++ to allow the software to be adapted to new situations.

A summary of the topics that we discuss in this chapter is:

- Creating and testing the properties of *Pascal's triangle* using the lattice ADT.
- Computing option prices using *Bernoulli paths*.
- Option pricing with dividends.
- Computing option hedge sensitivities (*greeks*).
- Numerical analysis: accuracy and efficiency of the binomial method.
- Two-factor binomial pricing.
- The trinomial method and its extension to explicit finite difference methods.

We use the CRR, Tian three-moment and Chang–Palmer models. For a detailed discussion of the binomial method, see Müller (2009).

This chapter also introduces the finite difference method that we elaborate on in later chapters.

Some of the reasons for discussing the lattice method in Chapters 11 and 12 are:

- Most quant developers are familiar with the method and have probably implemented it in one or more programming languages such as VBA and Matlab, for example. They can relate to the C++ code and associated design patterns presented here.
- Lattice mechanisms can be viewed as explicit finite difference methods and studying them is a good preparation for later chapters where we introduce a range of robust and second-order accurate finite difference schemes.
- Lattice methods support the pricing of plain and early-exercise call and put options. We can use them to produce benchmark values when developing finite difference schemes that may not have an analytical solution.

- We deliver working code using a combination of C++11 functionality and programming styles. In this way we show how the new and existing features in C++ can be used to create maintainable code and later chapters will expand on the initial designs that we have produced in Chapters 11 and 12.

12.2 STRESS TESTING THE LATTICE DATA STRUCTURES

In this chapter we apply the code and classes from Chapter 11 to a range of option pricing problems. For this reason, we need to ensure that the code that we have created is working correctly. To this end, we carry out some extended tests on the well-known *Pascal's triangle* (which is essentially a special kind of binomial lattice). We create the triangle using the code from Chapter 11 and we test the accuracy of the generated data by comparing it with data that can be found in the triangle:

a) The elements of each row are *binomial coefficients*; we check the accuracy by comparing the values with those generated by the Boost C++ *Math Toolkit* library.
b) The sum of elements in row n should be equal to the nth power of 2.
c) Generating *Fibonacci numbers* and checking the output.
d) Creating triangular numbers.

We discuss each of these topics with a view to discovering errors and bugs in the code. Passing the tests in a) to d) should convince us that the code is functioning correctly and then we can move on to option pricing problems using the binomial method. In a later chapter we use the binomial method to price compound and closer options (Mun, 2002).

12.2.1 Creating Pascal's Triangle

Pascal's triangle is a triangular array containing binomial coefficients. The rows of the triangle are numbered starting at row 0 at the apex (the *top row*). The column entries in each row are numbered starting at index 0. The entry at a given row n and column position k is computed by adding the two numbers directly above it to the left and to the right. The values are the same as the *binomial coefficients* at the corresponding row and column positions. The construction is related to binomial coefficients by *Pascal's rule*, given by the following binomial expansion:

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k}y^k \quad \text{where} \quad \binom{n}{k} = \frac{n!}{(n-k)!k!} \quad \text{and} \quad n! = n(n-1)(n-2)\ldots 3.2.1. \tag{12.1}$$

The binomial coefficients are related by:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}, \quad 1 \le k \le n. \tag{12.2}$$

In fact, these coefficients are the entries in the triangle. For example, the first six rows are $\{1\}$, $\{1, 1\}$, $\{1, 2, 1\}$, $\{1, 3, 3, 1\}$, $\{1, 4, 6, 4, 1\}$ and $\{1, 5, 10, 10, 5, 1\}$. We can use the

binomial lattice ADT to compute the up and down parameters using equation (12.2). In other words, we can use this formula to implement the forward induction algorithm for this special lattice.

We now show the code for this problem. First, we construct a Pascal's triangle with five rows:

```
const int TYPEB = 2;                // BinomialLatticeType
std::size_t numRows = 5;            // Number of rows in the triangle
Lattice<std::size_t,TYPEB> pascalTriangle(numRows, 0);
```

We then populate the triangle with values based on formula (12.2) ($j = n, i = k$):

```
// Apex value of triangle (can be modified)
pascalTriangle[0][0] = 1;

// Generate the triangle by implementing the formula (12.2)
for (std::size_t j = pascalTriangle.MinIndex()+1; j <=
pascalTriangle.MaxIndex(); ++j)
{
   // Generate edge values (gives possibility to define
   // edge values other than 1).
   pascalTriangle(j,0) = pascalTriangle(j-1,0);

   pascalTriangle(j,pascalTriangle.MaxIndex(j))
            = pascalTriangle(j-1,pascalTriangle.MaxIndex(j-1));

   // Generate interior values (binomial coefficients)
   for (std::size_t i = 1; i < pascalTriangle[j].size()-1; ++i)
   {
      pascalTriangle(j,i) = pascalTriangle(j-1,i) + pascalTriangle
      (j-1,i-1);
   }
}
```

The entries in the triangle are binomial coefficients that we now discuss; in particular we can check their values against those generated by the Boost C++ *Math Toolkit* library, for example.

12.2.2 Binomial Coefficients

The entry at row n and column k in the generated triangle should have the same value as the binomial coefficient at that position. We now test the accuracy by comparison with the values from the Boost C++ *Math Toolkit* for all values in the triangle. The number of *mismatches* should be zero. The code for this use case is:

```
std::size_t latticeValue; double boostValue;
long counter = 0; // Number of mismatches
// Extract binomial coefficients from the Pascal triangle.
// For each row, print the values and compare with the
// Boost values. We check the values against each other.
```

```
for (std::size_t n = pascalTriangle.MinIndex(); n <=
pascalTriangle.MaxIndex(); ++n)
{
   for (std::size_t k=0;k<pascalTriangle[n].size(); ++k)
   {
      latticeValue = pascalTriangle(n,k);
      boostValue = boost::math::binomial_coefficient<double>(n,k);
      if (static_cast<double>(latticeValue) != boostValue)
      {
         counter++;
      }
   }
}
std::cout << "Number of binomial coefficient mismatches: " <<
counter << "\n";
```

The number of mismatches should be zero if the code is correct. Incidentally, we prefer to use Boost to compute binomial coefficients rather than writing our own code.

12.2.3 Computing the Powers of Two

The sum of the elements in row n is equal to 2 to the nth power. The following code checks this assertion for each row in the triangle. We also check the code against the exact value:

```
std::size_t sum = 0;
counter = 0;
// For each row n, the sum of its elements == 2^n
for (std::size_t n = pascalTriangle.MinIndex(); n <=
pascalTriangle.MaxIndex(); ++n)
{
   sum = 0;
   for (std::size_t k=0;k<pascalTriangle[n].size(); ++k)
   {
      sum += pascalTriangle(n,k);
   }
   if (sum != std::pow(2.0, static_cast<double>(n)))
   {
      counter++;
   }
}
std::cout << "Number of row sum (2^n) mismatches: " << counter
<< "\n";
```

12.2.4 The Fibonacci Sequence

It is possible to locate the famous *Fibonacci numbers* in the triangle. These numbers are defined by the *recurrence relation*:

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}, \quad n \geq 2. \tag{12.3}$$

There is also an explicit formula for the *n*th Fibonacci number:

$$F_n = \frac{1}{\sqrt{5}}[\alpha^n - \beta^n] \quad \text{where} \quad \alpha = \frac{1+\sqrt{5}}{2}, \quad \beta = \frac{1-\sqrt{5}}{2}. \tag{12.4}$$

Using Pascal's triangle we can compute the *n*th Fibonacci number by summing the elements in consecutive rows starting with row *n*. The corresponding code is:

```
// Fibonacci numbers by 'shallow' diagonals
std::size_t N = 4;   // Fib(N)
std::size_t i = 0;   // Running total
double val = 0.0;
for (std::size_t n = N; n >= i; --n)
{
   val += pascalTriangle(n,i);
   i++;
}
std::cout << N << "th Fibonacci number is: " << val;
```

Now define the sequence of numbers by:

$$x_n = F_n / F_{n-1}, \quad n \geq 1.$$

Then, by using equation (12.4) we can show that:

$$\lim_{n\to\infty} x_n = \alpha = \frac{1+\sqrt{5}}{2} \text{ (the Golden Ratio).}$$

Fibonacci numbers have many applications, for example in single-variable optimisation in an interval using the methods *Fibonacci search* and *Golden-Mean* search for unimodal functions (Bronson and Naadimuthu, 1997). A *unimodal function* is one that has only one local minimum or local maximum in an interval. We discuss these minimisation algorithms in Chapter 19.

12.2.5 Triangular Numbers

The final example relating to Pascal's triangle is to locate its triangular numbers. In general, the *n*th *triangular numbers* are formed by traversing down the triangle parallel to the right side of the triangle. For example, the code for the third triangular numbers is:

```
// 'Diagonal' elements of triangle
std::size_t row = 3; // The 3rd triangular numbers start
std::size_t k = 0;
for (std::size_t n=row;n <= pascalTriangle.MaxIndex(); ++n)
{
   std::cout << pascalTriangle(n,k++) << ", ";
}
```

12.2.6 Summary: Errors, Defects and Faults in Software

We have discussed Pascal's triangle and its implementation in C++ for a number of reasons. First, it is fun to do; second, it contains *precomputed data* that can be used as an alternative to computing values at run-time (think about binomial coefficients, for example). Finally, we use it to determine whether our code and logic are correct (Meyer, 1997). By definition, *correctness* is the ability of software products to perform their exact tasks as defined by their specification.

In the current situation we wish the code in Layers 1 and 2 of the framework to be bug free and it is for this reason that we have chosen Pascal's triangle as an initial test of the code. Modelling it as a lattice ADT means that we can test it in different ways and check the output. This is particularly important when testing option pricers because if a bug does occur we then wish to pinpoint the source of the bug. To this end, we formalise the fuzzy term 'software bug'. In particular, we introduce a number of related concepts. An *error* is a wrong decision that was made during the development of a software product. A *defect* is a property of a software system that may cause that system to depart from its intended behaviour. A *fault* is the event of a software system departing from its intended behaviour during one of its executions.

Summarising: ideally, we wish to discover and flag errors as soon as possible, thus ensuring that the software is defect free. In general, faults are due to defects and defects result from errors (Meyer, 1997). In short, we wish to determine if problems are caused by the problem description, in the mathematical specification or in code.

12.3 OPTION PRICING USING BERNOULLI PATHS

In some cases (for example, non-path-dependent European options) it is possible to compute call and put prices as a sum using *Bernoulli paths*. For example, for one-factor problems the number of paths to node (n, k) is equal to $\binom{n}{k}$ and the corresponding probability of reaching node (n, k) is $\binom{n}{k} p^k(1-p)^{n-k}$ where p is the probability that the stock increases by a given fixed amount u. In the case of a general payoff the formulae for call and put options is given by (Haug, 2007):

$$C = e^{-rT} \sum_{j=0}^{n} \binom{n}{j} p^j (1-p)^{n-j} g_1(S_j(T), K) \tag{12.5}$$

and:

$$P = e^{-rT} \sum_{j=0}^{n} \binom{n}{j} p^j (1-p)^{n-j} g_2(S_j(T), K), \tag{12.6}$$

respectively, where:

p = probability that stock increases at the next time step

n = number of time steps in the binomial process

g_1, g_2 = generalised payoff functions, for example $g_1(S, K) = \max(S - K, 0)$, $g_2(S, K) = \max(K - S, 0)$

r = risk-free interest rate

T = expiration

$S_j(T) = S_0 u^j d^{n-j}$, $1 \leq j \leq n$, u = up parameter, d = down parameter

S_0 = stock value where we wish to price the option.

We now give the code that implements these formulae. For readability reasons, we hard-code the input. First, we initialise the option data and CRR parameters:

```
long numRows = 501;

// Option parameters
// p = 5.84769, c = 2.13478 (based on numRows = 501)
double K = 65.0;
double T = 0.25;
double sig = 0.3;
double r = 0.08;
double S = 60.0;

double dt = T/static_cast<double>(numRows);

// Prepare CRR parameters, manually
double u = std::exp(sig*std::sqrt(dt)); double d= 1.0/u;
double discounting = std::exp(- r*T);
double p = (std::exp(r*dt) - d)/(u-d);
```

Next, the code that implements the algorithms in equations (12.5) and (12.6) is given by:

```
#include <boost/math/special_functions.hpp>
using namespace boost::math;

// ...

   double call = 0.0;       // Call price
   double put = 0.0;        // Put price
   double tmp;
   for (long i = 0; i <= numRows; ++i)
   { // Code not optimised

      tmp = binomial_coefficient<double>(numRows,i)
         *std::pow(p,i)*std::pow(1.0-p,numRows-i);
      call += tmp*std::max(S*std::pow(u,i) *std::pow(d,numRows-i) - K,
      0.0);
      put += tmp*std::max(K-S*std::pow(u,i)*std::pow(d,numRows-i), 0.0);
   }

   call *= discounting;
   put *= discounting;

   std::cout << "Call price, put price: " << call << ", " << put <<
   std::endl;
```

This formula is easy to program and performance improvements are possible. See also Section 12.12.

12.4 BINOMIAL MODEL FOR ASSETS WITH DIVIDENDS

We now give a short overview of option pricing in the presence of dividends. A *dividend* is a lump-sum payment that the owner of stock receives each quarter or half-year. The amount paid depends on the profitability of the company in a given year. When a dividend is paid we say that the stock goes *ex-dividend* and then the stock value drops by the dividend amount. In general, we require the dividend date to correspond to one of the nodes in the binomial tree. There are different kinds of dividends:

- Continuous dividend yield.
- Discrete dividend.
- Uncertain dividend (Wilmott, 2006): in this case we do not know the exact value of the dividend. All we can say is that the dividend value D lies in an interval $D^- \leq D \leq D^+$ where D^- and D^+ are known. This problem can be solved using the PDE approach. See also Lewis (2016).

We discuss the first two cases in this section. An analysis of uncertain dividends is outside the scope of this book.

12.4.1 Continuous Dividend Yield

The SDE describing a dividend-paying stock is given by:

$$dS_t = (r - D)S_t dt + \sigma S_t dW_t \, (S_t \equiv S(t),\ W_t \equiv W(t)) \tag{12.7}$$

where:

r = (constant) interest rate
D = constant dividend
σ = constant volatility
$\mathrm{d}W_t$ = increments of the Wiener process.

In other words, the asset pays a continuous dividend yield at a rate of D per unit time. This model is applicable to various kinds of options, for example:

- *Stock indices* (D is the dividend yield on the index).
- *Currencies* (D is the foreign exchange rate).
- *Futures contracts* (D is the risk-free rate ensuring that the contract has zero risk-neutral drift).
- *Commodities* (D is interpreted as a convenience yield on the commodity).

The code to price these kinds of options is similar to that given in Chapter 11. We take a simple example of a put option ($p = 6.066$) on an asset paying a constant dividend of 2%:

```
// I. Asset paying continuous dividend yield

// Option data (Haug 2007 page 4)
OptionData opt;

opt.K = 65.0;
opt.T = 0.25;
opt.r = 0.08;
opt.q = 0.02;       // dividend
opt.sig = 0.3;

// Payoff as a lambda function
double K = opt.K;
auto Payoff = [&K] (double S)-> double {return std::max<double>(K - S,
0.0);};
//auto Payoff = [&K] (double S)
// -> double {return std::max<double>(S - K, 0.0);};

// Time steps
int N = 400;
double dt = opt.T / static_cast<double>(N);

// Function that implements forward/backward induction
LatticeMechanisms::CRRLatticeAlgorithms algorithm(opt, dt);

// Create basic structure for plain options
Lattice<double,2> lattice(N, 0.0); // init
double rootVal = 60.0;
LatticeMechanisms::ForwardInduction<double>(lattice, algorithm, rootVal);

// Price a plain option
double res = LatticeMechanisms::BackwardInduction<double>(lattice,
algorithm, Payoff);
std::cout << "Plain Option price, BM classic #1: ";
```

This code has already been discussed in detail in Chapter 11. No change to the framework code is needed because dividends are modelled as input data already.

12.4.2 Binomial Method with a Known Discrete Proportional Dividend

We now take the case of a dividend that is paid at one known time in the future. In particular, we assume that the dividend date corresponds to one of the mesh points (dates) in the binomial tree. Across a dividend date the stock falls by a certain amount, for example:

- Independently of the stock price (a cash amount).
- Depending on the stock price (the dividend is proportional to the value of the stock).

At the *ex-dividend date* the value of the stock drops in value at time n to:

$$S(1-D)u^j d^{n-j}, \quad 1 \le j \le n. \tag{12.8}$$

When using dividends with the binomial method we note that the nodes prior to the ex-dividend date remain unchanged while the nodes after the ex-dividend date are modified based on formula (12.8). We need to give the dividend value and the ex-dividend date when we design the C++ code.

In the case of a simple lattice ADT the ex-dividend date is a number while in the case of a generalised lattice ADT it can be a date object, for example.

From a C++ perspective we note that the current framework can handle discrete proportional dividends but we must modify the lattice to reflect formula (12.8) after the basic forward induction algorithm has been executed. A typical example is:

```
// Create basic structure for plain options
Lattice<double,2> lattice(N, 0.0); // init
double rootVal =100.0;
LatticeMechanisms::ForwardInduction<double>(lattice, algorithm, rootVal);

// Modify the basic lattice to account for known
// discrete proportional dividend
double div = 0.03;
double factor = 1.0 - div;
int n = 2; // Ex-dividend date

// Modify values to multiply by factor in [n,end]
for (std::size_t j = n; j <= lattice.MaxIndex(); ++j)
{
   for (std::size_t i = 0;i<lattice[j].size(); ++i)
   {
      lattice[j][i] *= factor;
   }
}
```

See Exercise 8 where we discuss possible performance improvements.

12.4.3 Perpetual American Options

In general, there is no closed-form solution for the price of dividend-paying American options. There is one exception (Haug, 2007) and this is the case of a *perpetual option*, that is one with an infinite expiration. For a call option the price is:

$$c = \frac{K}{y_1 - 1}\left(\frac{y_1 - 1}{y_1}\frac{S}{K}\right)^{y_1} \quad \text{where} \quad y_1 = \frac{1}{2} - \frac{b}{\sigma^2} + \sqrt{\left(\frac{b}{\sigma^2} - \frac{1}{2}\right)^2 + \frac{2r}{\sigma^2}} \tag{12.9}$$

while for a put option the price is:

$$p = \frac{K}{1 - y_2}\left(\frac{y_2 - 1}{y_2}\frac{S}{K}\right)^{y_2} \quad \text{where} \quad y_2 = \frac{1}{2} - \frac{b}{\sigma^2} - \sqrt{\left(\frac{b}{\sigma^2} - \frac{1}{2}\right)^2 + \frac{2r}{\sigma^2}}. \tag{12.10}$$

The C++ code for these formulae is given by:

```
double PerpetualCall (double K, double S, double sig,
                      double r, double b)
{ // Dividend q = r - b
  // b is cost of carry (b = r-q for a dividend-paying stock)

   double sig2 = sig*sig;
   double fac = b/sig2 - 0.5; fac *= fac;
   double y1 = 0.5 - b/sig2 + std::sqrt(fac + 2.0*r/sig2);

   double fac2 = ((y1 - 1.0)*S) / (y1 * K);
   double c = K * std::pow(fac2, y1) / (y1 - 1.0);

   return c;
}

double PerpetualPut (double K, double S, double sig,
                     double r, double b)
{

   double sig2 = sig*sig;
   double fac = b/sig2 - 0.5; fac *= fac;
   double y2 = 0.5 - b/sig2 - std::sqrt(fac + 2.0*r/sig2);

   double fac2 = ((y2 - 1.0)*S) / (y2 * K);
   double p = K * std::pow(fac2, y2) / (1.0 - y2);

   return p;
}
```

In general, the above code is useful when we wish to test the accuracy of various numerical methods applied to American option pricing problems with large expiration.

12.5 COMPUTING OPTION SENSITIVITIES

It is possible to compute approximations to option sensitivities using the binomial method. In particular, we wish to approximate the *greeks*:

$$\begin{aligned}
&\text{Delta } \Delta = \frac{\partial V}{\partial S}.\\
&\text{Gamma } \Gamma = \frac{\partial \Delta}{\partial S} = \frac{\partial^2 V}{\partial S^2}.\\
&\text{Theta } \Theta = -\frac{\partial V}{\partial t}.\\
&\text{Rho } \rho = \frac{\partial V}{\partial r}.\\
&\text{Strike } = \frac{\partial V}{\partial K}.\\
&\text{Vega } = \frac{\partial V}{\partial \sigma}.
\end{aligned} \tag{12.11}$$

We can approximate delta and gamma on a single lattice by the following approximations:

$$\text{Delta} \sim \frac{C_1^1 - C_0^1}{S_1^1 - S_0^1} \tag{12.12}$$

$$\text{Gamma} \sim \frac{\dfrac{C_2^2 - C_1^2}{S_2^2 - S_1^2} - \dfrac{C_1^2 - C_0^2}{S_1^2 - S_0^2}}{\dfrac{1}{2}\left(S_2^2 - S_0^2\right)} \tag{12.13}$$

where, in general, the quantity f_j^n signifies the value of f at row n and column j.

Computing the other greeks such as:

$$\begin{aligned}
\text{Vega} &= \frac{\partial C}{\partial \sigma} \cong \frac{C(\sigma + \Delta\sigma) - C(\sigma - \Delta\sigma)}{2\Delta\sigma} \\
\text{Theta} &= \frac{\partial C}{\partial t} \cong \frac{C(t + \Delta t) - C(t)}{\Delta t} \\
\text{Rho} &= \frac{\partial C}{\partial r} \cong \frac{C(r + \Delta r) - C(r - \Delta r)}{2\Delta r}
\end{aligned} \tag{12.14}$$

entails running the binomial method twice (possibly in parallel).

To conclude this section we show the 'get it working' code that implements the approximation of delta and gamma. We create two lattices, one for the underlying (stock S) and the other for the option price. The main steps are:

1. Define the option data.
2. Create the CRR-based underlying lattice or a lattice based on another algorithm.
3. Define the payoff and (if applicable) early-exercise constraint using lambda functions.
4. Create the lattice of option prices.
5. Compute delta and gamma based on the values in the lattice of option prices.

The following code is based on these steps:

```
// 1. Option data
const int TYPEB = 2;
// Option pricer (Clewlow and Strickland 1998 page 25)
OptionData opt;

opt.K = 100.0;
opt.T = 1.0;
opt.r = 0.06;
opt.q = 0.0;
opt.sig = 0.2;

int N = 501;
double dt = opt.T / static_cast<double>(N);

// 2. Create structure containing underlying values
Lattice<double,TYPEB> lattice(N, 0.0);
```

```
double S = 100.0;
CRRLatticeAlgorithms pricer(opt, dt);
ForwardInduction<double>(lattice, pricer, S);
//LatticeMechanisms::print(lattice);

// 3. Payoff and early exercise constraint
double K = opt.K;
auto Payoff = [K] (double S)-> double
       {return std::max<double>(K - S, 0.0);};
//auto Payoff = [K] (double S)-> double
//     {return std::max<double>(S - K, 0.0);};

// The early exercise constraint
auto AmericanAdjuster=[&Payoff] (double& V,double S)->void
{ // e.g. early exercise

   V = std::max<double>(V, Payoff(S));
};

// 4. Create the 'secondary' lattice of option prices
Lattice<double,TYPEB> latticePrice(lattice);
double V = BackwardInduction<double> (lattice, latticePrice,
                                      pricer,Payoff,AmericanAdjuster);
//LatticeMechanisms::print(latticePrice);
std::cout << "Option price: " << V << std::endl;

// 5a. Delta
double delta = (latticePrice(1,1) - latticePrice(1,0)) / (lattice(1,1) -
lattice(1,0));
std::cout << "Option delta: " << delta << std::endl;

// 5b. Gamma
double deltaU= (latticePrice(2,2) - latticePrice(2,1)) / (lattice(2,2) -
lattice(2,1));
double deltaL= (latticePrice(2,1) - latticePrice(2,0)) / (lattice(2,1) -
lattice(2,0));
double gamma = (deltaU - deltaL)                 /(0.5*(lattice(2,2)-
                                                 lattice(2,0)));
std::cout << "Option gamma: " << gamma << std::endl;
```

See Exercises 5 and 6 where we discuss the topics in this section in more detail. In later chapters we show how to compute option sensitivities using the finite difference method.

12.6 (QUICK) NUMERICAL ANALYSIS OF THE BINOMIAL METHOD

We have discussed the software framework for lattice models and we have given a number of examples to test the accuracy of the software. We also wish to analyse the mathematical and numerical properties of the binomial method. Much effort and many articles have been written on the relative merits of variants of the binomial method as applied to option

pricing problems. A complete discussion is outside the scope of this book. Instead, we draw attention to a number of issues that we should be aware of when using the method. We can distinguish between two types of error; first, *distribution error* that arises from the binomial approximation to the lognormal distribution and second, *nonlinearity error* that arises from not having the lattice nodes aligned correctly with the option features, for example when we are confronted with discontinuities in the payoff or in its derivatives of the strike price.

12.6.1 Non-monotonic (Sawtooth) Convergence

Intuitively, we might expect that increasing the number of time steps in the binomial method will improve its accuracy. Unfortunately, the error between the exact and approximate solutions can deteriorate when we increase the number of time steps. In order to understand why this erratic behaviour occurs, we can refer to the literature. For example, Walsh (2003) considers payoff functions that are piecewise C^2 and furthermore the payoff function and its derivatives are polynomially bounded. In the case of an even number of periods n the error estimate in the case of a call option is:

$$V^C_{BT(n)}(K) = V^C_{BS}(K) + \frac{S_0^{\frac{-d_1^2}{2}}}{24\sigma\sqrt{2\pi T}} \frac{A - 12\sigma^2 T\left(\Delta^2_{n-1}\right)}{n} + 0\left(\frac{1}{n\sqrt{n}}\right) \tag{12.15}$$

where:

$$\begin{aligned} K &= \text{strike price} \\ V^C_{BS} &= \text{Black–Scholes price} \\ V^C_{BT(n)}(K) &= \text{binomial price} \end{aligned}$$

$$d_1 = \frac{\log(S_0/K) + (r - q + \sigma^2/2)T}{\sigma\sqrt{T}}$$

$$u = e^{\sigma\sqrt{\Delta t} + r\Delta t}, \quad d = e^{-\sigma\sqrt{\Delta t} + r\Delta t}$$

$$A = \text{constant}, \quad S_0 = \text{initial asset price}$$

$$\Delta_n = 1 - 2 \text{ frac}\left[\frac{\log(S_0/K) + n\log d}{\log u - \log d}\right].$$

Convergence is not (yet) smooth because the approximate price can oscillate around the Black–Scholes price. Hence, we get the famous *sawtooth effect* and standard extrapolation methods to improve accuracy will probably not work properly. We note that the rate of convergence in equation (12.15) can be improved to $O(n^{-1})$ if all discontinuities of the payoff discontinuities are situated at lattice points. A discussion is outside the scope of this book. See Müller (2009) for details.

12.6.2 'Negative' Probabilities and Convection Dominance

One of the dangers when using the binomial method is that the probability term can become negative unless we take many time steps. For example, in the CRR model the up and down probabilities are given by:

$$\begin{aligned} &p_u = (a-d)/(u-d). \\ &a = e^{r\Delta t}, \quad d = e^{-\sigma\sqrt{\Delta t}}, \quad u = e^{\sigma\sqrt{\Delta t}}. \\ &p_d = 1 - p_u = \frac{u-a}{u-d}. \end{aligned} \tag{12.16}$$

In particular, in order to ensure that the probabilities remain positive we must satisfy the following inequality (we see that $u > d$):

$$u > a \Rightarrow \sigma\sqrt{\Delta t} > r\Delta t \Rightarrow \sigma > r\sqrt{\Delta t}. \tag{12.17}$$

In other words, for small volatility and large drift (or both), the CRR will produce negative values for the probabilities (we avoid the term '*negative probability*' as it is ill-defined in the author's opinion). This phenomenon is related to the well-known *convection-dominance* problem that occurs in the numerical solution of convection–diffusion–reaction equations (for example, the Black–Scholes PDE). In particular, the proposed *exponentially fitted* finite difference schemes that are stable and convergent independently of the (relative) sizes of the volatility and drift terms (Duffy, 1980, 2006).

Some numerical experiments show that the CRR model gives terrible results for convection-dominated problems while the Tian model is more accurate (Tian, 1999).

12.6.3 Which Norm to Use when Measuring Error

An important issue is to decide how to measure the accuracy of the binomial method. In other words, we wish to determine how close the approximate solution is to the exact solution in some *norm*. In general, for a binomial method with n steps the convergence rate fluctuates between $\left(\frac{1}{\sqrt{n}}\right)$ and $\left(\frac{1}{n}\right)$. Furthermore, the lack of smoothness of the payoff function will have a negative impact on the convergence rate. Some error measures are:

$$\begin{aligned} &\text{RMS (root-mean-squared) } \sqrt{\frac{\sum_{j=1}^{n}(0_j - A_j)^2}{n}} \\ &\text{Max norm } \max_{1\le j\le n} \left|0_j - A_j\right| \\ &\text{Relative error } \max_{1\le j\le n} \left|(0_j - A_j)/A_j\right| \end{aligned} \tag{12.18}$$

where:

$$\begin{aligned} A_j &= \text{exact value at node } j \\ 0_j &= \text{approximate value at node } j. \end{aligned}$$

Related to accuracy is the concept of *numerical efficiency.* This is a measure of the minimum number of steps required to reach a given level of accuracy. An example is that in order to get an approximation error of five cents we need 40 steps. We can apply this measure to a range of models; for example, is the CRR model more numerically efficient than the Tian model for American put options?

We shall discuss the issue of error estimation in more detail in later chapters when we discuss the finite difference method. We see the binomial method as a specific case of an explicit finite difference method.

12.7 RICHARDSON EXTRAPOLATION WITH BINOMIAL LATTICES

In general, the accuracy of the binomial method is first order as a function of the step-size. Assuming first-order smooth convergence we can relate the approximate and exact Black–Scholes values by the following asymptotic formula:

$$X_n = BS + \frac{A}{n} + 0(n^{-2}) \tag{12.19}$$

where:

X_n = binomial solution with n steps
BS = Black–Scholes solution.

In the author's practice, we have seen that Richardson extrapolation does not always produce the predicted accuracy improvements, especially in cases where oscillations are present. These oscillations can be caused by discontinuous payoff functions, for example. We do mention that we have not examined this problem in greater detail. We do not pursue this issue here because we shall achieve uniform second-order accuracy by other means in later chapters.

12.8 TWO-DIMENSIONAL BINOMIAL METHOD

We give a short introduction to the problem of pricing an option having two underlying assets by applying the binomial method.

The SDEs governing the correlated assets (using standard notation) are given by:

$$\begin{aligned} dS_1 &= (r - D_1)S_1 dt + \sigma_1 S_1 dW_1 \\ dS_2 &= (r - D_2)S_2 dt + \sigma_2 S_2 dW_2. \end{aligned} \tag{12.20}$$

The assets S_1 and S_2 are correlated:

$$dW_1 dW_2 = \rho dt, \quad \rho = \text{correlation}. \tag{12.21}$$

As with the one-factor binomial method, we can transform the SDEs to give new SDEs in log space:

$$\begin{cases} dx_j = \nu_j + \sigma_j dW_j, \quad x_j = \log(S_j) \\ \nu_j = r - D_j - \dfrac{1}{2}\sigma_j^2, \quad j = 1, 2. \end{cases} \tag{12.22}$$

The assumption in the binomial method is that the asset price can go up or down; in this case there are four possibilities, as depicted in Figure 12.1. The calculation of the values of the four probabilities is well known and we refer the reader to Clewlow and Strickland (1998). We use the notation:

$$C_{ij}^n \text{ or } P_{ij}^n$$

to denote the price of a call or put option, at time level n and at mesh point (i, j). Then the backward induction process is:

$$C_{ij}^n = p_{dd} C_{i-1,j-1}^n + p_{ud} C_{i+1,j-1}^n + p_{du} C_{i-1,j+1}^n + p_{uu} C_{i+1,j+1}^n. \tag{12.23}$$

We need to define the payoff function; for example, a *call spread option* for two underlying's S_1 and S_2:

$$\max(S_1 - S_2 - K, 0). \tag{12.24}$$

Since the binomial method is a special case of an explicit finite difference scheme we cannot prescribe the mesh sizes at will and we thus have the following constraints:

$$\Delta x_1 = \sigma_1 \sqrt{\Delta t}\ \Delta x_2 = \sigma_2 \sqrt{\Delta t} \tag{12.25}$$

where

$$x_j = \log(S_j), \quad j = 1, 2.$$

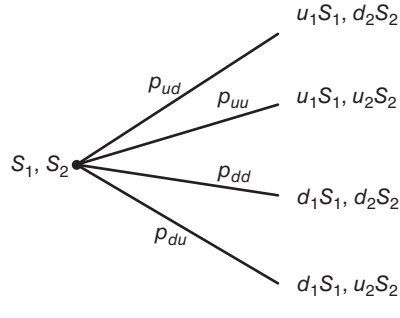


FIGURE 12.1 Two-factor binomial process

12.9 TRINOMIAL MODEL OF THE ASSET PRICE

In Chapter 11 we introduced the binomial method and we used it to price European and American options. In general, the method is not as robust as other numerical methods. We now introduce the *trinomial method* that generalises the binomial method. The trinomial method is also easier to work with because the corresponding grid is more regular and flexible than the corresponding binomial grid. In general, we can achieve the same accuracy using the trinomial method compared to the binomial method but needing fewer mesh points.

We now discuss option pricing based on the well-known methods and pseudocode presented in Clewlow and Strickland (1998) in which the authors transform the SDE for risk-neutral geometric Brownian motion:

$$dS = (r - D)Sdt + \sigma SdW \tag{12.26}$$

where:

r = risk-free interest rate
D = continuous dividend yield
S = asset price
σ = volatility
dW = Wiener process
dt = small interval of time (mesh size).

Equation (12.26) is the starting point for more advanced models. Our interest in this chapter is in showing how to use the trinomial method to calculate the price of an option. It is convenient to use a logarithmic variable $x = \log(S)$ in which case, by using Ito's lemma the SDE in equation (12.26) becomes:

$$dx = \nu dt + \sigma dW \quad \text{where} \quad \nu = r - \frac{1}{2}\sigma^2. \tag{12.27}$$

Then, the stochastic variable x can move up, down or retain the same value in a small interval of time Δt. Each movement has an associated probability having the values:

$$\begin{aligned} p_u &= \frac{1}{2}(\alpha + \beta) \\ p_m &= 1 - \alpha \\ p_d &= \frac{1}{2}(\alpha - \beta) \end{aligned} \tag{12.28}$$

where:

$$\alpha = \frac{\sigma^2 \Delta t + \nu^2 \Delta t^2}{\Delta x^2}, \ \beta = \frac{\nu \Delta t}{\Delta x}.$$

The probabilities add up to 1:

$$p_u + p_m + p_d = 1.$$

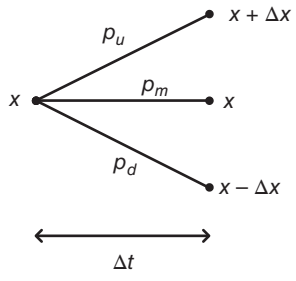


FIGURE 12.2 Trinomial tree model

We extend the trinomial process to a trinomial tree in much the same way as in Chapter 11 for the binomial method. In particular, we define forward and backward induction processes:

- *Forward induction*: we create a trinomial tree structure in the time interval $[0, T]$ where T is the expiration. Since the method is a special case of an explicit finite difference scheme, the step sizes in the x and t directions are related by the expression:

$$\Delta x \geq \sigma\sqrt{3\Delta t}. \tag{12.29}$$

  We then build the tree using this expression and the information in Figure 12.2.

- *Backward induction*: at $t = T$ we define the payoff function. For example, in the case of a call option we have:

$$C_j^N = \max\left(S_j^N - K, 0\right) \tag{12.30}$$

  where K is the strike price.

  We now compute option values as discounted expectations in a risk-neutral world:

$$C_j^n = e^{-r\Delta t}\left(p_u C_{j+1}^{n+1} + p_m C_j^{n+1} + p_d C_{j-1}^{n+1}\right). \tag{12.31}$$

One possible UML design of the trinomial method is shown in Figure 12.3 (this is discussed in Duffy, 2006A).

12.10 STABILITY AND CONVERGENCE OF THE TRINOMIAL METHOD

The trinomial method is commonly quoted as being an example of an *explicit finite difference scheme*. This means that the space step Δx cannot be chosen independently of the time step Δt (and vice versa). In general, we need to ensure that inequality (12.29) is valid to ensure that the probabilities in equation (12.28) remain positive. In particular, the critical case is when the volatility is small compared to the drift term and in this case the probability p_d can become negative. This case is called *convection dominance* and special methods have been devised

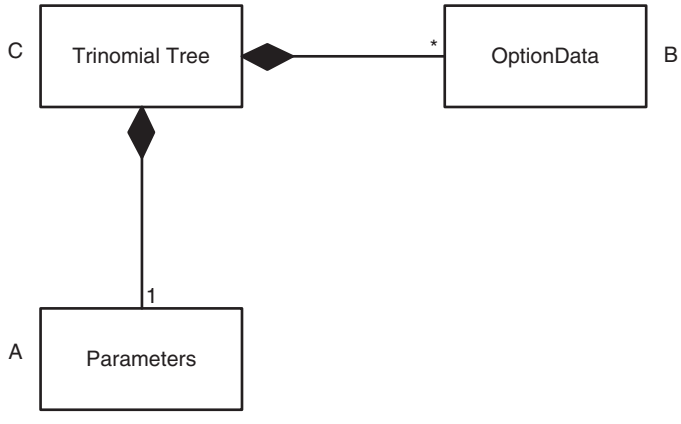


FIGURE 12.3 Trinomial tree model

to ensure that the solution does not oscillate, for example by using *upwinding* or *exponential fitting* (see Duffy, 1980, 2006).

It is this unpredictability of the stability properties of the trinomial method that leads us to search for more robust and accurate numerical methods to approximate the solution of the Black–Scholes partial differential equation. To this end, we introduce the simplest example of a *conditionally stable*, first-order explicit finite difference scheme and then we move to second-order *unconditionally stable* explicit schemes in later chapters.

12.11 EXPLICIT FINITE DIFFERENCE METHOD

The one-factor Black–Scholes equation is similar to the *diffusion* equation but it includes extra *convection* (first-order derivative term) and *reaction* (zero-order derivative term) terms. We recall the Black–Scholes PDE:

$$-\frac{\partial V}{\partial t}+\frac{1}{2}\sigma^2 S^2\frac{\partial^2 V}{\partial S^2}+rS\frac{\partial V}{\partial S}-rV=0. \tag{12.32}$$

The explicit scheme on a uniform mesh is:

$$-\frac{V_j^{n+1}-V_j^n}{\Delta t}+\frac{1}{2}\sigma^2\, S_j^2\left(\frac{V_{j+1}^n-2V_j^n+V_{j-1}^n}{h^2}\right)+rS_j\left(\frac{V_{j+1}^n-V_{j-1}^n}{2h}\right)-rV_j^n=0 \tag{12.33}$$

which we can write in the equivalent form:

$$V_j^{n+1}=\alpha_j V_{j-1}^n+\beta_j V_j^n+\gamma_j V_{j+1}^n.$$

The coefficients α_j, β_j and γ_j are given by:

$$\begin{aligned}
\alpha_j &= \frac{\Delta t\sigma^2 S_j^2}{2h^2} - \frac{r\Delta t S_j}{2h} = \frac{\Delta t\sigma^2 (jh)^2}{2h^2} - \frac{r\Delta t(jh)}{2h} = \Delta t\sigma^2 j^2/2 - \frac{r\Delta tj}{2} (S_j = jh) \\
\beta_j &= 1 - \frac{2\Delta t\sigma^2 (jh)^2}{2h^2} - r\Delta t = 1 - \sigma^2 j^2 \Delta t - r\Delta t \\
\gamma_j &= \Delta t^2 \sigma^2 j^2/2 + \frac{r\Delta tj}{2}.
\end{aligned} \tag{12.34}$$

It is possible to get incorrect answers (negative values, for example) if the time step Δt is inappropriately chosen. To motivate what we mean, we note that the scheme (12.33) is in fact equivalent to the trinomial method. Furthermore, the coefficients in (12.34) correspond to *positive* probabilities in the trinomial method:

- α_j: probability that the stock price decreases.
- β_j: probability that the stock price remains the same.
- γ_j: probability that the stock price increases for all $j = 0, \ldots, NT$.

These values should be positive if we are to retain the physical or financial meanings in the numerical results. We say that the explicit scheme is *conditionally stable*. This means that there are restrictions on the mesh sizes Δt and h and there is a constraint that must be satisfied at all times if we wish to have a stable approximation.

We interpreted jumps in the trinomial method as probabilities. In general, the sum of the probabilities is equal to one and each probability should be non-negative. A probability can become negative depending on the relative sizes of the drift and volatility terms. We can analyse the stability by more methods such as *von Neumann stability analysis* and the *maximum principle* (see Duffy, 2006).

We now examine the stability of the *explicit finite difference method* (also known as *Forward in Time Centred in Space* (FTCS)) applied to the model *convection–diffusion equation* in *non-conservative form*:

$$\frac{\partial u}{\partial t} = -\mu \frac{\partial u}{\partial x} + \sigma \frac{\partial^2 u}{\partial x^2} \tag{12.35}$$

where:

$$\begin{aligned}
u &= \text{vorticity (or other advected (convected) quantity)} \\
\sigma &= \text{diffusion} = \tfrac{1}{\text{Re}} \text{ where Re is the Reynolds number} \\
\mu &= \text{linearised advection speed.}
\end{aligned}$$

The *FTCS* (explicit Euler) scheme now becomes:

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = -\mu \left(\frac{u_{j+1}^n - u_{j-1}^n}{2h} \right) + \sigma \left(\frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{h^2} \right) \tag{12.36}$$

or:

$$u_j^{n+1} = (\beta + \alpha)\, u_{j-1}^n + (1 - 2\beta)\, u_j^n + (\beta - \alpha)\, u_{j+1}^n$$

where:

$$\alpha = \frac{\mu \Delta t}{2h}, \quad \beta = \frac{\sigma \Delta t}{h^2}.$$

We see that this scheme is explicit in time and centred in space. We can examine this scheme from the viewpoint of the *maximum principle* that determines the conditions under which the solution at time level $n + 1$ is non-negative given that the solution is non-negative at time level n. We can *brainstorm* by considering the conditions under which all coefficients on the right-hand side of equation (12.36) are non-negative and what happens when they become negative. To be precise, there are two forms of instability associated with the above FTCS (Roache, 1998) scheme:

- *Dynamic instability*: the scheme experiences oscillatory errors due to large step sizes in time. The condition for *no overshoot* is:

$$1 - 2\beta \geq 0 \Rightarrow \beta \leq \frac{1}{2}. \tag{12.37}$$

  This inequality places a dependency relationship between the mesh sizes in the space and time dimensions.

  We resolve this problem by choosing not to use explicit schemes.

- *Static instability*: this form of instability is caused by the convection (advection) term in the PDE (12.36), especially when it is large compared to the volatility term:

$$\beta - \alpha \geq 0 \Rightarrow \frac{\mu h}{\sigma} \leq 2. \tag{12.38}$$

This is called the *convection-dominance* phenomenon. We resolve this problem by *exponential fitting* in later chapters. This phenomenon can also occur in time-independent problems.

We conclude this section by discussing an implicit scheme that does not produce dynamic instability. It is called the *Backward in Time Centred in Space* (BTCS) scheme for PDE (12.35) defined by:

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = -\mu \left(\frac{u_{j+1}^{n+1} - u_{j-1}^{n+1}}{2h} \right) + \sigma \left(\frac{u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1}}{h^2} \right). \tag{12.39}$$

We rewrite this scheme in the following form:

$$u_j^{n+1} - u_j^n = -\alpha \left(u_{j+1}^{n+1} - u_{j-1}^{n+1} \right) + \beta \left(u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1} \right) \tag{12.40}$$

or:

$$-(\alpha + \beta)u_{j-1}^{n+1} + (1 + 2\beta)u_j^{n+1} + (\alpha - \beta)u_{j+1}^{n+1} = u_j^n.$$

This scheme still suffers from possible static stability problems and we can resolve this problem by employing *upwinding schemes*, which are one-sided difference schemes for the convection term:

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = -\mu \left(\frac{u_j^{n+1} - u_{j-1}^{n+1}}{h} \right) + \sigma \left(\frac{u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1}}{h^2} \right) \ (\mu > 0) \quad (12.41)$$

or:

$$-(2\alpha + \beta)u_{j-1}^{n+1} + (1 + 2\alpha + 2\beta)u_j^{n+1} - \beta u_{j+1}^{n+1} = u_j^n. \quad (12.42)$$

The disadvantage of upwinding is that it is only first-order accurate and we must take into account that the coefficient of the convection term can be positive or negative and the correct kind of upwinding must be used in each case.

Another solution to static stability problems is to use the *exponential fitting method* as discussed in Duffy (1980, 2006). In particular, we use a combination of von Neumann stability analysis, the maximum principle for PDEs and *M-matrix theory* to provide a general method to prove the stability for a range of finite difference schemes. In particular, it can be used to analyse the stability of schemes (12.40) and (12.41). We discuss these topics in more detail in later chapters.

12.12 SUMMARY AND CONCLUSIONS

In this chapter we continued with the analysis and application of the basic software framework that we created in Chapter 11. The focus in this chapter was on testing the reliability, accuracy and functionality of the code in the framework. First, we tested the lattice ADT against the properties of Pascal's triangle. We then computed one-factor option prices, option sensitivities and dividends. We also carried out an abridged numerical analysis of the binomial method. Finally, we discussed the transition from lattice methods to explicit finite difference methods. This prepares us for later chapters.

12.13 EXERCISES AND PROJECTS

1. (Pascal's Triangle Functionality)

There are many applications of Pascal's triangle. We discuss some of them and the objective is to implement them in C++. Answer the following questions:

a) Construct the *Sierpinski triangle* that we create by filling the Pascal's triangle black if the value is odd and white if the value is even. This tactic entails reducing each element modulo two. The output is a regular pattern of inverted triangles with sizes differing by powers of two.

b) Construct the following analogue to Pascal's triangle. In this case we expand $(x + 2)^n$ instead of $(x + 1)^n$ where n is the row number. We set row 0 to $\{1\}$ and row 1 to $\{2\}$. The triangles are constructed according to the recursive formula:

$$\binom{n}{k} = 2\binom{n-1}{k-1} + \binom{n-1}{k}, \quad 1 \leq k \leq n.$$

Verify that the sum of elements of row n is 3^{n-1}.

c) It is possible to compute the binomial distribution from Pascal's triangle. To this end, let p be the probability of an event occurring and let us assume that $p = 1/2$. Divide the nth row by 2^n and verify that the triangle becomes the binomial distribution.

2. (Performance Testing)

We examine the code in Section 12.2 with a view to improving its run-time performance. Answer the following questions by writing code to compare the performance of candidate solutions:

a) The sum of the elements in row n versus the nth power of 2.

b) Computing the binomial coefficients using Boost C++ *Math Toolkit* versus *lookup* in Pascal's triangle.

c) Computing the Fibonacci sequence by using elements in Pascal's triangle versus the formulae in equations (12.3) and (12.4).

You can use the C++11 *chrono* library or the stopwatch class from Chapter 10 to measure the performance.

3. (Binomial Option Pricing versus Exact Solution)

Use formula (12.5) to compute the call price for the following payoffs:

- Symmetric power call:

$$V(S, T) = \max((S - K)^p, 0). \tag{12.43}$$

- Asymmetric power call:

$$V(S, T) = \max(S^p - K, 0) \tag{12.44}$$

where p is a positive number. Compare the answers with the values produced by the binomial method from Chapter 11.

4. (American Option Pricing)

In this exercise we use the binomial method to price American options. Answer the following questions:

a) Compute the price of an American option for large values of the expiration T, for example T in the range [10, 100] years in steps of 10 years. How many time steps are needed in order to achieve convergence in each case?

b) For which expiration value T does the binomial method give the same value as the formulae (12.9) and (12.10) for a perpetual American option (if at all)?

5. (Computing Option Sensitivities I)

The goal of this exercise is twofold: based on the discussions in Section 12.5 we create a more user-friendly wrapper to compute option delta and gamma. We write code to compute the sensitivities in equation (12.11).

Answer the following questions:

a) Redesign the code in Section 12.5 to compute delta and gamma. We hide many of the details and write the new code as a black box. We start with the end in mind so that the desired output becomes a tuple whose elements are option price, delta and gamma. The input is a struct containing option data, the kind of payoff and the number of steps *NT* in the binomial method.

b) Compare the accuracy of the output by comparing it with the exact Black–Scholes values for price, delta and gamma.

c) Determine how to make your code as reusable as possible, for example creating a repository of payoff functions and placing the new code in Level 3 of the *Layers* patterns.

6. (Computing Option Sensitivities II, Small Project)

Write code to compute the sensitivities in equation (12.14). The requirements are that the code should be reusable, easy to understand, accurate and efficient. A guideline is to carry out a requirements analysis of the problem before embarking on a coding session. An added requirement is that your code should use as much of the existing code in the framework as possible.

7. (Simple Pricing)

Use the code in the framework to price the following kinds of option:

a) Futures contract, price = 4200, volatility = 15%, strike = 3800, $T = 0.5$. The put option premium is fully margined (Haug, 2007 gives $p = 65.6185$).

b) European 6-month call option on an index with index level = 810, strike = 800, risk-free rate = 5%, volatility = 20% and dividend yield = 2% (Hull, 2006 gives $c = 53.39$ on a two-step binomial tree).

c) Nine-month American put option on a futures contract with futures price = 31, strike price = 30, risk-free rate = 5%, volatility = 30% (Hull, 2007 gives $p = 2.84$ on a three-step binomial tree).

8. (Efficiency Improvement)

The code in Section 12.4.2 that computes the price of a discrete dividend-paying option is inefficient in the following sense: first, the (default) forward induction algorithm updates the lattice from the current time to expiration and second, the lattice is again updated to reflect the presence of a discrete dividend at the ex-dividend date. Modify the code in the framework to optimise the code by updating the lattice only once, for example:

- 'Normal' forward induction up to but not including the ex-dividend date.
- Implementing formula (12.8) starting from the ex-dividend date to expiration.

9. (Dividend-Paying Options)

Based on the code in Section 12.4.3 and the Black–Scholes formula, compute the following perpetual option values:

- American call and put with and without dividends.
- European call and put with and without dividends.

Optimise the code in Section 12.4.3 to create a single function that computes both call and put American option prices and that returns their values in a tuple.

10. (Numerical Efficiency)

We compare the relative efficiency of the CRR and Tian (see Tian, 1993) methods applied to the ATM European case $K = S = 60$, $T = 1$, $r = 006$, sig. $= 0.3$ (put $= 5.335399$, $c = 8.82952$).

Answer the following questions:

a) Compare the approximate values for CRR and Tian models for $NT = 100, 200, \ldots, 1000$. Which model is more accurate?

b) For a given tolerance level (for example, 5% precision level), determine how many steps are needed in the CRR and Tian models to achieve a given accuracy.

11. (Two-Dimensional Binomial Method, Small Project)

The goal of this exercise is to implement the algorithm for the two-dimensional binomial method as discussed in Section 12.8. In particular, we wish to write code for the different use cases and we then integrate the code into the software framework that we introduced in Chapter 11. Alternatively, you could design a new software framework for two-dimensional problems.

Answer the following questions:

a) Make an inventory of the entities and modules that the algorithm will need. We note that the core process is to price a range of two-factor European and American options using the two-factor binomial method. Describe the steps (activities) that map the core process's input to its output.

b) Determine how to design and implement two-factor option payoffs and data (for a list, see Haug, 2007):

$$\begin{aligned}
&\text{Spread Call: } \max(0, Q_1 S_1 + Q_2 S_2 - K) \\
&\text{Spread Put: } \max(0, K + Q_2 S_2 - Q_1 S_1) \\
&\text{Call option on maximum: } \max(0, \max(Q_1 S_1, Q_2 S_2) - K) \\
&\text{Put option on maximum: } \max(0, K - \max(Q_1 S_1, Q_2 S_2)) \\
&\text{Exchange option: } \max(0, Q_2 S_2 - Q_1 S_1)
\end{aligned} \tag{12.45}$$

where Q_1, Q_2 are the fixed quantities of the two assets, respectively, and K is the strike price. Furthermore, S_1 and S_2 are the underlying assets.

The design must be so reusable and flexible that it can be used in the current binomial pricer as well as playing the role of the initial condition in PDE/finite difference methods. Furthermore, the code will be used in Monte Carlo option pricers. A requirement is that the interface to the payoff code should be standardised (for example, using function objects or lambda functions) and it should be extendible without having to modify client code. You can choose between polymorphic class hierarchies, the CRTP pattern, function objects and free functions to implement your solutions.

c) Design the ADT that models the parameters of the two-factor SDE that describes the behaviour of the underlying stocks.

d) Decide how to implement the nodes of the underlying lattice ADT (recall that we used up and down parameters in Chapter 11). For example, we could use the explicit formulae:

$$\begin{aligned}
&S^n_{(1)} j, k = S_1 \exp(j \Delta x_1) \quad \text{(first stock)} \\
&S^n_{(2)} j, k = S_2 \exp(k \Delta x_2) \quad \text{(second stock)}
\end{aligned} \tag{12.46}$$

where Δx_1 and Δx_2 are defined in equation (12.25). S_1 and S_2 are the stock prices at $t = T$ and $S^n_{(i)}j, k$ are the stock prices at node (j, k), $i = 1, 2$. Alternatively, you can use the formulae for the up and down parameters as discussed in Haug (2007). Specify the interface of the two-dimensional lattice ADT.

e) Create the mechanism functions to price European and American options (note how we created the Level 2 functionality in Chapter 11 in the case of one-factor models).

f) Create the Level 3 functionality to configure and initialise the application. Test your application on a range of two-factor option pricing problems by comparing the solution with the exact solution (or an accurate approximate solution), the Monte Carlo solution and the solution from the PDE/finite difference approach.

12. (Trinomial Method, Project)

We focused in Chapters 11 and 12 on the binomial method to price one-factor plain and American options. In this exercise we are interested in creating a small software framework to do the same thing using the trinomial method. The requirements are that the resulting software base should be maintainable, reusable and extendible. We hope to achieve these ends by using design patterns and modern language features in C++. To this end, we adopt an approach similar to that used in the manufacturing industry. This is called the *PDCA cycle* (see Imai, 1986) and it consists of the following phases:

- *Plan (P phase)*: we determine what needs to be done, for example creating a software framework to price one-factor options using the trinomial method. This phase contains not only the plans for a new product but also plans for product and process improvements.
- *Do (D phase)*: we produce or make the software product.
- *Check (C phase)*: does the product operate according to the specifications? Is the customer satisfied? In the current context the code should produce accurate results and be efficient.
- *Action (A phase)*: process feedback and complaints from customers. This feedback is incorporated into the next planning phase. In other words, action refers to action for improvement.

A variation of PDCA is called *SDCA* (the '*S*' stands for *Standardisation*) and it refers to the fact that we establish and stabilise a standard before embarking on a new PDCA round. In the current case this could entail applying standardised design patterns, interfaces and language features.

Answer the following questions:

a) Compare the advantages and disadvantages of the PDCA as applied to software development compared to the approach that you now take.

b) Determine which code from Chapters 11 and 12 can be reused in the current context. In particular, examine the applicability of the lattice ADTs and mechanism functions that we used to price options using the binomial method. Can you reuse the existing software or will you need to write new code? How can the language features in C++11 help?

c) Implement the algorithm in Section 12.9. Test the code using the same examples as in Chapters 11 and 12.

d) Adapt the code in Section 12.9 to accommodate other strategies to compute jump probabilities. The first case is when the asset price at each node can go up, stay the same or go down. The jump sizes are:

$$u = e^{\sigma\sqrt{2\Delta t}}, \quad d = e^{-\sigma\sqrt{2\Delta t}}.$$

The corresponding probabilities are (Haug, 2007):

$$p_u = \left(\frac{e^{b\Delta t/2} - e^{-\sigma\sqrt{\Delta t/2}}}{e^{\sigma\sqrt{\Delta t/2}} - e^{-\sigma\sqrt{\Delta t/2}}} \right)^2$$

$$p_d = \left(\frac{e^{\sqrt{\Delta t/2}} - e^{-b\Delta t/2}}{e^{\sigma\sqrt{\Delta t/2}} - e^{-\sigma\sqrt{\Delta t/2}}} \right)^2$$

$$p_m = 1 - p_u - p_d$$

where b is the cost-of-carry. We need to avoid *negative probabilities* (see Haug, 2007, p. 300) which occur for p_m if:

$$\sigma < \sqrt{\frac{b^2 \Delta t}{2}}.$$

This is a CRR-type trinomial tree. We also consider the alternative set of parameters:

$$p_u = \frac{1}{6} + (b - \sigma^2/2)\sqrt{\frac{\Delta t}{12\sigma^2}}$$

$$p_d = \frac{1}{6} - (b - \sigma^2/2)\sqrt{\frac{\Delta t}{12\sigma^2}}$$

$$p_m = 2/3.$$

Answer the following questions:

a) Integrate the above two sets of probability parameters into the framework. You can deploy universal function wrappers that we introduced in Chapter 11 to calculate the up and down jumps as with the binomial method.

b) Test the new schemes using the examples in this chapter.

c) Determine the 'stability spectrum' of the trinomial method by varying the option parameters and step size.

d) Test the code for European and American options.

e) Review the flexibility of the code that you have written and determine how to make your framework more flexible. How will you test the flexibility of the software?

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 13

Numerical Linear Algebra: Tridiagonal Systems and Applications

13.1 INTRODUCTION AND OBJECTIVES

In this chapter we introduce a number of fundamental algorithms related to *numerical linear algebra*. In particular, we discuss efficient algorithms to solve linear systems of equations. Some of the objectives are:

- Designing algorithms and mapping them to C++ code.
- Using the classes in this chapter as a lightweight alternative to open source matrix libraries such as *Eigen*, for example.
- Porting legacy code from Duffy (2004B) to C++11.
- Hands-on experience in understanding, writing and debugging C++ code.

We focus on producing efficient code that we use in applications. We define a seamless mapping between an algorithm and the C++ code that implements it. This makes debugging easier and results in maintainable code, especially if it is clearly documented.

This chapter is useful for readers who wish to learn the essentials of the finite difference method.

13.2 SOLVING TRIDIAGONAL MATRIX SYSTEMS

A *band matrix* $A = (a_{ij})$ is a square matrix of size n and *width* $2K + 1$ such that $a_{ij} = 0$ when $|i - j| > K$, where K is a non-negative integer. All non-zero elements are positioned on the main diagonal and on the first K diagonals directly above and below it. Some special band matrices are:

- A *diagonal matrix* is a band matrix with $K = 0$.
- A *Toeplitz matrix* is a band matrix in which each diagonal consists of a single identical element but different diagonals may contain different elements.
- A *tridiagonal (Jacobi) matrix* is a band matrix of width three ($K = 1$).

Tridiagonal matrices are found in numerical analysis applications, for example when approximating the solution of ordinary and partial differential equations by the finite difference and finite element methods. To this end, we discuss how they arise by first considering the simple *two-point boundary value* problem on the interval (0,1) with Dirichlet boundary conditions:

$$\begin{cases} \dfrac{d^2u}{dx^2} = f(x), \quad 0 < x < 1 \\ u(0) = \varphi, \quad u(1) = \psi. \end{cases} \tag{13.1}$$

The function $f(x)$ and constants φ and ψ are assumed to be known constants.

We approximate the solution u by creating *discrete mesh points* defined by $\{x_j\}, j = 0, \ldots, J$ where J is a positive integer. At each interior mesh point we approximate the second-order derivative in equation (13.1) by a second-order divided difference. The corresponding discrete scheme is:

$$\begin{cases} U_{j+1} - 2U_j + U_{j-1} = h^2 f(x_j), \quad 1 \le j \le J-1 \, (h = 1/J) \\ U_0 = \varphi, U_j = \psi. \end{cases} \tag{13.2}$$

This scheme can be written as a tridiagonal matrix system as follows:

$$\begin{aligned} &AU = F \\ &\text{where } U = (U_1, \ \ldots, \ U_{J-1})^\top, \ F = \left(h^2 f(x_1) - \varphi, \ h^2 f(x_2), \ \ldots, \ h^2 f(x_{J-1}) - \psi\right)^\top. \end{aligned} \tag{13.3}$$

In this case A is a tridiagonal matrix and the symbol $\top$ denotes transpose. We solve this system for the unknown vector U.

We see that there are $J-1$ unknown values (under more general boundary conditions there would be $J+1$ unknown values) because in this current case we are dealing with Dirichlet boundary conditions, which means that values of U are known at $x = 0$ and at $x = 1$.

The next questions are: how do we model tridiagonal matrices and how do we solve the tridiagonal system (13.3)? The answer to the first question is to model the matrix as three vectors, one of length J (the diagonal) and two vectors of length $J-1$ (for the subdiagonal and superdiagonal). Regarding the second question we discuss two solutions.

13.2.1 Double Sweep Method

This method is discussed in Godounov and Riabenki (1977) and it is an algorithm to solve three-point difference schemes with given boundary values. It corresponds to a discretisation of two-point boundary value problems by the finite difference method. The objective is to find a *discrete function* $\{U_j\}$, $j = 0, \ldots, J$ satisfying the following set of equations:

$$\begin{cases} a_j U_{j-1} + b_j U_j + c_j U_{j+1} = f_j, j = 1, \ldots, J-1 \\ U_0 = \varphi, U_J = \psi \\ \{a_j\}, \{b_j\}, \{c_j\}, \{f_j\} \text{ known vectors.} \end{cases} \tag{13.4}$$

The Double Sweep method uses a *recurrence (backward) relation* to compute the solution starting from index J:

$$\begin{cases} U_J = \psi \\ U_j = L_j U_{j+1} + K_j, \quad j = J-1, \ldots, 0 \end{cases} \tag{13.5}$$

where the coefficients L_j, K_j are computed using a *forward recurrence relation*:

$$\begin{cases} L_j = c_j/(b_j + a_j L_{j-1}) \\ K_j = (f_j - a_j K_{j-1})/(a_j + a_j L_{j-1}) \\ j = 1, \ldots, J-1 \end{cases} \tag{13.6}$$

and

$$\begin{cases} L_0 = 0 \\ K_0 = \varphi. \end{cases} \tag{13.7}$$

It is possible to check that these formulae are correct by writing the solution of equation (13.4) in the following form:

$$U_j = (f_j - a_j U_{j-1} - c_j U_{j+1})/b_j, \quad j = 1, \ldots, J-1 \tag{13.8}$$

and by comparing the terms with the coefficients in equation (13.5). We leave this as an exercise.

Moving to code that implements this algorithm, we first see that we need four input arrays, three of which are the arrays comprising the tridiagonal matrix and the fourth is for the inhomogeneous term in equation (13.4) . We also need three work arrays, namely U (the solution), K and L from equations (13.5) and (13.6). The arrays K and L are essential it seems, while we recycle the array for the inhomogeneous term that will hold the values in the solution U. Then the number of work arrays can be reduced to two.

We design a class template for the Double Sweep method. It has two parameters, one for the underlying data type and one for the memory allocator. We use the *template–template parameter* trick to achieve this level of indirection. We thus have a template parameter for the numeric data type and a template parameter for the memory allocator. We include code for this class in a single header file for convenience. We model the tridiagonal system as four one-dimensional arrays. We also define Dirichlet boundary conditions `left` and `right` and two work arrays L and K:

```
// Using template template parameters to model vectors
template <typename T, template <typename T, typename Alloc> class
Container = std::vector, typename Alloc = std::allocator<T>>

class DoubleSweep
{ // The Balayage method from Godounov
```

```
private:

// The vectors of length J and start index 0
Container<T, Alloc> a, b, c, f;

// Dirichlet boundary conditions
T left;     // Left boundary condition
T right;    // Right boundary condition

// Work arrays
Container<T, Alloc> L;
Container<T, Alloc> K;

public:
// Constructors and destructor
DoubleSweep() = delete;
DoubleSweep(const DoubleSweep<T,Container, Alloc>& s2) = delete;

// Create members to initialise input for AU = F, A = (a,b,c)
DoubleSweep(const Container<T, Alloc>& lowerDiagonal, const
            Container<T, Alloc>& diagonal, const Container<T,
            Alloc>& upperDiagonal, const Container<T, Alloc>& F,
            const T& bc_left, const T& bc_right)
{

// Vectors are copied
   a = lowerDiagonal;
   b = diagonal;
   c = upperDiagonal;
   f = F;

   left = bc_left;
   right = bc_right;

   std::size_t N = a.size();

// Work arrays
   L = Container<T, Alloc>(N, 0);
   K = Container<T, Alloc>(N, 0);
}
virtual ~DoubleSweep() = default;

// Operator overloading
DoubleSweep<T,Container, Alloc>& operator = (const DoubleSweep<T,
Container>& i2) = delete;

Container<T, Alloc> solve()
{ // Result; vector in closed range [0, J], a vector of size J+1.

   std::size_t N = a.size();
```

```
    // equation 13.7
    L[0] = 0.0;
    K[0] = left;

    // Equation 13.6
    std::size_t SZ = L.size();
    for (std::size_t j = 1; j < SZ; ++j)
    { // L

       double tmp = b[j] + (a[j] * L[j - 1]);

       L[j] = -c[j] / tmp;
       K[j] = (f[j] - (a[j] * K[j - 1])) / tmp;
    }


    // Equation 13.5. Recycle array f for u
    f[0] = left;
    f[N - 1] = right;
    for (std::size_t j = f.size() - 2; j >= 1; --j)
    { // U

       f[j] = (L[j] * f[j + 1]) + K[j];
    }

    return f;
    }

    Container<T, Alloc> operator () ()
    {
       return solve();
    }
};
```

In general, the input vectors and return (output) vector have the same size and the boundary conditions' values are given as input arguments to the algorithm. We have written the code to mirror the structure of the algorithm as given by equations (13.5) to (13.7).

13.2.2 The Thomas Algorithm

This is a popular algorithm and it is well documented (Keller, 1968; Thomas, 1949, 1995). It is used when approximating the solution of convection–diffusion PDEs using the finite difference method. It is a special case of LU decomposition for tridiagonal matrices.

We implement the Thomas algorithm in two steps. In general, we solve a system of the form $Au = r$ where A is a tridiagonal matrix, r is a given vector and u is the unknown vector that we wish to find. We decompose A into the product of a *lower-triangular matrix* L and an *upper-triangular matrix* U as $A = LU$ leading to a simpler set of equations to solve:

$$\begin{cases} Au = r \Leftrightarrow (LU)u = r \\ Lz = r \\ Uu = z. \end{cases} \tag{13.9}$$

The bidiagonal matrices L and U are shown as follows:

$$L = \begin{pmatrix} \beta_0 & & & & \\ a_1 & & & & \\ & \ddots & \ddots & 0 & \\ & 0 & \ddots & \ddots & \\ & & & a_J & \beta_J \end{pmatrix}$$

$$U = \begin{pmatrix} 1 & \gamma_1 & & & \\ & \ddots & \ddots & 0 & \\ & 0 & \ddots & \gamma_{J-1} & \\ & & & 1 & \end{pmatrix}.$$

Their coefficients can be computed as follows:

$$\begin{cases} \beta_0 = b_0, \gamma_0 = c_0/\beta_0 \\ \beta_j = b_j - a_j\gamma_{j-1}, \quad j = 1, \ldots, J \\ \gamma_j = c_j/\beta_j, \quad j = 1, \ldots, J-1 \end{cases} \tag{13.10}$$

and the algorithm in equation (13.9) has the following component-wise form for the two iterations, respectively:

$$\begin{cases} z_0 = r_0/\beta_0 \\ z_j = (r_j - a_j z_{j-1})/\beta_j, \quad j = 1, \ldots, J \end{cases} \tag{13.11}$$

$$\begin{cases} U_J = z_J \\ U_j = z_j = \gamma_j U_{j+1}, \quad j = J-1, \ldots, 0. \end{cases} \tag{13.12}$$

Examining these algorithms we have used three work arrays in the code for convenience (we may be able to reduce this to two work arrays but we do not consider this optimisation step).

Finally, we note that for the Thomas algorithm to work the corresponding tridiagonal matrix must be *diagonally dominant*, that is:

$$|b_i| \geq |a_i| + |c_i|, \quad i = 0, \ldots, J. \tag{13.13}$$

For consistency, we assume that $a_0 = 0$ and $c_J = 0$.

The code that implements the Thomas algorithm is based on equations (13.10) to (13.12):

```
template <typename T>
    using Vector = std::vector<T>;
```

```
template <class T> class LUTridiagonalSolver
{ // Solve tridiagonal matrix equation
private:

   // Defining arrays (input)
   // V2 optimise so to work with pointers
   Vector<T> a;    // The lower-diagonal array [0..J]
   Vector<T> b;    // The diagonal array [0..J] "baseline array"
   Vector<T> c;    // The upper-diagonal array [0..J]
   Vector<T> r;    // The right-hand side of the equation
                   // Au = r [0..J]

   // Work arrays
   // Coefficients of Lower and Upper matrices: A = LU

   Vector<T> beta; // Range [0..J]
   Vector<T> gamma;// Range [0..J]
                   // Solutions of temporary and final problems
   Vector<T> z;    // Range [0..J]

   std::size_t Size;

   void calculateBetaGamma_ZU(Vector<T>& r)
   {

      // Equation 13.10
      beta[0] = b[0];
      gamma[0] = c[0] / beta[0];

      for (std::size_t j = 1; j < Size - 1; ++j)
      {
         beta[j] = b[j] - (a[j] * gamma[j - 1]);
         gamma[j] = c[j] / beta[j];

      }

      beta[Size - 1] = b[Size - 1] - (a[Size - 1] * gamma[Size - 2]);

      // Calculate z and u
      // Forward direction, equation 13.11
      z[0] = r[0] / beta[0];

      for (std::size_t j = 1; j < Size; ++j)
      {
         z[j] = (r[j] - (a[j] * z[j - 1])) / beta[j];

      }

      // Backward direction, equation 13.12
      r[Size - 1] = z[Size - 1];
```

```
      for (long i = Size - 2; i >= 0; --i)
      {
         r[i] = z[i] - (gamma[i] * r[i + 1]);
      }

   }

public:
   LUTridiagonalSolver() = delete;
   LUTridiagonalSolver(const LUTridiagonalSolver<T>& source) = delete;
   virtual ~LUTridiagonalSolver() = default;
   LUTridiagonalSolver<T>& operator = (const LUTridiagonalSolver<T>&
   source) = delete;

   LUTridiagonalSolver(Vector<T>& lower, Vector<T>& diagonal,
                       Vector<T>& upper, Vector<T>& RHS)
   {

      a = lower;
      b = diagonal;
      c = upper;
      r = RHS;

      Size = diagonal.size();

      beta = Vector<T>(Size);
      gamma = Vector<T>(Size);

      z = Vector<T>(Size);
   }

Vector<T> solve()
{
      calculateBetaGamma_ZU(r);      // Calculate beta and gamma

      return r;
}

Vector<T> operator () ()
{
   return solve();
}

bool diagonallyDominant() const
{
    if (std::abs(b[0]) < std::abs(c[0]))
       return false;

    if (std::abs(b[Size - 1]) < std::abs(a[Size - 1]))
       return false;
```

```
    for (std::size_t j = 1; j < Size - 1; ++j)
    {
      if (std::abs(b[j]) < std::abs(a[j]) + std::abs(c[j]))
         return false;
    }

    return true;
    }
};

#endif
```

In general, we have attempted to map the algorithm to code in a one-to-one manner in order to keep it readable and maintainable.

13.2.3 Examples

We now give a simple example to show how to prepare the input arrays to the algorithms, using the algorithms from Sections 13.2.1 and 13.2.2 and then examining the output. The first example is based on a special case of the two-point boundary value problem (13.1) as shown by the code comment:

```
/* Solve BVP u" = 1 in (0, 1) with u(0) = u(1) = 0
   Solution u(x) = x(1-x)
   Solve by FDM */
```

The C++ code is:

```
int main()
{
    using value_type = double;
    using Vector = std::vector<value_type>;

    std::size_t J = 20;
    std::cout << "Number of subdivisions J ";
    std::cin >> J;

    double h = 1.0 / static_cast<double>(J);

    // Boundary conditions
    double BCL = 0.0;       // LHS
    double BCR = 0.0;       // RHS

    // Double Sweep
    Vector a(J+1,1.0);
    Vector b(J+1,-2.0);
    Vector c(J+1,1.0);
    Vector r(J+1, -2.0*h*h);// Right-hand side

    // Thomas algorithm
    Vector a2(J-1, 1.0);
```

```
    Vector b2(J-1, -2.0);
    Vector c2(J-1, 1.0);
    Vector r2(J-1, -2.0*h*h);       // Right-hand side

    // Take the boundary conditions into consideration
    r2[0] -= BCL;
    r2[r2.size()-1] -= BCR;

    LUTridiagonalSolver<double> mySolver2(a2, b2, c2, r2);
    std::cout << "Matrix has a solution? "
    << mySolver2.diagonallyDominant() << '\n';
    DoubleSweep<value_type> mySolver(a, b, c, r, BCL, BCR);

    StopWatch<> sw;
    sw.Start();
    Vector result = std::move(mySolver());
    Vector result2 = std::move(mySolver2());
    sw.Stop();
    std::cout << "Elapsed time: " << sw.GetTime() << '\n';

    auto exact = [](double x) { return x*(1.0 - x); };
    double val = h;          // Double Sweep

    // Compare output from Double Sweep and Thomas with each other
    for (std::size_t j = 1; j < result.size()-1; ++j)
    { // The values should be zero

       std::cout<< j << ", " << result[j]-result2[j-1] <<",
       "<< exact(val)<< '\n';
       val += h;
    }
    return 0;
}
```

We note the difference in how the two algorithms handle the boundary values. This is worth stressing as we need to be clear on how to incorporate the Dirichlet boundary conditions into the algorithms. This is an important skill as you will need to be able to perform the same trick for more complicated problems in later chapters.

13.2.4 Performance Issues

We might consider improving the run-time performance of the algorithms in Sections 13.2.1 and 13.2.2. In their current form we have seen that the Double Sweep algorithm is approximately 20% more efficient than the Thomas algorithm (this might be due to the fact that the latter uses three work arrays while the former has been programmed using two work arrays). We also carried out some tests using the *Boost Pool* library (for fast memory allocation) which improves run-time efficiency even more. We have stress tested these solvers when applied to the diffusion equation (which we shall discuss in Section 13.3) using mesh sizes of 100,000 in both space and time. Both methods give accurate results. The processing times on a normal laptop *in seconds* were in the ranges [490, 588] for Double Sweep and [730, 780] for the Thomas algorithm.

We could also consider using move semantics instead of copying vectors. See Exercise 2 below. We could even consider letting the input vectors play the role of work arrays but we have not investigated this optimisation step.

Finally, you can use a different memory manager (for example, the *Boost C++ Pool* library or your own favourite library) to determine if it improves the run-time efficiency.

In Section 13.4 we introduce two variants of the *Alternating Direction Explicit* (ADE) method and we apply them to the one-dimensional heat equation. We also compare the run-time efficiency of ADE with that of the approximate solution of the heat equation using the Double Sweep and Thomas algorithms. We note that in the case of ADE no matrix inversion is needed.

13.2.5 Applications of Tridiagonal Matrices

In this book we are mainly interested in solving tridiagonal systems of equations that arise when we discretise PDEs and two-point boundary value problems. In particular, these systems arise when we approximate partial derivatives in the space dimension by three-point divided differences. In most cases we can use the popular Thomas algorithm to solve the resulting linear system, although the Double Sweep method performs better. In particular, the Thomas algorithm is used in the Alternating Direction Implicit (ADI) method which is one of the popular schemes in computational finance.

13.2.6 Some Remarks on Matrices

The matrix A in equation (13.3) should satisfy certain properties if the equation is to have a solution. To this end, we give a short discussion of some conditions for (13.3) to have a unique solution. We shall also deal with specific examples in later chapters when we model parabolic partial differential equations.

A matrix A is said to be *positive definite* if $v^{\top}Au > 0$ for any vector v. Here, v^{T} is the transpose vector of v and Av is the matrix–vector multiplication of A and v.

In general, it is difficult to directly prove that a matrix is positive definite so we need other conditions. One useful criterion is that of *diagonal dominance*. A matrix A is said to be diagonally dominant if for each row the absolute value of the diagonal element is greater than or equal to the sum of the absolute values of its non-diagonal elements for that row (using initial index equal to 1):

$$\begin{cases} A = (a_{ij})_{i,j=1,\ldots,n} \\ |a_{ii}| \geq \sum_{j=1}^{n} |a_{ij}| \text{ for } i = 1, \ldots, n, \quad i \neq j. \end{cases}$$

In the case where the matrix A is tridiagonal, this inequality takes on the following form as already seen in equation (13.13):

$$|b_j| \geq |a_j| + |c_j|, \quad j = 1, \ldots, n.$$

We introduce the class of M-matrices that arise when modelling convection–diffusion equations. An M-matrix is very attractive as we shall see in later chapters.

We say that a square matrix A is an *M-matrix* if it is non-singular, its inverse is non-negative and its off-diagonal elements are non-positive:

$$\begin{cases} A^{-1} \geq 0 \\ a_{ij} \leq 0 \; i \neq j, \forall \; 1 \leq i,j \leq n. \end{cases}$$

Sufficient conditions for the inverse of a matrix to be non-negative are given in Varga (1962). We restate the major result:

Lemma. Let the matrix A be irreducibly diagonally dominant and suppose that:

$$\left\{ \begin{array}{l} a_{ij} \leq 0 \; i \neq j \\ a_{ii} > 0 \end{array} \right\} \quad 1 \leq i,j \leq n$$

then A is non-singular and its inverse is strictly positive.

13.3 THE CRANK-NICOLSON AND THETA METHODS

We investigate the relative performance of the Double Sweep and Thomas algorithms for the initial boundary value problem for the one-dimensional heat equation (Tolstov, 1962) that describes heat flow in a rod of length L:

$$\begin{cases} \dfrac{\partial u}{\partial t} = a^2 \dfrac{\partial^2 u}{\partial x^2}, \quad 0 < x < L, \; t > 0 \\ u(x,0) = f(x), \quad 0 \leq x \leq L \\ u(0,t) = A, \; u(L,t) = B, \quad t > 0. \end{cases} \tag{13.14}$$

In this case we can assume without loss of generality that $L = 1$. Here, a, A and B are constants.

We can find a solution to the system (13.14) in the case when $A = B = 0$ and $a = 1$ by the method of *separation of variables* (Kreider et al., 1966). In this case the *analytical solution* is given by:

$$u(x,t) = \frac{8}{\pi^2} \sum_{n=1}^{\infty} \frac{1}{n^2} \left(\sin \frac{n\pi}{2} \right) (\sin n\pi x) \; \exp(-n^2 \pi^2 t) \tag{13.15}$$

and we use this solution as the benchmark against which the numerical solutions can be compared.

As a test case we discretise a parabolic PDE in the space direction only (using centred difference schemes, for instance) while keeping the time variable t continuous. We examine the following initial boundary value problem for the one-dimensional heat equation on the unit

interval with zero Dirichlet boundary conditions. It is easy to extend the idea to more general cases. The problem is:

$$\begin{cases} \dfrac{\partial u}{\partial t} = \dfrac{\partial^2 u}{\partial x^2}, 0 < x < 1, \quad t > 0 \\ u(0,t) = u(1,t) = 0, \quad t > 0 \\ u(x,0) = f(x), \quad 0 \le x \le 1. \end{cases} \tag{13.16}$$

We now partition the space interval (0;1) into J subintervals and we approximate (13.16) by the *semi-discrete* scheme:

$$\begin{cases} \dfrac{dU_j}{dt} = h^{-2}(u_{j+1} - 2U_j + U_{j-1}), \quad 1 \le j \le J-1 \\ U_0 = U_J = 0, \quad t > 0 \\ U_j(0) = f(x_j), \quad j = 1, \ldots, J-1 \end{cases} \tag{13.17}$$

where $h = 1/J$ is the constant mesh size.

We define the following vectors by:

$$U(t) = (U_1(t), \ldots, U_{J-1}(t))^\top$$
$$U_0 = (f(x_1), \ldots, f(x_{J-1}))^\top.$$

Then we can rewrite system (13.17) as a system of ordinary differential equations:

$$\begin{cases} \dfrac{dU}{dt} = AU, \quad t > 0 \\ U(0) = U_0 \end{cases} \tag{13.18}$$

where the matrix A is given by:

$$A = h^{-2} \begin{pmatrix} -2 & 1 & & & \\ & & & & 0 \\ 1 & \ddots & \ddots & & \\ 0 & \ddots & \ddots & 1 & \\ & & 1 & -2 & \end{pmatrix}.$$

Where do we go from here? There are a number of questions, for example:

- Q1: Does system (13.18) have a unique solution and what are its *qualitative* properties?
- Q2: How accurate is scheme (13.18) as an approximation to the solution of system (13.16)?
- Q3: How do we discretise (13.18) in time and how accurate is the resulting discretisation?

We shall discuss questions Q1 and Q2 in later sections and chapters. Here we discuss Q3. There are many alternatives ranging from *one-step methods* to *multi-step methods* (see Dahlquist and Björck, 1974, for example) and from *explicit* to *implicit methods*. We can use other approximate methods such as Runge–Kutta (Stoer and Bulirsch, 1980) that we discuss in Chapters 24 and 25. In this section we concentrate on one-step explicit and implicit methods to discretise system (13.18) defined by:

$$\begin{aligned} \frac{U^{n+1}-U^n}{\Delta t} &= \theta A U^{n+1} + (1-\theta) A U^n, \quad 0 \le n \le N-1, 0 \le \theta \le 1 \\ U^0 &= U_0. \end{aligned} \tag{13.19}$$

In this case Δt is the constant mesh size in time.

We can rewrite equation (13.19) in the equivalent form:

$$[I - \Delta t \theta A] U^{n+1} = (I + \Delta t(1-\theta)) A U^n \tag{13.20}$$

or formally as:

$$U^{n+1} = [I - \Delta t \theta A]^{-1} (I + \Delta t(1-\theta)) A U^n \tag{13.21}$$

where I is the identity matrix of size $J \times 1$.

Some special cases of θ are:

$$\begin{aligned} &\theta = 1, \text{ implicit Euler scheme} \\ &\theta = 0, \text{ explicit Euler scheme} \\ &\theta = \tfrac{1}{2}, \text{ Crank–Nicolson scheme.} \end{aligned} \tag{13.22}$$

When the schemes are implicit we can solve the system of equations (13.19) at each time level $n+1$ using the Double Sweep method or the Thomas algorithm. No matrix inversion is needed in the case of explicit schemes.

The formulation (13.18) is called the *Method of Lines* (MOL) and it corresponds to a semi-discretisation of system (13.16) in the space direction while keeping the time variable continuous (we discuss MOL in more detail in later chapters). We can write scheme (13.19)–(13.21) in component form:

$$\begin{aligned} \frac{U_j^{n+1}-U_j^n}{\Delta t} &= \theta\left(U_{j+1}^{n+1} - 2U_j^{n+1} + U_{j-1}^{n+1}\right)/h^2 \\ &\quad + (1-\theta)\left(U_{j+1}^n - 2U_j^n + U_{j-1}^n\right)/h^2, \quad 1 \le j \le J-1 \end{aligned} \tag{13.23}$$

or

$$\begin{aligned} U_j^{n+1} - U_j^n &= \lambda\theta\left(U_{j+1}^{n+1} - 2U_j^{n+1} + U_{j-1}^{n+1}\right) + \lambda(1-\theta)\left(U_{j+1}^n - 2U_j^n + U_{j-1}^n\right) \\ (\lambda &= \Delta t/h^2). \end{aligned} \tag{13.24}$$

Finally,

$$\begin{aligned} &-\lambda\theta U_{j+1}^{n+1} + (1 + 2\,\lambda\,\theta)U_j^{n+1} - \lambda\theta U_{j-1}^{n+1} \\ &\quad = \lambda(1-\theta)U_{j+1}^n + (1 - 2\lambda(1-\theta))U_j^n + \lambda(1-\theta)U_{j-1}^n \quad 1 \le j \le J-1. \end{aligned} \tag{13.25}$$

We see that system (13.25) is tridiagonal and we can apply the Double Sweep or Thomas algorithms to solve it. In the case of the explicit Euler scheme ($\theta = 0$) these algorithms are not needed because the solution at time level $n + 1$ can be explicitly computed:

$$U_j^{n+1} = \lambda U_{j+1}^n + (1 - 2\lambda)U_j^n + \lambda U_{j-1}^n, \quad 1 \le j \le J-1. \tag{13.26}$$

We note that the one-factor Black–Scholes PDE can be converted into the heat equation by a change of variables (Wilmott, Howison and Dewynne, 1995). We shall see how to assemble and solve more general problems in later chapters based on the methods that we have developed here.

13.3.1 C++ Implementation of the Theta Method for the Heat Equation

We give the C++ code that implements the algorithm (13.25). This is a one-step marching scheme called BTCS that computes the solution at time level $n + 1$ in terms of the solution at level n. Since there are three unknowns to be computed at each time level $n + 1$ we need to use the Double Sweep method or the Thomas algorithm. The main steps in the algorithm are:

- A1: Choose input parameters and generate meshes.
- A2: Define initial condition and boundary conditions.
- A3: Compute the solution at each time up to and including expiration.

This code can be used as a template for more general PDEs and schemes. We note that we use a vector to store the approximate solution at expiration. We would need a matrix to store the values at each time level for all mesh points in the more general case. The complete code for the scheme that approximates system (13.16) is:

```
double InitialCondition(double x)
{
   if (x >= 0.0 && x <= 0.5)
   {
      return 2.0 * x;
   }

   return 2.0 * (1.0 - x);
}

// BTCS scheme for the heat equation

long J = 20;
```

```
std::cout << "Number of space subdivisions: ";
std::cin >> J;

std::cout << "Number of time subdivisions: ";
long N = 100; std::cin >> N;

std::cout << "Theta method: 1) Implicit Euler, 2) Crank Nicolson: ";
long choice = 1; std::cin >> choice;
double theta = 1.0; if (2 == choice) theta = 0.5;

std::cout << "Expiration: ";
double T = 1.0; std::cin >> T;

// Space interval [A,B]
double A = 0.0;     // LHS
double B = 1.0;     // RHS
double h = (B-A) / static_cast<double>(J);
double k = T / static_cast<double>(N);

// Boundary conditions
double BCL = 0.0;
double BCR = 0.0;

double lambda = k /(h*h);

// Constructors with size, start index and value (Diagonals of matrix)
// J intervals, thus J-1 UNKNOWN internal points
// Dirichlet boundary conditions
Vector<double> a(J-1,-lambda*theta);
Vector<double> b(J-1,(1.0 + 2.0*lambda*theta));
Vector<double> c(J-1,-lambda*theta);
Vector<double> r(J-1, 0);   // Right-hand side NOT CONSTANT ANYMORE
                            // Boundary conditions into consideration

// Create mesh in space
Vector<double> xarr(J + 1); xarr[0] = A;
for (std::size_t j = 1; j < xarr.size(); ++j)
{
   xarr[j] = xarr[j - 1] + h;
}


Vector<double> vecOld(xarr.size()); // At time n
Vector<double> vecNew(xarr.size()); // At time n+1

// Initial condition
for (std::size_t j = 0; j < vecOld.size(); j++)
{
   vecOld[j] = InitialCondition(xarr[j]);
}
```

```
// We start at 1st time point
double current = k;

StopWatch<> sw;
sw.Start();

while (current <= T)
{
   // Update at new time level n+1

   // Compute inhomogeneous term
   for (std::size_t j = 1; j < r.size()-1; ++j)
   {
      r[j] = (lambda*(1.0-theta)*vecOld[j+1])
             + (1.0 - (2.0*lambda*(1.0-theta)))*vecOld[j]
             + (lambda*(1.0-theta)*vecOld[j-1]);
   }

   // DoubleSweep<double> mySolver(a, b, c, r, BCL, BCR);
   LUTridiagonalSolver<double> mySolver(a, b, c, r);
   vecNew = mySolver.solve();
   vecOld = vecNew;

   current += k;
}
sw.Stop();
std::cout << "Elapsed time: "
```

We use two arrays to store values at time levels n and $n + 1$. We note that this scheme becomes even easier to solve in the case of the explicit Euler method (13.26) (also known as the FTCS scheme) in which case we can compute the solution at time level $n + 1$ without the need to solve a tridiagonal system. This feature comes at a cost, because FTCS is only *conditionally stable* and a small time step is needed if we wish to avoid oscillations.

Having created the code, we test it to determine if it is a good approximation (in some sense) to the exact solution or to the de facto finite difference solution of the initial boundary value problem (13.16).

13.4 THE ADE METHOD FOR THE IMPATIENT

The explicit finite difference method for the heat equation *must satisfy* the constraint $k \leq h^2/2$ where k is the mesh size in time (some authors use Δt to denote the mesh size), in other words we must take very small time steps if we wish to get decent results. The method is first-order accurate in time which makes it unsuitable for real applications. However, an advantage of the method is that we do not need to use a tridiagonal solver to compute a solution at each time level. A question we could ask is whether we can discover explicit finite difference schemes that are explicit, unconditionally stable and (ideally) second-order accurate. Such a family of schemes would result in code that is easy to write and understand. Furthermore, we expect these methods to be very efficient. To this end, we introduce the ADE method (Barakat and

Clark, 1966; Duffy, 2009; Larkin, 1964; Saul'yev, 1964). We show how ADE works for the one-dimensional heat equation with Dirichlet boundary conditions. In later chapters we shall apply the method to more general PDEs (13.16).

To motivate ADE, let us first consider the *implicit Euler scheme* on a discrete mesh:

$$\frac{U_j^{n+1} - U_j^n}{k} = \frac{1}{h^2}\left(U_{j+1}^{n+1} - 2U_j^{n+1} + U_{j-1}^{n+1}\right) \quad 1 \le j \le J-1, \quad 0 \le n < NT. \quad (13.27)$$

In this case the index j corresponds to a mesh point in the x direction and the index n corresponds to a mesh point in the t direction. The mesh sizes h and k are defined by $h = \frac{1}{J}$, $k = \frac{1}{NT}$, respectively, where J is the number of subdivisions in the x direction and NT is the number of subdivisions in the t direction.

Scheme (13.27) is first-order accurate in time and second-order accurate in space. The solution at time level $n+1$ is found by solving a tridiagonal matrix system. On the other hand, the *explicit Euler scheme*:

$$\frac{U_j^{n+1} - U_j^n}{k} = \frac{1}{h^2}\left(U_{j+1}^{n} - 2U_j^{n} + U_{j-1}^{n}\right) \quad 1 \le j \le J, \quad 0 \le n \le NT \quad (13.28)$$

is also first-order accurate and the solution at time level $n+1$ can be found without having to solve a matrix system. However, this scheme is *conditionally stable* and this leads to the constraint $k \le h^2/2$ in order to keep it stable.

In order to motivate ADE, we consider equation (13.27) again because ADE is a variation on it. The ADE method consists of two *sweeps* (which incidentally can be executed in parallel); the first sweep computes the solution at time level $n+1$ at indices $j-1$ and j while the second sweep computes the solution at time level $n+1$ at indices $j+1$ and j (Barakat and Clark, 1966):

$$\frac{U_j^{n+1} - U_j^n}{k} = \frac{1}{h^2}\left(U_{j+1}^{n} - U_j^{n} - U_j^{n+1} + U_{j-1}^{n+1}\right) \quad 1 \le j \le J-1, \quad n \ge 0 \quad (13.29)$$

$$\frac{V_j^{n+1} - V_j^n}{k} = \frac{1}{h^2}\left(V_{j+1}^{n+1} - V_j^{n+1} - V_j^{n} + V_{j-1}^{n}\right), \quad J-1 \ge j \ge 1, \quad n \ge 0. \quad (13.30)$$

Finally, we average the solutions from each sweep to give the final approximate solution:

$$u_j^n = \frac{1}{2}\left(U_j^n + V_j^n\right), \quad 0 \le j \le J, \quad n \ge 0. \quad (13.31)$$

If we examine these schemes carefully, we see that the solutions U and V at time level $n+1$ can be explicitly computed without having to solve a matrix system. First, schemes (13.29) and (13.30) are *conditionally consistent* with equation (13.16) and have $O(\frac{k}{h})$ local truncation error. By adding these schemes to produce (13.31) we get a stable explicit scheme with $O(h^2 + k^2)$ local truncation error and hence it is second-order accurate; see, for example, Larkin (1964) where the two-dimensional diffusion equation is discussed while the original textbook on the method is Saul'yev (1964). See also Exercise 4.

The *computational* form of the schemes (13.29), (13.30) is:

$$U_j^{n+1}(1+\lambda) = U_j^n(1-\lambda) + \lambda\left(U_{j+1}^n + U_{j-1}^{n+1}\right), \quad \left(\lambda = \frac{k}{h^2}\right)$$

and

$$V_{\}j}^{n+1}(1+\lambda) = V_j^n(1-\lambda) + \lambda\left(V_{j+1}^{n+1} + V_{j-1}^n\right).$$

The terms on the right-hand side of these equations are known.

13.4.1 C++ Implementation of ADE (Barakat and Clark) for the Heat Equation

We have discussed the Crank–Nicolson method in Section 13.3. It is unconditionally stable and second-order accurate. However, we need to solve a tridiagonal matrix system at each time level, which can have an impact on run-time performance. The ADE method is unconditionally stable, explicit and second-order accurate (the Barakat and Clark version; see Barakat and Clark, 1966). The C++ implementation of the discrete scheme (13.29)–(13.31) is:

```
long J = 20;

std::cout << "Number of space subdivisions: ";
std::cin >> J;

std::cout << "Number of time subdivisions: ";
long N = 100; std::cin >> N;

std::cout << "Expiration: ";
double T = 1.0; std::cin >> T;

// Space interval [A,B]
double A = 0.0;      // LHS
double B = 1.0;      // RHS
double h = (B-A) / static_cast<double>(J);
double k = T / static_cast<double>(N);


// Boundary conditions
double BCL = 0.0;
double BCR = 0.0;

double lambda = k /(h*h);

// Create mesh in space
auto xarr = CreateMesh(J, A, B);

Vector<double> U(xarr.size());      // L-R sweep
Vector<double> V(xarr.size());      // R-L sweep
```

```
Vector<double> vecNew(xarr.size()); // At time n+1

// Initial condition
for (std::size_t j = 0; j < U.size(); j++)
{
   U[j] = V[j] = InitialCondition(xarr[j]);
}

// We start at 1st time point
double current = k;

StopWatch<> sw;
sw.Start();

double OnePlusLambda = 1.0/(1.0 + lambda);
double OneMinusLambda = 1.0 - lambda;

while (current <= T)
{
   // Update at new time level n+1

   // (Dirichlet) boundary conditions
   vecNew[0] = BCL;
   vecNew[vecNew.size()-1] = BCR;

   // Up Sweep
   for (std::size_t j = 1; j < U.size()-1; ++j)
   {
      U[j] = (U[j] * OneMinusLambda + lambda*(U[j + 1] + U[j - 1])) *
      OnePlusLambda;
   }

   // Down Sweep
   for (std::size_t j = V.size() - 1; j >= 1; --j)
   {
      V[j] = (V[j] * OneMinusLambda + lambda*(V[j + 1] + V[j - 1])) *
      OnePlusLambda;
   }

   // Barakhat and Clark update
   for (std::size_t j = 0; j < vecNew.size(); ++j)
   {
      vecNew[j] = 0.5*(U[j] + V[j]);
   }

   current += k;
}
sw.Stop();
std::cout << "Elapsed time: " << sw.GetTime() << '\n';
```

We conclude the discussion of the numerical approximation of the heat equation by comparing the run-time performance of the Crank–Nicolson (CN), Saul'yev ADE (ADEI) and Barakat and Clark ADE (ADEII) methods. In general, it is a trade-off between efficiency and the number of operations in order to achieve a given accuracy. We provide some basic conclusions:

- ADEI is fastest, then ADEII followed by CN (in general, ADEI is three times faster than CN, ADEII is [20, 50]% faster than CN depending on the size of the problem).
- CN and ADEII deliver more or less the same level of accuracy. ADEI is the least accurate of the three methods.
- For ADE in general, the rule of thumb for accurate results is $NT \sim 5*NX$.

We recommend that you perform your own experiments on the PDEs that are of interest to you.

13.5 CUBIC SPLINE INTERPOLATION

In this section we discuss a special kind of function called the *cubic spline* (Ahlberg, Nilson and Walsh, 1967). This is a function that is defined in an interval and whose values are equal to given values at prescribed mesh points in the interval. In each subinterval the spline is a third-order polynomial and both it and its first and second derivatives are continuous at the interior mesh points. We construct the cubic spline by solving a tridiagonal system of equations (that we already know how to solve and hence is a good way to test and debug our code). Our original aim in this chapter was primarily to test the accuracy of tridiagonal solvers but we then realised that it is useful in a wider context, for example:

- Interpolation of discrete data and producing a smooth function.
- Approximating continuous functions.
- Numerical integration (Stroud, 1974).
- Solving boundary value problems.

In interest rate applications cubic splines are used in the construction of yield curves and forward curves. In general, the x-axis corresponds to time while the y-axis can correspond to zero rates, forward rates, discount rates or some function of these rates (Duffy and Germani, 2013; Ron, 2000). More generally, cubic splines are used to interpolate between values that are known at discrete mesh points. We shall see examples in later chapters when we approximate the solution of the Black–Scholes PDE by the finite difference method. Having computed the option price at discrete points we can then interpolate these values to produce a value for any arbitrary value of the underlying variable.

We discuss two popular interpolation methods in this section. First, consider the pair of data values (x_0, y_0) and (x_1, y_1). We are interested in drawing a straight line between these points. Some algebra shows that the value y corresponding to a given value x is given by:

$$y = y_0 + (x - x_0)\frac{y_1 - y_0}{x_1 - x_0}. \tag{13.32}$$

This formula can now be extended to a data set consisting of n points. It has the advantage that it never overshoots or oscillates. However, the generated curve is only continuous as the first-order derivatives are discontinuous at the data points. Thus, the curve looks somewhat 'jagged'.

It is important to note that linear interpolation incurs an error that is bounded by the second derivative of the function being approximated.

We now introduce the cubic spline method. We work on the interval (a, b) and let us suppose that $\delta = \{x_j : j = 1, 2, \ldots, n\}$ is a *partition* of (a, b) with mesh points $a = x_0 < x_1 < \cdots < x_n = b$. We define function values as follows:

$$Y = \{y_j : j = 0, 1, \ldots, n\}$$

and the quantities:

$$h_{j+1} = x_{j+1} - x_j, \quad j = 0, 1, \ldots, n-1.$$

We are now ready to define what a cubic spline is. This is a continuous function whose derivatives up to and including order two are continuous at the interior mesh points. Furthermore, the spline is a polynomial of third degree on each subinterval.

In order to uniquely specify the spline $S_{\pm}$ we give 'boundary conditions' at $x = 1$ and at $x = b$. The mutually exclusive options are as follows:

$$S''_\delta(Y; a) = S''_\delta(Y; b) = 0$$

and (13.33)

$$S'_\delta(Y; a) = \alpha, \quad S'_\delta(Y; b) = \beta \quad (\alpha, \beta \text{ given}).$$

In this case, either the second-order derivatives of the spline function are zero at $x = a$ and at $x = b$ or the first-order derivatives of the spline function are given at $x = a$ and at $x = b$ and have values α and β, respectively. It can be shown (Stoer and Bulirsch, 1980) that:

$$S_\delta(Y; x) = M_j \frac{(x_{j+1} - x)^3}{6h_{j+1}} + M_{j+1} \frac{(x - x_j)^3}{6h_{j+1}} + A_j(x - x_j) + B_j$$

where:

$$A_j = \frac{y_{j+1} - y_j}{h_{j+1}} - \frac{h_{j+1}}{6}(M_{j+1} - M_j)$$

$$B_j = y_j - M_j \frac{h_{j+1}^2}{6}, \quad j = 0, \ldots, n-1.$$

All parameters and coefficients are known with the exception of $\{M_j\}_{j=0}^n$ and we can solve for these parameters by writing the problem as a *tridiagonal matrix system*:

$$AM = d \tag{13.34}$$

where:

$$A = \begin{pmatrix} 2 & \lambda_0 & & & & \\ \mu_1 & 2 & \lambda_1 & & 0 & \\ & \mu_2 & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & \ddots & \ddots & \lambda_{n-1} \\ & 0 & & & \mu_n & 2 \end{pmatrix}$$

$$M = (M_0, \ldots, M_n)^\top, \quad d = (d_0, \ldots, d_n)^\top$$

where some of the elements of the matrix A depend on the boundary conditions (13.33), namely zero values for the second-order derivatives (case (a) below) or given values for the first-order derivatives (case (b) below). Some cases are:

Case (a)

$$\lambda_0 = 0, \quad d_0 = 0, \ \mu_n = 0, \quad d_n = 0.$$

Case (b)

$$\lambda_0 = 1, \quad d_0 = \frac{6}{h_1}\left(\frac{y_1 - y_0}{h_1} - \alpha\right)$$

$$\mu_n = 1, \quad d_n = \frac{6}{h_n}\left(\beta - \frac{y_n - y_{n-1}}{h_n}\right)$$

$$\lambda_j = \frac{h_{j+1}}{h_j + h_{j+1}}, \quad \mu_j = 1 - \lambda_j = \frac{h_j}{h_j + h_{j+1}}$$

$$d_j = \frac{6}{h_j + h_{j+1}}\left\{\frac{y_{j+1} - y_j}{h_{j+1}} - \frac{y_j - y_{j-1}}{h_j}\right\} \quad j = 1, 2, \ldots, n-1.$$

We have reduced the problem of interpolating data points by cubic splines to a known problem of solving a tridiagonal matrix system.

We now discuss the C++ code that implements the algorithm for cubic spline interpolation. The main member functions are:

- Construct a cubic spline based on x (abscissa) and y (values) input arrays (discrete case).
- Construct a cubic spline based on x array (abscissa) and a function (continuous case).
- Compute a y value for a given x value.
- Compute the integral of the cubic spline function.
- Compute the first and second derivatives of the cubic spline function.

The code in this book supports the boundary conditions as stated in equation (13.33); however, periodic boundary conditions are not supported in this version of the software.

We discuss the code for each group of member functions. First, the member data of the class is:

```
enum CubicSplineBC {SecondDeriv, FirstDeriv};
```

The value `SecondDeriv` corresponds to the case of a *natural spline* (second derivatives are zero at end-points) while the value `FirstDeriv` corresponds to the *clamped spline* case (first derivatives specified at end-points).

```
class CubicSplineInterpolator
{
private:
   std::vector<double> x;   // Abscissa x-values x_0,...,x_N
   std::vector<double> y;   // Function values
   std::vector<double> h;   // Array of mesh sizes in x direction

   CubicSplineBC type;      // Type of BC

   std::vector<double> M;                // Moments of spline
   std::vector<double> A, B, C, r;       // Input arrays for LU

   // For first order derivatives
   double a, b;

   // ...
};
```

The two constructors are:

```
CubicSplineInterpolator(const std::vector<double>& xarr,
                        const std::vector<double>& yarr,
                        CubicSplineBC BCType,
                        double alpha = 0.0,   double beta = 0.0)
{ // Discrete function value case

   // Arrays must have the same size
   x = xarr;
   y = yarr;
   type = BCType;


   a = alpha;   // LHS
   b = beta;    // RHS
   std::size_t N = xarr.size();

   // Calculate array of offset
   h = std::vector<double>(N, 0.0);
   for (std::size_t j = 1; j < h.size(); ++j)
   {
      h[j] = x[j] - x[j-1];
   }
```

```
    // All arrays have start index 1
    // Compared to the equations in the book, M(j) --> M(j+1)
    M = std::vector<double>(N, 0.0); // Solution

    // LU Coefficients
    A = std::vector<double>(N, 0.0);
    B = std::vector<double>(N, 2.0);// Diagonal vector, constant == 2
    C = std::vector<double>(N, 0.0);
    r = std::vector<double>(N, 0.0);


    // Calculate the elements
    CalculateVectors();

    LUTridiagonalSolver<double> mySolver(A, B, C, r);


    M = mySolver.solve();
}

CubicSplineInterpolator(const std::vector<double>& xarr,
                        const std::function<double (double)>& fun,
                        CubicSplineBC BCType,
                        double alpha = 0.0, double beta = 0.0)
{ // Continuous function value case

    std::size_t N = xarr.size();

    // Arrays must have the same size
    x = xarr;
    y = CreateDiscreteFunction(xarr, fun);
    type = BCType;


    a = alpha;   // LHS
    b = beta;    // RHS

    // Calculate array of offset
    h = std::vector<double>(N, 0.0);
    for (std::size_t j = 1; j < h.size(); ++j)
    {
       h[j] = x[j] - x[j - 1];
    }

    // All arrays have start index 1
    // Compared to the equations in the book, M(j) --> M(j+1)
    M = std::vector<double>(N, 0.0); // Solution

    // LU Coefficients
    A = std::vector<double>(N, 0.0);
    B = std::vector<double>(N, 2.0);// Diagonal vector, constant == 2
```

```
    C = std::vector<double>(N, 0.0);
    r = std::vector<double>(N, 0.0);

    // Calculate the elements
    CalculateVectors();

    LUTridiagonalSolver<double> mySolver(A, B, C, r);

    M = mySolver.solve();
}
```

The code to generate the coefficients of the tridiagonal system is:

```
// Private member functions
void CalculateVectors()
{ // A, B, C and r

    std::size_t N = x.size()-1;

    if (type == SecondDeriv)
    {
       C[0] = 0.0;
       r[0] = 0.0;

       A[A.size()-1] = 0.0;
       r[r.size()-1] = 0.0;
    }
    else
    {
       C[0] = 1.0;
       r[0] = 6.0 * ((y[1] - y[0])/h[1] - a)/h[1];

       A[A.size()-1] = 1.0;
       r[r.size()-1] = 6.0 * (b - ((y[N] - y[N-1])/h[N]))/h[N];
    }

    double tmp;

    for (long j = 1; j < x.size()-1; ++j)
    {

       double fac = 1.0 / (h[j] + h[j + 1]);
       C[j] = h[j+1] * fac;
       A[j] = h[j] * fac;

       tmp = ((y[j+1] - y[j])/h[j+1]) - ((y[j] - y[j-1])/h[j]);
       r[j] = (6.0 * tmp) * fac;
    }

}
```

The code that solves system (13.34) is:

```
double Solve(double xvar) const
{ // Find the interpolated valued at a value x)

   std::size_t j = findAbscissa(x, xvar);// Index of LHS value <= x
   // Now use the formula
   double tmp = xvar - x[j];
   double tmpA = x[j+1] - xvar;

   double tmp3 = tmp * tmp * tmp;
   double tmp4 = tmpA * tmpA * tmpA;

   double A = (y[j+1] - y[j])/h[j+1] - (h[j+1] * (M[j+1] - M[j]))/6.0;
   double B = y[j] - (M[j] * h[j+1] * h[j+1])/6.0;

   double result = (M[j] * tmp4)/(6.0 * h[j+1])
                   + (M[j+1] * tmp3)/(6.0 * h[j+1])
                          + (A * tmp)
                                  + B;
   return result;
}
```

We notice that this code calls a function to find the subinterval containing a given abscissa value:

```
std::size_t findAbscissa(const std::vector<double>& x, double xvar)
{ // Will give index of LHS value <= xvar.


   if (xvar < x[0] || xvar > x[x.size() - 1])
   {
      std::string s = "\nValue "
         + boost::lexical_cast<std::string>(xvar)
         + " not in range "
         + "(" + boost::lexical_cast<std::string>(x[0]) + ","
         + boost::lexical_cast<std::string>(x[x.size() - 1]) + ")";

      throw std::out_of_range(s);
   }

   // This algo has log complexity
   auto posA = std::lower_bound(std::begin(x), std::end(x), xvar);

   std::size_t index = std::distance(std::begin(x), posA);

   return index;
}
```

This function can throw an exception, which means that client code must define a corresponding `try/catch` block.

The integral of the spline over the interval (a, b) results directly from its definition (Ahlberg, Nilson and Walsh, 1967, p. 44):

$$\int_a^b S_\delta(x)dx = \sum_{j=1}^{n} \frac{f_{j-1} + f_j}{2} h_j - \sum_{j=1}^{n} \frac{M_{j-1} + M_j}{24} h_j^3 \tag{13.35}$$

and the corresponding code is:

```
double Integral() const
{ // ANW page 45

   double r1 = 0.0; double r2 = 0.0;

   for (std::size_t j = 1; j < x.size(); ++j)
   {
      double hj = h[j];
      r1 += (y[j - 1] + y[j]) * hj;
      r2 -= (M[j - 1] + M[j]) * hj * hj * hj;
   }

   r1 *= 0.5;
   r2 /= 24.0;

   return r1 + r2;
}
```

In the case of a constant mesh size the integral in equation (13.35) has the simpler form:

$$\int_a^b S_\delta(x)dx = h \sum_{j=1}^{N} \frac{f_{j-1} + f_j}{2} - \frac{h^3}{12} \sum_{j=1}^{n} \frac{M_{j-1} + M_j}{2}. \tag{13.36}$$

Differentiation of the formula for the cubic spline leads to representations for its first two derivatives:

$$S''_\delta(x) = M_j \left(\frac{x_{j+1} - x}{h_{j+1}} \right) + M_{j+1} \left(\frac{x - x_j}{h_{j+1}} \right) \quad \text{for} \quad x \in [x_j, x_{j+1}] \tag{13.37}$$

$$S'_\delta(x) = -M_j \left(\frac{(x_{j+1} - x)^2}{2h_{j+1}} \right) + M_{j+1} \left(\frac{(x - x_j)^2}{2h_{j+1}} \right) + A_j \quad \text{for} \quad x \in [x_j, x_{j+1}]. \tag{13.38}$$

The code is:

```
std::tuple<double, double, double> ExtendedSolve(double xvar)
{ // Solve for S and derivatives S', S"

   auto j = findAbscissa(x, xvar);

   double Mj = M[j]; double Mjp1 = M[j + 1];

   double hjp1 = 1.0 / h[j + 1];
```

```
    double xj = x[j]; double xjp1 = x[j + 1];
    double tmp = xvar - x[j];
    double tmpA = x[j + 1] - xvar;

    double tmp3 = tmp * tmp * tmp;
    double tmp4 = tmpA * tmpA * tmpA;

    // S"
    double s2 = hjp1*(Mj*(xjp1 - xvar) + Mjp1*(xvar - xj));

    double A = (y[j + 1] - y[j]) / h[j + 1] - (h[j + 1] *
    (M[j + 1] - M[j])) / 6.0;
    double B = y[j] - (M[j] * h[j + 1] * h[j + 1]) / 6.0;

    // S'
    double s1 = -0.5*hjp1*Mj*tmpA*tmpA + 0.5*hjp1*Mjp1*tmp*tmp + A;

    // S
    double s0 = (M[j] * tmp4) / (6.0 * h[j + 1])
                + (M[j + 1] * tmp3) / (6.0 * h[j + 1])
                + (A * tmp)
                + B;

    return std::make_tuple(s0, s1, s2);
}

double Derivative(double xvar)
{ // Solve for derivative S'

    auto j = findAbscissa(x, xvar);

    double Mj = M[j]; double Mjp1 = M[j + 1];

    double hjp1 = 1.0 / h[j + 1];
    double xj = x[j]; double xjp1 = x[j + 1];
    double tmp = xvar - x[j];
    double tmpA = x[j + 1] - xvar;

    double tmp3 = tmp * tmp * tmp;
    double tmp4 = tmpA * tmpA * tmpA;

    double A = (y[j + 1] - y[j]) / h[j + 1] - (h[j + 1] *
    (M[j + 1] - M[j])) / 6.0;

    // S'
    double s1 = -0.5*hjp1*Mj*tmpA*tmpA + 0.5*hjp1*Mjp1*tmp*tmp + A;

    return s1;
}
```

It is worth noting that one of these functions returns a tuple. We stress that the use of tuples can impact performance depending on the frequency with which they are created and returned from functions.

13.5.1 Examples

The first example is to test cubic spline interpolation applied to the standard normal distribution. The code for the probability distribution function, its first and second derivatives as well as cumulative distribution functions is:

```
double Pdf(double x)
{ // Probability function for Gauss fn.

   double A = 1.0 / sqrt(2.0 * 3.14159265358979323846);
   return A * exp(-x*x*0.5);
}

double dPdfdx(double x)
{ // 1st derivative of probability function for Gauss fn.

   return -x * Pdf(x);
}

double d2Pdfdx2(double x)
{ // 2nd derivative of probability function for Gauss fn.

   return -Pdf(x) + x*x*Pdf(x);
}


double Cdf(double x)
{ // The approximation to the cumulative normal distribution

   // C++11
   return 0.5* (1.0 + std::erf(x / std::sqrt(2.0)));
}
```

We take these functions because they are important in computational finance and we can check our results against them. We test the accuracy of the interpolator by examining the values at a random point (we discuss random number generation in more detail in Chapter 26). The code is:

```
// Infinite interval, truncated to [a,b]
double a = -6.0;                  // Left of interval
double b = 6.0;                   // Right of interval
std::size_t n = 200000;           // N.B. n subintervals
double h = (b - a) / static_cast<double>(n);

std::vector<double> xarr = CreateMesh(n, a, b);
```

```
auto fun = Pdf;
auto yarr = CreateDiscreteFunction(xarr, fun);

// Generate a random number in [a,b]
std::default_random_engine eng;
std::random_device rd;
eng.seed(rd());

std::uniform_real_distribution<double> dist(a, b);
double xvar = dist(eng);
```

We now create a cubic spline interpolator and we interpolate at a random point as follows:

```
CubicSplineInterpolator csi(xarr, yarr, SecondDeriv);
try
{
   double result = csi.Solve(xvar);
   std::cout << "Interpolated value at " << xvar << " " <<
   std::setprecision(16) << result << ", is " << fun (xvar);

   std::cout << "Integral: " << csi.Integral() << '\n';
}
catch (std::exception& e)
{ // Catch not in range values

   std::cout << e.what() << '\n';
}
```

Finally, we test some other member functions as follows:

```
// Numerical Differentiation
auto derivs = csi.ExtendedSolve(xvar);

std::cout << "Derivatives, approx: " << std::get<0>(derivs) << ", "
          << std::get<1>(derivs) << ", "<< std::get<2>(derivs) << '\n';

std::cout << "Derivatives, exact: " << Pdf(xvar) << ", "
          << dPdfdx(xvar) << ", " << d2Pdfdx2(xvar) << '\n';

// 1st derivative
xvar = dist(eng);
std::cout << "1st derivative: " << xvar << ", " << dPdfdx(xvar)
          << ", " << csi.Derivative(xvar) << '\n';
```

This completes our example. The rationale can be applied to other use cases such as:

- U1: We have a discrete data set but no analytical form for a smooth function.
- U2: Efficiency reasons: in many cases it is easier to use splines rather than compute intensive or discontinuous functions.

- U3: The method can be used to interpolate a function as well as its first and second derivatives.

We are assuming that the function to be approximated is smooth (see Ahlberg, Nilson and Walsh, 1967 for the mathematical details). In a sense we see the cubic spline as a *surrogate* for the function that we are investigating.

13.5.2 Caveat: Cubic Splines with Sparse Input Data

Cubic splines are well behaved for many kinds of applications but they can overshoot and produce spurious negative values, especially for non-equidistant data. This problem manifests itself in yield curve construction. An example is (Duffy and Germani, 2013):

```
std::vector<double> t{0.1,1.0,4.0,9.0,20.0,30.0};        // Time in years
std::vector<double> r{0.081,0.07,0.044,0.07,0.04,0.03}; // Interest rate
```

In these cases other methods are better, for example the Hagan–West, Akima and Hyman methods (Duffy and Germani, 2013; Hagan and West, 2006, 2008).

We give a brief mathematical explanation of why cubic splines fail under certain circumstances. To this end, we concentrate on the constraints on the distribution of the mesh values. Let $\{\Delta_k\},\ k = 0,\ 1,\ 2,\ \ldots$ be a sequence of meshes consisting of N_k subintervals on the interval $[a, b]$ defined by:

$$\Delta_k : a = x_{k,0} < x_{k,1} < \cdots < x_{k,N_k} = b.$$

We define the norm of Δ_k as:

$$\|\Delta_k\| = \max_{1 \le j \le N_k} (h_{k,j}) \quad \text{where} \quad h_{k,j} = x_{k,j} - x_{k,j-1}, \quad j = 1, \ldots, N_k.$$

We are interested in sequences $\{\Delta_k\}$ for which $\|\Delta_k\| \to 0$ as $k \to \infty$. We also require that:

$$R_{\Delta_k} \equiv \max_{1 < j \le N_k} \frac{\|\Delta_k\|}{h_{k,j}} \le \beta < \infty. \tag{13.39}$$

This inequality states that the ratio of the largest subinterval and the smallest subinterval is bounded. The inequality occurs in many branches of numerical analysis.

As an example, consider the mesh $\{1,2,3,3.1,5.1,6,7,8\}$ that occurs in yield curve construction. The mesh sizes are $\{1,1,0.1,2,0.9,1,1\}$ and hence $\Delta = 2$, $R_\Delta = 20$ (we have dropped the dependence on k).

The importance of constraint (13.39) lies in the fact that cubic splines produce approximations to smooth functions. The following theorem is proved in Ahlberg, Nilson and Walsh (1967, pp. 29–34):

Theorem 1. Let $f(x)$ be of class $C^3[a, b]$ (f and its first three derivatives are continuous on the interval $[a, b]$). Assume that $\lim_{k\to\infty} \|\Delta_k\| = 0$ and that the mesh Δ_k satisfies constraint

(13.39). Then we have $f^{(p)}(x) - S^{(p)}(x) = o(\|\Delta_k\|^{3-p})(p = 0, 1, 2, 3)$ (where $f^{(p)}(x)$ is the pth derivative of f with respect to x).

13.6 SOME HANDY UTILITIES

In this chapter we examined a number of algorithms and their implementation in C++. In particular, the numerical methods produced an array of values as output. But how do we know that the results are accurate and that the algorithm is run-time efficient? We focus on accuracy and we discuss how to execute *initial tests*. Some attention areas are:

- A1: We need an exact or de-facto exact solution that we can test the approximate solutions against.
- A2: What is the maximum error (measured in some norm) between exact and approximate solutions and at which points in index space does it occur?
- A3: What is the error (measured in some norm) between exact and approximate solutions at some randomly chosen point?
- A4: In which subinterval is a given point?
- A5: Create a mesh array.
- A6: Create an array of values from functions of arity one and two.

We have used a number of these choices when testing the code in this chapter. For A2 the return type is a tuple consisting of the maximum error and where that error occurs:

```
double LInfinityNorm(const std::vector<double>& v1,
                     const std::vector<double>& v2)
{ // Max of absolute value of elements of v1 - v2

      double result = std::abs(v1[0] - v2[0]);

      for (std::size_t j = 1; j < v1.size(); ++j)
      {
         result = std::max<double>(result, std::abs(v1[j] - v2[j]));
      }

      return result;
}

std::tuple<double,std::size_t> HotSpotError(
                               const std::vector<double>& v1,
                               const std::vector<double>& v2)
{ // Max abs value of elements of v1 - v2; identify *where* it occurs

      double result = std::abs(v1[0] - v2[0]);
      std::size_t index = 0;

      for (std::size_t j = 1; j < v1.size(); ++j)
      {
```

```
            double tmp = std::max<double>(result, std::abs(v1[j] - v2[j]));

            if (result < tmp)
            { // Find the max difference

               result = tmp;
               index = j;
            }
         }

         return std::make_tuple(result, index);
   }
```

An example of use is:

```
Vector<double> vecNew(xarr.size());
Vector<double> exact(xarr.size());
std::cout << "Max Error: " << LInfinityNorm(exact, vecNew) << '\n';
auto val = HotSpotError(exact, vecNew);
std::cout << "Max Error at hotspot: " << std::get<0>(val) << ", "
          << std::get<1>(val); << '\n';
```

For use case A3 we use the functionality from the C++ `<random>` library to generate uniform random numbers in a bounded interval. An example of use is:

```
// Error between exact and approximate solutions at random index
std::default_random_engine eng;
std::random_device rd;
eng.seed(rd());

std::uniform_int_distribution<int> uniform(1, vecNew.size() - 1);

auto index = uniform(eng);
std::cout << "\nExact, approx values: " << exact[index] << ","
          << vecNew[index] << '\n';
```

The output produced from this code should be in line with expectations, otherwise we know that there is a bug in the code.

13.7 SUMMARY AND CONCLUSIONS

In this chapter we have examined a number of methods and algorithms in numerical analysis. The objective was to map these algorithms to C++ in an unambiguous way and to ensure that the C++ code produces accurate results. The experience and results in this chapter will be generalised and extended to larger problems in later chapters.

13.8 EXERCISES AND PROJECTS

1. (Double Sweep Method)

Generalise the Double Sweep method from one that supports Dirichlet boundary conditions to one that supports *Robin boundary conditions*:

$$\begin{cases} a_j U_{j-1} + b_j U_j + c_j U_{j+1} = f_j, \quad 1 \le j \le J-1 \\ U_0 = \alpha U_1 + \varphi \\ U_{J-1} = \beta U_J + \psi \end{cases}$$

where α, β, φ and ψ are given constants. You should get equations similar to those in Section 13.2.1.

Test the new algorithm by discretising the two-point boundary value problem:

$$\begin{cases} \dfrac{d^2u}{dx^2} = f(x), \quad 0 < x < 1 \\ u(0) = \varphi \\ \dfrac{du(1)}{dx} + \beta u(1) = \psi \end{cases}$$

using the following finite difference scheme:

$$\begin{cases} U_{j+1} - 2U_j + U_{j-1} = h^2 f_j, \quad 1 \le j \le J-1 \\ U_0 = \varphi \\ \dfrac{U_J - U_{J-1}}{h} + \dfrac{\beta}{2}(U_J + U_{J-1}) = \psi. \end{cases}$$

We note that the averaged approximation at the boundary $x = 1$ is second-order accurate (see Morton and Mayers, 1994).

Answer the following questions:

a) Modify the Double Sweep algorithm so that it can be applied to problems with Robin boundary conditions.

b) Test the new code on the above two-point boundary value problem. Assemble the system of equations and determine the conditions under which the tridiagonal matrix is diagonally dominant, hence ensuring that the system has a unique solution. When is the matrix an M-matrix?

c) Solve the two-point boundary value problem using the Thomas algorithm and compare the efficiency and accuracy with the results in part b).

d) Consider the first-order approximation at $x = 1$:

$$\frac{U_J - U_{J-1}}{h} + \beta U_J = \psi.$$

Integrate this option into the code that implements Double Sweep. Test the code on the above two-point boundary value problem. Does first-order accuracy or the boundary affect accuracy elsewhere?

2. (Performance of Tridiagonal Solvers)

The objective of this exercise is to improve the performance of the tridiagonal solvers. Answer the following questions:

a) Consider using move semantics instead of copy constructors when initialising the arrays in the solvers.

b) For the Thomas algorithm can we 'sacrifice' one or more of the input arrays to reduce the number of internal work arrays?

c) We can consider using specialised (your favourite) memory managers, as the following example shows:

```
// Double Sweep with a memory allocator
/*#include <boost/pool/pool.hpp>
#include <boost/pool/pool_alloc.hpp>*/

DoubleSweep<double, std::vector, boost::pool_allocator<double>>
mySolver3(a, b, c, r, BCL, BCR);

Vector result = mySolver3();
auto exact = [](double x) { return x*(1.0 - x); };
double val = 0;

// Compare output from Double Sweep and Thomas
for (std::size_t j = 0; j < result.size(); ++j)
{ // The values should be zero

   std::cout << j << ", " << result[j] << ", " << exact(val);
   val += h;
}
```

Run the code and compare the run-time performance with the default C++ allocator.

d) Most of the examples in this book use tridiagonal matrices of size 1000 at most. Is there any point trying to improve the run-time performance by parallelising the code using threads or tasks? Consider starting and stopping threads; the multithreaded code may have worse performance than the sequential code. We discuss parallel programming in Chapters 28 to 30.

3. (Numerical Differentiation)

In Chapter 8 we discussed numerical differentiation of a function based on divided differences. We saw that this was an ill-posed problem using floating-point arithmetic. In particular, smaller mesh sizes can lead to *catastrophic cancellation* (when two almost equal numbers are subtracted from each other) due to the finite precision of floating-point numbers. One possible solution is to use multiple precision data types.

In this exercise we investigate numerical differentiation of a function f by approximating it by a cubic spline and using formulae (13.38) and (13.37) as the surrogates for the first and second derivatives of the function, f, respectively.

Answer the following questions:

a) Compare the accuracy and applicability of this approach compared with the ill-posed divided differences that we use to approximate derivatives in Chapter 8.

b) Consider computing the price, delta and gamma of plain call and put options (analytical solutions are known; see Haug, 2007, for example). There are six options in total. Choose one set of three (for example, a call option price with its delta and gamma). Approximate the option price on a given interval, for example (0, 100).

c) Compare the exact solution for the option price with the value delivered by the cubic spline.

d) Compare the exact solution for the option delta and gamma with the values delivered by the cubic spline. For completeness, compute the delta and gamma using divided differences.

e) We focus on computing delta. Compare the relative run-time efficiency when computing delta using the exact form versus formula (13.38) as implemented in the class `CubicSplineInterpolator`.

In all cases, try to apply the utilities from Section 13.6.

4. (The Saul'yev ADE Method)

Implement this method for the heat equation as a variation of the discussion of the Barakat and Clark method in Section 13.4. In this case we use a single array to capture the approximate solution at each time level. The array is computed at uneven steps ($n = 1, 3, 5, \ldots$) using equation (13.29) and at even steps ($n = 0, 2, 4, \ldots$) using equation (13.30). Use a single array for both the upward and downward sweeps. It will always hold the approximate solution at all time levels up to and including expiration.

Compare the accuracy and run-time efficiency of this method with those of the Barakat and Clark ADE method and the Crank–Nicolson scheme.

5. (Cubic Spline Overshoots)

Consider the test data $x = \{1, 2, 3, 3.1, 5.1, 6, 7, 8\}$ and $y = \{1.8, 1.9, 1.7, 1.1, 1.1, 1.7, 1.4, 1.9\}$. Apply cubic spline interpolation to this data set using end/boundary condition types as discussed in this chapter (see equation (13.33)). Determine those subintervals in [1, 8] where negative values are produced. This is obviously an undesirable situation. In particular, what is the interpolated value when $x = 4$?

Carry out the same experiment with the data:

```
std::vector<double> x = { 0,10,30.50,70,90,100 };
std::vector<double> y = { 30,130,150,150,170,220,320};
```

6. (Generalisation of Code)

Generalise the use cases in Section 13.6 to support a range of vector types (for example, Boost vectors and home-grown vectors) as well as norms other than the max norm (for example, the Euclidean norm for vectors).

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 14

Data Visualisation in Excel

14.1 INTRODUCTION AND OBJECTIVES

At some stage in software development we need to have tools to determine the accuracy of numerical methods. These tools help us visualise data and results in some way. To this end, we have chosen to use a software library that the author developed to display array and matrix data in an Excel sheet. Our goal is to present data in Excel rather than using Excel as an input device (although this is possible by the use of *Excel addins* whose discussion is outside the scope of this book).

The Excel driver has been used for a number of years. In the current book the code has been upgraded, documented and improved. We have replaced legacy code by new code in C++11. Only one header file needs to be included. The driver can be used in a number of ways to promote developer productivity:

- Creating charts for presentations in articles and reports.
- Importing data into Excel for processing by Excel addins in C#, C++ or VBA. In this case we are using Excel as a kind of database.
- Quick testing and debugging: having developed a software prototype we can see if it is accurate by displaying its output in Excel.
- Experimenting with numerical algorithms: in this case we determine the effect of modifying certain parameters by displaying the results in Excel. For example, we may wish to examine what happens when we apply the explicit Euler scheme to the CIR interest-rate model when the *Feller condition* is not satisfied (Cox and Ross, 1976). See Exercises 4 and 5 of this chapter for some scenarios.

We shall use the Excel driver in later chapters when presenting output from numerical schemes.

14.2 THE STRUCTURE OF EXCEL-RELATED OBJECTS

We give a global overview of the objects in Excel (the *Excel Object model*) that can be accessed from the Excel driver software. Understanding these objects will help if you decide to examine the corresponding C++ source code. We mention that there is very little documentation and

most of our insights were based on reverse engineering of the code and documentation in VBA and C#. In all cases, the Excel objects use COM (*Component Object Model*) and they define interfaces that can be accessed programmatically. When writing code we note that Microsoft *IntelliSense* is very useful because it allows us to see the members of a given object. We give an introduction to COM in an appendix (Section 14.9) to this chapter.

We now discuss the most important Excel objects.

1. Application

   This object represents the Excel application itself. It is the *entry point* to the running application and related user objects. It contains application-wide settings as well as properties that return top-level objects.
2. Workbook

   This represents a single workbook in the Excel application. A *workbook* is a member of the *Workbooks* collection, the latter being a property of the Excel application. We note that this property does not include open add-ins which are a special kind of hidden workbook. A workbook has a number of properties:
   a) *Workbooks* property: we return a workbook by using a workbook name or index number.
   b) *ActiveWorkbook* property: this returns the workbook that is currently active.
   c) *ThisWorkbook* property: this is the workbook where the code is running. Normally this is the same as the active workbook.
3. Worksheet

   This is a member of the *Worksheets* collection that contains all the Worksheet objects in a workbook. A workbook consists of worksheets. The following properties for returning a Worksheet object are:
   a) *Worksheets* property: we return a worksheet by using a worksheet name or index number. The worksheet index number denotes the position of the worksheet in the *workbook's tab bar*.
   b) *ActiveSheet* property: this returns the worksheet that is currently active.

   We note that a Worksheet object is also a member of the *Sheets* collection.
4. Range

   This object represents a cell, a row, a column or a selection of cells containing one or more contiguous object blocks of a cell or a 3D range. It could even be a group of cells on multiple sheets. The *Range* object is the one that is the most used in Excel applications.
5. Chart

   This object represents a *chart* in a workbook. It can either be an embedded chart (in a *ChartObject*) or a separate chart sheet.

   For more information we refer the reader to the Microsoft online documentation. A comprehensive introduction is given in Bovey et al. (2009). However, C++ is not discussed in that book.

   The driver software that we introduce in this chapter is minimalist. It is possible to add new functionality to the driver by consulting the Microsoft online documentation with code examples in C# or VBA. This is enough in our experience to write the corresponding code in C++.

   We have created a class `ExcelDriver` and it contains member functions that delegate to the Excel COM-based objects.

14.3 SANITY CHECK: IS THE EXCEL INFRASTRUCTURE UP AND RUNNING?

In order to use the Excel driver, a version of Microsoft Office and Excel must be installed on your system. Furthermore, we define a file called `ExcelImports.cpp` that tells the compiler which version to use. A typical configuration is:

```
#import "C:\Program Files (x86)\Common Files
\Microsoft Shared\office14\mso.dll" \
rename("DocumentProperties", "DocumentPropertiesXL") \
rename("RGB", "RGBXL")

#import "C:\Program Files (x86)\Common Files
\Microsoft Shared\VBA\VBA6\vbe6ext.olb"

#import "C:\Program Files (x86)\Microsoft Office\Office14\EXCEL.EXE" \
rename("DialogBox", "DialogBoxXL") rename("RGB", "RGBXL") \
rename("DocumentProperties", "DocumentPropertiesXL") \
rename("ReplaceText", "ReplaceTextXL") \
rename("CopyFile", "CopyFileXL") no_dual_interfaces

// Excel 2016
   #import "C:\Program Files\Microsoft Office\root\VFS\ProgramFilesCommonX86
   \Microsoft Shared\OFFICE16\MSO.DLL"
   #import "C:\Program Files\Microsoft Office\root\VFS\ProgramFilesCommonX86
   \Microsoft Shared\VBA\VBA6\VBE6EXT.OLB"
   #import "C:\Program Files\Microsoft Office\root\Office16\EXCEL.EXE"
```

You need to check that your settings are correct before proceeding. You may need to modify this file to suit your version of Excel. Next, we have provided a test program to check if the interface to Excel is working:

```
// ExcelHelloWorld.cpp
//
// (C) Datasim Education BV 2012-2017
//
#include <string>
#include <iostream>
#include "ExcelImports.cpp"

#include "atlsafe.h"// Data types

// Extract a value from a cell
double GetDoubleFromCell(Excel::_WorksheetPtr sheet, CComBSTR cell)
{

   Excel::RangePtr range=sheet->GetRange(CComVariant(cell));
   _variant_t value=range->GetValue2();
   return value;
}
```

```
// Put a double value in an Excel cell
void SetCellValue(Excel::_WorksheetPtr sheet, CComBSTR cell, double value)
{
   Excel::RangePtr range=sheet->GetRange(CComVariant(cell));
   range->Value2=value;
}


int main()
{

   Excel::_ApplicationPtr xl;   // Pointer to Excel.

   try
   {
      // Initialize COM Runtime Libraries.
      CoInitialize(NULL);

      // Start excel application.
      xl.CreateInstance(L"Excel.Application");
      xl->Visible = VARIANT_TRUE;
      xl->Workbooks->Add((long) Excel::xlWorksheet);

      // Rename 1 to "Chart Data".
      Excel::_WorkbookPtr workbook = xl->ActiveWorkbook;

      // Language-independent
      Excel::_WorksheetPtr sheet = workbook->Worksheets->GetItem(1);
      sheet->Name = "Chart Data";

      double val = 2.0;

      // Display values in Excel
      SetCellValue(sheet, "B9", val);
      SetCellValue(sheet, "B10", val*3);

      // Get the parameters
      double T = GetDoubleFromCell(sheet, "B9");
      double r = GetDoubleFromCell(sheet, "B10");

      // Display values in Excel somwhere else
      SetCellValue(sheet, "C9", T);
      SetCellValue(sheet, "C10", r);
   }
   catch( _com_error & error )
   {
      bstr_t description = error.Description();
      if( !description )
      {
         description = error.ErrorMessage();
      }
```

```
        std::cout << std::string(description);
    }

    CoUninitialize();

    return 0;
}
```

You should run this program before proceeding. Please make sure that it compiles and runs without errors. Check your Excel imports file.

14.4 `ExcelDriver` AND MATRICES

Matrices are very important in numerical analysis and its applications. They are used as input to numerical processes as well as holding the results of computations. Some examples are:

- Two-dimensional grids of computed data, for example two-asset option prices; rows correspond to the first underlying value and columns correspond to the second underlying value.
- *Lookup tables* that are generated by discretising functions, for example a table of values of the non-central chi-squared distribution. The discrete values of the non-centrality and number of degrees of freedom parameters correspond to row and column values, respectively.
- Using Excel as a database to store data. For example, we could create data in C++ and then export it to Excel for further processing in VBA.

The current version of the software supports generic matrix types and we have tested it using Boost *uBLAS* matrices and a simple user-defined matrix class that is essentially a container for two-dimensional rectangular data (as already discussed in Chapter 10). Regarding vectors, we use `std::vector` in the current version of the software driver for convenience. The basic user-defined matrix class is:

```
#ifndef NestedMatrix_HPP
#define NestedMatrix_HPP

template <typename T> class NestedMatrix
{ // Simple class just to show Excel interoperability
private:
    std::vector<std::vector<T>> mat;

    std::size_t nr;
    std::size_t nc;
public:
    NestedMatrix(std::size_t rows, std::size_t cols) : nr(rows), nc(cols)
    {
        mat = std::vector<std::vector<T>>(nr);
        for (std::size_t i = 0; i < nr; ++i)
```

```
        {
            mat[i] = std::vector<T>(nc);
        }
    }

    NestedMatrix(const NestedMatrix<T>& m2) : nr(m2.nr), nc(m2.nc)
    {
        mat    = std::vector<std::vector<T>>(nr);

        for (std::size_t i = 0; i < nr; ++i)
        {
            mat[i] = m2.mat[i];
        }
    }

    T& operator ()(std::size_t row, std::size_t col)
    {
        return mat[row][col];
    }

    const T& operator ()(std::size_t row, std::size_t col) const
    {
        return mat[row][col];
    }

    NestedMatrix operator - (const NestedMatrix<T>& m2)
    {

        NestedMatrix<T> diff(nr, nc);

        for (std::size_t i = 0; i < nr; ++i)
        {
            for (std::size_t j = 0; i < nc; ++j)
            {
                diff(i, j) = (*this)(i, j) - m2(i, j);
            }
        }
    }

    std::size_t size1() const
    {
        return nr;
    }

    std::size_t size2() const
    {
        return nc;
    }
};

#endif
```

The class interface has been designed to conform to that of Boost *uBLAS*. Furthermore, we need two functions to convert in two directions between rows of a generic matrix and `std::vector` types:

```
template <typename Matrix, typename Vector>
   Vector CreateVector(const Matrix& mat, long row)
{

   Vector result(mat.size2());
   for (std::size_t i = 0; i < result.size(); ++i)
   {
      result[i] = mat(row, i);
   }

   return result;
}

template <typename Matrix, typename Vector>
   Matrix CreateMatrix(const Vector& vec)
{
   Matrix result(1, vec.size());
   for (std::size_t j = 0; j < result.size2(); ++j)
   {
      result(0,j) = vec[j];
   }
   return result;
}
```

There are two overloaded functions to present matrix data in Excel:

```
// Matrix visualisation
template <typename Matrix>
   void AddMatrix(const Matrix& matrix,
      const std::string& name = std::string("Matrix"),
      long row = 1, long col = 1);

template <typename Matrix>
   void AddMatrix(const Matrix& matrix,
                  const std::string& sheetName,
                  const std::list<std::string>& rowLabels,
                  const std::list<std::string>& columnLabels,
                  long row = 1, long col = 1);
```

Both functions print the matrix data in Excel. The second function annotates the rows and columns of the matrix with user-friendly labels. This option can be useful when inspecting output and when we wish to use label values as the variables of interest instead of using numerical indexes.

We now discuss some examples. For completeness, the classes tested are as follows:

```
#include <boost/numeric/ublas/matrix.hpp>
#include "NestedMatrix.hpp"
```

```
using NumericMatrix = boost::numeric::ublas::matrix<double>;
// using NumericMatrix = NestedMatrix<double>;
using Vector = std::vector<double>;
```

You can add your own favourite matrix class to this list and the code will run as long as the class conforms to the above matrix interface. The vector class used is hard coded. Other kinds of vectors are not supported.

14.4.1 Displaying a Matrix

We first take the simplest case of creating a matrix, setting up the input parameters and then displaying the matrix starting at given row and column positions in the upper left-hand corner. We construct the matrix as follows:

```
std::size_t N = 10; std::size_t M = 6; // rows and columns
NumericMatrix matrix(N + 1, M + 1);
for (std::size_t i = 0; i < matrix.size1(); ++i)
{
   for (std::size_t j = 0; j < matrix.size2(); ++j)
   {
      matrix(i, j) = static_cast<double>(i + j);
   }
}
```

We now start Excel and display the matrix using the following code:

```
// Start Excel
ExcelDriver& excel = ExcelDriver::Instance();

// Call Excel print function
std::string sheetName("Test Case 101 Matrix");
long row = 4; long col = 2;
excel.AddMatrix<NumericMatrix>(matrix, sheetName, row, col);
```

The output is shown in Figure 14.1.

0	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12
7	8	9	10	11	12	13
8	9	10	11	12	13	14
9	10	11	12	13	14	15
10	11	12	13	14	15	16

FIGURE 14.1 Test case 101 matrix

We applied the *Singleton* design pattern (GOF, 1995) to create a global instance of the Excel driver in the above code. It is also possible to create an instance using the following code:

```
ExcelDriver xl; xl.MakeVisible(true);
```

By default, Excel is started in invisible mode. Thus, you may call this latter method if you wish to see data on your screen.

14.4.2 Displaying a Matrix with Labels

In some cases we may wish to annotate the rows and columns of a matrix before presenting it in Excel. This feature promotes the readability and understandability of the displayed data. In general, the label values tell us more about the meaning of the data. This feature is useful when creating lookup tables, for example.

We take an example. First, we create row and column labels:

```
// Labels for rows and columns of the Excel matrix.
// Only labelled values are printed!!
std::list<std::string> rowLabels{"A","B","C","D","E", "F","K","L"};
std::list<std::string> colLabels{"C1", "C2", "C3", "C4","C5"};

std::string sheetName2("Matrix Labels Case");

ExcelDriver& excel = ExcelDriver::Instance();
excel.MakeVisible(true);           // Default is INVISIBLE!
```

We now present the annotated matrix using the same matrix data as in Figure 14.1 using the following code:

```
long rowPos = 4; long colPos = 3;
excel.AddMatrix<NumericMatrix>(matrix, sheetName2, rowLabels,
colLabels, rowPos, colPos);
```

The output is shown in Figure 14.2.

	C1	C2	C3	C4	C5		
A	0	1	2	3	4	5	6
B	1	2	3	4	5	6	7
C	2	3	4	5	6	7	8
D	3	4	5	6	7	8	9
E	4	5	6	7	8	9	10
F	5	6	7	8	9	10	11
K	6	7	8	9	10	11	12
L	7	8	9	10	11	12	13

FIGURE 14.2 Test case 101 matrix with annotation

We stress that only labelled rows are displayed in the current version of the software. In most cases, we would ensure that the number of labels should be the same as the corresponding matrix dimensions in order to view all the data. We thus see that not all rows from Figure 14.1 are visible in Figure 14.2. You need to be aware of this fact.

14.4.3 Lookup Tables, Continuous and Discrete Functions

A common use case is when we discretise real-valued functions of arity one and two. Focusing on the latter case we create two discrete mesh arrays and we compute the value of a function at these points. These values will be the values in the matrix. As an example, we discretise the bivariate normal probability density function defined in the following way for convenience (the correlation variable is global):

```
double rho = 0.5;
double NormalPdf2d(double x, double y)
{ // Bivariate normal density function

      double fac= 1.0/(2.0 * 3.14159265359 * std::sqrt(1.0 - rho*rho));
      double t = x*x - 2.0*rho*x*y + y*y;
      t /= 2.0 * (1.0 - rho*rho);

      return fac * std::exp(-t);
}
```

In order to discretise this function, we use the following utility function:

```
template <typename Matrix>
   Matrix CreateDiscreteFunction2d(const std::vector<double>& x ,
         const std::vector<double>& y,
         const std::function
                     <double (double x, double y)>& f)
{ // Create a discrete function from a continuous function m = f(x,y)

   std::size_t nr = x.size();
   std::size_t nc = y.size();

   Matrix m(nr,nc);

   for (std::size_t i = 0; i < nr; ++i)
   {
      for (std::size_t j = 0; j < nc; ++j)
      {
         m(i, j) = f(x[i],y[j]);
      }
   }

   return m;
}
```

This code can be parallelised, as discussed in Chapters 28 to 30.

Next, we define mesh points in the *x* and *y* directions by calling a mesh generation function `CreateMesh` (we show the code at the end of Section 14.4.3) as follows:

```
// Using mapping for continuous space to discrete space
std::size_t N = 20; std::size_t M = 10;
auto x = CreateMesh(N, -4.0, 4.0);
auto y = CreateMesh(M, -4.0, 4.0);

NumericMatrix matrix = DiscreteNormalPdf2d<NumericMatrix>(x, y);
```

where:

```
template <typename Matrix>
   Matrix DiscreteNormalPdf2d(const std::vector<double>& x,
                              const std::vector<double>& y)
{
      return CreateDiscreteFunction2d<Matrix>(x, y, NormalPdf2d);
}
```

We now present the matrix as follows:

```
//   ExcelDriver& excel = ExcelDriver::Instance();
std::string sheetName("Bivariate Normal pdf");
long row = 1; long col = 1;
excel.AddMatrix<NumericMatrix>(matrix, sheetName, row, col);
```

The output is shown in Figure 14.3.

This code can be generalised to work with arbitrary functions of arity one and two. See Exercise 1 below.

In the interest of completeness, as promised, we show the basic code that creates mesh points on an interval:

```
std::vector<double> CreateMesh(std::size_t n, double a, double b)
{ // Create a mesh of size n+1 on closed interval [a,b]
```

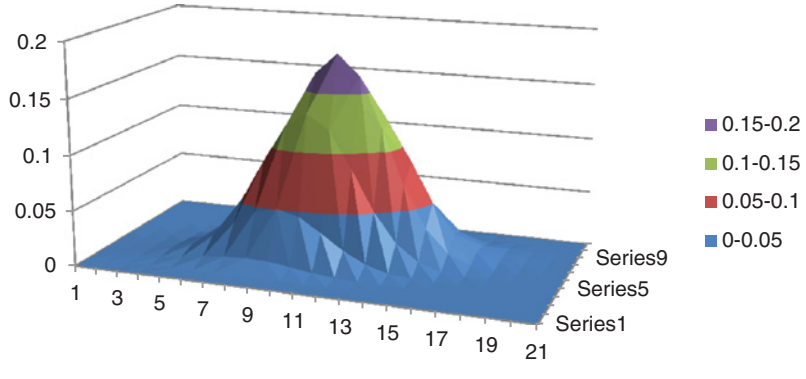


FIGURE 14.3 Bivariate normal PDF

```
        std::vector<double> x(n + 1);
        x[0] = a; x[x.size()-1] = b;

        double h = (b - a) / static_cast<double>(n);
        for (std::size_t j = 1; j < x.size() - 1; ++j)
        {
           x[j] = x[j - 1] + h;
        }

        return x;
}
```

14.5 ExcelDriver AND VECTORS

We see vectors as special cases of matrices in this context; a vector is a matrix with one row. The driver code converts a vector to a matrix and then calls the corresponding function that displays matrices. The corresponding interface is:

```
// Vector visualisation as numbers
template <typename Matrix>
   void AddVector(const std::vector<double>& vec,
                  const std::string& sheetName = std::string("Vector"),
                  long row = 1, long col = 1);

template <typename Matrix>
   void AddVector(const std::vector<double>& vec,
                  const std::string& sheetName,
                  const std::string& rowLabel,
                  const std::list<std::string>& columnLabels,
                  long row = 1, long col = 1);
```

This code is a special case of the discussion in Section 14.4 and for this reason we do not discuss it in detail. For completeness, we give an example:

```
std::string rowLabel("row1");
std::list<std::string> colLabels{"C1", "C2", "C3", "C4","C5"};

Vector myVector(colLabels.size());
for (std::size_t i = 0; i < myVector.size(); ++i)
{

   myVector[i] = static_cast<double>(i);
}


long row = 4; long col = 3;
excel.AddVector<NumericMatrix>(myVector, sheetName, rowLabel,
colLabels, row, col);
```

A possibly more interesting use case is when we wish to display one or more vectors as line graphs as a way to visualise them. This is the subject of the next section.

14.5.1 Single and Multiple Curves

We have two functions to display vectors. The first function displays a list of vectors while the second function displays a single vector. The interfaces are:

```
// Create chart with a number of functions. The arguments are:
// x:            std::vector<double> with input values
// labels:       labels for output values
// vectorList:   list of vectors with output values.
// chartTitle:   title of chart
// xTitle:       label of x axis
// yTitle:       label of y axis
void CreateChart(const std::vector<double> & x,
                 const std::list<std::string> & labels,
                 const std::list<std::vector<double>> & vectorList,
                 const std::string& chartTitle,
                 const std::string& xTitle = "X",
                 const std::string& yTitle = "Y");

// Create chart with a number of functions. The arguments are:
// x:            std::vector<double> with input values
// y:            std::vector<double> with output values.
// chartTitle:   title of chart
// xTitle:       label of x axis
// yTitle:       label of y axis

void CreateChart(const std::vector<double> & x,
                 const std::vector<double>& y,
                 const std::string& chartTitle,
                 const std::string& xTitle = "X",
                 const std::string& yTitle = "Y");
```

Using these functions is similar to the approach taken in Section 14.4. We initially consider the case of the function $f(x) = x^2$. Note that it is convenient to define it as a stored lambda function `fun`:

```
long N = 40;

// Create abscissa x array
double A = 0.0; double B = 10.0;
auto x = CreateMesh(N, A,B);

auto fun = [](double x) { return x*x; };
auto vec1 = CreateDiscreteFunction< std::vector<double>>(x, fun);

ExcelDriver xl; xl.MakeVisible(true);
xl.CreateChart(x, vec1, "Test 101 curve);
```

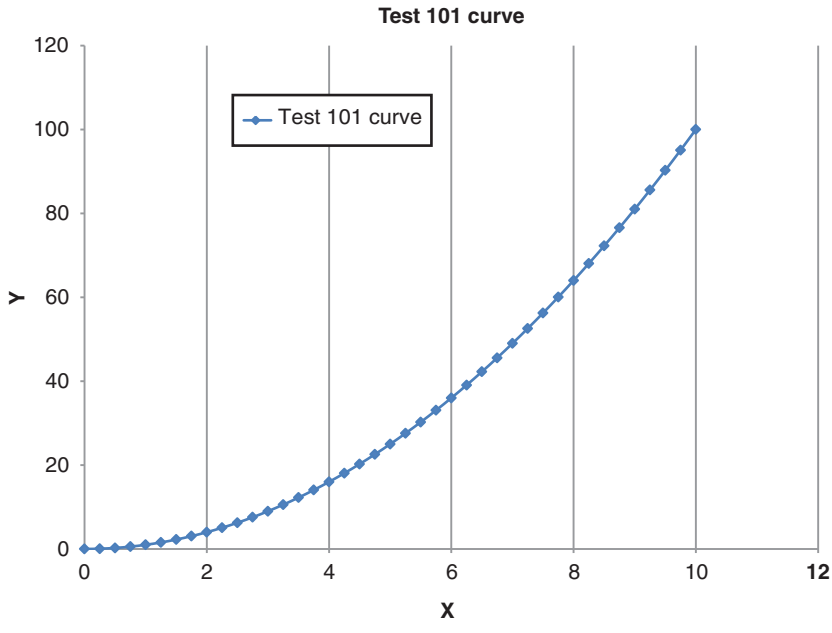


FIGURE 14.4 Single curve

The output is shown in Figure 14.4.

In the case of displaying multiple curves in an Excel sheet, we first define a number of functions and we then convert each of them to a vector (discrete function). Finally, we append each vector to a list and we call the appropriate driver function:

```
long N = 40;

// Create abscissa x array
double A = 0.0; double B = 3.0; // Interval
auto x = CreateMesh(N, A, B);

auto fun = [](double x) { return std::log(x+ 0.01); };
auto fun2 = [](double x) { return x*x; };
auto fun3= [](double x) { return x*x*x; };
auto fun4 = [](double x) { return std::exp(x); };
auto fun5 = [](double x) { return x; };

auto vec1 = CreateDiscreteFunction< std::vector<double>>(x, fun);
auto vec2 = CreateDiscreteFunction< std::vector<double>>(x, fun2);
auto vec3 = CreateDiscreteFunction< std::vector<double>>(x, fun3);
auto vec4 = CreateDiscreteFunction< std::vector<double>>(x, fun4);
auto vec5 = CreateDiscreteFunction< std::vector<double>>(x, fun5);
// Now Excel output in one sheet for comparison purposes

// Names of each vector
std::list<std::string> labels
{ "log(x+0.01)", "x^2", "x^3", "exp(x)","x" };
```

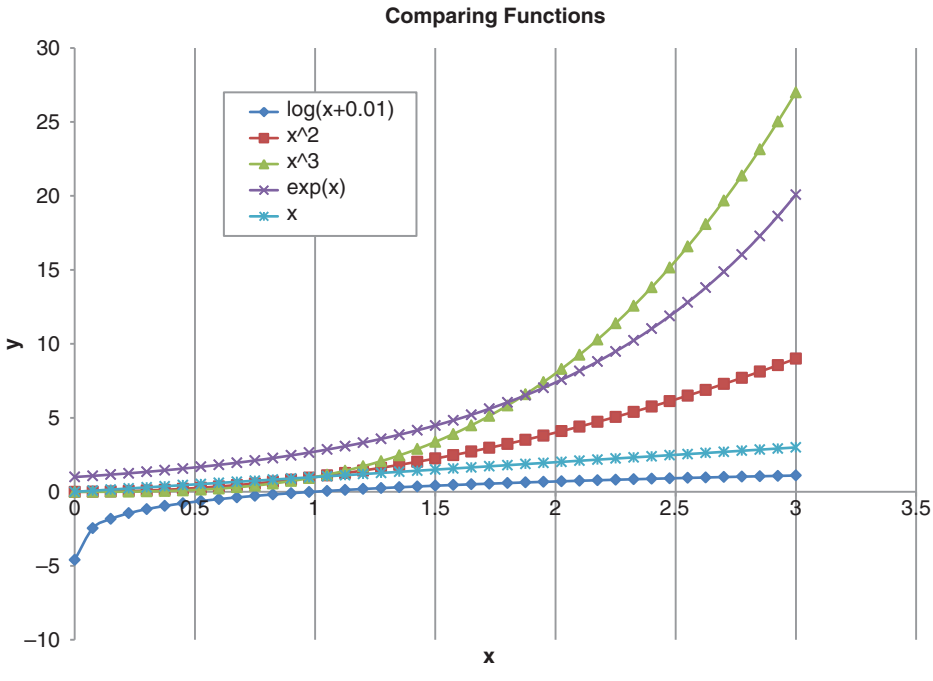


FIGURE 14.5 Multiple curves

```
// The list of Y values using uniform initialisation
std::list<std::vector<double> > curves{ vec1, vec2, vec3, vec4, vec5 };

std::cout << "Data has been created\n";

ExcelDriver xl; xl.MakeVisible(true);
xl.CreateChart(x, labels, curves, "Comparing Functions", "x", "y");
```

The output is shown in Figure 14.5.

For completeness, we show the code to create vectors and matrices of values from a function:

```
template <typename Vector>
   Vector CreateDiscreteFunction(const std::vector<double>& x,
                                 const std::function<double(double)>& f)
{ // Create a discrete function from a continuous function y = f(x)

   Vector y(x.size());

   for (std::size_t j = 0; j < y.size(); ++j)
   {
      y[j] = f(x[j]);
   }
```

```
   return y;
}

template <typename Matrix>
Matrix CreateDiscreteFunction2d(const std::vector<double>& x ,
                  const std::vector<double>& y,
                  const std::function
                              <double (double x, double y)>& f)
{ // Create a discrete function from a continuous function m = f(x,y)

   std::size_t nr = x.size();
   std::size_t nc = y.size();

   Matrix m(nr,nc);

   for (std::size_t i = 0; i < nr; ++i)
   {
      for (std::size_t j = 0; j < nc; ++j)
      {
         m(i, j) = f(x[i],y[j]);
      }
   }

   return m;
}
```

14.6 PATH GENERATION FOR STOCHASTIC DIFFERENTIAL EQUATIONS

The generic examples in the preceding sections were chosen to show how the Excel driver works without becoming embroiled in software design decisions or mathematically advanced discussions. In this section we take examples that are related to computational finance. To this end, we simulate the paths of quantities that are defined by SDEs. In most cases analytical solutions do not exist and we must resort to numerical approximations, in this case the finite difference method.

We now discuss some schemes to calculate the path of the following nonlinear *autonomous* SDE:

$$\begin{aligned} &dX(t) = \mu(X(t))dt + \sigma(X(t))dW(t) \quad 0 < t \leq T \\ &X(0) = A. \end{aligned} \tag{14.1}$$

Some finite difference schemes are:

- Explicit Euler:

$$\begin{aligned} &X_{n+1} = X_n + \mu_n \Delta t + \sigma_n \Delta W_n \\ &\text{where:} \\ &\mu_n = \mu(X_n) \;\; \sigma_n = \sigma(X_n). \end{aligned} \tag{14.2}$$

- Semi-implicit Euler:

$$\begin{aligned} &X_{n+1} = X_n + [\alpha\mu_{n+1} + (1-\alpha)\mu_n]\Delta t + \sigma_n \Delta W_n \\ &\text{with special cases:} \\ &\alpha = \tfrac{1}{2} \text{ (Trapezoidal)},\ \alpha = 1 \text{ (Backward Euler)}. \end{aligned} \tag{14.3}$$

- Heun:

$$\begin{aligned} &X_{n+1} = X_n + \tfrac{1}{2}[F_1 + F_2]\Delta t + \tfrac{1}{2}[G_1 + G_2]\Delta W_n \\ &\text{where:} \\ &F(x) \equiv \mu(x) - \tfrac{1}{2}\sigma'(x)\sigma(x) \text{ where } \sigma'(x) \equiv \tfrac{d\sigma}{dx} \\ &F_1 = F(X_n), \quad G_1 = \sigma(X_n) \\ &F_2 = F(X_n + F_1\Delta t + G_1\Delta W_n) \\ &G_2 = \sigma(X_n + F_1\Delta t + G_1\Delta W_n). \end{aligned} \tag{14.4}$$

- Milstein:

$$X_{n+1} = X_n + \mu_n\Delta t + \sigma_n\Delta W_n + \tfrac{1}{2}[\sigma'\sigma]_n((\Delta W_n)^2 - \Delta t). \tag{14.5}$$

- Derivative free

$$\begin{aligned} &X_{n+1} = X_n + F_1\Delta t + G_1\Delta W_n + [G_2 - G_1]\Delta t^{-1/2}\tfrac{(\Delta W_n)^2 - \Delta t}{2} \\ &\text{where:} \\ &F_1 = \mu(X_n), \quad G_1 = \sigma(X_n)\ G_2 = \sigma(X_n + G_1\Delta t^{1/2}). \end{aligned} \tag{14.6}$$

- First-order Runge–Kutta with Ito coefficient (FRKI):

$$\begin{aligned} &X_{n+1} = X_n + F_1\Delta t + G_2\Delta W_n + [G_2 - G_1]\Delta t^{1/2} \\ &\text{where:} \\ &F_1 = \mu(X_n), \quad G_1 = \sigma(X_n)\ G_2 = \sigma\left(X_n + \tfrac{G_1(\Delta W_n - \Delta t^{1/2})}{2}\right). \end{aligned} \tag{14.7}$$

In this section we extend the C++ code of Section 6.8 in Chapter 6 that models SDEs. In particular, we simulate the paths of SDEs using the above finite difference methods. We design part of the system context diagram of Figure 7.1 (and related topics in Chapter 9).

The main systems are:

- S1: The option (derivative) input data.
- S2: Modelling one-factor SDEs.
- S3: Modelling finite difference schemes.
- S4: Presenting SDE paths in Excel.
- S5: A *Builder*/*Factory* system to configure and initialise the systems S1 to S4.

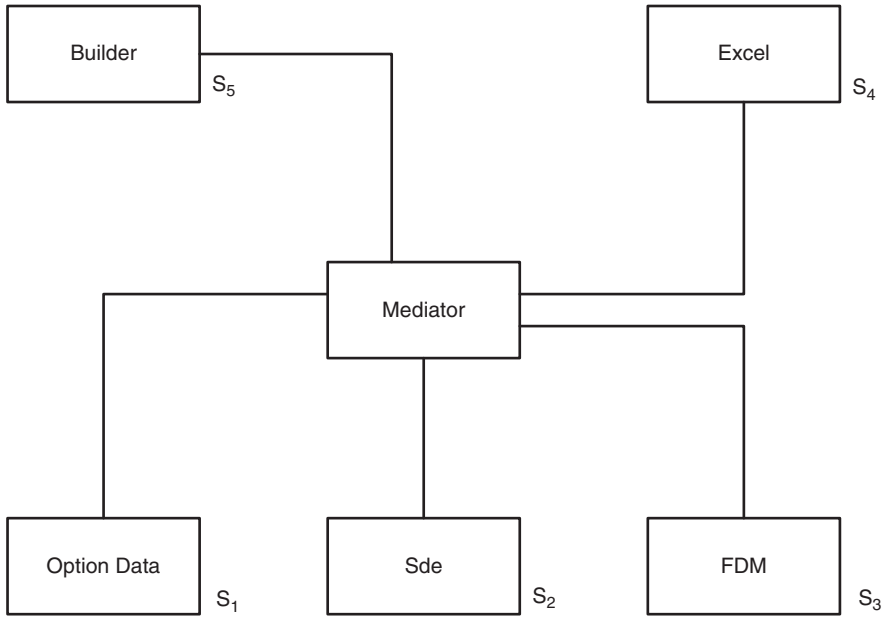


FIGURE 14.6 Context diagram for path evolver

We show how we have designed this system using modern C++ language features based on a multiparadigm approach. It is part of a larger system to compute one-factor option prices using the Monte Carlo method in Chapters 31 and 32. We give a preview of upcoming design and we display stochastic paths in a user-friendly manner.

14.6.1 The Main Classes

The design of the current problem is shown in Figure 14.6 and it is a special case of the context diagram in Figure 7.1 and the more general diagrams in Chapter 9. Furthermore, we have added a new data member to the SDE class that represents the upper limit of the time interval in which the SDE is defined.

```
// Functions of arity 2 (two input arguments)
template <typename T>
   using FunctionType = std::function<T (const T& arg1, const T& arg2)>;


// Interface to simulate any SDE with drift/diffusion
// Use a tuple to aggregate two functions into what is similar to an
// interface in C# or Java.
template <typename T>
   using ISde = std::tuple<FunctionType<T>, FunctionType<T>>;

template <typename T = double> class Sde
{
```

```
private:
   FunctionType<T> dr_;
   FunctionType<T> diff_;

public:
   T ic;   // Initial condition
   T B;   // SDE on interval [0,B]
public:
   Sde() = delete;
   Sde(const FunctionType<T>& drift, const FunctionType<T>& diffusion,
      const T& initialcCondition, const T& expiration)
      : dr_(drift), diff_(diffusion), ic(initialcCondition),
      B(expiration) {}
   Sde(const Sde<T>& sde2, const T& initialcCondition, const
   T& expiration)
      : dr_(sde2.dr_), diff_(sde2.diff_), ic(initialcCondition),
      B(expiration) {}
   Sde(const ISde<T>& functions, const T& initialcCondition, const
   T& expiration)
      : dr_(std::get<0>(functions)), diff_(std::get<1>(functions)),
            ic(initialcCondition), B(expiration) {}

   T drift(const T& S, const T& t) const { return dr_(S, t); }
   T diffusion(const T& S, const T& t) const { return diff_(S, t); }
};
```

We now discuss the design of the finite difference schemes to approximate the solution of an SDE. We wish to write as little code as possible and the options open to us are:

- C1: Using dynamic/subtype polymorphism and class hierarchies (as in Duffy and Kienitz, 2009).
- C2: Using the *Curiously Recurring Template Pattern* (CRTP) and static polymorphism with class hierarchies.
- C3: Creating a single class with universal function wrappers as data members.

In this chapter we have chosen option C2 for reasons of efficiency (all functions are known at compile-time) and maintainability (all data and functions are encapsulated in a set of related classes). In general, the FDM class needs to be acquainted with the SDE class. The FDM class models *one-step time-marching schemes*. Adhering to the SRP, we can say that such schemes *advance* the approximate solutions from a given time level n to time level $n + 1$.

The base class is:

```
template <typename Derived, typename T> class Fdm
{ // Using CRTP pattern

//private:
protected:
```

```
    std::normal_distribution<T> dist;
    std::default_random_engine eng;
    std::shared_ptr<Sde<T>> sde;
public:
    Fdm(const std::shared_ptr<Sde<T>>& oneFactorProcess,
        const::normal_distribution<T>& normalDist,
        const::default_random_engine& engine)
    : sde(oneFactorProcess), dist(normalDist), eng(engine){}

    // Compute x(t_n + dt) in terms of x(t_n)
    T advance(T xn, double tn, double dt)
    {
       return static_cast<Derived*> (this)->advance(xn, tn, dt);
    }

    T generateRN()
    {
       return dist(eng);
    }
};
```

Notice that we have used the C++ `<random>` library's functions to generate uniform variates (this can be made more generic but not just yet). Specific finite difference schemes implement the function `advance()` in their own way, for example explicit Euler and Heun methods:

```
template <typename T>
   class FdmEuler : public Fdm<FdmEuler<T>, T>
{ // Using CRTP pattern

   private:

   public:
      FdmEuler(const std::shared_ptr<Sde<T>>& oneFactorProcess,
               const std::normal_distribution<T>& normalDist,
               const std::default_random_engine& engine)
            : Fdm(oneFactorProcess, normalDist, engine)
      {}

      // Compute x(t_n + dt) in terms of x(t_n)
      T advance(T xn, T tn, T dt)
      {
         T normalVar = generateRN();
         return xn + sde->drift(xn, tn)*dt
                   +sde->diffusion(xn, tn) * std::sqrt(dt) * normalVar;
      }
};

template <typename T>
   class FdmHeun : public Fdm<FdmHeun<T>, T>
{ // Using CRTP pattern
```

```
    private:

    public:
       FdmHeun(const std::shared_ptr<Sde<T>>& oneFactorProcess,
                const std::normal_distribution<T>& normalDist,
                const std::default_random_engine& engine)
             : Fdm(oneFactorProcess, normalDist, engine)
       {}

       // Compute x(t_n + dt) in terms of x(t_n)
       T advance(T xn, T tn, T dt)
       {
          T a = sde->drift(xn, tn);
          T b = sde->diffusion(xn, tn);
          T normalVar = generateRN();
          T suppValue = xn + a * dt + b * std::sqrt(dt) * normalVar;

          return xn + 0.5 * (sde->drift(suppValue, tn) + a) * dt
                    + 0.5 * (sde->diffusion(suppValue, tn) + b)
                    * std::sqrt(dt) * normalVar;
       }
    };
```

We have built a *factory class* that creates an SDE (the choices are GBM, CEV and CIR) based on user choice:

```
namespace SdeFactory
{
   // Specific SDEs
   template <typename T>
      std::shared_ptr<Sde<T>> GbmSde(const OptionData& data, T S0)
   {
      T r = data.r; T sig = data.sig; T B = data.T;
      auto drift = [=](T t, T S) { return r * S; };
      auto diffusion = [=](T t, T S) { return sig * S; };

      ISde<T> Functions = std::make_tuple(drift, diffusion);

      // Create Sde
      return std::shared_ptr<Sde<T>>(new Sde<T>(Functions, S0, B));
   }

   template <typename T>
            std::shared_ptr<Sde<T>> CevSde(const OptionData&
            data, T S0)
   {
      T r = data.r; T sig = data.sig; T B = data.T;
      T beta = 0.5; // Hard-coded
      auto drift = [=](T t, T S) { return r * std::pow(S, beta); };
      auto diffusion = [=](T t, T S) { return sig * S; };

      ISde<T> Functions = std::make_tuple(drift, diffusion);
```

```
        // Create Sde
        return std::shared_ptr<Sde<T>>(new Sde<T>(Functions, S0, B));
    }

    template <typename T>
        std::shared_ptr<Sde<T>> CIRSde(T alpha, T b, T sigma, T S0, T B)
    {

        auto drift = [=](T t, T X) { return alpha*(b-X); };
        auto diffusion = [=](T t, T X) { return sigma * std::sqrt(X); };
        ISde<T> Functions = std::make_tuple(drift, diffusion);

        // Create Sde
        return std::shared_ptr<Sde<T>>(new Sde<T>(Functions, S0, B));
    }

    template <typename T>
        std::shared_ptr<Sde<T>> ChooseSde(const OptionData& data, T S0)
    { // Simple factory method

        std::cout << "1. GBM, 2. CEV, 3. CIR: ";
        int choice; std::cin >> choice;

        if (1 == choice)
        {
            return GbmSde<T>(data, S0);
        }
        if (2 == choice)
        {
            return CevSde<T>(data, S0);
        }

        if (3 == choice)
        {
            // Feller condition => 2 a b >= sigma^2
            double alpha = 0.01;
            double b = 7.0;
            double sigma = 0.9;
            double B = 1.0; // Upper limit of interval
            double X0 = 0.4;

            return CIRSde<T>(alpha, b, sigma, X0, B);

        }
        else
        {
            return GbmSde<T>(data, S0);

        }
    }

}
```

We note that we have not created factories for the FDM classes in this version in order to avoid cognitive overload. In any case, it would have a structure similar to that in the above SDE case. Having created the SDE and FDM classes we now design the *mediator* class in Figure 14.6 that connects the other components and constructs the simulated path.

The code is:

```
// Creating a path of an SDE approximated by FDM
template <typename Derived, typename T>
      std::vector<T> Path(const Sde<T>& sde,
                          Fdm<typename Derived, T>& fdm, long NT)
{

   std::vector<T> result(NT + 1);

   result[0] = sde.ic;

   double dt = sde.B / static_cast<T>(NT);
   double tn = dt;

   for (std::size_t n = 1; n < result.size(); ++n)
   {
      result[n] = fdm.advance(result[n - 1], tn, dt);
      tn += dt;
   }

   return result;
}
```

Finally, we encapsulate all relevant model constants in a convenient struct called `Option Data`:

```
#include <algorithm> // for max()
#include <boost/parameter.hpp>

namespace OptionParams
{
   BOOST_PARAMETER_KEYWORD(Tag, strike)
   BOOST_PARAMETER_KEYWORD(Tag, expiration)
   BOOST_PARAMETER_KEYWORD(Tag, interestRate)
   BOOST_PARAMETER_KEYWORD(Tag, volatility)
   BOOST_PARAMETER_KEYWORD(Tag, dividend)
   BOOST_PARAMETER_KEYWORD(Tag, optionType)
}

// Encapsulate all data in one place
struct OptionData
{ // Option data + behaviour
```

```
    double K;
    double T;
    double r;
    double sig;

    // Extra data
    double D;      // dividend

    int type;      // 1 == call, -1 == put

    explicit constexpr OptionData(double strike, double expiration,
                                  double interestRate,
                                  double volatility, double dividend,
                                  int PC)
       : K(strike), T(expiration), r(interestRate),
            sig(volatility), D(dividend), type(PC)
    {}

    template <typename ArgPack> OptionData(const ArgPack& args)
    {
       K = args[OptionParams::strike];
       T = args[OptionParams::expiration];
       r = args[OptionParams::interestRate];
       sig = args[OptionParams::volatility];
       D = args[OptionParams::dividend];
       type = args[OptionParams::optionType];

       std::cout << "K " << K << ", T " << T << ",r " << r << std::endl;
       std::cout << "vol " << sig << ", div " << D << ",type "
            << type << std::endl;
    }

};
```

We can initialise the elements of the struct `OptionData` by calling its constructor (the position of `args` is important), by uniform initialisation or by using *named variables* as supported in the *Boost Parameter* library. In this last case the relative order of the input arguments is immaterial, as the following example shows:

```
//OptionData opt2 { 100.0, 1.0, 0.06, 0.2, 0.03, 1 };

// Init via named Boost variables
OptionData myOption((OptionParams::strike = 65.0,
                     OptionParams::expiration = 1.0,
                     OptionParams::volatility = 0.001,
                     OptionParams::dividend = 0.0,
                     OptionParams::optionType = -1,
                     OptionParams::interestRate = 0.1));
```

14.6.2 Testing the Design and Presentation in Excel

We now discuss the configuration and initialisation of the classes in Figure 14.6. The steps are easy to follow:

```
// A. Create basic input data
//OptionData(double strike, double expiration, double interestRate,
// double volatility, double dividend, int PC)
OptionData myOption { 100.0, 10.0, 0.1, 0.9, 0.03, 1 };
double S0 = 20.0;

// B. Get the SDE
// Using factories
std::shared_ptr<Sde<double>> sde = SdeFactory::ChooseSde<double>
(myOption, S0);

// C. Set up the FDM classes; 1 to N relationship between SDE and FDM
std::default_random_engine eng;

std::normal_distribution<double> nor(0.0, 1.0);
std::random_device rd;
eng.seed(rd());

FdmEuler<double> fdm(sde, nor, eng);
FdmHeun<double> fdm2(sde, nor, eng);

// D. Joining up SDE and FDM in Mediator class
long NT = 200;
auto vec = Path(*sde, fdm, NT);
auto vec2 = Path(*sde, fdm2, NT);

double A = 0.0; double B = myOption.T;
auto x = CreateMesh(NT+1, A, B);

// E. 'Report' system; display the graphs in Excel
ExcelDriver xl; xl.MakeVisible(true);

// E_1: Display each curve in its own sheet
xl.CreateChart(x, vec, "Euler method");
xl.CreateChart(x, vec2, "Heun method");

// Names of each vector
std::list<std::string> labels{ "Euler", "Heun"};

// The list of Y values
std::list<std::vector<double> > curves{ vec, vec2};

// E_2: Display all curves in one sheet
xl.CreateChart(x, labels, curves, "Comparing FDM ", "t", "S");
```

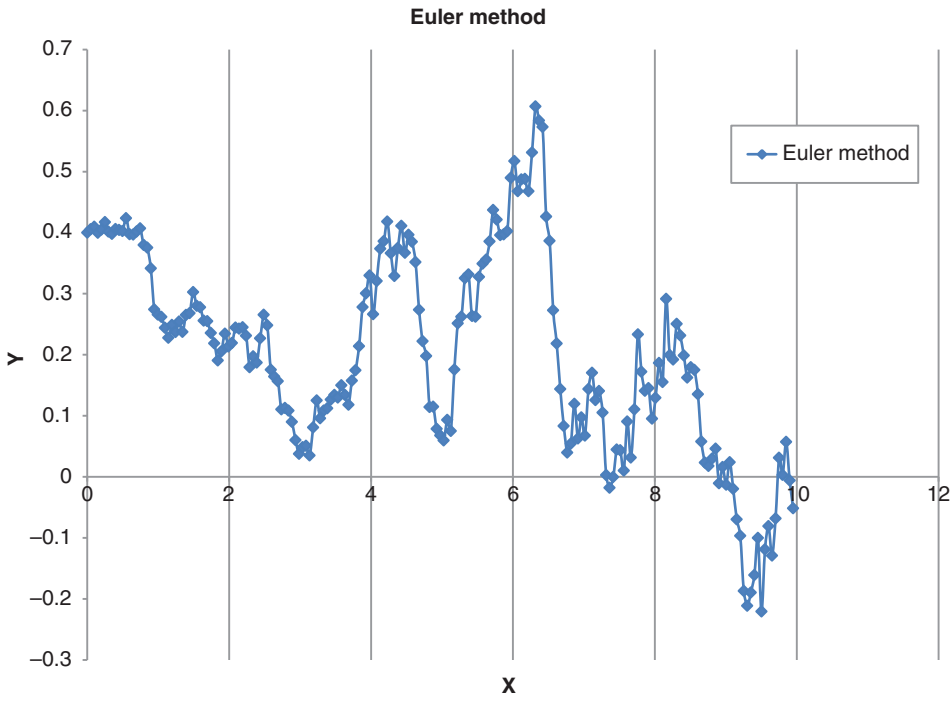


FIGURE 14.7 CIR path with Euler method

Typical output for the explicit Euler method is shown in Figure 14.7. In this case we simulate the short rate of the CIR model (*square-root diffusion process*) and we see (and it is well known) that the Euler method can produce negative values, which is incorrect. We discuss this issue in Exercise 4.

Finally, we show the code that implements the mediator component from Figure 14.6. In this version it is a free function but in larger and more complex applications it could be modelled as a *Whole–Part object* (POSA, 1996) to promote maintainability. It would have the following method:

```
//  Creating a path of an SDE approximated by FDM
template <typename Derived, typename T>
   std::vector<T> Path(const Sde<T>& sde, Fdm<typename Derived,
   T>& fdm, long NT)
{

   std::vector<T> result(NT + 1);

   result[0] = sde.ic;

   double dt = sde.B / static_cast<T>(NT);
   double tn = dt;

   for (std::size_t n = 1; n < result.size(); ++n)
```

```
    {
       result[n] = fdm.advance(result[n - 1], tn, dt);
       tn += dt;
    }

    return result;

}
```

14.7 SUMMARY AND CONCLUSIONS

In this chapter we introduced a software framework to display numeric data in Excel. We focused on visualising vectors, matrices and curves in the Excel spreadsheet program. We motivated the advantages and we gave a number of examples.

14.8 EXERCISES AND PROJECTS

1. (Extensions to the Excel Driver)

There are many ways to add new functionality to the Excel driver. In this exercise we focus on generalising the code. We wish to display functions of arity one and two. We hide low-level details such as the creation of mesh arrays and other supporting code. The desired functions have the following interfaces:

```
using Function1DType = std::function<double (double x)>;
using Function2DType = std::function<double(double x, vdouble y)>;

void printDiscreteFunctionValues(const Function1DType& f,
double A, double B,long nSteps,
                                 const std::string& title,
                                 const std::string& horizontal,
                                 const std::string& vertical,
                                 const std::string& legend);

void printDiscreteFunctionValues(const Function1DType& f,
                                 const std::vector<double>& mesh,
                                 const std::string& title,
                                 const std::string& horizontal,
                                 const std::string& vertical,
                                 const std::string& legend);

void printDiscreteFunctionValues(const Function2DType& f,
                                 double A, double B,long xSteps,
                                 double C, double D,long ySteps,
                                 const std::string& title,
                                 const std::string& horizontal,
                                 const std::string& vertical,
                                 const std::string& legend);
```

```
void printDiscreteFunctionValues(const Function2DType& f,
                                 const std::vector<double>& mesh,
                                 const std::vector<double>& mesh2,
                                 const std::string& title,
                                 const std::string& horizontal,
                                 const std::string& vertical,
                                 const std::string& legend);
```

Answer the following questions:

a) Write the code for the above functions based on existing code in the driver.

b) Test functions of arity one on Black–Scholes call option prices and the corresponding greeks:

$$\Delta_C = \frac{\partial C}{\partial S} = e^{(b-r)T}N(d_1)$$
$$\Gamma_C \equiv \frac{\partial^2 C}{\partial S^2} = \frac{\partial \Delta_C}{\partial S} = \frac{n(d_1)e^{(b-r)T}}{S\sigma\sqrt{T}}$$
$$Vega_C \equiv \frac{\partial C}{\partial \sigma} = S\sqrt{T}e^{(b-r)T}n(d_1)$$
$$\Theta_C \equiv -\frac{\partial C}{\partial T} = -\frac{S\sigma e^{(b-r)T}n(d_1)}{2\sqrt{T}} - (b-r)Se^{(b-r)T}N(d_1) - rKe^{-rT}N(d_2).$$

c) Test functions of arity two by displaying the probability density function of the *bivariate t-distribution* on a given interval (Nadarajah and Kotz, 2005):

$$f(x,\ y;\ \nu,\ \rho) = \frac{1}{2\pi\sqrt{1-\rho^2}}\left\{1 + \frac{x^2 - 2\rho xy + y^2}{\nu(1-\rho^2)}\right\}^{-(\nu+2)/2}.$$

We shall discuss this distribution in more detail in Chapter 16 when we discuss ways to compute the (cumulative) probability integral:

$$P(x,\ y;\ \nu,\ \rho) = \int_{-\infty}^{x}\int_{-\infty}^{y} f(x,\ y;\ \nu,\ \rho)dxdy.$$

2. (Finite Difference Schemes for the One-Dimensional Heat Equation)

An interesting and useful application of the Excel driver is in displaying the values of the approximate solutions (using FDM in this case) to the heat equation as discussed in Chapter 13. In particular, I personally have found it useful for the following cases:

- C1: Having written FD code, we wish to see if it is producing correct results. To this end, displaying the results in an easy format is a boon.
- C2: We would like to compare the accuracy of the ADE family of schemes with a baseline case such as the Crank–Nicolson method or an analytical solution of a PDE.
- C3: To display the accuracy of an approximate method as a function of some independent variable. An example is plotting option price in the binomial method as a function of the number of steps. The same exercise is applicable to the finite difference method.

Answer the following questions:

a) Use the two ADE schemes that we introduced in Chapter 13 to approximate the solution of the heat equation (see equation (13.14) in Chapter 13).

b) Compare the solutions in part a) with the exact solution (13.15) and the solution produced by the Crank–Nicolson method.
c) Display the four results from parts a) and b) as four curves in a single Excel sheet. Compare the results.
d) Experiment with various initial conditions in equation (13.14) and repeat steps a), b) and c).

3. (Cubic Spline Overshoots, Part II)

In Chapter 13 we gave a detailed introduction to cubic splines and some of their applications. In particular, Exercise 5 in Chapter 13 was concerned with a discussion on using cubic splines and possible overshoots than can occur with yield curve data. Answer the following questions:

a) Use the same data as in Exercise 5, Chapter 13 and display the interpolated cubic spline values in Excel.
b) Compare the results with those produced by linear interpolation (see equation (13.32)).
c) Display the first and second derivatives of the cubic spline from part a) based on the formulae (13.38) and (13.37), respectively.

4. (The Cox–Ingersoll–Ross (CIR) Model)

We examine the CIR interest-rate model (also known as *square-root diffusion*) defined as the following SDE with constant coefficients (Glasserman, 2004):

$$dr = a(b - r)dt + \sigma\sqrt{r}dW.$$

This is a model of the short rate and it is referred to as the CIR model. In this case the short rate $r(t)$ is pulled towards b at a speed controlled by a. Furthermore, if $r(0) > 0$ then $r(t)$ will never be negative and $r(t)$ remains strictly positive for all t if the following *Feller condition* is satisfied:

$$ab > \frac{1}{2}\sigma^2.$$

The objective of this exercise is to introduce this model into the framework in Figure 14.6 and gain an insight into its analytical and numerical properties.

Answer the following questions:

a) Create a class for the CIR SDE and integrate it into the framework.
b) The CIR SDE does not have an explicit solution. We then resort to numerical methods. Apply the explicit Euler and Heun methods for various time-step sizes and expirations.
c) Investigate the finite difference schemes when the Feller condition is not satisfied.
d) Experiment with the finite difference schemes and determine positivity for small and large values of drift, volatility and expiration parameters.

We remark that the *transition density* for the CIR process is known and it can be represented in terms of a non-central chi-squared distribution (Glasserman, 2004, p. 122).

5. (SDEs with Analytical Solutions)

In some cases (we might get lucky) it is possible to find analytical solutions to SDEs (Kloeden and Platen, 1995, pp. 118–126). Some examples of SDEs and their solutions are:

$$\begin{aligned} dX &= \tfrac{1}{2}a^2 X\,dt + a\,X\,dW \\ X(t) &= X_0 \exp(aW(t)). \end{aligned}$$

$$dX = aX\,dt + bX\,dW$$
$$X(t) = X_0 \exp((a - \tfrac{1}{2}b^2)t + bW(t)).$$
$$dX = \tfrac{1}{2}X\,dt + X\,dW$$
$$X(t) = X_0 \exp(W(t)).$$

Here $W(t)$ is a Wiener process.

Answer the following questions:

a) Integrate these equations into the framework and decide how to determine the difference between the explicit solution and the solution defined by a given finite difference scheme.

b) Experiment with the schemes by varying the number of time steps. Is there some way to determine what the order of convergence of a given scheme is?

c) We focus on the explicit Euler method. Produce a multi-curve graph in the style of Figure 14.5 in which each curve is a path produced by the Euler method with a given number of time steps, for example step sizes $NT = 10$, 20, 50, 100, 200, 300. Do you see monotonic convergence to the 'exact path' as NT increases?

6. (Log–Log and Semi-log Plots)

In some cases we may wish to transform data in some way before presenting it in Excel. In general, the *linear plot* between the horizontal x and vertical y axes can be transformed in two main ways:

- *Log–log plot*: uses a logarithmic scale for both the horizontal and vertical axes. This transformation is suitable for *monomials* (relationships of the form $y = ax^k$). These become straight lines in log space, as the following steps show:

$$\begin{cases} y = ax^k \\ \log y = k \log x + \log a \\ X = \log x,\ Y = \log y \\ \\ Y = mX + b. \end{cases}$$

We thus see that $m = k$ is the slope (gradient) of the straight line while $b = \log a$ is the intercept on the log y axis, so this graph can recognise relationships and estimate parameters (in this case the parameters a and k).

- *Semi-log plot*: in this case only one of the variables is log transformed. We distinguish between *log-lin* in which the y axis is transformed and *lin-log* in which the x axis is transformed. Equations of the form $y = \lambda a^{\gamma x}$ become straight lines when plotted semi-logarithmically, as the following steps show:

$$\log y = \log(\lambda a^{\gamma x}) = \log \lambda + \log(a^{\gamma x}) = \log \lambda + (\gamma \log a)x.$$

In general, any log base can be used but most applications use base 10, 2 or e.

Answer the following questions:

a) Determine how to introduce log transformations into the current Excel driver framework.

b) As test 101 case, we take the example $y = x^2/2$. Generate x and y array values on the interval [1,10]. Then perform a log–log transform on the arrays and plot the new arrays in Excel. Can you reproduce or 'see' the original parameters from the slope and intercept of the straight line?

7. (Extending the Framework in Figure 14.6, Brainstorming)

The solution to this exercise is discussed in Chapter 31 but it does no harm to address the problem as early as possible in the book. The focus is on extending the design in Figure 14.6 to create a small one-factor Monte Carlo option pricer.

Answer the following questions:

a) Consider the initial case of pricing one-factor call and put option prices. Which new systems and classes need to be added to the framework? A requirement is that the framework should compute option price and the standard error.

b) The mediator class needs to be extended because in its current form it computes a single path only. Using the Monte Carlo method entails that we create a large number of paths and that we use well-known averaging and discounting algorithms.

c) Test the code from parts a) and b) for call and put options and compare exact and approximate solutions with each other.

d) Extend the framework so that it can be configured to simultaneously price several options for the same path information.

14.9 APPENDIX: COM ARCHITECTURE OVERVIEW

In this section we give a global overview of the *Component Object Model* (COM). We introduce it because it underpins much of Excel. You may skip Sections 14.9, 14.10 and 14.11 without loss of continuity, although you should read Sections 14.12 and 14.13. The two key entities in COM are *interfaces* and *components*. In this section we explain what they are from a conceptual viewpoint and we then discuss how COM realises them. An *interface* is a pure specification of intended behaviour. In other words, it consists of a set of *abstract methods*. Each method is devoid of implementation and an interface may not contain non-abstract methods nor may it contain member data. This description is consistent with how C# and Java support interfaces. C++ does not support interfaces as such (abstract classes in C++ are not interfaces because they are structural entities), but they can be emulated by classes having no member data and consisting solely of pure virtual functions. COM has a keyword called `interface` that is defined as follows:

```
#define interface struct
```

which is defined in the file `objbase.h` header included in the Microsoft Win32 *Software Development Kit* (SDK). We take an initial example in C++ that emulates COM functionality. To this end, we define two interfaces that contain methods for presenting data in different ways. We then use *multiple inheritance* to create a class that implements both interfaces. The code is:

```
#include <iostream>
#include <objbase.h>                // Define interface.
```

```
// Abstract interfaces
interface IDraw
{ // Present information

   virtual void __stdcall display() = 0;
   virtual void __stdcall print() = 0;
};

interface ISave
{ // Save to permanent storage

   virtual void __stdcall saveXML() = 0;
   virtual void __stdcall saveDB() = 0;
};

// Interface implementation
class CA : public IDraw, public ISave
{
public:

   // Implement interface IDraw.
   virtual void __stdcall display()
            {std::cout << "CA::display" << std::endl;}
   virtual void __stdcall print()
            {std::cout << "CA::print" << std::endl;}

   // Implement interface ISave.
   virtual void __stdcall saveXML()
            {std::cout << "CA::saveXML" << std::endl;}
   virtual void __stdcall saveDB()
            {std::cout << "CA::saveDB"<< std::endl;}
};
```

We first note that this code is still C++ and not COM code. Second, this code contains the Microsoft-specific extension to the compiler called `__stdcall`. This is a somewhat technical issue and it is concerned with method arguments, how they are pushed onto the stack and who (whether it is the *caller* or the *callee*) removes them from the stack. The general options are:

- *cdecl*: arguments are pushed onto the stack in reverse order, that is from right to left and the caller (calling function) cleans up the stack. This is the C language convention.
- *pascal*: arguments are pushed onto the stack from left to right and the callee cleans up the stack. This was the calling convention in the Win16 operating system.
- *stdcall*: this is the standard and most common calling convention. It is a variation of the *pascal* calling convention and in this case the callee cleans the stack but the parameters are pushed onto the stack from right to left. It is the convention used by the application programming interfaces in the Win32 operating system.

A simple test program based on the above code is:

```
// Client
int main()
{
   CA* pA = new CA ;

   // Get an IDraw pointer.
   IDraw* pd = pA ;

   std::cout << "Client: Use the IDraw interface.";
   pd->display() ;
   pd->print() ;

   // Get an ISave pointer.
   ISave* py = pA;

   std::cout << "Client: Use the ISave interface.";
   py->saveXML() ;
   py->saveDB() ;

   delete pA ;

   return 0 ;
}
```

In general, each method in an interface can have input, output and input/output parameters as well as a return type. In the above simple example the methods had no input parameters and they all had a `void` return type.

14.9.1 COM Interfaces and COM Objects

We now discuss two important concepts in COM, namely *COM interfaces* and *COM objects*. COM interfaces are special in a number of ways. We need to discuss a number of issues which will help us achieve a good understanding of how to define, discover and use these interfaces and objects.

Each system that supports COM must include an implementation of the COM library. The library contains groups of functions that provide basic services to objects and to their clients. For example, it contains functionality for clients to start an object's server. The COM library's services are accessed through ordinary function calls.

An interface is essentially a contract between an object that implements the interface and clients of the object. In order to make the contract work between the object and client we must address the following issues:

- A1: A way to explicitly identify an interface.
- A2: How to describe (define) the methods in an interface.
- A3: How to actually implement an interface.

Regarding activity A1, there are two ways to identify an interface. The first way is to identify the interface by a string name which humans find easy to read while the second way uses a 16-byte (128-bit) *Globally Unique Identifier* (GUID) to identify the interface in software. GUIDs are guaranteed to be unique in space and time and the chance is extremely small that two interfaces (or other entities in COM) will be assigned the same value. In the case of interfaces the unique identifier is called an *interface identifier* (IID). The next challenge is to have some *global clearinghouse* that keeps track of all names (including interface identifiers). This clearinghouse is called the *Registry* and it is the shared system database for the Windows operating system. It contains information about system hardware, software, configuration and users.

A COM component maintains a number in its state called the *reference count*. This count is incremented when a client gets an interface and it is decremented when the client is finished using the interface. Each interface must support the `IUnknown` interface, which has three methods:

- `QueryInterface`: the client can call this method to discover whether a component supports a particular interface. A pointer to the interface will be returned if the component supports the interface.
- `AddRef`: increments the reference count on the interface.
- `Release`: releases a reference to the interface.

The developer must implement `IUnknown` and its *reference counting mechanism* is similar to how both C++11 and the *Boost Smart Pointer* library automatically manage the lifetime of heap-based objects. The precise specification for `IUnkown` is:

```
interface IUnknown
{
   virtual HRESULT QueryInterface( [in] IID& iid, [out] void** ppv) =0;
   virtual ULONG AddRef(void) =0;
   virtual ULONG Release(void) =0;
};
```

We see that `QueryInterface` has two arguments: the first input parameter `iid` is the identifier for an interface that we wish to determine if it is supported and the second argument `ppv` is set to `NULL` before the call to `QueryInterface`. If the call succeeds we get a pointer to `ppv` and we can then call its methods. Notice that we can specify parameters as input or output parameters.

14.9.2 `HRESULT` and Other Data Types

Components use the 32-bit field `HRESULT` to communicate with clients. Its structure is shown in Figure 14.8. For example, it can signal success or failure when a method is called. Most COM functions return this integer as return type.

The possible return types are:

- `E_ABORT`: the operation was aborted because of an unspecified error.
- `E_ACCESSDENIED`: a general access-denied error.

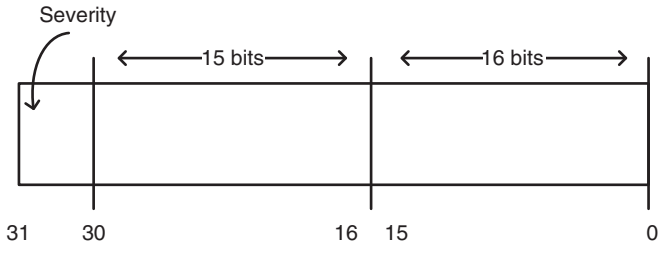


FIGURE 14.8 Format of `HRESULT`

- `E_FAIL`: an unspecified failure has occurred.
- `E_HANDLE`: an invalid handle was used.
- `E_INVALIDARG`: one or more arguments are invalid.
- `E_NOINTERFACE`: the `QueryInterface` method of the interface `IUnknown` did not recognise the requested interface. The interface is not supported.
- `E_NOTIMPL`: the method is not implemented.
- `E_OUTOFMEMORY`: the method failed to allocate memory.
- `E_PENDING`: the data necessary to complete the operation is not yet available.
- `E_POINTER`: an invalid pointer was used.
- `E_UNEXPECTED`: a catastrophic failure occurred.
- `S_FALSE`: the method succeeded and returned the Boolean value `FALSE`.
- `S_OK`: the method succeeded. If a Boolean return value is expected, the returned value is `TRUE`.

We can check the return type in code by calling the macros `SUCCEEDED` and `FAILED` as the following examples show:

```
HRESULT hr = SetStrings ();
if (FAILED (hr))
   return hr;

HRESULT hr2 = PropBag->Read();
if (SUCCEEDED(hr2))
{
   // code
}
```

It is advisable to use these macros when testing a method's return value instead of comparing the result hr against the raw values; however, it is allowed to use these values as return types in code, for example:

```
HRESULT FinalConstruct()
{
   return S_OK;
}
```

14.9.3 Interface Definition Language

COM supports a tool that allows us to define interfaces in a standard way. It is called *Interface Definition Language* (IDL) and it is an extension of the IDL in Microsoft's *Remote Procedure Call* (RPC) which is based on the IDL in the Open Software Foundation's *Distributed Computing Environment* (OSF DCE). IDL describes a COM object's interfaces in an unambiguous manner. It allows us to associate the methods of an interface with its IID. We can also specify the details of the interface in a form that can be machine processed to produce marshalling code when we create *dynamic link libraries* (DLLs):

```
// Interface IDraw
[
   object,                                          // 1.
   uuid(D6FCCE1F-8A2D-4303-A974-09AEC422EFFE),      // 2.
   helpstring("My drawing interface"),              // 3.
   pointer_default(unique)                          // 4.
]

interface IDraw : IUnknown
{
   HRESULT draw();
};
```

14.9.4 Class Identifiers

A COM object is an instance of some class. Each class is assigned a GUID called a *class identifier* (CLSID). A client can pass a CLSID to the COM library to create an instance of the class. However, this is not mandatory. The primary use of a CLSID is to identify a specific piece of code for the COM library to load and execute for instances of the class corresponding to the CLSID.

14.10 AN EXAMPLE

We give a simple example to show how to define both COM interfaces and COM objects. We focus on the language issues that we have discussed. In this case we define interfaces and a class that implements one of them. The interface definitions are:

```
// Interfaces
interface IDraw : IUnknown
{
      virtual void __stdcall display() = 0 ;
};

interface ISave : IUnknown
{

      virtual void __stdcall save() = 0 ;
};
```

The corresponding generated IDs are:

```
// IIDs
//
// {32bb8320-b41b-11cf-a6bb-0080c7b2d682}
static const IID IID_IDraw =
      {0x32bb8320, 0xb41b, 0x11cf,
      {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}};

// {32bb8321-b41b-11cf-a6bb-0080c7b2d682}
static const IID IID_ISave =
      {0x32bb8321, 0xb41b, 0x11cf,
      {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}};
```

We now create a class called `MyClass` that implements `IDraw` that we defined earlier in Section 14.9; notice that this class must also implement the methods of `IUnknown`:

```
//
// Component
//
class MyClass : public IDraw
{

public:
   // IUnknown implementation
   virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv) ;

   virtual ULONG __stdcall AddRef() ;
   virtual ULONG __stdcall Release() ;

   // Interface IDraw implementation
   virtual void __stdcall display() { cout << "display" << endl ;}

   // Constructor
   MyClass() : m_cRef(0) {}

   // Destructor
   ~MyClass() { std::cout << "Bye, bye\n";}

private:
   long m_cRef;
};
```

The implementation of this component is:

```
HRESULT __stdcall MyClass::QueryInterface(const IID& iid, void** ppv)
{

   if (iid == IID_IUnknown)
   {
```

```
      std::cout << "MyClass QI: Return pointer to IUnknown.\n";
      *ppv = static_cast<IDraw*>(this) ;
   }
   else if (iid == IID_IDraw)
   {
      std::cout << "MyClass QI: Return pointer to IDraw.\n";
      *ppv = static_cast<IDraw*>(this) ;
   }
   else
   {
      std::cout <<"MyClass QI: Interface not supported.\n";
      *ppv = NULL ;
      return E_NOINTERFACE;
   }
   reinterpret_cast<IUnknown*>(*ppv)->AddRef() ;
   return S_OK ;
}

ULONG __stdcall MyClass::AddRef()
{
   // Use InterlockedIncrement to ensure thread-safeness
   return InterlockedIncrement(&m_cRef) ;
}

ULONG __stdcall MyClass::Release()
{

   // Use InterlockedDecrement to ensure thread-safeness
   if (InterlockedDecrement(&m_cRef) == 0)
   {
      delete this ;
      return 0 ;
   }
   return m_cRef ;
}

//
// Creation function
//
IUnknown* CreateInstance()
{ // Factory method pattern in the form of a function

   IUnknown* pI = static_cast<IDraw*>(new MyClass) ;
   pI->AddRef() ;
   return pI ;
}
```

A test program to show how to use the component is:

```
int main()
{
   HRESULT hr ;
```

```
    IUnknown* pIUnknown = CreateInstance() ;

    std::cout << "Client: Get interface IDraw.";

    IDraw* pIDraw = NULL ;
    hr = pIUnknown->QueryInterface(IID_IDraw, (void**)&pIDraw) ;

    if (SUCCEEDED(hr))
    {
       std::cout <<"Client: Succeeded getting IDraw.";
       pIDraw->display() ;         // Use interface IDraw.
       pIDraw->Release() ;
    }

    std::cout << "Client: Get interface ISave which MyClass
    does _not_ support.\n";

    ISave* pISave = NULL ;
    hr = pIUnknown->QueryInterface(IID_ISave, (void**)&pISave) ;
    if (SUCCEEDED(hr))
    {
       std::cout << "Client: Succeeded getting ISave.";
       pISave->save() ;             // Use interface ISave.
       pISave->Release() ;
    }

    std::cout << "Client: Release IUnknown interface.";
    pIUnknown->Release() ;

    return 0;
}
```

The output from this code is:

```
Client: Get interface IDraw.MyClass QI: Return pointer to IDraw.
Client: Succeeded getting IDraw.display
Client: Get interface ISave which MyClass does _not_ support.
MyClass QI: Interface not supported.
Client: Release IUnknown interface.Bye, bye
```

14.11 VIRTUAL FUNCTION TABLES

We now discuss how interfaces work in COM and we discuss some of the consequences of the related design choices that have been made in COM. In particular, COM interfaces are implemented using *pure abstract base classes*, that is classes consisting solely of pure virtual member functions and having no member data. An important point to note is that such classes define the specific memory structure that COM requires for an interface. In fact, we are defining the *layout* of a block of memory.

Let us take an example of an interface that models functions to price financial derivatives and to calculate their sensitivities (such as delta and gamma):

```
interface IOption
{ // Functions related to option pricing

   virtual void __stdcall price(double* myPrice) = 0;
   virtual void __stdcall delta(double* myDelta) = 0;
   virtual void __stdcall gamma(double* myGamma) = 0;
};
```

As is usual with C++, we create a class that implements these pure virtual member functions (for convenience, we do not yet implement the full Black–Scholes formula):

```
class Option : public IOption
{ // All values are simulated, used for motivational purposes

public:
   virtual void __stdcall price(double* myPrice) { *myPrice = 1.0;}
   virtual void __stdcall delta(double* myDelta) { *myDelta = 0.5;}
   virtual void __stdcall gamma(double* myGamma) { *myGamma = 0.0;}
};
```

A simple usage is given by:

```
// Option class
IOption* opt = new Option;
double value;
opt->price(&value);
std::cout << value << std::endl;
delete opt;
```

All implementations (for example, class `Option`) are blocks of memory having the same basic layout as shown in Figure 14.9. The abstract class defines the memory structure and

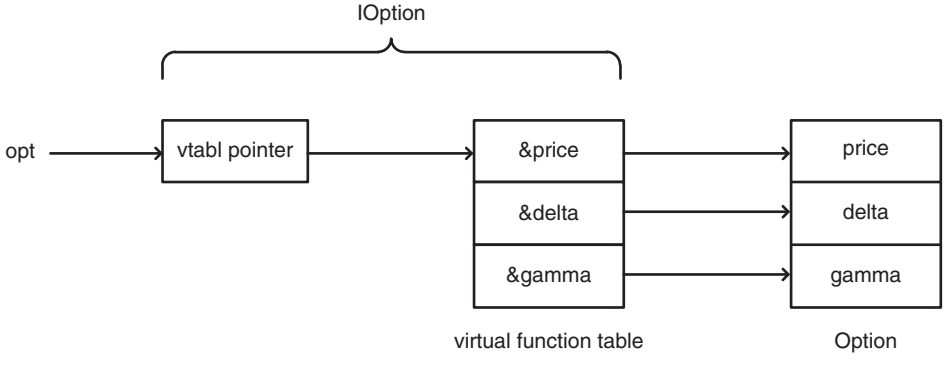


FIGURE 14.9 Memory layout

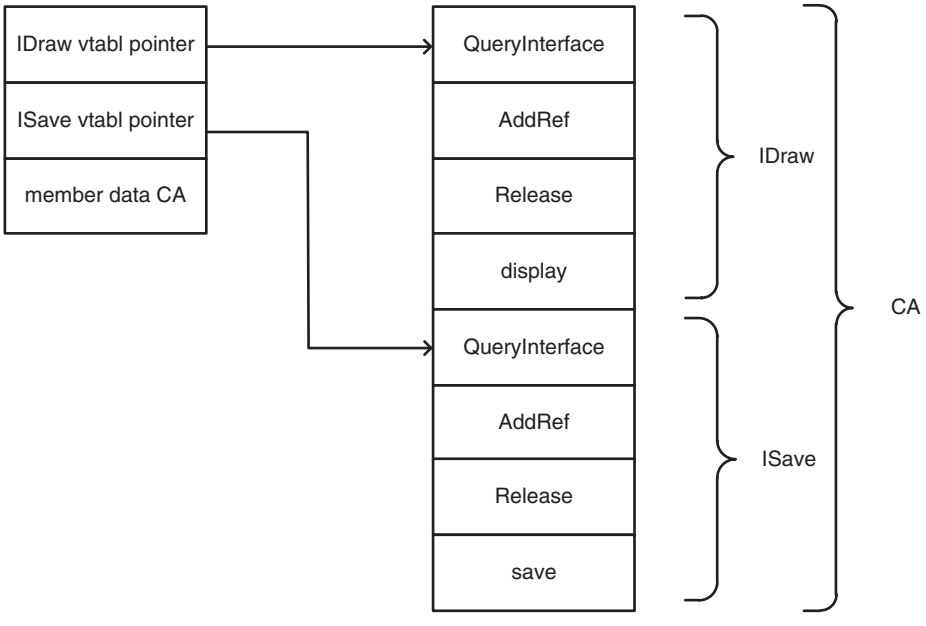


FIGURE 14.10 Memory layout of a class with multiple inheritance

it performs no memory allocation, this being done by derived classes of the abstract class. The *virtual function table* is an array of pointers that point to the implementations of the pure virtual functions. The first entry in the *virtual table* (vtabl) contains the address of the function `price` as it is implemented in derived classes. The second entry in the table is the address of the function `delta` and so on. In general, the layout for a COM interface is the same as the memory layout that the C++ compiler generates for an abstract base class. Thus, abstract classes can be used to define COM interfaces.

We now discuss how memory layout works in the case of the class `CA` from Section 14.9 that implements multiple interfaces. We note that these interfaces are all derived from `IUnknown`. The layout is shown in Figure 14.10.

One final remark; once a client gets an interface pointer from a component then the only thing connecting the client and the component is the binary layout of the interface. In this sense the client is in fact asking for a chunk of memory when it requests an interface to a component.

14.12 DIFFERENCES BETWEEN COM AND OBJECT-ORIENTED PARADIGM

We conclude this chapter with a general discussion on the differences between the COM approach and the C++ object-oriented model. First, COM relies solely on *interface inheritance*, which means that classes can only be derived from interfaces. This is in contrast to C++ *implementation inheritance* in which a derived class *can* inherit functionality and data from one or more base classes. This latter approach can and does lead to maintenance problems, sometimes called the *fragile base class problem* (by which we mean that changes in the base

class can have knock-on effects on the integrity of derived classes). Implementation inheritance corresponds to the *ISA semantic relationship* between structural entities (such as classes) while the term *interface inheritance* is somewhat of a misnomer because neither functionality nor data is inherited. It is more accurate to say that a class *implements an interface*.

Finally, we give some more differences:

a) A class is not a component. It is possible to implement a single COM component using several C++ classes and it is even possible to implement a COM component without using any C++ classes at all.
b) It is not always necessary to inherit interfaces. It is possible to implement an interface in a class and then use this class in client code.
c) COM uses multiple inheritance to allow components to support multiple interfaces.
d) Interfaces in COM never change. If you wish to modify the methods in an interface or even remove or add a method then you should not change the interface but instead you should create a *new version* of the interface and leave the old one alone.
e) COM supports polymorphism because of the ability to switch between multiple interfaces. This feature improves application reusability.

14.13 INITIALISING THE COM LIBRARY

Finally, in order to initialise the COM library we call `CoInitialize` before any of the functions in COM can be used. When we no longer need the COM library we must call `CoUninitialize`.

The COM library is initialised only once per process. In-process components do not need to initialise the COM library. These components do not change the fundamental structure of a *Windows* program.

Calling `CoInitialize` multiple times in a process is allowed as long as each `CoInitialize` has a corresponding `CoUninitialize`.

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 15

Univariate Statistical Distributions

15.1 INTRODUCTION, GOALS AND OBJECTIVES

In this chapter we give an introduction to *univariate statistical distributions*, their properties and some initial applications. We focus on the C++ `<random>` library and the *Boost Statistics* library. The advantage of this approach in our opinion lies in the fact that we use standardised and de-facto standard libraries. The main topics are:

- How C++ and Boost support univariate distributions.
- Black–Scholes–Merton option pricing using the C++11 error function. Computing option sensitivities.
- Generating input data to numerical processes using random number generators.
- Creating generic classes to encapsulate functionality across a wide range of distribution types in both `<random>` and Boost.

Much of the flexibility of the code can be attributed to features that C++ supports, for example variadic parameters, tuples and template–template parameters. We also introduce the *Command* design pattern (GOF, 1995) in the context of callback functions and event triggering mechanisms.

15.2 THE ERROR FUNCTION AND ITS UNIVERSALITY

The *error function* (also called the *Gaussian error function*) is defined by:

$$\text{erf}(x) = \frac{1}{\sqrt{\pi}} \int_{-x}^{x} e^{-t^2} dt = \frac{2}{\sqrt{\pi}} \int_{0}^{x} e^{-t^2} dt. \tag{15.1}$$

In statistics this function has the following interpretation: let X be a normally distributed random variable with mean 0 and variance 1/2. Then $\text{erf}(x)$ describes the probability of X falling in the range $[-x,\ x]$ where $x \geq 0$. The *complementary error function* is defined by:

$$\text{erfc}(x) = 1 - \text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_{x}^{\infty} e^{-t^2} dt. \tag{15.2}$$

	Erf	Erf c
+1	+1	+ 0
i 1	–1	2
Na N	NaN	NaN

FIGURE 15.1 Error handling and extreme values for the error functions

C++11 supports both the error function and the complementary error function. Both functions accept `float, double, long double` and integral arguments:

```
// Error functions
int i = 1;
float f = 3.0F;
double d = 3.14;
long double d2 = 2.71;
std::cout << "i, erf(x): " << i << ", " << std::erf(i) << '\n';
std::cout << "f, erf(x): " << f << ", " << std::erf(f) << '\n';
std::cout << "d, erfc(x): " << d << ", " << std::erfc(d) << '\n';
std::cout << "d2, erfc(x): " << d2 << ", " << std::erfc(d2) << '\n';
```

In Figure 15.1 we give the values of the error functions for special cases of their arguments.

The error functions are related to the *cumulative distribution* Φ (integral of the standard normal distribution) by the formulae:

$$\Phi(x) \equiv N(x) = \frac{1}{2} + \frac{1}{2}\,\mathrm{erf}(x/\sqrt{2}). \tag{15.3}$$

We can check the following relationships:

$$\begin{aligned} \mathrm{erf}(x) &= 2\Phi(x\sqrt{2}) - 1 \\ \mathrm{erfc}(x) &= 2\Phi(-x\sqrt{2}) = 2(1 - \Phi(x\sqrt{2})). \end{aligned} \tag{15.4}$$

The error function has many applications in statistics, probability and partial differential equations.

15.2.1 Approximating the Error Function

Since C++11 officially supports the error function we recommend using it instead of other approximations unless there is a good reason for not doing so. It is possible to use other approximations based on polynomials or rational functions (Abramowitz and Stegun, 1965). For example, a fifth-order polynomial approximation (called '26.2.16' in Abramowitz and Stegun, 1965) for non-negative values of the argument is given by:

```
const double SQRT2PI = std::sqrt(2.0*3.14159);
const double p = 0.33267;
const double a1 = 0.4361836;
```

```
const double a2 = -0.1201876;
const double a3 = 0.9372980;

double Cdf_26_2_16(double x)
{ // 26.2.16 O(-5) accurate

      double t = 1.0 / (1.0 + p*x);
      double Z = std::exp(-0.5*x*x) / SQRT2PI;

      return 1.0 - Z*(t * (a1 + t*(a2 + a3*t)));
}
```

Other implementations are possible, such as '26.2.17' (order 8) and '26.2.18' (order 4) and their accuracy can be compared with that of the C++11 implementation (the code is on the distribution medium delivered by the author):

```
   // 101 tests of cumulative normal distribution
   double x = 2.3;
   std::cout << "Cdf 26_2_16 O(5): " << std::setprecision(15)
             << Cdf_26_2_16(x) << '\n';
   std::cout << "Cdf 26_2_17 O(8): " << std::setprecision(15)
             << Cdf_26_2_17(x) << '\n';
   std::cout << "Cdf 26_2_18 O(4): " << std::setprecision(15)
             << Cdf_26_2_18(x) << '\n';
   std::cout << "Cdf C++11 via erfc: " << std::setprecision(15)
             << N(x) << '\n';
```

In the final case we have conveniently defined a function for the cumulative distribution that we use in code (for example, in option pricing formulae):

```
double N(double x)
{ // aka CdfN(x)

      return 0.5*std::erfc(-x / std::sqrt(2.0));
}
```

For example, this function can be used in the Black–Scholes formula to price a call option that we also discuss in this chapter.

15.2.2 Applications of the Error Function

Our main interest in the error function lies in the ability to recover the cumulative distribution function. It is similar to a probability in the sense that its integral on the real line is one:

$$\frac{1}{\sqrt{\pi}} \int_{-\infty}^{\infty} e^{-t^2} dt = 1. \tag{15.5}$$

The cumulative function plays an important role in computational finance. For completeness, we show how the error function arises when looking for the solution of PDEs. To this end, we discuss the initial value problem for the one-dimensional heat equation on the positive semi-infinite interval:

$$\begin{aligned} &\frac{\partial u}{\partial t} = a^2 \frac{\partial^2 u}{\partial x^2}, \quad 0 < x < \infty, t > 0 \\ &u(x, 0) = f(x), \quad 0 \leq x < \infty. \end{aligned} \tag{15.6}$$

An analytical solution to this problem can be found using Laplace or Fourier transform methods (see Stakgold, 1998). In the case of constant initial condition $f(x) = u_0$ the solution becomes:

$$u(x, t) = u_0 \text{erf}\left(\frac{x}{2a\sqrt{t}}\right). \tag{15.7}$$

Then we can solve system (15.6) by using the C++11 error function. For the case of non-constant initial condition (payoff), see Exercise 1 in the context of the Black–Scholes PDE.

15.3 ONE-FACTOR PLAIN OPTIONS

In this section we discuss the Black–Scholes formula to price one-factor plain put and call options. A full discussion is given in Haug (2007) and the focus here is on examining the formula from a number of perspectives:

- Testing the accuracy of the C++ error function. We implement the cumulative normal distribution in terms of the error function and we test if expected results are produced.
- Generating option call prices. We compare option prices produced by the finite difference method with the exact prices.
- Introducing a number of design patterns (GOF, 1995) that are useful in the current context.
- Events, notification and callbacks.

We introduce the generalised Black–Scholes formula to calculate the price of a call option on an underlying asset. In general, the call price is a function:

$$C = C(S,\ K,\ T,\ r,\ \sigma) \tag{15.8}$$

where:

S = asset price
K = strike (exercise) price
T = exercise (maturity) date
r = risk-free interest rate
σ = constant volatility.

We can view the call option price C as a function because it maps a vector of parameters into a real value. The well-known exact formula for C is given by:

$$C = Se^{(b-r)T}N(d_1) - K\,e^{-rT}N(d_2) \tag{15.9}$$

where $N(x)$ is the standard cumulative normal (Gaussian) distribution function defined by:

$$N(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-y^2/2} dy \tag{15.10}$$

and where:

$$\begin{aligned} d_1 &= \frac{\log(S/K) + (b + \sigma^2/2)T}{\sigma\sqrt{T}} \\ d_2 &= \frac{\log(S/K) + (b - \sigma^2/2)T}{\sigma\sqrt{T}} = d_1 - \sigma\sqrt{T}. \end{aligned} \tag{15.11}$$

The *cost-of-carry* parameter b has specific values depending on the kind of security (Haug, 2007):

$b = r$ is the Black–Scholes stock option model

$b = r - q$ is the Merton model with continuous dividend yield q

$b = 0$ is the Black–Scholes futures option model

$b = r - R$ is the Garman and Kohlhagen currency option model, where R is the foreign risk-free interest rate.

Since we are mainly interested in computing option prices and their sensitivities, we have decided to design the software here using the *Command* design pattern based on subtype polymorphism in combination with a class hierarchy (GOF, 1995). The definition is:

> Command *is a behavioural design pattern in which an object is used to encapsulate all information needed to perform an action or trigger an event at a later time. This information includes the method name, the object that owns the method and values for the method parameters.*

In the current case each dedicated object computes either an option price or one of its greeks. The corresponding UML diagram is shown in Figure 15.2 for the case of call options, for convenience (the case of put options as well as option vega, rho and theta have been left out). We note the presence of a *composite* command class that can contain other commands. In this way we can create *trees* and *recursive aggregates* of commands. Finally, each command (when executed) sends its result to a *Receiver* object which in the current case is the console but it could be adapted to other receivers such as another algorithm, Excel or a remote API. Composite receivers are also possible. In this case the event is multicasted to several receivers.

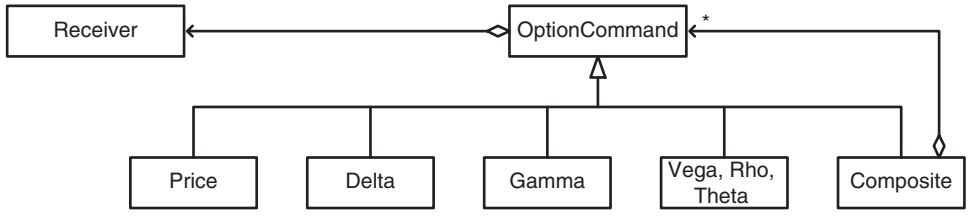


FIGURE 15.2 Commands for call and put options

This kind of functionality is supported in the *Boost C++ Signals2* library (see Demming and Duffy, 2010).

We now discuss how we designed the configuration from Figure 15.2 in C++. The design is based on subtype polymorphism:

```
class OptionCommand
{
private:

protected:
   double K;   double T; double r; double b; double sig;

public:
   OptionCommand() = default;

   explicit OptionCommand(double strike, double expiration,
                          double riskFree, double costOfCarry, double
                          volatility)
                                 : K(strike), T(expiration), r(riskFree),
                                 b(costOfCarry), sig(volatility) {}

   // Want to forbid copy constructor and assignment operator
   OptionCommand(const OptionCommand& c) = delete;
   OptionCommand& operator = (const OptionCommand& c) = delete;

   // The abstract interface
   virtual void execute(double S) = 0;

   // Implement as function object; example of Template Method Pattern
   virtual void operator () (double S)
   {
      // Call derived class' execute()
      execute(S);
   }
};
```

We see that this class is also a function object because it implements the *function call operator* whose implementation is an example of the *Template Method* design pattern

(GOF, 1995). We now reduce the scope by concentrating on call option price, delta and gamma. The three derived classes are:

```
class CallPrice final : public OptionCommand
{
public:
   explicit CallPrice(double strike, double expiration, double riskFree,
                      double costOfCarry, double volatility)
      : OptionCommand(strike, expiration, riskFree, costOfCarry,
      volatility) {}

   virtual void execute(double S) override
   {
      double tmp = sig * std::sqrt(T);
      double d1 = (std::log(S / K) + (b + (sig*sig)*0.5) * T) / tmp;
      double d2 = d1 - tmp;

      std::cout << "Call Price: " <<
                  (S * std::exp((b - r)*T) * N(d1)) (K * std::
                  exp(-r * T)* N(d2))
                  << '\n';
   }
};

class CallDelta final : public OptionCommand
{
public:
   explicit CallDelta(double strike, double expiration, double riskFree,
                      double costOfCarry, double volatility)
      : OptionCommand(strike, expiration, riskFree, costOfCarry,
      volatility) {}

   virtual void execute(double S)  override
   {
      double tmp = sig * std::sqrt(T);
      double d1 = (std::log(S / K) + (b + (sig*sig)*0.5) * T) / tmp;

      std::cout << "Call delta: " << std::exp((b - r)*T) * N(d1) << '\n';
   }
};

class CallGamma final : public OptionCommand
{
public:
   explicit CallGamma(double strike, double expiration, double riskFree,
                      double costOfCarry, double volatility)
      : OptionCommand(strike, expiration, riskFree, costOfCarry,
      volatility) {}

   virtual void execute(double S) override
   {
```

```
      double tmp = sig * std::sqrt(T);
      double d1 = (std::log(S / K) + (b + (sig*sig)*0.5) * T) / tmp;

      std::cout << "Call gamma: "
                << n(d1) * std::exp((b - r)*T)  / (S * tmp) << '\n';
   }
};
```

A simple example of use is:

```
void TestCP()
{ // Haug 2007 page 3, C = 2.13337, P = 5.84628

      double K = 65.0;
      double T = 0.25;
      double r = 0.08;
      double b = r;
      double sig = 0.3;

      double S = 60.0;

      CallPrice cp(K, T, r, b, sig);
      cp.execute(S);

      PutPrice pp(K, T, r, b, sig);
      pp.execute(S);
}
```

We now create collections of commands and then execute them. The design is:

```
using CompositeCommandData = std::list<std::shared_ptr<OptionCommand>>;

class CompositeCommand  : public OptionCommand
{
private:
   CompositeCommandData data;
public:
   CompositeCommand() : data(CompositeCommandData()) {}

   void add(OptionCommand* cmd)
   {
      data.push_back(std::shared_ptr<OptionCommand>(cmd));
   }

   std::size_t size() const
   {
      return data.size();
   }
```

```
    void execute(double S) override
    { // Iterate over each element of the composite and call
      // its execute()

       for (auto it = data.begin(); it != data.end(); ++it)
       {
          (*it)->execute(S); // nested pointer
       }
    }
};
```

This class supports the creation of trees of option commands to any level of nesting. It is a special case of the *Composite* design pattern (GOF, 1995). The implementation can also be generalised to one that is a generic composite using shared pointers (Demming and Duffy, 2010). In other words, you can apply the *Command* pattern in new applications and "use the solution a million times over without ever doing it the same way twice" (Alexander, 1979).

We take our first example of a composite command:

```
void TestCompositeCommand()
{ // Create a composite option command

      double K = 65.0;
      double T = 0.25;
      double r = 0.08;
      double b = r;
      double sig = 0.3;

      double S = 60.0;

      // Create composite command
      std::cout << "Command on options\n";
      CompositeCommand optionCommand;

      // Add prices and their deltas
      optionCommand.add(new CallPrice (K, T, r, b, sig));
      optionCommand.add(new PutPrice(K, T, r, b, sig));

      optionCommand.add(new CallDelta(K, T, r, b, sig));
      optionCommand.add(new PutDelta(K, T, r, b, sig));


      std::cout << "Size of commandlist: "
                << optionCommand.size();
      optionCommand.execute(S);
}
```

The output produced is:

```
Command on options
Size of commandlist: 4
```

```
Call Price: 2.13337
Put Price: 5.84628
Call delta: 0.372483
Put delta: -0.627517
```

We now take an example in which we group option prices, deltas and gammas into their own '*mini-composites*':

```
void TestCompositeTreeCommand()
{ // Create a (deep) composite option command in tree structure

   double K = 65.0;
   double T = 0.25;
   double r = 0.08;
   double b = r;
   double sig = 0.3;

   double S = 60.0;

   // Create composite command
   std::cout << "Command on options, tree structure\n";
   CompositeCommand optionCommandPrice;
   CompositeCommand optionCommandDelta;
   CompositeCommand optionCommandGamma;

   // Group 1
   optionCommandPrice.add(new CallPrice(K, T, r, b, sig));
   optionCommandPrice.add(new PutPrice(K, T, r, b, sig));

   // Group 2
   optionCommandDelta.add(new CallDelta(K, T, r, b, sig));
   optionCommandDelta.add(new PutDelta(K, T, r, b, sig));

   // Group 3
   optionCommandGamma.add(new CallGamma(K, T, r, b, sig));
   optionCommandGamma.add(new PutGamma(K, T, r, b, sig));

   CompositeCommand optionCommand;
   optionCommand.add(&optionCommandDelta);
   optionCommand.add(&optionCommandPrice);
   optionCommand.add(&optionCommandGamma);

   std::cout << "Executing commandlist of size : "
             << optionCommand.size() << '\n';
   optionCommand.execute(S);
   std::cout << "\nFinished executing commandlist of size : "
             << optionCommand.size() << '\n';

}
```

The output is:

```
Command on options, tree structure
Executing commandlist of size : 3
Call delta: 0.372483
Put delta: -0.627517
Call Price: 2.13337
Put Price: 5.84628
Call gamma: 0.0420434
Put gamma: 0.0420434

Finished executing commandlist of size : 3
```

15.3.1 Other Scenarios

In the previous section we computed prices and sensitivities in a command object and we sent the result to a *receiver object*. In some cases we may wish to incorporate the code directly in a module to compute prices and sensitivities. We are specifically thinking of comparing price and sensitivities as a function of one or two independent variables. Typically, we produce arrays and matrices of values. Some examples are:

- Option price as a function of some parameter.
- Option sensitivity as a function of some parameter.
- Option price as a function of two parameters.
- Option sensitivity as a function of two parameters.

We discuss a number of these use cases in this and the next section by replicating the graphs in Cox and Rubinstein (1985) and Haug (2007). We first discuss the case of a call option. We create an array of stock values and then compute the call price (by calling a stored lambda function) and finally we display the result in Excel:

```
void DisplayCallPrice()
{ // Cox and Rubinstein 1985 Figure 5-4

   double K = 50.0;
   double T = 0.4;
   double r = 0.0106;
   double b = r;
   double sig = 0.3;

   // Use lambda function for convenience
   auto callPrice = [&](double S)
   {
      double tmp = sig * std::sqrt(T);
      double d1 = (std::log(S / K) + (b + (sig*sig)*0.5) * T) / tmp;
      double d2 = d1 - tmp;

      return (S * std::exp((b - r)*T) * N(d1)) -
      (K * std::exp(-r * T)* N(d2));
   };
```

```
    std::size_t N = 100;
    std::vector <double> S(N);
    std::vector <double> C(S.size());

    // Generate array of underlyings
    double startS = 5.0;
    std::iota(std::begin(S), std::end(S), startS);

    for (std::size_t i = 0; i < S.size(); ++i)
    {
       C[i] = callPrice(S[i]);
    }

    // Display in Excel
    ExcelDriver xl; xl.MakeVisible(true);
    xl.CreateChart(S, C, "Call Value as a function of S");

}
```

The output is shown in Figure 15.3.

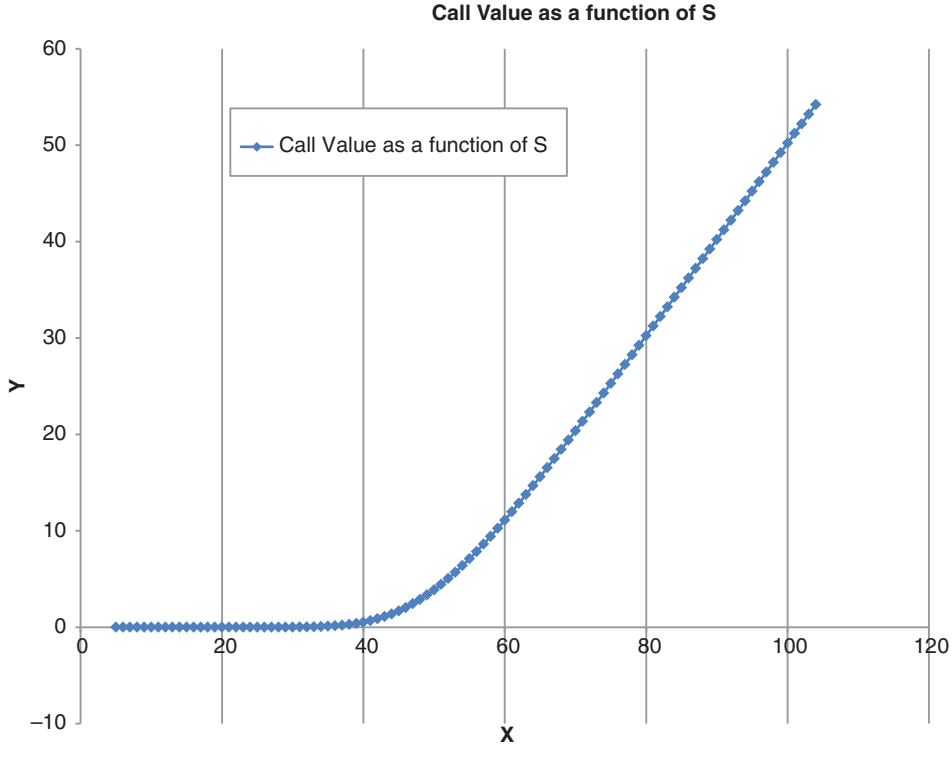


FIGURE 15.3 Option price

It is possible to differentiate the call price C with respect to any of the parameters to produce a formula for option sensitivities. For example, some differentiation allows us to show that in the case of call options:

$$\begin{aligned}
\Delta_C &\equiv \frac{\partial C}{\partial S} = e^{(b-r)T} N(d_1) \\
\Gamma_C &\equiv \frac{\partial^2 C}{\partial S^2} = \frac{\partial \Delta_C}{\partial S} = \frac{n(d_1) e^{(b-r)T}}{S\sigma\sqrt{T}} \\
Vega_C &\equiv \frac{\partial C}{\partial \sigma} = S\sqrt{T} e^{(b-r)T} n(d_1) \\
\Theta_C &\equiv -\frac{\partial C}{\partial T} = -\frac{S\sigma e^{(b-r)^T n(d_1)}}{2\sqrt{T}} - (b-r) S e^{(b-r)T} N(d_1) - rKe^{-rT} N(d_2).
\end{aligned} \tag{15.12}$$

We can also carry out the same exercise for delta and gamma as follows:

```
void DisplayCallDelta()
{ // Cox and Rubinstein 1985 Figure 5-12

   double K = 50.0;
   double T = 0.4;
   double r = 0.0106;
   double b = r;
   double sig = 0.3;

   // Use lambda function for convenience
   auto callDelta = [&](double S)
   {
      double tmp = sig * std::sqrt(T);
      double d1 = (std::log(S / K) + (b + (sig*sig)*0.5) * T) / tmp;

      return std::exp((b - r)*T) * N(d1);
   };

   std::size_t N = 100;
   std::vector <double> S(N);
   std::vector <double> C(S.size());

   // Generate array of underlyings
   double startS = 5.0;
   std::iota(std::begin(S), std::end(S), startS);

   for (std::size_t i = 0; i < S.size(); ++i)
   {
      C[i] = callDelta(S[i]);
   }
```

```
   // Display in Excel
   ExcelDriver xl; xl.MakeVisible(true);
   xl.CreateChart(S, C, "Call Delta as a function of S");
}

void DisplayCallGamma()
{ // Cox and Rubinstein 1985 Figure 5-16

   double K = 50.0;
   double T = 0.4;
   double r = 0.0106;
   double b = r;
   double sig = 0.3;

   // Use lambda function for convenience
   auto callGamma = [&](double S)
   {
      double tmp = sig * std::sqrt(T);
      double d1 = (log(S / K) + (b + (sig*sig)*0.5) * T) / tmp;

      return n(d1) * std::exp((b - r)*T) / (S * tmp);
   };

   std::size_t N = 100;
   std::vector <double> S(N);
   std::vector <double> C(S.size());

   // Generate array of underlyings
   double startS = 5.0;
   std::iota(std::begin(S), std::end(S), startS);

   for (std::size_t i = 0; i < S.size(); ++i)
   {
      C[i] = callGamma(S[i]);

   // Display in Excel
   ExcelDriver xl; xl.MakeVisible(true);
   xl.CreateChart(S, C, "Call Gamma as a function of S");
}
```

The output is shown in Figures 15.4 and 15.5, respectively.

15.4 OPTION SENSITIVITIES AND SURFACES

We now discuss displaying sensitivities in Excel as a function of two independent variables. There are several parameters that are used as input to the Black–Scholes formula and we select two of them as independent variables. They vary in a range while keeping the other parameters constant. For example, we may wish to compute option delta as a function of asset price and days to maturity. We have a formula to compute delta but in this case we model it as a stored

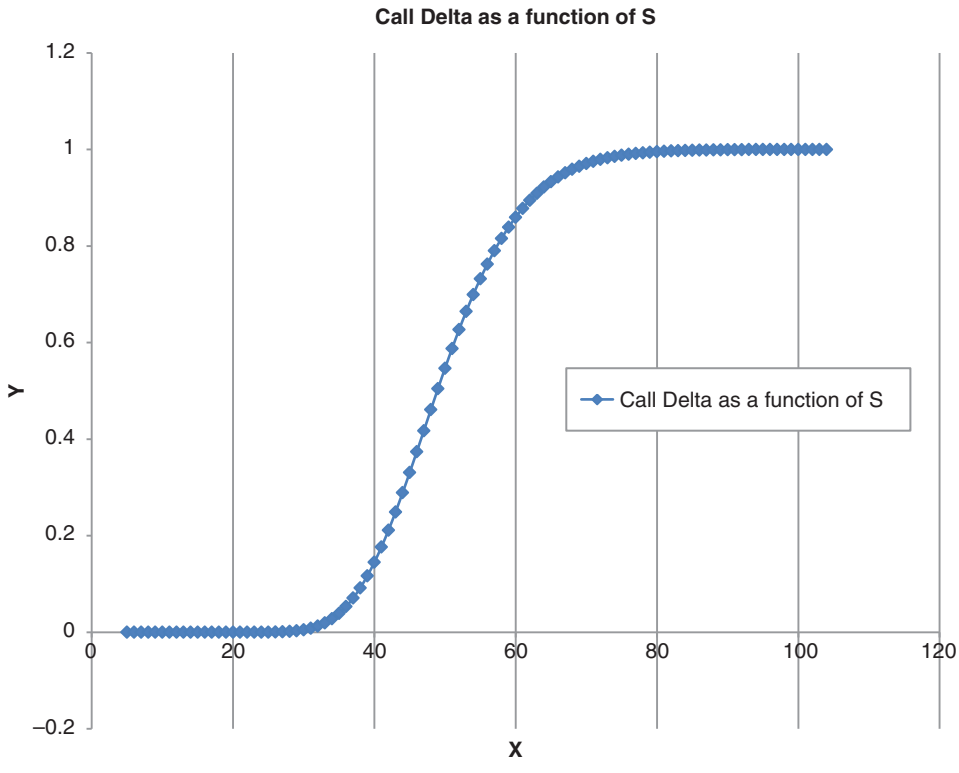


FIGURE 15.4 Call delta

lambda function of arity two while the other parameters are captured variables. In general, we can create families of functions in this way by letting certain parameters play the role of input arguments while letting the other parameters play the role of captured variables. *It can be seen as a kind of design pattern*. In the case of two input arguments we speak of a *surface*.

We take an example of computing the delta of a call option. The asset price and days to maturity play the roles of the horizontal and vertical axes, respectively. We create a matrix of values (using one of the matrix classes that we introduced in Chapter 10), as the following code shows:

```
void DisplayCallDeltaSurface()
{ // Haug 2007 page 26

   // Delta as a function of S and T
   // A. Set up option's captured variables
   double K = 100.0;
   //   double T = 0.4;
   double r = 0.07;
   double b = 0.04;
   double sig = 0.3;

   // B. Define function for delta. lambda function for convenience
   auto callDelta = [&](double S, double T)
```

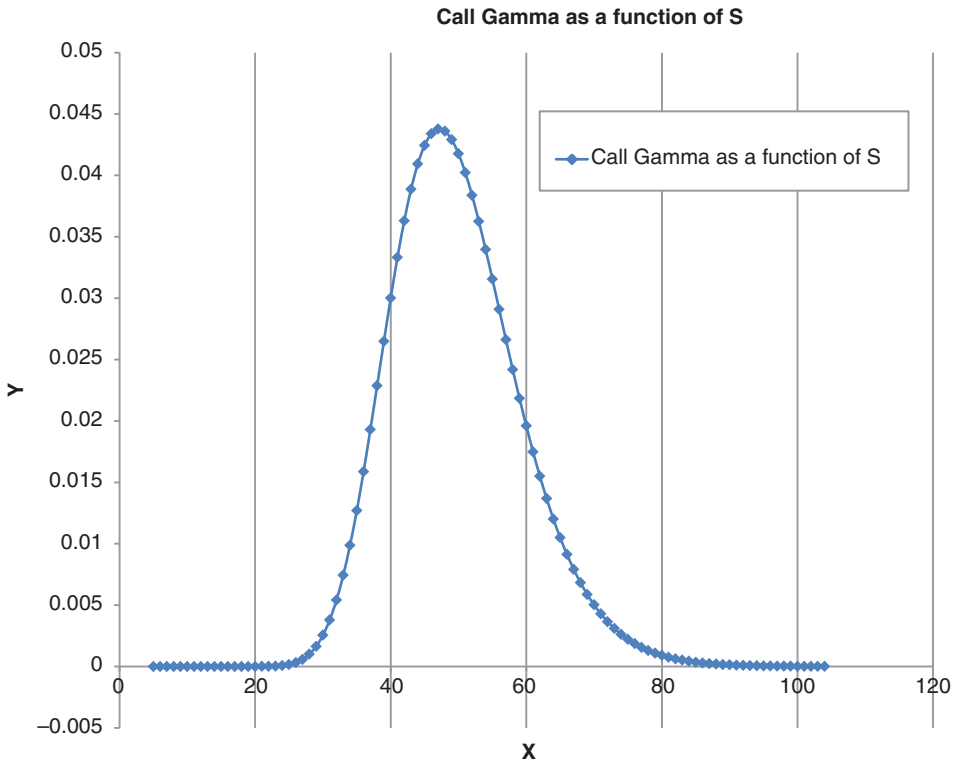


FIGURE 15.5 Call gamma

```
{
   double tmp = sig * std::sqrt(T);
   double d1 = (std::log(S / K) + (b + (sig*sig)*0.5) * T) / tmp;

   return std::exp((b - r)*T) * N(d1);
};

// C. Create meshes in the S and T directions
std::size_t N = 20;
std::vector <double> S = CreateMesh(N, 25,150);
for (auto el : S)
{
   std::cout << el << '\n';
}

std::size_t M = 20;
std::vector <double> TExpire = CreateMesh(M, 0.01, 0.5);

// D. Compute the matrix of deltas
NestedMatrix<double> C(N + 1, M + 1);
for (std::size_t i = 0; i < C.size1(); ++i)
```

```
    {
        for (std::size_t j = 0; j < C.size2(); ++j)
        {
            C(i, j) = callDelta(S[i], TExpire[j]);
        }
    }

    // E. Display in Excel
    ExcelDriver xl; xl.MakeVisible(true);
    std::string sheetName("Call delta surface");
    long row = 1; long col = 1;
    xl.AddMatrix<NestedMatrix<double>>(C, sheetName, row, col);
}
```

The output is shown in Figure 15.6.

15.5 AUTOMATING DATA GENERATION

This book is concerned with numerical methods for applications in computational finance. Once a method has been programmed we need to write programs, organise test cases and test suites as well as control their run-time execution. In general, developers can use libraries such as *Boost.Test* or *Google Test*. A treatment of these libraries is outside the scope of this book. However, we discuss a number of related topics to automate the testing process and to generate data that can be used as input to numerical processes. In particular, we are interested in *datasets* that are in fact collections of *polymorphic tuples* or *samples*. The *arity* of a dataset is the arity of the samples that it contains. *Monomorphic datasets* are simpler because all samples in such datasets have the same type and the same arity. Our main focus here is on automatically generating both polymorphic data (parameters in the main) and monomorphic data in the forms of collections containing deterministic or randomly generated data. Some

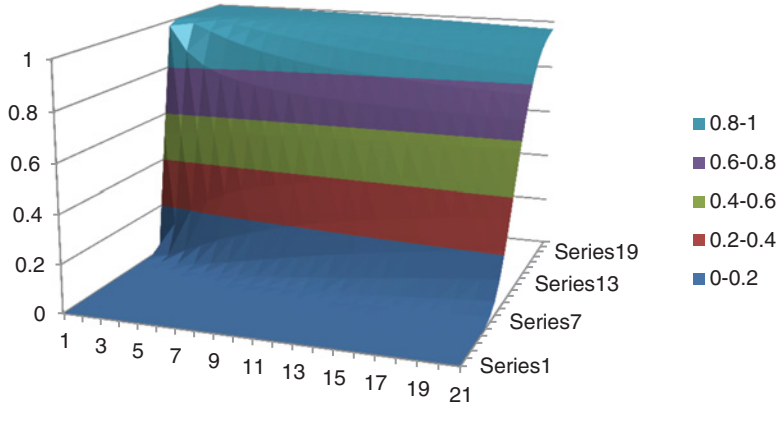


FIGURE 15.6 Spot delta call

general use cases are:

- *Generating functions*: choosing a function that will generate deterministic or random numbers to be used as input arguments to algorithms that we wish to test.
- *Scalability*: extending a test that works with a finite set of data to one that works with larger and possibly infinite datasets.
- *Composition*: testing a function `func3` that takes type `T3` as argument and that calls functions `func1` and `func2` having types `T1` and `T2` as arguments, respectively.

We now discuss how to generate input data to algorithms using STL and `<random>` libraries in combination with user-defined code.

15.5.1 Data Generation Using Random Number Generators: Basics

In addition to providing input data to algorithms using the console we can generate the data that must lie in a given interval using a random number generator. The generated random numbers follow a particular distribution. This way of generating numbers is in many ways preferable to eliciting user input from the console.

The first use case is to generate random numbers in a given interval `(a, b)` based on the uniform distribution using the following free function:

```
template <typename T>  T generateRN(T a, T b)
{ // Generate a uniform random number in interval [a,b]

   std::default_random_engine eng;
   std::random_device rd;
   eng.seed(rd());

   std::uniform_real_distribution<T> uniformVar(a, b);

   return uniformVar(eng);
}
```

This function is easy to use but it makes use of a hard-coded random number engine and the uniform distribution. We get a different random number each time we call this function because we seed the engine for each invocation. An example of use is:

```
double A = 0; double B = 1.0;
std::cout << generateRN(A, B) << '\n';
```

This function allows you to generate uniform random numbers in an interval without too much hassle. For more general cases we create a special generic class that supports the various distributions in the C++ `<random>` library, including the ability to create STL collections of variates from these distributions.

15.5.2 A Generic Class to Generate Random Numbers

The C++ `<random>` library supports approximately 20 statistical univariate distributions. Each distribution is a function object and it allows us to generate random numbers that are distributed

according to the associated probability function. The function call operator needs an input argument representing a *generator*. We take an initial example of generating extreme value (*Gumbel Type 1*) distribution variates using a linear congruential engine:

```
// aka minstd_rand0 typedef
      std::linear_congruential_engine
                        <std::uint_fast32_t, 6807, 0, 2147483647> eng;

// Reinit internal state of random-number engine using new seed value.
std::random_device rd;
eng.seed(rd());

// Create a distribution
double scale = 0.23; double location = 0.5;
std::extreme_value_distribution<double> evd(scale, location);

std::cout << "Extreme value variate: " << evd(eng) << '\n';
```

Looking at this code we see that it is useful but not yet generic. We generalise the code to support the following requirements:

- Distributions with a variable number of input arguments (we use *variadic arguments*).
- Some distributions use *initialiser lists* in their constructors.
- It must be possible to create *collections* of random variates, for example vectors, lists and other STL sequence containers (to realise this requirement we use the *template–template parameter* method).
- Write as little code as possible and reuse as many of the STL algorithms as possible. In this case we use `std::generate()` in combination with the random number generator to create a collection of random numbers.
- The associated engine (stateful source of randomness) is hard-coded. It can be generalised by elevating it to the status of a template parameter. This exercise is outside the current scope. The current engine class is `std::default_random_engine`.

The class that implements the above requirements is:

```
template < typename T, template <typename T> class Dist,
           template <typename T, typename Alloc> class Container =
           std::vector,
                 typename Alloc = std::allocator<T> >

class RNGenerator
{ // Generate random numbers of a given distribution based on
  // ctor parmeters
private:
      Dist<T> dist;
      std::default_random_engine eng; // Others possible, e.g. mt19937
public:
      RNGenerator() = default;
```

```
template <typename... Args> RNGenerator(Args... args)
   : dist (Dist<T>(args...)), eng(std::default_random_engine())
{
   // Hard-coded
   std::random_device rd;
   eng.seed(rd());
}

template<typename T2> RNGenerator(const std::initializer_list<T2>&
                                  initList)
   : dist(initList), eng(std::default_random_engine())
{
   // Hard-coded
   std::random_device rd;
   eng.seed(rd());
}

T operator() ()
{ // STL-compatible function

   return dist(eng);
}

// Template member function using template template parameter trick
Container<T,Alloc> RandomArray(std::size_t n)
{
   Container<T,Alloc> result(n);

   // STL algorithm to assign values generated by a call to this->()
   std::generate(std::begin(result), std::end(result), *this);


   return result;
}

Container<T, Alloc> RandomArrayII(std::size_t n)
{ // Second approach

   Container<T, Alloc> result(n);

   // STL algorithm to assign values generated by a call to this->()
   for (std::size_t i = 0; i < result.size(); ++i)
   {
      result[i] = (*this)();
   }
   return result;
}

void Randomise(std::size_t count, Container<T, Alloc>& con)
{ // Assign random values to the first count elements of
  // container 'con'
```

```
            std::generate_n(std::begin(con), count, *this);
        }
};
```

We take some examples on how to use this class. The code is:

```
// Generate list and vector of uniform <int> and <double>
double A = 0; double B = 1.0;
RNGenerator<double, std::uniform_real_distribution, std::vector> rng(A, B);
std::cout << rng() << '\n';
std::cout << generateRN(A, B) << '\n';

int a = 1; int b = 6;
RNGenerator<int, std::uniform_int_distribution, std::list> rng2(a,b);
std::cout << rng2() << '\n';

std::size_t N = 7;
auto vec = rng.RandomArray(N);
print(vec);

std::size_t M = 30;
auto vec2 = rng2.RandomArray(M);
print(vec2);

auto vec3 = rng.RandomArrayII(N);
print(vec3);

// Randomise 1st N elements of a container
N  = 4;
std::vector<double> v(8, 3.1);
rng.Randomise(N, v);
print(v);
```

15.5.3 A Special Case: Sampling Distributions in C++

C++ supports three kinds of *sampling distributions*:

- `discrete_distribution`: produces random integers in an interval [0, *n*) where the probability of each individual integer is its weight divided by the sum of all *n* weights.
- `piecewise_constant_distribution`: produces random floating-point numbers that are uniformly distributed within each of several subintervals. Each subinterval has its own weight.
- `piecewise_linear_distribution`: produces random floating-point numbers that are uniformly distributed according to a linear probability density function within each of several subintervals. The probability density at each interval boundary is exactly some predefined value.

We are interested in these classes because their constructors use initialiser lists that define the appropriate weights, densities and intervals. To this end, the class `RNGenerator` should be able to support these cases in addition to supporting variadic arguments. We discuss `discrete_distribution` in this section. This distribution produces random integers in the interval $[0, n)$ and the individual probabilities are:

$$p_j = \frac{w_j}{S}, \quad j = 1, \ldots, n$$

where:

$$\{w_j\}_{j=1}^{n} \text{ are the weights and } S = \sum_{j=1}^{n} w_j.$$

We take an example. We create a large number of variates and we place each value in a map. In the limit the frequency histogram should reflect the weight values. The code is:

```
// Discrete distributions that use arrays as input
std::cout << "\nDiscrete distribution\n";
std::size_t NTrials = 100'000;
std::discrete_distribution<int> dist({ 10,25,30,25,10 });

// See if we can compute the frequencies
std::random_device rd;
std::mt19937 eng(rd());
std::map<int, int> freq;
for (std::size_t n = 1; n <= NTrials; ++n)
{
   ++freq[dist(eng)];
}
for (auto p : freq)
{
   std::cout << p.first << " generated " << p.second << " times\n";
}
```

The output from this code is:

```
Discrete distribution
0 generated 9853 times
1 generated 24831 times
2 generated 30179 times
3 generated 25106 times
4 generated 10031 times
```

In this case there are five random integers that can be produced.

15.5.4 Generating Numbers Using a Producer-Consumer Metaphor

In some applications we may wish to create random numbers in a thread, place the numbers in a thread-safe queue and let other threads use these numbers in some way. A good example is a Monte Carlo option pricing framework based on the system decomposition methods from Chapter 9. In this case one component produces random numbers while other components consume these numbers as part of a path-creation algorithm, for example. We can choose a `std::deque` as container because it allows for fast insertion and deletion at its beginning and end. We have encapsulated some test code in a function:

```
template <typename T, template <typename T> class Dist, typename... Args>
          void ProduceAndConsume(std::size_t N, Args... args)
{ // Create a deque (fast insertion and deletion at both ends)
     // Can use a FIFO (first-in first-out) apps, e.g. Monte Carlo

     // Create the object of the appropriate type that will generate the
     // numbers we need.

      RNGenerator<T, Dist, std::deque> rng(args...);

      // Generate the collection of random numbers
      auto rnCollection = rng.RandomArray(N);

      // Process the numbers "Use and lose"
      while (rnCollection.size() > 0)
      {
         std::cout << rnCollection.front() << ",";
         rnCollection.pop_front();
      }
      std::cout << "\nQueue size " << rnCollection.size() << '\n';
}
```

We have also created a similar function that uses an initialiser in addition to variadic arguments:

```
template <typename T, typename T2, template <typename T> class Dist>
          void ProduceAndConsume(std::size_t N, const std::initializer_
                                      list<T2>& initList)
{ // Create a deque (fast insertion and deletion at both ends)
         // Can use a FIFO (first-in first-out) apps, e.g. Monte Carlo

         // Exercise: fill in the details
         // See Exercise 5
}
```

Some examples of use are:

```
std::cout << "\nExponential distribution\n";
std::size_t N = 10; float lambda = 0.5;
ProduceAndConsume<float, std::exponential_distribution>(N,lambda);
```

```
std::cout << "\nGamma distribution\n";
std::size_t M = 20; double alpha = 0.5; double beta = 0.4;
ProduceAndConsume<double, std::gamma_distribution>(M, alpha, beta);
ProduceAndConsume<int, double, std::discrete_distribution> (NTrials,
                  std::initializer_list<double>({ 5,10,20,30,30,10,5 }));
```

Here we see the power of variadic parameters; we write supplier code once and reuse it many times.

15.5.5 Generating Numbers and Data with STL Algorithms

It is worthwhile noting that STL has a number of functions to generate, modify and mutate values in data containers. We can use these functions to generate data that is then used as input to numerical algorithms. We discuss the main categories of STL algorithms in more detail in Chapters 17 and 18 but for the moment we discuss some of the more useful and directly applicable ones:

- `std::iota`: this *modifying algorithm* creates a collection with monotonically increasing values with each value one larger than its predecessor. We need to give a *start value*. An example is:

```
std::size_t N = 100;
std::vector <double> S(N);

// Generate array of underlyings
double startS = 5.0;
std::iota(std::begin(S), std::end(S), startS);
```

- `std::shuffle`: this *mutating algorithm* is supported since C++11 (and it essentially replaces `std::random_shuffle`). It shuffles the elements of a range using a uniform random number generator from `<random>`. An example is:

```
// C++ 11
std::vector<int> vecR{ 1, 2, -2, 3, -3, 4, -4, 5, -5, 8, 9, 7 };
std::default_random_engine eng;
std::shuffle(vecR.begin(), vecR.end(), eng);

std::random_shuffle(vecR.begin(), vecR.end());   // olde style
```

- `std::generate`: this *modifying algorithm* assigns a value to each element in a range using a given function object. An example that generates mesh points for use in a finite difference scheme is:

```
// Mesh
std::size_t N = 4; double h = 1.0 / static_cast<double>(N);
std::vector<double> mesh(N + 1, 0.0);
double val = -h;
auto sequence = [&h, &val]() { val += h; return val; };
```

```
std::generate(mesh.begin(), mesh.end(), sequence);
for (auto i = 0; i < mesh.size(); ++i)
{
   std::cout << mesh[i] << std::endl;
}
```

This option can also be used to generate random numbers.

- `std::generate_n`: this modifying algorithm applies a function to the first *n* elements of a range, for example:

```
// Generate 1) new values, 2) overwrite existing values
std::list<double> myList;
int dupFactor = 4;
std::generate_n(std::back_inserter(myList), dupFactor,std::rand);
```

We mention that the functionality in `<random>` for statistical distributions is limited to producing random variates of univariate distributions.

15.6 INTRODUCTION TO STATISTICAL DISTRIBUTIONS AND FUNCTIONS

The C++ random number library contains classes for random number engines and random number distributions that convert the output of random number engines into variates of statistical distributions. In many applications we need more functionality and for this reason we give an overview of how Boost supports some of these needs.

We give an overview of the univariate statistical distributions and functions in the *Boost Math Toolkit*. The emphasis is on discussing the functionality in the Toolkit, in particular:

- Discrete and continuous distributions, their member functions and defining properties.
- Other non-member functions, for example the probability function and cumulative density functions, kurtosis and skewness.
- Some examples to motivate how to use the classes in the Toolkit.

All distributions use random variables which are mappings of a probability space into some other space, typically a real number. A *discrete probability distribution* is one in which the distribution of the random variable is discrete while a *continuous probability distribution* is one whose cumulative distribution is continuous.

The discrete probability distributions are:

- Bernoulli (a single trial whose outcome is 0 (failure) or 1 (success)).
- Binomial (used to obtain the probability of observing k successes in N trials).
- Negative binomial (used to obtain the probability of k failures and r successes in $k + r$ trials).
- Hypergeometric (describes the number of events k from a sample n drawn from a total population N without replacement).

- Poisson (expresses the probability of a number of events occurring in a fixed period of time).

The continuous probability distributions are:

- Beta (used in Bayesian statistics applications).
- Cauchy–Lorentz (used in physics, spectroscopy and to solve differential equations).
- Chi-squared (used in statistical tests).
- Exponential (models the time between independent events).
- Extreme value (models rare events).
- *F* (the Fisher *F*-distribution that tests if two samples have the same variance).
- Gamma (and Erlang) (used to model waiting times).
- Laplace (the distribution of differences between two independent variates with identical exponential distributions).
- Logistic (used in logistic regression and feedforward neural network applications).
- Log normal (used when the logarithm of the random variable is normally distributed).
- Non-central beta (a generalisation of the beta distribution).
- Non-central chi-squared (a generalisation of the chi-squared distribution).
- Non-central *F* (a generalisation of the Fisher *F*-distribution).
- Non-central *t* (a generalisation of Student's *t*-distribution).
- Normal (Gaussian) (probably the best-known distribution).
- Pareto (compares large and small numbers).
- Rayleigh (combines two orthogonal components having an absolute value).
- Student's *t* (the 'best' approximate distribution approximating an unknown distribution).
- Triangular (used when a distribution is only vaguely known, for example in software projects).
- Weibull (used in failure analysis models).
- Uniform (also known as the *rectangular distribution*, it models a probability distribution with a constant probability).

Each of the above distributions is implemented by a corresponding template class with two template parameters. The first parameter is the underlying data type used by the distribution (the default type is `double`) and the second parameter is the *policy*. In general, a policy is a fine-grained compile-time mechanism that we can use to customise the behaviour of a library. It allows us to change error-handling mechanisms or numerical precision at both program level and the client site.

The *non-member functions* for the above distributions are:

- Cdf (cumulative distribution function).
- Cdf complement (this is 1 − cdf).
- Hazard (the event rate at time *t* conditional on survival until time *t* or later; useful when modelling failure in mechanical systems).
- Chf (cumulative hazard function that measures the accumulation of hazard over time).
- Kurtosis (a measure of the 'peakedness' of a probability distribution).
- Kurtosis_excess (does a distribution have fatter tails than a normal distribution?)
- Mean (the expected value).
- Median (the value separating the lower and higher halves of a distribution).

- Mode (the point at which the probability mass or density function takes its maximum).
- Pdf (probability density function).
- Range (the length of the smallest interval which contains all the data).
- Quantile (points taken at regular intervals from the cdf).
- Skewness (a measure of the asymmetry of a probability distribution).
- Support (the smallest closed interval/set whose complement has probability zero).
- Variance (how far do values differ from the mean?)

We discuss a well-known case. The normal (or Gaussian) distribution is one of the most important statistical distributions because of its ability to model many kinds of phenomena in diverse fields such as economics, computational finance, physics and the social sciences. In general, the normal distribution is used to describe variables that tend to cluster around a mean value. We now show how to implement the normal distribution in Boost and how to call its member and non-member functions:

```
#include <boost/math/distributions/normal.hpp>
#include <boost/math/distributions.hpp>   // Non-member functions.

#include <iostream>
using namespace std;

int main()
{
   // Don't forget to tell compiler which namespace.
   using namespace boost::math;

   normal_distribution<> myNormal(1.0, 10.0); // Default is 'double'.
   cout << "Mean: " << myNormal.mean() << ", standard deviation: "
        << myNormal.standard_deviation() << endl;

   // Distributional properties.
   double x = 10.25;

   cout << "pdf: " << pdf(myNormal, x) << endl;
   cout << "cdf: " << cdf(myNormal, x) << endl;

   // Choose another data type and now a N(0,1) variate.
   normal_distribution<float> myNormal2;
   cout << "Mean: " << myNormal2.mean() << ", standard deviation: "
         << myNormal2.standard_deviation() << endl;

   cout << "pdf: " << pdf(myNormal2, x) << endl;
   cout << "cdf: " << cdf(myNormal2, x) << endl;

   // Choose precision.
   cout.precision(10); // Number of values behind the comma.

   // Other properties.
   cout << "\n***normal distribution:\n";
```

```
    cout << "mean: " << mean(myNormal) << endl;
    cout << "variance: " << variance(myNormal) << endl;
    cout << "median: " << median(myNormal) << endl;
    cout << "mode: " << mode(myNormal) << endl;
    cout << "kurtosis excess: " << kurtosis_excess(myNormal) << endl;
    cout << "kurtosis: " << kurtosis(myNormal) << endl;
    cout << "characteristic function: " << chf(myNormal, x) << endl;
    cout << "hazard: " << hazard(myNormal, x) << endl;

    return 0;
}
```

You can run this code and examine the output. It is fairly straightforward.

15.6.1 Some Examples

Each distribution has defining parameters. The underlying parameters are of numeric type but in most cases we work with double-precision numbers. Finally, we can use *policies* to customise how errors are handled or how quantiles of discrete distributions behave. We take a '101' example to show how to create instances of two distributions, namely uniform and gamma distributions:

```
#include <boost/math/distributions/uniform.hpp>
#include <boost/math/distributions/gamma.hpp>
#include <iostream>
using namespace std;

int main()
{

    // Don't forget to tell compiler which namespace
    using namespace boost::math;

    uniform_distribution<> myUniform(0.0, 1.0);   // Default 'double'
    cout << "Lower value: " << myUniform.lower()
         << ", upper value: " << myUniform.upper() << endl;

    // Choose another data type
    uniform_distribution<float> myUniform2(0.0, 1.0);
    cout << "Lower value: " << myUniform2.lower()
         << ", upper value: " << myUniform2.upper() << endl;

    // Distributional properties
    double x = 0.25;
    cout << "pdf: " << pdf(myUniform, x) << endl;
    cout << "cdf: " << cdf(myUniform, x) << endl;

    // Gamma distribution
    double alpha = 3.0; // Shape parameter, k
    double beta = 0.5; // Scale parameter, theta
```

```
    gamma_distribution<double> myGamma(alpha, beta);

    return 0;
}
```

The class's interfaces are small in the sense that they do not have functions for statistical functions, the latter being realised by overloaded non-member functions. To this end, the main functions are:

- probability distribution function
- cumulative distribution function
- quantile
- hazard function
- cumulative hazard function
- mean, median, mode, variance, standard deviation
- kurtosis, kurtosis excess
- range (the valid range of the random variable over a given distribution)
- support (smallest closed set outside of which the probability is zero).

We take an example to show how to use these functions. In this case we take a *Student's t-distribution* with 30 degrees of freedom:

```
#include <boost/math/distributions/students_t.hpp>
students_t_distribution<float> myStudent(30);
```

Then the above list of functions translates into the following code for this distribution:

```
// Other properties
cout << "\n***Student's t distribution: \n";
cout << "mean: " << mean(myStudent) << endl;
cout << "variance: " << variance(myStudent) << endl;
cout << "median: " << median(myStudent) << endl;
cout << "mode: " << mode(myStudent) << endl;
cout << "kurtosis excess: "
      << kurtosis_excess(myStudent) << endl;
cout << "kurtosis: " << kurtosis(myStudent) << endl;
cout << "characteristic function: " << chf(myStudent, x) << endl;
cout << "hazard: " << hazard(myStudent, x) << endl;

// Range and support
pair<double, double> mySupport = support(myStudent);
cout << "Support: [" << mySupport.first << ","
      << mySupport.second << "]" << endl;
pair<double, double> myRange = range(myStudent);
cout << "Range: [" << myRange.first << ","
      << myRange.second << "]" << endl;
```

Finally, we show how to compute quantiles and the number of degrees of freedom needed in the *t*-distribution for a given significance level:

```
// Quantiles and conversions between significance levels
// (fractions, 0.05) and confidence levels (in percentages,
// e.g. 95%)
double Alpha = 0.25;
cout << "Confidence 1: "
      << quantile(myStudent, Alpha / 2) << endl;
cout << "Confidence, in %: "
      << quantile(complement(myStudent, Alpha / 2)) << endl;

// Required sample sizes for Students t-distribution
double M = 1.2;     // True mean
double Sm = 1.8;    // Sample mean
double Sd = 2.4;    // Sample standard deviation
try
{
   double df = students_t::find_degrees_of_freedom (fabs(M-Sm), Alpha,
   Alpha, Sd);
   int dof = ceil(df) + 1;
   cout << "One-sided degrees of freedom: " << dof << endl;
}
catch(const std::exception& e)
{
   cout << e.what() << endl;
}
```

15.7 ADVANCED DISTRIBUTIONS

There are approximately 35 classes in the *Boost Statistics* library, each of which models a particular univariate distribution. In order to avoid code duplication we have created a single generic class that encapsulates common properties and algorithms. The design rationale is similar to that of the generic class that we created in Section 15.5.2 where we created generic code to generate random numbers. In particular, we create an *adapter* class that supports all distributions and that uses variadic arguments in its constructor. Its interface mimics the corresponding Boost interface in the sense that we use the same member function names (notice that we use the *template–template parameter* trick):

```
template <typename T, template <typename T> class Dist>
      class Distribution
{
private:
      Dist<T> dist;
public:
      template <typename... Args>
      Distribution(Args... args) : dist(Dist<T>(args...)) {}
```

```
T operator () (T x) const
{ // pdf, as function object

   return boost::math::pdf(dist, x);
}

T pdf (T x) const
{ // pdf

   return (*this)(x);
}

T mean() const
{
   return boost::math::mean(dist);
}

T mode() const
{
   return boost::math::mode(dist);
}

T median() const

   return boost::math::median(dist);
}

T standard_deviation() const
{
   return boost::math::standard_deviation(dist);
}

T variance() const
{
   return boost::math::variance(dist);
}

T cdf(T x) const
{ // cdf

   return boost::math::cdf(dist, x);
}

T hazard(T x) const
{ // Hazard function pdf/(1-cdf)

   return boost::math::hazard(dist, x);
}
```

```
        T chf(T x) const
        { // Cumulative hazard function integral of hazard
          // from -inf to x

                          return boost::math::chf(dist, x);
        }

        // The interval [A,B] in which distribution is defined
        T A() const
        {
           return boost::math::range(dist).first;
        }

        T B() const
        {
           return boost::math::range(dist).second;
        }

        std::pair<T, T> range() const
        {
           return boost::math::range(dist);
        }

        std::pair<T, T> support() const
        {
           return boost::math::support(dist);
        }

        T quantile(T p) const
        {
           return boost::math::quantile(dist, p);
        }

        T kurtosis() const
        {
           return boost::math::kurtosis(dist);
        }

        T skewness() const
        {
           return boost::math::skewness(dist);
        }
};
```

We can create a function to test the class:

```
template <typename T, template <typename T> class Dist>
      void TestAnyDistribution(const Distribution<T, Dist>& dist)
{
      // Calling member functions of Generic distribution class
```

```
        T x = 1.1;
        T p = 0.05;

        std::cout << "pdf: " << dist.pdf(x) << '\n';
        std::cout << "cdf: " << dist.cdf(x) << '\n';

        std::cout << "mean: " << dist.mean() << '\n';
        std::cout << "mode: " << dist.mode() << '\n';
        std::cout << "median: " << dist.median() << '\n';

        std::cout << "standard dev: "<<dist.standard_deviation() << '\n';
        std::cout << "variance: " << dist.variance() << '\n';

        std::cout << "hazard: " << dist.standard_deviation() << '\n';
        std::cout << "cumulative hazard: " << dist.variance() << '\n';

        std::cout << "kurtosis: " << dist.kurtosis() << '\n';
        std::cout << "skewness: " << dist.skewness() << '\n';

        std::cout << "quantile " << p << ", "<< dist.quantile(p) << '\n';
}
```

We can now call this test function using a number of specific distributions. We include a `try/catch` block in order to flag incorrect input parameters, for example:

```
template <typename T>
      using DistNormal = boost::math::normal_distribution<T>;

template <typename T>
      using DistBinomial = boost::math::binomial_distribution<T>;
try
{
   Distribution<double, DistNormal> normalDist;
   TestAnyDistribution(normalDist);

   long N = 100; double p = 0.5;
   Distribution<double, DistBinomial> binDist(N,p);
   TestAnyDistribution(binDist);
}
catch(std::exception& e)
{
   std::cout << e.what() << '\n';
}
```

There are of course many other ways to use the new generic class in client code.

15.7.1 Displaying Boost Distributions in Excel

We conclude this chapter by showing how to 'display' the data produced by Boost distribution functions in Excel. In the first case we display the pdf and cdf of an arbitrary distribution. We

use `std::transform` to create arrays and we also use the Excel driver that we introduced in Chapter 14:

```
template <typename T, template <typename T> class Dist>
      void DisplayDistributionCurves(const Distribution<T, Dist>& dist,
                                     T a, T b, std::size_t N)
{ // Display a number of functions of a distribution as line graphs

      // Try to use as much of standard C++ and STL as possible

      // A. Create abscissa array
      T h = (b - a) / static_cast<T>(N);
      std::vector<T> x(N+1, h);
      x[0] = a;
      for (std::size_t i = 1; i < x.size(); ++i)
      {
         x[i] = x[i - 1] + h;
      }

      // B. Create arrays for pdf, cdf and hazard functions
      std::vector<T> pdf(x.size());
      std::transform(std::begin(x), std::end(x), std::begin(pdf),
                     [&](T t) { return dist.pdf(t); });

      std::vector<T> cdf(x.size());
      std::transform(std::begin(x), std::end(x), std::begin(cdf),
                     [&](T t) { return dist.cdf(t); });

      // C. Labels and assemble input for Excel
      std::list<std::string> labels{"pdf", "cdf"};

      // The list of Y values
      std::list<std::vector<double> > curves{pdf,cdf};

      ExcelDriver xl; xl.MakeVisible(true);
      xl.CreateChart(x, labels, curves, "pdf, cdf", "x", "y");
}
```

In other words, we create a single object and we call two of its member functions (pdf and cdf). An example of use is:

```
double a = 0.0; double b = 6.0;
std::size_t n = 50;
DisplayDistributionCurves(normalDist,a,b,n);
```

The output is shown in Figure 15.7.

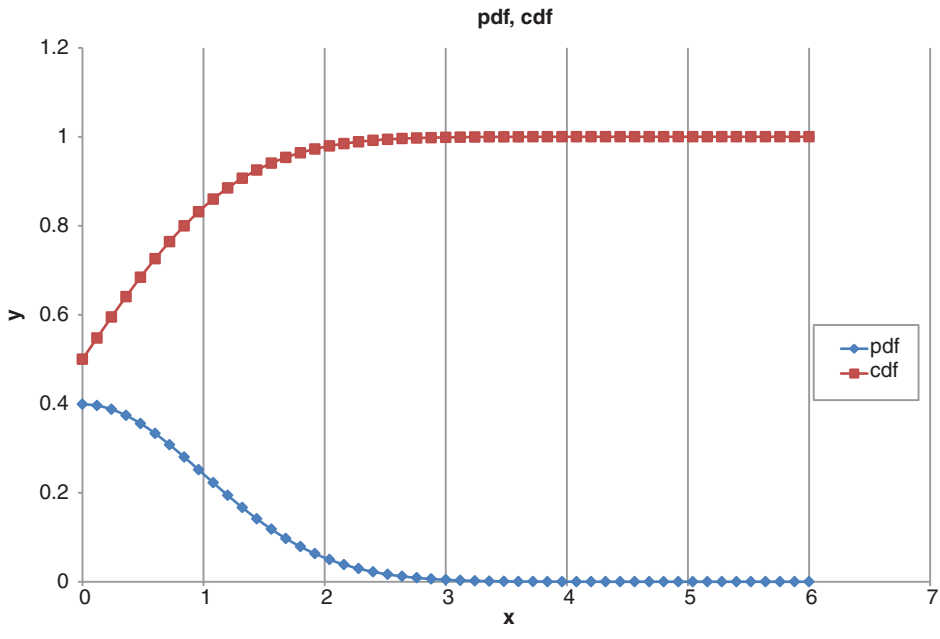


FIGURE 15.7 Normal distribution

We now give an example of the *non-central chi-squared distribution*. Its constructor takes two parameters k and λ that represent the *number of degrees of freedom* and the *non-centrality parameter*, respectively. The code is:

```
void DisplayNonCentralChiSquared(double a, double b, std::size_t N)
{
   // Display non-central chi^2 pdf for a range of k and lambda.

   using T = double;

   // A. Create abscissa array
   T h = (b - a) / static_cast<T>(N);
   std::vector<T> x(N + 1, h);
   x[0] = a;
   for (std::size_t i = 1; i < x.size(); ++i)
   {
      x[i] = x[i - 1] + h;
   }

   // B. Create arrays for pdf for different values of k and lamdba
   Distribution<T, DistNCChiSquared> dist21(2,1);
   Distribution<T, DistNCChiSquared> dist22(2, 2);
   Distribution<T, DistNCChiSquared> dist43(4, 3);

   std::vector<T> pdf21(x.size());
   std::transform(std::begin(x), std::end(x), std::begin(pdf21),
      [&](T t) { return dist21.pdf(t); });
```

```
    std::vector<T> pdf22(x.size());
    std::transform(std::begin(x), std::end(x), std::begin(pdf22),
             [&](T t) { return dist22.pdf(t); });

    std::vector<T> pdf43(x.size());
    std::transform(std::begin(x), std::end(x), std::begin(pdf43),
             [&](T t) { return dist43.pdf(t); });

    // C. Labels and assemble input for Excel
    std::list<std::string> labels{
       "k=2, lambda = 1", "k=2, lambda = 2", "k=4, lambda = 3" };

    // The list of Y values
    std::list<std::vector<double> > curves{ pdf21, pdf22, pdf43 };

    ExcelDriver xl; xl.MakeVisible(true);
    xl.CreateChart(x, labels, curves, "NonCentral Chi^2 pdf", "x", "y");

}
```

An example of use is:

```
std::size_t M = 10;
double L = 0.01; double R = 8.0;
DisplayNonCentralChiSquared(L,R, M);
```

The output is shown in Figure 15.8.

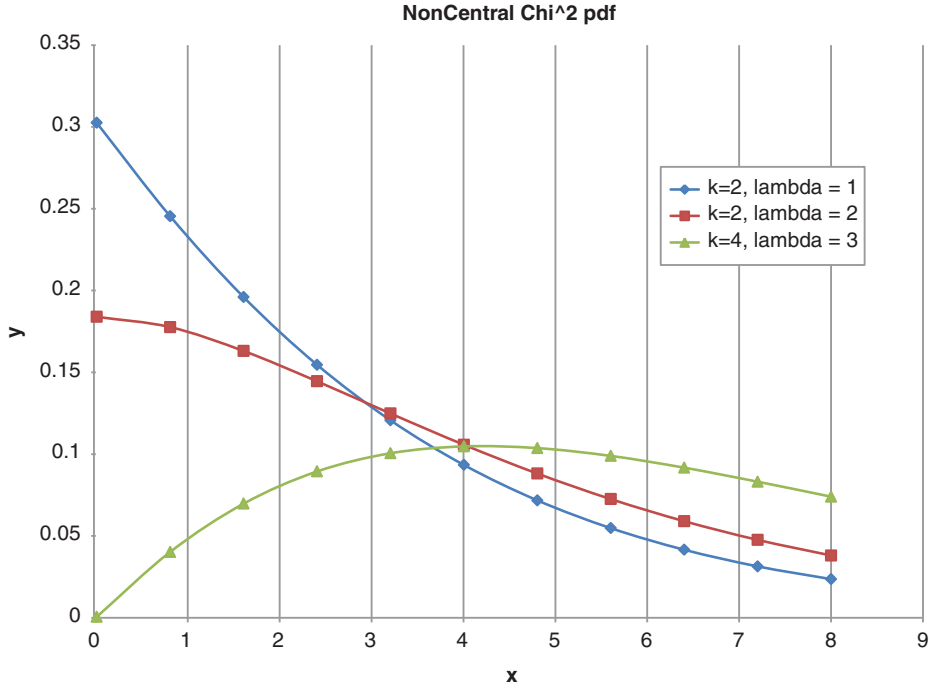


FIGURE 15.8 Modifying parameters

15.8 SUMMARY AND CONCLUSIONS

In this chapter we introduced two C++ libraries that support random number generation and univariate statistical distributions, namely the standard C++ `<random>` library and the *Boost Statistics* library. We discussed the functionality that they deliver and we gave some applications to option pricing and to computing option sensitivities. We also created two generic classes that encapsulate the functionality of the distributions in `<random>` and Boost. Finally, we also showed how to create input datasets to numerical processes based on random number generators.

15.9 EXERCISES AND PROJECTS

1. (PDE and Numerical Analysis Project)

In this exercise we reduce the one-factor Black–Scholes PDE to a simpler heat equation PDE that we then solve using the finite difference methods that we introduced in Chapter 13. The PDE is:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

and let us define the new variable u by:

$$V(S,\ t) = e^{\alpha x + \beta \tau} u(x,\ \tau)$$

where:

$$\alpha = -\frac{1}{2}(2r/\sigma^2 - 1)$$
$$\beta = -\frac{1}{4}(2r/\sigma^2 + 1)^2$$
$$S = e^x,\ t = T - 2\tau/\sigma^2.$$

We can then show that the function u satisfies the heat equation:

$$\frac{\partial u}{\partial \tau} = \frac{\partial^2 u}{\partial x^2}.$$

Answer the following questions:

a) Check that the above transformation leads to a heat equation PDE on an infinite interval.

b) Let $h(x)$ be the payoff (initial condition) corresponding to the new heat equation. It can be shown (Tolstov, 1962) that the solution is:

$$u(x,\ t) = \frac{1}{2\sqrt{\pi\tau}} \int_{-\infty}^{\infty} h(v) e^{-\frac{(x-v)^2}{4\tau}} dv.$$

Compute this solution numerically using your favourite numerical quadrature scheme. For example, we could use *Tanh–Sinh* quadrature (Takahasi and Mori, 1974) which uses

a transformation of the interval $(-1, 1)$ to the real line. The integrand decays at a double-exponential rate. For this reason the method is also known as the *double-exponential (DE) formula*.

c) Now approximate the heat equation using the Crank–Nicolson and ADE (Barakat and Clark variant) methods that we introduced in Chapter 13. You will need to truncate the (infinite) domain of integration to a bounded interval in the space direction and apply numerical boundary conditions.

d) Compare the relative accuracy of the methods in part c). Use the Excel driver software from Chapter 14 to visualise the option price as an array of values at expiration.

e) Compare the values produced by the finite difference method with the solution based on the error function.

2. (Computing Option Sensitivities/Greeks, Small Project)

We focus on computing an option's delta and gamma. *Delta* is an option's sensitivity to small changes in the underlying asset price while *gamma* is the delta's sensitivity to small changes in the underlying asset price. The objective is to use a number of methods to compute delta and gamma and then to analyse their results:

- Analytical solutions (see Haug, 2007 for an extensive list of formulae).
- Numerical greeks (Haug, 2007). In this case we use divided differences to compute delta and gamma, as discussed in a more general setting in Chapter 8. The advantage is that this approach is model independent. The disadvantage is that numerical differentiation is an ill-posed process in general.
- Using cubic splines as discussed in Chapter 13. See also Exercise 3 in Chapter 13.

Answer the following questions for the case of a call option:

a) Create an array of stock prices and compute the corresponding array of option prices for each stock price (keeping the other parameters constant).

b) Compute the arrays containing the delta and gamma values for each stock price. Use analytical formulae for delta and gamma (see Haug, 2007, for example).

c) Compute the arrays containing the delta and gamma values for each stock price. Use divided difference formulae for delta and gamma (see Chapter 8).

d) Approximate the values in part a) using cubic splines. Apply formulae (13.38) and (13.37) to approximate delta and gamma, respectively.

e) Compare the solutions in parts b), c) and d) by displaying them in Excel using the software from Chapter 14. Do you see major differences in the accuracy produced by the different solutions?

f) Compare the relative run-time performance of the methods. Which one performs best?

3. (Computing Option Sensitivities/Greeks Part II, Small Project)

This exercise is in a sense a mirror image of Exercise 2 except that in this we use the array of option prices that are generated by the finite difference method instead of the analytical option price as used in Exercise 2 (Chapters 20 to 25).

Answer the following questions:

a) Create an array of stock prices and compute the corresponding array of option prices for each stock price (keeping the other parameters constant). Use the Crank–Nicolson method to generate the array of option prices.

b) Compute the arrays containing the delta and gamma values for each stock price. Use divided differences for delta and gamma (see Chapter 8). Experiment with forward, backward and centred differencing.

c) Approximate the values in part a) using cubic splines. Use formulae (13.38) and (13.37) to approximate delta and gamma, respectively.

d) Compare the accuracy and run-time performance of the methods in this exercise with those from Exercise 2.

4. (Quiz and Brainstorming, Review Questions)

We used some C++11 syntax in Section 15.3 (we have already discussed the syntax in this book). Specifically, we used the following keywords in the code: `final`, `override`, `virtual`, `default` and `explicit`.

Answer the following questions:

a) Why are they being used and what are the consequences of using them?

b) How would the same goals from part a) be realised in older versions of C++?

c) How useful are lambda functions compared to traditional function objects, free functions and member functions in terms of readability and maintainability?

5. (Creating Code)

Fill in the code that is needed for the producer–consumer functions that use an initialiser list in Section 15.5.4. Test your code.

6. (Alternative Design for a Generic Distribution Class, Brainstorming)

In Section 15.7 we created a generic *adapter* class for Boost statistical distributions. The Boost online documentation proposes a similar design of non-member functions:

```
template <class RealType>
class any_distribution
{
public:
   template <class Distribution>
      any_distribution(const Distribution& d);
};

// Get the cdf of the underlying distribution:
   template <class RealType>
      RealType cdf(const any_distribution<RealType>& d, RealType x);
// etc....
```

Answer the following questions:

a) Create the above class and test it using the same examples that we used in this chapter.

b) Compare the two approaches in terms of ease of use, robustness and maintainability.

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 16

Bivariate Statistical Distributions and Two-Asset Option Pricing

16.1 INTRODUCTION AND OBJECTIVES

In this chapter we continue our discussion of statistical distributions that we introduced in Chapter 15. We create efficient algorithms to compute the *cumulative bivariate normal distribution* (BVN) function defined by:

$$M(a,b;\rho) = \int_{-\infty}^{a}\int_{-\infty}^{b} f(x,y;\rho)dxdy \tag{16.1}$$

where in this case:

$$f(x,y;\rho) = \frac{1}{2\pi\sqrt{1-\rho^2}}\exp\left[-\frac{x^2-2\rho xy+y^2}{2(1-\rho^2)}\right],\ -1<\rho<1.$$

This distribution has many applications and our interest is in using it to price options on two assets (see, for example, Haug, 2007). Contrary to the univariate case there seems to be little software available to compute bivariate distributions and in these cases we resort to a combination of third-party libraries and proprietary code. We propose a number of alternatives and the final choice will be determined by a number of requirements such as efficiency, accuracy, robustness and maintainability:

- S1: The Drezner method (Drezner, 1978).
- S2: The Genz method (Genz, 2004) and its implementation in *Quantlib* (www.quantlib.org) and by West (West, 2004).
- S3: The formula 26.3.3 in Abramowitz and Stegun (1965) in combination with a Gauss–Legendre numerical quadrature scheme.
- S4: Converting the two-dimensional integral (16.1) defining the cumulative bivariate normal distribution into a *hyperbolic* PDE that we approximate using the finite difference method.

Summarising, the main goal of this chapter is to propose a number of algorithms to compute the bivariate normal distribution and to compare their relative efficiency, accuracy and usability characteristics. In this way we choose the specific algorithm from S1 to S4 that best suits our requirements in a given context. We apply these algorithms to price two-factor plain options. The code is implemented using *callable objects* that the option pricing code uses as flexible *plug-ins* (*mixins*), thus allowing us to switch between implementations.

16.2 COMPUTING INTEGRALS USING PDEs

Looking at equation (16.1), we might naively suspect that we can compute the double integral either exactly or by using numerical quadrature. There seems to be a general consensus that an exact solution is not possible in this case, and computational efficiency and accuracy will be a major concern when we decide which numerical quadrature scheme to use. We discuss this topic in later sections.

In this section we take a completely different approach by remarking that certain kinds of integrals can be written as a differential equation and vice versa. These two approaches complement each other, as we shall see when we discuss ordinary differential equations and their applications in Chapters 24 and 25. In the current context we can differentiate equation (16.1) with respect to the variables a and b (which we immediately change to x and y, respectively) to produce a partial differential equation of *hyperbolic type*:

$$\frac{\partial^2 u}{\partial x \partial y} = f(x, y; \rho). \tag{16.2}$$

This is an example of a hyperbolic PDE and it can be seen as a second-order wave equation. In fact, the transformation:

$$\xi = (x + y)/2, \quad \eta = (x - y)/2 \tag{16.3}$$

reduces equation (16.2) to a PDE of the form:

$$\frac{\partial^2 u}{\partial \xi^2} - \frac{\partial^2 u}{\partial \eta^2} = f. \tag{16.4}$$

Both equations (16.2) and (16.4) are special cases of a *Goursat* PDE (Courant and Hilbert, 1968). A full treatment is beyond the scope of this book because the goal is to use the corresponding techniques to compute the integral in equation (16.1). To this end, we realise that the boundary value problem associated with equation (16.2) is different from that used for parabolic PDEs as introduced in Chapter 13. In the current case we model the problem as a wave travelling in a given direction from a start point. In other words, information is travelling from a single part of the boundary and not from the full boundary, as is the case with parabolic problems. We truncate the infinite domain of integration to a rectangle $(AL, x) \times (BL, y)$ where $AL = BL = -8$ and x and y are the values where we wish to compute the integral (16.1). This leads us to the lower boundary values:

$$u(AL, y) = u(x, BL) = 0, \quad AL \leq x < \infty, \quad BL \leq Y < \infty \tag{16.5}$$

This approach uses *domain truncation*. See Exercise 1 for an alternative approach based on *domain transformation*.

Summarising, equations (16.2) and (16.5) are the defining equations and starting point for the finite difference method that we employ in Section 16.2.1.

Some of the use cases in which this approach can be applied are:

- U1: It is robust, accurate and we can tune the desired level of accuracy for a range of arithmetic types such as `double`, `float` and multiprecision data types.
- U2: The algorithm is easy to program and it does not need third-party libraries with the exception of Boost matrices. It is easy to modify the code to work with user-defined matrix classes.
- U3: We use it as a benchmark to test the accuracy of other algorithms, for example the algorithm in Genz (2004) and the corresponding implementations in West (2004) and *Quantlib*.
- U4: The algorithms compute a matrix of values at all discrete mesh points up to and including the point where the value is desired. We can then use this matrix as a *lookup table* in combination with linear and cubic spline interpolation as discussed in Chapter 13. The matrix is computed once.
- U5: The algorithm can be applied to any bivariate distribution as well as to trivariate distributions without our having to carry out extensive and complex mathematical analysis.
- U6: The algorithm uses methods from PDE and FDM theory that we discuss in Chapters 20 to 25.

In general, the requirements for a given problem will determine which particular algorithm to use.

16.2.1 The Finite Difference Method for the Goursat PDE

We now discuss the application of the finite difference method to compute an approximation to the solution of system (16.2) and (16.5). We define discrete meshes in the x and y directions as described in Chapter 13. We store the discrete solution as a Boost matrix. The resulting second-order finite difference scheme is given by:

$$\frac{V_{i,j} - V_{i,j-1} - V_{i-1,j} + V_{i-1,j-1}}{h_x h_y} = f_{i-1/2,j-1/2} \tag{16.6}$$

where:

$$f_{i-1/2,j-1/2} = f(x_{i-1} + h_x/2,\ y_{j-1} + h_y/2), \quad 1 \le i \le I, \quad 1 \le j \le J,$$

and h_x and h_y are the constant mesh sizes in the x and y directions, respectively.

For readability reasons, we show how the core algorithm in this scheme is implemented in C++. Notice that we have rearranged equation (16.6) by placing all known terms on the right-hand side:

```
ublas::matrix<T> Values(NX + 1, NY + 1, 0.0);

T hxy = hx*hy;
```

```
for (std::size_t j = 1; j < Values.size2(); ++j)
{
   for (std::size_t i = 1; i < Values.size1(); ++i)
   {
      Values(i,j)= Values(i, j-1) + Values(i-1, j) - Values(i-1, j-1)
            + hxy*(fun(xarr[i-1] + hx/2, yarr[j-1] + hy/2));
   }
}
```

We have given this snippet of code to show how the finite difference scheme maps to code. The scheme is based on the second-order scheme in Aziz and Hubbard (1964).

16.2.2 Software Design

We now map the finite difference algorithm (16.6) to C++. We make a number of design choices relating to how clients interact with the software. In the present case we distinguish between a distribution's parameters (such as correlation) and the x and y values where we wish to compute the integral. The two design extremes are first to create a function object with the correlation parameter as data member and that uses the following function call syntax:

```
T operator () (T x, T y) const // T is data type
```

This is the approach taken in *Quantlib*, as the following code snippet shows:

```
double rho = -0.411002;
QuantLib::BivariateCumulativeNormalDistributionWe04DP nor(rho);
auto genz2 = nor(a, b);
```

The second choice is to have the correlation parameter as input argument to a free function as in the approach taken in West (2004):

```
auto genz1 = bivarcumnorm(a, b, rho);
```

Each approach has its advantages and disadvantages. The class that implements the finite difference scheme (16.6) is:

```
template <typename T>
   class GoursatFdm
{
private:
   // Domain
   T AL, BL; // left/lower boundaries
   std::vector<T> xarr, yarr;
   Function<T> fun;

public:
   GoursatFdm(T xLower, T yLower, const Function<T>& function,
         const std::vector<T>& xMesh, const std::vector<T>& yMesh)
         : AL(xLower), BL(yLower), fun(function),xarr(xMesh), yarr(yMesh) {}
```

```
        T operator () (T x, T y) const // T is data type
        {
           std::size_t NX = xarr.size()-1;
           std::size_t NY = yarr.size()-1;
           T hx = std::abs(x - AL) / static_cast<T>(NX);
           T hy = std::abs(y - BL) / static_cast<T>(NY);

           // Matrix to hold results, boundary conditions == 0
           ublas::matrix<T> Values(NX + 1, NY + 1, 0.0);

           T hxy = hx*hy;


           for (std::size_t j = 1; j < Values.size2(); ++j)
           {
            for (std::size_t i = 1; i < Values.size1(); ++i)
            {
                 Values(i,j)= Values(i,j-1)+Values(i-1,j)-Values(i-1, j-1)
                        + hxy*(fun(xarr[i-1]+hx/2,yarr[j-1]+hy/2));
            }
           }

           return Values(Values.size1() - 1, Values.size2() - 1);
        }
};
```

We see that this class can be used with a range of bivariate distributions. It is necessary to know how to truncate the domain of integration in such a way that accuracy is not adversely affected. In the case of the normal distribution it is relatively easy to compute the boundaries of the truncated domain but in general it may not be so easy. An alternative is to transform the domain of integration to the bounded domain $[-1, 1] \times [-1, 1]$ as discussed in Exercise 1.

16.2.3 Richardson Extrapolation

This is a technique to improve the accuracy of a numerical scheme. In this case we improve the accuracy of scheme (16.6) from second order to fourth order. The following code achieves this end:

```
template <typename T>
   class GoursatFdmExtrapolation
{
private:
      // Domain
      T AL, BL; // left/lower boundaries
      Function<T> fun;
      std::size_t N1, N2;
```

```
public:
      GoursatFdmExtrapolation(T xLower, T yLower,
                              const Function<T>& function,
                              std::size_t NX, std::size_t NY)
         : AL(xLower), BL(yLower), fun(function),N1(NX), N2(NY) {}

      T operator () (T x, T y) const
      {
         std::vector<T> xarr = CreateMesh(N1, AL, x);
         std::vector<T> yarr = CreateMesh(N2, BL, y);

         std::vector<T> xarr2 = CreateRefinedMesh(xarr);
         std::vector<T> yarr2 = CreateRefinedMesh(yarr);

         GoursatFdm<T> fdm(AL, BL, fun, xarr, yarr);
         GoursatFdm<T> fdm2(AL, BL, fun, xarr2, yarr2);

         double v1 = fdm(x,y);
         double v2 = fdm2(x,y);

         return (4.0*v2 - v1) / 3.0;
      }
};
```

This code can be optimised for better performance but our main concern is in using it to test the accuracy of other schemes and as a check that scheme (16.6) has indeed second-order accuracy. In practice, we choose the Genz algorithm, but only when we wish to compute the integral (16.1) at a single point.

16.2.4 Test Cases

We give some examples of how to approximate the integral (16.1) by scheme (16.6) and its extrapolated version. We compare the values with the result produced by the Genz algorithm:

```
int main()
{
   using value_type = double;
   //using value_type = float;
   // Truncated lower boundaries
   value_type AL = -8.0; value_type BL = -8.0;

   // Meshes
   std::size_t NX = 200; std::size_t NY = 200;
   value_type a = 6.0; value_type b = 6.0;
   auto xarr = CreateMesh(NX, AL, a);
   auto yarr = CreateMesh(NY, BL, b);

   // Construct FDM object
   value_type rho = -0.1995;
   BVNFunction<value_type> bvn(rho);
   GoursatFdm<value_type> fdm(AL, BL, bvn, xarr, yarr);
```

```
    // Compute a value
    std::cout << "Value: " << std::setprecision(16) << fdm(a, b) << '\n';

    // Extrapolation
    GoursatFdmExtrapolation<value_type> fdm2(AL, BL, bvn, NX, NY);
    std::cout << "Value2: " << std::setprecision(12) << fdm2(a, b) << '\n';

    // Genz/West comparison
    bivarcumnorm(a, b, rho);
    std::cout << "Genz/West: " << bivarcumnorm(a, b, rho) << '\n';
}
```

The output from this program is:

```
Value: 0.9999999980415676
Value2: 0.999999998027
Genz/West: 0.999999998027
```

This code could be the basis for more extensive tests of efficiency and accuracy.

16.3 THE DREZNER ALGORITHM

We discuss this method for completeness. It approximates the integral (16.1) by first performing a change of variables to transform the domain of integration to a quarter plane (Drezner, 1978). Then the integral is approximated using *Gauss quadrature* (Stroud and Secrest, 1966). The method is further discussed in Haug (2007) and it is supported in the *Quantlib* library. An example of use in the latter case is:

```
double a = 0.0; double b = 0.0;
double rho = 0.0;
QuantLib::BivariateCumulativeNormalDistributionDr78 qlDrezner1978(rho);
std::cout << "*Drezner 1978 Quantlib 1.8 : " << std::setprecision(16)
          << qlDrezner1978(a, b) << '\n';
```

The Drezner method uses single precision and hence is less accurate than the Genz method. Furthermore, the Drezner method shows signs of inaccuracy near $\rho = \pm 1$.

16.4 THE GENZ ALGORITHM AND THE WEST/QUANTLIB IMPLEMENTATIONS

The Genz method seems to be the method of choice for approximating integral (16.1). The algorithm uses numerical integration to approximate a transformed version of the integral (16.1). The mathematical details are discussed in Genz (2004). We focus on two open-source implementations based on West (2004) and *Quantlib*. The West implementation uses a free function defined in header and code files while the *Quantlib* library implementation uses a function object.

We now compare the relative accuracy of the two Genz implementations, the finite difference scheme and the Drezner method. We tested the schemes a (very large) number of times

and tried to spot if there are any discrepancies between the different results. In order to have a fair trial we produced the input parameters using the random number generators in C++:

```
#include <random>

std::default_random_engine eng;
std::default_random_engine eng2;
std::default_random_engine eng3;
std::random_device rd;
eng.seed(rd());
eng2.seed(rd());
eng3.seed(rd());

// Generators for the parameters of BVN
std::uniform_real_distribution<double> uniformX(-8,8);     // x
std::uniform_real_distribution<double> uniformY(-8,8);     // y
std::uniform_real_distribution<double> uniformRho(-1,1);   // rho
```

We now create a loop in which we compute integral (16.1) by the four methods S1, S2, S3 and S4 as introduced in Section 16.1. We also monitor the maximum error between the West and *Quantlib* implementations, as well as between the West and finite difference implementations. The code is:

```
// Compare the values for different implementations of BVN
double AL = -8.0; double BL = -8.0;
std::size_t NX = 50; std::size_t NY = 50;

int N = 10;// 5'00'000;
double errorGenz = 0.0;
double errorFdm = 0.0;
double errorFdm2 = 0.0;
for (int n = 1; n <= N; ++n)
{
   double a = uniformX(eng);
   double b = uniformY(eng2);
   double rho = uniformRho(eng3);
   std::cout << std::setprecision(8)<< "\na, b, rho: "
             << a << "," << b << ", " << rho << '\n';

   double val1 = bivarcumnorm(a, b, rho);
   std::cout << "*Genz West                  : "
             << std::setprecision(16) << val1 << '\n';

   QuantLib::BivariateCumulativeNormalDistributionWe04DP nor(rho);
   double val2 = nor(a, b);
   std::cout << "*Genz QuantLib 1.8          : "
             << std::setprecision(16) << val2 << '\n';
   errorGenz = std::max<double>(errorGenz, std::abs(val1 - val2));

   // Drezner for good measure
   QuantLib::BivariateCumulativeNormalDistributionDr78 qlDrezner1978(rho);
```

```
    std::cout << "\n*Drezner 1978 Quantlib 1.8: "
              << std::setprecision(16)<<qlDrezner1978(a,b) << '\n';

    // FDM solution
    BVNFunction<double> bvn(rho);
    auto xarr = CreateMesh(NX, AL, a);
    auto yarr = CreateMesh(NY, BL, b);
    GoursatFdm<double> fdm(AL, BL, bvn, xarr, yarr);
    double val3 = fdm(a, b);
    std::cout << "*FDM approx                 : "
              << std::setprecision(16) << val3 << '\n';
    errorFdm = std::max<double>(errorFdm, std::abs(val1 - val3));

    // Extrapolated FDM
    GoursatFdmExtrapolation<double> fdm2(AL, BL, bvn, NX, NY);
    double val4 = fdm2(a, b);
    std::cout << "FDM approx extrap           : "
              << std::setprecision(12) << val4 << '\n';
    errorFdm2 = std::max<double>(errorFdm2, std::abs(val1 - val4));
}

std::cout << "\nMax error West/QL: " << errorGenz << '\n';
std::cout << "Max error Fdm/West: " << errorFdm << '\n';
std::cout << "Max error Fdm Extrap/West: " << errorFdm2 << '\n';
```

Typical output is:

```
a, b, rho: 0.88323078,1.0534402, -0.35704502
*Genz West                  : 0.6746688773191766
*Genz QuantLib 1.8          : 0.6746688773191766

*Drezner 1978 Quantlib 1.8  : 0.6746688163961295
*FDM approx                 : 0.6752241640580708
FDM approx extrap           : 0.674668544805

a, b, rho: -6.3771756,-0.2349262, -0.19490095
*Genz West                  : 5.617457071256493e-12
*Genz QuantLib 1.8          : 5.617457076309991e-12

*Drezner 1978 Quantlib 1.8  : 5.451316734811666e-12
*FDM approx                 : 5.588220577826419e-12
FDM approx extrap           : 5.61742692845e-12

a, b, rho: -3.6033799,5.1965627, 0.69074304
*Genz West                  : 0.000157052961665832
*Genz QuantLib 1.8          : 0.000157052961665489

*Drezner 1978 Quantlib 1.8  : 0.000157052961665489
*FDM approx                 : 0.000156352775084809
FDM approx extrap           : 0.000157052567171
```

```
a, b, rho: -1.8787828,-5.8789232, 0.3055131
*Genz West                    : 1.003499500688108e-09
*Genz QuantLib 1.8            : 1.003499502515318e-09

*Drezner 1978 Quantlib 1.8    : 9.854280102848721e-10
*FDM approx                   : 1.001043599137774e-09
FDM approx extrap             : 1.00349809494e-09

a, b, rho: -0.32177054,-2.9262971, -0.46594091
*Genz West                    : 3.560500239620657e-05
*Genz QuantLib 1.8            : 3.560500239620712e-05

*Drezner 1978 Quantlib 1.8    : 3.555883404780285e-05
*FDM approx                   : 3.511401126791778e-05
FDM approx extrap             : 3.56043363362e-05

a, b, rho: 4.7155323,7.8416032, -0.69093606
*Genz West                    : 0.9999987946014995
*Genz QuantLib 1.8            : 0.9999987946014997

*Drezner 1978 Quantlib 1.8    : 0.9999987946014997
*FDM approx                   : 0.9999988671747321
FDM approx extrap             : 0.999998795263

a, b, rho: -1.946981,3.1876528, 0.054292586
*Genz West                    : 0.02575683532947779
*Genz QuantLib 1.8            : 0.02575683532947772

*Drezner 1978 Quantlib 1.8    : 0.02575684354544463
*FDM approx                   : 0.02568591162309218
FDM approx extrap             : 0.025756830355

a, b, rho: -4.7346574,3.0501621, 0.948175
*Genz West                    : 1.097127038323874e-06
*Genz QuantLib 1.8            : 1.097127038329759e-06

*Drezner 1978 Quantlib 1.8    : 1.097127038329759e-06
*FDM approx                   : 1.092586880408237e-06
FDM approx extrap             : 1.09712429482e-06

a, b, rho: 7.2957037,-7.3891475, -0.80360421
*Genz West                    : 7.245942013444848e-14
*Genz QuantLib 1.8            : 7.245942070366185e-14

*Drezner 1978 Quantlib 1.8    : 7.260858581048524e-14
*FDM approx                   : 7.194843962877875e-14
FDM approx extrap             : 7.19003890218e-14

a, b, rho: -2.726639,4.9965896, 0.17220467
*Genz West                    : 0.003199150045498287
*Genz QuantLib 1.8            : 0.003199150045498298
```

```
*Drezner 1978 Quantlib 1.8  : 0.003199150046013107
*FDM approx                 : 0.003186916800067802
FDM approx extrap           : 0.00319914563772

Max error West/QL: 1.11022302463e-16
Max error Fdm/West: 0.000555286738894
Max error Fdm Extrap/West: 3.32514473067e-07
```

We notice that the finite difference method gives between four and six digits accuracy even when $NX = NY = 50$.

We can draw the following conclusions based on our experiments:

- The West and *Quantlib* implementations of the Genz algorithms give the same accuracy up to machine precision. We used the finite difference method to check them to ensure that they do not give the same incorrect output for certain values of the parameters.
- The added value of the finite difference approach is that it can be tuned to suit our accuracy requirements. The scheme is second-order accurate and we can use Richardson extrapolation to achieve fourth-order accuracy. Furthermore, these schemes deliver a matrix as output which saves recomputing for certain classes of applications.
- The finite difference approach can be applied to a wide range of bivariate and trivariate distributions, for example the *bivariate-t distribution* and the *trivariate normal distribution*.
- The Drezner (1978) algorithm is less accurate than the other schemes and it degrades when $\rho = \pm 1$.

We now discuss two more algorithms to compute (16.1) for completeness.

16.5 ABRAMOWITZ AND STEGUN APPROXIMATION

A number of methods to compute (16.1) are given in Abramowitz and Stegun (1965). We focus on the specific cases ('AS 26.3.3') defined by the nested integral:

$$M(a, b; \rho) = \int_{-\infty}^{a} Z(x)dx \int_{-\infty}^{w(x)} Z(t)dt \tag{16.7}$$

where:

$$Z(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$$

and

$$w(x) = \frac{b - \rho x}{\sqrt{1 - \rho^2}}.$$

We conjecture that this formulation might be easier to compute than other methods. To this end, we write (16.7) in the form:

$$M(a, b; \rho) = \int_{-\infty}^{a} Z(x)N(w(x))dx \tag{16.8}$$

where in general:

$$N(t) = \int_{-\infty}^{t} Z(s)ds.$$

Assuming that we have a way to compute the univariate cumulative function $N(t)$ (and we do), we can apply some of the numerical quadrature schemes to approximate the integral in (16.8). In this case we use the second-order *midpoint rule* and the application of repeated Richardson extrapolation to achieve higher-order accuracy. First, we define the following functions:

```
// Normal variates etc.
double Pdf(double x)
{
   const double A = 1.0 / std::sqrt(2.0 * 3.141592653589793238462);
   return A * std::exp(-x*x*0.5);
}

// C++11 supports the error function
auto cndN = [](double x)
    { return 0.5 * (1.0 - std::erf(-x / std::sqrt(2.0))); };

double Cdf(double x)
{ // The approximation to the cumulative normal distribution

   return cndN(x);
}
```

The code to apply the midpoint rule to (16.8) is:

```
double BivariateNormal(double a, double b, double rho,
          double ALower, double BLower, long NX)
{ // 26.3.3 Abramowitz and Stegun

   double res = 0.0;
   double hx = (a - ALower) / NX;

   const double fac = 1.0 / std::sqrt(1 - rho*rho);
   double s = ALower + (0.5*hx);
   double hx2 = hx / 2;
   double x = s;
   double w;

   for (long n = 0; n < NX; n += 1)
   {
      w = fac*(b - rho*(x));
      res += hx*Pdf(x)*Cdf(w);
      x += hx;
   }
```

```
    return res;
}
```

and the code for repeated Richardson extrapolation is given by:

```
double BivariateNormalExtrapolate(double a, double b, double rho,
                                  double ALower, double BLower, long NX)
{ // 26.3.3 Abramowitz and Stegun

    // Repeated Richardson extrapolation
    double uh = BivariateNormal(a, b, rho, ALower, BLower, NX);
    double uh2 = BivariateNormal(a, b, rho, ALower, BLower, 2 * NX);
    auto vh = (4 * uh2 - uh) / 3.0;

    double uh4 = BivariateNormal(a, b, rho, ALower, BLower, 4 * NX);
    auto vh2 = (4 * uh4 - uh2) / 3.0;

    return (16 * vh2 - vh) / 15;

    return vh;
}
```

Finally, we create a test program to show how to use the algorithm on some of the examples in Section 16.4:

```
int main()
{
    // Range of values to test with
    //double a = 0.88323078; double b = 1.0534402; double rho =
    //-0.35704502;
    //double a = -3.6033799; double b = 5.1965627; double rho = 0.69074304;
    //double a = -6.3771756; double b = 0.2349262; double rho =
    //-0.19490095;
    double a = -1.946981; double b = 3.1876528; double rho = 0.054292586;
    //double a = 6.088323078; double b = 6.0534402; double rho =
    //0.999935704502;

    long N = 20; // Number of subdivisions
    double AL = -8.0; double BL = -8.0;

    std::cout << "*A&S 26.3.3   : " << std::setprecision(16)
              << BivariateNormal(a, b, rho, AL, BL, N) << '\n';

    std::cout << "*A&S 26.3.3 Extrapolated   : " << std::setprecision(16)
              << BivariateNormalExtrapolate(a, b, rho, AL, BL, N) << '\n';

    return 0;
}
```

We introduce the 26.3.3 example mainly for completeness.

16.6 PERFORMANCE TESTING

For some use cases we may wish to compute a matrix of values of the BVN distribution, for example when we wish to precompute these values, in an extended option pricing program or computing its *quantiles* (*isolines*). We can do this by using the Genz algorithm:

```
ublas::matrix<double> GenzWest(double a, double b, double rho,
          const std::vector<double>& x,
          const std::vector<double>& y)
{ // Matrix of integral values at mesh points

   // Matrix to hold results
   double bdyValues = 0.0;
   ublas::matrix<double> Values(x.size(), y.size(), bdyValues);

   for (std::size_t i = 0; i < Values.size1(); ++i)
   {
      for (std::size_t j = 0; j < Values.size2(); ++j)
      {
         Values(i, j) = bivarcumnorm(x[i], y[j], rho);
      }
   }

   return Values;
}
```

The other option is to use the finite difference approach because it produces a matrix of values *for free* as it were, as part of the algorithm. We have adapted the corresponding class `Gonrat` FDM to save the generated matrix as a public data member:

```
   public:
      // Matrix to hold results, boundary conditions == 0
      ublas::matrix<T> Values;
```

We compare the two approaches with regard to efficiency and accuracy. Genz accuracy is not customisable but the accuracy of the finite difference method can be customised by increasing or decreasing the number of mesh points. A test program is:

```
   int main()
   {
      using value_type = double;

      std::size_t NX = 150;        std::size_t NY = 150;
      double AL = -8.0; double BL = -8.0;
      value_type a = 6.0; value_type b = 6.0; value_type rho = 0.5;
      auto xarr = CreateMesh(NX, AL, a);
      auto yarr = CreateMesh(NY, BL, b);

      StopWatch<> sw;
      sw.Start();
```

```
    BVNFunction<value_type> bvn(rho);
    GoursatFdm<value_type> fdm(AL, BL, bvn, xarr, yarr);
    auto val = fdm(a, b);
    sw.Stop();
    std::cout << "FDM: " << sw.GetTime() << '\n';

    StopWatch<> sw2;
    sw2.Start();
    auto mat2 = GenzWest(a, b, rho, xarr, yarr);
    sw2.Stop();
    std::cout << "Genz West: " << sw2.GetTime() << '\n';

    auto mat3 = mat2 - fdm.Values;
    double norm = 0.0;

    // Are the two solutions 'close'
    for (std::size_t i = 0; i < mat3.size1(); ++i)
    {
       for (std::size_t j = 0; j < mat3.size2(); ++j)
       {
          norm = std::max<double>(norm, std::abs(mat3(i, j)));
       }
    }
    std::cout << "Error between Genz and FDM: " << norm << '\n';
}
```

In general, we can make the finite difference solution as close to the Genz solution as we wish but the Genz algorithm is slower than the former for this particular use case.

16.7 GAUSS–LEGENDRE INTEGRATION

We include a discussion of this technique because it is popular in finance and it is used in several algorithms to compute integral (16.1). An *n-point Gaussian quadrature rule* is a quadrature rule constructed to yield an exact result for *polynomials* of degree $2n - 1$ or less by a suitable choice of the points (abscissae) x_j and weights w_j for $j = 1, \ldots, n$. The general formulation is:

$$\int_a^b f(x)dx = \sum_{j=1}^{\infty} w_j f(x_j) \simeq \sum_{j=1}^{n} w_j f(x_j) \tag{16.9}$$

where w_j = weights, x_j = abscissae, $j = 1, \ldots, n$.

Strictly speaking, Gaussian–Legendre integration is only defined on the interval $[-1, 1]$ but a transformation allows it to be used for a general interval (a, b):

$$\int_a^b f(x)dx = \frac{b-a}{2}\int_{-1}^{1} f\left(\frac{b-a}{2}x + \frac{b+a}{2}\right)dx \simeq \frac{b-a}{2}\sum_{j=1}^{n} w_j f\left(\frac{b-a}{2}x_j + \frac{b+a}{2}\right). \tag{16.10}$$

In the case of 'AS 26.3.3' (equation (16.8)) we get the quadrature scheme:

$$M(a,b,\rho) \cong \int_{AL}^{a} Z(x)N(w(x))dx \simeq \left(\frac{a-AL}{2}\right)\sum_{j=1}^{n} w_j Z(\tilde{x}_j)\, N(w(\tilde{x}_j)) \qquad (16.11)$$

where $\tilde{x}_j = \frac{a-AL}{2}x_j + \frac{a+AL}{2}$, $j = 1, \ldots, n$ and AL = truncation point.

We have implemented equation (16.11) by the following free function (notice that we use *loop unrolling* to improve performance):

```
double BivariateNormalGaussLegendre(double a, double b, double rho,
double ALower)
{ // 26.3.3 Abramowitz and Stegun

   // 20-point Gauss Legendre rule
   // Nodes
   const std::vector<double> x = { -0.0765265211334973,
         0.0765265211334973, -0.2277858511416451,
         0.2277858511416451, -0.3737060887154195,
         0.3737060887154195, -0.5108670019508271,
         0.5108670019508271, -0.6360536807265150,
         0.6360536807265150, -0.7463319064601508,
         0.7463319064601508, -0.8391169718222188,
         0.8391169718222188, -0.9122344282513259,
         0.9122344282513259, -0.9639719272779138,
         0.9639719272779138, -0.9931285991850949,
         0.9931285991850949 };

   // Weights
   const std::vector<double> w = { 0.1527533871307258,
         0.1527533871307258, 0.1491729864726037,
         0.1491729864726037, 0.1420961093183820,
         0.1420961093183820, 0.1316886384491766,
         0.1316886384491766, 0.1181945319615184,
         0.1181945319615184, 0.1019301198172404,
         0.1019301198172404, 0.0832767415767048,
         0.0832767415767048, 0.0626720483341091,
         0.0626720483341091, 0.0406014298003869,
         0.0406014298003869, 0.0176140071391521,
         0.0176140071391521 };

   // Using Gauss Legendre 20-point method
   long N = 20;

   double res = 0.0;
   double alpha = (a - ALower) / 2.0;
   double beta = (a + ALower) / 2.0;

   const double fac = 1.0 / std::sqrt(1 - rho*rho);
   double xl, wl;
```

```
    // Using loop unrolling for (possible) improved performance
    for (long i = 0; i < N; i += 4)
    {
       xl = alpha*x[i] + beta;
       wl = fac*(b - rho*(xl));
       res += w[i] * Pdf(xl)*Cdf(wl);

       xl = alpha*x[i + 1] + beta;
       wl = fac*(b - rho*(xl));
       res += w[i + 1] * Pdf(xl)*Cdf(wl);

       xl = alpha*x[i + 2] + beta;
       wl = fac*(b - rho*(xl));
       res += w[i + 2] * Pdf(xl)*Cdf(wl);

       xl = alpha*x[i + 3] + beta;
       wl = fac*(b - rho*(xl));
       res += w[i + 3] * Pdf(xl)*Cdf(wl);
    }

    return alpha*res;
}
```

This code implements the 20-node Gauss–Legendre scheme. We have already discussed other variants in Chapter 8. A test program is:

```
double a = -1.946981; double b = 3.1876528; double rho = 0.054292586;
double AL = -8.0; double BL = -8.0;

std::cout << "*A&S 26.3.3 Gauss Legendre   : " << std::setprecision(16)
          << BivariateNormalGaussLegendre(a, b, rho, AL) << '\n';
```

Gauss–Legendre integration plays an important role in Genz (2004). An attention point is to choose a large number of Gauss points when the correlation is close to 1 in absolute value, while six points are sufficient when the correlation is less than 0.3; 12 points are needed when the correlation is between 0.3 and 0.75 in order to achieve machine precision.

An interesting discussion on polynomial-based interpolation and quadrature is given in Trefethan (2011).

16.8 APPLICATIONS TO TWO-ASSET PRICING

One of the applications of the bivariate normal distribution is to use it as part of analytical formulae for two-asset plain option pricing formulae. It is outside the scope of this book to discuss them in detail. Instead, we take some examples and program the formulae in C++. We also compare the generated option prices with those generated using VBA in Haug (2007).

We focus on four cases as motivation to show how to use the code:

- Two-asset correlation options.
- Minimum of two risky assets (Haug, 2007, p. 211).
- Best-or-worst cash-or-nothing (from Haug, 2007, p. 224).
- Two-asset cash-or-nothing options (four types) (Haug, 2007, p. 221).

The resulting C++ code is relatively straightforward and it is a direct mapping from the algorithms in Haug (2007). The added value in this context is that we have several efficient and accurate methods to compute integral (16.1). We show the code without further comment as it is relatively straightforward:

```
int main()
{

   double a = 8.5; double b = 8.5; double Rho = 0.0;
   std::cout << M(a,b,Rho) << std::endl;
   {
      // 2-asset correlation option (Haug 2007)
      double S1 = 52.0; double S2 = 65.0;
      double T = 0.5;
      double K1 = 50.0; double K2 = 70.0;
      double s1 = 0.2; double s2 = 0.3;
      double r = 0.1;
      double b1 = 0.1; double b2 = 0.1;
      double rho = 0.75;

      // Other variables
      double y1 = (std::log(S1/K1) + (b1 - s1*s1/2)*T) / (s1*sqrt(T));
      double y2 = (std::log(S2/K2) + (b2 - s2*s2/2)*T) / (s2*sqrt(T));

      // 2-asset correlation options
      double call = S2*std::exp((b2-r)*T)*M(y2 + s2*sqrt(T), 1
                  + rho*s2*sqrt(T), rho)
                  - K2*std::exp(-r*T)*M(y2,y1,rho);
      std::cout << "Call correlation: " << call << std::endl;

      double put = K2*std::exp(-r*T)*M(-y2, -y1, rho)
                 - S2*std::exp((b2-r)*T)*M(-y2-s2*sqrt(T),
                 -y1-rho*s2*sqrt(T),rho);
      std::cout << "Put correlation: " << put << std::endl;

   }


   // Minimum of two risky assets (Haug 2007, page 211)
   {
   /* double S1 = 100.0; double S2 = 105.0;
```

```
    double T = 0.5;
    double K = 98.0;
    double s1 = 0.11; double s2 = 0.16;
    double r = 0.05;
    double b1 = 0; double b2 = 0; // Investigate b < 0
    double rho = 0.63;*/

    double S1 = 85.0; double S2 = 60.0;
    double T = 2.0;
    double K = 100.0;
    double s1 = 0.40; double s2 = 0.25;
    double r = 0.08;
    double b1 = r; double b2 = r; // Investigate b < 0
    double rho = -0.70;


    // Other variables
    double s = std::sqrt(s1*s1 + s2*s2 - 2.0*rho*s1*s2);
    double d = (std::log(S1/S2) + (b1 - b2 + s*s/2)*T)
        / (s*std::sqrt(T));

    double y1 = (std::log(S1/K) + (b1 + s1*s1/2)*T) / (s1*sqrt(T));
    double y2 = (std::log(S2/K) + (b2 + s2*s2/2)*T) / (s2*sqrt(T));

    double rho1 = (s1 - rho*s2)/s;
    double rho2 = (s2 - rho*s1)/s;

    // As a function of S1,S2,K,T
    double cMinK =
       S1*std::exp((b1-r)*T)*M(y1,-d,-rho1)
       + S2*std::exp((b2-r)*T)*M(y2,d-s*std::sqrt(T), -rho2)
         - K*std::exp(-r*T)
       *M(y1 - s1*std::sqrt(T), y2 - s2*std::sqrt(T), rho);
    std::cout << "Call min K: " << cMinK << std::endl;

    double cMaxK =
                S1*std::exp((b1-r)*T)*M(y1,d,rho1)
                + S2*std::exp((b2-r)*T)*M(y2,-d+s*std::sqrt(T), rho2)
                - K*std::exp(-r*T)*(1.0 - M(-y1 + s1*std::sqrt(T),-y2
                + s2*std::sqrt(T), rho));
    std::cout << "Call max K: " << cMaxK << std::endl;
}

// Best-or-worst cash-or-nothing (Haug 2007 page 224)
{
    double S1 = 105.0; double S2 = 100.0;
    double T = 0.5;
    double Q = 5.0;
    double s1 = 0.3; double s2 = 0.2;
    double r = 0.08;
    double b1 = 0.06; double b2 = 0.02;
```

```
    double rho = 0.75;

    double K = 100.0;

    // Other variables
    double s = std::sqrt(s1*s1 + s2*s2 - 2.0*rho*s1*s2);
    double y = (std::log(S1/S2) + (b1 - b2 + s*s/2)*T)
        / (s*std::sqrt(T));

    double z1 = (std::log(S1/K) + (b1 + s1*s1/2)*T)
        / (s1*std::sqrt(T));
    double z2 = (std::log(S2/K) + (b2 + s2*s2/2)*T)
        / (s2*std::sqrt(T));


    double rho1 = (s1 - rho*s2)/s;
    double rho2 = (s2 - rho*s1)/s;

    // As a function of S1,S2,K,T
    double cBest = Q*std::exp(-r*T)*(M(y,z1,-rho1) + M(-y,z2,-rho2));
    std::cout << "Call worst cash-or-nothing best: "
              << cBest << std::endl;
}

// 2-asset cash-or-nothing options (4 types)(Haug 2007 page 221)
{
    double S1 = 100.0; double S2 = 100.0;
    double K1 = 110.0; double K2 = 90.0;
    double T = 0.5;
    double Q = 10.0;
    double s1 = 0.2; double s2 = 0.25;
    double r = 0.1;
    double b1 = 0.05; double b2 = 0.06;
    double rho = 0.5;

    double d11 = (std::log(S1/K1) + (b1 - s1*s1/2)*T)
          / (s1*std::sqrt(T));
    double d12 = (std::log(S1/K2) + (b1 - s1*s1/2)*T)
          / (s1*std::sqrt(T));
    double d21 = (std::log(S2/K1) + (b2 - s2*s2/2)*T)
          / (s2*std::sqrt(T));
    double d22 = (std::log(S2/K2) + (b2 - s2*s2/2)*T)
          / (s2*std::sqrt(T));

    double cType1 = Q*std::exp(-r*T)*M(d11,d22,rho);
    double cType2 = Q*std::exp(-r*T)*M(-d11,-d22,rho);
    double cType3 = Q*std::exp(-r*T)*M(d11,-d22,-rho);
    double cType4 = Q*std::exp(-r*T)*M(-d11,d22,-rho);

    std::cout << "Two-asset cash-or-nothing types: " << std::endl;
    std::cout << "Type 1: " << cType1 << std::endl;
```

```
        std::cout << "Type 2: " << cType2 << std::endl;
        std::cout << "Type 3: " << cType3 << std::endl;
        std::cout << "Type 4: " << cType4 << std::endl;
    }

    return 0;
}
```

We tested these algorithms using the Genz/West code:

```
double M(double a, double b, double rho)
{
    return bivarcumnorm(a, b, rho);
}
```

The output is:

```
Call correlation: 4.707330012666851
Put correlation: 3.909280147364093
Call min K: 0.0180004745810719
Call max K: 20.94494846250169
Call worst cash-or-nothing best: 2.793935852582142
Two-asset cash-or-nothing types:
Type 1: 2.498749451519521
Type 2: 2.156694921716584
Type 3: 0.2128123872130617
Type 4: 4.644037484557975
```

We now price these options using the extrapolated version of the finite difference scheme:

```
double M(double a, double b, double rho)
{

    using value_type = double;

    std::size_t NX = 50;    std::size_t NY = 50;
    value_type AL = -8.0; value_type BL = -8.0;
    auto xarr = CreateMesh(NX, AL, a);
    auto yarr = CreateMesh(NY, BL, b);

    BVNFunction<value_type> bvn(rho);
    GoursatFdmExtrapolation<value_type> fdm(AL, BL, bvn, NX, NY);

    return fdm(a, b);
}
```

The output in this case is:

```
Call correlation: 4.707325487726628
Put correlation: 3.909281409246201
```

```
Call min K: 0.01787816522478947
Call max K: 20.9451108618852
Call worst cash-or-nothing best: 2.793934246406374
Two-asset cash-or-nothing types:
Type 1: 2.498749988890371
Type 2: 2.156695455034736
Type 3: 0.2128125164628895
Type 4: 4.644034994433958
```

We increase the number of subintervals to 500 and the output in this case becomes:

```
Call correlation: 4.70733001221495
Put correlation: 3.909280147488811
Call min K: 0.01800046275512779
Call max K: 20.94494847804961
Call worst cash-or-nothing best: 2.793935852422795
Two-asset cash-or-nothing types:
Type 1: 2.498749451572514
Type 2: 2.156694921769651
Type 3: 0.2128123872261226
Type 4: 4.644037484309941
```

As expected, the accuracy of the finite difference schemes improves as we increase the number of subdivisions of the domain of integration. You can experiment with the code in order to test other two-asset problems.

16.9 TRIVARIATE NORMAL DISTRIBUTION

We conclude this chapter with a discussion on the computation of the *trivariate cumulative normal* distribution (called TVN for short) using both the Genz algorithm and the generalisation of the PDE (16.2) and finite difference scheme (16.6) to three dimensions. We discuss the implementation of the Genz algorithm as described in West (2004) and compare its accuracy with that produced by our finite difference scheme.

The trivariate normal distribution has its probability density function defined by:

$$N(b_1, b_2, b_3; \Sigma) = \frac{1}{(2\pi)^{3/2}} \int_{-\infty}^{b_1} \int_{-\infty}^{b_2} \int_{-\infty}^{b_3} f(x, y, z; \Sigma) dz dy dx \tag{16.12}$$

where Σ is the positive-definite covariance matrix and $|\Sigma|$ is its determinant:

$$\Sigma = \begin{pmatrix} 1 & \rho_{12} & \rho_{13} \\ \rho_{12} & 1 & \rho_{23} \\ \rho_{13} & \rho_{23} & 1 \end{pmatrix}$$

$$f(x, y, z; \Sigma) = \exp\left(-\frac{1}{2} w^{\mathrm{T}} \Sigma^{-1} w\right), \quad w = (x, y, z)^{\mathrm{T}}$$

and Σ^{-1} is the inverse of Σ.

We have implemented this function in C++ by explicitly computing the inverse of the covariance matrix. We also used a more explicit form for the probability density function:

$$f(b_1, b_2, b_3; \Sigma) = \frac{e^{-w} \Big/ \left[2\left(\rho_{12}^2 + \rho_{13}^2 + \rho_{23}^2\right) - 2\rho_{12}\rho_{13}\rho_{23}\right]}{2} \sqrt{2}\pi^{3/2} \sqrt{1 - \left(\rho_{12}^2 + \rho_{13}^2 + \rho_{23}^2\right) + 2\rho_{12}\rho_{13}\rho_{23}} \tag{16.13}$$

where:

$$w = b_1^2\left(\rho_{23}^2 - 1\right) + b_2^2\left(\rho_{13}^2 - 1\right) + b_3^2\left(\rho_{12}^2 - 1\right) + 2[b_1 b_2(\rho_{12} - \rho_{13}\rho_{23}) + b_1 b_3(\rho_{13} - \rho_{12}\rho_{23}) + b_2 b_3(\rho_{23} - \rho_{12}\rho_{13})].$$

The code for this function is:

```
double f(double x, double y, double z, double a12, double a13, double a23)
{
   double q = 2.0*(a12*a12 + a13*a13 + a23*a23 - 2.0*a12*a13*a23 - 1.0);

   // Covariance matrix must be positive semi-definite
   double tmp = 1.0 - (a12*a12 + a13*a13 + a23*a23) + 2.0*a12*a13*a23;
   if (tmp <= 0.0)
      return std::numeric_limits<double>::max();

   double fac = std::sqrt(tmp);

   double fac2 = 2.0*std::sqrt(2.0)*std::pow(3.14159, 1.5);
   // x = b, y = b2. Z = b3

   double w = x*x*(a23*a23 - 1.0) + y*y*(a13*a13 - 1.0) + z*z*
   (a12*a12 -1.0)
    2.0*(x*y*(a12 - a13*a23) + x*z*(a13 - a12*a23) + y*z*(a23 - a12*a13));

   return std::exp(-w / q) / (fac2 * fac);
}
```

When testing the finite difference code we used both forms for the probability density function when computing the integral in equation (16.12).

Similar to the two-dimensional case we can differentiate the integral to produce a *third-order hyperbolic* PDE:

$$\frac{\partial^3 u}{\partial x \partial y \partial z} = f. \tag{16.14}$$

We solve (16.14) by a finite difference method that is the three-dimensional analogue of scheme (16.6):

$$V_{i,j,k} = V_{i-1,j,k} + V_{i,j-1,k} - V_{i-1,j-1,k} + V_{i,j,k-1} - V_{i-1,j,k-1} - V_{i,j-1,k-1} + V_{i-1,j-1,k-1} + h_x h_y h_z f_{i-1/2,2,j-1/2,k-1/2} \tag{16.15}$$

where:

$$\begin{aligned} x_{i-1/2} &= x_{i-1} + h_{x/2} \\ y_{j-1/2} &= y_{j-1} + h_{y/2} \\ z_{k-1/2} &= z_{k-1} + h_{z/2}, \qquad 1 \le i \le NX,\ 1 \le j \le NY,\ 1 \le k \le NZ. \end{aligned}$$

You can create this scheme from first principles as we did in the two-dimensional case by composing divided differences in the x, y and z directions. Furthermore, we adopt the heuristic by defining the boundary values (similar to (16.5)):

$$U(AL, y, z) = u(x, BL, z) = u(x, y, CL)\ \forall\ -\infty < x, y, z < \infty \tag{16.16}$$

where AL, BL and CL are truncation values (typically $AL = BL = CL = -8$).

Since we are working in three-dimensional space we need a data structure to hold three-dimensional discrete data. To this end, we use the *Boost multi_array* structure (Demming and Duffy, 2010). We define the structure as follows:

```
#include <boost/multi_array.hpp>
const int DIM = 3; // 3d matrix
typedef boost::multi_array<double, DIM> Tensor;
typedef Tensor::index Index;
```

The code for scheme (16.15) consists of creating three mesh arrays, creating and initialising the *tensor* to hold the values to be generated and then updating the values by mapping scheme (16.15) to C++ code:

```
// Call this function once and then do table lookup at a given (x,y)
double IntegralValues(double b1, double b2, double b3,
                      double a21, double a31, double a32,
                      int NX, int NY, int NZ)
{ // Matrix of integral values at mesh points

   double hx = std::abs(b1-a1)/static_cast<double>(NX);
   double hy = std::abs(b2-a2)/static_cast<double>(NY);
   double hz = std::abs(b3-a3)/static_cast<double>(NZ);


   // Tensor to hold results
   Tensor Values;
   Values.resize(boost::extents [NX+1][NY+1][NZ+1]);

   // Create arrays of mesh points
   ublas::vector<double> xarr(NX+1, 0.0);
   xarr[0] = a1;
   for (std::size_t i = 1; i < xarr.size(); ++i) xarr[i] = xarr[i-1] + hx;

   ublas::vector<double> yarr(NY+1, 0.0);
   yarr[0] = a2;
   for (std::size_t i = 1; i < yarr.size(); ++i) yarr[i] = yarr[i-1] + hy;
```

```
    ublas::vector<double> zarr(NZ+1, 0.0);
    zarr[0] = a3;
    for (std::size_t i = 1; i < zarr.size(); ++i) zarr[i] = zarr[i-1] + hz;

    // Init the tensor
    for (Index i = 0; i != NX+1; ++i)
    {
       for (Index j = 0; j != NY+1; ++j)
       {
          for (Index k = 0; k != NZ+1; ++k)
          {
             Values[i][j][k] = 0.0;
          }
       }
    }

    for (Index i = 1; i != NX+1; ++i)
    {
       for (Index j = 1; j != NY+1; ++j)
       {
          for (Index k = 1; k != NZ+1; ++k)
          {
             Values[i][j][k] = Values[i - 1][j][k]
             + Values[i][j - 1][k] - Values[i - 1][j - 1][k]
             + Values[i][j][k - 1] - Values[i - 1][j][k - 1]
             - Values[i][j - 1][k - 1] + Values[i - 1][j-1][k - 1]

             // Value of RHS as centre of gravity)
             + hx*hy*hz*f(xarr[i-1] + hx/2, yarr[j-1] + hy/2,
                          zarr[k-1] + hz/2,
          }
       }
    }

    return Values[NX][NY][NZ];
}

double M(double b1, double b2, double b3,
         double a21, double a31, double a32,
         int NX, int NY, int NZ)
{ // The cumulative trivariate normal distribution at (b1,b2,b3,rho)

    double result = IntegralValues(b1,b2,b3, a21, a31, a32, NX,NY, NZ);

    return result;
}
```

An initial test case is:

```
std::cout << "Give the values of x, y and z" << std::endl;
double x, y, z;
```

```
std::cin >> x;
std::cin >> y;
std::cin >> z;
std::cout << "Give the values of rho12, rho13 and rho23" << std::endl;
double rho12, rho13, rho23;
std::cin >> rho12;
std::cin >> rho13;
std::cin >> rho23;
std::cout << std::setprecision(6)
          << "The value of TVN Genz(" << x << "," << y << "," << z
          << "," << rho12 << ","    << rho13 << "," << rho23 << ") is "
          <<  trivarcumnorm(x,y,z,rho12,rho13,rho23) << std::endl;

std::cout << std::setprecision(6)
           << "The value TVN Goursat(" << x << "," << y << "," << z
           << "," << rho12 << ","<< rho13 << "," << rho23 << ") is "
           << IntegralValues(x, y, z,rho12,rho13,rho23,550,550,550);
```

An extended example is to stress test the Genz algorithm against the finite difference solution. In this case the input is randomly generated and we run the algorithms a number of times. We first create objects to generate random numbers:

```
std::default_random_engine eng;
std::default_random_engine eng2;
std::default_random_engine eng3;
std::default_random_engine eng4;
std::default_random_engine eng5;
std::default_random_engine eng6;

std::random_device rd;

eng.seed(rd());
eng2.seed(rd());
eng3.seed(rd());
eng4.seed(rd());
eng5.seed(rd());
eng6.seed(rd());

// Generators for the parameters of BVN
std::uniform_real_distribution<double> uniformX(-8.0, 8.0);    // x
std::uniform_real_distribution<double> uniformY(-8.0, 7.9);    // y
std::uniform_real_distribution<double> uniformZ(-8.0, 7.9);    // z
std::uniform_real_distribution<double> uniformRho21(-0.6, 0.6);// rho12
std::uniform_real_distribution<double> uniformRho31(-0.6, 0.6);// rho13
std::uniform_real_distribution<double> uniformRho32(-0.6, 0.6);// rho23

const int NX = 550;
const int NY = NX;
const int NZ = NY;
```

```
double maxError = 0.0;
double minError = std::numeric_limits<double>::max();
const int NTrials = 100;
int coun = 0;
for (long long n = 1; n <= NTrials; ++n)
{
   if ((n / 100000) * 100000 == n)
   {
      std::cout << n << ",";
   }
   double b1 = uniformX(eng);
   double b2 = uniformY(eng2);
   double b3 = uniformY(eng3);
   double rho21 = uniformRho21(eng4);
   double rho31 = uniformRho31(eng5);
   double rho32 = uniformRho21(eng6);

   double val1 = trivarcumnorm(b1, b2, b3, rho21, rho31, rho32);
   double val2 = M(b1, b2, b3, rho21, rho31, rho32, NX, NY, NZ);

   std::cout << "Values, error: " << std::setprecision(6)
             << ++coun << "/ " << val1 << "," << val2 << ","
             << std::abs(val1 - val2)<< '\n';
   if (!std::isnan(val2))
   {
      maxError = std::max(maxError, std::abs(val1 - val2));
      minError = std::min(minError, std::abs(val1 - val2));
   }
}
std::cout << "Max, Min error: " << std::setprecision(16)
          << maxError << ", " << minError << '\n';
```

Output is displayed on the console. A snippet is:

```
Values, error: 1/ 3.36056e-25,8.60479e-26,2.50008e-25
Values, error: 2/ 5.55148e-15,5.55106e-15,4.2039e-19
Values, error: 3/ 0.998061,0.998062,1.55523e-06
Values, error: 4/ 0.667054,0.667056,2.38335e-06
Values, error: 5/ 3.57559e-08,3.57549e-08,9.60296e-13

// ...

Values, error: 96/ 7.25962e-15,7.21828e-15,4.13414e-17
Values, error: 97/ 0.0118934,0.0118931,2.91526e-07
Values, error: 98/ 6.82096e-17,6.19754e-17,6.23427e-18
Values, error: 99/ 9.44464e-07,9.44434e-07,2.96727e-11
Values, error: 100/ -4.16418e-25,1.23604e-73,4.16418e-25

Max, Min error: 6.003088278805357e-06, 1.36752501534285e-45
```

16.9.1 Four-Dimensional Distributions

The finite difference method can also be applied to four-dimensional distributions. The PDE in this case is given by:

$$\frac{\partial^4 u}{\partial x \partial y \partial z \partial \rho} = f. \tag{16.17}$$

The data structure to hold the generated discrete data is:

```
const int DIM = 4; // 3d matrix
typedef boost::multi_array<double, DIM> Tensor;
typedef Tensor::index Index;
```

The essence of the algorithm that describes the finite difference scheme is:

```
for (Index i = 1; i != NX+1; ++i)
{
   for (Index j = 1; j != NY+1; ++j)
   {
      for (Index k = 1; k != NZ+1; ++k)
      {
         for (Index p = 1; p != NP+1; ++p)
         {
            Values[i][j][k][p] =

   // A
   Values[i][j][k][p-1] + Values[i-1][j][k][p] -  Values[i-1][j][k][p-1] +

   // B
   Values[i][j-1][k][p] - Values[i][j-1][k][p-1] - Values[i-1][j-1][k][p]
          +  Values[i-1][j-1][k][p-1] +

   // C
   Values[i][j][k-1][p] - Values[i][j][k-1][p-1] - Values[i-1][j][k-1][p]
          +  Values[i-1][j][k-1][p-1] +
   // D
   -Values[i][j-1][k-1][p] + Values[i][j-1][k-1][p-1]
         + Values[i-1][j-1][k-1][p] -  Values[i-1][j-1][k-1][p-1] +


   // Value of RHS as centre of gravity (more accurate)
   + hx*hy*hz*hp*f(xarr[i-1] + hx/2, yarr[j-1] + hy/2,
            zarr[k-1] + hz/2,      parr[p-1] + hp/2);}

   // etc.
```

We have provided the code to compute the four-dimensional normal distributions for zero correlations. In general, we would need to manually invert the 4×4 covariance and compute

the quadratic form $X^{\mathrm{T}}\Sigma^{-1}X$ where $X = (X_1, X_2, X_3, X_4)^{\mathrm{T}}$ in equation (16.12), which we have not done. We leave this to the assiduous reader.

For more background on continuous multivariate distributions, see Johnson and Kotz (1972) and Kotz, Balakrishnan and Johnson (2000).

16.10 CHOOSER OPTIONS

A *chooser option* is an option where the holder has the right to choose whether it is a call or a put at some point in its life. The value of a chooser option at the point t_1 where the choice is made is given by:

$$w(S, X_c, X_p, t, T_c, T_p) = \max[CBSM(S, X_c, T_c),\ PBSM(S, X_p, T_p)] \tag{16.18}$$

where $CBSM(S, X, T)$ and $PBSM(S, X, T)$ are the values of the call and put underlying the option, respectively. These types of options can be priced using lattice methods, as discussed in Mun (2002) using the binomial method. A complex chooser option is one that is based on two expiries T_c, T_p and two strikes K_c, K_p. The price of the complex chooser option is then:

$$\begin{aligned} w = {} & Se^{(b-r)T_c} M(d_1, y_1; \rho_1) \\ & - X_c e^{-rT_c} M(d_2, y_1 - \sigma\sqrt{T_c}; \rho_1) - Se^{(b-r)T_p} M(-d_1, -y_2; \rho_2) \\ & + X_p e^{-rT_p} M(-d_2, -y_2 + \sigma\sqrt{T_p}; \rho_2) \end{aligned} \tag{16.19}$$

where:

$$\begin{aligned} d_1 &= \frac{\log(S/I) + (b + \sigma^2/2)t}{\sigma\sqrt{t}}, \quad d_2 = d_1 - \sigma\sqrt{t} \\ y_1 &= \frac{\log(S/X_c) + (b + \sigma^2/2)T_c}{\sigma\sqrt{T_c}}, \quad y_2 = \frac{\log(S/X_p) + (b + \sigma^2/2)T_p}{\sigma\sqrt{T_p}} \\ \rho_1 &= \sqrt{t/T_c}, \quad \rho_2 = \sqrt{t/T_p} \end{aligned}$$

and the value I is the solution of the nonlinear equation:

$$\begin{aligned} & Ie^{(b-r)(T_c-t)} N(z_1) - X_c e^{-r(T_c-t)} N\left(z_1 - \sigma\sqrt{T_c - t}\right) \\ & \quad + Ie^{(b-r)(T_p-t)} N(-z_2) - X_p e^{-r(T_p-t)} N\left(-z_2 + \sigma\sqrt{T_p - t}\right) = 0 \\ & z_1 = \frac{\log(I/X_c) + (b + \sigma^2/2)(T_c - t)}{\sigma\sqrt{T_c - t}}, \quad z_2 = \frac{\log(I/X_p) + (b + \sigma^2/2)(T_p - t)}{\sigma\sqrt{T_p - t}}. \end{aligned} \tag{16.20}$$

We implement the code for equation (16.19) as follows (price is 6.0508):

```
namespace ComplexChooser
{
   double S = 50.0;
   double Xc = 55.0;
   double Xp = 48.0;
   double Tc = 0.5;
   double Tp = 0.5883;
   double t = 0.25;
   double r = 0.1;
   double b = 0.1 - 0.05;   //0.05
   double sig = 0.35;
   double I = 51.1158;      // COMPUTE by NR as auxiliary process

   double ComplexChooserOption()
   { // Haug 2007, page 131

      double tmp = sig*std::sqrt(t);
      double d1 = (std::log(S / I) + (b + (sig*sig)*0.5) * t)
          / tmp;
      double d2 = d1 - tmp;

      double y1 = (std::log(S / Xc) + (b + sig*sig / 2)*Tc)
          / (sig*sqrt(Tc));
      double y2 = (std::log(S / Xp) + (b + sig*sig / 2)*Tp)
          / (sig*sqrt(Tp));

      double rho1 = std::sqrt(t / Tc); double rho2
            = std::sqrt(t / Tp);

      double result = S*std::exp((b - r)*Tc)*M(d1, y1, rho1)
         - Xc*std::exp(-r*Tc)*M(d2,y1-sig*std::sqrt(Tc), rho1)
         - S*std::exp((b - r)*Tp)*M(-d1, -y2, rho2)
         + Xp*std::exp(-r*Tp)*M(-d2,-y2+sig*std::sqrt(Tp), rho2);

      return result;
   }
}
```

In this way we can check the accuracy of the *Quantlib* implementation of BVN.

A simple chooser option gives the holder the right to choose whether the option is to be a standard call or put after a time t_1, with strike X and time to maturity T_2. The payoff from a simple chooser option at time $t_1 (t_1 < T_2)$ is:

$$w(S, X, t_1, T_2) = \max[CBSM(S, X, T_2),\ PBSM(S, X, T_2)]$$

where $CBSM(S, X, T_2)$ and $PBSM(S, X, T_2)$ are the general Black–Scholes–Merton call and put formulae.

The analytic solution is given by (Haug, 2007):

$$w = Se^{(b-r)T_2}N(d) - Xe^{-rT_2}N(d - \sigma\sqrt{T_2}) - Se^{(b-r)T_2}N(-y) + Xe^{-rT_2}N(-y + \sigma\sqrt{t_1}) \tag{16.21}$$

where:

$$d = \frac{\log(S/X) + (b + \sigma^2/2)T_2}{\sigma\sqrt{T_2}}, \quad y = \frac{\log(S/X) + bT_2 + \sigma^2 t_1/2}{\sigma\sqrt{t_1}}.$$

The C++ code that implements this formula is given by:

```
namespace SimpleChooser
{
   double S = 50.0;
   double X = 50.0;
   double T2 = 0.5;
   double t1 = 0.25;
   double r = 0.08;
   double b = 0.08;
   double sig = 0.25;

   double SimpleChooserOption()
   { // Haug 2007, page 128

      double d = (std::log(S / X) + (b + (sig*sig)*0.5) * T2)
          / (sig*sqrt(T2));

      double y = (std::log(S / X) + b*T2 + sig*sig*t1/2)
          / (sig*sqrt(t1));

      double result = S*std::exp((b - r)*T2)*N(d)
         - X*std::exp(-r*T2)*N(d - sig*std::sqrt(T2))
         - S*std::exp((b - r)*T2)*N(-y)
         + X*std::exp(-r*T2)*N(-y + sig*std::sqrt(t1));

      return result;
   }
}
```

The answer in this case is 6.1071.

16.11 CONCLUSIONS AND SUMMARY

In this chapter we gave a detailed discussion of the bivariate normal distribution and its implementation in C++. We introduced several solutions based on the Genz algorithm, numerical quadrature and a novel approach by posing the problem as a partial differential equation that

we then solve using the finite difference method. We compared these solutions based on efficiency, accuracy and maintainability characteristics. You can choose the solution that best fits your requirements.

The bivariate normal distribution has many applications. In this chapter we discussed its use in the algorithms to compute analytical prices of plain two-factor options. We note that the finite difference method can be applied to a wide range of bivariate distributions, for example the *bivariate t-distribution* (BVT), as the following code snippet shows:

```
class BVT
{
private:
   double nu;
   double rho;

public:
BVT(int dof, double correlation)
   : nu(static_cast<double>(dof)), rho(correlation) {}

   // Call the probability function
   double operator () (double x, double y)
   {
      double z = x*x - 2.0*rho*x*y + y*y;

      double d = 1.0/(2.0*(1.0 - rho*rho)*nu);
      double factor = 1.0 / (2.0*3.1415926*std::sqrt(1.0 - rho*rho));

      return factor*std::pow(1.0 + d*z, -(nu + 2) / 2);
   }
};
```

See Exercise 2. The method can also be applied to trivariate distributions. This topic is outside the scope of the book.

16.12 EXERCISES AND PROJECTS

1. (Domain Transformation, Project)

In this exercise we focus on transforming the PDE (16.2) to a finite domain, typically the square $[-1, 1] \times [-1, 1]$. The function to transform the two-dimensional infinite domain to this square is given by:

$$z = \coth x, \; w = \coth y.$$

Answer the following questions:

a) Prove that the PDE in the new coordinates is given by:

$$a(z, w)\frac{\partial^2 u}{\partial z \partial w} = \tilde{f}(z, w; \rho)$$

where:

$$a(z, w) = 4(1 - z)(1 + z)(1 - w)(1 + w)$$

and where $\tilde{f}$ tilde is a variant of f.

Discretise this new PDE using the same second-order scheme as in equation (16.6). In the interest of reusability and effort, modify the code to accommodate the form of this new PDE. Test the scheme.

b) Compute the relative accuracy of the finite difference schemes based on domain truncation and domain transformation. Incidentally, we shall use the domain transformation technique to transform a PDE on a semi-infinite interval to a PDE on the unit interval. We discuss this technique in Chapter 23.

c) Modify the class `GoursatFdmExtrapolation` to support the new PDE and test its accuracy.

2. (Modelling Bivariate t-Distribution, Medium-Size Project)

Apply the finite difference method to compute the BVT. The possible definitions are (Cornish, 1954; Genz, 2004):

$$T_\nu(b, \rho) = \frac{1}{2\pi\sqrt{1 - \rho^2}} \int_{-\infty}^{b_1} \int_{-\infty}^{b_2} f(x_1, x_2; \rho, \nu) dx_2 dx_1, \quad b = (b_1, b_2)^\top$$

where:

$$f(x_1, x_2; \rho, \nu) = \left(1 + \frac{x_1^2 + x_2^2 - 2\rho x_1 x_2}{(1 - \rho^2)\nu}\right)$$

and

$$T_\nu(b, \rho) = \frac{2^{1-\nu/2}}{\Gamma(\nu/2)} \int_0^\infty f(s, b; \rho, \nu) ds$$

where:

$$f(s, b; \rho, \nu) = s^{\nu-1} e^{-s^2/2} M\left(\frac{s}{\sqrt{\nu}} b_1, \frac{s}{\sqrt{\nu}} b_2, \rho\right)$$

and $\Gamma(z)$ is the Gamma function, $\Gamma(z) = \int_0^\infty x^{z-1} e^{-x} dx$.

Answer the following questions:

a) Apply the finite difference method to compute the above integrals. The class `GoursatFdm` can be used without modification as it has been designed to operate with any two-dimensional distribution.

b) Determine whether domain transformation or domain truncation is better as a means of reducing the region of integration to a bounded domain in this case.

c) Test and debug the new code by first taking examples whose values you know (for example, the integral should be 1 for the complete domain). This would be the first 'sanity' check.

d) Test the accuracy and efficiency by comparing the values with those produced by *Quantlib*. As a hint, we give some sample code:

```
#include ql/math/distributions/bivariatestudenttdistribution.hpp>
double dof = 4; double rho = 0.5;
QuantLib::BivariateCumulativeStudentDistribution st1(dof, rho);
std::cout << "BVT QuantLib 1.8 : " << std::setprecision(12)
          << st1(a,b) << '\n';
```

3. (Two-Asset Option Pricing)

In Section 16.8 we discussed two-asset option pricing in combination with the bivariate normal distribution. The code is easy to write, understand and extend.

Answer the following questions:

a) Perform a review/proofread of the code to make sure that there are no coding errors or typos. You can check your results with the algorithms and examples using VBA in Haug (2007), for example.

b) Can you see opportunities to optimise the code?

c) We tested the algorithms using the Genz, West and finite difference methods. Now test the algorithms using the *Quantlib* implementation.

d) Modify the code to allow it to switch between the different approximations to integral (16.1).

e) Use the code to implement the algorithm in Bjerksund and Stensland (2002) that provides a (coarse) approximation to the price of a one-factor American option.

4. (Improving the Code in Section 16.9, Small Project)

The code in Section 16.9 was included to show how to implement a three-dimensional finite difference scheme. In this exercise we discuss how to improve the code.

Answer the following questions:

a) We tested the code on a range of parameter values. We restricted the correlations to $[-0.5, 0.5]$ because we were getting NaN values when outside this range. But for these values the matrix is not a valid correlation matrix. We added this check to the code in order to avoid including NaNs in the errors:

```
if (!std::isnan(val2))
{
   maxError = std::max(maxError, std::abs(val1 - val2));
   minError = std::min(minError, std::abs(val1 - val2));
}
```

Modify the code so that is works for all parameter values. Run the program again with larger correlation values. Make sure that there are no NaNs appearing in the output. You will need to create a valid correlation matrix.

b) Compare the Genz and finite difference with regards to efficiency. Consider the cases of how much effort it takes to compute a single value and a matrix of values (we note that the finite difference method produces a matrix of values 'for free' as it were).

c) We expect (and it should be proved) that scheme (16.15) is second-order accurate. One way to verify that it is second order is to apply Richardson extrapolation to promote the accuracy to fourth order.

d) The finite difference method produces a matrix of values. Import these values into Excel and display them using the Excel driver that we introduced in Chapter 14.

e) Investigate the use of *Boost multiprecision* (introduced in Appendix 1) to improve the accuracy and robustness of the scheme.

f) Use the generated matrix of values as a *lookup table*. What are the use cases and how would you implement them?

5. (Trivariate t-Distribution)

Consider the multivariate distribution:

$$\frac{\Gamma[(\nu+p)/2]}{\Gamma(\nu/2)\nu^{p/2}\pi^{p}\sqrt{|\Sigma|}}\left[1+\frac{1}{\nu}(x-\mu)^{\mathrm{T}}\Sigma^{-1}(x-\mu)\right]^{-(\nu+p)/2} \tag{16.22}$$

where:

$\mu = (\mu_1, \ldots, \mu)^{\mathrm{T}}$ is location

Σ = positive-definite $p \times p$ covariance matrix

ν = number of degrees of freedom

$\Gamma(z)$ = Gamma function.

Test the finite difference on this distribution in the case $p = 3$ in much the same way that we did with the trivariate normal distribution. How would you determine how accurate your results are? You could try smaller mesh sizes and Richardson extrapolation, for example. Consider when you let the number of degrees of freedom go to infinity. In that case this distribution should give values different to those generated from the trivariate normal distribution. Run the program and compare the resulting matrices. Are they 'close'?

6. (Chooser Options)

This exercise is based on Section 16.10. Answer the following questions:

a) The complex chooser example that we took in Section 16.10 was specific in the sense that we assumed the value I of the nonlinear equation (16.20) was known. In general, we must solve this equation for I using a solver such as Newton–Raphson, for example. To this end, use the code from Chapter 19 and test the software again.

b) Another test of the code in part a) is in the case when both the strikes and the expiries are the same. Use the same data as in the simple chooser example. As before, you should get the value 6.1071.

We discuss the valuation of compound and chooser options using lattice and finite difference methods in Chapter 25.

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 17

STL Algorithms in Detail

17.1 INTRODUCTION AND OBJECTIVES

The goal of this chapter is to discuss how STL supports a range of algorithms that operate on STL containers. In general, an *algorithm* is a complete, step-by-step procedure that solves a given problem. Each step must be expressed in terms of a finite number of rules and each step is guaranteed to terminate in a finite number of application of the rules. In most cases a rule corresponds to the execution of one or more operations. A *sequential* algorithm performs operations one at a time in sequence. This is in contrast to *distributed* and *parallel* algorithms that can perform many operations simultaneously (Berman and Paul, 2005). In this chapter we discuss how STL supports algorithms that operate on sequence and associative containers. It is important to state that the design of the STL algorithms is based on the *functional programming paradigm*. Instead of encapsulating data and operations in a class or struct (as we are accustomed to do with the object-oriented programming paradigm), STL separates data from algorithms by iterators playing the role of *mediator* between them. In particular, a common scenario is to iterate in a generic container from beginning to end and then apply an operation to each element in that container. The code in Chapters 17 and 18 is machine readable and it can be used as exemplar code for your own applications.

In order to motivate this paradigm we introduce the `for_each()` algorithm (all algorithms are in the namespace `std`) that allows us to access, process and modify the elements in a container in different ways (see Josuttis, 2012). This algorithm expects two iterators and a function object (or lambda function), the latter operating on each element between these iterators. Let us take two examples; we consider a list and we define lambda functions to print the elements of the list and to multiply each element of the list by a constant factor:

```
#include <list>
#include <algorithm>
#include <iostream>


int main()
{
    std::list<double> myList;
    myList.push_back(1.0);myList.push_back(2.0);
    myList.push_back(3.0);myList.push_back(4.0);
```

```
     // Modify the elements in the container
    std::for_each(myList.begin(), myList.end(), [](double& d)
                            {d *= 2.0;});

    // Print the elements of the container
    std::for_each(myList.begin(), myList.end(), [] (double d)
                            {std::cout << d << ", ";});


    return 0;
}
```

Here we see how lambda functions are applied to each element of the list. We can also rewrite the above code using free functions:

```
// Functions that operate on types
template <typename T>
    void Modify(T& t)
{
    t *= 2.0;
}

template <typename T>
    void Print (const T& t)
{
    std::cout << t << ", ";
}

int main()
{ // Using free functions

    std::list<double> myList;
    myList.push_back(1.0); myList.push_back(2.0);
    myList.push_back(3.0);myList.push_back(4.0);

    // Modify the elements in the container
    std::for_each(myList.begin(), myList.end(), Modify<double>);

    // Print the elements of the container
    std::for_each(myList.begin(), myList.end(),
                                [] (double d) {std::cout <<d << ", ";});


                                                return 0;
}
```

In both cases we see the basic principles underlying the functional programming paradigm. In particular, most STL algorithms are based on the general principle of iterating in a

container and performing some operation on each of its elements in sequence. As a final example we introduce the `transform()` algorithm that is more flexible than `for_each()` (but slightly less efficient). This algorithm transforms a source into a destination in some way. We consider two cases in the following code; in the first case we multiply each element of a source by two and in the second case we also multiply each element of the source by two and we then make a copy which we transfer to the destination sequence:

```
int main()
{ // Using transform() with lambda

    std::list<double> myList;
    myList.push_back(1.0);myList.push_back(2.0);
    myList.push_back(3.0);myList.push_back(4.0);

    // Modify elements in container; note return type of lambda function
    std::transform(myList.begin(), myList.end(), myList.begin(),
                   [] (const double& d) -> double {return d*2.0;});

    // Transform modified contents of source (list) into destination
    // (vector)
    std::vector<double> myVector(myList.size());
    std::transform(myList.begin(), myList.end(), myVector.begin(),
                   [] (const double& d) -> double {return d*2.0;});

   return 0;
}
```

Even now we see how flexible STL is; for example, it is possible to copy lists to vectors (and vice versa). The complexity of the above algorithms is linear because we perform one operation for each element in the container in which we iterate.

Summarising, we have motivated the use of STL algorithms. Once you understand the intuition underlying the code examples in this section you should have little difficulty understanding the other algorithms. To this end, we focus on a number of attention points and major objectives:

- To give a global and thorough overview of the algorithm categories in STL. Each category contains several generic and customisable algorithms. In general, the algorithms should satisfy the needs of many applications and they can be used instead of creating similar functionality yourself.
- Learning how to use algorithms by taking basic and illustrative examples in order to become accustomed to the syntax and what is on offer.
- We give some general guidelines on how to determine which algorithm category to choose in a given context. This *pattern-recognition* activity will be elaborated in succeeding chapters when we apply the algorithms in frameworks and applications.

This chapter, as well as Chapter 18, is meant to be used as reference.

17.2 BINDERS AND `std::bind`

A *binder* is a kind of *function adapter* that allows otherwise incompatible code and functions to interoperate. We introduce binders as the first major topic in this chapter because they can be used in combination with STL algorithms. In general, we create *supplier code* consisting of free functions, object and static member functions as well as function objects, pointers to functions and stored lambda functions. *Client code* does not always satisfy the requirements that the supplier expects. The two major issues that cause the mismatch between supplier and client are function name and function arity.

17.2.1 The Essentials of `std::bind`

This library originated in Boost (see Demming and Duffy, 2010 for a discussion of `boost::bind`). From the *Boost.Bind* online documentation:

> boost::bind *is a generalization of the standard functions* `std::bind1st` *and* `std::bind2nd`*. It supports arbitrary function objects, functions, function pointers, and member function pointers, and it is able to bind any argument to a specific value or route input arguments into arbitrary positions. Bind does not place any requirements on the function object; in particular, it does not need the* `result_type`*,* `first_argument_type` *and* `second_argument_type` *standard typedefs.*

We take an example. In general, *binders* support *higher-order functions* which are functions that can take other functions as input and they can also return functions as return types. In the following examples we define a function of arity two and bind it to other functions of arity one and arity zero. The given function of arity two is defined by:

```
double func(double x, double y)
{
       return x-y;
}
```

We now define a number of functions in which some parameters are given values (are *bound*) while others are *unbound* and thus have not yet been initialised. To this end, we use *placeholders* to identity these unbound arguments. The placeholder names are numbered `_1`, `_2`, … up to an implementation-defined maximum number. An initial example gives the value `1` in all three cases (please check):

```
// Placeholder objects; the _nth object is the nth unbound argument
// starting from the left-hand side of argument list

using namespace std::placeholders;

{ // Different ways to execute partial function application

    auto f1 = std::bind(&func, _1, 1.0);
    std::cout << "Value == 1? : " << f1(2.0) << '\n';
```

```
    auto f2 = std::bind(&func, 2.0, _1);
    std::cout << "Value == 1? : " << f2(1.0) << '\n';

    auto f3 = std::bind(&func, 2.0, 1.0);
    std::cout << "Value == 1? : " << f3() << '\n';
}
```

We now discuss introducing two placeholder objects into the code:

```
// Bind params 1 and 2, answer = -1
double result = (std::bind(&func, _1, _2))(1.0, 2.0);
std::cout << "_1, _2: " << result << std::endl;

// Bind params 2 and 1, answer = 1
result = (std::bind(&func, _2, _1))(1.0, 2.0);
std::cout << "_2, _1: " << result << std::endl;

// Binding to params chosen at run-time
std::cout << "Give a value 1: "; int v1; std::cin >> v1;
std::cout << "Give a value 2: "; int v2; std::cin >> v2;

result = (std::bind(&func, _1, _2))(v1, v2);
std::cout << "_1, _2: " << result << std::endl;

// Using placeholders in more than one place in an expression
result = (std::bind(&func, _1, _1))(v1, v2);
std::cout << "_1, _1: " << result << std::endl;

result = (std::bind(&func, _2, _2))(v1, v2);
std::cout << "_2, _2: " << result << std::endl;

result = (std::bind(&func, _2, _1))(v1, v2);
std::cout << "_2, _1: " << result << std::endl;
```

It is recommended that you run this code and examine the output to see the interaction between bound and unbound variables. The syntax can be somewhat cryptic at times but practice makes perfect! See Exercise 1.

17.2.2 Further Examples and Applications

Since classes and their instances play an important role in software development we need to know how to use binders with their member functions (and member data, although we do not discuss this just yet). We take a representative example:

```
class C
{ // Simple class to which we apply std::bind
//private:
public:
```

```
    double _data;
public:
    C(double data) : _data(data) {}

    double operator () (double t1, double t2)
    { // Function object

        return t1 + t2;
    }

    static double compute (double t1, double t2)
    {
        return t1 * t2;
    }

    double computeII(double t)
    {
        return std::exp(t) + _data;
    }

    void print() const
    {
        std::cout << "A class C\n";
    }
};
```

Let us suppose that we have client software that uses universal function wrappers and their target methods. We wish to instantiate them as it were by binding them to `C`'s (note that `C` is a function object) members, as the following examples show:

```
// Bind with function objects
C c(0.0);
std::function<double (double, double)> f2
                  = std::bind(c, _1, _2);
std::cout << "f2 : " << f2(4.0, 2.0) << '\n'; // 6

std::function<double (double, double)> f2 = std::bind
(&C::compute, _1, _2);
std::cout << "f2 : " << f2(4.0, 2.0) << '\n'; // 6

std::function<double(double)> f3 = std::bind(&C::computeII, c, _1);
std::cout << "f3 : " << f3(5.0) << '\n'; // 148.413
```

In this way we can see how to resolve mismatch problems in function names and function arity.

17.2.3 Deprecated Function Adapters

The following predefined function adapter classes are deprecated since C++11: `bind1st()`, `bind2nd()`, `ptr_fun()`, `mem_fun()`, `mem_fun_ref()`, `not1()`, `not2()`. We do not discuss them in this book. A discussion can be found in Josuttis (2012).

We discuss `std::mem_fn` in C++11 that generates wrapper objects for pointers to members that can store, copy and invoke a pointer to a member. We use the same example as in Section 17.2.2 and we show how to define a wrapper for pointers to members. We first define the pointer and then we *invoke* the function on the object while providing the necessary input parameters:

```
// C++11 mem_fn
C c2(3.1415);

auto f1 = std::mem_fn(&C::print);
f1(c2);

auto f2 = std::mem_fn(&C::computeII);
std::cout << "Compute a value: " << f2(c2, 1.0) << '\n';

auto f3 = std::mem_fn(&C::_data);
std::cout << "data: " << f3(c2) << '\n';
```

The use of `std::mem_fn` is an alternative to `std::bind` and in some ways it is easier to use, especially when accessing member data. Clients can then use the function as if it were a simple free function. You can decide if it is easier to use than binders.

17.2.4 Conclusions

There are several ways to let software components interoperate with each other. It feels like programming in C or in some procedural language. The choice depends on a number of criteria (as always), such as robustness, readability and maintainability. The choices are:

- Lambda functions and stored lambda functions.
- `std::bind`.
- `std::mem_fn`.
- Creating dedicated function objects.

The best way to determine which option is optional in a given context is by trying it!

17.3 NON-MODIFYING ALGORITHMS

These are algorithms that read and use the elements of the containers that they operate on. They change neither the order nor the value of these elements. The algorithms operate with input and forward iterators. In general, the *containers do not need to be sorted*. The main categories are:

- Counting the number of elements satisfying a certain condition.
- Min/max values in a container.
- Searching for {first, consecutive, subrange} consecutive elements; searching for subranges.
- Comparing ranges (testing for: equality, unordered equality, first difference, less than).
- Predicates for ranges ((partial) sorting, partitioned, is it a heap?, {all/any/none}).

We discuss these algorithms in detail by giving examples. Where possible, we give some hint on how they can be used and extended to applications. A point to remember is that each algorithm has a complexity. For example, finding the maximum value in a container has linear complexity. For a further discussion, see Josuttis (2012).

In general, a good way to understand the algorithms is to inspect the code samples, the output that they produce and then understand why the output is produced. After that, we recommend coding examples.

17.3.1 Counting the Number of Elements Satisfying a Certain Condition

There are two versions of the `count()` algorithm; the first counts the number of elements in a container having a given value while the second counts the number of elements that satisfy a certain condition:

```
void Count()
{
    std::cout << "\n**** Block I, Counting ****\n";
    // I. COUNT the number of elements satisfying a certain condition
    std::vector<int> v = { 1, 2, 3, 4, 5, 5, -6 };
    print(v, "vector");

    // Count the number of elements equal to a given value.
    int val = 5;
    std::cout << "Number of elements equal to " << val << " = "
              << std::count(std::begin(v), std::end(v), val) << std::endl;

    // Count the number of even values. O(N) complexity.
    auto Even = [](int n) { return n % 2 == 0; };
    auto Odd = [&](int n) { return !Even(n); };
    std::cout << "Number of even values: "
              << std::count_if(std::begin(v), std::end(v), Even)
              << std::endl;
    std::cout << "Number of odd values: "
              << std::count_if(std::begin(v), std::end(v), Odd)
              << std::endl;

    double upperThreshold = 3; // Captured variable, C++ 11
    auto Threshold=[&upperThreshold](int n)
                    { return n > upperThreshold; };

    std::cout << "Number of elements greater than " << upperThreshold
              << " is: "
              << std::count_if(std::begin(v),std::end(v),Threshold)
              << std::endl;

    // Use std::bind to do the same thing
    std::cout << "Number of elements using bind greater than "
              << upperThreshold << " is: "
              << std::count_if(v.begin(), v.end(),
```

```
            std::bind(std::greater<int>(),
            std::placeholders::_1, upperThreshold)) << std::endl;
}
```

The output is:

```
**** Block I, Counting ****
vector: 1,2,3,4,5,5,-6,
Number of elements equal to 5 = 2
Number of even values: 3
Number of odd values: 4
Number of elements greater than 3 is: 3
Number of elements using bind greater than 3 is: 3
```

17.3.2 Minimum and Maximum Values in a Container

There are six algorithms in this category to find the maximum and minimum values in a container, namely as value, using a function object or delivering the output as a pair:

```
void MinMax()
{
   std::cout << "\n**** Block II, min/max values ****\n";
   std::vector<int> v = { 1, 2, 3, 4, 5, 5, -6 };
   print(v, "vector");
   // II. Minimum and maximum; these functions
   // return an iterator O(N) complexity
   // 4 options {min | max}, {"real" min/max | comparison operation)

   std::cout << "Minimum and maximum values: "
             << *std::min_element(std::begin(v), std::end(v)) << ", "
             << *std::max_element(std::begin(v), std::end(v)) << std::endl;

   // Minimum and maximum using function object to compare elements;
   // these functions return an iterator O(N).
   auto AbsLess = [] (double d1, double d2)
                    {return std::abs(d1) < std::abs(d2);};

   std::cout << "Minimum and maximum ABS values: "
             << *std::min_element(std::begin(v), std::end(v), AbsLess)
             << ", "
             << *std::max_element(std::begin(v), std::end(v), AbsLess);

   // Min and max in one sweep as it were
   std::vector<int> vec(3); vec[0] = 44; vec[1] = -23; vec[2] = 889;
   auto result = std::minmax_element(std::begin(v), std::end(v));
   std::cout << "Min, max pair: "
             << *(result.first) << ", " << *(result.second) << std::endl;
```

```
    result = std::minmax_element(std::begin(v), std::end(v),
    std::greater<int>());
    std::cout << "Min, max pair with function object: "
              << *(result.first) << ", " << *(result.second) << std::endl;
}
```

The output is:

```
**** Block II, min/max values ****
vector: 1,2,3,4,5,5,-6,
Minimum and maximum values: -6, 5
Minimum and maximum ABS values: 1, -6
Min, max pair: -6, 5
Min, max pair with function object: 5, -6
```

17.3.3 Searching for Elements and Groups of Elements

This category contains algorithms to search for values in *unordered ranges*. The complexity is linear. In Chapter 18 we discuss searching in *ordered ranges* in which case complexity is logarithmic or linear. The return type in these cases is an iterator; notice how we determine if a value is not in the collection.

```
void Find()
{
    using namespace std::placeholders;

    std::cout << "\n**** Block III, Finding ****\n";
    std::vector<int> v = { 1, 2, 3, 4, 5, 5, -6 };
    print(v, "vector");
    // Searching for elements; find the first element in a container
    // equal to a given value;
    int searchValue = 4;

    // 1. Find position of 1st element in a range
    std::vector<int>::iterator posS
                            = std::find(std::begin(v), std::end(v),
                            searchValue);

    // Test if value has been found
    if (posS != std::end(v))
    {
       std::cout << "Value found: " << *posS << std::endl;
    }
    else
    {
       std::cout << "Value " << searchValue << " not found\n";
    }

    searchValue = 4;
    // Find position 1st element in range for which unary
```

```
    // predicate is true
    std::vector<int>::iterator posF
                = std::find_if(std::begin(v), std::end(v),
                                    [&searchValue](double d)
                        {return d > searchValue - 2; });
    if (posF != std::end(v))
    {
       std::cout << "First value *found* > "
                 << searchValue - 2 << " is: " << *posF << std::endl;
    }
    else
    {
        std::cout << "Value not found\n";
    }
}
```

The output is:

```
**** Block III, Finding ****
vector: 1,2,3,4,5,5,-6,
Value found: 4
First value *found* > 2 is: 3
```

When using the algorithms in this category we distinguish between *offset* and *position*; an offset is zero-based and a position has initial index one. For example, the first element of a container has offset zero and position one.

17.3.4 Searching for Subranges

Whereas the algorithms in Section 17.3.3 search for a *single value* (usually the first-found element) in a range, the algorithms in this section search for the occurrence of a *subrange* within a range:

```
void Search()
{
    std::cout << "\n**** Block IV, Searching ****\n";
    // 3. Find all occurrences of a subrange in a range.
    // Output is the start position.
    std::vector<int> myRange
     = { 0, 1, 2, 3, 4, 5, 6, 6, 1, 2, 2, 9, 9, 9, 8, 7, 1, 2, 3 };
    std::vector<int> mySubRange = { 1, 2, 3 };
    print(myRange, "range to be searched in");
    print(mySubRange, "subrange to be searched for");

    auto pos = myRange.begin();

    // Search first subrange
    pos = std::search(myRange.begin(), myRange.end(),
                      mySubRange.begin(), mySubRange.end());
```

```
// Loop while subrange found
while (pos != myRange.end())
{
    // Distance between the iterators (plus 1):
    // position := distance + 1
    std::cout << "Subcollection search found at offset: "
              << std::distance(std::begin(myRange), pos) << std::endl;

    pos = std::search(++pos, myRange.end(), mySubRange.begin(),
    mySubRange.end());
}

// Search first n matching *consecutive* elements with value
// *compared* to myValue. Overloaded version with binary predicate.
std::vector<int> myRangeA
 = { 0,2, 2, 2, 2, 2, 9, 9, 8, 7, 6, 6, 6, 6, 6, 1, 3, 3, 3, 3,3 };
print(myRangeA, "range to be searched for consecutive elements");

int count = 4; // Number of consecutive elements
int myValue = 6; // The value with 'consecutive replication'
auto pos22 = std::search_n(std::begin(myRangeA), std::end(myRangeA),
count, myValue);

if (pos22 != std::end(myRangeA))
{
    std::cout << count << "Consecutive elems(position==offset + 1): "
              << std::distance(std::begin(myRangeA), pos22) + 1
              << std::endl;
}
else
{
    std::cout << "Consecutive elements not found\n";
}

// Find count *consecutive* elements with value > myValue
// Tricky: think about this a bit
std::vector<int> myRangeB
 = { 0,2, 2, 2, 9, 1, 8, 7, 0, 6, 0, 6, 6, 1, 3, 3, 3, 3,3 };
print(myRangeB, "range to be searched");
count = 3;
myValue = 2;
auto posB = std::search_n(std::begin(myRangeB), std::end(myRangeB),
                          count, myValue, std::greater<int>());

if (posB != std::end(myRangeB))
{
    std::cout << count << "Consecutive elems{position): >> "
              << myValue << ": "
              << std::distance(std::begin(myRangeB), posB) + 1
              << std::endl;
}
```

```
    else
    {
        std::cout << "Consecutive elements not found\n";
    }
}
```

The output from this code is:

```
**** Block IV, Searching ****
range to be searched in: 0,1,2,3,4,5,6,6,1,2,2,9,9,9,8,7,1,2,3,
subrange to be searched for: 1,2,3,
Subcollection search found at offset: 1
Subcollection search found at offset: 16
range to be searched for consecutive elements:
0,2,2,2,2,2,9,9,8,7,6,6,6,6,6,1,3,3,3,3,3,
4 Consecutive elements (position == offset + 1): 11
range to be searched: 0,2,2,2,9,1,8,7,0,6,0,6,6,1,3,3,3,3,3,
3 Consecutive elements {position): >> 2: 15
```

17.3.5 Advanced Find Algorithms

The algorithms in this category are concerned with finding the first and subsequent occurrences of a subrange in a range as well as algorithms that find elements based on values of their *adjacent elements*:

```
void AdvancedFind()
{
    std::cout << "\n**** Block V, Advanced Find ****\n";
    std::vector<int> myRange
          = { 0, 1, 2, 3, 4, 5, 6, 6, 1, 2, 2, 9, 9, 9, 8, 7, 1, 2, 3 };
    std::vector<int> mySubRange = { 1, 2, 3 };

    print(myRange, "range to be searched");
    print(mySubRange, "subrange to be searched for");


    // Find position of the first element of the first
    // subrange matching the input range.
    auto pos = std::find_first_of(std::begin(myRange), std::end(myRange),
               std::begin(mySubRange), std::end(mySubRange));
    std::cout << "First subrange found at position: "
              << distance(std::begin(myRange), pos) + 1 << std::endl;

    // First position of last subcollection in a collection
    pos = std::find_end(std::begin(myRange), std::end(myRange),
                        std::begin(mySubRange),
                        std::end(mySubRange));
    std::cout << "Last subrange found at position: "
              << std::distance(std::begin(myRange), pos) + 1;
```

```
    // Return the first element in input range myRange that
    // has a value equal to the value of the following element,
    // i.e. 1st two elements with equal values.

    std::vector<int> myRangeB
       = { 1, 3, 3, 2, 4, 4, 5, 5, 0 };     // Josuttis' example.
    print(myRangeB, "adjacent find");
    auto posB = std::adjacent_find(std::begin(myRangeB),
    std::end(myRangeB));

    int count = 3;
    if (posB != std::end(myRangeB))
    {
       std::cout << "1st two elements with equal value at position: "
                 << distance(std::begin(myRangeB), posB) + 1 << std::endl;
    }

    // First adjacent elements satisfying some relation
    // between these elements. Could be used for series,
    // geometric progressions etc.
    myRangeB.clear();
    myRangeB = { 1, 3, 2, 4, 3, 27, 0 };// Josuttis' example.
    print(myRangeB, "input range");
    count = 2;
    posB = std::adjacent_find(std::begin(myRangeB), std::end(myRangeB),
             [](double d1, double d2) -> bool { return d2 == d1*d1*d1; });

    // 1st two elements with n = p^3
    if (posB != std::end(myRangeB))
    {
        std::cout << "1st two elements with next = prev^3: "
                  << std::distance(std::begin(myRangeB), posB) + 1;

    }
}
```

The output is:

```
**** Block V, Advanced Find ****
range to be searched: 0,1,2,3,4,5,6,6,1,2,2,9,9,9,8,7,1,2,3,
subrange to be searched for: 1,2,3,
First subrange found at position: 2
Last subrange found at position: 17
adjacent find: 1,3,3,2,4,4,5,5,0,
1st two elements with equal value at position: 2
input range: 1,3,2,4,3,27,0,
1st two elements with next = prev^3: 5
```

17.3.6 Predicates for Ranges

The algorithms in this category test if two ranges are equal and they compute the first position where two ranges differ. We also include an algorithm to perform *lexicographical comparisons*:

```
void Ranges()
{
    std::boolalpha;
    std::cout << "\n**** Block VI, Ranges and their algorithms ****\n";
    // IV. Relationships between ranges
    std::vector<int> myRangeB = { 1, 3, 2, 4, 5, 6, 7};
    // Testing equality of ranges.
    std::vector<int> myRangeC = { 1, 4, 3 };
    print(myRangeB, "range B");
    print(myRangeC, "range C");
    std::cout << "Ranges B, C equal: " << std::boolalpha
              << std::equal(std::begin(myRangeB), std::end(myRangeB),
                            std::begin(myRangeC)) << std::endl;

    // Test two ranges element-wise and apply a function operator.
    std::vector<int> myRangeD; myRangeD = { 1, 3, 2, 4, 3, 27, 0 };
    std::vector<int> myRangeE; myRangeE = { 2, 4, 7, 5, 8, 43, 3 };
    print(myRangeD, "range D");
    print(myRangeE, "range E");
    boolalpha(std::cout);
    std::cout << "Ranges D,E satisfy less relation: " << std::boolalpha
              << std::equal(std::begin(myRangeD), std::end(myRangeD),
    std::begin(myRangeE), [](double d1, double d2) -> bool
             { return (d1 < d2); }) << std::endl;

    // Compare the individual elements of two containers;
    // in this the elements must be squares
    myRangeD.clear();       myRangeD = { 1, 2, 3, 4, 5, 6 };
    myRangeE.clear();       myRangeE = { 1, 4, 9, 16, 25, 36 };
    print(myRangeD, "range D");
    print(myRangeE, "range E");
    std::cout << "Ranges D, E satisfy squared relation: "
    << std::boolalpha
              << std::equal(std::begin(myRangeD),
    std::end(myRangeD),std::begin(myRangeE),
         [](double d1, double d2) -> bool { return (d2 == d1*d1); })
         << std::endl;


    // Search the first difference between 2 ranges.
    // Have we found a mismatch?
    myRangeD.clear();       myRangeD = { 1, 2, 3, 4, 5, 6 };
    myRangeE.clear();       myRangeE = { 1, 2, 3, 8, 16, 3 };
    print(myRangeD, "range D");
    print(myRangeE, "range E");
```

```
std::pair<std::vector<int>::iterator, std::vector<int>::iterator>
  values;
values = std::mismatch(std::begin(myRangeD), std::end(myRangeD),
                       std::begin(myRangeE));

if (values.first == std::end(myRangeD))
{
    std::cout << "No mismatch found in D<E\n";
}
else
{
    std::cout << "First mismatch D,E VALUES: " << *values.first
              << "," << *values.second << std::endl;
}

// Return 1st two corresponding elements of ranges that return
false values = std::mismatch(std::begin(myRangeD), std::end(myRangeD),
                       std::begin(myRangeE), std::less_equal<int>());

if (values.first == std::end(myRangeD))
{
    std::cout << "No mismatch D,E found\n";
}
else
{
    std::cout << "First mismatch (with FO) D,E VALUES: "
              << *values.first << "," << *values.second << std::endl;
}

// Lexicographical comparisons. We use ASCII collating sequence.
std::vector<char> myRangeF; myRangeF = { 'a', 'b', 'c' };// 61, 62, 63
std::vector<char> myRangeG; myRangeG = { 'A', 'B', 'C' };// 41, 42, 43
print(myRangeF, "range F");
print(myRangeG, "range G");
boolalpha(std::cout);
std::cout << "Equality F,G: "
          << std::lexicographical_compare(std::begin(myRangeF),
             std::end(myRangeF), std::begin(myRangeG),
             std::end(myRangeG)) << std::endl;
             std::cout << "Comparison using function object: "
                   << std::lexicographical_compare(std::begin(myRangeF),
                         std::end(myRangeF),std::begin(myRangeG),
                         std::end(myRangeG),
                               [](double d1, double d2) -> bool
                                     { return (d1 > d2); }) << std::endl;}
```

The output is:

```
**** Block VI, Ranges and their algorithms ****
range B: 1,3,2,4,5,6,7,
```

```
range C: 1,4,3,
Ranges B, C equal: false
range D: 1,3,2,4,3,27,0,
range E: 2,4,7,5,8,43,3,
Ranges D,E satisfy less relation: true
range D: 1,2,3,4,5,6,
range E: 1,4,9,16,25,36,
Ranges D, E satisfy squared relation: true
range D: 1,2,3,4,5,6,
range E: 1,2,3,8,16,3,
First mismatch D,E VALUES: 4,8
First mismatch (with function object) D,E VALUES: 6,3
range F: a,b,c,
range G: A,B,C,
Equality F,G: false
Comparison using function object: true
```

17.4 MODIFYING ALGORITHMS

These algorithms change the values of elements in a container. They may modify the elements of a range directly or they may modify them as they are being copied into another range.

The algorithm categories are:

- Perform an operation on each element of the range.
- Copy a range.
- Merge two ranges; swap two ranges.
- Replace each element value (or *n* elements) with a given value.
- Replace each element value (or *n* elements) with the result of an operation.
- Replace elements with a special value by another value.
- Remove elements from a range.

We now discuss these algorithms in detail.

17.4.1 Copying and Moving Elements

Copying and moving the elements of one container to another container is a common occurrence. For this reason we examine how STL supports these operations. In general, it is preferable to use them rather than creating your home-grown versions. We notice that the operations have copy and move variants. The main algorithms are:

- Copy all elements from source to destination.
- Copy only those elements from source to destination that satisfy a certain criterion.
- Copy the first *n* elements from source to destination.
- Copy source to destination, starting at the end of destination (copy backwards).
- Move all elements from source to destination.
- Move source to destination, starting at the end of destination.

Sample code to show how the algorithms work:

```
void Copy()
{
    // I. Copy algorithms
    std::cout << "\n**** Block I, Copying ranges ****\n";
    std::vector<double> v1 = { 1.0, 2.0, 3.0, 4.0 };
    print(v1, "input vector");
    std::list<double> myListA(v1.size());

    // Copy Algorithm; iterate forward in v1.
    std::copy(v1.begin(), v1.end(), myListA.begin());
    print(myListA, "Forward copy to a list");

    // copy_if() and copy_n() are C++11
    myListA.clear();
    // Copy only those elements that satisfies certain criteria
    std::copy_if(v1.begin(), v1.end(), std::inserter(myListA,
             myListA.begin()),[](const double& d) {return d > 2.0; });
    print(myListA, "Forward copy to a list");

    // Copy 1st n values of input container to output container
    myListA.clear();
    int n = 3; // Copy 3 elements
    std::copy_n(v1.begin(), n, std::back_inserter(myListA));
    print<double, std::list>(myListA, "Forward copy 3 elements to a list");

    // Copy from the back
    std::vector<int> from_vector;
    for (int i = 0; i < 10; i++) {from_vector.push_back(i);}
    print(from_vector, "input vector");
    std::vector<int> to_vector(15);
    std::copy_backward(from_vector.begin(), from_vector.end(),
                       to_vector.end());
    print(to_vector, "output vector of length 15");

    // Do something to each element of a container

    // Modifier, user-defined function
    auto MultiplyByTwo = [](auto t) { t *= 2; };
    std::for_each(v1.begin(), v1.end(), MultiplyByTwo);
    print(v1, std::string("Modifier, multiply by 2, std::for_each():"));

}
```

The output is:

```
**** Block I, Copying ranges ****
input vector: 1, 2, 3, 4,
Forward copy to a list: 1, 2, 3, 4,
Forward copy to a list: 3, 4,
Forward copy 3 elements to a list: 1, 2, 3,
```

```
input vector: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
output vector of length 15: 0, 0, 0, 0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
Modifier, multiply by 2, std::for_each(): : 1, 2, 3, 4,
```

17.4.2 Transforming and Combining Elements

The two widely applicable algorithms in this category are:

- Transform a source to a destination using a *unary* function.
- Combine elements from two sources and write the results to a destination. A *binary* function combines the elements of the sources to produce a value for the destination. In other words, it is a *scalar-valued* function of arity two. This is a useful function.
- Swap two ranges.

The sample code is:

```
void Transform()
{
    // II. std::transform Algorithms
    std::cout << "\n**** Block II, Transform ranges ****\n";
    std::vector<double> v1 = { 1.0, 2.0, 3.0, 4.0 };
    std::vector<double> v2;

    // Modifier, user-defined function
    auto Square = [](auto t) { return t*t; }; // generic lambda
    std::transform(v1.begin(), v1.end(), v1.begin(), Square);
    print(v1, std::string ("Modifier,square each element,
    std::transform(): "));

    // Modifier, predefined function objects
    std::transform(v1.begin(), v1.end(), v1.begin(),
    std::negate<double>());
    print(v1, std::string("Negated values: "));
    std::transform(v1.begin(), v1.end(), v1.begin(), [](double d)
    { return -d; });
    print(v1, std::string("Negated values: "));

    double val = 2.0;
    // Combining the elements of two sequences
    std::vector<double> v3(v1.size(), 2.1*val);
    std::transform(v1.begin(), v1.end(), v3.begin(), Square);
    print(v3, std::string("Squares of elements, I: "));

    // Squaring the elements using lambda functions
    v3.clear(); v3.resize(v1.size());
    std::transform(v1.begin(), v1.end(), v3.begin(), [](double d)
                   { return d*d; });
    print(v3, std::string("Squares of elements, II: "));
```

```
    std::cout << "\n**** Block III, Swapping ranges ****\n";
    // Swap ranges starting from a given position in v1 container
    std::vector<int> vA = {1, 2, 3, 4, 5, 6};
    std::vector<int> vB = { 9, 8, 7, 6, 5, 4, 3, 2, 1 };

    // Swap elements in a range in vA with the corresponding
    // elements in vB
    std::swap_ranges(vA.begin(), vA.end(), vB.begin()); // From L to R
    print<int, std::vector>(vA, std::string("std::vector A after swap:"));
    print<int, std::vector>(vB, std::string("std::vector B after swap:"));

}
```

The output is:

```
**** Block II, Transform ranges ****
Modifier, square each element, std::transform(): : 1, 4, 9, 16,
Negated values: : -1, -4, -9, -16,
Negated values: : 1, 4, 9, 16,
Squares of elements, I: : 1, 16, 81, 256,
Squares of elements, II: : 1, 16, 81, 256,

**** Block III, Swapping ranges ****
std::vector A after swap:: 9, 8, 7, 6, 5, 4,
std::vector B after swap:: 1, 2, 3, 4, 5, 6, 3, 2, 1,
```

We now discuss how to combine two containers to form a new one. In this case we can use a *generic lambda*:

```
void TransformII()
{ // 2 input sequences and Func to produce output sequence
    std::cout << "\n**** Block II, part II, Transform ranges ****\n";
    std::vector<double> v1 = { 1,2,3,4 };
    std::vector<double> v2(v1.size());
    std::reverse_copy(v1.begin(), v1.end(),v2.begin());
    print(v1, "first source"); print(v2, "second source");

    auto plus = [](auto x, auto y) { return x + y; };

    // Add real vectors
    std::vector<double> dest(v1.size());
    std::transform(v1.begin(), v1.end(), v2.begin(), dest.begin(), plus);
    print(dest, "destination");

    // Add complex arrays
    std::size_t N = 10;
    std::vector<std::complex<double>>
                c1(N, std::complex<double> (1.0, 0.0));
    std::vector<std::complex<double>>
                c2(N, std::complex<double> (0.0, 1.0));
```

```
    std::vector<std::complex<double>> dest2(N);
    std::transform(c1.begin(), c1.end(), c2.begin(), dest2.begin(), plus);
    print(dest2, "complex destination");
}
```

The output in this case is:

```
**** Block II, part II, Transform ranges ****
first source: 1, 2, 3, 4,
second source: 4, 3, 2, 1,
destination: 5, 5, 5, 5,
complex destination: (1,1), (1,1), (1,1), (1,1), (1,1), (1,1),
(1,1), (1,1), (1,1), (1,1),
```

17.4.3 Filling and Generating Ranges

The two algorithms in this category assign a given value to all elements of a range or assign a value to *n* elements of a range starting at a given position. It is also possible to *generate* values in a range by applying a scalar-valued function of arity zero to it:

```
void Fill()
{
    // III. Populating algorithms
    std::cout << "\n**** Block IV, Filling ranges ****\n";
    // Assigning new values
    int N = 5;
    char cval = 'A';
    std::vector<char> vecC(N, cval);
    std::fill(vecC.begin(), vecC.end(), 'z'); // Replace all values by 'z'
    print<char, std::vector>(vecC, std::string("std::vector after fill:"));

    int nFill = 3;
    // Replace nFill values from the beginning by 'Z'
    std::fill_n(vecC.begin(), nFill, 'Z');
    print<char, std::vector>(vecC,
                std::string("std::vector after refill:"));

    // Generate 1) new values, 2) overwrite existing values
    std::list<double> myList;
    int dupFactor = 4;
    std::generate_n(std::back_inserter(myList), dupFactor, std::rand);
    print(myList, std::string("generated random variables:"));

    // Modify values in myList; assign values generated by a
    // call to a lambda function
    std::generate(myList.begin(), myList.end(), []() {return 2.71; });
    print(myList, std::string("Modify generate function:"));
}
```

The output is:

```
**** Block IV, Filling ranges ****
std::vector after fill:: z, z, z, z, z,
std::vector after refill:: Z, Z, Z, z, z,
generated random variables:: 41, 18467, 6334, 26500,
Modify generate function:: 2.71, 2.71, 2.71, 2.71,
```

17.4.4 Replacing Elements

The two algorithms in this category replace a given value by another value and they replace those values that satisfy a certain condition by another value:

```
void Replace()
{
    // IV. Replacing algorithms
    std::cout << "\n**** Block V, Replacing ranges ****\n";
    std::vector<double> v1 = { 1.0, 2.0, 3.0, 4.0 };
    std::vector<double> v2;
    int M = 10;
    std::vector<char> vecChar(M, 'd');

    char oldChar = 'd';
    char newChar = 'D';
    std::replace(vecChar.begin(), vecChar.end(), oldChar, newChar);
    print<char, std::vector>(vecChar, std::string("After replace (all):"));

    vecChar[1] = vecChar[3] = 'X';

    print<char, std::vector>(vecChar, std::string("Modified
    character array:"));

    // Replace char 'X' by char 'D'
    std::replace_if(vecChar.begin(), vecChar.end(),
                    std::bind2nd(std::equal_to<char>(), 'X'), 'Z');
    print(vecChar, std::string("After replace and predicate:"));

    // Copying and replacing. This is a combination of copy and replace.
    double oldValue = 5;
    double newValue = 4;
    v1.clear();
    v1 = { 5, 1, 2, 3, 4, 5, 6, 5 };
    v2.resize(v1.size()); // Ensure that v2 is big enough to hold data.
    std::replace_copy(v1.begin(), v1.end(), v2.begin(), oldValue,
    newValue);
    print(v1, "v1 after replace_copy()");
    print(v2, "v2 after replace_copy()");

    // Replace all values > 3 and put them into v2.
    v1.clear(); v1 = { 1, 2, 3, 5, 6, 2, 7 };
    v2.clear(); v2.resize(v1.size());
```

```
    double thresholdValue = 3;
    newValue = 99;
    std::replace_copy_if(v1.begin(), v1.end(), v2.begin(),
                         std::bind2nd(std::greater<double>(),
                         thresholdValue), newValue);
    print(v2, "v2 after replace_copy_if()");
}
```

The output is:

```
**** Block V, Replacing ranges ****
After replace (all):: D, D, D, D, D, D, D, D, D, D,
Modified character array:: D, X, D, X, D, D, D, D, D, D,
After replace and predicate:: D, Z, D, Z, D, D, D, D, D, D,
v1 after replace_copy(): 5, 1, 2, 3, 4, 5, 6, 5,
v2 after replace_copy(): 4, 1, 2, 3, 4, 4, 6, 4,
v2 after replace_copy_if(): 1, 2, 3, 99, 99, 2, 99,

**** Block VI, Removing elements ****
Before remove (all):: 1, 2, 0, 2, 3, 3, 1, 3, 2, 2, 1,
After remove (all):: 2, 0, 2, 3, 3, 3, 2, 2,
After remove (criterion):: 0,
List without duplicates:: 1, 3, 5, 6,
v1 after remove_copy(): 5, 1, 2, 3, 4, 5, 6, 5,
v2 after remove_copy(): 1, 2, 3, 4, 6,
v2 after remove_copy_if(): 1, 2, 3, 2, 0, 0, 0,
unique elements: 1, 2, 3, 4, 1,
remove elements with previous greater element: 2, 13, 16,
unique elements with copy: 2, 13, 4, 16, 7,
Fixed-size arrays
```

17.4.5 Removing Elements

Removing algorithms are special modifying algorithms. They can remove the elements in a single range or they can remove the elements while they are being copied into another range. The main algorithms are:

- Remove elements with a given value or elements that match/do not match a given criterion.
- Remove adjacent duplicates.
- Copy elements while removing adjacent duplicates.

Some sample code is:

```
void Remove()
{
    // V. Remove algorithms

    // Removing elements from a range or based on a criterion
    std::cout << "\n**** Block VI, Removing elements ****\n";
    std::vector<double> v1 = { 1.0, 2.0, 3.0, 4.0 };
    std::vector<double> v2;
```

```
std::vector<int> vecInt = { 1, 2, 0, 2, 3, 3, 1, 3, 2, 2, 1 };
print<int, std::vector>(vecInt, std::string("Before remove (all):"));

int removeIntValue = 1;
std::vector<int>::iterator pos
      = remove(vecInt.begin(), vecInt.end(), removeIntValue);
vecInt.erase(pos, vecInt.end());

print(vecInt, std::string("After remove (all):"));

pos = std::remove_if(vecInt.begin(), vecInt.end(),
                     std::bind2nd(std::greater<int>(), 0));
vecInt.erase(pos, vecInt.end()); // Physical operation
print(vecInt, std::string("After remove (criterion):"));

// Removing duplicates, C arrays
int src[] = { 1, 1, 1, 3, 5, 5, 6, 6, 6 };
std::list<int> myList1;
std::copy(src, src + (sizeof(src) / sizeof(src[0])),
          back_inserter(myList1));

// remove consecutive duplicates
std::list<int>::iterator end = unique(myList1.begin(), myList1.end());
myList1.erase(end, myList1.end()); // Physical MultiplyByTwo
print(myList1, std::string("List without duplicates:"));

// Copying and removing. This is a combination of copy and remove.
double removeValue = 5; // Copy everything that is not this value
v1.clear(); v1 = { 5, 1, 2, 3, 4, 5, 6, 5 };
v2.clear(); v2.resize(v1.size());    // v2 is big enough to hold data.
std::remove_copy(v1.begin(), v1.end(), v2.begin(), removeValue);

// You need to physically replace unwanted elements.
// Remove redundant space

v2.erase(remove(v2.begin(), v2.end(), 0.0), v2.end());

print(v1, "v1 after remove_copy()");
print(v2, "v2 after remove_copy()");

// Replace all values > 3 and put them into v2.
v1.clear(); v1 = { 1, 2, 3, 5, 6, 2, 7 };
v2.clear(); v2.resize(v1.size());
double thresholdValue = 3;
int newValue = 99;
std::remove_copy_if(v1.begin(), v1.end(), v2.begin(),
                    std::bind2nd(std::greater<double>(),
                    thresholdValue));
print(v2, "v2 after remove_copy_if()");

// Removing *consecutive* duplicates. Sequence must be sorted,
// at the least elements with same value are adjacent.
```

```
    v1.clear(); v1 = { 1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 1 };
    v1.erase(std::unique(v1.begin(), v1.end()), v1.end());
    print(v1, "unique elements");

    // Remove elements if there was a previous greater element.
    v1.clear(); v1 = { 2, 13, 4, 16, 7 };
    v1.erase(unique(v1.begin(), v1.end(), std::greater<int>()), v1.end());
    print(v1, "remove elements with previous greater element");

    // Remove duplicates while copying.
    v1.clear(); v1 = { 2, 2, 13, 13, 4, 16, 7, 7 };
    v2.clear(); v2.resize(v1.size());
    v2.erase(std::unique_copy(v1.begin(), v1.end(), v2.begin()), v2.end());
    print(v2, "unique elements with copy");

}
```

The output is:

```
**** Block VI, Removing elements ****
Before remove (all):: 1, 2, 0, 2, 3, 3, 1, 3, 2, 2, 1,
After remove (all):: 2, 0, 2, 3, 3, 3, 2, 2,
After remove (criterion):: 0,
List without duplicates:: 1, 3, 5, 6,
v1 after remove_copy(): 5, 1, 2, 3, 4, 5, 6, 5,
v2 after remove_copy(): 1, 2, 3, 4, 6,
v2 after remove_copy_if(): 1, 2, 3, 2, 0, 0, 0,
unique elements: 1, 2, 3, 4, 1,
remove elements with previous greater element: 2, 13, 16,
unique elements with copy: 2, 13, 4, 16, 7,
Fixed-size arrays
```

17.5 COMPILE-TIME ARRAYS

We conclude this chapter by discussing how C arrays and compile-time arrays can be used with STL algorithms. This option may be useful for legacy applications. Some examples are:

```
void Array()
{
    std::cout << "Fixed-size arrays\n";
    const std::size_t N = 10;
    std::array<double, N> a = { 1,2,3,4 }; a[N - 1] = 3.14;
    Print(a);

    // Copy
    std::vector<double> v1(a.size());
    std::copy(std::begin(a), std::end(a), std::begin(v1));
    Print(v1);
```

```
    std::array<double, N> a2;
    for (std::size_t i = 0; i < a2.size(); ++i)
    a2[i] = static_cast<double>(i);
    Print(a2);

    // Copy
    std::copy(std::begin(v1), std::end(v1), std::begin(a2));
    Print(a2);

    // Move
    std::array<double, N> a3 = std::move(a2);
    Print(a2);
    Print(a3);

    // Removing duplicates, C arrays
    int src[] = { 1, 1, 1, 3, 5, 5, 6, 6, 6 };
    std::list<int> myList1;
    std::copy(src, src + (sizeof(src) / sizeof(src[0])),
              back_inserter(myList1));

}
```

The output is:

```
Fixed-size arrays
1,2,3,4,0,0,0,0,0,3.14,
1,2,3,4,0,0,0,0,0,3.14,
0,1,2,3,4,5,6,7,8,9,
1,2,3,4,0,0,0,0,0,3.14,
1,2,3,4,0,0,0,0,0,3.14,
1,2,3,4,0,0,0,0,0,3.14,
```

17.6 SUMMARY AND CONCLUSIONS

In this chapter we introduced two categories of algorithms; first, *non-modifying algorithms* access the elements of a range without modifying their values or their real-time order in the range while *modifying algorithms* modify the elements of a range. We provided complete machine-readable code and corresponding output. We strongly recommend that you do the exercises. We give a summary of STL as a concept map in Figure 17.1. It is assumed that these concepts are known to you.

17.7 EXERCISES AND PROJECTS

1. (Simple Binder Example)

In this exercise we examine a simple function of arity three:

```
double func3d(double x, double y, double z)
{
       return x - y + z;
}
```

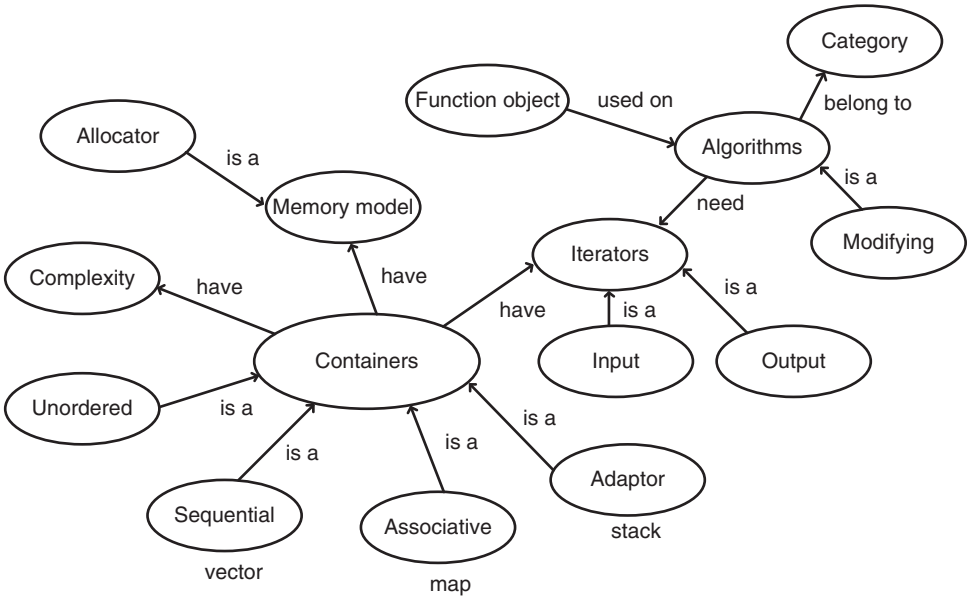


FIGURE 17.1 STL components

The objective is to experiment with `std::bind` along the lines of the examples in Section 17.2 where we discuss binding functions of arity two.

Answer the following questions:

a) Create functions of arity zero, one, two and three based on the function `func3d` and examine the output. Experiment with placeholders. Do you get the results that you expect?

b) Carry out the same exercise as in part a) by defining a function of arity three as a stored lambda function and assigning a universal function wrapper to a lambda function.

c) An alternative to `std::bind` is to model bound variables as captured variables and unbound variables as input arguments to a lambda function.

Carry out the experiment with functions of arity three. Compare binders and lambda functions with regard to readability, maintainability and efficiency. In particular, in application code we may have a choice of implementing functions that model PDEs or SDEs in a certain way and a concern is when these functions are called in loops. Are they more or less efficient than calling `virtual` or pure `virtual` functions?

d) Investigate using `std::mem_fn` (Section 17.2.3) for the examples in this exercise. Which approach do you prefer?

2. (Non-modifying Algorithms in C++11, Small Project)

In this exercise we discuss a number of new STL algorithms that are supported in C++11:

- A1: `is_permutation()`: determines whether two sequences contain the same elements, albeit in a different order. There are two variants, the second of which takes a *comparator* function. Complexity is at worst quadratic.
- A2: `is_sorted()`: determines if elements in a range are sorted. There are four variants.

- A3: `is_partitioned()`: determines whether a range is partitioned so that all elements satisfying a unary predicate condition are positioned before all elements not satisfying the predicate. An example is $\{5,3,9,1,3,2,4,8,2,6\}$ in which odd numbers are segregated from non-odd numbers.
- A4: `partition_point()`: returns the position of the first element in the partitioned range.
- A5: `is_heap()`: determines if a container is a heap (*maximum element first*). There are four variants.
- A6: `all_of()`, `any_of()`, `none_of()`: these algorithms determine whether all, any (at least one) or none of the elements are in a range by applying a unary predicate to them.

Answer the following questions:

a) Consult Josuttis (2012) and the C++ online documentation (for example, www.cppreference.com) to determine how to use algorithms A1 to A6 for simple test cases.

b) Create a vector with different elements. Create a copy and apply `std::shuffle()` to it. Determine whether the two vectors are permutations of each other.

c) Consider the container $\{97,88,95,66,55,95,48,66,35,48,55,62,77,25,38,18,40,30,26,24\}$. Is it a max heap?

d) Consider the container $\{1,2,3,4,5,6,7,8,9\}$. Determine what happens when we apply algorithm A6 to this container.

3. (`for_each()` Algorithm and Lambda)

The objective of this exercise is to adapt and extend the code in Section 17.1 in order to gain some experience with STL algorithms.

Answer the following questions:

a) Generalise the free function `Modify()` in order to remove the hard-wired (*magic*) number in the code. We wish to be able to scale a container based on a user–supplier number. Implement the new functionality using both a function object and a lambda function with a corresponding captured variable.

b) Test the new functionality using the STL algorithm `transform()`.

c) Compare the relative advantages of free functions, lambda functions and function objects in the current context with regard to code understandability and maintainability.

4. (Non-modifying Algorithms)

We include a number of exercises that process STL containers and produce some output, usually a number or a data structure. For convenience, use STL vectors as the input container and note that there are several possible solutions to a given problem, some more efficient and/or readable than others. It is up to you to decide how to design the algorithm and the structure/type of the output.

a) Count the frequency of each value in a container. For example, for $\{1,2,4,5,4,1\}$ we get the output $\{\{1,2\}, \{2,1\}, \{4,2\}, \{5,1\}\}$ as pairs of {value, frequency}. There are different ways to model the output depending on your requirements.

b) Write a function to compute the minimum, maximum, sum and average of the values of the elements in a container with numeric values. The output is a tuple with four elements.

c) Consider the container $S = \{1,2,-3,4,5,5,5,6\}$. Use an STL algorithm to find the first occurrence of the value.

5. Now use the following functions:

`std::bind2nd()`

`std::bind()`

`Lambda expression`

to find the position of the first even number in S.

d) Consider the container $S = \{1,2,5,5,-3,4,5,5,5,6,3,4,5\}$.

Determine how to do the following:

- Return the position of the first three consecutive elements having the value 5.
- Return the position of the first element of the first subrange matching $\{3,4,5\}$.
- Return the position of the first element of the last subrange matching $\{3,4,5\}$.

e) Consider the container $S = \{1,2,5,5,-3,4,5,5,5,6,3,4,5\}$. Find the first element in S that has a value equal to the value of the following element.

f) Consider the container $S = \{1,2,5,5,-3,4,5,5,5,6,3,4,5\}$ and $S1 = \{1,2,5\}$. Determine whether the elements in $S1$ are equal to the elements in S.

6. (Which Style to Use?)

Some STL algorithms use *unary* and *binary predicates*. Both predicate types return a `bool`. A *unary predicate* has one input argument while a *binary predicate* has two input arguments. We can model these predicates and other kinds of functions in a number of ways:

- User-defined function objects.
- Predefined STL function objects (for example, `std::multiplies<T>()`).
- Using lambda functions (possibly with captured variables).

Answer the following questions:

a) Compare these three solutions with regard to quality issues such as readability, understandability and maintainability.

b) Consider transforming a vector of integers into a set of integers. Only those elements whose absolute value is strictly greater than a given *threshold value* should appear in the destination. An example is the vector $\{1,2,1,4,5,5,-1\}$. If the threshold value is 2 then the output set will be $\{4,5\}$. Implement this problem using the three bespoke methods above.

c) Having developed and debugged the code in part b), review the three solutions from the perspective of understandability, maintainability and efficiency.

7. (Modifying the Elements of a Container)

We give some exercises on applying the modifying, replacing and removing algorithms in STL. The initial focus is more on functionality and less on efficiency. Most of the exercises use specific examples and it is your task to use them as the test cases for your code.

Answer the following questions.

a) Consider the set $S1 = \{a,b,c,d,e,k\}$ and the set $S2 = \{a,e\}$. Remove those elements from $S1$ that are not in $S2$. The output set is $\{b, c, d, k\}$.

b) Create a class `Point` that models two-dimensional points. Provide constructors, member functions to access the coordinates of a point and a member function to compute the distance between two points. Define a binary predicate that tests two points for equality (they have the same values for their x and y coordinates). Now create an array of points (duplicates allowed). Transform this array to a *set of points* with no duplicates. Finally, *filter* this set (that is, remove points) of those points that are not within a given distance from some predefined point.

c) We wish to create functions that process strings in some way (in a sense, we are emulating a simple version of the *Boost C++ String Algorithm* library). A string can be seen as a special kind of vector whose elements are characters. Create functions to do the following (in all cases the input is a string):
- Trim all leading and trailing blanks (space, tabs, etc.) from the string.
- Trim all leading and trailing blanks based on a unary predicate (e.g. is a digit, is a member of some set of characters).
- Produce a vector of strings from a character-separated string.
- Join two strings.

8. (How Many Ways to Print a Container?)

The objective of this exercise is to focus on the `std::for_each()` algorithm and use it to print the elements of an instance of `std::vector<double>`. We use functionality from C++11 and we attempt to print a vector in as many ways as possible. The advantage is that we can get some hands-on experience with new syntax and we can apply the results to more advanced applications.

Answer the following questions:

a) Write a function that accepts a vector as argument and that iterates over its elements using standard functionality for STL iterators.

b) Now apply the `std::for_each()` algorithm to print the elements of the vector. Use a function object and a lambda function as the unary procedure (last argument of the algorithm).

c) Apply the `std::for_each()` algorithm to modify the values of the elements of the vector. Use both a function object and a lambda function as the *unary procedure* (last argument of algorithm).

d) Consider using *range-based for loops* to print the elements of the vector:

```
void print(const std::vector<double>& vec)
{
      for (const auto& elem : vec)
      {
         std::cout << elem << ",";
      }
}
```

Consider the functionality for *uniform initialisation* and *initialiser lists* to use to initialise vectors before printing them.

e) We use the `std::for_each()` algorithm to iterate in a vector and we compute the maximum, minimum and average of the elements in the vector. Use captured variables for these three quantities and encapsulate them as a tuple whose return type is `std::tuple<double, double, double>`.

f) We generalise the function to print a vector so that it can be used with other sequence containers. To this end, use the *template–template parameter* trick.

Hint:

```
template <typename T, template <typename S,typename Alloc >
   class Container,typename TAlloc>
```

```
void print(const Container<T, TAlloc>& container,
           const std::string& comment)
{ // A generic print function for general containers

      std::cout << comment << ": ";

      // TBD
}
```

9. (Examples of Complexity)
Verify the following:

$$17n^{1/6} \in O(n^{1/5})$$
$$10^6 n^2 \in O(n^2)$$
$$n\log_2 n - 100 \in \Omega(n).$$

10. (Calculating Orders of Complexity)

a) Let $p(n)$ be any polynomial of degree n whose leading coefficient is positive and let a be any real number such that $a > 1$. Show that $p(n) \in O(a^n)$.

b) Show that $a^n \in O(b^n)$ if $1 < a < b$.

c) Show that $S(n, -k) \in \Theta(1)$ for all integers $k \geq 2$ where $S(n, -k) = \sum_{j=1}^{n} (1/j)^k$.

11. (Greatest Common Denominator (`gcd`) as a Recursive Function)

Implement and test a recursive function to compute the greatest common divisor `gcd(m,n)` of two positive integers. The code corresponding to this recursive function is given by:

```
int gcd(int m, int n)
{
       if (n == 0) return m;

       int rem = m%n;

       if (rem == 0) return n;

       return gcd(n, rem);
}
```

Answer the following questions:

a) Test the above code and compare your answers with another source, for example the *Boost Math Common Factor* library.

b) Write code to find the `gcd` of an array of integers. In Demming and Duffy (2010) we converted the array to a stack. We then pop two elements m and n from the stack, push `gcd(m,n)` onto the stack and repeat the process until there is only one element left on the stack. This value will be the `gcd` of the array.

c) How do you debug this recursive function?

12. (Bessel Functions of the First Kind)

These functions are important in mathematical physics and they can be computed as an infinite series involving the gamma function. In this exercise, however, we compute these functions using the recurrence relationship:

$$J_{n+1}(x) = \frac{2n}{x} J_n(x) - J_{n-1}(x), \quad n \geq 1$$

and we assume that the initial Bessel functions $J_1(x)$ are available in a library (for example, the *Boost Math Toolkit*; see Demming and Duffy, 2012). The objective of this exercise is to compute Bessel functions of the first kind based on the above recurrence relation and to compare the accuracy and efficiency of your results with those by running the code for Bessel functions in the *Boost Math Toolkit*.

13. (Sets and Multisets)

This is a review question.

a) Enumerate the most important operations that can be applied to sets. Are these operations supported in STL?

b) The *power set* of a set S (denoted by $Power(S)$) is the set of all subsets of S. How many elements does $Power(S)$ have if S has n elements? Find $Power(S)$ if $S = \{a, b, c, d\}$.

c) Which data containers would you use if you wish to model i) all letters in the English language and ii) all letters in all languages?

d) In general, what are the advantages of using sets in C++ applications? For example, when would it be better to use a set instead of a list or a vector?

14. (Lists versus Vectors)

Lists and vectors are similar in structure but it is important to know the context in which to use each one. Determine whether it is better to use a list, a vector or a deque in the following cases:

a) Iterating in a container from the beginning to the end and calculating the average of all its elements.

b) Moving to a given (integral) position in the container.

c) Appending and removing elements at the beginning and end of the container.

d) Finding the position in the container corresponding to an element with a given value.

e) Using the container as a building block for creating a container to model matrices.

Determine the complexity for each of the above operations. Consider using a map or an unordered map. Would they help in improving performance?

15. (Traffic Flow)

Which containers are most suitable for the following cases and applications?

a) Football supporters entering a football stadium via a turnstile. There are multiple turnstiles at the stadium.

b) Washing dishes in a restaurant and piling them up so that they can be reused for the next group of customers.

c) A queue at an airport in which passengers who precheck can proceed directly to the boarding gate.

d) Computing moving averages based on real-time data feeds.

17.8 APPENDIX: REVIEW OF STL CONTAINERS AND COMPLEXITY ANALYSIS

In applications we need to create, store and retrieve data. Data will be stored in various ways depending on the context. Knowing which particular data structure to use is important. In this section we give an overview of the STL containers and container types.

17.8.1 Sequence Containers

These containers are *ordered collections* in which each element has a certain position in its respective container. The position is independent of the element value but it may depend on the time and place of the insertion. There are five predefined sequence container classes in STL:

- `array`: models a fixed-size array by wrapping a static C-style array. It is a sequence of elements and the array has constant size. The values are stored on the stack.
- `vector`: models a dynamic array. It manages its elements with a dynamic C-style array. The values are stored on the heap.
- `deque`: similar to a vector because its interface is almost the same as that of a vector. Whereas a vector is open at one end only, a deque is open at both ends.
- `list`: a list manages its elements as a *doubly linked list*. In other words, each element in the list has a value as well as pointers to its previous and next elements in the list.
- `forward_list`: this container was introduced in C++ 11. It manages its elements as a *singly linked list*.

In general, sequence containers are implemented as arrays or linked lists.

17.8.2 Associative Containers

Associative containers are sorted containers in which the position of an element depends on its value (or its key). There are four predefined associative container classes in STL:

- `set`: this is the implementation of the *set type* in mathematics. No duplicates are allowed in sets.
- `multiset`: similar to sets but duplicates are allowed. This data structure is sometimes called a *bag*.
- `map`: manages key/value pairs as elements. Maps do not allow duplicates.
- `multimap`: manages key/value pairs as elements. Multimaps allow duplicate keys.

In general, associative containers are implemented as *binary trees*.

17.8.3 Unordered (Associative) Containers

These are unordered collections in which the position of an element is not of importance. Conceptually, these collections contain all the elements that are inserted in an arbitrary order.

These containers are similar to a bag in the sense that there is no sorting criterion. In contrast to sequence containers there are no semantics to put an element into a specific position. There are four predefined unordered container classes in STL:

- `unordered_set`.
- `unordered_multiset`.
- `unordered_map`.
- `unordered_multimap`.

In general, unordered containers are implemented as *hash tables*.

17.8.4 Special Containers

STL has support for a number of containers that are needed in certain applications. These are called *container adapters* because they are wrappers for STL containers. They hide these latter containers in their implementation and they provide special interfaces to clients. This design is similar to that taken by the *Adapter* design pattern (see GOF, 1995). Finally, STL also supports the class `bitset<size_t N>` that models fixed-size arrays of bits.

In general, only specialised kinds of applications will need to use these containers and they are probably not needed for most development work.

17.8.5 Other Data Containers

In the interests of completeness, we give a brief discussion of some other containers in C++:

- *String*: STL supports the string classes `basic_string`, `string` and `wstring`. A string is a vector whose elements are characters. We do not discuss strings in this book, although they have many applications. See Josuttis (2012) for a comprehensive treatment. See also the *Boost String Algorithm* library.
- *C-Style Array*: these arrays are not classes nor are they STL containers because they do not have the member functions that we expect of STL containers. We do not discuss C-style arrays in this book.
- STL has support for a number of low-level and commonly needed containers and related algorithms. In general, we recommend using them rather than writing our own versions. However, application developers may need more functionality than what STL has to offer. Some use cases are:
  - U1: Create a user-defined container whose interface allows us to directly use STL algorithms (for example, vectors in Boost *uBlas* have this property).
  - U2: Create a user-defined container that uses the functionality of other STL containers.
  - U3: Create application-specific containers and algorithms.

In general, we extend the functionality in STL by using the HASA relationship, such as *composition* and *aggregation*. We might also be tempted to extend the behaviour of STL containers by deriving from them and adding behaviour (see Josuttis, 2012, p. 251). This is considered to be bad practice.

17.8.6 Complexity Analysis

When developing and using algorithms we need to have some way to analyse them. In other words, we must have some criteria to measure the *efficiency of algorithms.* To this end, let M be an algorithm containing n elements. Then one way to determine the efficiency of M is to measure the space and time used by it. For example, we can measure time by counting the number of *key operations*:

- For sorting and searching algorithms we can count the number of comparisons.
- In a numerical algorithm we can count the number of multiplications.

Key operations are defined in such a way that they take more time to compute than other operations. For example, division is a key operation and addition is not a key operation because addition is less computationally intensive than division. Finally, we measure space by counting the memory needed by an algorithm.

Definition 17.1 The *complexity* of an algorithm M is a function $f(n)$ that describes the running time or the storage space requirements of M in terms of the size n of the input set.

In general, the 'complexity of algorithm M' refers to the running time of M. The *complexity function* $f(n)$ gives the running time of M and it depends on the input size n and on the kind of data being used. Summarising, the problem of determining the complexity of an algorithm reduces to one of analysing the properties of its corresponding complexity function. For example, we might like to bound $f(n)$ above and below by other functions $g(n)$ and $h(n)$, respectively when n goes to infinity. In particular, we are interested in two specific cases:

- *Worst case*: the maximum value of $f(n)$ for input data of any size.
- *Average case*: the *expected value* of $f(n)$. This case can be analysed by probability theory.

We now give a detailed mathematical account of complexity analysis.

17.8.7 Asymptotic Behaviour of Functions and Asymptotic Order

In order to describe algorithmic complexity, we restrict our attention to real-valued functions $f : \mathbb{N} \to \mathbb{R}$. In other words, the domain is the set of non-negative integers $\mathbb{N}$ while the range is the set of real numbers $\mathbb{R}$. We are interested in the kinds of functions that are *eventually positive*, that is there exists a positive integer n_0 (depending on f) such that $f(n) > 0$ for all $n > n_0$. We let $\mathfrak{F}$ denote the space of all functions having this property. We now set up a mathematical notation to compare functions in this space $\mathfrak{F}$.

Definition 17.2 Given a function $g \in \mathfrak{F}$ we define $\Theta(g(n))$ to be the space of all functions $f \in \mathfrak{F}$ such that:

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \tag{17.1}$$

where c_1, c_2 and n_0 are positive constants and $n \geq n_0$.

If $f \in \Theta(g(n))$ we say that $f(n)$ is *big theta* of $g(n)$. Informally, we say that f is bounded above and below by g.

Intuitively, we say that the function f grows at the same rate as the function g and it effectively gets squeezed between two constant multiples of g. Graphing the functions in inequalities (17.1) is possible but we need to extend the domain of these functions from the set of positive integers to the set of positive real numbers.

We take some examples:

$$n^3 + 2n\log_2 n - 3 \in \Theta\left(n^3\right)$$
$$3\sqrt{n} + \log_2 n - 3 \in \Theta\left(\sqrt{n}\right).$$

Definition 17.3 Two functions $f, g \in \mathfrak{F}$ have the *same order* if $f \in \Theta(g(n))$.

Many algorithms in computer science have linear, quadratic, cubic, logarithmic, exponential or $n \log n$ complexities. In particular, we are interested in the following measure.

Definition 17.4 Given a function $g \in \mathfrak{F}$ we define $O(g(n))$ to be the set of all functions $f \in \mathfrak{F}$ that satisfy the inequality:

$$f(n) \leq cg(n) \tag{17.2}$$

for positive constants c and n_0 for all $n \geq n_0$.

If $f \in O(g(n))$ we say that f is *big oh* of g. Informally, f is bounded above by g.

Some examples are:

$$7n^2 - 9n + 4 \in O(n^2)$$
$$\log_4 n \in O(n^2).$$

We can now produce an increasing chain in the hierarchy of orders as follows:

$$O(1), O(\log n), O(\sqrt{n}), O(n), O(n \log n), O(n^2), O(n^3), O(2^n), O(n2^n), O(n!).$$

We thus see that the factorial function increases most quickly.

For example, the complexity of some well-known searching and sorting algorithms is:

- Linear search: $O(n)$.
- Binary search: $O(\log n)$.
- Bubble sort: $O\left(n^2\right)$.
- Merge sort: $O(n \log n)$.

Definition 17.5 Given a function $g \in \mathfrak{F}$ we define $\Omega(g(n))$ to be the set of all functions $f \in \mathfrak{F}$ that satisfy the inequality:

$$cg(n) \leq f(n). \tag{17.3}$$

If $f \in \Omega(g(n))$ we say that f is *big omega* of g. Informally, f is bounded below by g.

In general, big oh is often used as an upper bound on the performance of a particular algorithm for solving a problem, while big omega is often used as a lower bound for the complexity of the problem itself.

17.8.8 Some Examples

We conclude this section with some examples. The first example discusses linear search in a one-dimensional data container. In other words, given a linear array of length n we wish to determine if a specific value y is in the array. Success could be indicated by the position of y in the array (this is a value in the closed range $[1, n]$) or zero, in which case the value y is not in the array. Assuming that we search from index 1, the worst case occurs either when the last element in the array contains y in the last position or the array does not contain y. Then $f(n) = n$ is the worst-case complexity of the linear search algorithm. The average case assumes a certain probabilistic distribution of the input data. We assume that the possible permutations of the data set are equally likely. More generally, assume that the numbers $n_1, n_2, \ldots, n_k$ occur with probability $p_1, p_2, \ldots, p_k$. Then the *expectation* or *average value* E is given by:

$$E = \sum_{j=1}^{k} n_j p_j. \tag{17.4}$$

Returning to the current problem, we see that the value y is equally likely to occur at any position in the array. Thus, the number of comparisons can be any of the numbers $1, 2, \ldots, n$ and each number occurs with equal probability $p = 1/n$. Hence, the average complexity is a special case of equation (17.4) and is given by:

$$E = \sum_{j=1}^{n} j/n = \frac{n+1}{2}. \tag{17.5}$$

Intuitively, equation (17.5) states that the average number of comparisons needed to find the location of the value y is approximately equal to half the number of elements in the array.

The second example is to examine the sum:

$$L(b, n) = \sum_{j=1}^{n} \log_b(j) = \log_b(n!) \tag{17.6}$$

for any base b. For example, $L(2, n)$ is a lower bound for the worst-case complexity of any comparison-based sorting algorithm. We now claim that:

$$\log_b(n!) \in \Theta(n \log_b n). \tag{17.7}$$

Clearly, $L(b, n) \in O(n \log_b n)$ because $\sum_{j=1}^{n} \log_b(j) \leq \sum_{j=1}^{n} \log_b(n) = n \log_b(n)$.

We now show that $L(b,n) \in \Omega(n\log_b n)$. Let x be a variable and let $\lfloor x \rfloor$ be the largest integer not greater than x. Let $m = \lfloor n/2 \rfloor$. Then:

$$L(b,n) = \sum_{j=1}^{m} \log_b(j) + \sum_{j=m+1}^{n} \log_b(j) \geq \sum_{j=m+1}^{n} \log_b(j) \geq (n-m)\log_b(m+1)$$
$$\geq (n/2)\log_b(n/2) = (n/2)(b\log_b n - \log_b 2) \geq \left(\frac{n}{2}\right)\left(\log_b n - \frac{1}{2}\log_b n\right) \text{ for all } n \geq n_0.$$

Hence, $L(b,n) \geq \frac{1}{4} n\log_b n$ for all $n \geq n_0$ and we thus conclude that $L(b,n) \in \Omega(n\log_b n)$. Hence, relationship (17.7) has been proved.

For more information on complexity analysis of sequential, parallel and distributed algorithms, see Berman and Paul (2005).

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 18

STL Algorithms Part II

18.1 INTRODUCTION AND OBJECTIVES

In this chapter we continue with our discussion of the STL algorithms that we introduced in Chapter 17. In particular, we discuss the following major algorithm categories:

- *Mutating algorithms*: these algorithms change the order of elements (and not their values) in a range by assigning and swapping their values in some way. Example: reverse the order of elements in a container.
- *Sorting algorithms*: these are special mutating algorithms because they change the order of elements in a range. When choosing a particular sorting algorithm we must take complexity into account. Example: sort a container using *heapsort*. This algorithm has $O(n \log n)$ complexity.
- *Sorted range algorithms*: sorted range algorithms require that the range on which they operate be sorted according to their *sorting criterion*. Example: process the sorted union of two ranges.
- *Numeric algorithms*: these algorithms combine numeric elements in different ways. Although this category only contains four algorithms they can be used by providing them with many kinds of functions, thus allowing developers to customise them. Example: compute the inner product of two ranges or the inner product of two containers.

As in Chapter 17, we discuss the intent of each category and we describe its algorithms using compact examples. We also show the output produced by each algorithm in order to help us understand the code. This chapter can be used as a reference; you can take the example code and adapt it to your own application.

18.2 MUTATING ALGORITHMS

These algorithms change the order of elements (and not their values) in a container by assigning and swapping their values in some way. The main algorithms are:

- Reverse the order of the elements in a range (with and without copy).
- Rotate the elements.

- Permute the elements.
- Bring elements into a random order.
- Change the order of elements so that elements that match a criterion are at the front of the range.

18.2.1 Reversing the Order of Elements

The two algorithms in this category reverse the order of the elements in a range, either by modifying the range or by creating a new range. The *caller must ensure that the destination range has enough capacity* to hold the elements or alternatively you can use an *insert iterator*. If not, a run-time error will occur. In the case of lists we note that using the `reverse()` member function is more efficient than the corresponding STL algorithm because it relinks element pointers instead of assigning element values.

An example of use is:

```
void Reverse()
{
    std::cout << "\n**** Block I, Reverse ****\n";
    int size = 6;
    double val = 2.0;
    std::vector<double> vec(size, val);
    vec[0] = 2.0; vec[vec.size() - 1] = -23.4;
    print(vec, std::string("Original vector: "));

    //************************************************
    // I. Reversing the order of elements
    std::reverse(vec.begin(), vec.end());
    print(vec, std::string("Modifier, vector reversed: "));

    // Reverse order of elements while copying them.
    std::vector<double> vec2;

    // Ensure that vec2 can hold the data or use an insert iterator
    std::reverse_copy(vec.begin(), vec.end(), std::back_inserter(vec2));
    print(vec2, std::string("Selector(copy), vector reversed: "));
}
```

The output is:

```
**** Block I, Reverse ****
Original vector: : 2, 2, 2, 2, 2, -23.4,
Modifier, vector reversed: : -23.4, 2, 2, 2, 2, 2,
Selector(copy), vector reversed: : 2, 2, 2, 2, 2, -23.4,
```

18.2.2 Rotating Elements

We can rotate the elements of a range *to the left* or *to the right* so that a given element becomes the new first element. An example is:

```
void Rotate()
{
    // II. Rotations
    std::cout << "\n**** Block II, Rotate ****\n";
    // Rotating elements in a sequence
    using namespace boost::assign;
    std::vector<int> myVec = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    print(myVec, "input vector");

    //std::cout << "Choose size of offset to right: "; int offset;
    // std::cin >> offset;
    int offset = 3;

    offset = std::abs(offset);
    std::rotate(myVec.begin(), myVec.begin() + offset, myVec.end());
    print(myVec, std::string("Modifier, vector rotated "));

    // Rotate with copy
    std::multiset<int> mySet = { 11, 22, 33, 44, 55, 66, 77, 88, 99 };
    std::cout << "Input multiset: ";
    for (auto el : mySet)
    {
       std::cout << el << ",";
    }
    std::cout << '\n';
    std::multiset<int>::iterator pos = mySet.begin();
    std::advance(pos, offset);    // increment the position of an iterator

    std::vector<int> vecOut;
    std::rotate_copy(mySet.begin(), pos, mySet.end(),
                     std::back_inserter(vecOut));
    print(vecOut, std::string("Selector, set rotated "));

    // Rotate from position having a given value
    std::multiset<int>::iterator posValue = mySet.find(55);
    vecOut.clear();

    std::rotate_copy(mySet.begin(), posValue, mySet.end(),
    std::back_inserter(vecOut));
    print(vecOut, std::string("Selector, set rotated: "));

}
```

The output is:

```
**** Block II, Rotate ****
input vector: 1, 2, 3, 4, 5, 6, 7, 8, 9,
Modifier, vector rotated : 4, 5, 6, 7, 8, 9, 1, 2, 3,
Input multiset: 11,22,33,44,55,66,77,88,99,
Selector, set rotated : 44, 55, 66, 77, 88, 99, 11, 22, 33,
Selector, set rotated: : 55, 66, 77, 88, 99, 11, 22, 33, 44,
```

$$\epsilon = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}, \quad \sigma_2 = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix}, \quad \phi_1 = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix}$$

$$\sigma_1 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix}, \quad \sigma_3 = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix}, \quad \phi_2 = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix}$$

FIGURE 18.1 **Permutations of S_3**

18.2.3 Permuting Elements

Let $S = \{1, 2, \ldots, n\}$ and consider the set S_n of all $n!$ permutations of these n symbols (no significance should be given to the fact that the elements are natural numbers because the set can contain other kinds of entities). A *permutation operation* (or *permutation* for short) is a one-to-one mapping of S_n onto itself. For example, when $n = 3$ we can map 1 to 2, 2 to 3 and 3 to 1. You can verify that there are $3! = 3 \times 2 \times 1 = 6$ possible permutations in total. In general, we say that S_n is the *symmetric group* of order n. In other words, it forms a group under the composition of permutations. For example, the symmetric group S_3 has 6 elements, as shown in Figure 18.1 (Knuth, 1997; Lipschutz and Lipson, 1997). The multiplication table for this group is shown in Figure 18.2.

A permutation that involves the interchange of only two elements of the symmetric group is called a *transposition*. For example, σ_1, σ_2 and σ_3 are transpositions. A permutation is *even* (*odd*) if it can be expressed as a product of an even (odd) number of transpositions. For example, the notation (12) means that 1 and 2 are exchanged while the other elements remain in their place. Let us take $n = 5$; the notation (12345) means that 1→2, 2→3, 3→4, 4→5, 5→1 and this even permutation can be written as a product (from left to right) of the transpositions (12)(13)(14)(15). You should check this by hand.

We now give some examples of how STL supports permutations. We assume that the container is sorted in ascending order:

```
void Permutation()
{
    // III. Permutations
    std::cout << "\n**** Block III, Permutations ****\n";
    // Permuting elements
    std::vector<int> vecA = { 1, 2, 3 };
```

	ϵ	σ_1	σ_2	σ_3	ϕ_1	ϕ_2
ϵ	ϵ	σ_1	σ_2	σ_3	ϕ_1	ϕ_2
σ_1	σ_1	ϵ	ϕ_1	ϕ_2	σ_2	σ_3
σ_2	σ_2	ϕ_2	ϵ	ϕ_1	σ_3	σ_1
σ_3	σ_3	ϕ_1	ϕ_2	ϵ	σ_1	σ_2
ϕ_1	ϕ_1	σ_3	σ_1	σ_2	ϕ_2	ϵ
ϕ_2	ϕ_2	σ_2	σ_3	σ_1	ϵ	ϕ_1

FIGURE 18.2 **Multiplication table for S_3**

```
    print(vecA, "input vector");

    while (std::next_permutation(vecA.begin(), vecA.end()) == true)
    {
       print(vecA, std::string("next permutation I, vecA: "));
    }
    while (std::next_permutation(vecA.begin(), vecA.end()))
    {
       print(vecA, std::string("previous permutation II, vecA: "));
    }

    // Creating permutations with vectors; note Boost String Algo library
    std::string myString("A,B,C,D"); // sorted ASC,generate 23 = 4! - 1
    std::cout << "\ninput string " << myString << '\n';
    std::vector<std::string> vecB;
    boost::split(vecB, myString, boost::is_any_of(","));

    std::vector<std::string> resultSet;
    std::string str;

    print(vecB, std::string("Original set, char: "));

    // Create all permutations of {A,B,C,D}; It will be of size 4! = 24,
    // including the identity permutation
    while (std::next_permutation(vecB.begin(), vecB.end()))
    { // Sort ASCending

       str = boost::join(vecB, ",");
       resultSet += str;
       print(vecB, std::string("next permutation : "));
    }
    std::cout << "Now the result of size: " << resultSet.size() << "\n";
    std::set<std::string> resultSet2;
    std::copy(resultSet.begin(), resultSet.end(),
    std::inserter(resultSet2, resultSet2.begin()));
    std::for_each(resultSet2.begin(), resultSet2.end(),
             [](std::string s) { std::cout << s << "} {"; });

    // Creating permutations with sets
    std::string myString2("A,B,C");
    std::cout << "input string " << myString2 << '\n';
    std::set<std::string> vecB2;
    boost::split(vecB2, myString2, boost::is_any_of(","));

    std::cout << "Now the result of size: " << resultSet2.size() << "\n";
}
```

The output is:

```
**** Block III, Permutations ****
input vector: 1, 2, 3,
```

```
next permutation I, vecA: : 1, 3, 2,
next permutation I, vecA: : 2, 1, 3,
next permutation I, vecA: : 2, 3, 1,
next permutation I, vecA: : 3, 1, 2,
next permutation I, vecA: : 3, 2, 1,
previous permutation II, vecA: : 1, 3, 2,
previous permutation II, vecA: : 2, 1, 3,
previous permutation II, vecA: : 2, 3, 1,
previous permutation II, vecA: : 3, 1, 2,
previous permutation II, vecA: : 3, 2, 1,
input stringA,B,C,D
Original set, char: : A, B, C, D,
next permutation : : A, B, D, C,
next permutation : : A, C, B, D,
next permutation : : A, C, D, B,
next permutation : : A, D, B, C,
next permutation : : A, D, C, B,
next permutation : : B, A, C, D,
next permutation : : B, A, D, C,
next permutation : : B, C, A, D,
next permutation : : B, C, D, A,
next permutation : : B, D, A, C,
next permutation : : B, D, C, A,
next permutation : : C, A, B, D,
next permutation : : C, A, D, B,
next permutation : : C, B, A, D,
next permutation : : C, B, D, A,
next permutation : : C, D, A, B,
next permutation : : C, D, B, A,
next permutation : : D, A, B, C,
next permutation : : D, A, C, B,
next permutation : : D, B, A, C,
next permutation : : D, B, C, A,
next permutation : : D, C, A, B,
next permutation : : D, C, B, A,
Now the result of size: 23
{A,B,D,C} {A,C,B,D} {A,C,D,B} {A,D,B,C} {A,D,C,B} {B,A,C,D}
{B,A,D,C} {B,C,A,D} {B,C,D,A} {B,D,A,C} {B,D,C,A} {C,A,B,D}
{C,A,D,B} {C,B,A,D} {C,B,D,A} {C,D,A,B} {C,D,B,A} {D,A,B,C}
{D,A,C,B} {D,B,A,C} {D,B,C,A} {D,C,A,B} {D,C,B,A}
input string A,B,C
Now the result of size: 23
```

Permutations are a very important tool in the analysis of algorithms and in pure mathematics (for example, group theory and its applications). See Knuth (1997) for more details.

18.2.4 Shuffling Elements

The algorithms in this category shuffle the order of the elements in a range:

```
void Shuffle()
{
    // IV. Shuffling
    std::cout << "\n**** Block IV, Shuffling ****\n";
    // Shuffle
    std::vector<int> vecR = { 1, 2, -2, 3, -3, 4, -4, 5, -5, 8, 9, 7 };

    std::random_shuffle(vecR.begin(), vecR.end());
    print(vecR, std::string("Modified, random shuffle: "));

    // C++11
    std::default_random_engine eng;
    std::shuffle(vecR.begin(), vecR.end(), eng);
    print(vecR, std::string("vecR after shuffle"));

    // Moving elements to the front
    std::vector<int> vecP = { 1, 2, 3, 4, 5, 6, 2, 7, 8, 9 };
    print(vecP, "vecP");
}
```

The output is:

```
**** Block IV, Shuffling ****
Modified, random shuffle: : 9, 2, 8, -2, 1, 7, 5, 3, -3, -4, -5, 4,
vecR after shuffle: 1, 9, -2, 2, -5, 3, 5, 4, -4, -3, 7, 8,
vecP: 1, 2, 3, 4, 5, 6, 2, 7, 8, 9,
```

These algorithms are useful when shuffling randomly generated numbers.

18.2.5 Creating Partitions

There are three algorithms in this category:

- Return the first position for which a unary predicate returns `false` (two functions).
- Partition a range into two subranges.

A code sample is:

```
void RangePartition()
{
    // V. Partitiong a range
    std::cout << "\n**** Block V, Moving elements to front ****\n";
    std::vector<int> vecP = { 1, 2, 3, 4, 5, 6, 2, 7, 8, 9 };
    print(vecP, "vecP");

    // Partitioning: moving all the elements in a range to the front.
    // All elements greater than 6; relative order not preserved
    std::vector<int>::iterator posP = std::partition(vecP.begin(), vecP.end(),
```

```
    std::bind2nd(std::greater_equal<int>(), 5));
    print(vecP, std::string("Partitioned,relative order not
    preserved: "));
    std::cout << "Position 1st element: " << *posP << std::endl;

    // All elements greater than 6; relative order _is_ preserved
    posP = std::stable_partition(vecP.begin(), vecP.end(),
                                 std::bind2nd(std::greater_equal<int>(),5));
    print(vecP, std::string("Partitioned, relative order preserved: "));
    std::cout << "Position 1st element: " << *posP << std::endl;

    // Split elements in a range into two subranges based on a predicate
    std::vector<int> coll = { 1, 2, 3, 5, 6, 7, 9, 11 };

    std::vector<int> evenColl;
    std::vector<int> oddColl;

    std::partition_copy(coll.begin(), coll.end(),
                        std::back_inserter(evenColl),
                        std::back_inserter(oddColl),
                        [](int elem) { return elem % 2 == 0; });

    std::cout << "Even and odd sizes: "
              << evenColl.size() << ", " << oddColl.size() << std::endl;

    // Two sets, above barrier H
    double H = 100.0;
    std::vector<double> stock = { 1, 2, 3, 6, 7, 9, 11, 30, 50, 60, 110, 120, -1 };
    print(stock, "input values");
    std::vector<double> outsideColl;
    std::vector<double> insideColl;

    auto constraint = [&H](double x) {return x > H; };
    std::partition_copy(stock.begin(), stock.end(),
                        std::back_inserter(outsideColl),
                        std::back_inserter(insideColl),
                        constraint);

    print(outsideColl, "Outside values");
    print(insideColl, "Inside values");
}
```

The output is:

```
**** Block V, Moving elements to front ****
vecP: 1, 2, 3, 4, 5, 6, 2, 7, 8, 9,
Partitioned, relative order not preserved: 9,8,7,6,5,4,2,3,2,1,
Position 1st element: 4
Partitioned, relative order preserved: 9, 8, 7, 6, 5, 4, 2, 3, 2, 1,
Position 1st element: 4
```

```
Even and odd sizes: 2, 6
input values: 1, 2, 3, 6, 7, 9, 11, 30, 50, 60, 110, 120, -1,
Outside values: 110, 120,
Inside values: 1, 2, 3, 6, 7, 9, 11, 30, 50, 60, -1,
```

18.3 NUMERIC ALGORITHMS

These algorithms combine numeric elements in range:

- Combine all elements in some way (for example, sum, product).
- Inner products using different kinds of operators.
- Adjacent difference; combine each element with its predecessors.
- Partial sum; combine each element with all of its predecessors.

18.3.1 Accumulating the Values in a Container Based on Some Criterion

This is an algorithm to sum the elements of a range. In this way we can aggregate the elements of a range to produce mathematically relevant properties such as sums, averages, norms and statistical properties, for example. Some examples are:

```
void Accumulate()
{
    // I. Accumulate
    std::cout << "\n**** Block I, Accumulate ****\n";
    std::vector<int> vec = { 1,2,3,4,5 };
    // Accumulate: Computing the result of one sequence.
    int initVal = 1;        // Must be initialised to a value; Add
                            // initVal to all elements
    int acc1 = std::accumulate(vec.begin(), vec.end(), initVal);
    std::cout << "Sum 1, classic sum: " << acc1 << std::endl;

    // A. Multiply (by predefined function object) each element by initVal
    int acc2 = std::accumulate(vec.begin(), vec.end(), initVal,
                               std::multiplies<int>());
    std::cout << "Sum 2, predefined FO: " << acc2 << std::endl;

    // B. C++11 Lambda style
    int acc3 = initVal;
    std::for_each(vec.begin(), vec.end(), [&](int i) -> double
    {
       acc3 *= i; return acc3;
    });
    std::cout << "Sum 3, embedded lambda: " << acc3 << std::endl;

    // C. Using a 'reusable' Lambda function and new 'auto' variable decl
    auto MyMultiply = [](auto x, auto y) { return x*y; };
```

```
    int acc4 = std::accumulate(vec.begin(), vec.end(), initVal,
                               MyMultiply);
    std::cout << "Sum 4, generic lambda: " << acc4 << std::endl;

    // User-defined function object
    int accA = accumulate(vec.begin(), vec.end(), initVal, FOMultiply());
    std::cout << "Sum 5, with user-defined function object: " << accA;

}
```

The output is:

```
**** Block I, Accumulate ****
Sum 1, classic sum: 16
Sum 2, predefined FO: 120
Sum 3, embedded lambda: 120
Sum 4, generic lambda: 120
Sum 5, with user-defined function object: 120
```

18.3.2 Inner Products

The concept of the *inner* (or *dot*) product of vectors is well known in mathematics. For example, let $(a_0, \ldots, a_J)$ and $(b_0, \ldots, b_J)$ be two containers. Then the following properties (and more!) are supported:

$$\begin{cases} \prod_{j=0}^{J} (a_j \pm b_j), & \prod_{j=0}^{J} a_j * b_j \\ \sum_{j=0}^{J} (a_j \pm b_j), & \sum_{j=0}^{J} a_j * b_J. \end{cases} \tag{18.1}$$

More generally, we can define two operators (let's call them $op1$ and $op2$). Then the algorithm for an inner product would take the form:

$$\text{init Val } op1(a_0 op2 b_0) \ldots op1(a_J op2 b_J). \tag{18.2}$$

Some examples are:

```
void InnerProduct()
{
    // II. Inner products
    std::cout << "\n**** Block II, Inner Products ****\n";
    std::vector<int> vec = { 1,2,3 };

    // Sums of products + initVal
    int initVal = 0;
```

```
    int ip1 = std::inner_product(vec.begin(), vec.end(), vec.begin(),
    initVal);
    std::cout << "Inner product 1: " << ip1 << std::endl; // 14

    // Two predefined function objects using inner and outer operations
    double ip2 = std::inner_product(vec.begin(), vec.end(), vec.begin(), 1,
                                    std::multiplies<int>(),       // outer
                                    std::plus<int>());            // inner
    std::cout << "Inner product 2: " << ip2 << std::endl;         // 48

    double ip3 = inner_product(vec.begin(), vec.end(), vec.begin(), 0,
                               std::plus<int>(),                  // outer
                               std::multiplies<int>());           // inner
    std::cout << "Inner product 3: " << ip3 << std::endl;         // 14

    // Sum of max values
    std::vector<int> vec1 = { 1,-2,3,-4,5 };
    std::vector<int> vec2 = { -1,2,-3,10,-5 };

    auto myMax = [](auto x, auto y) {return std::max(x,y); };
    auto myPlus = [](auto x, auto y) {return x+y; };
    double ip4 = inner_product(vec1.begin(), vec1.end(), vec2.begin(), 0,
                               myPlus,                            // outer
                               myMax);                            // inner
    std::cout << "Sum of max values: " << ip4 << std::endl;       // 21

    double ip5 = inner_product(vec1.begin(), vec1.end(), vec2.begin(), 0,
                               myMax,                             // outer
                               myPlus);                           // inner
    std::cout << "Max of summed values: " << ip5 << std::endl;    // 6
}
```

The output is:

```
**** Block II, Inner Products ****
Inner product 1: 14
Inner product 2: 48
Inner product 3: 14
Sum of max values: 21
Max of summed values: 6
```

18.3.3 Partial Sums

The two algorithms in this category convert *relative values* into *absolute values*. An example is:

```
void PartialSum()
{
    // III. Relative and absolute values conversions: partial sums
    // Converting relative values into absolute values
```

```
    std::cout << "\n**** Block III, Partial sums ****\n";
    std::vector<int> vec = { 1,2,3,4,5 };
    std::partial_sum(vec.begin(), vec.end(), vec.begin());
    print(vec, std::string("partial sum: "));

    // Call multiply for every element in the source (vec)
    std::vector<int> vec2(vec.size());
    print(vec2, std::string("input vector: "));
    partial_sum(vec.begin(), vec.end(), vec2.begin(),
                std::multiplies<int>());
    print(vec2, std::string("partial sum with mult. operator: "));
}
```

The output is:

```
**** Block III, Partial sums ****
partial sum: : 1,3,6,10,15,
input vector: : 0,0,0,0,0,
partial sum with mult. operator: : 1,3,18,180,2700,
```

18.3.4 Adjacent Difference

The two algorithms in this category convert *absolute values* into *relative values*:

```
void AdjacentDifference()
{
    // IV. Converting absolute values into relative values.
    std::cout << "\n**** Block IV, Adjacent differences ****\n";
    // Converting absolute values into relative values
    std::vector<int> vecA(6);
    for (int j = 0; j < vecA.size(); ++j) { vecA[j] = j + 1; }
    std::vector<int> vecB(vecA.size());

    print(vecA, std::string("input vector to adjacent_difference"));
    std::adjacent_difference(vecA.begin(), vecA.end(), vecB.begin());
    print(vecB, std::string("adjacent difference 1: "));

    std::adjacent_difference(vecA.begin(), vecA.end(), vecB.begin(),
                             std::plus<int>());
    print(vecB, std::string("adjacent difference 2: "));

    std::adjacent_difference(vecA.begin(), vecA.end(), vecB.begin(),
                             std::multiplies<int>());
    print(vecB, std::string("adjacent difference 2: "));

}
```

The output is:

```
**** Block IV, Adjacent differences ****
input vector to adjacent_difference: 1,2,3,4,5,6,
```

```
adjacent difference 1: : 1,1,1,1,1,1,
adjacent difference 2: : 1,3,5,7,9,11,
adjacent difference 2: : 1,2,6,12,20,30,
```

18.4 SORTING ALGORITHMS

STL has a number of algorithms to sort the elements of a range:

- Sort all elements based on the operator <.
- Sort all elements based on a *binary predicate*.
- Partial sort of all elements based on the operator <.
- Partial sort of all elements based on a binary predicate.
- Heap algorithms.

We concentrate on the main algorithms in this category. For more information, see Josuttis (2012).

18.4.1 Full Sort

There are four algorithms in this subcategory. The default comparator is the operator < while it is possible to provide user-defined comparison functions. The algorithm `std::sort()` (usually based on *quicksort*) guarantees a good performance and it has $O(n \log n)$ complexity in general. We should use `std::stable_sort()` if worst-case performance is important. We take an example to sort a range in two different ways:

```
void FullSort()
{
    std::cout << "\n**** Block I, Full sort ****\n";
    std::vector<double> v = { 1.0, -1.0, 2.0, -2.0, 3.0, -3.0 };
    print(v, "input vector: ");

    // Sort using default <
    std::sort(std::begin(v), std::end(v));
    print(v, "sorted vector using operator <: ");

    auto DESCENDING = [](int d1, int d2) { return (d1 > d2); };
    std::sort(std::begin(v), std::end(v), DESCENDING);
    print(v, "sorted vector using lambda op: ");

    // Stable sort: avoids worst-case performance
    std::stable_sort(std::begin(v), std::end(v));
    print(v, "stable sorted vector using operator <: ");

    std::sort(std::begin(v), std::end(v), DESCENDING);
    print(v, "stable sorted vector using lambda op: ");

    auto ASCENDING = [](int d1, int d2) { return (d1 < d2); };
    std::sort(std::begin(v), std::end(v), ASCENDING);
    print(v, "stable sorted vector using lambda op: ");
}
```

The output is:

```
**** Block I, Full sort ****
input vector:  1, -1, 2, -2, 3, -3,
sorted vector using operator <:  -3, -2, -1, 1, 2, 3,
sorted vector using lambda op:  3, 2, 1, -1, -2, -3,
stable sorted vector using operator <:  -3, -2, -1, 1, 2, 3,
stable sorted vector using lambda op:  3, 2, 1, -1, -2, -3,
stable sorted vector using lambda op:  -3, -2, -1, 1, 2, 3,
```

18.4.2 Partial Sort

There are four algorithms in this category: first, we can choose between the option to sort a range based on the operator < or by providing a user-defined comparison function and second, we can sort a range *in-place* as it were or we can create a new sorted array from an input range.

In general, partial sorting entails sorting a range until the first *n* elements have been sorted.

A code sample to show how to use the five algorithms is as follows:

```
void PartialSort()
{
    std::cout << "\n**** Block II, Partial sort ****\n";
    std::vector<double> v
                    = { 1.0,-1.0,2.0,-2.0,3.0,-3.0,100.0,-100.0,200.0,
                        -200.0 };
    print(v, "input vector: ");

    // Sort using default <
    auto pos = std::begin(v) + 5;
    std::partial_sort(std::begin(v), pos, std::end(v));
    print(v, "sorted vector using operator <: ");

    auto DESCENDING = [](int d1, int d2) { return (d1 > d2); };
    std::partial_sort(std::begin(v), pos, std::end(v), DESCENDING);
    print(v, "sorted vector using lambda op: ");

    // Copy from source range sorted into the destination range
    std::deque<double> dq(v.size());
    std::partial_sort_copy(std::begin(v), std::end(v),
                           std::begin(dq), std::end(dq));
    print(dq, "sorted copied vector using operator <: ");

    std::partial_sort_copy(std::begin(v), std::end(v),
                           std::begin(dq), std::end(dq), DESCENDING);
    print(dq, "sorted copied vector using lambda op: ");

}
```

The output is:

```
**** Block II, Partial sort ****
input vector:  1, -1, 2, -2, 3, -3, 100, -100, 200, -200,
```

```
sorted vector using operator <:            -200, -100, -3, -2, -1, 3, 100,
2, 200, 1,
sorted vector using lambda op:            200, 100, 3, 2, 1, -200, -100,
-3, -2, -1,
sorted copied vector using operator <:    -200, -100, -3, -2, -1,
 1, 2, 3, 100, 200,
sorted copied vector using lambda op:     200, 100, 3, 2, 1, -1, -2,
-3, -100, -200,
```

18.4.3 Heap Sort

A *max heap* is a special kind of *binary tree* with the properties:

- The value of each node is not less than the values stored in each of its children.
- The tree is perfectly balanced and the leaves in the last level are all in the leftmost positions.

It is also possible to define a *min heap*. Heaps and heap algorithms are supported in both Boost and C++. A special case is the *priority queue* in STL.

Sample code is:

```
void Heap()
{
    std::cout << "\n**** Block III, heap sort etc. ****\n";
    std::vector<int> v4 = { 3, 4, 5, 6, 7, 5, 6, 7, 8, 9, 1, 2, 3, 4 };
    print(v4, "input vector");

    // Construct a max heap using < as comparison operator
    // as well as a user-defined one
    std::make_heap(v4.begin(), v4.end());
    print(v4, "heap sorted vector");

    // Descending
    std::vector<int> v2 = { 3, 4, 5, 6, 7, 5, 6, 7, 8, 9, 1, 2, 3, 4 };
    print(v2, "2nd input vector");
    auto comp = [](int n, int m)->bool { return n > m; };
    std::make_heap(v2.begin(), v2.end(), comp);
    print(v2, "2nd heap sorted input vector");

    std::pop_heap(v4.begin(), v4.end());  // Largest to end and vice versa
    print(v4, "popped");
    v4.pop_back();                        // No largest element in the heap
    print(v4, "popped vector");

    v4.push_back(17);
    std::push_heap(v4.begin(), v4.end());
    print(v4, "Value 17 pushed");
```

```
    std::sort_heap(v4.begin(), v4.end());
    print(v4, "Sorted heap");
}
```

The output is:

```
**** Block III, heap sort etc. ****
input vector 3, 4, 5, 6, 7, 5, 6, 7, 8, 9, 1, 2, 3, 4,
heap sorted vector 9, 8, 6, 7, 7, 5, 5, 3, 6, 4, 1, 2, 3, 4,
2nd input vector 3, 4, 5, 6, 7, 5, 6, 7, 8, 9, 1, 2, 3, 4,
2nd heap sorted input vector 1, 3, 2, 6, 4, 3, 4, 7, 8, 9, 7,5,5, 6,
popped 8, 7, 6, 7, 4, 5, 5, 3, 6, 4, 1, 2, 3, 9,
popped vector 8, 7, 6, 7, 4, 5, 5, 3, 6, 4, 1, 2, 3,
Value 17 pushed 17, 7, 8, 7, 4, 5, 6, 3, 6, 4, 1, 2, 3, 5,
Sorted heap 1, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 17,
```

18.5 SORTED-RANGE ALGORITHMS

The algorithms in this category operate on ranges that have already been sorted according to their sorting criterion. These algorithms have significantly better performance than similar algorithms for *unsorted ranges* that we discussed in Chapter 17.

The algorithms address the following functionality:

- Does a sorted range contain a given value?
- Does a sorted range contain all elements of another sorted range?
- Give the position of the first element in a sorted range with value equal to or greater than a given value.
- Give the position of the first element in a sorted range with value greater than a given value.
- The range of elements that is equal to a given value.
- Merge sorted ranges.
- Set algorithms (union, intersection, difference, symmetric difference).
- Merge consecutive sorted ranges.

18.5.1 Binary Search

Binary search in an ordered range is a *divide-and-conquer* approach. Instead of sequentially searching in a range to determine if a given value is in that range (linear $O(n)$ complexity) we decompose a problem into two or more smaller subproblems. Complexity in this case is logarithmic.

An example of use is:

```
void BinarySearch()
{
    std::cout << "\n** Block I, Binary search\n";
    // I. Binary search
    // Binary search: check ifone element is present. Range must be sorted!
```

```
    std::list<int> myList = { 3,5,6,-1,234,100,32,99,0,-2 };
    print(myList, "input list, not sorted ");
    int val = 100;
    if (std::binary_search(myList.begin(), myList.end(), val))
    {
       std::cout << "Value " << val << " found in list\n";
    }
    else
    {
       std::cout << "Value " << val << " NOT found in list\n";
    }

    // Binary search with defined sorting algo
    myList.sort(DESCENDING); // Algo will not work if list is not sorted
    print(myList, "list, DESC");

    val = 100;
    if (std::binary_search(myList.begin(), myList.end(), val, DESCENDING))
    {
       std::cout << "Value " << val << " found in list\n";
    }
    else
    {
       std::cout << "Value " << val << " NOT found in l\n";
    }
}
```

The output in this case is:

```
** Block I, Binary search
input list, not sorted  3, 5, 6, -1, 234, 100, 32, 99, 0, -2,
Value 100 NOT found in list
list, DESC 234, 100, 99, 32, 6, 5, 3, 0, -1, -2,
Value 100 found in list
```

The container must be sorted for the binary search algorithm to work.

18.5.2 Inclusion

The two algorithms in this category determine whether one range contains all the elements of another range. The complexity is at most $2 * (N + M)$, where N is the number of elements to be searched in and M is the number of elements in the range whose occurrence in the first range of size N we are trying to determine:

```
void Includes()
{
    std::cout << "\n** Block II, Range includes\n";
    std::list<int> range = { 1,2,3,4,5 };
    std::list<int> searchRange = { 2,3,4 };
```

```
    std::cout << ": found? " << std::boolalpha
              << std::includes(range.begin(), range.end(),
                    searchRange.begin(), searchRange.end())<< '\n';

    auto CmpNoCase = [](char a, char b)
             {return std::tolower(a) < std::tolower(b);};

    std::vector<char> v1 = { 'a', 'b', 'c', 'f', 'h', 'x' };

    std::vector<char> v2 = { 'A', 'B', 'C' };
    std::cout << ": (case-insensitive) " << std::boolalpha
              << std::includes(v1.begin(), v1.end(), v2.begin(), v2.end(),
                    CmpNoCase) << '\n';
}
```

The output is:

```
** Block II, Range includes
: found? true
: (case-insensitive) true
```

18.5.3 First and Last Positions

The four algorithms in this category are similar to the non-modifying algorithms `std::find` and `std::find_if` that we already discussed in Chapter 17 (Section 17.3.3) but in the current case complexity is logarithmic for random-access iterators and linear otherwise. The complexity of the algorithms in Section 17.3.3 is always linear.

The main functionality is:

- Find the first element that has a value not less than a given target value (`lower_bound`).
- Return the position of the first element that has a value greater than the target value (`upper_bound`).

An example of use is:

```
void Bounds()
{
    std::cout << "\n** Block III, Lower and upper bounds\n";
    // Searching for first and last possible positions.
    // We do not take the option of using a binary predicate.
    std::list<int> myListA
          = { 3, 5, 6, -1, 234, 100, 100, 100, 32, 99, 0, -2 };

    // Default sort is ascending.
    myListA.sort();
    print(myListA, "list, ASC");

    int searchValue = 100;
    std::list<int>::iterator posFirst, posLast;
```

```
    posFirst = std::lower_bound(myListA.begin(), myListA.end(),
               searchValue);
    posLast = std::upper_bound(myListA.begin(), myListA.end(),
              searchValue);

    std::cout << "Value " << searchValue << " position "
              << distance(myListA.begin(), posFirst) + 1
              << " up to " << distance(myListA.begin(), posLast) + 1
              << " contiguously" << std::endl;
}
```

The output is:

```
** Block III, Lower and upper bounds
list, ASC -2, -1, 0, 3, 5, 6, 32, 99, 100, 100, 100, 234,
Value 100 position 9 up to 12 contiguously
```

In order to obtain the result from both the lower bound and the upper bound algorithms and return them as a pair, we use the `std::equal_range` algorithm that we now discuss.

18.5.4 First and Last Possible Positions as a Pair

Both algorithms in this category return the range of elements whose value is equal to a target value. The range contains the first position and the last position of an element whose value could be inserted without breaking the sorting order. In other words, the range is the *closed–open interval* `[begin, end)`.

```
void EqualRanges()
{
    std::cout << "\n** Block IV, Equal ranges\n";
    std::list<int> myListA = { 3,5,6,-1,234,100,100,100,32,6,99,0,
                               -2,6,6 };
    int searchValue = 6;
    print(myListA, "list, unsorted");
    myListA.sort();
    print(myListA, "list, sorted");

    // Find the range of values that contain a given value.
    typedef std::list<int>::iterator it;
    std::pair<it, it> range
          = equal_range(myListA.begin(),
                myListA. end(), searchValue);

    std::cout << "Value " << searchValue
              << " lies in OFFSET closed-open range ["
              << distance(myListA.begin(), range.first)
              << "," << distance(myListA.begin(), range.second)
              << ")" << std::endl;
}
```

The output is:

```
** Block IV, Equal ranges
list, unsorted 3,5,6,-1,234,100,100,100,32,6,99,0,-2,6,6,
list, sorted -2,-1, 0, 3, 5, 6, 6, 6, 6, 32, 99, 100, 100, 100, 234,
Value 6 lies in OFFSET closed-open range [5,9)
```

18.5.5 Merging

The algorithms in this category merge two ranges in a number of ways:

- Merge two source ranges into a destination range.
- Set operations on set-like containers: union, intersection, difference, symmetric difference ('Venn diagram stuff').

An example of using the first algorithm is:

```
void Merge()
{
    std::cout << "\n** Block V, Merge\n";
    std::list<int> myListA = { 3,5,6,1,2,3 };
    std::list<int> myListB = { -3,-5,-6,1,2,3 };

    myListA.sort(); myListB.sort();
    print(myListA, "List A");
    print(myListB, "List B");
    std::list<int> mergedList;
    std::merge(myListA.begin(), myListA.end(), myListB.begin(),
               myListB.end(),
    std::front_insert_iterator<std::list<int>>(mergedList));
    print(mergedList, "Merged list front insert");

    std::list<int> mergedListII;
    std::merge(myListA.begin(), myListA.end(), myListB.begin(), myListB.end(),
    std::back_insert_iterator<std::list<int>>(mergedListII));
    print(mergedListII, "Merged list back insert");
}
```

The output is:

```
** Block V, Merge
List A 1, 2, 3, 3, 5, 6,
List B -6, -5, -3, 1, 2, 3,
Merged list front insert 6, 5, 3, 3, 3, 2, 2, 1, 1, -3, -5, -6,
Merged list back insert -6, -5, -3, 1, 1, 2, 2, 3, 3, 3, 5, 6,
```

We give a final example to show how to compute the union and intersection of two sets:

```
void SetUnion()
{ // Union etc. of sets
```

```
    std::cout << "\n** Block VI, Union\n";
    std::set<int> v1 = { 1, 2, 3, 4, 5 };
    std::set<int> v2 = { 3, 4, 5, 6, 7 };
    std::vector<int> dest;

    std::set_union(std::begin(v1), std::end(v1),
                   std::begin(v2), std::end(v2),
                   std::back_inserter(dest));

    for (auto e : dest) { std::cout << e << ", "; }

    dest.clear();
    std::set_intersection(std::begin(v1), std::end(v1),
                          std::begin(v2), std::end(v2),
                          std::back_inserter(dest));

    std::cout << '\n';
    for (auto e : dest) { std::cout << e << ", "; }
}
```

The output is:

```
** Block VI, Union
1, 2, 3, 4, 5, 6, 7,
3, 4, 5,
```

Finally, we recall that the algorithms in Exercise 2 of Chapter 17 can be used to test whether a range is sorted.

18.6 AUXILIARY ITERATOR FUNCTIONS

For completeness, we discuss a number of auxiliary functions that deal with iterators and that are useful when using and extending STL algorithms. In particular, the `distance()` algorithm is important and useful.

The algorithms solve the following problems:

- Advance the position of an iterator by a positive or negative offset.
- Helper functions to move to the next or previous iterator positions.
- Find the difference between two iterators.
- Swap the values to which two iterators refer.

18.6.1 `advance()`

This algorithm increments the position of an iterator that is passed as an argument. The iterator is allowed to step forward or backward:

```
void Advance()
{
```

```
        using List = std::list<int>;
        List myList{ 1, 2, 3, 4, 5, 6 };

        // advance()
        auto pos = myList.begin();  // 1
        std::cout << *pos << '\n';

        std::advance(pos, 3);       // 4
        std::cout << *pos << '\n';
        std::advance(pos, -1);      // 3
        std::cout << *pos << '\n';

        // In general, advancing outside container
        // results in undefined behaviour
        pos = myList.begin();
        std::advance(pos, 8);       // I get '2'
        std::cout << *pos << '\n';
}
```

The output is:

```
**** Block I, advance() ****
1,4,3,2,
```

The algorithm does not check whether it crosses the end of a sequence; in such cases the result leads to *undefined behaviour*. Complexity is constant for random-access iterators and it is linear for other iterators.

18.6.2 `next()` and `prev()`

These helper functions are used to move to subsequent or previous iterator positions. They can be used to check the values of next or previous elements when iterating in a range. They can also be used to simplify the pre- and post-increment operators associated with iterators:

```
void NextPrev()
{
    // prev() left as an exercise!

    std::cout << "\n**** Block II, next() and prev() ****\n";
    using List = std::list<int>;
    List myList{ 1, 2, 3, 4, 5, 6 };

    // advance()
    auto pos = myList.begin();    // 1
    std::cout << *pos << ",";
    while (pos != std::end(myList) && std::next(pos) != std::end(myList))
    {
       ++pos;
       std::cout << *pos << ",";
    }
}
```

The output is:

```
**** Block II, next() and prev() ****
1,2,3,4,5,6,
```

As with `advance()`, `next()` does not check whether it has crossed the end of a sequence nor does `prev()` check whether it has crossed the beginning of a sequence.

18.6.3 distance()

This function computes the difference between two iterators. For random-access iterators the value is the difference of the iterators and it has constant complexity. For other iterators the complexity is linear. For this reason it is advisable not to use this function with non-random-access iterators:

```
void Distance()
{
       std::cout << "\n**** Block III, distance() ****\n";

       using List = std::list<int>;
       List myList{ 1, 2, 3, 4, 5, 6 };

       auto pos2 = std::find(std::begin(myList), std::end(myList), 3);
       std::cout << std::distance(std::begin(myList), pos2)<<'\n';    // 2

       auto pos3 = std::find(pos2, std::end(myList), 6);
       if (pos3 != std::end(myList))
       {
          std::cout << std::distance(pos2, pos3) << '\n';             // 3
       }

       auto pos4 = std::find(std::begin(myList), std::end(myList), 6);
       std::cout << std::distance(std::begin(myList), pos4) << '\n';// 5

       auto pos5 = std::find(std::begin(myList), std::end(myList), 1);
       std::cout << std::distance(std::begin(myList), pos5)<<'\n';    // 0
}
```

The output is:

```
**** Block III, distance() ****
2
3
5
0
```

18.6.4 iter_swap()

This function swaps the values to which two iterators refer:

```
void IterSwap()
{
    std::cout << "\n**** Block IV, iter_swap() ****\n";
    // 101 examples
    int* j = new int; int* i = new int;
    *j = 1; *i = 2;
    std::cout << "Before swap: " << *i << ", " << *j << '\n';
    std::iter_swap(j, i);
    std::cout << "After swap: " << *i << ", " << *j << '\n';
    delete j; delete i;

    // Part 2
    std::vector<int> v1 = { 0,1 };
    std::vector<bool> v2 = { true, false };

    print(v1, "v1");
    print(v2, "v2");
    std::iter_swap(std::begin(v1), std::begin(v1) + 1);
    std::iter_swap(std::begin(v2), std::begin(v2) + 1);
    print(v1, "v1");
    print(v2, "v2");
}
```

The output is:

```
**** Block IV, iter_swap() ****
Before swap: 2, 1
After swap: 1, 2

v1 0, 1,
v2 true, false,

v1 1, 0,
false, true,
```

18.7 NEEDLE IN A HAYSTACK: FINDING THE RIGHT STL ALGORITHM

We have now completed our discussion of STL algorithms. In this section we provide some rules to help find the most suitable algorithm or combination of algorithms that realise a given requirement. First, it is important to know what the problem is and what the requirements are (for example, efficiency, interoperability with your current software frameworks and maintainability). Second, we determine which category (for example, the sorted range algorithms) the requirement belongs to and then we determine which specific algorithm in the category realises the requirement. Some general questions are:

- A1: What is the most suitable container?
- A2: What are the typical operations to be performed on the container?

- A3: To which categories do the operations in A2 belong? Non-modifying, modifying and sorted range algorithms might be good choices to start with.
- A4: Using predefined function objects, user-defined function objects, binders and lambda functions; the choice will probably depend on readability and maintainability requirements.
- A5: Do we need to create *adapters* and *wrappers* to allow the software to interoperate with STL?

These are just some of the questions that we can ask to determine if we can use STL algorithms in our applications.

18.8 APPLICATIONS TO COMPUTATIONAL FINANCE

Many STL algorithms can be used in the numerical applications that we discuss in this book. We can create function mechanisms, wrappers and adapters (as seen in Chapters 11 and 12, for example) that use these algorithms without our having to reinvent the wheel. To this end, we draw up a list of typical examples that can be realised by one or more algorithms:

- E1: Count the number of times a stock crosses a barrier.
- E2: Maximum and minimum values of a stock.
- E3: Locate the first occurrence of a value in a container.
- E4: Create discrete arrays based on a function object (as in the Excel driver code of Chapter 14).
- E5: Investigate performance gains when comparing `std::copy()` and `std::move()`.
- E6: Generate and reuse randomly generated data for use in Monte Carlo and other numerical schemes.
- E7: Search for data and index values in (sorted) mesh arrays.

These algorithms apply to one-dimensional containers. In many cases we need two-dimensional matrices. To this end, we note that the Boost C++ *uBLAS* matrix library is STL compatible. The means that the algorithms which apply to `std::vector` are directly applicable to Boost vectors, as defined by `boost::numeric::ublas::vector`.

Finally, we note that STL algorithms are not thread-safe in general. This means that non-deterministic behaviour occurs if multiple threads concurrently access container data. We discuss multithreading and parallel processing in Chapters 28 to 32.

18.9 ADVANTAGES OF STL ALGORITHMS

We have given a detailed and compact summary of the STL algorithms that C++ supports. In general, using STL algorithms leads to efficient, portable and standardised code. For these reasons we recommend their use rather than creating your own home-grown versions of the algorithms.

18.10 SUMMARY AND CONCLUSIONS

In this chapter we have introduced *mutating*, *sorting*, *sorted range* and *numeric algorithms*. We provided complete machine-readable and machine-runable code and corresponding output. We recommend that you do the exercises.

18.11 EXERCISES AND PROJECTS

1. (Choosing the Appropriate Algorithm Investigation)

Consider a general sequence container, for example `std:list` for convenience. This involves reading through Chapters 17 and 18 again. To which categories do the following algorithms belong and which specific algorithm would you use in each case? Determine what the output is in each case.

a) Scale all values by a given factor.
b) Count the number of elements whose values are in a given range.
c) Find the average, minimum and maximum values in a container.
d) Find the first element that is (is not) in a range.
e) Search for all occurrences of '3456' in the container.
f) Determine if the elements of two ranges are equal.
g) Determine if a set is some permutation of '12345'.
h) Is a container already sorted?
i) Copy a container into another container.
j) Move the last 10 elements of a container to the front of the container.
k) Swap two ranges at a given position.
l) Generate values in a container based on some formula.
m) Replace all uneven numbers by zero.
n) Remove all elements whose value is less than 100.
o) Shuffle a container randomly using both pre- and post-C++11 versions.
p) Compute one-sided divided differences of the values in a container.

Now take specific containers, for example `std:vector`, and implement each of the above operations. The challenge here lies in finding the most appropriate STL algorithm to use and then implementing it.

For each of the above questions a) to p) ask yourself the following questions:

- What is algorithm input? What is the output?
- Is the input modified on output?
- Which category does the algorithm belong to?
- Choose the most appropriate algorithm and implement it.
- Check and understand the output.
- Think about the process of producing output from input.
- Think about (other examples) where you can use the algorithm.

You will know that you have successfully completed the exercises when you have executed these steps. Understand the output. After having completed these exercises you will have gained a good understanding of STL algorithms for sure.

2. (Priority Queue)

In this exercise we create a *priority queue* that is supported in STL:

```
template<class T, class Container = std::vector<T>,
       class Compare = std::less<typename Container::value_type>>
class priority_queue;
```

In this case elements are read according to their priority.

Answer the following questions:

a) Create a default priority queue of integers with elements $\{10,5,20,30,25,7,40\}$. Print the queue.

b) Modify the queue in part a) using `std::greater<int>` as comparator function. Print the queue.

Create a priority queue using an underlying data container of your choice and the following lambda function as comparator:

```
auto cmp = [](int left, int right)->bool { return (left > right); };
```

3. Which of the following arrays is a *heap*?

a) 9 8 6 7 7 5 5 3 6 4 1 2 3 4

b) 1 2 3 3 4 4 5 5 6 6 7 7 8 9

c) 8 22 33 25 44 40 55 55 33

d) 1 2 −4 6

Hint: draw the corresponding *complete binary tree* representation of the heap.

4. Which of the following statements regarding heaps are true?

a) It is a binary tree that is full at each level (that is, it is full at the last level).

b) It is a complete binary tree with either the min or max.

c) It is a complete full binary tree.

d) It is an extended binary tree.

5. (STL Algorithms and Computational Finance)

Consider the requirements E1, E2, …, E7 in Section 18.8. Determine which STL algorithms to use in order to implement them.

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 19

An Introduction to Optimisation and the Solution of Nonlinear Equations

19.1 INTRODUCTION AND OBJECTIVES

In this chapter we introduce a number of algorithms to solve nonlinear equations and to find the minimum value of a function in one unknown variable. This is called *univariate root finding* and *minimisation*, respectively. We reduce the scope to functions of a single variable and to the computation of implied volatility. There are enough topics to consider even for this seemingly simple problem.

The goals are:

- Algorithms to solve nonlinear equations and to compute function minima. In this case we are interested in the mathematical and numerical foundations of the problem.
- Instantiate the software model that we introduced in Chapter 9 to create a generic system to solve nonlinear problems. This is a design step.
- Apply the newly developed code to compute *implied volatility*.
- Employ modern C++ multiparadigm syntax and language features to promote developer productivity.

We examine the twin topics of computing the *roots of functions* and *function minimisation* and subsequent implementation in C++. Some of the advantages are that we are employing a defined process to create a software framework to solve problems, as introduced in Chapter 9.

Some of the benefits that this chapter delivers are:

- It discusses a number of robust nonlinear solvers for univariate functions based on C++11 and the Boost C++ libraries. We assemble these solvers in one place for easy reference.
- We motivate how and why the algorithms work. Furthermore, these algorithms work with a variety of function types in C++ in combination with input arguments and associated parameters.
- We implement the code using the design blueprints in Chapter 9. It is a small reusable software framework and the design can be generalised to other applications.
- We also discuss the important topic of how to compute implied volatility.

More detailed and technical information can be found in Appendix 2. This chapter concentrates on the mathematics and software design issues.

19.2 MATHEMATICAL AND NUMERICAL BACKGROUND

Consider the *scalar function* $f:\mathbb{R}^n \to \mathbb{R}^1,\ n \geq 1$ that maps n-dimensional Euclidean space to the real line. In general, we wish to find a point x in n-dimensional Euclidean space that minimises or maximises this function. The general *optimisation problem* is:

$$\text{optimise } z = f(x), \quad x = (x_1, \ldots, x_n)^\top. \tag{19.1}$$

Here, f is called the *objective function* and $x^\top$ is the transpose of the vector x. The problem is *unconstrained* because there are no restrictions placed on the whereabouts of the solution x. We note that maximisation of $f(x)$ is the same as the minimisation of its additive inverse $-f(x)$. In the main, we are usually interested in *minimisation problems*.

In general, there are no constraints on the point x that solves the optimisation problem (19.1). The objective function has a *local* (or *relative*) *minimum* at x_0 if there exists an interval (or more generally, a *neighbourhood*) around x_0 such that $f(x) \geq f(x_0)$ for all x for which the function is defined. If $f(x) \geq f(x_0)$ for all x for which the function is defined then x_0 is called a *global* (*absolute*) minimum. In general, the optimisation problem (19.1) may have zero, one or many solutions.

We can extend problem (19.1) by discussing multivariable optimisation problems with constraints. There are many kinds of constraints, for example:

- Equality constraints:

$$g(x) = 0. \tag{19.2}$$

- Inequality constraints:

$$g(x) \geq 0. \tag{19.3}$$

- Linear constraints:

$$g(x) = Ax^\top - b(g(x) = (g_1(x), \ldots, g_m(x))^\top \tag{19.4}$$

where A is a constant $n \times m$ matrix, b is an m-dimensional vector, x is an n-dimensional vector, g is a given function of its arguments and the symbol $\top$denotes the transpose of a vector or of a matrix.

A special case is a *quadratic program* in which each constraint is linear and the objective function has the form:

$$f(x_1, \ldots, x_n) = \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_i x_j + \sum_{j=1}^{n} d_j x_j \tag{19.5}$$

where c_{ij} and d_j are known constants, $1 \leq i, j \leq n$.

We now reduce the scope of the problem in this chapter to *univariate unconstrained nonlinear problems* of the form:

$$\text{optimise } z = f(x),\ x \in (-\infty, \infty). \tag{19.6}$$

In some cases we may be interested in finding a local or global minimum in an interval, thus leading to a one-variable *constrained problem*:

$$\text{optimise } z = f(x) \text{ subject to } a \leq x \leq b. \tag{19.7}$$

Closely related to optimisation is the problem of finding the zeros (or roots) of a nonlinear function:

$$f(x) = 0,\ x \in (-\infty, \infty). \tag{19.8}$$

In general, it is not possible to find an analytical solution to this problem and we must employ numerical methods, for example the *bisection* or *Newton–Raphson* method.

It is possible to find the solution of a system of n nonlinear equations:

$$f(x) = 0,\ x = (x_1, \ldots, x_n)^\top \in \mathbb{R}^n \tag{19.9}$$

where:

$$f(x) = \left(f_1(x), \ldots, f_n(x)\right)^\top,\ f : \mathbb{R}^n \to \mathbb{R}^n$$
$$F(x) = \sum_{j=1}^{n} f_j^2(x) = f(x)^\top f(x)$$

by defining the following *objective function*:

$$f\left(x^*\right) = 0 \Leftrightarrow x^* \text{ is a minimun of } F(x). \tag{19.10}$$

The desired solution is $F(x^*) = 0$ and this is true only when x^* satisfies (19.9). The function $F(x)$ may have any number of local minima with $F(x^*) > 0$ which have no relevance to the current problem except as nuisances. Hence if there is any solution at all it is the global minimum that we require. This is advantageous in the one-dimensional case for unimodal functions. A *unimodal function* $f(x)$ defined in an interval is one that has one and only one point at which $f(x)$ has a local minimum or local maximum.

19.3 SEQUENTIAL SEARCH METHODS

In some cases it may be possible to find a function minimum by using school calculus, but in general we need to employ numerical methods. In this section we discuss two methods for unimodal functions in an interval:

- The three-point interval search.
- Golden-mean search.

These methods can be seen as *divide-and-conquer* methods because we bracket the interval into subintervals and we discard those subintervals in which we know that the minimum value does not lie. The three-point interval search method divides the interval into quarters and the objective function is evaluated at the *three equally spaced interior* points. The interior point yielding the best value of the objective function is determined. The new subinterval is now centred at this point and made up of two-quarters of the current interval. The *golden-mean search method* is applicable to a strictly unimodal function by successively narrowing the range of values inside which the extremum is known to exist. In this case the algorithm maintains the function values for triples of points whose distances form a *golden ratio*. A readable discussion of these methods can be found in Bronson and Naadimuthu (1997).

19.4 SOLUTIONS OF NONLINEAR EQUATIONS

The theory of nonlinear equations is well established. In this section we discuss the problem of finding real values x that satisfy the equation:

$$f(x) = 0$$

where f is a real-valued function. The methods to be discussed are:

- Bisection method.
- Newton's method.
- Secant method.
- Steffensen iteration.

The *bisection method* assumes that the function $f(x)$ has a zero in the interval (a, b) and we assume that the signs are opposite at the end points, that is $f(a)f(b) < 0$. For convenience, let us assume that $f(a) < 0$. The method divides the interval into equal parts until we arrive at an interval that is so small that it contains the zero of the function and is small enough to satisfy the given tolerance. The basic algorithm is defined using a sequence of intervals of ever-diminishing size:

$$(a, b) \supset (a_1, b_1) \supset (a_2, b_2) \supset (a_3, b_3) \supset \ldots$$

where:

$$(a_k, b_k) = \begin{cases} (m_k, b_{k-1}) & \text{if } f(m_k) < 0 \\ (a_{k-1}, m_k) & \text{if } f(m_k) > 0 \end{cases} \tag{19.11}$$

and

$$m_k = \tfrac{1}{2}(a_{k-1} + b_{k-1}).$$

After n steps the root is in an interval having a length given by:

$$b_n - a_n = 2^{-1}(b_{n-1} - a_{n-1}) = 2^{-n}(b - a) \tag{19.12}$$

$$m_{n+1} \text{ (is the root).} \tag{19.13}$$

Thus, the deviation from the exact root α is given by:

$$\alpha = m_{n+1} \pm d_n,\ d_n = 2^{-n-1}(b-a). \tag{19.14}$$

In general, we are interested in locating the zero of the function to within a given *tolerance* TOL. This means that we calculate the number of subdivisions of the original interval (a, b). To this end, some arithmetic based on equation (19.14) gives us the following estimate:

$$n > \frac{\log\left(\frac{b-a}{\text{TOL}}\right)}{\log 2} - 1. \tag{19.15}$$

The advantage of the bisection method is that we can always define an interval of arbitrary size in which the zero is located. The disadvantage is that convergence is slow. In fact, at each step we gain one binary digit in accuracy. We note also that the rate of convergence is independent of the given function $f(x)$. The method may be used to give us good initial approximations to second-order nonlinear solvers.

Newton–Raphson is probably one of the most famous iterative schemes in numerical analysis. The main advantage of the Newton–Raphson method is that it converges quickly. We say that its order of convergence is two, by which we mean that the error at each iteration decreases *quadratically*:

$$x_{n+1} = x_n + h_n,\ h_n = -\frac{f(x_n)}{f'(x_n)}, \quad n = 0, 1, 2, \ldots \tag{19.16}$$

The disadvantage is that the choice of the initial approximation is vitally important. If it is not chosen carefully the method may not converge at all or it may even converge to the wrong solution. A possible solution is to first use the bisection method to bracket the solution to a given accuracy (for example, 10^{-1}) and then apply the Newton–Raphson method. Furthermore, we must have an analytical expression for the derivative of f and this may not always be available.

The *secant method* can be derived from the Newton–Raphson method by approximating the derivative by divided differences. The resulting iterative scheme now becomes:

$$x_{n+1} = x_n + h_n,\ h_n = -f_n \frac{x_n - x_{n-1}}{f_n - f_{n-1}},\ f_n \neq f_{n-1} \quad \text{where} \quad f_n = f(x_n). \tag{19.17}$$

We note that the secant method needs two initial approximations in contrast to Newton–Raphson which only needs one initial approximation. In general, we must be careful when programming the secant method when the function values are close to each other or when the solutions at levels n and $n+1$ are close to each other; in such cases we calculate terms that are effectively $0/0$!

A disadvantage of the secant method is that it is only first-order accurate. In order to achieve second-order accuracy without having to evaluate derivatives we propose *Steffensen's method*, given by the scheme:

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

where:

$$g(x_n) = \frac{f(x_n + f(x_n)) - f(x_n)}{f(x_n)}. \tag{19.18}$$

This scheme requires two function evaluations but no computation of a derivative is needed.

With the exception of the bisection method, the choice of a good initial approximation is crucial and it must be 'close' to the exact solution, otherwise the iterative scheme may not converge. There are iterative methods based on the *continuation* (or *homotopy*) methods that converge to the true solution even when the initial guess is not good, and a discussion of these techniques is found in Appendix 2. We have already discussed homotopy theory in Chapter 6.

19.5 FIXED-POINT ITERATION

Let f be a mapping from some space X to a space Y. Let x_0 be some element of x and consider the iterative scheme defined by:

$$x_{n+1} = f(x_n),\ n = 0, 1, 2, \ldots \tag{19.19}$$

Under certain circumstances this sequence may converge to some point called a *fixed point*, that is:

$$x = f(x),\ \lim_{n\to\infty} x_n = x. \tag{19.20}$$

Many problems can be written in the form (19.20), thus allowing us to prove existence and uniqueness of solutions of linear and nonlinear problems in areas such as (Haaser and Sullivan, 1991):

a) Computing the solution of systems of linear algebraic equations (Hageman and Young, 1981). For example, solving the linear system $Au = b$ results in the iteration:

$$u_{n+1} = Gu_n + k,\ n = 0, 1, 2, \ldots \tag{19.21}$$

Here A and G are $N \times N$ matrices, b and k are known vectors and u is the vector to be solved for. Different choices for G lead to different schemes each with its own stability properties and rate of convergence, for example $G = I - A$ gives the Richardson method where I is the identity matrix of order N. Other *splittings* give rise to the Jacobi, Gauss–Seidel, SOR and SSOR methods.

b) Existence and uniqueness results for the solution of Fredholm and Volterra integral equations.

c) Existence and uniqueness results for the solution of systems of ordinary differential equations (Picard–Lindelöf theory). We discuss this topic in Chapter 24.

Some one-dimensional examples of fixed-point iterations that can be formed from equation (19.8) are:

- $x = \cos x$.
- $x^2 = 2,\ x = \frac{1}{2}\left(x + \frac{2}{x}\right)$.
- $x^3 + 2x^2 + 10x - 20 = 0,\ x = 20/(x^2 + 2x + 10)$ (Fibonacci).

Each of these mappings has a fixed point in some interval.

In the appendix to this chapter we discuss the *Banach fixed-point theorem* that generalises the fixed-point iteration to Banach spaces.

19.6 AITKEN'S ACCELERATION PROCESS

In general, scheme (19.19) has *linear convergence*. The sequence's rate of convergence can be improved by employing *Aitken's delta-squared process*. To this end, let:

$$\{x_n\},\ n = 0, 1, 2, \ldots \tag{19.22}$$

be a sequence and consider the new sequence defined by:

$$y_n = x_{n+2} - \frac{(x_{n+2} - x_{n+1})^2}{(x_{n+2} - x_{n+1}) - (x_{n+1} - x_n)},\ n = 0, 1, 2, \ldots \tag{19.23}$$

or

$$y_n = x_{n+2} - \frac{(\Delta x_{n+1})^2}{\Delta^2 x_n} \tag{19.24}$$

where:

$$\Delta x_n = (x_{n+1} - x_n),\ \Delta x_{n+1} = (x_{n+2} - x_{n+1})$$
$$\Delta^2 x_n = x_{n+2} - 2x_{n+1} + x_n = \Delta x_{n+1} - \Delta x_n.$$

We give more code details in Appendix 2.

19.7 SOFTWARE FRAMEWORK

We now map the mathematical and numerical entities that we have just discussed to software components. For each method we need to know when to use it and the amount of effort needed in order to achieve a given accuracy. We also need to state what happens when the method fails to converge, for example do we give the best estimate to date or do we throw an exception? Having addressed these issues we then integrate the numerical algorithms into a software framework based on the system decomposition techniques that we introduced in Chapter 9. At this stage we design a generic software framework that is easy to create and to extend (by the supplier) on the one hand and that is easy to use by client code on the other hand. Each aspect of the problem is customisable and the design is shown in Figure 19.1. Each module has a single responsibility and the modules encompass the functionality needed to realise the

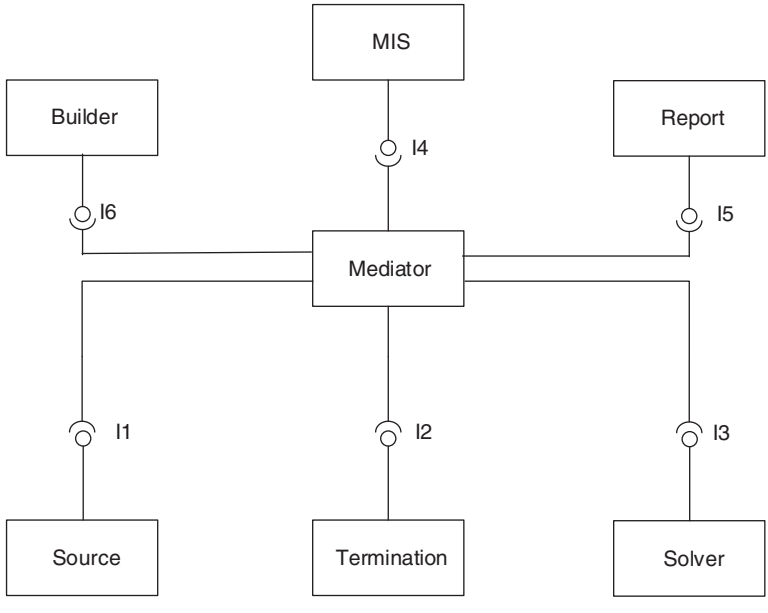


FIGURE 19.1 Context diagram for minimisation algorithms

requirements. It is important to understand the *data flow* in the system in which each module receives input and produces output.

The main steps in the *core process* of finding the minimum are:

- Choose the function to be minimised.
- Define the *termination criterion* for the algorithm.
- Choose a solver and compute the minimum.
- Present the ongoing and final results of the computations.

The design of the system in Figure 19.1 is based on the following interfaces (the rationale here is that we emulate each interface with one function using `std::function`):

```
// Interfaces.hpp
//
// The specifications for the components with interval search methods.
// These will be the interfaces between the mediator and the other
// components in the system context diagram.
//
// (C) Datasim Education BV 2017

#ifndef Interfaces_HPP
#define Interfaces_HPP

#include <functional>
#include <tuple>
```

```
// Interface I1. Scalar-valued functions of a single variable
template <typename T>
       using IFunctionType = std::function<T (T)>;

// 2. Termination I2; test if interval [a,b] is less than a TOL
template <typename T>
       using ITerminationType = std::function<bool (T a, T b)>;

// 3. Solver I3; compute new interval [a,b] and values
template <typename T>
       using ISolverType = std::function<void (T& a,T& b,T& fa,T& fb)>;

// 4. Progress MIS component i4; send current interval [a,b] and the
// function values at a and b.
template <typename T>
       using IMISType = std::function<void (T a, T b, T fa, T fb)>;

// 5. Core data to client I5, i.e. comprising a, b, fa, fb. A function
// that returns a tuple
template <typename T>
       using IReportType = std::function<void(T a, T b, T fa, T fb)>;;

#endif
```

When using this software framework we can instantiate these interfaces using function objects, lambda functions, free functions or binded member functions. We have created classes for the following algorithms: *three-point interval search*, *golden-mean search*, *fixed-point iteration* and *Aitken*. We have also created an *adapter class* that delegates to the *Brent solver* in the *Boost C++ Math Toolkit* library. For example, the code for the three-point interval search method is:

```
template <typename T>
    class ThreePointIntervalSolver
{
private:
    IFunctionType<T> f;     // Function to be minimised

    // Extra vars
    T h, a1, a2, a3, fa1, fa2, fa3;
    T a, b;

public:
    ThreePointIntervalSolver(const IFunctionType<T>& function, T A, T B)
       : f(function)
    {
       // std::cout << "3-pt scheme\n";
       // Iniitalisation
       h = 0.25 * std::abs(B-A);
       a = (A < B) ? A : B;
       a1 = a + h;
```

```
        a2 = a1 + h;
        a3 = a2 + h;
        b = a3 + h;
        fa1 = f(a1);
        fa2 = f(a2);
        fa3 = f(a3);
    }

    void operator () (T& lower, T& upper, T& funcLower, T& funcUpper)
    {

        funcLower = std::numeric_limits<double>::max();
        funcUpper = std::numeric_limits<double>::max();

        if ((fa1 < fa2) && (fa1 < fa3))
        {
            b = a2; a2 = a1; funcUpper = fa2; fa2 = fa1;
        }
        else if ((fa2 < fa1) && (fa2 < fa3))
        {
            a = a1; funcLower = fa1; b = a3; funcUpper = fa3;
        }
        else if ((fa3 < fa1) && (fa3 < fa2))
        {
            a = a2; a2 = a3; funcLower = fa2; fa2 = fa3;
        }

        h *= 0.5;
        a1 = a + h;
        a3 = a2 + h;
        fa1 = f(a1);
        fa3 = f(a3);

        if (funcLower == std::numeric_limits<T>::max()) funcLower = f(a);
        if (funcUpper == std::numeric_limits<T>::max()) funcUpper = f(b);

        lower = a;
        upper = b;

    }

    T value() const
    {
        return 0.5*(a + b);
    }
};
```

In order to use a solver we need to instantiate the modules in Figure 19.1. You can choose between using predefined classes or providing your own code. For example, typical code to define the termination condition is:

```
template <typename T>
    class TerminatorI
{
private:
    T tol;   // Tolerance

public:
    TerminatorI() : tol(1.0e-6) {}
    TerminatorI(T tolerance) : tol(tolerance) {}

    bool operator ()(T a, T b)
    {
       T m = 0.5*std::abs(a + b);
       if (m <= 1.0)
       {
          if (std::abs(b - a) < tol)
          {
              return true;
          }
          return false;
       }

       if (std::abs(b - a) < tol*m)
       {
          return true;
       }
       return false;
    }
};
```

Typical reporting and MIS classes are:

```
template <typename T>
    class ReportingDefault
{
private:

public:
    ReportingDefault()  {}

    void operator () (T a, T b, T fa, T fb)
    {
       std::cout << "** Final report ** \n";
       std::cout << std::setprecision(16)
                 << "Bracket + avg Value:  ["<< a << "," <<b<< "]
                     "<<(a+b)/2 << '\n';
       std::cout << std::setprecision(16) << "Averaged function value "
                 << 0.5*(fa + fb) <<  "\n";
    }
};
```

```
template <typename T>
    class MISDefault
{
private:

public:
    MISDefault() {}

    void operator () (T a, T b, T fa, T fb)
    {
       std::cout << std::setprecision(12)
                 <<  "***Stats [A,B]  [" << a << "," << b << "]\n";
       std::cout << std::setprecision(12)
                 << "***Stats [fA,fB] [" << fa << "," << fb << "]\n";
    }

};
```

19.7.1 Using the Mediator to Reduce Coupling

The *Mediator* design pattern (GOF, 1995) is a powerful technique to encapsulate interaction between a set of objects and modules. Its use promotes loose coupling by ensuring that objects do not refer to each other explicitly. In the current problem we implement the (default) mediator class as follows:

```
template <typename T>
       class Mediator
{
private:

       T A, B, fA, fB;
       IFunctionType<T> f;
       ITerminationType<T> stop;
       ISolverType<T> sol;
       IMISType<T> mis;
       IReportType<T> rep;

public:                                                    // Interfaces
       Mediator(const IFunctionType<T> func, T a, T b,     // 1
                const ITerminationType<T>& stopping,       // 2
                const ISolverType<T>& solver,              // 3
                const IMISType<T>& statistics,             // 4
                const IReportType<T>& report)              // 5
          : f(func), stop(stopping), sol(solver), rep(report)
       {
          A = a; B = b; fA = fB = 0.0;
          mis = statistics;
       }

       void run()
```

```
        {
           do
           {
               mis(A, B, fA, fB);
               sol(A, B, fA, fB);
           } while (!stop(A, B));

           rep(A, B, fA, fB);
        }

        T value() const
        {
           return 0.5*(A + B);
        }
};
```

In general, the mediator implements a state machine or some kind of loop. It communicates with its *satellite modules* but in keeping with the SRP creation of these latter modules should be left to a *builder*. See Exercise 3.

In general, the *Mediator* pattern seems not to be popular or widely applied by software developers and this may be attributed to the fact that object-oriented code results in a network of tightly coupled objects. Furthermore, object-oriented technology tends to be used in a bottom-up manner. This means that there is no coordinating object in typical applications. This situation leads to an inflexible design and reduces the reusability of the individual modules.

Finally, the design that we discuss in this section is similar to the policy-based design approach that we introduced in Chapter 9. See also Exercise 4.

19.7.2 Examples of Use

In this section we discuss how to use the software framework. In general, there are no constraints or limits on the size of the participating modules. They could be:

- C1: Simple lambda functions and free functions.
- C2: Medium-sized function objects and classes.
- C3: Complex systems in their own right, possibly with their own context diagram.

We choose between these options depending on the requirements. Because we are using universal function wrappers we can employ a multiparadigm approach to design the modules using whatever software style is most convenient for us. We concentrate on choices C1 and C2 in this section. The simplest example (case C1) is when we use lambda functions:

```
void Test101()
{ // Using lambda functions to show how the optimiser works, x* = 5

    ITerminationType<double> stopping = [](double a, double b)
                                        { return std::abs(b-a) < 0.0001; };

    IMISType<double> mis = [](double a, double b, double fa, double fb)  };
```

```
    auto report = [](double a, double b, double fa, double fb)
    {
    std::cout << std::setprecision(16)
              << "\n Golden Mean  " << (a + b) / 2;
    };

    auto func = [](double x) { return x + 1.0 / x; };
    double A = 5.0; double B = 10.0;
    GoldenSectionIntervalSolver<double> solver(func, A, B);
    Mediator<double> med(func, A, B, stopping, solver, mis, report);
    med.run();

    ThreePointIntervalSolver<double> solver2(func, A, B);
    auto report2 = [](double a, double b, double fa, double fb)
    {std::cout << std::setprecision(16) << "\n Three point  "
               << (a + b) / 2 << '\n'; };
    Mediator<double> med2(func, A, B, stopping, solver2, mis, report2);
    med2.run();
}
```

The next two cases could be considered as examples of using free functions:

```
double func(double x)
{
    return (std::log(x) - 5.0)* (std::log(x) - 5.0);
}

void TestThreeSchemes(const IFunctionType<double>& func, double A, double B)
{
    std::cout << "\nstart\n";
    ThreePointIntervalSolver<double> solver1(func, A, B);
    GoldenSectionIntervalSolver<double> solver2(func, A, B);
    BrentIntervalSolver<double> solver3(func, A, B);

    // Define common satellite systems
    TerminatorI<double> stopping(1.0e-6);
    IMISType<double> mis = [](double a,double b, double fa, double fb) { };
    ReportingDefault<double> report;

    // Build mediators
    Mediator<double> med1(func, A, B, stopping, solver1, mis, report);
    Mediator<double> med2(func, A, B, stopping, solver2, mis, report);

    ITerminationType<double> stopping3 = [](double a, double b)
    { return true; };

    Mediator<double> med3(func, A, B, stopping3, solver3, mis, report);

    // Run the algos
    med1.run();
    med2.run();
```

```
    med3.run();
    std::cout << "\nend\n";
}

// Functions suitable for fixed point
double func7(double x)
{ // g(x) = x - f(x)/m (g(x) == x <=> f(x) == 0)

    return  std::cos(x);
}

double func8(double x)
{ // x^2 = 2 => x = (x + 1/x)/2

    return  0.5*(x + 25.0 / x);
}

double func9(double x)
{ // Fibonacci x^3 + 2 x^2 + 10x - 20 = 0 ==> x = 1.368 808 107

    return 20.0 / (x*x + 2.0*x + 10.0);

}

void TestAitken()
{
    // Define common satellite systems
    TerminatorI<double> stopping(1.0e-7);
    IMISType<double> mis = [](double a, double b, double fa, double fb) {};
    ReportingDefault<double> report;

    // Test on functions 7, 8 and fibonacci

    std::cout << "\nstart Aitken\n";
    AitkenFixedPointSolver<double> solver1(func7, -1.0,1.5, 0.7, stopping);
    AitkenFixedPointSolver<double> solver2(func8, 1, 2, 1, stopping);
    AitkenFixedPointSolver<double> solver3(func9, 1,2, 1,stopping);

    // Build mediators
    Mediator<double> med1(func7, 0,2, stopping, solver1, mis, report);
    Mediator<double> med2(func8, 1,2, stopping, solver2, mis, report);

    Mediator<double> med3(func9, 1,2, stopping, solver3, mis, report);

    // Run the algos
    med1.run();
    med2.run();
    med3.run();
    std::cout << "\nend\n";
}
```

We recommend that you run the programs and examine the output.

19.8 IMPLIED VOLATILITY

The Black–Scholes model uses a variety of input parameters to derive a theoretical value for option price. The volatility is the only parameter that cannot be observed in the market. One way to estimate it is to assume that we know the market price of a call option. Then we can compute the corresponding volatility from the Black–Scholes model by solving a nonlinear equation using a numerical solver. An analytical solution does not exist in general as far as we know.

There is a plethora of results on how to compute implied volatility at varying levels of robustness, performance and accuracy:

- M1: Bisection and Newton–Raphson methods (Haug, 2007).
- M2: Approximations to the implied volatility (for example, Corrado and Miller, 1996; Grunspan, 2011). In this category we can include methods based on *asymptotic expansions*.
- M3: Mapping the nonlinear equation to a nonlinear least-squares form that we then solve using Brent's method or the golden-mean search method, for example.
- M4: More generally, evolutionary algorithms, stochastic optimisation and differential evolution (DE) (Price, Storn and Lampinen, 1996). In general, we have found DE to be the most robust algorithm to compute implied volatility. A discussion of DE is unfortunately outside the scope of this book. We discuss it in a forthcoming work.

Some of the issues and requirements that we take into consideration are:

- Is the method robust with respect to the band of option *moneyness* $K = S$, where K is the strike and S is the spot price?
- Small and large expirations.
- Small volatility.
- Possible round-off errors and the use of *multi-precision data types*. We wish to avoid *catastrophic cancellation*, which is the event that occurs when two numbers of almost equal size are subtracted leading to incorrect results.

Implied volatility is used as a more useful measure of an option's relative value than its price. Furthermore, implied volatility can be seen as a price. It is a more convenient way to communicate option prices than currency (Natenberg, 2007).

19.9 SOLVERS IN THE BOOST C++ LIBRARIES

The *Boost C++ Math Toolkit* contains several algorithms that are useful in the current context:

- The bisection method; TOMS 748 algorithm.
- The Newton–Raphson method (second order).
- The Halley and Schroeder methods (third order).
- Brent's method for function minimisation.

We discuss a number of iterative methods to compute implied volatility in Appendix 2.

19.10 SUMMARY AND CONCLUSIONS

In this chapter we designed and implemented a software framework for univariate optimisation problems based on the defined process in Chapter 9. The framework should be customisable and extendible and for this reason we have chosen the context diagram in Figure 19.1. In one sense this is the most far-reaching chapter because it addresses many of the topics that we have introduced in this book. We summarise them here:

- Constructive mathematics (Bishop, 1967), in particular the Banach fixed-point theorem.
- Numerical algorithms, in this case algorithms to find function minima.
- A PBD solution and standardised system interfaces.
- Application of C++ syntax and language features.
- Computing implied volatility.

We offload some of the more technical details and code to Appendix 2, including higher-order mathematical functions and multivariate optimisation.

The methods and techniques from this chapter can be applied to the construction of software libraries and frameworks in numerical analysis applications such as interpolation and numerical quadrature.

19.11 EXERCISES AND PROJECTS

1. (Convex and Concave Functions)

We now discuss convex and concave functions. A real function f defined on the interval $[a, b]$ where $-\infty < a < b < \infty$ is called *convex* if:

$$f((1-\lambda)x + \lambda y) \leq (1-\lambda)f(x) + \lambda f(y) \text{ for } x, y \in (a, b) \text{ and } 0 \leq \lambda \leq 1.$$

A function f is said to be *concave* if the function $F = -f$ is convex, in other words:

$$f((1-\lambda)x + \lambda y) \geq (1-\lambda)f(x) + \lambda f(y).$$

Any local minimum of a convex function is also a global minimum.

Answer the following questions:

a) Prove that the following functions are convex: $x^2, x^4, |x|^p$ for $1 \leq p$, $\sqrt{x}$, $\log x$ and e^x.

b) Let the function f be increasing in an interval (a, b). Prove that the function $A(x) = \int_a^x f(t)dt$, $\forall x \in (a, b)$ is convex.

c) Prove that if f and g are convex functions then so are $m(x) = \max\{f(x), g(x)\}$ and $h(x) = f(x) + g(x)$.

d) Approximate the *global* maximum of the function $f(x) = x^2 \sin x$ on $[0, \pi]$ using the three-point interval search, Brent and golden-mean methods. Compare the relative accuracy achieved by each of these methods.

2. (Ternary Search)

This technique finds the minimum or maximum of a unimodal function in an interval. It determines either that the minimum or maximum cannot be in the first third of the interval

or the last third of the interval. Having done that, we continue the search in the remaining middle third of the interval.

We describe the method as follows; let the interval be (a, b) and define the points:

$$m_1 = a + (b - a)/3$$
$$m_2 = b - (b - a)/3.$$

The objective is to find the minimum or maximum by modifying the interval (a, b) as the following pseudo-code shows:

$$\text{if } f(m_1) < f(m_2),\ b = m_2 \text{ else } a = m_1.$$

We keep iterating until the length of the interval is less than a prescribed size.

Answer the following questions:

a) Create the code to implement this algorithm based on the standard interfaces from Section 19.7. Test the code using the examples already presented.

b) Does the framework remain stable when this new module is added in the sense of our having to modify some of the modules, classes and interfaces in Figure 19.1?

c) Apply the new code to the function $f(x) = (x - 1)^2(x - 2)^2$ on the interval (0, 3). Does it work?

3. (Builder Pattern (GOF, 1995))

In Figure 19.1 we see the presence of a *Builder* module whose responsibility is to create the objects that the mediator communicates with. The reason for introducing this pattern is to localise object creation in dedicated factory classes and hence to avoid cluttering the file containing `main()` with many constructor calls, initialisation routines, if…else logic and so on.

Answer the following questions:

a) Examine a typical file containing `main()` and determine which code can be moved to a dedicated builder module.

b) Determine the builder's interface *I6* (it usually returns a tuple of newly created objects) and how it interacts with the mediator class in Figure 19.1.

c) Create a specific builder class and test the new code using examples with known output.

d) How far does the use of builders promote code maintainability and what are the circumstances under which it would be applied?

4. (*Policy-Based Design*, Medium-Sized Project)

In Section 19.7 we applied the design methodology from Chapter 9 to create a framework for univariate function optimisation. We designed the context diagram in Figure 19.1 for the case in which the interfaces (each one consisting of a single method) were defined using `std::function` (call it the Approach I solution). We can then customise these interfaces by instantiating them using free functions, lambda functions, static methods and object methods.

The goal of this exercise is to implement the design blueprint in Figure 19.1 using two other design approaches that we introduced in Chapter 9:

- Approach II: Create a templated mediator class whose template parameters are the other modules in Figure 19.1. We use private inheritance as discussed in the toy example in Section 9.3.4. This is a policy-based design approach.

- Approach III: Create a templated mediator class whose template parameters are the other modules in Figure 19.1. But in this case we are using *composition* (and not private inheritance) in the spirit of the C# example that we gave in Section 9.5.1, namely:

```
/*
interface IInput
{
    string message();
}

interface IOutput
{
    void print(string s);
}
*/

    class SUD<I,O>
          // where I : IInput
          // where O : IOutput
    {

       private I i_;
       private O o_;

    public SUD(I i, O o)
    {
       i_ = i;
       o_ = o;
    }

    public void run()
    {
          o_.print(i_.message());
    }
}
```

C++ does not support *concepts* (*constraints* in C#) at the moment of writing, which is the reason for the above commented code. Nonetheless, the code is the starting point for an equivalent implementation in C++.

Answer the following questions:

a) Implement Approach II for the design blueprints of Figure 19.1 and test the code using the examples already discussed in this chapter.

b) Implement Approach III for the design blueprints of Figure 19.1 and test the code using the examples already discussed in this chapter.

c) Compare the solutions and code based on the three approaches. Can you grade them based on run-time efficiency, interoperability and maintainability? What about readability?

19.12 APPENDIX: THE BANACH FIXED-POINT THEOREM

We work with sets and other mathematical structures in which it is possible to assign a *distance function* or *metric* between any two of their elements. Let us suppose that X is a set and let x, y and z be elements of X. Then a *metric* d on X is a non-negative real-valued function of two variables having the following properties:

$$D1 : d(x, y) \geq 0;\ d(x, y) = 0 \text{ if and only if } x = y$$

$$D2 : d(x, y) = d(y, x) \text{ (symmetry)}$$

$$D3 : d(x, y) \leq d(x, z) + d(z, y) \text{ where } x, y, z \in X \text{ (triangle inequality).}$$

The concept of distance is a generalisation of the concept of difference of two real numbers or the distance between two points in n-dimensional Euclidean space, for example.

Having defined a metric d on a set X we then say that the pair (X, d) is a *metric space*. We give some examples of metrics and metric spaces:

1. The set X of all continuous real-valued functions of one variable on the interval $[a, b]$ (we denote this space by $C[a, b]$) and we define the metric:

$$d(f, g) = \max |f(t) - g(t)|\ ,\ t \in [a, b].$$

   Then (X, d) is a metric space.
2. The n-dimensional Euclidean space, consisting of vectors of real or complex numbers of the form:

$$x = (x_1, \ldots, x_n),\ y = (y_1, \ldots, y_n)$$

   with metric $d(x, y) = \max\{|x_j - y_j|,\ j = 1, \ldots, n\}$ or using the notation for a norm $d(x, y) = \|x - y\|_\infty$.
3. Let $L^2[a, b]$ be the space of all square-integrable functions on the interval $[a, b]$:

$$\int_a^b |f(x)|^2 dx < \infty.$$

   We can then define the distance between two functions f and g in this space by the metric:

$$d(f, g) = \|f - g\|_2 \equiv \left\{ \int_a^b |f(x) - g(x)|^2 \right\}^{1/2}.$$

   This metric space is important in many branches of mathematics, including probability theory and stochastic calculus.
4. Let X be a non-empty set and let the metric d be defined by:

$$d(x, y) = \begin{cases} 0, & \text{if } x = y \\ 1, & \text{if } x \neq y. \end{cases}$$

Then $(X,\ d)$ is a metric space. You should verify that the metric d satisfies axioms D1, D2 and D3.

Many of the results and theorems in mathematics are valid for metric spaces and this fact means that the same results are valid for all specialisations of these spaces.

We define the concept of *convergence* of a sequence of elements of a metric space X to some element that may or may not be in X. We introduce some definitions that we state for the set of real numbers but they are valid for any *ordered field*, which is basically a set of numbers for which every non-zero element has a multiplicative inverse and there is a certain ordering between the numbers in the field.

Definition 19.1 A sequence $\{a_n\}$ of elements on the real line $\mathbb{R}$ is said to be *convergent* if there exists an element $a \in \mathbb{R}$ such that for each positive element ε in $\mathbb{R}$ there exists a positive integer n_0 such that:

$$|a_n - a| < \varepsilon \text{ whenever } n \geq n_0.$$

A simple example is to show that the sequence $\left\{\frac{1}{n}\right\},\ n \geq 1$ of rational numbers converges to 0. To this end, let ε be a positive real number. Then there exists a positive integer $n_0 > 1/\varepsilon$ such that $\left|\frac{1}{n} - 0\right| = \frac{1}{n} < \varepsilon$ whenever $n \geq \varepsilon^{-1}$.

We remark that Definition 19.1 also holds for any ordered field.

Definition 19.2 A sequence $\{a_n\}$ of elements of an ordered field F is called a *Cauchy sequence* if for each $\varepsilon > 0$ in F there exists a positive integer n_0 such that:

$$|a_n - a_m| < \varepsilon \text{ whenever } m, n \geq n_0.$$

In other words, the terms in a Cauchy sequence get close to each other while the terms of a convergent sequence get close to some fixed element. A convergent sequence is always a Cauchy sequence but a Cauchy sequence whose elements belong to a field X does not necessarily converge to an element in X. To give an example, let us suppose that X is the set of rational numbers; consider the sequence of integers defined by the Fibonacci recurrence relation:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2},\ n \geq 2. \end{aligned}$$

By solving this recurrence relationship it can be shown that:

$$F_n = \frac{1}{\sqrt{5}}[\alpha^n - \beta^n] \quad \text{where} \quad \alpha = \frac{1+\sqrt{5}}{2},\ \beta = \frac{1-\sqrt{5}}{2}.$$

Now define the sequence of rational numbers by:

$$x_n = F_n/F_{n-1},\ n \geq 1.$$

We can show that:

$$\lim_{n\to\infty} x_n = \alpha = \frac{1+\sqrt{5}}{2} \text{ (the Golden Ratio)}$$

and this limit is not a rational number. Incidentally, the Fibonacci numbers are useful in many kinds of applications such as optimisation (finding the minimum or maximum of a function) and random number generation.

We now define a *complete metric space X* as one in which every Cauchy sequence converges to an element in *X*. Examples of complete metric spaces are:

- Euclidean space $\mathbb{R}^n$.
- The metric space $C[a,b]$ of continuous functions on the interval $[a,b]$.
- By definition, Banach spaces are complete normed linear spaces. A normed linear space has a norm based on a metric, that is, $d(x,y) = \|x-y\|$.
- $L^p(0,1)$ is the Banach space of functions $f : [0,1] \to \mathbb{R}$ defined by the norm:

$$\|f\|_p = \left(\int_0^1 |f(x)|^p dx\right)^{1/p} < \infty \text{ for } 1 \le p < \infty.$$

Definition 19.3 An *open cover* of a set E in a metric space X is a collection $\{G_j\}$ of open subsets of X such that $E \subset \bigcup_j G_j$. Finally we say that a subset K of a metric space X is *compact* if every open cover of K contains a finite subcover, that is $K \subset \bigcup_{j=1}^{N} G_j$ for some finite N.

We now examine functions that map one metric space into another one. In particular, we discuss the *concepts of continuity* and *Lipschitz continuity* because they are used when proving the existence and uniqueness of solutions and ordinary and stochastic differential equations, a topic that we introduce in Chapter 24.

It is convenient to discuss these concepts in the context of metric spaces.

Definition 19.4 Let (X,d_1) and (Y,d_2) be two metric spaces. A function f from X into Y is said to be *continuous at the point* $a \in X$ if for each $\varepsilon > 0$ there exists $\delta > 0$ such that:

$$d_2(f(x),f(a)) < \varepsilon \text{ whenever } d_1(x,a) < \delta.$$

We should note that this definition refers to the continuity of a function at a single point. Thus, a function can be continuous at some points and discontinuous at other points.

Definition 19.5 A function f from a metric space (X,d_1) into a metric space (Y,d_2) is said to be *uniformly continuous* on a set $E \subset X$ if for each $\varepsilon > 0$ there exists $\delta > 0$ such that:

$$d_2(f(x),f(y)) < \varepsilon \text{ whenever } x,y \in E \text{ and } d_1(x,y) < \delta.$$

If the function f is uniformly continuous then it is continuous but the converse is not necessarily true. Uniform continuity holds for all points in the set E whereas 'normal' continuity is applicable at a single point.

Definition 19.6 Let $f : [a, b] \to \mathbb{R}$ be a real-valued function and suppose we can find two constants M and α such that $|f(x) - f(y)| \leq M|x - y|^{\alpha}$, $\forall x, y \in [a, b]$and $\alpha > 0$. Then we say that f satisfies a Lipschitz condition of order α and we write $f \in Lip(\alpha)$.

We take an example. Let $f(x) = x^2$ on the interval $[a, b]$. Then:

$$\begin{aligned}
&|f(x) - f(y)| = \left|x^2 - y^2\right| = |(x + y)(x - y)| \leq |x + y|\,|x - y| \\
&\leq (|x| + |y|)\,|x - y| \\
&\leq M\,|x - y|\,, \text{ where } M = 2\max(|a|\,, |b|).
\end{aligned}$$

Hence, $f \in Lip(1)$. A concept related to Lipschitz continuity is contraction.

Definition 19.7 Let (X, d_1) and (Y, d_2) be metric spaces where d_1 and d_2 are metrics. A transformation T from X into Y is called a *contraction* if there exists a number $\lambda \in (0, 1)$ such that:

$$d_2(T(x), T(y)) < \lambda d_1(x, y)\ \forall x, y \in X.$$

In general, a contraction maps a pair of points into another pair of points that are closer together. A contraction is always continuous.

The ability to discover and apply contraction mappings has considerable theoretical and numerical value. For example, it is possible to prove that SDEs have unique solutions by the application of fixed-point theorems (Bharucha-Reid, 1972; Tsokos and Padgett, 1974). Some specific examples are:

- Brouwer's fixed-point theorem.
- Kakutani's fixed-point theorem.
- Banach's fixed-point theorem.
- Schauder's fixed-point theorem.

Our interest here lies in the following fixed-point theorem.

Theorem 19.1 (*Banach Fixed-Point Theorem*) Let T be a contraction of a complete metric space X into itself:

$$d\,(T(x), T(y)) \leq \lambda d\,(x, y)\,,\ \lambda \in (0,1).$$

Then T has a unique fixed point x. Moreover, if x_0 is any point in X and the sequence $\{\, x_n\}$ is defined recursively by the formula $x_n = T(x_{n-1})$, $n = 1, 2, \ldots$ then $\lim x_n = \bar{x}$ and:

$$d(\overline{x}, x_n) \leq \frac{\lambda}{1 - \lambda} d(x_{n-1}, x_n) \leq \frac{\lambda}{1 - \lambda} d(x_0, x_1). \tag{19.25}$$

In general, we assume that X is a Banach space and that T is a linear or nonlinear mapping of X into itself. We then say that x is a fixed point of T if $Tx = x$.

We take some examples. Let X be the real line and define $Tx = x^2$. Then $x = 0$ and $x = 1$ are fixed points of T. Another example is the *integral equation*:

$$Tx = x(0) + \int_0^1 x(\xi)d\xi$$

where X is the set of continuous functions on the interval $[0, 1]$. The fixed points of this mapping are functions of the form $x(t) = ke^t,\ t \in [0, 1]$ where k is a constant.

It is important to distinguish between two classes of fixed-point theorems. First, *algebraic* (or *constructive*) *fixed-point theorems* give a method for finding the fixed point which can also be called the *iteration* or *successive approximation* procedure (see Bharucha-Reid, 1972). Second, *topological fixed-point theorems* are strictly *existence theorems*, that is they establish conditions under which a fixed point exists but they do not provide an algorithm or a method for actually finding the fixed point.

A generalisation of the above is due to Kolmogorov and it is used in proving a number of relevant theorems.

Theorem 19.2 If T is a mapping of a Banach space X into itself and if T^n is a contraction for some positive integer n, then T has a fixed point.

For a nice overview of fixed-point theorems, see Smart (1974).

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 20

The Finite Difference Method for PDEs: Mathematical Background

20.1 INTRODUCTION AND OBJECTIVES

In this chapter we give an overview of linear one-factor partial differential equations that are used to price derivatives. Mathematically, these are time-dependent *convection–diffusion–reaction* equations. We study their qualitative properties including transforming them to more manageable forms. In most cases these PDEs do not have an analytical solution and for this reason we use numerical methods. In this book we have chosen the popular *Finite Difference Method* (FDM) because it is a mature branch of numerical analysis, it is easy to implement and it has good run-time performance and accuracy characteristics. We are interested in the PDE that describes the Black–Scholes equation and its generalisations and many of our results are applicable to a range of one-factor and two-factor PDEs.

The goal of this chapter is to introduce enough mathematics to model partial differential equations, approximate them by FDM algorithms and then map these algorithms to C++. In particular, we focus on option pricing using the Black–Scholes PDE. We assume that the reader has some knowledge of PDEs and FDM. For background, see Thomas (1995) for a discussion of FDM for time-dependent parabolic PDEs and their numerical approximation by FDM and Duffy (2006) for applications to computational finance, including a numerical analysis of finite difference schemes.

A good way to prepare for the material in this and the following chapters is to review the topics in Chapter 13, especially Sections 13.3 and 13.4 (Crank–Nicolson and ADE methods), Section 13.2 (solving tridiagonal matrix systems) and Section 13.5 (spline interpolation).

The material in this chapter is at an intermediate level of complexity. We have tried to make it accessible to readers with limited exposure to PDE and FDM theory. For this group of readers we recommend Smith (1978) as a practical introduction. For computational finance applications, see Duffy (2006).

20.2 GENERAL CONVECTION–DIFFUSION–REACTION EQUATIONS AND BLACK–SCHOLES PDE

We give an overview of convection–diffusion–reaction equations in n space dimensions and we then discuss special cases for the Black–Scholes equation.

In general, the PDEs of relevance are of the *convection–diffusion–reaction* type in n space variables and one time variable. The space variables correspond to underlying financial quantities such as an asset, volatility or interest rate while the non-negative time variable t is bounded above by the *expiration* T. The space variables are usually defined in their respective positive half-planes.

We model derivatives by *initial boundary value problems* of parabolic type. To this end, consider the general parabolic equation:

$$Lu \equiv \sum_{i,j=1}^{n} a_{ij}(x,t)\frac{\partial^2 u}{\partial x_i \partial x_j} + \sum_{j=1}^{n} b_j(x,t)\frac{\partial u}{\partial x_j} + c(x,t)u - \frac{\partial u}{\partial t} = f(x,t) \tag{20.1}$$

where the functions a_{ij}, b_j, c and f are real valued, $a_{ij} = a_{ji}$ and:

$$\sum_{i,j=1}^{n} a_{ij}(x,t)a_i a_j > 0 \text{ if } \sum_{j=1}^{n} a_j^2 > 0. \tag{20.2}$$

In equation (20.1) the variable x is a point in n-dimensional space and t is considered to be a positive time variable. Equation (20.1) is the general equation that describes the behaviour of many derivative types. For example, in the one-dimensional case ($n = 1$) it reduces to the Black–Scholes equation (here $t^* = T - t$):

$$\frac{\partial V}{\partial t^*} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + (r - D)S\frac{\partial V}{\partial S} - rV = 0 \tag{20.3}$$

where V is the derivative type (for example, a call or put option), S is the underlying asset (or stock), σ is the constant volatility, r is the constant interest rate and D is a constant dividend. Equation (20.3) is a one-factor case and it can be generalised, for example to the multivariate case:

$$\frac{\partial V}{\partial t^*} + \sum_{j=1}^{n}(r - D_j)S_j\frac{\partial V}{\partial S_j} + \frac{1}{2}\sum_{i,j=1}^{n}\rho_{ij}\sigma_i\sigma_j S_i S_j \frac{\partial^2 V}{\partial S_i \partial S_j} - rV = 0. \tag{20.4}$$

This equation models a multi-asset environment. In this case σ_i is the volatility of the ith asset and ρ_{ij} is the correlation ($-1 \leq \rho_{ij} \leq 1$) between assets i and j. The term D_j is the constant dividend for asset j. In this case we see that the elliptic part of equation (20.4) ($t^* = T - t$, where T is the expiration) is written as the sum of three terms:

- Interest earned on cash position:

$$r\left(V - \sum_{j=1}^{n} S_j \frac{\partial V}{\partial S_j}\right). \tag{20.5}$$

- Gain from dividend yield:

$$\sum_{j=1}^{n} D_j S_J \frac{\partial V}{\partial S_j}. \tag{20.6}$$

- Hedging costs or slippage:

$$-\frac{1}{2}\sum_{i,j=1}^{n} \rho_{ij}\sigma_i\sigma_j S_i S_j \frac{\partial^2 V}{\partial S_i \partial S_j}. \tag{20.7}$$

Our interest is in discovering robust numerical schemes that produce reliable and accurate results irrespective of the size of the parameter values in equation (20.4).

Equation (20.1) has an infinite number of solutions in general. In order to define a unique solution, we need to define some constraints. To this end, we define *initial condition* and *boundary conditions* for (20.1). We achieve this by defining the space in which equation (20.1) is assumed to be valid. In general, we note that there are three types of boundary conditions associated with equation (20.1). These are:

- First boundary value problem (*Dirichlet problem*).
- Second boundary value problem (*Neumann, Robin problems*).
- *Cauchy problem*.

The first boundary value problem is concerned with the solution of equation (20.1) in a bounded domain $D = \Omega \times (0, T)$, where Ω is a bounded subset of $\mathbb{R}^n$ and T is a positive number. In this case we seek a solution of equation (20.1) satisfying the conditions:

$$\begin{aligned} &u\,\big|_{t=0} = \phi(x) \quad \text{(inital condition)} \\ &u\,|\Gamma = \psi(x,t) \ \text{(boundary condition)} \end{aligned} \tag{20.8}$$

where Γ is the boundary of Ω. The boundary conditions in (20.8) are called *Dirichlet boundary conditions* because the solution is given on the boundary. These conditions arise when we model single and double barrier options in the one-factor case (see Duffy, 2006). They also occur when we model plain options on a transformed or truncated domain.

The second boundary value problem is similar to (20.8) except that instead of giving the value of u on the boundary Γ the *directional derivatives* are included, as seen in the following example:

$$\left(\frac{\partial u}{\partial \eta} + a(x,t)u\right) = \psi(x,t), \quad x \in \Gamma. \tag{20.9}$$

In this case $a(x,t)$ and $\psi(x,t)$ are known functions of x and t, and $\frac{\partial}{\partial \eta}$ denotes the derivative of u with respect to the outward normal η at Γ. A special case of (20.9) is when $a(x,t) \equiv 0$; then

(20.9) represents *Neumann boundary conditions*. Finally, the solution of the Cauchy problem for equation (20.1) in the infinite region $\mathbb{R}^n \times (0, T)$ is given by the initial condition:

$$u\big|_{t=n} = \varphi(x) \tag{20.10}$$

where $\varphi(x)$ is a given continuous function and $u(x, t)$ satisfies equation (20.1) in $\mathbb{R}^n \times (0, T)$ and the initial condition (20.10). This problem allows negative values of the components of the independent variable $x = (x_1, \ldots, x_n)$. A special case of the Cauchy problem can be seen when modelling one-factor European and American options where x plays the role of the underlying asset S. Boundary conditions are given by values at $S = 0$ and at $S = \infty$. For European call options these conditions are:

$$\begin{aligned} &C(0, t) = 0 \\ &C(S, t) \to S \text{ as } S \to \infty. \end{aligned} \tag{20.11}$$

Here C (the role played by u in equation (20.1)) is the price of the call option. For European put options the boundary conditions are:

$$\begin{aligned} &P(0, t) = Ke^{-rt} \\ &P(S, t) \to 0 \text{ as } S \to \infty. \end{aligned} \tag{20.12}$$

Here P (the role played by u in equation (20.1)) is the variable representing the price of the put option, K is the strike price, r is the risk-free interest rate, T is the expiration and t is the current time.

We sometimes assume the following *'canonical' form* for the operator L in equation (20.1) in the one-factor case:

$$Lu \equiv -\frac{\partial u}{\partial t} + \sigma(s, t)\frac{\partial^2 u}{\partial x^2} + \mu(x, t)\frac{\partial u}{\partial x} + b(x, t)u = f(x, t) \tag{20.13}$$

where σ, μ, b and f are known functions of x and t.

For completeness, we formulate the one-factor initial boundary value problem whose solution we wish to approximate using the finite difference method. To this end, we define the interval $\Omega = (A, B)$ where A and B are two real numbers with $A < B$. Further, let $T > 0$ and $D = \Omega \times (0, T)$. The statement is:

Find a function $u : D \to \mathrm{R}^1$ such that:

$$Lu = -\frac{\partial u}{\partial t} + \sigma(x, t)\frac{\partial^2 u}{\partial x^2} + \mu(x, t)\frac{\partial u}{\partial x} + b(x, t)u = f(x, t) \text{ in } D \tag{20.14}$$

$$u(x, 0) = \varphi(x), x \in \Omega \tag{20.15}$$

$$u(A, t) = g_0(t), u(B, t) = g_1(t), t \in (0, T). \tag{20.16}$$

The initial boundary value problem (20.14)–(20.16) subsumes many specific cases (in particular it is a generalisation of the Black–Scholes equation).

In general, the coefficients $\sigma(x, t)$ and $u(x, t)$ represent *volatility* (diffusivity) and *drift* (convection), respectively. Equation (20.14) is called a *convection–diffusion–reaction* equation. It serves as a model for many kinds of problems. Much research has been carried out in this

area, both on the continuous problem and its discrete formulations (for example, using finite difference and finite element methods).

The essence of the finite difference method is to discretise equations (20.14)–(20.16) by defining *discrete mesh points* and approximating the derivatives of the unknown solution of this system at these mesh points. The goal is to find accurate schemes that will be implemented in a programming language such as C++ and C#. Some typical attention points are:

- The PDE being approximated may need to be preprocessed in some way, for example transforming it from one on a semi-infinite domain to one on a bounded domain.
- Determining which specific finite difference scheme(s) to use based on requirements such as accuracy, efficiency and maintainability.
- Essential difficulties to resolve: *convection dominance*, avoiding oscillations and how to handle discontinuous initial conditions, for example.
- Developing the algorithms and assembling the discrete system of equations prior to implementation.

20.3 PDE PREPROCESSING

A PDE is defined in a region of two-dimensional space defined by the variables x (the underlying space) and time t. Regarding the x variable, the range of values can be:

$-\infty < x < \infty$ (infinite domain)

$0 < x < \infty$ (semi-infinite domain)

$A < x < B$, where $-\infty < A < B < \infty$ (bounded domain).

When approximating PDEs by FDM the underlying domain must be transformed to a bounded domain. Thus, we can decide to modify the PDE in some way before approximating it by the FDM:

- Change the coefficients of the PDE.
- Truncate an infinite or semi-infinite domain to a bounded domain.
- Transform an infinite or semi-infinite domain to a bounded domain by a change of independent variables.
- Change the structure of a PDE (for example, we can use an *integrating factor* to transform a non-conservative PDE to a conservative PDE).

It is important to determine what the consequences are when modifying a PDE. We now discuss each of the above topics.

20.3.1 Log Transformation

This is a popular method to transform the Black–Scholes PDE on a semi-infinite interval to a PDE with constant coefficients on an infinite interval. For example, we define

$y = \log S$ and we use this new variable in equation (20.3) to produce a PDE with constant coefficients:

$$\frac{\partial V}{\partial t^*} + \frac{1}{2}\sigma^2 \frac{\partial^2 v}{\partial y^2} + v\frac{\partial V}{\partial y} - rV = 0 \text{ where } v = r - D - \frac{1}{2}\sigma^2 \tag{20.17}$$

which may be more computationally attractive in certain cases. Furthermore, we can remove the explicit dependence on the convection term by writing equation (20.17) in *conservative form*:

$$\frac{\partial V}{\partial t^*} + A\frac{\partial}{\partial y}\left(B\frac{\partial V}{\partial y}\right) - rV = 0 \text{ where } A = \frac{1}{2}\sigma^2 e^{cy}, B = e^{-cy}, c = \frac{2v}{\sigma^2}. \tag{20.18}$$

20.3.2 Reduction of PDE to Conservative Form

Equation (20.18) is an example of a PDE that results from the application of an integrating factor to reduce a PDE to a simpler form. We now apply the technique to the ordinary differential equation:

$$Lu \equiv a(x)u'' + b(x)u' + c(x). \tag{20.19}$$

We transform equation (20.19) into a more convenient form (the so-called *normal form*) by a change of variables. For convenience we assume that the right-hand-side term f in equation (20.19) is zero. To this end, define:

$$p(x) = \exp \int \frac{b(x)}{a(x)}dx, q(x) = \frac{c(x)p(x)}{a(x)}. \tag{20.20}$$

If we multiply equation (20.19) (note $f = 0$) by $p(x)/a(x)$ we get:

$$\frac{d}{dx}p(x)\frac{du}{dx} + q(x)u = 0. \tag{20.21}$$

This equation is sometimes known as the *self-adjoint form*. A further change of variables:

$$\varsigma = \int \frac{dx}{p(x)} \tag{20.22}$$

allows us to write (20.21) in an even simpler form:

$$\frac{d^2u}{d\varsigma^2} + p(x)q(x)u = 0. \tag{20.23}$$

Equation (20.23) is easier to solve than equation (20.19) in general.

20.3.3 Domain Truncation

Many PDEs in computational finance are defined on semi-infinite intervals. Unfortunately, infinity does not fit into a computer very well and we then decide to transform the domain to a bounded domain. A common rule of thumb is to define the truncated *far-field boundary* as a multiple (three or four) of the strike price. In other words, an *artificial boundary* is created (see Kangro and Nicolaides, 2000 for a detailed analysis). Some of the mathematical issues that the authors address are:

- Prove that the solutions of the PDEs (both in the original and truncated forms) exist and are unique (Friedman, 1992, especially Theorem 10 of Chapter 2 of that book).
- The solution of the truncated PDE is an accurate approximation of the solution of the original PDE.
- Computing an artificial boundary and defining boundary conditions on it.

A detailed discussion of these topics is outside the scope of this book. Regarding the last topic, one approach is to define Dirichlet boundary conditions at the artificial boundary by using the PDE's initial condition (or payoff function in the case of option pricing problems). This could be one ploy when determining what the most accurate boundary conditions are for a given problem.

20.3.4 Domain Transformation

Instead of truncating a domain (in Section 20.3.3) we can map an infinite or semi-infinite domain to a bounded domain, for example to the interval (0, 1). This is an elegant approach and it avoids our having to truncate the domain. We have discussed the technique in Duffy and Germani (2013); more examples for two-factor problems may be found in Lewis (2016) and Wilmott, Lewis and Duffy (2014). In fact, we discuss the technique in detail in Chapter 23.

The idea behind domain transformation is to define a new independent space variable on the unit interval (0, 1). We take the transformation:

$$y = \frac{x}{x + \alpha} \tag{20.24}$$

where α is a *scale factor* that can be chosen by the user. We have found this transformation to be useful for our applications and we remark that there are other possible transformations, for example:

$$\begin{aligned} y &= \tanh\ ax \\ y &= \frac{1}{1 + ax} \\ y &= e^{-as} \\ y &= 1 + e^{-ax} \\ y &= \coth\ ax. \end{aligned} \tag{20.25}$$

These and other formulae are used in fluid dynamics and computational finance. For example, the second formula in (20.25) has been used in two-factor mortgage modelling in which

the independent variables are house price and interest rate (Sharp, 2006). The third transformation is suitable for the *constant elasticity of variance* (CEV) model. We discuss the CEV model in Chapters 31 and 32.

Having decided on the transformation (20.24) we can now rewrite the PDE (20.13) in the new coordinates (y, t). To this end, we perform some algebra. First, the *inverse transformation* is given by:

$$x = \frac{\alpha y}{1-y}$$

and furthermore:

$$\begin{aligned}
\frac{dy}{\partial x} &= \frac{(1-y)^2}{\alpha}, \frac{d^2y}{dx^2} = -\frac{2}{\alpha}(1-y)^3 \\
\frac{\partial u}{\partial x} &= \alpha^{-1}(1-y)^2\frac{\partial u}{\partial y} \\
\frac{\partial^2 u}{\partial x^2} &= \alpha^{-2}(1-y)^2\frac{\partial}{\partial y}\left\{(1-y)^2\frac{\partial u}{\partial y}\right\}.
\end{aligned}$$

In general, we model convection–diffusion–reaction PDEs of the form:

$$\frac{\partial u}{\partial t} = a(x,t)\frac{\partial^2 u}{\partial x^2} + b(x,t)\frac{\partial u}{\partial x} + c(x,t)u + f(x,t), \quad 0 < x < \infty. \tag{20.26}$$

When applying the transformation (20.24) we can transform the PDE (20.26) into one involving the variable y with new (modified) drift and diffusion terms:

$$\frac{\partial u}{\partial t} = A(x,t)\frac{\partial^2 u}{\partial y^2} + B(x,t)\frac{\partial u}{\partial y} + C(y,t)u + F(y,t), \quad 0 < x < \infty. \tag{20.27}$$

In computational finance the drift and diffusion terms tend to be low-order monomials in x and when transformed they will be products of low-order polynomials, for example in the case when $\alpha = 1$:

$$x^\beta\frac{\partial^2 u}{\partial x^2} = \left(\frac{y}{1-y}\right)^\beta(1-y)^2\frac{\partial}{\partial y}\left\{(1-y)^2\frac{\partial u}{\partial y}\right\} = y^\beta(1-y)^{2-\beta}\frac{\partial}{\partial y}\left\{(1-y)^2\frac{\partial u}{\partial y}\right\}$$

and

$$x^\beta\frac{\partial^2 u}{\partial x} = y^\beta(1-y)^{2-\beta}\frac{\partial u}{\partial y}, \text{ for } \beta \in \mathbb{R}.$$

We now examine the Black–Scholes PDE:

$$\frac{\partial V}{\partial t} = \frac{1}{2}\sigma^2 S^2\frac{\partial^2 u}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV, \quad 0 < S < \infty, 0 < t < T. \tag{20.28}$$

We apply the transformation (20.24) and we then get a PDE that is a special case of PDE (20.28) in the case when $\alpha = 1$, namely:

$$\frac{\partial V}{\partial t} = \frac{1}{2}\sigma^2 y^2 \frac{\partial}{\partial y}\left\{(1-y)^2 \frac{\partial V}{\partial y}\right\} + ry(1-y)\frac{\partial V}{\partial y} - rV$$

or equivalently:

$$\frac{\partial V}{\partial t} = \frac{1}{2}\sigma^2 y^2 (1-y)^2 \frac{\partial^2 V}{\partial y^2} + \left\{ry(1-y) - \sigma^2 y^2 (1-y)\right\}\frac{\partial V}{\partial y} - rV.$$

Finally, we mention that these techniques to simplify PDEs can also be used for two-factor PDE models.

20.4 MAXIMUM PRINCIPLES FOR PARABOLIC PDEs

We now introduce a number of results that describe how solutions of equation (20.1) (and its one-factor equivalent subcase (20.13)) behave in the domain of interest. In particular, we discuss the *continuous maximum principle* that states the conditions under which the solution of (20.1) remains positive when its corresponding initial and boundary values are positive. Let $D = \Omega \times (0, T)$, where Ω is a bounded region in space and let $\bar{D}$ be its closure. The following results are proven in Il'in, Kalashnikov and Oleinik (1962).

Theorem 20.1 Assume that the function $u(x, t)$ is continuous in D and assume that the coefficients in (20.13) are continuous. Suppose that $Lu \leq 0$ in $\bar{D}/\Gamma$ where $b(x, t) < M$ (M is some constant) and suppose furthermore that $u(x, t) \geq 0$ on Γ. Then:

$$u(x, t) \geq 0 \text{ in } \bar{D}.$$

This theorem states that positive initial and boundary conditions lead to a positive solution in the interior of the domain D.

Theorem 20.2 Suppose that $u(x, t)$ is continuous and satisfies (20.13) in $\bar{D}\backslash\Gamma$ where $f(x, t)$ is a bounded function ($|f| \leq N$) and $b(x, t) \leq 0$. If $|u(x, t)|\,\Gamma \leq m$ then:

$$|u(x, t)| \leq Nt + m \text{ in } \bar{D}. \tag{20.29}$$

We can sharpen the results of Theorem 20.2 in the case where $b(x, t) \leq b_0 < 0$. In this case estimate (20.29) is replaced by:

$$|u(x, t)| \leq \max\left\{\frac{-N}{b_0}, m\right\}. \tag{20.30}$$

Proof: Define the *'barrier' function* $w^{\pm}(x, t) = N_1 \pm u(x, t)$ where $N_1 = \max\left\{\frac{-N}{b_0}, m\right\}$. Then $w^{\pm} \geq 0$ on $\Gamma$and $Lw^{\pm} \leq 0$. By Theorem 20.1 we deduce that $w^{\pm} > 0$ in $\bar{D}$. The desired result follows. ■

The inequality (20.29) states that the growth of u is bounded by its initial and boundary values. It is interesting to note in the special case $b \equiv 0$ and $f \equiv 0$ that we can deduce the following maximum and minimum principles for the heat equation.

Corollary 20.1 Assume that the conditions of Theorem 20.2 are satisfied and that $b \equiv 0$, $c \equiv 0, a \equiv 1$ and $f \equiv 0$ in equation (20.13). Then the function $u(x, t)$ takes its least and greatest values on Γ, that is:

$$m_1 = \min u(x, t) \leq u(x, t) \leq \max\ u(x, t) \equiv m_2.$$

The results from Theorems 20.1 and 20.2 and Corollary 20.1 are very appealing: you cannot get negative values of the solution u from positive input. We shall produce similar results for the finite difference schemes that approximate (20.13) and we shall show how it is possible to prove stability by a *discrete maximum principle* without having to resort to the more restricted and misused *von Neumann stability technique*. We now conclude this section with a result that is of particular importance when proving positivity of the solutions of (20.13) in unbounded domains, for example as is the case with European and American option problems.

Theorem 20.3 (*Maximum Principle for the Cauchy Problem*) Let $u(x, t)$ be continuous and bounded below in $D = \mathbb{R}^n \times (0, T)$, that is $u(x, t) \geq -m, m > 0$. Suppose further that $u(x,t)$ has continuous derivatives in D up to second order in x and first order in t and that $Lu \leq 0$. Let the functions σ, μ and b satisfy:

$$|\sigma(x, t)| \leq M(x^2 + 1), \quad |\mu(x, t)| \leq M\sqrt{x^2 + 1}, \quad b(x, t) \leq M.$$

Then $u(x, t) \geq 0$ everywhere in D if $u(x, 0) \geq 0$.

We can apply Theorem 20.3 to the one-factor Black–Scholes equation in order to convince ourselves that the price of an option can never take negative values.

20.5 THE FICHERA THEORY

Examining PDE (20.13) again we see that it has a second-order derivative in x. We now restrict our attention to the case in which the PDE is defined on the unit interval (0, 1). Then there are two mutually exclusive possibilities for adding boundary conditions at the points $\{0, 1\}$:

- Option 1: no boundary conditions are needed nor are they allowed.
- Option 2: boundary conditions are needed and can be determined.

The topic of discovering appropriate boundary conditions is somewhat of a fuzzy one. In this section we define a process to help us find the mathematically and financially appropriate boundary conditions for general parabolic partial differential equations with *non-negative characteristic* form (Oleinik and Radkevic, 1973) and in particular for one-factor and multi-factor PDEs in derivatives pricing. To this end, we discuss the *Fichera theory*.

The Fichera theory is applicable to a range of elliptic, parabolic and hyperbolic PDEs and it is particularly useful for PDEs whose coefficients are zero on certain boundaries of a

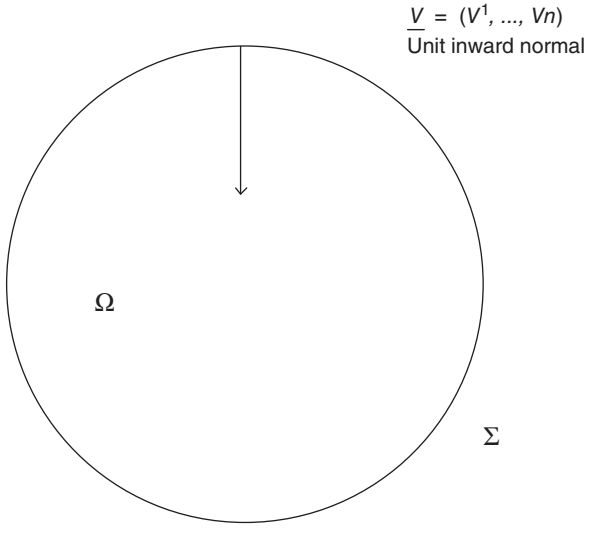


FIGURE 20.1 Region and boundary

bounded domain Ω in n-dimensional space (for more information, see Fichera, 1956, Oleinik and Radkevic, 1973). We depict this domain, its boundary Σ and inward unit normal $\underline{\nu}$ in Figure 20.1. Let us examine the elliptic equation defined by:

$$Lu \equiv \sum_{i,j=1}^{n} a_{ij} \frac{\partial^2 u}{\partial x_i \partial x_j} + \sum_{i=1}^{n} b_i \frac{\partial u}{\partial x_i} + cu = f \text{ in } \Omega \tag{20.31}$$

where:

$$\sum_{i,j=1}^{n} a_{ij} \xi_i \xi_j \geq 0 \text{ in } \Omega \cup \Sigma \quad \forall \xi = (\xi_1, \ldots, \xi_n) \in \mathbb{R}^n. \tag{20.32}$$

The *characteristic form* (20.32) is strictly positive in most cases but we are interested in where it is zero, and in particular in finding the subsets of the boundary Σ where it is zero. To this end, we define:

$$\sum\nolimits_3 = \left\{ x \in \sum : \sum_{i,j=1}^{n} a_{ij} \nu_i \nu_j > 0 \right\}. \tag{20.33}$$

Only on the boundary where the characteristic form is zero (the *characteristic boundary*, that is, $\Sigma - \Sigma_3$) do we define the *Fichera function*:

$$b \equiv \sum_{i=1}^{n} \left(b_i - \sum_{k=1}^{n} \frac{\partial a_{ik}}{\partial x_k} \right) \nu_i \tag{20.34}$$

where ν_1 is the ith component of the inward normal $\underline{\nu}$ on Σ. Having computed the Fichera function, we then determine its sign on all parts of the characteristic boundary. There are three mutually exclusive options:

$$\begin{aligned} &\Sigma_0 : b = 0 \\ &\Sigma_1 : b > 0 \\ &\Sigma_2 : b < 0. \end{aligned} \tag{20.35}$$

In other words, the boundary consists of the following sub-boundaries:

$$\Sigma \equiv \Sigma_0 \cup \Sigma_1 \cup \Sigma_2 \cup \Sigma_3.$$

We demand that no boundary conditions (BC) are allowed when the Fichera function is zero or positive (in other words, Σ_0 and Σ_1) and then the PDE (20.31) degenerates to a lower-order PDE on these boundaries. When $b > 0$ (that is, on Σ_2) we need to define a boundary condition. What that boundary condition is depends on the context.

We write parabolic PDEs in the form:

$$\frac{\partial u}{\partial t} = Lu + f \text{ or } -\frac{\partial u}{\partial t} + Lu = -f \tag{20.36}$$

where the elliptic operator L has already been defined in equation (20.31). Then the same conclusions concerning characteristic and non-characteristic boundaries hold as in the elliptic case. In other words, we focus on the elliptic part of the PDE when we wish to calculate the Fichera function.

Let us take an example. In this case we examine the PDE that prices a zero-coupon bond under a *Cox–Ingersoll–Ross (CIR) interest-rate model*:

$$\frac{\partial B}{\partial t} + \frac{1}{2}\sigma^2 r \frac{\partial^2 B}{\partial r^2} + (a - cr)\frac{\partial B}{\partial r} - rB = 0. \tag{20.37}$$

Please note that we are using backward time in this case, hence the sign difference when compared with equation (20.36). You should be aware of this subtle point. Using the definition in equation (20.34) we see that the Fichera function is given by:

$$b = ((a - cr) - \sigma^2/2)\nu \tag{20.38}$$

where ν is the inward unit normal at $r = 0$ ($\nu = 1$) or at $r = 1(\nu = -1)$.

We are interested in the case $r = 0$ (because this is the only characteristic boundary for this problem) and we see that each choice in (20.35) can be valid depending on the relative sizes of the parameters a and σ:

$$\begin{aligned} &\Sigma_2 : b < 0 \rightarrow \sigma > \sqrt{2a} \text{ (BC needed)} \\ &\Sigma_0 : b = 0 \rightarrow \sigma = \sqrt{2a} \text{ (BC not needed)} \\ &\Sigma_1 : b > 0 \rightarrow \sigma < \sqrt{2a} \text{ (BC not needed).} \end{aligned} \tag{20.39}$$

In the last two cases we see that no boundary condition is allowed (or needed) and then the PDE (20.37) reduces to the hyperbolic PDE:

$$\frac{\partial B}{\partial t} + a\frac{\partial B}{\partial r} = 0 \tag{20.40}$$

on $r = 0$. These results are consistent with the conclusions in Tavella and Randall (2000, pp. 126–128). In general, we need to solve (20.40) either analytically or numerically. From a financial perspective, the third condition in (20.39) when the Fichera function is positive states that the interest rate cannot become negative. The inequality in this case is called the *Feller condition* for the CIR process. We have also investigated it for the process for the Heston square-root model and we have reproduced the well-known *Feller condition*:

$$\kappa\theta \geq \tfrac{1}{2}\sigma^2. \tag{20.41}$$

A full discussion of this topic and the corresponding notation in equation (20.41) is outside the scope of this book. For a discussion on solutions of the Heston PDE using finite difference methods, see the thesis by Sheppard (2007) (also available from the site www.datasimfinancial.com).

We summarise the steps to take when applying the Fichera theory to the determination of boundary conditions corresponding to initial boundary value problems. Readers may have difficulty in calculating the Fichera function. The mathematics is not difficult to understand but the steps must be correctly executed in the following prescribed order:

1. Determine the boundary of the domain and its unit inward normal. In many cases the boundary is the union of hyperplanes parallel to the coordinate axes.
2. Calculate the characteristic form (equation (20.33)) that leads to the characteristic boundary.
3. Calculate the Fichera function (equation (20.34)) on the characteristic boundary.
4. Determine the subset of the boundary where no boundary conditions are needed (equation (20.35)). Determine what kind of equation is defined on this subset.
5. In the case where the Fichera function is negative you must also determine what the boundary condition will be. This is a Dirichlet boundary condition in many cases.

20.5.1 Example: Boundary Conditions for the One-Factor Black–Scholes PDE

We now apply the Fichera function (20.34) to the transformed Black–Scholes PDE described by the last equation in Section 20.3.4. In this case the Fichera function becomes:

$$b = (ry(1-y) - 2\sigma^2 y(1-y)^2)v(y) \tag{20.42}$$

where:

$$v(0) = 1,\ v(1) = -1.$$

We thus see that the Fichera function is zero at the end points $y = 0$ and $y = 1$ and then no boundary conditions need be prescribed at these points. Instead, the PDE degenerates to the following ordinary differential equation at these points:

$$\frac{\partial V}{\partial t} + rV = 0 \text{ at } y = 0,\ y = 1. \tag{20.43}$$

The solution to (20.41) is given by:

$$V(y,t) = C(y)e^{-rt} \text{ at } y = 0, y = 1 \tag{20.44}$$

where the factor $C(y)$ can be found by demanding *compatibility* between the solution (20.44) and the initial condition corresponding to the PDE at the points $y = 0, y = 1$. Doing this will lead to the well-known boundary conditions for the Black–Scholes equation. For example, in the case of a put option $C(y) = K$.

20.6 FINITE DIFFERENCE SCHEMES: PROPERTIES AND REQUIREMENTS

Let us review what we have achieved in this chapter. We have described an *initial boundary value problem* (IBVP) consisting of the following components:

- A1: A PDE of convection–diffusion–reaction type.
- A2: The bounded or unbounded region (domain) of (x, t) space in which the PDE is defined.
- A3: The boundary conditions that the solution of the PDE satisfies.
- A4: The solution's initial condition at $t = 0$ (we are using forward time from $t = 0$ up to expiration).

It is advantageous to know something about the qualitative properties of the IBVP before we jump into the coding of the finite difference scheme that will approximate the IBVP. We need to have a good idea of how to analyse the following attention points before we start designing algorithms:

- A5: Initialising the coefficients of the PDE (there may be an analytical formula or we may need to calibrate them). Special attention to be paid to small diffusion or large convection terms (*convection dominance*).
- A6: Transforming an unbounded domain to a bounded/truncated domain.
- A7: Determining what the boundary conditions should be (if it is possible at this stage; we may have to wait until we can define *numerical boundary conditions*). The boundary conditions can be of Dirichlet, Neumann or Robin types. It is even possible that no boundary condition is needed or allowed (see Section 20.5) and we may end up with the challenge of solving a PDE or ODE on a boundary.
- A8: How to handle *discontinuities* in the initial condition (payoff function). We may need to perform a smoothing operation on the initial condition.

It is possible to understand the complexities of these points by studying the relevant literature and attempting to implement them as part of an FDM algorithm. To this end, we address the following steps when mapping the PDE to the FDM:

- A9: Creating a *discrete domain* in (x, t) space; do we choose constant meshes or adaptive meshes?
- A10: Approximating the time, diffusion and convection derivatives by *divided differences*.
- A11: *Numerical boundary conditions*, for example the numerical approximations of the boundary conditions in point A7.
- A12: Computing discrete variants of the initial condition taking points of discontinuity into consideration.

We are not there yet! Approximating derivatives by divided differences introduces errors and this leads us to a discussion of the stability and accuracy of the finite difference scheme that we propose to use. More detailed attention points are:

- A13: Do we use *one-step* or *multistep* methods in time?
- A14: The kind of marching scheme from $t = 0$ to $t = T$ to use (*explicit or implicit methods*).
- A15: Do we create our own solvers to march in the time direction (for example, Crank–Nicolson) or do we use a commercial or open source ODD (ordinary differential differential) solver, for example the Boost C++ *odeint* library.
- A16: Accuracy requirements (*first order*, *second order*, higher order).
- A17: Improving accuracy by using *extrapolation methods*, for example.
- A18: Assembling the discrete system of equations to form a matrix system or nonlinear system of equations.

These 18 attention points form a reasonably complete description of what skills and action points are needed when approximating linear initial boundary value problems using finite difference methods. We shall see examples in this and succeeding chapters.

The above steps and action points are also applicable to two-factor and to nonlinear problems.

20.7 EXAMPLE: A LINEAR TWO-POINT BOUNDARY VALUE PROBLEM

We begin our journey into the land of FDM by examining *time-independent second-order two-point boundary value problems* (TPBVPs) with Dirichlet boundary conditions defined on a bounded interval. TPBVPs are interesting for a number of reasons:

- We can approximate TPBVPs by the finite difference method, assemble the system of equations and solve the system using the Thomas or Double Sweep algorithms that we introduced in Chapter 13. This can be seen as preparation for our treatment of finite difference methods for the Black–Scholes PDE (see Exercise 5). It helps up gain insight into the numerical issues that arise with larger problems.

- *Horizontal Method of Lines* (*Rothe's Method*): we can reduce an initial boundary value problem to a TPBVP by discretising in the t direction (Ladyženskaja, Solonnikov and Ural'ceva, 1988, p. 241; Meyer, 2015, p. 61; Rothe, 1930). For example, we can discretise equation (20.13) in time by the *fully implicit method* (BTCS) (for notation, see Chapter 13) to produce the system of second-order ordinary differential equations:

$$-\frac{u^{n+1}-u^n}{\Delta t}+\sigma(x,t_{n+1})\frac{d^2u^{n+1}}{dx^2}+\mu(x,t_{n+1})\frac{du^{n+1}}{dx}+b(x,t_{n+1})u^{n+1}=f(x,t_{n+1}),$$
$$0\le n\le N-1.$$

In this case we use first-order backward divided differences to approximate the time derivative (another choice is the second-order Crank–Nicolson method based on centred differencing). We augment these ODEs with appropriate boundary and initial conditions.

- Rothe's method is useful for problems with discontinuous payoffs, early-exercise features and problems with jumps and solutions of the Black–Scholes equation that change rapidly with respect to the underlying S at any given time (Meyer, 2015).
- The example shows how some finite difference methods lead to *non-monotone schemes* and how they can be made monotone by applying the *exponential fitting method* (Duffy, 1980, 2006). For a detailed discussion of numerical methods for two-point boundary value problems, see Keller (1968).

20.7.1 The Example

Our aim is to approximate the initial boundary value problem (20.14)–(20.16) by finite difference schemes. We first define some notation. To this end, we divide the interval $[A,B]$ into J equal subintervals:

$$A=x_0<x_1<\cdots<x_J=B$$

and we assume for convenience that the *mesh points* $\left\{x_j\right\}_{j=0}^{J}$ are equidistant, that is:

$$x_j=x_{j-1}+h,\quad j=1,\ldots,J\left(h=\frac{B-A}{J}\right).$$

Furthermore, we divide the interval $[0,T]$ into N equal subintervals:

$$0=t_0<t_1<\cdots<t_N=T$$

where:

$$t_n=t_{n-1}+k,\quad n=1,\ldots,N\left(k=\frac{T}{N}\right).$$

(It is possible to define non-equidistant mesh points in the x and t directions but doing so would complicate the mathematics at this early stage.)

The essence of the finite difference approach lies in replacing the derivatives in equation (20.14) by divided differences at the mesh points (x_J, t_n). We thus define the difference operators in the x direction as follows:

$$D+u_j = (u_{j+1} - u_j)/h, \qquad D_-u_j = (u_j + u_{J-1})/h$$
$$D_0u_j = (u_{j+1} - u_{J-1})/2h, \quad D_+D_-u_j = (u_{j+1} - 2u_j + u_{j-1})/h^2.$$

It can be shown by Taylor expansion that D_+ and D_- are first-order approximations to $\frac{\partial}{\partial x}$ while D_0 is a second-order approximation to $\frac{\partial}{\partial x}$. Finally, D_+D_- is a second-order approximation to $\frac{\partial^2}{\partial x^2}$.

We first consider the following simple TPBVP. We find a function u such that:

$$\begin{aligned} &\sigma\frac{d^2u}{dx^2} + 2\frac{du}{dx} = 0 \quad \text{in } (0,1) \\ &u(0) = 1, u(1) = 0. \end{aligned} \tag{20.45}$$

We assume that σ is a positive constant. We now replace the derivatives in (20.45) by their corresponding centred differences to produce the following finite difference method. We find a mesh function $\left\{U_j\right\}_{j=1}^{J-1}$ such that:

$$\begin{aligned} &\sigma D_+D_-U_j + 2D_0U_j = 0, j = 1, \ldots, J-1 \\ &U_0 = 1, U_J = 0. \end{aligned} \tag{20.46}$$

Scheme (20.46) is sometimes called the *centred difference scheme* and it is a popular way to approximate convection–diffusion equations.

It can be checked that the exact solution of (20.45) is given by:

$$u(x) = \frac{e^{-2x/\sigma} - e^{-2/\sigma}}{1 - e^{-2/\sigma}}. \tag{20.47}$$

It can also be checked that the exact solution of (20.46) is given by:

$$U_j = (\lambda^j - \lambda^J)/(1 - \lambda^J), \quad \text{where } \lambda = \frac{1 - h/\sigma}{1 + h/\sigma}. \tag{20.48}$$

Let us now assume that $\sigma < h$ in equation (20.48). This means that $\lambda < 0$ and thus $U_1 - u(x_1) = \frac{\lambda - \lambda^J}{1-\lambda^J} - \frac{r - r^J}{1-r^J}$, $r = e^{-2h/\sigma}$ is positive or negative depending on whether j is even or odd. Furthermore,

$$\lim_{\sigma \to 0} U_j = ((-1)^j + 1)/2 \text{ if } J \text{ is odd.}$$

Hence U_j oscillates in a bounded fashion for all σ satisfying $\sigma < h$. We thus conclude that centred finite difference schemes are unsuitable for the numerical solution of problem (20.45) when $\sigma < h$. It can even be shown that this difference scheme produces a solution that goes to

infinity as $\sigma \to 0$. This very simple one-dimensional example leads us to conclude that similar problems can be experienced with standard finite difference schemes.

Is there hope? As an alternative to centred divided differences we could approximate (20.45) by *upwind finite difference schemes*:

$$\begin{aligned} &\sigma D_+ D_- U_j + 2D_+ U_j, \quad j = 1, \ldots, j-1 \\ &U_0 = 1, U_j = 0. \end{aligned} \tag{20.49}$$

The solution of (20.49) is given by:

$$U_j = \frac{\lambda^j - \lambda^J}{1 - \lambda^J}, \quad \lambda = \frac{1}{1 + \dfrac{2h}{\sigma}}.$$

We note that:

$$U_1 - u(x_1) = \frac{\lambda - \lambda^J}{1 - \lambda^J} - \frac{r - r^J}{1 - r^J}, \quad r = e^{-2h/\sigma}$$

where $u = u(x)$ is the solution defined by (20.47). We now set $\frac{\sigma}{h} = 1$ and we let $J \to \infty$. Then:

$$U_1 - u(x_1) = \frac{1}{3} - e^{-2} = 0.197998.$$

This means that the maximum pointwise error is approximately 20% no matter what the size of h is. Our conclusion is that centred or one-sided difference schemes will result in either *spurious oscillations* and/or an inaccurate solution for this problem.

We now introduce the class of *exponentially fitted schemes* for general two-point boundary value problems and we apply these schemes to the Black–Scholes equation. Exponentially fitted schemes are stable, have good convergence properties and do not produce spurious oscillations (Duffy, 1980). In order to motivate what an exponentially fitted difference scheme is, let us examine the two-point boundary value problem:

$$\begin{aligned} &\sigma \frac{d^2u}{dx^2} + \mu \frac{du}{dx} = 0 \text{ in } (A, B) \\ &u(A) = \beta_0, \quad u(B) = \beta_1. \end{aligned} \tag{20.50}$$

Here we assume that σ and μ are positive constants for the moment. We now approximate (20.50) by the difference scheme defined as follows:

$$\begin{aligned} &\sigma \rho D + D_- U_j + \mu D_0 U_j = 0, \quad j = 1, \ldots, J-1 \\ &U_0 = \beta_0, U_J = \beta_1 \end{aligned} \tag{20.51}$$

where ρ is a *fitting factor* (this factor is identically equal to 1 in the case of the centred difference scheme (20.46)). We now choose ρ so that the solutions of (20.50) and (20.51) are identical

at the mesh points in the case of constant coefficients. Some simple arithmetic shows that the parameter which realises this requirement is given by:

$$\rho = \frac{uh}{2\sigma}\coth\frac{\mu h}{2\sigma}$$

where $L_k^h U_j^n \equiv -\frac{U_j^{n+1}-U_j^n}{k} + \rho_j^{n+1} D_+ D_- U$ is the *hyperbolic cotangent function* defined by:

$$\coth x = \frac{e^x + e^{-x}}{e^x - e^{-x}} = \frac{e^{2x}+1}{e^{2x}-1}.$$

The fitting factor will be used when developing fitted difference schemes for more general problems, in particular:

$$\begin{aligned} &\sigma(x)\frac{d^2u}{dx^2} + \mu(x)\frac{du}{dx} + b(x)u = f(x) \\ &u(A) = \beta_0,\ u(B) = \beta_1 \end{aligned} \tag{20.52}$$

where σ, μ and b are given continuous functions, and:

$$\sigma(x) \geq 0), \quad \mu(x) \geq \alpha > 0, \quad b(x) \leq 0 \text{ for } x \in (A, B).$$

The fitted difference scheme that approximates (20.52) is now defined by:

$$\begin{aligned} &\rho_j^h D_+ D_- U_j + \mu_j D_0 U_j + b_j U_j = f_j, \quad j = 1, \ldots, j-1 \\ &U_n = \beta_n, \quad U_J = \beta_1 \end{aligned} \tag{20.53}$$

where:

$$\rho_j^h = \frac{\mu_j h}{2}\coth\frac{\mu_j h}{2\sigma_j}, \quad \sigma_j = \sigma(x_j), \quad \mu_j = \mu(x_j), \quad b_j = b(x_j), \quad f_i = f(x_j). \tag{20.54}$$

We now see that the fitting factor is not a constant anymore but a discrete mesh function.

20.8 EXPONENTIALLY FITTED SCHEMES FOR TIME-DEPENDENT PDEs

We discuss how to apply exponentially fitted schemes to the parabolic initial boundary value problem:

$$\begin{aligned} &Lu \equiv -\frac{\partial u}{\partial t} + \sigma(x,t)\frac{\partial^2 u}{\partial x^2} + \mu(x,t)\frac{\partial u}{\partial x} + b(x,t)u = f(x,t) \text{ in } D = (A,B)\times(0,T) \\ &u(x,0) = \varphi(x), \quad x \in (A,B) \\ &u(A,t) = g_0(t), \quad u(B,t) = g_1(t), \quad t \in (0,T). \end{aligned} \tag{20.55}$$

We now discuss how to approximate (20.55). In particular, we propose an exponentially fitted scheme in the space direction and fully implicit discretisation (BTCS) in the time direction. The results are based on Duffy (1980, 2006) where the main theorems are proposed and proven and these results are valid for coefficients that depend on x and t.

We discretise the rectangle $[A, B] \times [0, T]$ as follows:

$$A = x_0 < x_1 < \cdots < x_J = B(h = x_j - x_{j-1}), h \text{ constant}$$
$$0 = t_0 < t_1 < \cdots < t_N = T(k = T/N), k \text{ constant}.$$

Consider again the operator L in equation (20.55) defined by:

$$Lu \equiv -\frac{\partial u}{\partial t} + \sigma(x,t)\frac{\partial^2 u}{\partial x^2} + u(x,t)\frac{\partial u}{\partial x} + b(x,t)u.$$

We replace the derivatives in this operator by their corresponding divided differences and we define the *discrete operator* L_k^h by:

$$L_k^h U_j^n \equiv -\frac{U_j^{n+1} - U_j^n}{k} + \rho_j^{n+1} D_+ D_- U_j^{n+1} + \mu_j^{n+1} D_0 U_j^{n+1} + b_j^{n+1} U_j^{n+1}. \tag{20.56}$$

Here we use the notation:

$$\varphi_j^{n+1} = \varphi(x_j, t_{n+1})$$

and

$$\rho_j^{n+1} \equiv \frac{\mu_j^{n+1} h}{2} \coth \frac{\mu_j^{n+1} h}{2\sigma_j^{n+1}}.$$

Having defined the operator L_h^k we now formulate the *fully discrete scheme* that approximates system (20.55). We find a discrete function $\{U_j^n\}$ such that:

$$\begin{aligned} &L_k^h U_j^n = f_j^{n+1},\ j = 1, \ldots,\ J-1,\ n = 0, \ldots, N-1 \\ &U_0^n = g_0(t_n),\ U_j^n = g_1(t_n),\ n = 0, \ldots, N \\ &U_j^0 = \varphi(x_j),\ j = 1, \ldots,\ J-1. \end{aligned} \tag{20.57}$$

This is a one-step implicit scheme. We wish to prove that scheme (20.57) is stable and *consistent* with the initial boundary value problem. We prove stability of (20.57) by the *discrete maximum principle* instead of the von Neumann stability analysis. The von Neumann approach is well known but the discrete maximum principle is more general, easier to understand and more mathematically correct. It is also the de-facto standard technique for proving stability of finite difference schemes.

Lemma 20.1 Let the discrete function w_j^n satisfy $L_k^h w_j^n \leq 0$ in the interior of the discrete mesh domain with $w_j^n \geq 0$ on the discrete boundary. Then:

$$w_j^n \geq 0, \ \forall j = 0, \ldots, J, n = 0, \ldots, N.$$

Proof: We transform the inequality $L_k^h w_j^n \leq 0$ into an equivalent vector inequality. To this end, define the vector $W^n = (w_1^n, \ldots, w_{j-1}^n)$. Then the inequality $L_k^h w_j^n \leq 0$ is equivalent to the vector inequality:

$$A^n W^{n+1} \geq W^n \tag{20.58}$$

where:

$$A^n = \begin{pmatrix} \ddots & & \ddots & & 0 \\ & \ddots & & t_j^n & \\ \ddots & & s_j^n & & \ddots \\ & r_j^n & & \ddots & \\ 0 & & \ddots & & \ddots \end{pmatrix}$$

$$r_j^n = \left(-\frac{\rho_j^n}{h^2} + \frac{\mu_j^n}{2h} \right) k, \quad s_j^n = \left(-\frac{2\rho_j^n}{h^2} - b_j^n + k^{-1} \right) k, \quad t_j^n = \left(-\left(\frac{\rho_j^n}{h^2} + \frac{\mu_j^n}{2h} \right) \right) k.$$

It is easy to show that the matrix A^n has non-positive off-diagonal elements, has strictly positive diagonal elements and is irreducibly diagonally dominant. Hence (see Varga, 1962, pp. 84–85) A^n is non-singular and its inverse is non-negative; that is, all its elements are non-negative, $(A^n)^{-1} \geq 0$. This matrix is an *M-matrix*. ■

Using this result in (20.58) gives the desired result.

Lemma 20.2 Let $\{U_j^n\}$ be the solution of scheme (20.57) and suppose that:

$$\max \left| U_j^n \right| \leq m \text{ for all } j \text{ and } n$$
$$\max \left| f_j^n \right| \leq N \text{ for all } j \text{ and } n.$$

Then:

$$\max_j \left| U_j^n \right| \leq -\frac{N}{\beta} + m \text{ in } \bar{Q} \text{ where } b(x,t) \leq \beta < 0.$$

Proof: Define the discrete barrier function:

$$w_j^n = -\frac{N}{\beta} + m \pm U_j^n.$$

Then $w_j^n \geq 0$ on Γ. Furthermore, $L_k^h w_j^n \leq 0$. Hence $w_j^n \geq 0$ in $\bar{Q}$ which proves the result.

Let $u(x,t)$ and $\{U_j^n\}$ be the solutions of (20.55) and (20.57), respectively. Then:

$$\left|u(x_j,t_n) - U_j^n\right| \leq M(h+k) \text{ for } j = 0,\ldots,J, n = 0,\ldots,N \tag{20.59}$$

where M is a constant that is independent of h, k and σ. This inequality is proved in Duffy (1980).

Remark: This result shows that convergence is assured regardless of the size of σ. No classical scheme (for example, centred differencing in x and Crank–Nicolson in time) has error bounds of the form (20.59) with *M independent of h, k and* σ.

Summarising, the advantages of the fitted scheme are:

- It is uniformly stable for all values of h, k and σ.
- It is oscillation free. Its solution converges to the exact solution of (20.55). In particular, it is a powerful scheme for the Black–Scholes equation and its generalisations.
- It is easily programmed.

20.8.1 What Happens When the Volatility Goes to Zero?

The limiting case for the Crank–Nicolson scheme when the volatility goes to zero is an example of a *weakly stable scheme*. This is not a good state of affairs because rounding errors can affect the solution. In the case of the fitted scheme, however, the news is brighter.

We examine some 'extreme' cases in system (20.57). In particular, we examine the cases:

$$\begin{aligned}&\text{(pure convection/drift)} \quad \sigma \to 0\\ &\text{(pure diffusion/volatility)} \quad \mu \to 0.\end{aligned}$$

We shall see that the 'limiting' difference schemes are well known. To examine the first extreme case we must know what the limiting properties of the hyperbolic cotangent function are:

$$\lim_{\sigma\to 0} \rho_j^n = \lim_{\sigma\to 0} \frac{\mu_j^n h}{2} \coth \frac{\mu_j^n h}{2\sigma_j^n}.$$

We use the formula:

$$\lim_{\sigma\to 0} \frac{\mu h}{2} \coth \frac{\mu h}{2\sigma} = \begin{cases} +\frac{\mu h}{2} \text{ if } \mu > 0 \\ -\frac{\mu h}{2} \text{ if } \mu < 0. \end{cases}$$

Inserting this result into equation (20.57) gives us the first-order scheme:

$$\mu > 0, -\frac{U_j^{n+1} - U_j^n}{k} + \mu_j^{n+1} \frac{\left(U_{j+1}^{n+1} - U_j^{n+1}\right)}{h} + b_j^{n+1} U_j^{n+1} = f_j^{n+1}$$
$$\mu < 0, -\frac{U_j^{n+1} - U_j^n}{k} + \mu_j^{n+1} \frac{\left(U_j^{n+1} - U_{j-1}^{n+1}\right)}{h} + b_j^{n+1} U_j^{n+1} = f_j^{n+1}.$$

These are *implicit upwind schemes* and are stable and convergent. We conclude that the fitted scheme degrades to an acceptable scheme in the limit. The case $\mu \to 0$ uses the formula:

$$\lim_{x \to 0} x \coth x = 1.$$

Then the first equation in system (20.57) reduces to the equation:

$$-\frac{U_j^{n+1} - U_j^n}{k} + \sigma_j^{n+1} D_+ D_- U_j^{n+1} + b_j^{n+1} U_j^{n+1} = f_j^{n+1}.$$

This is a standard first-order (BTCS) approximation to diffusion reaction problems.

20.9 RICHARDSON EXTRAPOLATION

The BTCS (*Backward in Time Centred in Space*) scheme (20.57) is first-order accurate in time. We can upgrade the accuracy to second order by employing *Richardson extrapolation* (see Lawson and Morris, 1978). To this end, we first discuss the method for a *scalar initial value problem*:

$$\left.\begin{aligned} &Lu \equiv u'(t) + a(t)u(t) = f(t), \quad t \in [0, T] \\ &u(0) = A \\ &a(t) \geq \alpha > 0, \quad \forall t \in [0, T]. \end{aligned}\right\} \tag{20.60}$$

First, the *explicit Euler method* is given by:

$$\left.\begin{aligned} &\frac{u^{n+1} - u^n}{k} + a^n u^n = f^n, \quad n = 0, \ldots, N-1 \\ &u^0 = A \end{aligned}\right\} \tag{20.61}$$

whereas the *implicit Euler method* is given by:

$$\left.\begin{aligned} &\frac{u^{n+1} - u^n}{k} + a^{n+1} u^{n+1} = f^{n+1}, \quad n = 0, \ldots, N-1 \\ &u^0 = A. \end{aligned}\right\} \tag{20.62}$$

Notice the difference: in (20.61) the solution at level $n + 1$ can be calculated directly in terms of the solution at level n while in (20.62) we must rearrange terms in order to calculate the solution at level $n + 1$.

The next scheme is called the *Crank–Nicolson (or Box) scheme* and it can be seen as an average of explicit and implicit Euler schemes. It is given as:

$$\begin{aligned}&\frac{u^{n+1}-u^n}{k}+a^{n,\frac{1}{2}}u^{n,\frac{1}{2}}=f^{n,\frac{1}{2}},\quad n=0,\ldots,N-1\\&u^0=A \text{ where } u^{n,\frac{1}{2}}\equiv\frac{1}{2}(u^n+u^{n+1}) \text{ and } a^{n,\frac{1}{2}}=a(t_n+k/2).\end{aligned}\tag{20.63}$$

We now give an introduction to a technique that allows us to improve the accuracy of finite difference schemes. This is called *Richardson extrapolation*. We take a specific case to show the essence of the method, namely the implicit Euler method (20.62). We know that this scheme is first-order accurate and that it is unconditionally stable. We now apply this method on meshes of size k and $k/2$ and we can show that the approximate solutions u^k and $u^{k/2}$ can be represented as:

$$\begin{aligned}u^k&=u+mk+0(k^2)\\u^{k/2}&=u+m\left(\frac{k}{2}\right)+0(k^2)\end{aligned}$$

then:

$$2u^{k/2}-u^k=u+0(k^2).$$

The constant m is independent of k and this is why we can eliminate it in the first equations to produce a scheme that is second-order accurate. The same trick can be employed with the Crank–Nicolson scheme to produce a *fourth-order accurate* scheme as follows:

$$\begin{aligned}u^k&=u+mk^2+0(k^4)\\u^{k/2}&=u+m\left(\frac{k}{2}\right)^2+0(k^4).\end{aligned}$$

Then:

$$\frac{4}{3}u^{k/2}-\frac{1}{3}u^k=u+0(k^4).$$

In general, with *repeated extrapolation methods* we state the accuracy we desire. We divide the interval $[0, T]$ into smaller subintervals until the difference between the solutions on consecutive meshes is less than a given tolerance.

We now discuss Richardson extrapolation in the context of PDEs and FDM. We initially discuss the heat equation (13.14) (Chapter 13) and its semi-discretisation (13.17) in the space variable that we write in matrix form (13.18). The exact solution is given in terms of the exponential of a matrix (where we show the explicit dependence of the solution on the mesh size k):

$$\begin{aligned}&U(t)=\exp(tA)U(0)\\&U(t+k)=\exp(kA)U(t),\ t=0,k,2k,\ldots \text{ where } k \text{ is a step size.}\end{aligned}$$

In this case the time variable is still continuous. We can discretise it in time using the BTCS (implicit Euler) scheme:

$$\frac{U(t+k) - U(t)}{k} = AU(t+k)$$

or

$$(I - kA)U(t+k) = U(t)$$

or

$$U(t+k) = (I - kA)^{-1}U(t).$$

This is now a first-order approximation to the exact solution. As with the previous examples we can define two approximate solutions on meshes of size k and $2k$ to get a second-order accurate solution extrapolation algorithm:

$$\begin{aligned} U^{(1)}(t+2k) &= (I - 2kA)^{-1}U(t) \\ U^{2}(t+2k) &= (I - kA)^{-1}(I - kA)^{-1}U(t) \\ U(t+2k) &= 2 * U^{(2)} - U^{(1)}. \end{aligned}$$

Richardson extrapolation of the BTCS has a number of advantages:

- It is second-order accurate in time.
- It is an *L-stable method* (Lawson and Morris, 1978) which makes it suitable for *stiff equations* and equations with discontinuous initial conditions. The Crank–Nicolson method is only *A-stable* and it can produce oscillations. A workaround is to apply the method described in Rannacher (1984) in which the BTCS scheme is employed for the first few time steps and then Crank–Nicolson is used when initial oscillations have subsided.
- It can be parallelised; the schemes on mesh size k and $k/2$ can run in parallel.

We shall discuss the implementation of this method in Chapter 21.

20.10 SUMMARY AND CONCLUSIONS

In this chapter we have introduced one-factor PDEs and we defined how they are incorporated into an IBVP that we then approximate using the finite difference method. We have attempted to give an overview of major mathematical and numerical topics that need to be understood before designing the related algorithms and their subsequent implementation in C++. In particular, in Section 20.6 we proposed 18 attention points that you should be acquainted with when working with FDM. For a more detailed account of the finite difference method, see Duffy (2006).

20.11 EXERCISES AND PROJECTS

1. (Kinds of PDEs in Computational Finance, Initial Investigations and Small Project)

One-factor PDEs are common in computational finance, for example:

- CIR (see Cox, Ingersoll and Ross, 1985).
- CEV (see Beckers, 1980).
- Hull–White and extended Vasicek models (see Hull and White, 1996).

Answer the following questions:

a) Determine the coefficients of the PDEs in these cases (see equation (20.13)).

b) Can you change the independent variable to simplify the PDE in some way (see Section 20.3)? What is the range of values that the underlying variable takes before and after a change of variable?

c) PDEs are usually defined on infinite or semi-infinite intervals. Discuss the use of domain truncation and domain transformation to reduce these PDEs to a bounded domain. Which transformations in Section 20.3.4 would be good candidates?

d) Based on part c), determine which boundary conditions to apply based on heuristic reasoning, Fichera theory or Feller theory. In particular, which boundaries need boundary conditions and which boundaries do not?

2. (Quiz: Crank Nicolson Method)

Which of the following statements is true regarding the Crank–Nicolson method?

a) It is second-order accurate in time.

b) It uses second-order divided differences in space.

c) It is a *monotone* scheme.

d) It leads to a tridiagonal matrix system when applied to equation (20.13).

3. (Choice of Domain Transformer)

In this exercise we discuss the applicability of domain transformations similar to that in equation (20.24), namely:

$$y = \tanh \alpha x \text{ (Case 1) and } y = \frac{1}{1+\alpha x} \text{(Case 2).}$$

a) Verify the following:

$$x = \frac{1}{2\alpha} \log\left(\frac{1+y}{1-y}\right) = \log\left(\frac{1+y}{1-y}\right)^{1/2\alpha}$$

$$\frac{dy}{dx} = \alpha(1-y^2)$$

$$\frac{\partial u}{\partial x} = \alpha(1-y^2)\frac{\partial u}{\partial y}$$

$$\frac{\partial^2 u}{\partial x^2} = \alpha^2(1-y^2)\frac{\partial}{\partial y}\left(\left(1-y^2\right)\frac{\partial u}{\partial y}\right)$$

for Case 1 and

$$x = \frac{1-y}{\alpha y},\ \frac{dy}{dx} = -\alpha y^2$$

$$\frac{\partial u}{\partial x} = -\alpha y^2 \frac{\partial u}{\partial y}$$

$$\frac{\partial^2 u}{\partial x^2} = \alpha^2 y^2 \frac{\partial}{\partial y}\left(y^2 \frac{\partial u}{\partial y}\right)$$

for Case 2. In the latter case we note that the mapping of the semi-interval is to the interval (1, 0), that is the end points are crossed! Case 2 is popular in interest-rate applications.

b) Determine the coefficients of the transformed PDE (as in equation (20.13)) when applying both of these transformations to the one-factor Black–Scholes PDE. Which transformed PDE is more amenable to the finite difference method?

4. (Domain Transformation for Infinite Intervals)

Consider the following partial differential equations for the one-dimensional *wave* and *heat equations* on an infinite interval:

$$\frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = 0, \quad -\infty < x < \infty, \quad t > 0,$$
$$\frac{\partial u}{\partial t} = a^2\frac{\partial^2 u}{\partial x^2}, \quad -\infty < x < \infty, \quad t > 0.$$

Answer the following questions:

a) Transform these PDEs to PDEs on the interval $[-1,\ 1]$ by using the transformations:

$$y = \tanh \alpha x$$
$$y = \coth \alpha x.$$

Compare the forms of the PDEs resulting from these transformations.

b) Compute the Fichera function for each PDE in part a) of this exercise. What are the resulting boundary conditions?

c) Apply ADE to each of the transformed PDEs. Compare the approach with the Crank–Nicolson method.

5. (Solving Two-Point Boundary Value Problems, Non-trivial Project)

We examine the example in Section 20.7.1 as preparation for later chapters. We focus attention on the TPBVP defined by system (20.50) and we analyse it from various perspectives. Answer the following questions:

a) Approximate system (20.50) using a centred difference scheme on a constant mesh. The scheme's code should be flexible enough so as to accommodate the exponential fitting method in scheme (20.51) and the classical finite difference scheme with a fitting factor equal to 1 (you may need to use a *mixin method* or a function object to achieve this level of flexibility).

b) Set up the tridiagonal system of equations from part a) and solve it using the Thomas and Double Sweep algorithms. Do you get the same answers in each case?

c) Prove that the matrix corresponding to the fitting scheme is *monotone* for all values of the diffusion and convection terms. In the case of the classical finite difference schemes with centred differencing in space, what are the constraints on the constant mesh size in order for the matrix to be monotone?

d) Consider the case of small diffusion and/or large convection for both schemes. Do you get correct answers?

e) We employ *Richardson extrapolation* on the classical finite difference scheme to improve accuracy from second order to fourth order (Keller, 1968). To this end, let

$u_{2j}(h/2)$ and $u_j(h),\ j = 0, \ldots, J$ be the finite difference solutions on the meshes $h/2$ and h, respectively. Now construct the array:

$$\bar{u}_j = \frac{4}{3}u_{2j}(h/2) - \frac{1}{3}u_j(h)\ 0 \le j \le J.$$

Verify by numerical experiment that it is a fourth-order approximation to the exact solution.

f) Use first-order upwinding as in equation (20.47). Is the resulting matrix monotone? Can you reproduce the numerical results for this scheme as reported in Section 20.7.1?

g) Convert system (20.50) to conservative form and approximate it using the finite difference method.

h) We now return to Chapter 13 by solving the heat equation using Rothe's method. The *semi-discrete system* of ODEs will be a special case of the following equation that uses implicit Euler in time:

$$-\frac{u^{n+1} - u^n}{\Delta t} + \sigma(x, t_{n+1})\frac{d^2u^{n+1}}{dx^2} + \mu(x, t_{n+1})\frac{du^{n+1}}{dx} + b(x, t_{n+1})u^{n+1} = f(x, t_{n+1}),$$
$$0 \le n \le N - 1.$$

Implement the scheme in C++ using explicit Euler (for convenience) as the approximation to the time derivative. The focus is on creating an algorithm that can easily be mapped to C++. The algorithm will need to implement the time-marching aspect of the problem.

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 21

Software Framework for One-Factor Option Models

21.1 INTRODUCTION AND OBJECTIVES

In this chapter we concentrate on implementing the algorithms that describe the finite difference schemes from Chapter 20. We use a combination of styles that C++ has to offer to show what is possible. Chapter 20 deals with the mathematics of the problem and associated numerical methods, whereas this chapter focuses on how to map the algorithms to C++. There is thus a clear *separation of concerns*. More precisely, the goals in this chapter are:

- To create a seamless mapping from FD algorithms to C++, thus helping us to create code that is built to be maintained and extended. We achieve this end by using system decomposition methods and design patterns.
- To experiment with a range of FD schemes (some of which are new in finance) and determine how stable and accurate they are.
- Having determined that a scheme is to our liking, to optimise the code by modifying the design or the code in some way in order to improve the performance.

We discuss a number of design and coding styles to satisfy the first two goals. We then measure the run-time performance of the code to determine whether it satisfies the third goal.

The skills developed in this chapter can be applied to a range of applications. We make this claim based on how we designed problems in Chapter 9. The key issues are to decide what the initial high-level architecture is going to be and how we design classes, class hierarchies and polymorphic functions.

In Chapters 22 and 23 we discuss how to price other kinds of one-factor and two-factor options problems as well as extending the software to suit new requirements.

21.2 A SOFTWARE FRAMEWORK: ARCHITECTURE AND CONTEXT

We examine *linear convection–diffusion–reaction PDEs* of non-conservative type and their approximation by finite difference schemes. The initial system layout is shown in Figure 21.1

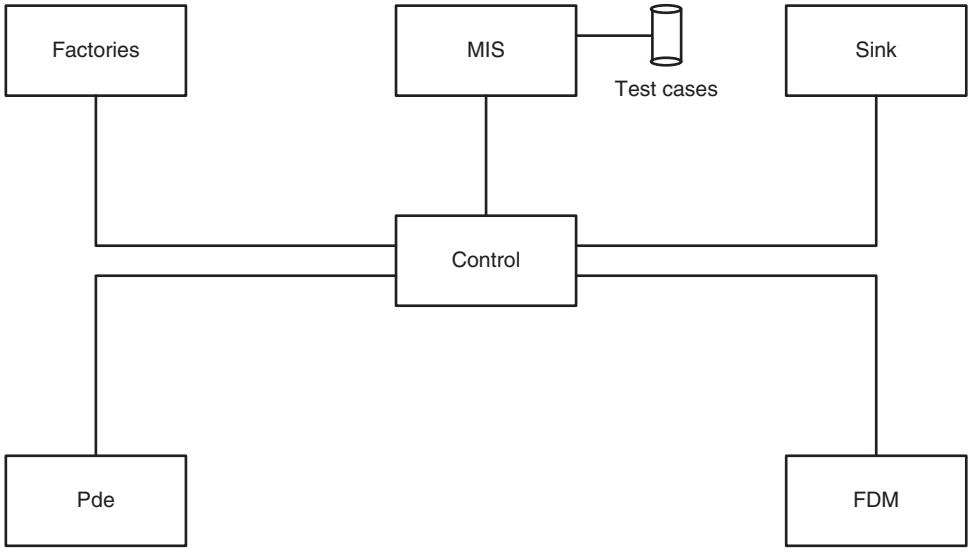


FIGURE 21.1 Simple block diagram of architecture

and it is based on the design philosophy that we introduced in Chapter 9. Each block will contain classes having certain responsibilities.

- Pde: the C++ classes that model the PDEs introduced in Chapter 20.
- FDM: classes in a class hierarchy that approximate the solution of PDEs. We review some FD schemes in Section 21.4.
- Sink: the system to which the results from the FD solver are sent.
- MIS: a system containing information that can be used when comparing the accuracy and efficiency of a candidate FD scheme with some target/baseline scheme; for example, the exact solution or another FD scheme. Optionally, this system could contain a database (repository) of test cases and solutions.
- Factories: dedicated classes that are responsible for the creation of instances of PDE and FDM classes.
- Control: the central class or function that mediates between the other classes in Figure 21.1. It could be a dedicated *mediator* class or it could be hard-coded in `main()`. In Chapter 23 we shall design a software framework for path-dependent PDEs based on the functional programming model and interface programming.

21.3 MODELLING PDEs AND FINITE DIFFERENCE SCHEMES: WHAT IS SUPPORTED?

We implement a number of *one-step schemes* in this chapter; they all share the property that they compute a solution at time level $n + 1$ in terms of the solution at time level n. Furthermore, some schemes are implicit (for example, S2, S4 and S5 below) which demands our having to solve a *tridiagonal* system at each time level (using the two matrix solvers from Chapter 13)

while others are explicit (S1, S3) and then the solution at time level $n+1$ is computed directly in terms of the solution at time level n without the need to solve a matrix system. The schemes are as follows.

- S1: Explicit Euler (FTCS) scheme.
- S2: Fully implicit (BTCS) scheme.
- S3: ADE (several variants based on how we approximate the diffusion, convection and reaction terms in the PDE).
- S4: Crank–Nicolson.
- S5: Richardson extrapolation applied to scheme S2 (discussed in Chapter 20, Section 20.9).

The schemes are applicable to a wide range of linear convection–diffusion–reaction PDEs of non-conservative type and our interest lies in pricing plain options.

We map semi-infinite domains to bounded ones, as discussed in Chapter 20. We have chosen *domain truncation* in this chapter because this technique does not affect the form of the PDE being modelled and it allows us to focus on the details of the software design. It is almost as easy to use as *domain transformation*. See Exercise 1.

21.4 SEVERAL VERSIONS OF ALTERNATING DIRECTION EXPLICIT

The Alternating Direction Explicit (ADE) method was originally announced in Saul'yev (1964) and first applied to computational finance in Duffy (2009). Since then, a number of results in this area have been published, for example Pealat and Duffy (2011), Duffy and Germani (2013) and Buchova, Ehrhardt and Guenther (2015) and a number of the author's MSc students.

We note that ADE is a family of schemes; there are several variants based on how the diffusion, convection and reaction terms are approximated. We consider each option in turn. First, let us examine the diffusion term by taking the *heat equation* and its ADE variants (we are assuming Dirichlet boundary conditions).

- Saul'yev (1964): we use a single storage array. At odd time levels we use the *forward sweep*:

$$\frac{u_j^{n+1}-u_j^n}{k}=\frac{u_{j-1}^{n+1}-u_j^{n+1}-u_j^n+u_{j+1}^n}{h^2} \tag{21.1}$$

while at even time levels we use the *backward sweep*:

$$\frac{u_j^{n+2}-u_j^{n+1}}{k}=\frac{u_{j-1}^{n+1}-u_j^{n+1}-u_j^{n+2}+u_{j+1}^{n+2}}{h^2}. \tag{21.2}$$

This method is unconditionally stable, has truncation error $0\left[k^2,h^2,(k/h)^2\right]$ and hence is only first-order accurate because of the inconsistent term $(k/h)^2$. We note that the schemes *sweep* the solution from left to right and from right to left, respectively.

- Barakat and Clark (B&C) (1966): in this case the scheme sweeps in both directions simultaneously and then the resulting solutions are averaged:

$$\frac{U_j^{n+1} - U_j^n}{k} = \frac{1}{h^2}\left(U_{j+1}^n - U_j^n - U_j^{n+1} + U_{j-1}^{n+1}\right), 1 \le j \le J-1, n \ge 0 \quad (21.3)$$

$$\frac{V_j^{n+1} - V_j^n}{k} = \frac{1}{h^2}\left(V_{j+1}^{n+1} - V_j^{n+1} - V_j^n + V_{j-1}^n\right), J-1 \ge j \ge 1, n \ge 0 \quad (21.4)$$

$$u_j^{n+1} = \frac{1}{2}\left(U_j^{n+1} + V_j^{n+1}\right), 0 \le j \le J, n \ge 0. \quad (21.5)$$

The truncation error is $0\left[k^2, h^2\right]$ because the terms from the sweeps in $(k/h)^2$ cancel. Hence, the method is second-order accurate. A detailed analysis can be found in Buchova, Ehrhardt and Guenther (2015).

- Larkin (1966): this scheme is similar to the B&C scheme:

$$\frac{U_j^{n+1} - u_j^n}{k} = \frac{U_{j-1}^{n+1} - U_j^{n+1} - u_j^n + u_{j+1}^n}{h^2} \quad (21.6)$$

$$\frac{V_j^{n+1} - u_j^n}{k} = \frac{U_{j-1}^n - u_j^n - V_j^{n+1} + V_{j+1}^{n+1}}{h^2} \quad (21.7)$$

$$u_j^{n+1} = \frac{1}{2}\left(U_j^{n+1} + V_j^{n+1}\right), 0 \le j \le J, n \ge 0. \quad (21.8)$$

There is anecdotal evidence to suggest that this method is less accurate than the B&C scheme (Tannehill, Anderson and Pletcher, 1997). We have also noticed a loss in accuracy when using this scheme.

In order to approximate the convection term we refer the reader to Pealat and Duffy (2011) and Duffy and Germani (2013).

It is interesting to note that ADE predates both Alternating Direction Implicit (ADI) and locally one-dimensional (LOD) ('Soviet Splitting') methods. The first application of LOD to computational finance was seemingly due to joint work by Alex Levin and the author. Sheppard (2007) discusses the application of LOD to the Heston model. A *stability analysis* of ADE for the convection–diffusion equation can be found in Campbell and Yin (2006).

21.4.1 Spatial Amplification and ADE

The B&C ADE method is unconditionally stable and second-order accurate. However, the von Neumann method does not pick up *spatial amplification errors* that can be caused by machine round-off, for example (see Campbell and Yin, 2006; Roache, 1998). Any error in a boundary condition at the beginning of a left or right computational sweep may be amplified as the computation proceeds forward in space (not as the time proceeds).

We take an example of pricing a put option (we will explain the full context in later sections):

```
Option myOption;
myOption.sig = 0.3; myOption.K = 65.0; myOption.T = 0.25;
myOption.r = 0.08; myOption.b = 0.08; myOption.beta = 1.0;
myOption.SMax = 325.0; myOption.type = 'P';
myOption.earlyExercise = false;
```

for fixed NT = 325, NX = {325, 650, 1300, 2600, 5200, 10400}. The put values get progressively worse as we increase NX while fixing NT, namely the values become {5.841754, 5.844434, 5.84294, 5.834118, 5.801135, 5.72656}. We should be aware of this somewhat pathological behaviour. It is like a law of gravity for this class of finite difference schemes. We have not resolved this issue. A rule-of-thumb upper limit is NX $\sim$ 4*NT. On the other hand, keeping NX fixed and increasing NT does not lead to amplification errors.

An interesting exercise is to determine whether the original Saul'yev scheme causes spatial amplification errors.

21.5 A SOFTWARE FRAMEWORK: DETAILED DESIGN AND IMPLEMENTATION

Having agreed in global terms on what the architectural design will be (the context diagram in Figure 21.1) we now need to decide on a detailed class design and the level of flexibility that the code should realise. There are several alternatives:

- A1: Procedural solution; implement the modules in Figure 21.1 as free functions or as non-polymorphic classes.
- A2: Traditional class hierarchy with inheritance and subtype (dynamic) polymorphism.
- A3: Model class hierarchies with curiously recurring template pattern (CRTP) and static (compile-time) polymorphism.
- A4: Instead of a class hierarchy (as with choices A2 and A3) we create a single class consisting of universal function wrappers (instances of `std::function`). This is the approach that we shall adopt in Chapter 23.
- A5: Using policy-based design (PBD) and duck typing as discussed in Chapter 9.
- A6: A combination of choices A1–A5. In other words, we choose the approach that we think is most suitable for each of the modules in Figure 21.1.

In this chapter we implement options A2 (for the PDE classes and several FDM classes) and A3 (for the classes that implement the ADE variants) while we use option A1 (essentially the `main()` function) to configure the application. The UML diagram is shown in Figure 21.2. Its design has a number of desirable features:

- Each class has a single major responsibility.
- Separation of concerns: each FDM class is a client of a generic convection–diffusion reaction PDE. This promotes maintainability.

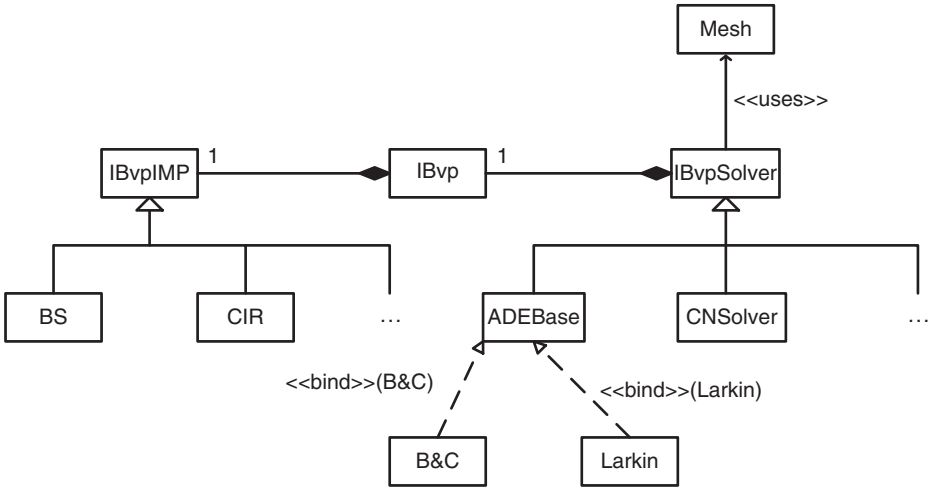


FIGURE 21.2 UML structure of framework

- All PDE and FDM classes have polymorphic interfaces and they can be extended by using a number of design patterns (namely *Bridge*, *Template Method* and *Factory Method*; see GOF, 1995).

We now discuss the classes in Figure 21.2 in more detail.

21.6 C++ CODE FOR PDE CLASSES

It is obvious that the software framework should be as flexible and customisable as possible if we wish it to be used. For example, we write algorithms that apply to a range of PDE types. To this end, we focus on linear convection–diffusion–reaction equations in non-conservative form and their numerical approximation as discussed in Chapter 20. This translates into a clear requirement to apply the *Bridge* design pattern (see GOF, 1995). Its intent is to:

- Decouple an abstraction from its implementations so that the two can vary independently.
- Publish interfaces in an inheritance hierarchy, and place implementation classes in an inheritance hierarchy.

In this case we identify the class `IBvp` that is the abstract interface which clients (for example, FDM classes) use. `IBvp` also contains a reference to an instance of `IBvpImp` that is the base class for all specific PDE implementations in the *Bridge* pattern. The interfaces of these two classes are:

```
class IBvp
{ // Models linear convection diffusion reaction PDE with associated
  // boundary and initial conditions
private:
public:
    Range<double> xaxis;           // Space interval
    Range<double> taxis;           // Time interval
```

```
    // Bridge pattern: IBvp is the 'abstraction' object that is decoupled
    // from its various implementations.
    IBvpImp* imp;                  // Bridge implementation

public:

    IBvp() = delete;

public:
    IBvp (IBvpImp& executor, const Range<double>& xrange,
          const Range<double>& trange);

    // Coefficients of parabolic second order operator

    // Coefficient of second derivative
    double Diffusion(double x, double t) const;

    // Coefficient of first derivative
    double Convection(double x, double t) const;

    // Coefficient of zero derivative
    double Reaction(double x, double t) const;

    // Inhomogeneous forcing term
    double Rhs(double x, double t) const;

    // Boundary and initial conditions

    // Left hand boundary condition
    double Bcl(double t) const;

    // Right hand boundary condition
    double Bcr(double t) const;

    // Initial condition
    double Ic(double x) const;

    // The domain in which the PDE is 'played'
    const Range<double>& xrange() const;
    const Range<double>& trange() const;
};

class IBvpImp
{ // Abstract base class modelling the implementation objects in the
    // Bridge pattern. Derived classes (e.g. BS, CEV, CIR) must implement
    // all the pure virtual functions in IBvpImp.

public:

    // Selector functions
    // Coefficients of parabolic second order operator
    virtual double Diffusion(double x, double t) const = 0;
```

```
    virtual double Convection(double x, double t) const = 0;
    virtual double Reaction(double x, double t) const = 0;
    virtual double Rhs(double x, double t) const = 0;

    // Boundary and initial conditions
    virtual double Bcl(double t) const = 0;
    virtual double Bcr(double t) const = 0;
    virtual double Ic(double x) const = 0;

};
```

We note that `IBvp` also has two members (instances of `Range`) representing the rectangular domain in which the PDE is defined. Furthermore, Dirichlet boundary conditions are supported. We use raw pointers in this version of the software.

We are now in a position to define specific PDE classes. We take the one-factor Black–Scholes equation as an initial example:

```
class BlackScholesPde : public IBvpImp
{
public:

    Option opt;

    BlackScholesPde(const Option& option);

    double Diffusion(double x, double t) const;
    double Convection(double x, double t) const;
    double Reaction(double x, double t) const;
    double Rhs(double x, double t) const;

    // Boundary and initial conditions
    double Bcl(double t) const;
    double Bcr(double t) const;
    double Ic(double x) const;
};
```

We also give the code of the member functions for completeness:

```
BlackScholesPde::BlackScholesPde(const Option& option) : opt(option)
{
}

double BlackScholesPde::Diffusion(double x, double t) const // Diffusion
{

    double v = opt.sig;
    return 0.5*v*v*x*x;
}

double BlackScholesPde::Convection(double x, double t) const // Drift
```

```
{
    return (opt.b) * x; // r == interest rate
}

double BlackScholesPde::Reaction(double x, double t) const // Zero
                                                           // derivative
{
    return - (opt.r);

}

double BlackScholesPde::Rhs(double x, double t) const  // Inhomogeneous
                                                       // term
{
    return 0.0;
}

// Boundary and initial conditions
double BlackScholesPde::Bcl(double t) const // Left hand boundary
                                            // condition
{

    if (opt.type == 'C')
    {
        return 0.0;    //C
    }
    else
    {
        return (opt.K) * std::exp(- (opt.r) * ((opt.T) - t)); // P
    }

}

double BlackScholesPde::Bcr(double t) const    // Right boundary condition
{
    if (opt.type == 'C')
    {
        return opt.SMax - (opt.K) * std::exp(- (opt.r) * ((opt.T) - t));


    }
    else
    {
        return 0.0; //P
    }
}

double BlackScholesPde::Ic(double x) const        // Initial condition
{
    if (opt.type == 'C')
    {
```

```
        return std::max<double>(x - opt.K, 0.0);

    }
    else
    {
        return std::max<double>(opt.K - x, 0.0);
    }

}
```

We note that we have used hard-coded if…else logic to distinguish between call and put options. This can be made more flexible. We have also created a simple structure to hold relevant data:

```
class Option : public Instrument
{
public:
    // PUBLIC, FOR CONVENIENCE
    double r;               // Interest rate
    double sig;             // Volatility
    double K;               // Strike price
    double T;               // Expiry date
    double b;               // Cost of carry
    double beta;            // Elasticity factor
    double SMax;            // Far field condition
    char type;              // Call or put
    bool earlyExercise;     // American option

Option()
{

}

void print()
{

        std::cout << "Interest: " << r << std::endl;
        std::cout << "Vol: " << sig << std::endl;
        std::cout << "Strike: " << K << std::endl;
        std::cout << "Expiry: " << T << std::endl;
        std::cout << "Cost of carry: " << b << std::endl;
        std::cout << "Elasticity factor: " << beta << std::endl;
        std::cout << "Option type: " << type << std::endl;
        std::cout << "Far field: " << SMax << std::endl;
        std::cout << "American?: " << std::boolalpha << earlyExercise;

}

OneFactorPayoff OptionPayoff;
};
```

An example of use is:

```
Option myOption;
myOption.sig = 0.3; myOption.K = 65.0; myOption.T = 0.25;
myOption.r = 0.08; myOption.b = 0.08; myOption.beta = 1.0;
myOption.SMax = 325.0; myOption.type = 'P'; myOption.earlyExercise = false;

BlackScholesPde myImp(myOption);

Range<double> rangeX(0.0, myOption.SMax);
Range<double> rangeT(0.0, myOption.T);

IBvp currentImp(myImp, rangeX, rangeT);
```

Please refer to the source code provided with the book for more details.

More generally, the *Bridge* pattern is used in applications where portability is important. In these cases classes can have several implementations and we wish to shield clients from the latter's specific code. Some examples are:

- Portable graphical user interfaces (GUIs). For example, the *Qt* toolkit is a cross-platform application framework that is used for developing software that runs on various software and hardware platforms. Clients interact with graphical objects without having to know their internals or the details of the corresponding SDK (software developer kit). See Ezust and Ezust (2006) for a discussion on design patterns in the context of *Qt*.
- In embedded and hardware systems. Hardware and software change and we wish to shield clients from changes when systems upgrade to new environments. Clients communicate with a *virtual machine* that can have multiple implementations. A good example is a sensor abstraction that specifies a certain abstract interface. Commercial sensor drivers can then be *adapted* to conform to these abstract interfaces.
- There are many opportunities to apply the *Bridge* pattern in computational finance. In this section, we have seen its application in the context of convection–diffusion–reaction PDEs. We define a PDE abstraction that clients communicate with. Clients define implementations to suit new requirements without the need to modify other parts of the code. Another example is to define an abstraction for a one-factor stochastic differential equation (SDE). Then clients create implementations corresponding to well-known SDEs such as geometric Brownian motion (GBM), for example.
- We have seen in this section that a PDE can be approximated by many FD schemes while a given FD scheme can be applied to many PDEs. This is possible mainly due to the *Bridge* pattern.

In Exercise 5 we discuss a number of alternative implementations of the *Bridge* pattern.

21.7 C++ CODE FOR FDM CLASSES

The classes that we use to model finite difference algorithms are shown in Figure 21.2 and they implement the schemes from Chapter 20. This part of the framework uses a mixture of programming styles.

- `IBVpSolver`: the base class for all finite difference schemes. In a sense it is a central mediator because it creates the meshes in space and time and it contains the state machine logic to march the approximate solution to expiration. The result is a vector of option prices at discrete mesh points. Option values at non-mesh points can be produced by linear or cubic spline interpolation (as discussed in Chapter 13).
- Classes to implement explicit Euler (FTCS), fully implicit (BTCS), Crank–Nicolson and Richardson extrapolation schemes. We use subtype polymorphism to vary behaviour.
- Variants of the ADE scheme using the CRTP.

The interface of `IBvpSolver` is:

```
using Vector = std::vector<double>;

class IBvpSolver
{ // Base class for a Set of finite difference schemes
  // to solve scalar initial boundary value problems

private:

    // Utility functions
    void initMesh(long NSteps, long JSteps);
    void initIC();

protected:

    IBvp* ibvp;      // Pointer to 'server'

    long N;          // The number of subdivisions of interval in IBVP
    double k;        // Step length; redundant data but is efficient

    long J;          // The number of subdivisions of interval in IBVP
    double h, h2;    // Step length; redundant data but is efficient

    double DN;
    double DJ;
    double DJJ;

    double tprev, tnow, T;
    long currentIndex, maxIndex, tIndex;

public:
    Vector xarr; // Useful work array
    Vector tarr; // Useful work array

    // Other data
    long n;  // Current counter
    Vector vecOld;
    Vector vecNew;

public:
    IBvpSolver(const IBvpSolver& source) = delete;
```

```
    IBvpSolver& operator = (const IBvpSolver& source) = delete;

public:
    IBvpSolver() {}
    IBvpSolver(IBvp& source, long NSteps, long JSteps);
    virtual ~IBvpSolver();


    virtual Vector& result();       // The result of the calculation

    const Vector& XValues() const;         // Array of x values
    const Vector& TValues() const;         // Array of time values

    // Hook functions for Template Method pattern
    virtual void calculate() = 0;          // Tells how to calculate sol.
                                           // at n+1

};
```

The member functions in `IBvpSolver` have the following body:

```
void IBvpSolver::initMesh(long NSteps, long JSteps)
{ // Utility function to initialise the discrete meshes

    N = NSteps;
    J = JSteps;

    T = ibvp->trange().spread();
    k = T/ static_cast<double>(N);

    h = ibvp -> xrange().spread()/ static_cast<double>(J);

    h2 = h*h;

    // Other numbers
    DN = static_cast<double>(N);
    DJ = static_cast<double>(J);
    DJJ = static_cast<double>(J*J);

    xarr = ibvp->xrange().mesh(J);

    // Array in t direction
    tarr = ibvp->trange().mesh(N);
    tIndex = 0;

    vecOld = Vector (xarr.size(), 0.0);
    vecNew = Vector (xarr.size(), 0.0);

}
```

```
void IBvpSolver::initIC()
{ // Utility function to initialise the payoff function

    // Initialise at the boundaries
    vecOld[ 0 ] = ibvp->Bcl( ibvp->trange().low() );
    vecOld[ vecOld.size()-1 ] = ibvp->Bcr( ibvp->trange().high() );

    // Initialise values in interior of interval using
    // the initial function 'IC' from the PDE
    for( std::size_t j = 1; j < xarr.size() - 1; ++j)
    {
        vecOld[ j ] = ibvp->Ic( xarr[ j ]);
    }
}

IBvpSolver::IBvpSolver(IBvp& source, long NSteps, long JSteps)
{
    ibvp = &source;

    // Create a mesh
    initMesh(NSteps, JSteps);

    // Initialise from the payoff function
    initIC();

}

IBvpSolver::~IBvpSolver(){}

Vector& IBvpSolver::result()
{
    // Initialise time.

    // The state machine, really; we march from t = 0 to t = T.
    for (std::size_t n = 1; n < tarr.size(); ++n)
    {
        tnow = tarr[n];

        // The two methods that represent the variant parts
        // of the Template Method Pattern.
        calculate();

        tprev = tnow;

        for (std::size_t j = 0; j < vecNew.size(); ++j)
        { // Combine in previous loop

                vecOld[j] = vecNew[j];
        }
    }
```

```
    return vecNew;
}

const Vector& IBvpSolver::XValues() const
{ // Array of x values

    return xarr;
}

const Vector& IBvpSolver::TValues() const
{ // Array of time values

    return tarr;
}
```

We see that the computation takes place in `result()`; incidentally, we see an example of the *Template Method Pattern* (GOF, 1995) in which case an algorithm consists of an *invariant part* (common to all derived classes), in this case the code in `result()` and a *variant part* `calculate()` that each derived class of `IBvpSolver` must implement in its own way.

The use of this design pattern has consequences when we create derived classes, in particular the fact that *implementation inheritance* is used. We now describe specific FD classes.

21.7.1 Classes Based on Subtype Polymorphism

The classes in this group are one-step explicit and implicit finite difference methods. We focus on the Crank–Nicolson method because it is a stable second-order scheme and it is very popular despite some quirks (see Duffy, 2004A for an analysis of the method). The scheme results in a tridiagonal matrix system whose non-zero elements are stored as arrays `A`, `B` and `C`:

```
class CNIBVP : public IBvpSolver
{
private:

    // Notice that we store the data that 'belongs' to
    // this class. It is private and will not pollute the
    // other classes.
    Vector A, B, C;         // Lower, diagonal, upper
    Vector F;               // Right-hand side of matrix

public:
    CNIBVP();
    CNIBVP(IBvp& source, long NSteps, long JSteps);

    void calculate();

};
```

The body is:

```
CNIBVP::CNIBVP(): IBvpSolver(), A(Vector(J + 1)), B(Vector(J + 1)),
                  C(Vector(J + 1)), F(Vector(J + 1)){}

CNIBVP::CNIBVP(IBvp& source, long NSteps, long JSteps)
        : IBvpSolver(source, NSteps, JSteps),
         A(Vector(J + 1)), B(Vector(J + 1)), C(Vector(J + 1)),
         F(Vector(J + 1)){}

void CNIBVP::calculate()
{ // Tells how to calculate sol. at n+1

    // In general we need to solve a tridiagonal system

    double t1, t2, t3, Low, Mid, Upp;

    for (std::size_t i = 0; i < F.size(); i++)
    {
        t1 = (0.5*k*ibvp->Diffusion(xarr[i],tnow));
        t2 = 0.25*k * h* ibvp->Convection(xarr[i], tnow);
        t3 = 0.5*k*h2*ibvp->Reaction(xarr[i],tnow);

        // Coefficients of the U terms
        A[i] = t1 - t2;
        B[i] = -h2 - 2.0*t1 + t3;
        C[i] = t1 + t2;

        // Coefficients of the U terms
        double t1A = 0.5*k*ibvp->Diffusion(xarr[i],tnow );
        double t2A =
              0.25 * k * h* ibvp->Convection(xarr[i], tprev);
        double t3A = 0.5*k*h2*ibvp->Reaction(xarr[i],tprev);

        Low = -t1A + t2A;
        Mid = -h2 + 2.0*t1A - t3A;
        Upp = -t1A - t2A;

        F[i] = Low*vecOld[i-1] + Mid*vecOld[i] + Upp*vecOld[i+1]
              +0.5*k*h2*ibvp -> Rhs(xarr[i], tnow)
              +0.5*k*h2*ibvp -> Rhs(xarr[i], tprev);
    }

    double BCL = ibvp->Bcl(tnow);
    double BCR = ibvp->Bcr(tnow);
    DoubleSweep<double> mySolver(A, B, C, F, BCL, BCR);

    // The matrix must be diagonally dominant; we call the
    // assert macro and the program stops
    // assert (mySolver.diagonallyDominant() == true);

    vecNew = mySolver.solve();
}
```

We have used the *Double Sweep* method to solve the system of equations at each time level. We introduced this method in Chapter 13 and we reported that it was 20% faster than the Thomas algorithm. A nice exercise would be to use the latter method in the above code.

We take an example of using the Crank–Nicolson method:

```
long J = 500;
long N = 500;
std::cout << "Number of space divisions: ";
std::cin >> J;

std::cout << "Number of time divisions: ";
std::cin >> N;

CNIBVP fdmCN(currentImp, N, J);
```

A possible optimisation is that the Double Sweep method only needs to be called once if the coefficients of the problem are time independent.

21.7.2 Classes Based on CRTP

We model all the ADE variants as derived classes of `IBvpSolver`. The interface and code body of the base class for all ADE schemes are:

```
template <typename D>
    class ADE_CRTP: public IBvpSolver
{
protected:

    // Derived classes contain the necessary data structures

public:
    ADE_CRTP();
    ADE_CRTP(IBvp& source, long NSteps, long JSteps);

    // Hook function for Template Method pattern
    void calculate();
};

#include "ADESolver_CRTP.cpp"

template <typename D>
    ADE_CRTP<D>::ADE_CRTP(): IBvpSolver()
{
}

template <typename D>
    ADE_CRTP<D>::ADE_CRTP(IBvp& source, long NSteps, long JSteps)
       : IBvpSolver(source, NSteps, JSteps){ }
```

```
template <typename D>
    void ADE_CRTP<D>::calculate()
{ // Tells how to calculate sol. at n+1, Explicit ADE schemes

    static_cast<D*>(this) -> calculate();
}
```

The most interesting function is `calculate()` because it forwards the request to its template parameter. We discuss the derived class that implements the Barakat and Clark scheme for the diffusion term and the (derived class) Towler–Yang scheme for the convection term:

```
class ADE_BC_CRTP: public ADE_CRTP<ADE_BC_CRTP>
{
private:

    // Intermediate values
    Vector U;
    Vector V;
    Vector UOld;
    Vector VOld;

public:
    ADE_BC_CRTP(IBvp& source, long NSteps, long JSteps);

    // Hook function for Template Method pattern
    void calculate();
};

#include "ADESolver_BC_CRTP.cpp"
```

The corresponding code is based on the algorithms in Chapter 20:

```
ADE_BC_CRTP::ADE_BC_CRTP(IBvp& source, long NSteps, long JSteps)
       : ADE_CRTP<ADE_BC_CRTP>(source, NSteps, JSteps),
       U(Vector(vecNew)),V(Vector(vecNew)), UOld(Vector(vecOld)),
       VOld(Vector(vecOld)){ }

// Sequential version
void ADE_BC_CRTP::calculate()
{ // Tells how to calculate sol. at n+1, Explicit ADE_BC_CRTP schemes

       U[0] = ibvp->Bcl(tnow);
       U[U.size()-1] = ibvp->Bcr(tnow);

       // Necessary?
       V[0] = ibvp->Bcl(tnow);
       V[V.size()-1] = ibvp->Bcr(tnow);
```

```
        //std::cout << tnow << std::endl;
        for (std::size_t j = 1; j < U.size()-1; ++j)
        {
              UOld[j] = VOld[j] = vecOld[j];
        }

        double t1, t2, t3, t4;
        double H = h/2.0;

        // Upward sweep
        for (std::size_t j = 1; j < U.size()-1; ++j)
        {

           // Towler-Yang
           //t1 = k* fitting_factor(xarr[j], tnow)/ h2;
           t1 = k* ibvp->Diffusion(xarr[j], tnow) / h2;
           t2 = (0.5 * k * (ibvp->Convection(xarr[j], tnow))) / h;
           t3 = ( 1.0 + t1 - ibvp->Reaction(xarr[j],tnow) * k );
           t4 = -k * ibvp->Rhs(xarr[j],tnow);
           U[j] = ((t1 - t2)*U[j-1] + (1.0 - t1)*UOld[j]
                             + (t1 + t2)*UOld[j+1] + t4) / t3;

        }

        // Downward sweep
        for (std::size_t j = V.size()-2; j >= 1; --j)
        {

            // Towler-Yang
            t1 = k* ibvp->Diffusion(xarr[j], tnow) / h2;
            t2 = (0.5 * k * (ibvp->Convection(xarr[j],tnow)))/ h;
            t3 = ( 1.0 + t1 - ibvp->Reaction(xarr[j],tnow) * k );
            t4 = -k * ibvp->Rhs(xarr[j],tnow);

            V[j] = ((t1 - t2)*VOld[j-1] + (1.0 - t1)*VOld[j]
                             + (t1 + t2)*V[j+1] + t4) / t3;
        }

        for (std::size_t j = 0; j < vecNew.size(); ++j)
        { // Combine in previous loop

            vecNew[j] = 0.5 * (U[j] + V[j]);
        }
}
```

Finally, an example of use is:

```
ADE_BC_CRTP fdmADEBC(currentImp, N, J);
auto vADEBC = OptionPrice(fdmADEBC);
```

where we have created a utility function:

```
std::vector<double> OptionPrice(IBvpSolver& fdm)
{ // Compute option price using a FD scheme

        return fdm.result();
}
```

This function produces the array of option prices at expiration for each discrete mesh point.

21.7.3 Assembling FD Schemes from Simpler Schemes

It is possible to upgrade the accuracy of the fully implicit scheme (BTCS) from first order to second order in time (Lawson and Morris, 1978). We define a derived class of `IBVpSolver` and we override its member function `result()`:

```
class ExtrapolatedImplicitEulerIBVP : public IBvpSolver
{
private:
    // We must build these in constructors

    // Notice that we store the data that 'belongs' to
    // this class. It is private and will not pollute the
    // other classes.
    Vector A, B, C;  // Lower, diagonal, upper
    Vector F;        // Right-hand side of matrix

    // Vectors on mesh of size k/2, i.e. extrapolated values
    Vector vecNewE;  // Computed 'real' value time level n+1 (new)
    Vector vecOldE;  // Computed 'real' value at time level n (old)

    void initICExtrapolated();

public:
    ExtrapolatedImplicitEulerIBVP(IBvp& source, long NSteps, long JSteps);

    // The overridden functions in the Template Method pattern
    void calculate() override;

    Vector& result() override;

};

void ExtrapolatedImplicitEulerIBVP::initICExtrapolated()
{ // Initialise the extrapolated array at t = 0.

    // Initialise at the boundaries
    vecOldE[ 0 ] = ibvp->Bcl( ibvp->trange().low() );
    vecOldE[ vecOldE.size()-1 ] = ibvp->Bcr( ibvp->trange().high() );
```

```
    // Now initialise values in interior of interval using
    // the initial function 'IC' from the PDE
    for(std::size_t j = 1; j < xarr.size() - 1; ++j )
    {
       vecOldE[j] = ibvp->Ic( xarr[ j ]);
    }
}

ExtrapolatedImplicitEulerIBVP::ExtrapolatedImplicitEulerIBVP(IBvp& source,
                    long NSteps, long JSteps)
       : IBvpSolver(source, NSteps, JSteps),
       A(Vector(J + 1)), B(Vector(J + 1)), C(Vector(J + 1)),
       F(Vector(J + 1)),
       vecNewE(Vector(vecNew)), vecOldE(Vector(vecOld))
{
       initICExtrapolated();
}

void ExtrapolatedImplicitEulerIBVP::calculate()
{ // Tells how to calculate sol. at n+1

    ImplicitEulerIBVP i1(*ibvp, N, J);
    auto vecNew = i1.result();

    ImplicitEulerIBVP i2(*ibvp,2*N, J);
    auto vecNewE = i2.result();

    for (std::size_t j = 0; j < vecNew.size(); ++j)
    { // Combine in previous loop

       vecNew[j] = 2.0*vecNewE[j] - vecNew[j];
    }

}

Vector& ExtrapolatedImplicitEulerIBVP::result()
{
    ImplicitEulerIBVP i1(*ibvp, N, J);
    vecNew = i1.result();

    ImplicitEulerIBVP i2(*ibvp, 2 * N, J);
    auto vecNewE = i2.result();

    for (std::size_t j = 0; j < vecNew.size(); ++j)
    { // Combine in previous loop

       vecNew[j] = 2.0*vecNewE[j] - vecNew[j];
    }

    return vecNew;
}
```

We conclude our discussion of finite difference schemes by comparing the accuracy of fully implicit (IE), Crank–Nicolson (CN) and Richardson extrapolation (RIE) schemes for put and call option prices (exact prices are 5.84628209 and 2.133368 for $S = 60$, respectively) for the following data:

```
Option myOption;
myOption.sig = 0.3; myOption.K = 65.0; myOption.T = 0.25;
myOption.r = 0.08; myOption.b = 0.08; myOption.beta = 1.0;
myOption.SMax = 325.0; myOption.type = 'P';
myOption.earlyExercise = false;
```

In general, CN and RIE are the most accurate and they produce similar results. IE is less accurate, as would be expected because it is first-order accurate. We give some numeric results and we focus on NX = NT = {325, 650, 1300}. For puts, we get:

- IE: 5.841284, 5.844843, 5.845828.
- CN: 5.842044, 5.845223, 5.846018.
- RIE: 5.842043, 5.845223, 5.846017.

For calls, we get for NX = NT = {325, 650, 1300, 2600}:

- IE: 2.128331, 2.131910, 2.132904, 2.133202.
- CN: 2.129132, 2.132315, 2.133104, 2.133303.
- RIE: 2.129129, 2.132309, 2.133104, 2.133302.

The ADE method is also second-order accurate. For example, the Barakat–Clark–Towler–Yang and Larkin–Roberts–Weiss methods give the following results, respectively:

- ADE_1: 5.841828, 5.845019, 5.84582, 5.84602.
- ADE_2: 5.841764, 5.844956, 5.845756, 5.845902.

We see that the Larkin method is less accurate than the Barakat–Clark method. We do not include a discussion of the performance of these schemes.

21.8 EXAMPLES AND TEST CASES

We give some initial examples. In this case we are interested in comparing the accuracy of the different schemes. For each scheme we display an array of option prices at expiration. In order to visualise the results we use the functionality of the Excel driver that we introduced in Chapter 14. We have already shown fragments of the following code in Chapter 14 and for this reason we give the code for a full program without further comment:

```
std::vector<double> OptionPrice(IBvpSolver& fdm)
{ // Compute option price using a FD scheme

    return fdm.result();
}
```

```
int main()
{
    /*double r;             // Interest rate
    double sig;             // Volatility
    double K;               // Strike price
    double T;               // Expiry date
    double b;               // Cost of carry

    double beta;            // Elasticity factor
    double SMax;            // Far field condition
    char type;              // Call or put
    bool earlyExercise;     // American option

    Option myOption;
    myOption.sig = 0.3; myOption.K = 65.0; myOption.T = 0.25;
    myOption.r = 0.08; myOption.b = 0.08; myOption.beta = 1.0;
    myOption.SMax = 325.0; myOption.type = 'P';
    myOption.earlyExercise = false;

    BlackScholesPde myImp(myOption);

    Range<double> rangeX(0.0, myOption.SMax);
    Range<double> rangeT(0.0, myOption.T); // Time horizon

    IBvp currentImp(myImp, rangeX, rangeT);

    // B. Finite Difference Schemes
    long J = 500;
    long N = 500;
    std::cout << "Number of space divisions: ";
    std::cin >> J;

    std::cout << "Number of time divisions: ";
    std::cin >> N;

    // Bunch of methods to test accuracy
    ExplicitEulerIBVP fdmEE(currentImp, 20*N, J);
    CNIBVP fdmCN(currentImp, N, J);
    ImplicitEulerIBVP fdmIE(currentImp, N, J);
    ExtrapolatedImplicitEulerIBVP fdmEIE(currentImp,N, J);
    ADE_BC_CRTP fdmADEBC(currentImp, N, J);
    ADE_Larkin_RW_CRTP fdmADELarkin(currentImp, N, J);
    ADE_BCUpwind_CRTP fdmADEBCUpwind(currentImp, N, J);

    // Run several schemes
    // V2 use C++11 futures
    auto vEE = OptionPrice(fdmEE);
    auto vCN = OptionPrice(fdmCN);
    auto vIE = OptionPrice(fdmIE);
    auto vEIE = OptionPrice(fdmEIE);
    auto vADEBC = OptionPrice(fdmADEBC);
```

```
    auto vADELarkin= OptionPrice(fdmADELarkin);
    auto vADEBCUpwind = OptionPrice(fdmADEBCUpwind);

    // Names of each vector
    std::list<std::string> labels
                          { "EE", "CN", "IE", "RIE","ADEBC",
                            "Larkin", "ADE upwind"};

    // The list of Y values
    std::list<std::vector<double> > curves
                         { vEE, vCN, vIE, vEIE, vADEBC,
                           vADELarkin, vADEBCUpwind };

    ExcelDriver xl; xl.MakeVisible(true);
    xl.CreateChart(fdmEE.XValues(), labels, curves,
                   "Comparing Schemes", "S", "price");

    return 0;
}
```

We give a snapshot from the generated Excel sheet so that the values produced by the different schemes can be compared as shown in Table 21.1.

TABLE 21.1 Comparison of methods

S	EE	CN	IE	RIE	ADEBC	Larkin	ADE upwind
0	65	65	65	65	65	65	65
1	62.71292	62.71292	62.71295	62.71292	62.71256	62.71295	62.70956
2	61.71291	61.71291	61.71293	61.71291	61.71293	61.71293	61.71279
3	60.71291	60.71291	60.71294	60.71291	60.71294	60.71294	60.71293
4	59.71291	59.71291	59.71294	59.71291	59.71294	59.71294	59.71294
5	58.71291	58.71291	58.71294	58.71291	58.71294	58.71294	58.71294
6	57.71291	57.71291	57.71294	57.71291	57.71294	57.71294	57.71294
7	56.71291	56.71291	56.71294	56.71291	56.71294	56.71294	56.71294
8	55.71291	55.71291	55.71294	55.71291	55.71294	55.71294	55.71294
9	54.71291	54.71291	54.71294	54.71291	54.71294	54.71294	54.71294
10	53.71291	53.71291	53.71294	53.71291	53.71294	53.71294	53.71294
11	52.71291	52.71291	52.71294	52.71291	52.71294	52.71294	52.71294
12	51.71291	51.71291	51.71294	51.71291	51.71294	51.71294	51.71294
13	50.71291	50.71291	50.71294	50.71291	50.71294	50.71294	50.71294
14	49.71291	49.71291	49.71294	49.71291	49.71294	49.71294	49.71294
15	48.71291	48.71291	48.71294	48.71291	48.71294	48.71294	48.71294
16	47.71291	47.71291	47.71294	47.71291	47.71294	47.71294	47.71294
17	46.71291	46.71291	46.71294	46.71291	46.71294	46.71294	46.71294
18	45.71291	45.71291	45.71294	45.71291	45.71294	45.71294	45.71294
19	44.71291	44.71291	44.71294	44.71291	44.71294	44.71294	44.71294
20	43.71291	43.71291	43.71294	43.71291	43.71294	43.71294	43.71294
21	42.71291	42.71291	42.71294	42.71291	42.71294	42.71294	42.71294

TABLE 21.1 (*Continued*)

S	EE	CN	IE	RIE	ADEBC	Larkin	ADE upwind
//…							
50	13.90229	13.90232	13.90291	13.90232	13.90242	13.9024	13.90944
51	12.97235	12.97238	12.97298	12.97238	12.97248	12.97246	12.9811
52	12.06168	12.06171	12.06228	12.06171	12.0618	12.06178	12.07216
53	11.17351	11.17353	11.17404	11.17353	11.17362	11.17359	11.18582
54	10.31111	10.31113	10.31155	10.31113	10.31119	10.31117	10.3253
55	9.47769	9.477705	9.478009	9.477705	9.477753	9.477725	9.493789
56	8.676324	8.676332	8.676496	8.676332	8.676356	8.676326	8.694279
57	7.90982	7.90982	7.909825	7.90982	7.909815	7.909785	7.929533
58	7.180647	7.180638	7.180475	7.180638	7.180603	7.180574	7.20197
59	6.490865	6.490849	6.490517	6.490849	6.490782	6.490754	6.513605
60	**5.842068**	**5.842044**	**5.84155**	**5.842043**	**5.841946**	**5.84192**	**5.865992**
61	5.235345	5.235313	5.234673	5.235313	5.235187	5.235163	5.260194
62	4.671269	4.671231	4.670464	4.67123	4.671079	4.671058	4.696763
63	4.149897	4.149854	4.148989	4.149853	4.149681	4.149664	4.175751
64	3.670797	3.670751	3.669818	3.67075	3.670562	3.67055	3.696726
65	3.233083	3.233035	3.232065	3.233035	3.232837	3.232829	3.258812
66	2.835464	2.835416	2.83444	2.835415	2.835213	2.83521	2.860739
67	2.476302	2.476255	2.475302	2.476254	2.476053	2.476055	2.500892
68	2.153675	2.153631	2.152728	2.15363	2.153436	2.153443	2.17738
69	1.865444	1.865403	1.864572	1.865402	1.86522	1.865231	1.888096
70	1.609313	1.609276	1.608533	1.609275	1.609109	1.609124	1.630778
71	1.382891	1.382859	1.382218	1.382858	1.382711	1.382729	1.40307
72	1.183749	1.183722	1.18319	1.183722	1.183596	1.183617	1.202574
73	1.009467	1.009446	1.009025	1.009445	1.009342	1.009365	1.026901
74	0.857675	0.85766	0.857349	0.85766	0.85758	0.857604	0.87371
75	0.726091	0.726081	0.725876	0.726081	0.726024	0.72605	0.740741
76	0.612542	0.612537	0.61243	0.612537	0.612502	0.612528	0.625841
77	0.514989	0.514987	0.514969	0.514987	0.514973	0.514999	0.526988

In general, the ADE methods are approximately 40% faster than Crank–Nicolson or the fully implicit method. Richardson extrapolation is the slowest as we would expect, but its performance can be improved by code parallelisation.

21.9 SUMMARY AND CONCLUSIONS

In this chapter we have introduced a software framework (Figure 21.2) that allows us to approximate the solution of linear convection–diffusion–reaction PDEs in non-conservative form by finite difference methods. We then discussed the applicability to one-factor plain option pricing problems. The designs and documented experience can be applied to other kinds of applications.

We continue with more examples and extensions in Chapter 22.

21.10 EXERCISES AND PROJECTS

1. (FDM and Domain Transformation)

The boundary conditions for put options are always bounded for both $y = 0$ and $y = 1$. And the same conclusion holds for the payoff function (initial condition); it is always bounded. For call options on the other hand, the payoff function becomes:

$$\max(S - K, 0) = \max\left(\frac{\alpha y}{1 - y} - K, 0\right).$$

In this case we see that the payoff is unbounded at $y = 1$. In such cases we adopt the heuristic approach by offsetting the boundary by a small amount ϵ (typically 0.001) and then using $1 - \epsilon$ as the new boundary point.

2. (ADE for the Cox–Ingersoll–Ross Model)

In this exercise we approximate the CIR PDE using the standard implicit difference method (BTCS). Compare the results with the ADE method using domain transformation. In both cases we assume that the *Feller condition* $\sigma < \sqrt{2a}$ holds and then no boundary conditions need be prescribed at $r = 0$.

We take the example of a one-factor zero-coupon bond (see Tavella and Randall, 2000). The initial boundary value problem is given by:

$$\begin{cases} -\dfrac{\partial B}{\partial t} + \dfrac{1}{2}\sigma^2 r \dfrac{\partial^2 B}{\partial r^2} + (a - br)\dfrac{\partial B}{\partial r} - rB = 0, \quad 0 < r < r_{\max}, \quad t > 0 \\ B(r_{\max}, t) = 0, \quad t > 0 \\ -\dfrac{\partial B}{\partial t}(0, t) + a\dfrac{\partial B}{\partial r}(0, t) = 0, \quad t > 0, \quad a > 0 \\ B(r, 0) = H(r) \text{ (payoff)}, \quad 0 < r < r_{\max}. \end{cases}$$

Please note that we are using forward time. The tricky part is the boundary condition at $r = 0$; it is a *first-order hyperbolic equation*. We thus conclude that the bond price B is not known at $r = 0$. To this end, the implicit Euler scheme is:

$$\begin{cases} -\dfrac{B_j^{n+1} - B_j^n}{k} + \sigma_j^{n+1} D_+ D_- B_j^{n+1} + \mu_j^{n+1} D_0 B_j^{n+1} - r_j B_j^{n+1} = 0, \quad 1 \le j \le J - 1 \\ \begin{cases} \sigma \equiv \dfrac{1}{2}\sigma^2 r \text{ (slight misuse of notation)} \\ \mu \equiv a - br \end{cases} \end{cases}$$

while the scheme at $r = 0$ is given by the *upwind scheme* (h and k are the step sizes in the r and t directions, respectively):

$$\begin{cases} -\dfrac{B_j^{n+1} - B_j^n}{k} + a\dfrac{B_{j+1}^{n+1} - B_j^n}{h} = 0, \text{ when } \quad (j = 0) \\ \text{or} \\ B_0^{n+1}(1+\lambda) = B_0^n + \lambda B_1^{n+1} \quad \left(\lambda \equiv \dfrac{ak}{h}\right). \end{cases}$$

We now assemble the discrete equations. Define the unknown vector B by:

$$B^{n+1} = \left(B_0^{n+1}, \ldots, B_{J-1}^{n+1}\right)^\top.$$

Then the system of equations is:

$$A^{n+1}B^{n+1} = F^n$$

where:

$$A^n = \begin{pmatrix} 1+\lambda & -\lambda & & & & \\ a_2^n & b_1^n & c_1^n & & & \\ & \ddots & \ddots & \ddots & 0 & \\ 0 & & \ddots & \ddots & \ddots & \\ & & & a_J^n & b_{j-1}^n & c_{J-1}^n \end{pmatrix}$$

and

$$F^n = \left(B_0^n, 0, \ldots, 0\right)^\top.$$

The matrix A is an M-matrix and hence it has a positive inverse. We thus conclude that our finite difference scheme is monotone.

Answer the following questions:

a) Implement the fully implicit method. You will need to experiment with the determination of the truncated domain far-field value and the choice of boundary conditions at the far field. Compare the approximate solution with the well-known exact affine solution.

b) We now apply the ADE method to the CIR PDE. First, use domain transformation to transform the CIR PDE to a PDE on the unit interval (0, 1). Assuming the Feller condition holds, determine the boundary conditions at $y = 0$ and at $y = 1$.

c) Apply ADE to the non-conservative form of the transformed CIR PDE. Use Barakat–Clark averaging.

d) Convert the CIR PDE to conservative form.

e) We need to approximate a first-order hyperbolic PDE at $y = 0$ and integrate it into the main ADE scheme. Ideally, this approximation should be unconditionally stable and

second-order accurate. One possibility is the *box scheme* (see Duffy, 2006, p. 110) which is essentially one-step averaging in space and time:

$$-\frac{B_{j+1/2}^{n+1}-B_{j+1/2}^{n}}{\Delta t}+\alpha\frac{B_{j+1}^{n+1/2}-B_{j}^{n+1/2}}{h}=0$$

where:

$$B_{j+1/2}^{n}\equiv\tfrac{1}{2}(B_{j+1}^{n}+B_{j}^{n}),\quad B_{j}^{n+1/2}\equiv\tfrac{1}{2}(B_{j}^{n+1}+B_{j}^{n}).$$

f) Create a mini-ADE scheme near $y=0$ by solving for B_0^{n+1} and B_1^{n+1} and then apply the standard ADE method from the second mesh point in space.

g) Compare the accuracy of all ADE schemes and their variations with the exact solution.

3. (ADE for Two-Dimensional Problems Project)

In the interests of completeness, we discuss the applicability of ADE to n-factor PDEs. To this end, we consider the two-dimensional heat equation with a given inhomogeneous forcing term $f(x,y,t)$ and Dirichlet boundary conditions $g(x,y)$ on the unit square:

$$\frac{\partial u}{\partial t}=\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}+f(x,y,t)\text{ in }(0,1)^2\times(0,T)$$
$$u(x,y,0)=g(x,y)\text{ on the boundary of }(0,1)^2.$$

Answer the following questions:

a) In the case of the heat equation in the quarter plane $(0,\infty)\times(0,\infty)$ use a domain transformation defined by:

$$z=\frac{x}{x+\alpha_1},w=\frac{y}{y+\alpha_2}$$

In this case α_1 and α_2 are user-defined parameters.
What is the new PDE in the coordinates z and w?

b) We consider a two-dimensional mesh and corresponding discrete mesh functions indexed by indices i and j in the directions x and y, respectively. A straightforward generalisation of the Barakat–Clark method to two dimensions is:

$$\frac{U_{i,j}^{n+1}-U_{i,j}^{n}}{k}=\frac{1}{h_1^2}\left(U_{i-1,j}^{n+1}-U_{i,j}^{n+1}-U_{i,j}^{n}+U_{i+i,j}^{n}\right)+\frac{1}{h_2^2}\left(U_{i,j-1}^{n+1}-U_{i,j}^{n+1}-U_{i,j}^{n}+U_{i,j+1}^{n}\right),$$
$$\frac{V_{i,j}^{n+1}-V_{i,j}^{n}}{k}=\frac{1}{h_1^2}\left(V_{i-1,j}^{n}-V_{i,j}^{n}-V_{i,j}^{n+1}+V_{i+i,j}^{n+1}\right)+\frac{1}{h_2^2}\left(V_{i,j-1}^{n}-V_{i,j}^{n}+V_{i,j}^{n+1}+V_{i,j+1}^{n+1}\right),$$
$$u_{i,j}^{n}=\frac{1}{2}\left(U_{i,j}^{n}+U_{i,j}^{n}\right).$$

This scheme consists of two 'sweeps'; the first sweep uses boundary conditions starting at corner (0, 0) while the second sweep uses boundary conditions starting at corner (1, 1). See Figure 21.3.

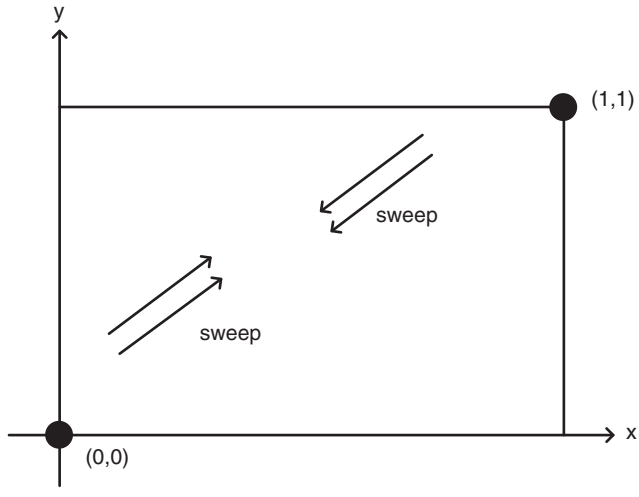


FIGURE 21.3 Visualisation of ADE process

A modification of this scheme is due to Larkin (1964) and uses the values of the averaged solution at each time level:

$$\frac{U_j^{n+1} - u_j^n}{k} = \frac{U_{j-1}^{n+1} - U_j^{n+1} - u_j^n + u_{j+1}^n}{h^2}$$
$$\frac{V_j^{n+1} - u_j^n}{k} = \frac{u_{j-1}^n - u_j^n - V_j^{n+1} + V_{j+1}^{n+1}}{h^2}$$
$$u_j^{n+1} = \frac{1}{2}\left(U_j^{n+1} + V_j^{n+1}\right).$$

The two-dimensional version can be achieved by modifying the Barakat and Clark scheme in part b).

Implement these schemes and compare their relative accuracy.

c) For two-factor PDEs with convection terms consider the Towler–Yang and Roberts–Weiss schemes.

d) In the case of mixed derivatives, use the Yanenko strategy (Duffy, 2006; Yanenko, 1971) in combination with ADE.

e) Implement the algorithms in parts c) and d) and integrate them into the ADE scheme for the two-factor Black–Scholes PDE. Compare your answers with some exact solutions, for example the Margrabe closed-form equation (see Haug, 2007).

4. (Combining Saul'yev ADE with Upwinding in Convection Term)

It is known that the Barakat–Clark method in combination with the Roberts–Weiss method can produce *spatial amplification errors* due to machine round-off errors at the boundaries, for example (see Roache, 1998 where it is also claimed that the Saul'yev method for the diffusion equation does not produce amplification errors). The von Neumann stability analysis does not given any clue to the presence of these computational instabilities.

Our aim in this exercise is to use the Saul'yev method in combination with upwinding for the convection term. We first concentrate on the constant-coefficient convection–diffusion equation:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = v\frac{\partial^2 u}{\partial x^2}\,(c, v > 0)$$

in addition to Dirichlet boundary conditions and an initial condition. Answer the following questions:

a) Verify that the Saul'yev scheme with upwinding leads to the following sweeps for odd and even time levels:

$$(1+\rho)\,u_j^{n+1} = (1-\rho)\,u_j^n + \rho\left(u_{j-1}^{n+1} + u_{j+1}^n\right) - \tau\left(u_j^n - u_{j-1}^n\right)$$

and

$$(1+\rho)\,u_j^{n+1} = (1-\rho)\,u_j^n + \rho\left(u_{j+1}^{n+1} + u_{j-1}^n\right) - \tau\left(u_j^n - u_{j-1}^n\right)$$

where:

$$\tau = \frac{c\Delta t}{h}, \rho = \frac{v\Delta t}{h^2}.$$

b) Implement this scheme for the linear convection–diffusion equation:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = v\frac{\partial^2 u}{\partial x^2}, 0 \le x \le 20, t \ge 0$$

with initial condition:

$$u(x,0) = \exp\left(-\alpha\left(1-x_1\right)^2\right), \alpha = 6, x_1 = 2.$$

The exact solution is given by:

$$u(x,t) = (1+4\alpha vt)^{-1/2}\exp\left\{-\frac{\alpha\left(x-x_1-ct\right)^2}{1+4\alpha vt}\right\}.$$

c) Implement this scheme for the viscous *Burgers' equation*:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = v\frac{\partial^2 u}{\partial x^2}, 0 \le x \le 10, t \ge 0$$

with initial condition:

$$u(x,0) = \begin{cases} 1, & 0 \le x < 1 \\ 0, & 1 < x \le 10. \end{cases}$$

The exact solution is given by:

$$u(x,t) = \frac{1}{2}\left(1 - \tanh\left[\frac{\frac{1}{2}(x - t/2 - 1)}{2v}\right]\right).$$

You can use the following approximation to the nonlinear term:

$$u\frac{\partial u}{\partial x} \sim u_j^n \frac{\left(u_j^n - u_{j-1}^n\right)}{h}.$$

d) Create a class in the software framework that implements the Saul'yev method in combination with upwinding:

$$u\frac{\partial u}{\partial x} \sim u_j^n \frac{\left(u_j^n - u_{j-1}^n\right)}{h}.$$

We remark that this method is stable if $|\tau| \leq 1$, where $\tau = \frac{c\Delta t}{h}$ as shown in Roache (1998).

e) Implement the B&C method with upwinding in the convection term. Do you get amplification errors? (Experiment with $\Delta t >> h$. Test the scheme on the Black–Scholes equation.)

5. (*Bridge* Pattern Revisited)

In Section 21.6 we applied the *Bridge* pattern as introduced in GOF (1995). In this exercise we examine some alternatives to this design and discuss the relative merits of each alternative. Answer the following questions:

a) Modify the code to produce a design based on CRTP (as we did with some of the FDM/ADE classes). What effects has this change had on the classes in Figure 21.2?

b) Modify the code to produce a design based on universal function wrappers. In other words, we create a *single class* consisting of a number of instances of `std::function`. What effects has this change had on the classes in Figure 21.2?

c) Determine the ease with which instances of the PDE classes corresponding to these three alternatives can be used in the design. In which cases are interface and implementation decoupled as discussed in GOF (1995)?

d) Measure the run-time performance of the three solutions.

6. (Quiz on *Bridge* Design Pattern)

In Exercise 5 we discussed three approaches to implementing the *Bridge* design pattern. Determine how far each solution satisfies the following *applicability requirements* as discussed in GOF (1995):

a) Avoiding a permanent binding between an abstraction and its implementation.

b) Both the abstraction and their implementations are extendible by subclassing.

c) Changes in the implementation of an abstraction should have no impact on clients.

d) We hide the implementation of an abstraction from clients.

e) We may wish to share an implementation among multiple objects.

f) Avoiding the proliferation of classes and class hierarchies.

In general, the objective of this exercise is to analyse how each solution from Exercise 5 satisfies the above requirements a)–f) to a lesser or greater extent. In other words, the solutions are compared with each other.

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 22

Extending the Software Framework

22.1 INTRODUCTION AND OBJECTIVES

This chapter is an application of the methods and code from Chapters 20 and 21. We are particularly interested in estimating delta and gamma, which are the first and second derivatives of the option price with respect to the stock, respectively. These two sensitivities are used to manage risk due to an option position. It is for this reason that we discuss them in some detail and we propose several methods to compute them.

The approach taken is to compute the option price array at expiration and then use it to compute delta and gamma using divided differences. We must take care with the choice of the finite difference scheme because some schemes (such as Crank–Nicolson, for example) can produce oscillations due to the fact that the payoff can be discontinuous or can have kinks at certain points, most notably at the strike (see Duffy, 2004A for a discussion). We resolve the problem by taking more time steps to dampen unwanted eigenvalues (see Lawson and Morris, 1978) or by using the fully implicit (BTCS) scheme. However, this scheme is only first-order accurate. Another alternative is to use BTCS for the first few time steps and then switch to Crank–Nicolson when the oscillations near the strike have disappeared.

We can also view this chapter from a software testing perspective, namely the code uses the functions and classes from Chapters 13, 20 and 21. Since we have analytical formulae for delta and gamma (and for other sensitivities as enumerated in Haug, 2007), we can test how accurate, robust and efficient our code is by comparing approximate results against the exact results. In this way, we test the finite difference schemes, matrix solvers and cubic spline interpolator algorithm to produce the end result. It is a comprehensive test of the code. Finally, the facility to display results in Excel is a boon because we can see results at a glance. We also give a discussion of software design patterns that are used to extend and customise the functionality of software systems. We also propose some guidelines on improving these design patterns.

22.2 SPLINE INTERPOLATION OF OPTION VALUES

In Chapters 20 and 21 we introduced the finite difference method and its implementation in C++. The method approximates the exact solution of a PDE at discrete mesh points. In order

to compute approximate values between mesh points we need to perform some kind of interpolation. Some popular choices are (Duffy and Germani, 2013):

- Linear interpolation.
- Cubic spline interpolation (see Chapter 13 of the current book).
- Akima method.
- Hermite monotonicity-preserving cubic spline (Doughtery, Edelman and Hyman, 1989). We discuss the implementation of the Akima and Hyman methods in Chapter 27.

In all cases we wish to interpolate data at certain points in an interval. In this section we use cubic splines because the mesh points in this case are evenly distributed and it does not suffer from the somewhat pathological cases described in Section 13.5.2.

The design tactic in this section is the following:

A. Choose a PDE and approximate it by an FD scheme.
B. Use the FD's mesh array and option value array as input to the cubic spline interpolator algorithm.
C. Compute option values for a range of underlying values.

In the initial example we price a call option using the Crank–Nicolson method. We use the generated option price array as input to the cubic spline solver. We carry out a simple test by randomly generating an underlying (abscissa) value where we compare the exact and approximate option prices:

```
#include "General.hpp"
#include "ExcelDriverlite.hpp"
#include <random>
#include <iomanip>
#include "CubicSpline.hpp"
#include "OptionCommand.hpp"

int main()
{
    Option myOption;
    myOption.sig = 0.3; myOption.K = 65.0; myOption.T = 0.25;
    myOption.r = 0.08; myOption.b = 0.08; myOption.beta = 1.0;
    myOption.SMax = 325.0; myOption.type = 'C';
    myOption.earlyExercise = false;

    BlackScholesPde myImp(myOption);

    Range<double> rangeX(0.0, myOption.SMax);
    Range<double> rangeT(0.0, myOption.T); // Time horizon

    IBvp currentImp(myImp, rangeX, rangeT);

    // Finite Difference Schemes
    long J = 5;
```

```
    long N = 50;
    std::cout << "Number of space divisions: ";
    std::cin >> J;

    std::cout << "Number of time divisions: ";
    std::cin >> N;

    CNIBVP fdmCN(currentImp, N, J);

    std::cout << "Starting fdm...";

    auto sol = fdmCN.result();

    ExcelDriver xl; xl.MakeVisible(true);
    std::string title = std::string("Crank Nicolson");
    auto xarr = fdmCN.XValues();
    xl.CreateChart(xarr, sol, title);

    // Now do some interpolation with xarr and sol as input arrays
    // Generate a random number in [a,b]
    std::default_random_engine eng;
    std::random_device rd;
    eng.seed(rd());

    double a = 57.0; double b = 68.0;
    std::uniform_real_distribution<double> dist(a, b);
    double xvar = dist(eng);

    // Discussed in chapter 13, section 13.5
    CubicSplineInterpolator csi(xarr, sol, SecondDeriv);

    // Generate the exact price
    CallPrice cp(myOption.K, myOption.T, myOption.r, myOption.b,
    myOption.sig);
    try
    {
       double result = csi.Solve(xvar);
       std::cout << "Interpolated value at " << xvar << " "
                 << std::setprecision(16) << result << ", "
                 << cp.execute(xvar) << '\n';
    }
    catch (std::exception& e)
    { // Catch not in range of values

       std::cout << e.what() << '\n';
    }
    return 0;
}
```

For other use cases and scenarios related to interpolation, see Exercise 1.

22.3 NUMERICAL DIFFERENTIATION FOUNDATIONS

In this section we discuss numerical techniques to approximate the first and second derivatives of a function. We compute these quantities in a range of applications such as the numerical approximation of ordinary and partial differential equations, integral equations and more specifically computing option sensitivities. We discuss these numerical methods and their implementation in C++. We also discuss why numerical differentiation is an unstable process and how to remedy this problem.

22.3.1 Mathematical Foundations

In this section we examine a real-valued function of a real variable:

$$y = f(x). \tag{22.1}$$

In general we are interested in finding approximations to the first and second derivatives of the function f. This is needed in this book because in general the form of the function f may be unknown and it is thus impossible to calculate its derivatives analytically. Second, it may be costly to evaluate the derivatives. To this end, we resort to numerical approximations. Suppose that we wish to approximate the first derivative of y at some point a (see Figure 22.1) and assume that h is a (small) positive number.

The first approximation (called the *centred-difference formula*) is given by:

$$f'(a) \approx \frac{f(a+h) - f(a-h)}{2h}. \tag{22.2}$$

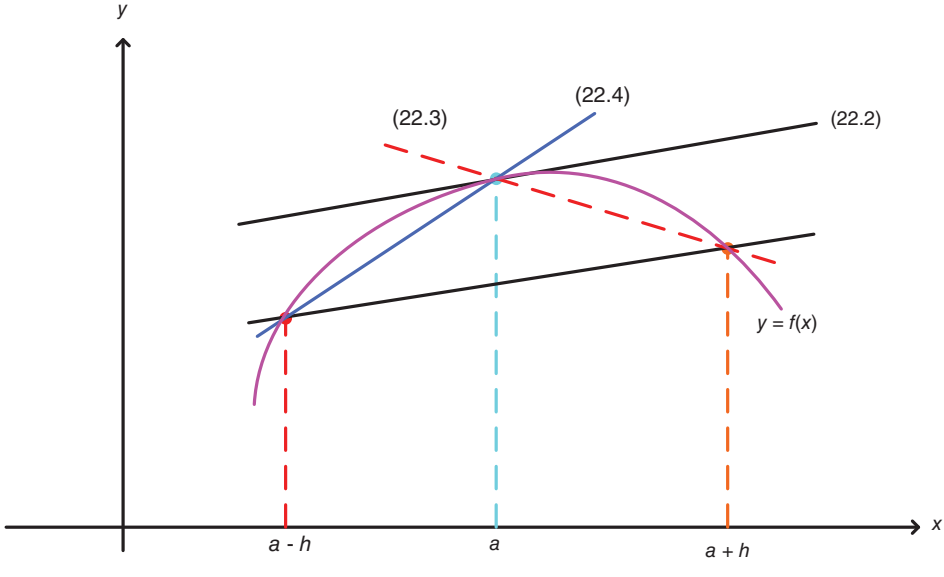


FIGURE 22.1 Motivating divided differences

Another approximation is called the *forward-difference formula* given by:

$$f'(a) \approx \frac{f(a+h) - f(a)}{h}. \tag{22.3}$$

Finally, the backward-difference formula is given by:

$$f'(a) \approx \frac{f(a) - f(a-h)}{h}. \tag{22.4}$$

We use the following notation:

$$D_0 f(a) \equiv \frac{f(a+h) - f(a-h)}{2h} \tag{22.5a}$$

$$D_+ f(a) \equiv \frac{f(a+h) - f(a)}{h} \tag{22.5b}$$

$$D_- f(a) \equiv \frac{f(a) - f(a-h)}{h}. \tag{22.5c}$$

Let us examine the centred-difference case, equation (22.5a) . We use Taylor's expansion with *exact remainder* to show that:

$$\begin{cases} f(a \pm h) = f(a) \pm h f'(a) + \dfrac{h^2}{2!} f''(a) \pm \dfrac{h^3}{3!} f'''\left(\eta_\pm\right) \\ \eta_- \in (a-h, a), \eta_+ \in (a, a+h) \end{cases} \tag{22.6}$$

from which we conclude that in this particular case:

$$D_0 f(a) = f'(a) + \frac{h^2}{6}\left(\frac{f'''\left(\eta_+\right) + f'''\left(\eta_-\right)}{2}\right). \tag{22.7}$$

We see that centred differences give a second-order approximation to the first derivative if h is small enough and if f has continuous derivatives up to order three. Similarly, some arithmetic shows that forward and backward differencing approximations to the first derivative of f at the point a are:

$$\begin{aligned} D_+ f(a) &= f'(a) + \tfrac{h}{2} f''\left(\eta_+\right),\ \eta_+ \in (a, a+h) \\ D_- f(a) &= f'(a) - \tfrac{h}{2} f''\left(\eta_-\right),\ \eta_- \in (a-h, a). \end{aligned} \tag{22.8}$$

We see that these one-sided schemes are only first-order accurate. On the other hand, they place low continuity constraints on the function f, namely we only need to assume that its second derivative is continuous.

We now discuss divided differences for the second derivative of f at some point a. To this end, we propose the following popular *three-point formula* (see Conte and de Boor, 1981):

$$D_+ D_- f(a) \equiv \frac{f(a-h) - 2f(a) + f(a+h)}{h^2}. \tag{22.9}$$

Thus, this divided difference is a second-order approximation to the second derivative of f at the point a and we assume that this function has continuous derivatives up to and including order four. The *discretisation error* is given by:

$$D_+D_-f(a) \equiv f''(a) + \frac{h^2}{4!}\left(f^{(iv)}\left(\eta_+\right) + f^{(iv)}\left(\eta_-\right)\right). \tag{22.10}$$

We normally apply the divided differences as defined in equation (22.5) to PDEs.

22.3.2 Using Cubic Splines

We have seen in Chapter 13 (equations (13.38) and (13.37)) how to approximate the first- and second-order derivatives of a function using cubic splines. This is an alternative to the approach in Section 22.3.1. The mathematical theory is discussed in Ahlberg, Nilson and Walsh (1967). Spline approximation is useful for high-precision numerical differentiation when sufficiently accurate data are available. Referring to the notation in Section 13.5 we see that the cubic polynomial has four unknown constants in each subinterval, hence there are $4n$ constants to be evaluated in total. The following continuity conditions at the interior mesh points allow us to find $3(n-1)+n+1$ (in total $4n-2$) of these constants, as can be checked from the following relationships:

$$\left.\begin{cases} S\left(x_j^-\right) &= S\left(x_j^+\right) \\ S'\left(x_j^-\right) &= S'\left(x_j^+\right) \\ S''\left(x_j^-\right) &= S''\left(x_j^+\right) \end{cases}\right\} 1 \le j \le n-1$$
$$S\left(x_j\right) = y_j, 0 \le j \le n. \tag{22.11}$$

We still need to find two extra conditions. To this end, one of the following sets of boundary conditions must be satisfied (Gao, Zhang and Cao, 2011):

a) *Clamped spline*: $S'\left(x_0\right) = y_0{}', S'\left(x_n\right) = y_n{}'$

b) *Curved-adjusted cubic spline*: $S''\left(x_0\right) = y_0{}'', S''\left(x_n\right) = y_n{}''$ (22.12)

c) *Periodic spline*: $S\left(x_0\right) = S\left(x_n\right), S'\left(x_0\right) = S'\left(x_n\right), S''\left(x_0\right) = S''\left(x_n\right)$

Periodic splines are not discussed in this book. Incidentally, cubic splines can be used as a numerical integrator of functions. In general, numerical differentiation is more difficult than numerical integration.

22.3.3 Initial Examples

We give an introduction to some of the issues to be aware of when computing numerical derivatives. We mention that numerical differentiation is an *ill-posed* problem. This means that a small change in the input parameters can result in large changes in output. Second, the accuracy can be influenced by catastrophic *cancellation*, for example when two almost equal numbers are subtracted from each other as discussed in Chapter 8.

We take an example based on the naïve assumption that a polynomial that is a good approximation to a function will be a good approximation to the function's derivatives by just differentiating the polynomial! The following counterexample shows how erroneous this premise is. To this end, we approximate the exponential function by the first three terms of its Taylor expansion:

$$f(x) = e^x$$
$$Q(x) = 1 + x + \tfrac{1}{2}x^2.$$

Then:

$$f = f' = f'' = f'''$$
$$Q'(x) = 1 + x, Q''(x) = 1, Q'''(x) = 0.$$

For $x \in [-0.1, 0.1]$ we get:

$$\max |f(x) - Q(x)| \approx 2 \times 10^{-4}$$
$$\max \left|f'(x) - Q'(x)\right| \approx 5 \times 10^{-3}$$
$$\max \left|f''(x) - Q''(x)\right| \approx 10^{-1}$$
$$\max \left|f'''(x) - Q'''(x)\right| \approx 1.$$

We see that the accuracy becomes progressively worse. As a new experiment, we use cubic splines to approximate the above function and its first two derivatives. We use the C++ class for spline interpolation from Chapter 13 and its member functions to compute these quantities. The code to approximate the exponential function is:

```
std::cout << "Cubic spline\n";
// Now choose Clamped Spline type, 1st derivatives specified at
// end-points. Approximate exp(x) on the interval [-0.1, 0.1]
double a = -0.1;
double b = 0.1;

double leftBC = std::exp(a);
double rightBC = std::exp(b);

std::size_t N = 10;
auto xarr = CreateMesh(N, a,b);

auto fun = [](double x) { return std::exp(x); };

CubicSplineInterpolator csi(xarr,fun, FirstDeriv, leftBC, rightBC);
double xVal = 0.009; // N.B. make sure this value is in the range [a,b]
double result = csi.Solve(xVal);
std::cout << "Interpolated + exact values: " << std::setprecision(12)
          << result << ", " << fun(xVal) << '\n';
std::cout << "Derivative value: " << std::setprecision(12)
          << csi.Derivative(xVal) << '\n';

auto tup = csi.ExtendedSolve(xVal);
```

```
std::cout << "Zero, 1st and 2nd derivatives: " << std::setprecision(12)
          << std::get<0>(tup) << ", " << std::get<1>(tup) << ", "
          << std::get<2>(tup) << '\n';

double exactIntegral = fun(b) - fun(a);
std::cout << "Integrals: " << std::setprecision(12)
          << csi.Integral() << ", " << exactIntegral << '\n';
```

The code produces the following output:

```
Cubic spline
Interpolated + exact values: 1.00904061683, 1.00904062177
Derivative value: 1.00904183989
Zero, 1st and 2nd derivative value:
1.00904061683, 1.00904183989,1.00883252589
Integrals: 0.200333499995, 0.20033350004
```

As expected, the higher the derivative the less accurate is the approximation.

22.3.4 Divided Differences

In Section 22.3.1 we discussed several formulae to compute approximations to a function's first and second derivatives. These methods are useful when the function and its derivatives that are being approximated are expensive or difficult to compute. In these cases the use of finite difference provides a suitable alternative. The main challenge is to decide how accurate we want our approximation to be by choosing an appropriate step size h. The two main sources of error in the formulae in Section 22.3.1 are:

- *Truncation* (*discretisation*) *error* caused by higher-order terms in the Taylor's series expansion.
- *Roundoff error* caused by the fact that computers have *limited word length* together with *loss of significance* caused when nearly equal quantities are subtracted from each other. One remedy is to increase the number of significant digits to which a function is computed as the step size h becomes smaller by using *guard bits or multiprecision arithmetic*, for example.

We take some code examples using 32-bit, 64-bit and multiprecision data types. The free functions implement the formulae in Section 22.3.1:

```
#include <cmath>
#include <iostream>
#include <limits>
#include <functional>
#include <boost/multiprecision/cpp_dec_float.hpp>

template <typename T>
    using FunctionType = std::function<T(const T& t)>;

// Choice of floating point types
//using value_type = float;
//using value_type = double;
using value_type = boost::multiprecision::cpp_dec_float_100;
```

```
//using value_type = boost::multiprecision::cpp_dec_float_50;

template <typename T>
    T firstDividedDifference(const FunctionType<T>& f, T xval, T h)
{ // 2nd order approximation h^2/6

    return (f(xval+h) - f(xval-h)) / (2.0 * h);
}

//double optmalStep()
template <typename T>
    T secondDividedDifference(const FunctionType<T>& f, T xval, T h)
{ // 2nd order h^2/12

    return (f(xval+h) - (2.0 * f(xval)) + f(xval-h)) / (h * h);
}
```

A simple test program is:

```
int main()
{
    namespace mp = boost::multiprecision;

    // Take numerical derivative at specific point
    value_type x = 2.0;
    value_type h;
    char ans;

L1:

    std::cout << "Give step size: ";
    std::cin >> h;

    std::cout << "Approximation to exp(x)...\n";

    // You can define your own functions here to test the errors
    FunctionType<value_type> fun = [](value_type x) -> value_type
                { return x*x; };

    std::cout << std::setprecision(12)
              << "1st divided difference "
              << firstDividedDifference <value_type> (fun, x, h)
              << "\n2nd divided difference "
              << secondDividedDifference<value_type>(fun,x,h);

    std::cout << "continue? ";
    std::cin >> ans;

    if (ans == 'y') goto L1;

    return 0;
}
```

You can experiment with the code using various floating-point types and values for h. For example, we could keep decreasing h by a factor of 10 to determine when the divided differences start producing incorrect results for a given floating-point type. For example, in some cases the divided difference formula (22.9) can produce the value zero even though the exact second derivative is not zero! We discuss numerical differentiation with multiprecision data types in Appendix 1 to this book.

22.3.5 What is the Optimum Step Size?

What is the optimum value of h that minimises the sum of the magnitudes of the roundoff and discretisation errors (Conte and de Boor, 1981)? To answer this question in a concrete case, we take the example in Section 22.3.3 again. Let us assume that the error in computing e^x is $\pm 1 \times 10^{-8}$. Then it can be shown that the roundoff when using formula (22.7) is:

$$R = \pm \frac{2 \times 10^{-8}}{2h}.$$

The discretisation error can be computed approximately from equation (22.7):

$$T = -\frac{1}{6} h^2 \left(f''' \sim 1 \right).$$

The function to be minimised is thus:

$$g(h) = |R| + |T| = \frac{10^{-8}}{h} + \frac{1}{6} h^2.$$

The minimum value is the solution of $g'(h) = 0$ or $h^3 = 3 \times 10^{-8}$.
The optimum value is thus:

$$h = 10^{-3} \sqrt[3]{30} \approx 0.003.$$

You can experiment with various values to verify this result. See also Exercise 4.

22.4 NUMERICAL GREEKS

We now come to the topic of computing option sensitivities and how they are used to reduce risk. For more details, we refer to Haug (2007) and Hull (2006). *Option sensitivity* is the derivative of an option with respect to one or more parameters in the Black–Scholes equation. We focus on option *delta*, which is the derivative of the option value with respect to the underlying S while *gamma* is the derivative of delta with respect to S. In general, we wish to compute options accurately and efficiently. We list some choices:

- G1: Exact Greeks (Haug, 2007).
- G2: Numeric Greeks (Haug, 2007).
- G3: Using Monte Carlo and related simulation methods (Glasserman, 2004).
- G4: Using the binomial method.
- G5: Using the values from an FD option pricer (Duffy, 2004) and applying method G2.

- G6: Using the values from an FD option pricer and constructing a cubic spline to produce the option's delta and gamma.
- G7: '*Spline on spline*': in this case we use two splines. This is an effective tool for computing second derivatives. The technique is as follows: we apply a cubic spline to data in the usual way and we compute the spline's slope (first derivative). We then fit these slopes themselves (they are now input) and we use the resulting derivative as the desired second derivative. In this sense, we approximate the first derivative of the first derivative.

It is important to calculate an option's sensitivities in order to manage its position. First, the delta measures the absolute change in the option price with respect to a small change in the price of the underlying asset, for example in the case of a call option. The formula is:

$$\Delta c = \frac{\partial C}{\partial S}. \tag{22.13}$$

The delta represents the *hedge ratio*, the number of options to write or to buy in order to create a risk-free portfolio. The delta varies from zero for deep out-of-the money options to one for deep in-the-money calls. This is clear if we examine the payoff function. However, the delta is not continuous at the strike price K because it is zero to the left of K and one to the right of K for a call option. We thus expect problems near K and this is borne out in practice by the appearance of *spurious or non-physical oscillations* when we use Crank–Nicolson time averaging, for example (Duffy, 2004A).

Approximation of the delta takes place by using divided differences. We can choose between forward, backward and centred-difference schemes. For example, we use centred differences in the interior of the domain while we use one-sided divided differences at the boundaries:

Discrete delta:

$$\begin{aligned} &\frac{c_{j+1}^n - c_{j-1}^n}{2h},\ 1 \le j \le J-1 \\ &\frac{c_1^n - c_0^n}{h}\ (j=0) \\ &\frac{c_J^n - c_{J-1}^n}{h}\ (j=J). \end{aligned} \tag{22.14}$$

The gamma measures the change in delta:

$$\Gamma_C = \frac{\partial^2 C}{\partial S^2} = \frac{\partial \Delta c}{\partial S}. \tag{22.15}$$

The value is greatest for at-the-money options and it is nearly zero for deep in-the-money or deep out-of-the-money options. The gamma gives us an indication of the vulnerability of the hedge ratio.

We approximate the formula (22.15) for the gamma by using the divided differences:

$$\text{Discrete gamma} = \frac{c_{j+1}^n - 2c_j^n + c_{j-1}^n}{h^2},\ 1 \le j \le J-1. \tag{22.16}$$

22.4.1 An Example: Crank–Nicolson Scheme

In this section we examine the application of the Crank–Nicolson scheme that we discussed in Chapters 13, 20 and 21 to compute option price as well as delta and gamma. Our approach is to compute the array of option prices at expiration and then to compute delta and gamma based on formulae (22.14) and (22.16), respectively. This is the same as option G5 above. We are interested in determining if the results are accurate because in this case we know that Crank–Nicolson can produce oscillations with discontinuous payoffs (Duffy, 2004A; Lawson and Morris, 1978). It is possible to apply *Rannacher smoothing* or *Richardson extrapolation*.

We discuss the code to compute option delta using both divided differences and cubic splines. The main steps are:

1. Choose an option type and approximate the related PDE using an FD scheme. The output is an array (call it A) of computed option prices at expiration.
2. Compute the delta at each underlying value using divided differences. The output is an array whose size is two less than that of the price array of A (the values of delta of the boundaries are not included).
3. Compute the delta for each underlying value using cubic splines. The output is an array whose size is two less than that of the price array for the same reasons as in step 2.
4. Compute the delta for each underlying value using the exact formula (see Haug, 2007). The output is an array whose size is two less than that of the price array.
5. Display the arrays in steps 1, 2 and 3 using the Excel driver and compare the results.

This approach is a good way to test and debug the software because inconsistencies and anomalies can be spotted immediately. We now show the C++ code that realises the above steps. We use the same example and variables as in Chapter 21 in order to avoid code duplication. First of all, we choose an FD scheme and we compute the array of option prices:

```
CNIBVP fdm(currentImp, N, J);
//   ImplicitEulerIBVP fdm(currentImp, N, J);
//   ADE_BC_CRTP fdm(currentImp, N, J);

auto sol = fdm.result();
auto xarr = fdm.XValues();
```

The code to compute delta in three ways is:

```
// Greeks I: delta
// Underlying arrays for divided differences
double h = rangeX.spread() / static_cast<double> (J);

std::vector<double> zarr(xarr.size() - 2);
for (std::size_t j = 0; j < zarr.size(); ++j)
{
    zarr[j] = xarr[j + 1];
}

// Delta array: eq. (22.14) but centred difference variant
std::vector<double> delta(zarr.size());
```

```
for (std::size_t j = 0; j < zarr.size(); ++j)
{
    delta[j] = (sol[j+2] - sol[j]) / (2*h);
}

// Exact solution
// OptionCommand(double strike, double expiration,
// double riskFree, double costOfCarry, double volatility)
CallDelta cDelta (myOption.K, myOption.T, myOption.r,
myOption.b, myOption.sig);
// PutDelta cDelta(myOption.K, myOption.T, myOption.r,
// myOption.b, myOption.sig);

std::vector<double> cDeltaPrices(zarr.size());
for (std::size_t j = 0; j < zarr.size(); ++j)
{
    cDeltaPrices[j] = cDelta.execute(zarr[j]);
}

// Compute delta from cubic splines
CubicSplineInterpolator csi2(xarr, sol, SecondDeriv);
std::vector<double> splineDelta(zarr.size());
for (std::size_t j = 0; j < zarr.size(); ++j)
{
    splineDelta[j] = csi2.Derivative(zarr[j]);
}
```

The code to compute option gamma is:

```
std::vector<double> splineGamma(zarr.size());
for (std::size_t j = 0; j < zarr.size(); ++j)
{
    splineGamma[j] = std::get<2>(csi2.ExtendedSolve(zarr[j]));
}

// CallGamma cGamma(myOption.K, myOption.T, myOption.r,
// myOption.b, myOption.sig);
PutGamma cGamma(myOption.K, myOption.T, myOption.r, myOption.b,
myOption.sig);
std::vector<double> cGammaPrices(zarr.size());
for (std::size_t j = 0; j < zarr.size(); ++j)
{
    cGammaPrices[j] = cGamma.execute(zarr[j]);
}

// Gamma array: eq. (22.16)
std::vector<double> gamma(zarr.size());
for (std::size_t j = 0; j < zarr.size(); ++j)
{
    gamma[j] = (sol[j + 2] - 2*sol[j+1] + sol[j]) / (h * h);
}
```

We can display the arrays in the Excel driver as discussed in Chapter 14:

```
ExcelDriver xl; xl.MakeVisible(true);
std::list<std::string> labels{ "d exact", "d Fdm", " d spline",
                               "g exact", "g fdm", "g spline" };
std::list<std::vector<double>> curves
                    {cDeltaPrices,delta,splineDelta,cGammaPrices,gamma,
                     splineGamma };
xl.CreateChart(zarr, labels, curves, "Deltas", "S", "dV/dS");
```

A snapshot of the Excel output is shown in Table 22.1.

Examining the accuracy of the results in Table 22.1 (see also Exercise 9) we see that both the Crank–Nicolson and cubic spline interpolator produce similar results for delta while the interpolator's accuracy for gamma is less than that of the Crank–Nicolson method. What is the reason? For smooth functions, centred divided difference approximations to the first and second derivatives result in second-order accuracy. This approach is based on Taylor expansions. Cubic splines are polynomials and they are a third-order accurate approximation to a smooth function. The first derivative of the spline is a second-order approximation to the exact delta while the second derivative of the spline is a first-order approximation to the exact gamma. This theoretical result is proven mathematically in Alberg, Nilson and Walsh (1967, Theorem 3.11.1, p. 94). Cubic spline interpolators can overshoot in order to always create a visually pleasing curve and this might account for the fact that the values for gamma are larger than the exact gamma or even the Crank–Nicolson gamma.

When approximations to the second derivatives of a smooth function are needed, we can first approximate it by using the values of the first derivative as input in order to compute an approximation to the second derivative. This is called *spline-on-spline* computation (Alberg, Nilson and Walsh, 1967, pp. 48–50; Hilderbrand, 1974, p. 492) and the improvements in accuracy are visible. In the current case a useful exercise would be to compare the values for gamma based on this variant with the original approach taken to produce the results in Table 22.1. We expect to get a better approximation to gamma.

TABLE 22.1 Output from Steps 1–5; data is K = 65, T = 0.25, r = b = 0.8, v = 0.3, NS = 325, NT = 1000

/* S	exact delta	CN delta	Spline delta	exact gamma	CN gamma	Spline gamma
59	0.330935233	0.330702564	0.330478248	0.040967715	0.04098453	0.041068871
60	0.372482798	0.372232265	0.372093554	0.042043376	0.042074873	0.042161739
61	0.41484882	0.414593361	0.414541127	0.042603904	0.042647319	0.042733408
62	0.457519085	0.457270132	0.457302102	0.042654217	0.042706223	0.042788541
63	0.499993235	0.499760129	0.499871257	0.042216737	0.04227377	0.042349769
64	0.541801022	0.541590678	0.541773643	0.041328768	0.041387329	0.041455004
65	0.582515647	0.582332471	0.582578239	0.040039354	0.040096257	0.040154187
66	0.62176381	0.621609821	0.621908227	0.038405882	0.038458444	0.038505789
67	0.659232357	0.659107471	0.659447781	0.036490702	0.036536856	0.03657332
68	0.694671633	0.694574052	0.694945475	0.034357967	0.034396306	0.034422069
69	0.727895851	0.727822509	0.728214631	0.032070838	0.032100608	0.032116243
70	0.758780913	0.758727922	0.759131055	0.029689185	0.029710218	0.029716604
71	0.787260172	0.787223244	0.787628682	0.027267796	0.027280425	0.027278649*/

22.5 CONSTANT ELASTICITY OF VARIANCE MODEL

The PDE describing this model is given by:

$$\frac{\partial V}{\partial t} = \frac{1}{2}\sigma^2 S^{2\beta}\frac{\partial^2 v}{\partial S^2} + rS\frac{\partial V}{\partial t} - rV \tag{22.17}$$

which is a special case of a more generic convection–diffusion–reaction PDE:

$$\frac{\partial u}{\partial t} = a(x,t)\frac{\partial^2 u}{\partial x^2} + b(x,t)\frac{\partial u}{\partial x} + c(x,t)u + f(x,t),\ 0 < x < \infty, \quad 0 < t < T \tag{22.18}$$

We discuss transforming the PDE (22.18) to a PDE on a bounded domain. We describe the steps in bullet form and you should check the steps for accuracy:

1. Introduce a new independent variable:

$$y = \frac{x}{x+\alpha}. \tag{22.19}$$

2. The transformed PDE becomes:

$$\frac{\partial u}{\partial t} = A(y,t)\frac{\partial^2 u}{\partial y^2} + B(y,t)\frac{\partial u}{\partial y} + C(y,t)u + F(y,t),\ 0 < y < 1 \tag{22.20}$$

where:

$$\begin{aligned} A(y,t) &= \tfrac{1}{2}\sigma^2\alpha^{2,\beta-2}y^{2\beta}(1-y)^{4-2\beta} \\ B(x,t) &= -\sigma^2\alpha^{2\beta-2}y^{2\beta}(1-y)^{3-2\beta} + ry(1-y), \qquad y = \frac{S}{S+\alpha} \\ C(x,t) &= -r,\ F(y,t) = 0. \end{aligned} \tag{22.21}$$

3. The boundary conditions in untransformed coordinates are:

$$\begin{aligned} C(0,t) &= 0 \\ C(S,t) &\to S \text{ as } S \to \infty. \end{aligned} \tag{22.22}$$

4. The boundary conditions in transformed coordinates must be calculated based on equation (22.22).

We discuss the problem in more detail in later chapters.

22.6 USING SOFTWARE DESIGN (GOF) PATTERNS

In this section we give a high-level overview of the famous software design patterns that first entered mainstream C++ software development projects in the mid-1990s. The first textbook on the topic was GOF (1995) and it provided developers with design techniques and

a vocabulary to help them write flexible and maintainable software. A description (from Wikipedia) is:

> *In software engineering, a software design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. It is a description or template on how to solve a problem that can be used in many different situations. Design patterns are formalised best practices that the programmer can use to solve common problems when designing an application or system.*

Design patterns may be viewed as a structured approach to computer programming situated between the levels of a programming paradigm and a concrete algorithm.

Design patterns have become very popular in the last 20 years, as witnessed by the number of books devoted to them for object-oriented languages such as Java, C#, C++ and others. Neither the structure nor the number of patterns has changed much in the last 20 years, as most of the literature seems to imitate the 23 patterns in GOF (1995). In a sense, these patterns were invented when C++ was still in its infancy and when it only supported the traditional object-oriented technology based on subtype polymorphism (`virtual` functions) and class hierarchies. Our basic premise is that the GOF patterns represent knowledge that has not adapted to improvements in software, hardware and development methods. Some of the issues relating to the traditional use of design patterns that we wish to address are:

- A1: The object-oriented style is not always the most appropriate model for all applications.
- A2: The GOF patterns are too low level and too detailed as driver of the software process for large and complex systems.
- A3: It is not necessary to implement a given GOF pattern from scratch; for example, Boost has a number of ready-to-use patterns such as *Flyweight*, *Functional Factory* (Demming and Duffy, 2010, 2012), *Meta State Machine* (MSM) and Boost *signals2*.
- A4: It seems that the documentation on design patterns has not been updated to reflect new developments in C++, for example smart pointers, lambda functions and the functional programming style.
- A5: GOF does not support *parallel design patterns* and *design blueprints* for applications that run on multicore processors.
- A6: 'There is more to life than design patterns.' In general, design patterns are neither necessary nor sufficient for successful software systems in our experience. We can run the risk of using the wrong pattern for the wrong problem. There is *no silver bullet* (Brooks, 1995):

  > *Of all the monsters that fill the nightmares of our folklore, none terrify more than werewolves, because they transform unexpectedly from the familiar into horrors. For these, one seeks bullets of silver that can magically lay them to rest.*

- A7: Object-oriented technology, and in particular design patterns, approach code construction in a bottom-up manner. This state of affairs leads to code that is difficult to maintain, especially if it is badly documented.
- A8: We would like to integrate software design patterns with the top-down decomposition methods introduced in Chapter 9.

We expand on these topics in the coming sections. Due to space limitations, we are unable to discuss individual patterns in any great detail. We address these issues in a forthcoming text.

22.6.1 Underlying Assumptions and Consequences

The GOF design patterns are based on the object model, which means that the patterns are implemented using objects, classes and class hierarchies in combination with subtype polymorphism. This means that an abstract requirement regarding the flexibility of a software design must be turned into an explicit data model by introducing a *proxy* for a non-computable concept. This process is called *reification* and it allows any aspect of a programming language to be expressed in the language itself. In this sense we say that reification data is turned into a *first-class object*.

We take an example of reification from the patterns catalogue. A common requirement in computational applications is to avoid hard-coded algorithms in code. This is because we may wish to replace the algorithm by another one having the same signature but better performance or better accuracy, for example. A short description of the resulting *Strategy* pattern is:

> *Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it. Capture the abstraction in an interface; bury implementation details in derived classes.*

In order to implement this pattern we create a class hierarchy in which each class encapsulates a particular implementation of the algorithm. Clients can then use the algorithms by using *composition* (embed an algorithm object as client member data) or by calling a member function in the client with an algorithm object as input parameter.

In general, reification is needed in the current context because many software requirements do not have a direct mapping to objects or to the programming language in question. We then create classes to adapt and encapsulate these requirements in some way. To this end, some examples in GOF (1995) are:

- The *Factory Method* (also known as *virtual constructor*) pattern defines an abstract interface (via an abstract class) for creating an object. C++ still does not support interfaces as in C# and COM (Albahari and Albahari, 2016; Rogerson, 1997) and it is for this reason that the *Factory Method* pattern and several other creational patterns must create class hierarchies.
- The *Observer* pattern that implements event notification mechanisms based on the *signals and slots* metaphor. Again, we need to reify these concepts resulting in two class hierarchies. Other languages and libraries such as C# and Boost have direct support for these concepts in the form of (multicast) delegates and signals and slots, respectively.
- The *Command* pattern encapsulates a request or message as an object. We create a class hierarchy in which each derived class implements a request on a receiver.
- The *Visitor* pattern extends the functionality of a class (data) hierarchy by allowing the developer to create new methods without having to change the classes of the elements on which it operates.

Most of the other GOF patterns employ a reification process that is similar to the above discussion.

22.6.2 Pattern Classification

The origins of design patterns for software systems date back to the 1970s and 1980s. It was not until around 1995 that they were published in book form by Eric Gamma and co-authors (GOF, 1995). This influential book spurred interest in the application of design patterns to software projects in C++ and Smalltalk.

The motivation for using design patterns originated from the work of architect Christopher Alexander:

> *Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a millions times over, without ever doing it the same way twice.*

The author began working with design patterns in 1993 and applied them in different kinds of applications such as CAD and computer graphics, optical technology, process control, real time and finance. Once you learn how a pattern works in a certain context you will find that it is easy to apply in new situations. The GOF patterns are applicable to objects and to this end they model the *object lifecycle*, namely object creation, the structuring of objects into larger object networks and finally modelling how objects communicate with each other using *message passing*. The main categories are:

- *Creational*: these patterns reify the instantiation (object creation) process. The added value of these patterns is that they ensure that an application can use objects without having to be concerned with how these objects are created, composed or internally represented. To this end, we create dedicated classes whose instances (objects) have the responsibility for creating other objects. In other words, instead of creating all our objects in a main method (for example), we can delegate the object creation process to dedicated *factory objects*. This approach promotes the *single responsibility principle*.

  The specific creational patterns are:
  - *Builder* (for complex objects that we create in a step-by-step manner).
  - *Factory Method* (defines an interface for creating an object).
  - *Abstract Factory* (defines an interface for creating hierarchies of objects or families of related objects).
  - *Prototype* (creates an object as a copy of some other object).
  - *Singleton* (creates a class that has only one instance).
- *Structural*: these patterns compose classes and objects to form larger structures. We realise these class relationships by the appropriate application of structural modelling techniques such as *inheritance*, *association*, *aggregation* and *composition*.

  The structural patterns are:
  - *Composite* (recursive aggregates and tree structures).
  - *Adapter* (converts the interface of a class into another interface that clients expect).
  - *Facade* (defines a unified interface to a system instead of having to directly access the objects or subsystems in the system).
  - *Bridge* (a class that has multiple implementations).
  - *Decorator* (adds additional responsibilities to an object at run-time in a transparent way).
  - *Flyweight* (an object that is shared among other objects).

- *Proxy* (an object that is a surrogate/placeholder for another object to control access to it).

- *Behavioural*: these are patterns that are concerned with inter-object communication, in particular the implementation of *algorithms* and the sharing of responsibilities between objects. These patterns describe run-time control and data flow in an application. We can further partition these patterns as follows:
  - *Variations*: patterns that customise the methods of a class in some way. In general, these patterns externalise the code that implements member functions. The main patterns are:
    - *Strategy* (families of interchangeable algorithms).
    - *Template Method* (defines the skeleton of an algorithm in a base class; some variant steps are delegated to derived classes; common or *invariant* functionality is defined in the base class).
    - *Command* (encapsulates a request as an object; execute the command).
    - *State* (allows an object to change behaviour when its internal state changes).
    - *Iterator* (provides a means to access the elements of an aggregate object in a sequential way without exposing its internal representation).
  - *Notifications*: these patterns define and maintain dependencies between objects:
    - *Observer* (defines a one-to-many dependency between a *publisher* and its dependent *subscribers*).
    - *Mediator* (defines an object that allows objects to communicate without being aware of each other; this pattern promotes *loose coupling*).
    - *Chain of Responsibility* (avoids coupling between *sender* and *receiver* objects when sending requests; gives more than one object a chance to handle the request).
  - *Extensions*: patterns that allow us to add new functionality (in the form of member functions) to classes in a class hierarchy. There is only one such pattern in GOF (1995):
    - *Visitor* (defines a new operation on the classes in a class hierarchy in a non-intrusive way).

There are some other, somewhat less universal behavioural patterns in GOF (1995):

- *Memento* (captures and externalises an object's internal state so that it can be restored later).
- *Interpreter* (given a language, defines a representation for its grammar and defines an interpreter to interpret sentences in the language).

Which GOF patterns are useful when developing applications? An initial answer is that 20% of the design patterns are responsible for 80% of developer productivity in our experience. Many applications have similar characteristics and having determined what these characteristics are, we will be in a position to determine which design patterns to use. In general, application development involves the following activities:

- A1: Using *structural patterns* to compose classes and class hierarchies. In particular, we can extend the functionality of a class using inheritance and composition. The top three patterns are *Adapter*, *Composite* and *Decorator*.
- A2: Using *behavioural patterns* to extend and/or modify the classes in activity A1. On the one hand we may wish to extend the functionality of a class hierarchy (using the *Visitor* pattern), create flexible algorithms in single classes (*Strategy*) and in class hierarchies

(*Template Method* pattern). We also wish to keep objects in object networks consistent and in this case we could apply the *Observer* pattern, although its implementation as described in GOF (1995) is based on OOP (it is not even wrong) and a better solution is to use a signals-based approach (as described in Demming and Duffy, 2010) or to use .NET delegates which we shall describe in Chapter 27. For example, much of the functionality of the *Observer* pattern is realised by the *.NET Event* pattern.

- A3: Once we know which structural and behavioural patterns to use, we implement them using classes and objects. We also describe the structural relationships between these classes and objects. Of course, we instantiate these classes and this process must be customisable to allow us to instantiate classes in different ways and as flexibly as possible. To this end, we use creational patterns such as *Factory Method* (for specific classes), *Abstract Factory* (for inter-related classes and class hierarchies) and *Builder* (for configuring all objects in a complete application).

22.6.3 Patterns: Incremental Improvements

It is possible to *upgrade* the GOF patterns in a number of ways:

- S1: Keep and use the patterns in their current form without change.
- S2: Improve the patterns by using new improved C++ functionality such as shared pointers and the syntax that we discussed in the first 10 chapters of this book. These are relatively small improvements.
- S3: Re-engineer those patterns that can be implemented more easily and correctly using generic and functional programming models, for example.
- S4: Do not (yet) use design patterns but instead postpone their use in the design trajectory for as long as possible. Instead, we use the system decomposition techniques of Chapter 9 and hope to achieve the same (and possible improved) levels of flexibility as with GOF patterns but then by other means, specifically by defining standardised interfaces between components.

In general, we make a distinction between *legacy software systems* which may need to be upgraded and those systems that we design from scratch. In the former case the upgrade policies S1 and S2 might be appropriate while policy S3 is a possibility for design patterns that have been well designed and for which an upgrade has minimal impact on the stability of the software system. In the latter case we have the most freedom and we can choose the most appropriate design pattern (or combination of patterns) to satisfy our requirements.

22.7 MULTIPARADIGM DESIGN PATTERNS

In this book we discuss new C++ features that did not exist when the GOF patterns were born. We can speculate on whether these patterns were created in order to resolve some of the shortcomings in the language at the time of birth or whether they are indeed robust design techniques that can be reused in a variety of applications. Some patterns have fallen into disrepair, while others can be upgraded to give them a new lease of life. Yet others will be discovered that could not have been envisaged in GOF (1995).

We mention and introduce several design patterns as we progress in this book. We now give an overview of the top C++ features (in our opinion) that help upgrade design patterns. We also give some pointers on the features in the context of use cases S2 and S3.

- *Shared and unique pointers*: the original GOF patterns used raw pointers. Using smart pointers leads to more robust and reliable code.
- *Tuples*: provide the ability to group a collection of heterogeneous data as a single entity. They are particularly useful as part of the *Abstract Factory* and *Builder* patterns. The return type is a tuple of newly created objects. The GOF pattern interface has an abstract factory method for each product. Similarly, we can extend the *Builder* pattern to return a tuple whose elements can be created by *subcontractors*. At the moment we do not see many uses for tuples in improving structural and behavioural patterns. This may change as new applications for tuples are discovered.
- *Universal function wrappers* (`std::function`): these are abstract functions that can be 'instantiated' by assigning them to a *target method* of a callable object, for example. These are in fact polymorphic functions and using this fact allows us to reduce the number of derived classes that need to be created in patterns such as *Command*, *Strategy*, *Template Method* and *Observer*. This approach reduces the number of classes in an application and hence improves maintainability (see metrics M13 and M12 of Section 7.6.4).
- Boost *signals2*: this library supports *signals and slots* and is superior to the *Observer* and *Chain of Responsibility* patterns. Using class hierarchies to emulate events is somewhat of a fudge in our opinion.
- *Lambda functions* and stored lambda functions: these functions avoid our having to create classes and free functions, thus reducing drudgery and maintenance costs. These functions are most useful when developing code for creational patterns.
- *Variadic templates* and *variadic functions*: these features help to reduce code bloat. In the first case we only need to define a template class once and instantiate it many times while in the second case we only need to define a function with variadic arguments once.

A full discussion of these topics is outside the scope of this book and they will be discussed in a forthcoming work.

22.8 SUMMARY AND CONCLUSIONS

In this chapter we discussed how to compute some option sensitivities, notably delta and gamma. These are used for *hedging*. This is the reduction of risk by exploiting relationships between various risky investments. *Delta hedging* involves the perfect elimination of all risk by defining a hedge between an option and its underlying. *Gamma hedging* is used to reduce the size of each hedge which then results in a cost reduction. It is a more accurate form of hedging than delta hedging. In this chapter we discussed methods to compute these quantities.

22.9 EXERCISES AND PROJECTS

1. (Interpolation)

Create C++ code to perform linear interpolation. Test the code in the same style as the code in Section 22.2. Display the results in Excel. Compare the results with those produced by cubic spline interpolation in Section 22.2.

2. (Automated FDM Tester, Medium-Sized Project)

The objective of this exercise is to develop a *proof-of-concept* software framework to test the accuracy, robustness and run-time performance of a given finite difference scheme.

The rationale is that an automated procedure can be run without human intervention and it produces a *management report* (for example, the results could be stored in a database) on the quality of the finite difference scheme that we are testing. In order to scope the problem we consider testing ADE against the Crank–Nicolson (CN) method and we can assume that the results produced by CN are accurate. Some of the features and requirements (accuracy and efficiency are the most important quality issues) are:

- F1: Test the accuracy of ADE against CN. Determine those regions in which ADE produces values that deviate from those produced by CN.
- F2: Test the robustness of ADE for a range of input parameters (which may be randomly generated).
- F3: Investigate the accuracy in the case of *convection dominance*, large and small expiration and other stressful cases.
- F4: Examine performance and accuracy of the different ADE variants for a range of mesh sizes and time steps.
- F5: Compute option sensitivities (in particular, delta and gamma).

Answer the following questions:

a) Apply the methods in Chapter 9 to scope this problem by creating a system context diagram, describing what the major output and input are and how the core process maps input to output.

b) Determine the interfaces of the modules in the system context diagram. Implement each module as an *adapter class* that reuses existing code.

c) Determine how to model the input data. In an initial version it can be hardwired in the code. If time permits you can consider using XML files, regular expressions (C++ has a regex library) or the *Boost Parameter* library. An important part of the project is to define the data structures to hold both the input data and the generated output data.

d) Do you see any opportunities for parallelising the code?

3. (Numerical Integrals and Derivatives with Cubic Splines, Small Project)

We gave an example in Section 22.3.3 of using a cubic spline to approximate a function and its derivatives. We also saw in Chapter 13 that cubic splines can be used for numerical integration (see also Stroud, 1974 for a mathematical discussion).

The main goal of this exercise is to determine how to use cubic splines in your applications as a competitor to other methods. Some of the reasons for doing so could be:

- Other methods may not be robust enough.
- Cubic splines have many applications and are easy to use.
- Cubic splines are accurate and have good performance characteristics.

Answer the following questions:

a) Compute the following integrals using cubic splines with different mesh sizes. How does the accuracy compare with the exact values and with those produced by other numerical integration rules (for example, the Gaussian methods that we discussed in Chapter 8)?

$$\int_1^2 \frac{dx}{x} \text{ (exact value 0.69314718055)}$$

$$\int_0^{10} \frac{dx}{1+x^2} \text{ (exact value 1.471127674304)}$$

$$\int_0^x e^{-y^2/2} dy \text{ where } 0 < x < \infty.$$

b) In Section 22.4 we discussed a number of methods to compute option sensitivities. Take some relevant examples and compare the methods with regard to accuracy and performance.

4. (Optimum Step Size in Numerical Differentiation)

Generalise the analysis (as far as possible) in Section 22.3.5 to compute the optimum h for an arbitrary function whose third derivative can be bounded from above. The error in computing function values is also an input value.

Find the function of h to be minimised and find the optimum h by solving a polynomial equation or using one of the nonlinear solvers in Chapter 19.

You may also find the C++ features regarding numerical precision in Chapter 8 to be useful.

5. (Binary Call Option)

This exercise involves computing the price, delta and gamma of a *binary call* option that has a payoff of 0 when $S \leq K$ at expiration and 1 otherwise. In other words, the payoff is a *Heaviside function*:

$$\text{Payoff } (S) = H(S - K)$$

where:

$$H(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0. \end{cases}$$

Answer the following questions:

a) Carry out an analysis similar to that in Section 22.4.1 using this option type as test case. We wish to compare the relative accuracy of the CN and ADE methods.

b) Compute the delta and gamma of a binary call option using the exact form as well as the finite difference and cubic spline approaches (Haug, 2007).

6. (Estimating Speed, Open Question)

The *speed* of an option is the rate of change of the gamma with respect to the stock price. It is the same quantity for both calls and puts:

$$\frac{\partial^3 C}{\partial S^3} = \frac{\partial^3 P}{\partial S^3} = \frac{\partial \Gamma}{\partial S}$$

where Γ is the option gamma.

The corresponding exact formulae are (Haug, 2007; Wilmott, 2006):

$$\frac{-\Gamma}{S}\left(\frac{d_1}{\sigma\sqrt{T}} + 1\right) \text{ where } d_1 = \frac{\log(S/K) + \left(r - b + \frac{1}{2}\sigma^2\right)T}{\sigma\sqrt{T}}.$$

A large value indicates that the gamma is very sensitive to changes in the underlying asset and it is useful when gamma is at its maximum with respect to the asset price.

The objective of this exercise is to determine how to estimate speed accurately and efficiently. Investigate the following alternatives:

a) Using a simple finite difference approximation to the third derivative:

$$f'''(x) \approx (f(x + 2h) - f(x - 2h) - 2(f(x + h) - f(x - h))) + 0\left(h^2\right).$$

b) Using the analytical formula.

c) Using cubic splines to compute its third derivative (why do we not expect that the approximation will be very good?).

Then:

$$S''(x) = M_j\left(\frac{x_{j+1}-x}{h_{j+1}}\right) + M_{j+1}\left(\frac{x-x_j}{h_j}\right),\ x \in \left[x_j,\ x_{j+1}\right],\ j = 0, \ldots, n-1.$$

$$S'''(x) = -M_j/h_{j+1} + M_j/h_j,\ x \in \left[x_j,\ x_{j+1}\right]\ j = 0, \ldots, n-1.$$

d) Using the first-order divided difference on the analytical formula for gamma.

e) '*Spline on spline*': compute the array of gamma prices as second derivative of a cubic spline and then use these prices as the independent variable array for another round of cubic spline interpolation. In this latter case we take the first derivative to give us the value for the speed.

7. (Far-Field Condition, Again)

Examine the effects on accuracy of finite difference schemes (option price and greeks) when we truncate the asset domain at *six-sigma* from the spot price:

$$S_{\max} = S_0 e^{6\sigma\sqrt{T}}$$

where S_0 = spot price, σ = volatility, T = expiration.

8. (Software Maintenance and Design Patterns)

We have given a global overview of the traditional GOF patterns in Section 22.6 and the possibility of improving them in some sense by trying to incorporate C++ syntax features. To make the exercise more focused, we place these patterns under the microscope as it were from the viewpoint of the software metrics that we described in Section 7.6. Answer the following questions:

a) In general, how far do GOF patterns promote 'good' software practice by keeping the values of the metrics in Section 7.6 within reasonable limits?

b) Which C++ features help to achieve the goals of software metrics?

c) Can you think of metrics that are not discussed in Section 7.6?

d) Determine the impact on software metrics if we decide to employ universal function pointers and tuples. Will their use result in better software by reducing the values of the threshold values?

9. (Producing 'Better' Gammas, Mini Project)

In Section 22.4.1 we discussed the different ways to calculate option delta and gamma using the analytic, CN and cubic spline interpolation approaches. We explained why these methods give the results that they do. In this exercise we ask you to use the *spline-on-spline* computation (explained in Section 22.4.1) to improve the accuracy of the gamma approximation. Answer the following questions:

a) Implement the *spline-on-spline* algorithm and encapsulate it as a reusable C++ module.

b) Test the module by running it on the data used to generate the results in Table 22.1. Do you get improved results compared to the values in the last column of that table?

c) Stress test the module for large and small values of the input parameters, for example the *convection dominance* case (small diffusion and/or large convection terms).

d) Calculate the delta and gamma for the CEV model that we discuss in Section 22.5.

10. (Hitting Times and First Exit Times)

In this exercise we examine a topic that is related to the determination of when a stochastic process reaches or 'hits' a given subset of state space. This is called the *hitting time* (or *first hit time*). A good example is when modelling path-dependent options, for example barrier options. In this case the payoff depends on whether the barrier has been triggered and we are interested in determining the likelihood of the barrier being triggered before expiration.

We apply the ADE method to compute the *first exit time for diffusion processes* (Patie and Winter, 2008; Wilmott, 2006). We use it to compute the probability of the first exit time from a bounded domain for multidimensional distributions. To this end, let $(\Omega, (F_t)_{t\geq 0}, P)$ be a filtered probability space and $X := (X^1, \ldots, X^d)^\top$ be the solution to the following system of stochastic differential equations:

$$dX_t^i = b_i(X_t)dt + \sum_{j=1}^{d} \sigma_{ij}(X_t)dW_t^j, \quad 1 \leq i \leq d$$

where $W := (W^1, \ldots, W^d)^\top$ is a d-dimensional standard Brownian motion with $d \geq 1$ and the coefficients b_i and σ_{ij} smooth, for $1 \leq i, j \leq d$.

We denote the *infinitesimal generator* of X which has the form:

$$Lf(x) = \frac{1}{2}\sum_{i=1}^{d}\sum_{j=1}^{d} a_{ij}(x)\frac{\partial^2 f(x)}{\partial x_i \partial x_j} + \sum_{i=1}^{d} b_i(x)\frac{\partial f(x)}{\partial x_i}$$

where $\alpha = \sigma^\top \sigma$ and $f \in C_K^2(\mathbb{R}^d)$, the space of twice continuously differentiable functions on $\mathbb{R}^d$ with compact support; in other words, these functions are identically zero outside a closed bounded subset of $\mathbb{R}^n$.

Let A be an open bounded Borel subset of $\mathbb{R}^d$ with boundary ∂A. We define the *stopping time*:

$$\tau_A = \inf\{u \geq 0; X_u \notin A\}.$$

This is the *first exit time* of X from domain A where we assume that ∂A is smooth. It is a random variable.

Let us denote by $Q(t, x)$ the probability that X starting from x did not exit the domain A before t, that is:

$$Q(t, x) = 1 - P_x(\tau_A < t).$$

Then Q coincides with the solution of the following *backward Kolmogorov equation*:

$$\frac{\partial u}{\partial t}(t, x) = Lu(t, x) \text{ on } (t, x) \in \mathbb{R}^+ \times A$$

associated with the process X with initial condition and boundary conditions:

$$u(0,x) = 1, \quad x \in A$$

$$u(t,x) = 0, \quad x \in \partial A, t > 0.$$

In this case the function $u(x,t)$ is the probability of a random variable leaving the region A before a given time.

Answer the following questions:

a) Compute the solution to the $(n = 1)$ exit time PDE for the one-dimensional heat equation with initial condition = 1 and boundary conditions = 0 using ADE. Compute the solution at the space mesh points for T = 0.1, 1.0, 2.0, 10.0, 100.0 and 1000.0.

b) Check that the computed result is always in the range [0, 1].

c) Now implement this problem using the Crank–Nicolson scheme. Compare its accuracy and efficiency to that of the ADE method.

d) Carry out the same exercise for two-dimensional diffusions. In this case we solve a two-factor convection–diffusion PDE with zero Dirichlet boundary conditions.

Concluding, we can compute the distribution of the first exit time from a bounded domain for n-dimensional diffusions.

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 23

A PDE Software Framework in C++11 for a Class of Path-Dependent Options

23.1 INTRODUCTION AND OBJECTIVES

In this chapter we discuss how to analyse, design and implement a C++ software framework for a class of partial differential equations (an exemplar of which is discussed in Wilmott, Lewis and Duffy, 2014 (subsequently called WLD, 2014)). The focus is on applying the author's system decomposition techniques (Duffy, 2004) in combination with the new features in C++11 to produce a customisable software framework that reproduces the numerical results in WLD (2014) and that can also be extended to other kinds of PDEs and finite difference schemes. We realise a certain level of flexibility in the new framework due to the following features:

- Each subsystem has a single major responsibility and well-defined and narrow interfaces. Complex mathematical operations are hidden behind these interfaces.
- We model PDEs as compositions of universal function wrappers using the *functional programming model*. In this way we avoid code bloat and proliferation of classes that arise when creating traditional class hierarchies based on subtype polymorphism or by using the curiously recurring template pattern (CRTP).
- We begin with the C++ code that we used to implement the Alternating Direction Explicit (ADE) in WLD (2014) and we port it to code that fits into the new software framework.
- Lambda functions help reduce code bloat, especially when configuring the application. Their use promotes code readability and maintainability.

Having created the application we can then apply a range of finite difference schemes to various PDEs. We are interested in the relative accuracy of the schemes and we would like to compute approximate option values in an efficient manner. We can improve speedup by testing the resulting sequential code against multithreading and multitasking code. Specifically, we can use C++11 threads and tasks as well as tasks in the Microsoft *Parallel Patterns Library* (PPL). We shall discuss these topics in Chapters 28 to 32.

This chapter is useful for model validators and front-office developers who wish to create pluggable applications in which components can be replaced by other components having similar functionality. An important assumption underlying our approach is that we design

software systems with change in mind. We propose three different solutions for this problem and we discuss the flexibility of each one.

23.2 MODELLING PDEs AND INITIAL BOUNDARY VALUE PROBLEMS IN THE FUNCTIONAL PROGRAMMING STYLE

Mathematically, a PDE is an aggregation or *composition* of its defining functions such as diffusion, convection, reaction and inhomogeneous terms. Time-dependent PDEs have one time input argument and n input arguments that represent the space dimension (in this chapter we have $n = 2$ because we are investigating a special two-factor PDE). Furthermore, a PDE is defined in a bounded or unbounded region of space–time with corresponding initial and boundary conditions. To this end, we create a class that models the terms of the PDE and a class that models the domain in which the PDE is defined. The first class has the interface:

```
template <typename T> class TwoFactorPde
{ // Model a convection-diffusion-reaction PDE as a
  // composition of universal function wrappers.

public:
    // U_t = a11U_xx + a22U_yy + b1U_x + b2U_y + cU_xy + dU + F;
    // f = f(x,y,t) in general

    std::function<T (T,T,T)> a11;
    std::function<T (T,T,T)> a22;
    std::function<T (T,T,T)> c;
    std::function<T (T,T,T)> b1;
    std::function<T (T,T,T)> b2;
    std::function<T (T,T,T)> d;
    std::function<T (T,T,T)> F;

    TwoFactorPde() = default;
};
```

This interface supports many of the linear two-factor PDEs in computational finance, for example basket and rainbow options, the Heston model, Asian options and PDEs in fixed-income applications (Duffy, 2006).

Basket options:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma_1^2 S_1^2 \frac{\partial^2 V}{\partial S_1^2} + \left(r - D_1\right) S_1 \frac{\partial V}{\partial S_1} + \frac{1}{2}\sigma_2^2 S_2^2 \frac{\partial^2 V}{\partial S_2^2} + \left(r - D_2\right) S_2 \frac{\partial V}{\partial S_2} + \rho\sigma_1\sigma_2 S_1 S_2 \frac{\partial_2 V}{\partial S_1 \partial S_2} - rV = 0. \tag{23.1}$$

The Heston model:

$$\frac{\partial U}{\partial t} + L_s U + L_v U + \rho\sigma v S \frac{\partial^2 U}{\partial S \partial v} = 0 \tag{23.2}$$

$$L_S U \equiv \frac{1}{2} v S^2 \frac{\partial^2 U}{\partial S^2} + rS\frac{\partial U}{\partial S} - rU = 0$$

$$L_v U \equiv \frac{1}{2}\sigma^2 v \frac{\partial^2 U}{\partial v^2} + \{\kappa[\theta - v(t)] - \gamma(S, v, t)\}\frac{\partial U}{\partial v}.$$

A convertible bond is defined by the following PDE:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + \rho\sigma S w \frac{\partial^2 V}{\partial \partial r} + \frac{1}{2} w^2 \frac{\partial^2 V}{\partial r^2} + rS\frac{\partial V}{\partial S} + (u - w\gamma)\frac{\partial V}{\partial r} - rV = 0 \tag{23.3}$$

$\gamma(r, S, t)$: Market price of risk, $-1 \leq \rho(r, S, t) \leq 1$: correlation.

These PDEs have essentially the same structure and we see that the interface can be a composition of universal function wrappers (compare with the designs based on subtype polymorphism and CRTP in previous chapters).

The second class models the space–time domain and boundary conditions and it has the interface:

```
template <typename T> struct TwoFactorPdeDomain
{
    // 1. Domain
    Range<T> rx;
    Range<T> ry;
    Range<T> rt;

    // 2. (Dirichlet) Boundary conditions, anticlockwise
    std::function<T (T x,T t)> LowerBC;    // y = yMin
    std::function<T (T x,T t)> UpperBC;    // y = yMax
    std::function<T (T y,T t)> LeftBC;     // x = xMin
    std::function<T (T y,T t)> RightBC;    // x = xMax

    // 3. Initial condition
    std::function<T (T x,T y)> IC;
};
```

In general, the space domain is bounded and it is the responsibility of the developer to either truncate a quarter-plane problem to a bounded region or apply a domain transformation to convert an unbounded domain to a bounded one. The second point to note is that we impose Dirichlet boundary conditions. This is a simplification to an extent, because boundary conditions are not always of Dirichlet type. A detailed discussion is outside the scope of the present work. Incorporating such boundary conditions into your schemes is 95% perspiration and not very difficult to do, but it does take some time. You may also need to implement *numerical boundary conditions* in the code that implements your FD scheme.

One of the advantages of creating two separate classes lies in our ability to model *many-to-many (N:N) relationships*: a given PDE instance (and FD schemes that approximate

it) can be associated with several domains. For example, we could consider the following use cases:

- U1: A PDE that is defined on two separate domains (one using domain truncation and the other using domain transformation).
- U2: A PDE that is associated with different sets of boundary conditions.
- U3: Defining a single PDE instance that is shared by a domain for call options and a domain for put options.

These use cases are useful when we wish to test a new scheme and to determine the optimal far-field condition, choosing those boundary conditions that produce the most accurate results. It is also possible to associate multiple PDE instances with a single domain. The potential advantages are less code duplication and opportunities to write multitasking code.

We note that the class `TwoFactorPde` corresponds to the *abstraction* role in the *Bridge* pattern. Finally, we can model PDEs by creating instances of `TwoFactorPde` for all cases. For example, each of the models in equations (23.1), (23.2) and (23.3) is mapped to an object and not a class. Using traditional OOP with inheritance usually entails creating a class for each new application. Our approach reduces any possible explosion in the number of classes.

23.2.1 A Special Case: Asian-Style PDEs

A special case of the two-factor models just discussed is when there is one stochastic factor which leads to a PDE in which there is one diffusion term and no mixed derivative term. An example is an Asian option defined by the following PDE:

$$\begin{aligned} &\frac{\partial V}{\partial t} + S\frac{\partial V}{\partial I} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0 \\ &I = \int_n^t S(\tau)\, d\tau. \end{aligned} \tag{23.4}$$

Our interest is in the model that we have already discussed in WLD (2014):

$$\begin{aligned} &dA = \lambda (S - A)\, dt \\ &\text{where:} \\ &A = \lambda \int_{-\infty}^t e^{-\lambda(t-\tau)} S(\tau)\, d\tau \end{aligned} \tag{23.5}$$

with λ a measure of the extent of memory.

We then get the PDE:

$$\frac{\partial V}{\partial t} = \frac{1}{2}\sigma^2 \left(\frac{S}{A}\right) S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV + \lambda (S - A)\frac{\partial V}{\partial A}. \tag{23.6}$$

The parameter λ is a measure of the extent of memory in the *anchoring model* in WLD (2014). We now discuss how to model the PDE in equation (23.6) and to this end we create

a new dedicated class rather than instantiating `TwoFactorPde` and setting one diffusion term and the cross-derivative term to zero. This new approach also leads to more readable and maintainable code at the expense of less generality and some extra code duplication. The class is thus a special case of `TwoFactorPde`:

```
template <typename T> class TwoFactorAsianPde
{   // Model a convection-diffusion-reaction PDE as a composition
    // of universal function wrappers. Asian-style PDE.

public:
    // U_t = a11U_xx + b1U_x + b2U_y + dU + F;
    // f = f(x,y,t) in general
    std::function<T (T,T,T)> a11;
    std::function<T (T,T,T)> b1;
    std::function<T (T,T,T)> b2;
    std::function<T (T,T,T)> d;
    std::function<T (T,T,T)> F;

    TwoFactorAsianPde() = default;
    TwoFactorAsianPde(const TwoFactorAsianPde& pde)
       : a11(pde.a11),b1(pde.b1),b2(pde.b2),d(pde.d),F(pde.F) {}
};
```

The class `TwoFactorAsianPde` is the one that we will be using in the rest of this discussion. The original form defined by equation (23.6) is defined on a quarter plane $0 < S < \infty, 0 < A < \infty$. As in WLD (2014) we transform this domain to the unit square.

We note that `TwoFactorAsianPde` is an abstract class in a certain sense. It cannot be instantiated directly without its members (which are universal function wrappers) being assigned to concrete *target methods*. In this sense `TwoFactorAsianPde` plays the role of the *abstraction* in the *Bridge* pattern. Clients instantiate this class as objects without the need to create classes or heavyweight class hierarchies.

23.3 PDE PREPROCESSING

We transform PDE (23.6) to one using new variables x and y defined by:

$$x = \frac{S}{S+a},\ y = \frac{A}{A+b},\ 0 < x, y < 1.$$

Here a and b are scaling factors which can be chosen in such a way as to allow us to define *hotspot* values in both original and transformed spaces. For example, the hotspot value $(0.5, 0.5)$ in (x, y) space is usually the choice. Then the factors a and b are chosen in such a way that this hotspot corresponds to the user-defined hotspot (S_0, A_0) in (S, A) space, that is $a = S_0, b = A_0$.

The C++ class that models the PDE in equation (23.6) based on the new coordinates becomes:

$$\begin{aligned}\frac{\partial V}{\partial t} &= \frac{1}{2}\sigma^2\left(\frac{S}{A}\right)x^2(1-x)^2\frac{\partial^2 V}{\partial x^2} + \left\{rx(1-x) - \sigma^2x^2(1-x)^2\right\}\frac{\partial V}{\partial x} \\ &+ \left(\gamma(S-A)(1-y)^2/b\right)\frac{\partial V}{\partial y} - rV \\ \text{for } 0 < x < 1,\ 0 < y < 1 &\left(S = \frac{ax}{1-x}, A = \frac{by}{1-y}\right).\end{aligned} \tag{23.7}$$

We thus see that equation (23.7) is a PDE defined on the unit square $(0,1)\times(0,1)$. In order to describe the resulting initial boundary value problem we need to impose boundary conditions. For $x = 0$ and $x = 1$ we impose the well-known boundary conditions that apply to plain call or put options. For $y = 0$ and $y = 1$ no boundary conditions are needed, as mentioned in WLD (2014). This fact can be verified and motivated by application of the *Fichera theory* (see Duffy, 2009; Duffy and Germani, 2013) or by the *method of characteristics*.

23.4 THE ANCHORING PDE

The financial background to the current problem is given in WLD (2014). We define a *volatility function* $\sigma(\xi)$, where $\xi = S/A$ and the model is:

$$dS = \mu S dt + \sigma(\xi) S dW. \tag{23.8}$$

We model the problem as one with memory. To this end, we define a variable A that represents an average or an *anchoring value*. One of the simplest and most tractable models is given by the following *sigmoidal function*:

$$\sigma(\xi) = a + \frac{b-a}{1+e^{-c(\log\xi - d)}} \tag{23.9}$$

where a, b, c and d are known parameters.

The form of this function has been determined from *time series* (see WLD, 2014) and it will be used as the volatility function in the PDE that is represented in equations (23.6) and (23.7).

Moving to C++ design of the anchoring PDE, we first note that it plays the role of the *implementer* of `TwoFactorAsianPde` in the *Bridge* pattern. It is an instance of `TwoFactorAsianPde` that is created by a *factory method*. In this case we choose from the following alternatives:

- A1: Defining the PDE components in a namespace.
- A2: Creating a `void` function that initialises an already created reference to an instance of `TwoFactorAsianPde`.
- A3: A *factory method* (as discussed in GOF, 1995) that creates an instance of `std::shared_ptr<TwoFactorAsianPde<double>>`.

We discuss each of these options as possible ways to create the objects that we need. Each choice has its advantages and disadvantages. The ultimate choice depends on the context. The traditional object-oriented programming style would tend to employ subtype polymorphism or the CRTP, both of which necessitate the creation of class hierarchies. The choices A1, A2 and A3 avoid this step.

First, choice A1 involves placing all relevant data and functions in a humble namespace:

```
namespace AnchorPde
{ // Similar to an Asian PDE.

    double SIGMOID2(double t)
    { // t = S/A

        // Example of Table 2
        const double a=0.6; // (vol for low S/I)
        const double b=0.15; // (vol for high S/I)
        const double c=10.0;
        const double d=-0.3; // (when S=I vol is in the middle)
        const double ecd = std::exp(c*d);

        // Sigmoid2 function
        double val = exp(-c*(log(t) - d));

        double val2 = a + (b-a)/(1.0 + val);
        return val2;
    }

    // Scale factors (hotspots)
    double scale1;
    double scale2;

    // Parameters of PDE
    double r;
    double lambda;
    double K;

    // Domain information
    double T;
    double xMax, yMax;

    std::function<double(double S1, double S2)> payoff;

    double a11(double x, double y, double t)
    {

        double tx = 1.0-x; double ty = 1.0-y;

        double S = scale1*x/tx;
        double I = scale2*y/ty;
```

```
    // Straight computation, i.e. function call
    double s1 = SIGMOID2(S / I);

    return 0.5*s1*s1*x*x*tx*tx;
}

double b1(double x, double y, double t)
{
    double tx = 1.0-x; double ty = 1.0 - y;

    double S = scale1*x/tx;
    double I = scale2*y/ty;
    double s1 = SIGMOID2(S / I);
    return  r*x*tx - s1*s1*x*x*tx;
}

double b2(double x, double y, double t)
{
    double tx = 1.0-x; double ty = 1.0 - y;

    double S = scale1*x/tx;
    double I = scale2*y/ty;

    return lambda*(S - I)*ty*ty/scale2;
}

double d(double x, double y, double t)
{
    return -r;
}

double F(double x, double y, double t)
{
    return 0;
}

double IC(double x, double y)
{ // Initial condition

    // Workaround/fudge to avoid spike
    if (x == 1.0)
       x -= 0.0001;

    if (y == 1.0)
        y -= 0.0001;

    return payoff(scale1*x/(1.0-x), scale2*y/(1.0-y));
}

// Approach:
double BCLower(double x, double t)
```

```
    {

        // V1 rough and ready
        return 0;
    }

    double BCUpper(double x, double t)
    {
        // Tricky one for variable I
        return 0; // Force the solution to be BS classic.
    }

    double BCLeft(double y, double t)
    { // y is S2

        //    return 0; // C
        return K* exp(-r*t); // P

    }

    double BCRight(double y, double t)
    {
        return 0; // P
        double S = scale1*xMax / (1 - xMax+0.001);
        return S -K*std::exp(-r*t); // C
    }
}
```

The above variables in the namespace `AnchorPde` are initialised in `main()`:

```
using namespace AnchorPde;

r = 0.049;
std::cout << "T: "; std::cin >> T;
double S = 100; double I = 100;    // Values where price is calculated
                                   // S/I == 1

std::cout << "K: "; std::cin >> K;

lambda = 0.5;

// Define payoff as a lambda function
int cp = -1;
if (cp > 0) // Call
    payoff=[=](double S, double A)->double {return std::max(S - K, 0.0);};

else
    payoff=[=](double S, double I)->double {return std::max(K - S, 0.0);};

// Domain transformation data; we use the transform z = x/(x + scale1),
```

```
// w = y/(y + scale2). The hotspot (S1, S2) in (x,y) space gets mapped to
// (1/2, 1/2) (usually) in (z,w) space.
xMax = 1.0;  // x far field
yMax = 1.0;  // y FF

double xHotSpot = 0.5; double yHotSpot = 0.5;
scale1 = S*(1.0 - xHotSpot)/xHotSpot; scale2 = I*(1.0 - yHotSpot)/
yHotSpot;
```

Second, choice A2 is a variation on A1 in the sense that it uses the above variables and encapsulates the creation process in a single function:

```
void CreateAnchorPde(TwoFactorAsianPde<double>& pde)
{

    // Initialise all functions

    using namespace AnchorPde;

    pde.a11 = a11;

    // a22 = c = 0

    pde.b1 = b1;
    pde.b2 = b2;
    pde.d = d;
    pde.F = F;
}
```

We see that this function produces the side-effect of modifying the input parameter representing the PDE.

Finally, choice A3 is an example of the *Factory Method* design pattern that returns a `TwoFactorAsianPde` pointer. All low-level construction code is hidden behind the interface. Notice that we use lambda functions to create the coefficients of the PDE:

```
std::shared_ptr<TwoFactorAsianPde<double>> CreateAsianPde()
{ // Factory method to create an Asian pde for input to FDM

    // Steps
    // 1. Get input data
    // 2. Create the components of the PDE
    // 3. Return the instance of AsianPde

    double r = 0.049;
    double T;
    cout << "T: "; cin >> T;
    // Values where price is calculated S/I == 1
    double S = 100; double I = 100;
```

```
double K;
cout << "K: "; cin >> K;

double lambda = 0.5;

// Define payoff as a lambda function

double xMax = 1.0;              // x far field
double yMax = 1.0;              // y FF

double xHotSpot = 0.5; double yHotSpot = 0.5;
double scale1 = S*(1.0 - xHotSpot) / xHotSpot;
double scale2 = I*(1.0 - yHotSpot) / yHotSpot;

TwoFactorAsianPde<double> pde;

// Build components of pde
auto SIGMOID2 = [] (double t)
{ // t = S/A

  // Example of Table 2
   const double a = 0.6;  // (vol for low S/I)
   const double b = 0.15; // (vol for high S/I)
   const double c = 10.0;
   const double d = -0.3; // (when S=I vol is in the middle)
   const double ecd = std::exp(c*d);

    // Sigmoid2 function
    double val = exp(-c*(log(t) - d));

    double val2 = a + (b - a) / (1.0 + val);
    return val2;
};

auto a11 = [=] (double x, double y, double t)
{
    //    return 0;
    double tx = 1.0 - x; double ty = 1.0 - y;

    double S = scale1*x / tx;
    double I = scale2*y / ty;

    // Straight computation, i.e. function call
    double s1 =SIGMOID2(S / I);

    return 0.5*s1*s1*x*x*tx*tx;
};

auto b1 = [=] (double x, double y, double t)
{
    double tx = 1.0 - x; double ty = 1.0 - y;
```

```
        double S = scale1*x / tx;
        double I = scale2*y / ty;
        double s1 = SIGMOID2(S / I);
        return  r*x*tx - s1*s1*x*x*tx;
    };

    auto b2 = [=] (double x, double y, double t)
    {
        double tx = 1.0 - x; double ty = 1.0 - y;

        double S = scale1*x / tx;
        double I = scale2*y / ty;

        return lambda*(S - I)*ty*ty / scale2;
    };

    auto d = [=] (double x, double y, double t)
    {
        return -r;
    };

    auto F = [=] (double x, double y, double t)
    {
        return 0;
    };

    pde.a11 = a11;

    // a22 = c = 0

    pde.b1 = b1;
    pde.b2 = b2;
    pde.d = d;
    pde.F = F;

    return std::shared_ptr<TwoFactorAsianPde<double>>
    (new TwoFactorAsianPde<double>(pde));
}
```

Finally, we show the code (using design choice A1) that creates the domain in which the anchoring PDE is defined:

```
void CreateAnchorPdeDomain(TwoFactorPdeDomain<double>& pdeDomain,
            double xMax, double yMax, double T,
            const std::function<double(double, double)>& IC)
{
    using namespace AnchorPde;

    pdeDomain.rx = Range<double>(0.0, xMax);
```

```
    pdeDomain.ry = Range<double>(0.0, yMax);
    pdeDomain.rt = Range<double>(0.0, T);

    pdeDomain.LeftBC = BCLeft;
    pdeDomain.RightBC = BCRight;
    pdeDomain.UpperBC = BCUpper;
    pdeDomain.LowerBC = BCLower;

    pdeDomain.IC = IC;
}
```

We now discuss each of the solutions A1, A2 and A3 as input to the ADE scheme.

23.5 ADE FOR ANCHORING PDE

The ADE method is well documented. For more background information see Saul'yev (1964), Campbell and Yin (2006), Duffy (2009), Pealat and Duffy (2011), Duffy and Germani (2013) and Buchova, Ehrhardt and Guenther (2015). In the current case we apply both the Barakat–Clark and Saul'yev variants of ADE to the x factor terms in equation (23.7) while we use *implicit upwinding* for the y factor terms in equation (23.7).

We focus on the Barakat–Clark ADE variant with design choice A1. The corresponding class is called `TwoFactorAsianADESolver` and it has the following member functions:

- Constructor: initialises mesh arrays and matrices holding option prices.
- `calculate()`: computes the left-to-right and right-to-left sweeps and their averages at each time level.
- `calculateBC()`: calculates the boundary conditions on each of the four boundaries of the domain of integration at each time level.
- `initIC()`: initialises the solution at $t = 0$ by assigning it to the discretised values at the mesh points of the payoff function.
- `result()`: this is the implementation of what is essentially the *state machine* that marches the solution from payoff up to expiration. In a later version of the software it will be a member function of a *mediator* class, as already discussed in our work.

The code in this case is given by (notice that we are using the Boost *uBlas* library to model matrices):

```
namespace u = boost::numeric::ublas;
using namespace AnchorPde;

class TwoFactorAsianADESolver
{
private:
    TwoFactorPdeDomain<double> pdeDomain;  // Domain, BC, IC

    // Data structures
    u::matrix<double> U;                   // upper sweep, n+1
    u::matrix<double> V;                   // lower sweep, n+1
```

```
public:
    u::matrix<double> MatNew;              // averaged solution, level n+1
private:
    // Mesh-related data
    double hx, hy, delta_k, hx1, hy1, hx2, hy2;

    // Mesh-point values of coefficients
    double A,B,C,D,E,F,G;
    double t2,tx1,ty1;

    // Other variables
    double tprev, tnow, T;
    std::size_t NX, NY, NT;

public:
    std::vector<double> xmesh;
    std::vector<double> ymesh;
    std::vector<double> tmesh;

public:

    TwoFactorAsianADESolver(const TwoFactorPdeDomain<double>& domain,
                            const std::vector<double>& xarr,
                            const std::vector<double>& yarr,
                            const std::vector<double>& tarr)
            : xmesh(xarr), ymesh(yarr), tmesh(tarr)
    {
        cout << "Two-factor ADE Asian classic version\n";
        pdeDomain = domain;

        hx=pdeDomain.rx.spread() / static_cast<double>(xmesh.size() - 1);
        hy=pdeDomain.ry.spread() / static_cast<double>(ymesh.size() - 1);
        delta_k = pdeDomain.rt.spread()
                                / static_cast<double>(tmesh.size() - 1);

        T = pdeDomain.rt.high();

        // Extra, handy variables
        hx1 = 1.0/hx;
        hx2 = 1.0/(hx*hx);
        hy1 = 1.0/hy;
        hy2 = 1.0/(hy*hy);

        // Some optimising variables
        t2 = delta_k * hx2;
        tx1 = delta_k * hx1;
        ty1 = delta_k * hy1;

        // Initialise U, UOld, V, VOld, MatNew data structures
        // NumericMatrix(I rows, I columns, I rowStart, I columnStart);
        // Constructor with size & start index
```

```
    U = u::matrix<double>(xmesh.size(), ymesh.size());
    V = u::matrix<double>(xmesh.size(), ymesh.size());

    MatNew = u::matrix<double>(xmesh.size(), ymesh.size());

    initIC();
}

~TwoFactorAsianADESolver()
{

}

    ////////////////////////////////////////////////////////////////
void initIC()
{ // Utility function to initialise payoff function and BCs at t = 0

    tprev = tnow = tmesh[0];

    // Now initialise values in interior of interval using
    // the initial function 'IC' from the PDE
    for (std::size_t i = 0; i < xmesh.size(); ++i)
    {
        for (std::size_t j = 0; j < ymesh.size(); ++j)
        {
           MatNew(i,j) =  U(i, j) = V(i, j) = AnchorPde::IC(xmesh[i],
           ymesh[j]);
        }
    }
}

void calculateBC()
{ // Calculate the discrete BC on the FOUR edges of the boundary

    // Lower and Upper BC
    std::size_t lower = 0;
    std::size_t upper = ymesh.size()-1;
    for (std::size_t i = 0; i < xmesh.size(); ++i)
    {
       MatNew(i, lower) = U(i, lower) = V(i, lower)
                        = AnchorPde::BCLower(xmesh[i], tnow);// N/A
                                                             // anymore
       MatNew(i, upper) = U(i, upper) = V(i, upper)
                        = AnchorPde::BCUpper(xmesh[i], tnow);
    }

    // Left and Right BC
    std::size_t left = 0;
    std::size_t right = xmesh.size() -1;
    for (std::size_t j = 0; j < ymesh.size(); ++j)
    {
```

```
        MatNew(left, j) = U(left, j) = V(left, j) = AnchorPde::
        BCLeft(ymesh[j], tnow);
        MatNew(right, j) = U(right, j) = V(right, j) = AnchorPde::
        BCRight(ymesh[j], tnow);
    }
}

void calculate()
{ // Tells how to calculate sol. at n+1, Explicit ADE schemes

    double mx, my;
    int sgn;

    for (std::size_t i = 1; i <= xmesh.size()-2; ++i)
    {
       mx = xmesh[i];
       for (std::size_t j = 1; j <= ymesh.size()-2; ++j)
       {

          // Create coefficients
          // hU_t = aU_xx + bU_yy + cU_xy + dU_x + eU_y + fU + G;
          // f = f(x,y,t)
          my = ymesh[j];

          A = AnchorPde::a11(mx, my, tnow) * t2;
          D = 0.5*AnchorPde::b1(mx, my, tnow) * tx1;
          double sgnD = MySign(D);
          E = AnchorPde::b2(mx, my, tnow) * ty1;
          sgn = MySign(E);
          F = AnchorPde::d(mx, my, tnow) * delta_k;
          G = AnchorPde::F(mx, my, tnow) * delta_k;

          // Larkin + 1st order upwind in y
          U(i, j) = (U(i,j)*(1.0-A) + U(i+1,j)*(A+D)
                         + sgn*E*U(i,j+sgn) + U(i-1,j)*(A-D) + G)
                         / (1.0+A-F+sgn*E);
       }

    }

    for (std::size_t i = xmesh.size()-2; i >= 1 ; --i)
    {
       mx = xmesh[i];
       for (std::size_t j = ymesh.size() - 2; j >= 1; --j)
       {
          // Create coefficients
          my = ymesh[j];

          A = AnchorPde::a11(mx, my, tnow) * t2;
          D = 0.5*AnchorPde::b1(mx, my, tnow) * tx1;
          E = AnchorPde::b2(mx, my, tnow) * ty1;
```

```
                sgn = MySign(E);
                F = AnchorPde::d(mx, my, tnow) * delta_k;
                G = AnchorPde::F(mx, my, tnow) * delta_k;

                // Larkin + 1st order upwind
                V(i, j) = (V(i, j)*(1.0 - A) + V(i - 1, j)*(A - D)
                                + V(i+1, j)*(A + D) + sgn*E*V(i, j + sgn) + G)
                                / (1.0 + A - F + sgn*E);
            }

        }

        for (std::size_t i = 0; i < MatNew.size1() ; ++i)
        {
            for (std::size_t j = 0; j < MatNew.size2(); ++j)
            {
                MatNew(i,j) = 0.5 * (U(i,j) + V(i,j));
                U(i, j) = MatNew(i, j);
                V(i, j) = MatNew(i, j);
            }
        }
}

void result()
{ // The result of the calculation

        cout << "result";

        for (std::size_t n = 1; n < tmesh.size(); ++n)
        {
            if ((n/100)*100 == n)
            {
                cout << n << ", ";
            }

            tnow = tmesh[n];

            calculateBC();    // Calculate the BC at n+1
            calculate ();     // Calculate the solution at n+1

            tprev = tnow;
        }

}
```

An example of use is:

```
// Domain
TwoFactorPdeDomain<double> pdeDomain;
CreateAnchorPdeDomain(pdeDomain, xMax, yMax, T, payoff);
```

```
// Meshes. Create the mesh
long NX = 100; long NY = 100; long NT = 100;
cout << "Give NX \n"; cin >> NX;
cout << "Give NY \n"; cin >> NY;
cout << "Give NT \n"; cin >> NT;

std::vector<double> xmesh = pdeDomain.rx.mesh(NX);
std::vector<double> ymesh = pdeDomain.ry.mesh(NY);
std::vector<double> tmesh = pdeDomain.rt.mesh(NT);

cout << "Now creating solver\n";

TwoFactorAsianADESolver solver(pdeDomain, xmesh, ymesh, tmesh);
solver.result();
```

Then we can access the option values in `solver.MatNew` by displaying them in Excel, for example.

23.5.1 The Saul'yev Method and Factory Method Pattern

We now discuss how we applied choice A3 as mentioned in Section 23.4 to implement the original ADE method as developed in Saul'yev (1964). In this case we create a class that implements this method and that is composed of both a domain and a PDE that models equation (23.7). This design is more maintainable than the designs based on choices A1 and A2 because now the classes have tight *internal cohesion* and are *loosely coupled* with other classes. Furthermore, each class has a single major responsibility.

The class that we now discuss is almost the same as that in Section 23.5 except that in addition it contains an embedded PDE instance (in other words, we use composition):

```
class SaulyevTwoFactorAsianADESolverVersion2
{
private:
    TwoFactorPdeDomain<double> pdeDomain;           // Domain, BC, IC
    std::shared_ptr<TwoFactorAsianPde<double>> pde; // The new extra
                                                    // member

    // ETC. as in section 23.5
};

The components of this class are initialised through its constructor:
SaulyevTwoFactorAsianADESolverVersion2
(const std::shared_ptr<TwoFactorAsianPde<double>>&  asianPde,
                        const TwoFactorPdeDomain<double>& domain,
                        const std::vector<double>& xarr,
                        const std::vector<double>& yarr,
                        const std::vector<double>& tarr)
{
       // body
}
```

Since we have different ways to create a PDE we encapsulate each choice in a factory method whose sole responsibility is to create a PDE instance based on input from a given source. To this end, we consider a number of factories, one of which is (we comment out and remove the code that has already been introduced in Section 23.4):

```
std::shared_ptr<TwoFactorAsianPde<double>> CreateAsianPde(double K,
double T)
{ // Factory method to create an Asian pde for input to FDM

    // Steps
    // 1. Get input data
    // 2. Create the components of the PDE
    // 3. Return the instance of AsianPde

    double r = 0.049;

    // Values where price is calculated S/I == 1
    double S = 100; double I = 100;

    double lambda = 0.5;

    // Define payoff as a lambda function
    double xMax = 1.0;              // x far field
    double yMax = 1.0;              // y FF

    double xHotSpot = 0.5; double yHotSpot = 0.5;
    double scale1 = S*(1.0 - xHotSpot) / xHotSpot;
    double scale2 = I*(1.0 - yHotSpot) / yHotSpot;

    TwoFactorAsianPde<double> pde;

    // Build components of pde
    auto SIGMOID2 = [] (double t)
    { // t = S/A

        //

    };

    auto a11 = [=] (double x, double y, double t)
    {
        //

    };

    auto b1 = [=] (double x, double y, double t)
    {
        //
    };

    auto b2 = [=] (double x, double y, double t)
```

```
        {
            //
        };

        auto d = [=] (double x, double y, double t)
        {
            //
        };

        auto F= [=] (double x, double y, double t)
        {
            //
        };

        pde.a11 = a11;

        // a22 = c = 0

        pde.b1 = b1;
        pde.b2 = b2;
        pde.d = d;
        pde.F = F;

        return std::shared_ptr<TwoFactorAsianPde<double>>
           (new TwoFactorAsianPde<double>(pde));
    }
```

This design can be applied to a wide range of problems in which PDEs are approximated by finite difference methods. It is a combination of a structural pattern and a creational pattern, namely *Adapter* and *Factory Method*, respectively (as discussed in GOF, 1995). A summary of the design steps is:

1. Create an abstract PDE class consisting of universal function wrappers.
2. Create an FD class with an embedded pointer to an instance of the PDE class as well as an embedded domain object.
3. Create a factory method to instantiate the PDE class in step 1 and supply the newly created object in the constructor of the FD class in step 2.

In short, we use *composition* (HAS-A relationship) instead of the more pervasive IS-A relationship that characterises subtype polymorphism and class hierarchies. For example, it would be possible to implement the Barakat and Clark variant of ADE based on this design in C++. This would be a useful exercise.

23.6 USEFUL UTILITIES

We have used domain transformation to map (S, A) space to (x, y) space as discussed in Section 23.3. We then compute option prices by approximating the PDE in (x, y) space by a finite difference scheme. The resulting data structure is a matrix of discrete values in (x, y) space at expiration $t = T$. One specific scenario is to compute the option price at the hotspot $(0.5, 0.5)$ in

(x, y) space and then determine what this price is in (S, A) space. In this case we are interested in the specific values:

```
double S = 100; double I = 100;
```

But the question now is which cell of the matrix of option values to get the option value from! To answer this question we compute the pair of indices in the matrix corresponding to the hotspots and then use these indices in the matrix of option values to locate the actual price. A typical example is:

```
auto values = FindMeshValues
          (solver.xmesh, solver.ymesh, xHotSpot, yHotSpot);
std::cout << "MaxA, MaxB: " << std::get<0>(values)
          << ", " << std::get<1>(values) << endl;
```

We do not develop home-grown code to search in arrays because the functionality already exists in STL. To this end, we embed `std::upper_bound` in `FindMeshValues`:

```
std::tuple<std::size_t, std::size_t >
    FindMeshValues(const std::vector<double>& xarr,
                   const std::vector<double>& yarr,
                   double x, double y)
    { // Compute indices by searching in an array xarr for a 'threshold'
      // value x.

        // Find position of first element in vector that satisfies
        // the predicate d >= x.
        // Logarithmic complexity for random-access iterators
        auto posA = std::upper_bound(xarr.begin(), xarr.end(), x);
        auto posB = std::upper_bound(yarr.begin(), yarr.end(), y);

        auto maxA = std::distance(xarr.begin(), posA);
        auto maxB = std::distance(yarr.begin(), posB);

        return std::make_tuple(maxA, maxB);
    }
```

This is efficient code because the mesh arrays support random-access iterators. For cases in which the mesh arrays do not support random-access iterators we can use `std::find_if` but it is slower:

```
// Find position of 1st element in vector that satisfies the
// predicate d >= x.
// Linear complexity.
auto posA = std::find_if(xarr.begin(), xarr.end(), [&](double d)
             { return d >= x; });
auto posB = std::find_if(yarr.begin(), yarr.end(), [&](double d)
             { return d >= y; });
```

In general, it is recommended to use STL algorithms as building blocks when creating higher-level functionality. This avoids reinvention of the software wheel. In particular, we discussed the relevant STL algorithms in Chapters 17 and 18.

23.7 ACCURACY AND PERFORMANCE

We know that the Saul'yev scheme is *unconditionally stable* and that it is first-order accurate (Saul'yev, 1964). This is in contrast to the FTCS scheme that is also first-order accurate but only *conditionally stable.* This means that we need to choose a large number of time steps in order to avoid oscillations. Furthermore, we use *upwinding* or *downwinding* in the y variable in equation (23.7) depending on the sign of the convection coefficient. The method can be applied to the model problem:

$$\frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = 0, \quad -\infty < x < \infty, t > 0. \tag{23.10}$$

The simplest upwind scheme is the *explicit first-order upwind scheme* defined by:

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} + a\frac{u_j^n - u_{j-1}^n}{h} = 0 \text{ for } a > 0 \tag{23.11}$$

and

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} + a\frac{u_{j+1}^n - u_j^n}{h} = 0 \text{ for } a > 0. \tag{23.12}$$

It can be shown using von Neumann stability analysis (see Duffy, 2006) that this scheme is stable if the *Courant–Friedrichs–Lewy* (CFL) condition is satisfied:

$$c = \left|\frac{a\Delta t}{h}\right| \leq 1. \tag{23.13}$$

The schemes (23.11) and (23.12) are first-order accurate in space and time but they can introduce severe *numerical diffusion and dissipation* in the solution due to the presence of large gradients. In our work here we take a modified version of schemes (23.11) and (23.12), namely:

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} + a\frac{u_j^{n+1} - u_{j-1}^n}{h} = 0 \text{ for } a > 0 \tag{23.14}$$

and

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} + a\frac{u_{j+1}^n - u_j^{n+1}}{h} = 0 \text{ for } a > 0. \tag{23.15}$$

Schemes (23.14) and (23.15) could be called *semi-implicit schemes*.

We can verify (again by von Neumann stability analysis) that this scheme is unconditionally stable and hence we have ignored the constraint (23.13) in our code, which is highly

beneficial. We have incorporated schemes (23.14) and (23.15) into our C++ code in Section 23.5 (see member function `calculate()`).

We now test the accuracy of the Saul'yev scheme by examining the numerical values in Table 2 of WLD (2014) and attempting to reproduce them. We note that the results in that table were computed using the *Mathematica NDSolve* package. In general, second (or higher)-order divided differencing is applied to the spatial derivatives, resulting in a system of ordinary differential equations (ODEs) which is then solved in time by an adaptive high-order integrator such as the *Bulirsch–Stoer* method, for example. In WLD (2014) the mesh sizes NX = NY = 250 are specified. In our tests on ADE we take NX = NY = 500, NT = 2000 which makes a direct comparison between the methods somewhat difficult in a sense. However, we can get two or three-digit accuracy.

We automate the process of checking the computed values from Table 2 of WLD (2014). To this end, we first create two arrays representing expirations and strikes:

```
// Input data to the option pricer
std::vector<double> expiries = { 0.25, 1.0, 3.0, 10.0 };
std::vector<double> strikes = { 60.0, 80.0, 100.0, 120.0, 140.0 };
```

We also need to define some extra variables and objects:

```
// Values where price is calculated S/I == 1
double S = 100; double I = 100;

double xMax = 1.0;   // x far field
double yMax = 1.0;   // y FF

double xHotSpot = 0.5; double yHotSpot = 0.5;
double scale1 = S*(1.0 - xHotSpot) / xHotSpot;
double scale2 = I*(1.0 - yHotSpot) / yHotSpot;

long NX = 500; long NY = 500; long NT = 2000;
std::cout << "NX, NY, NT (put Saul'yev): "
          << NX << "," << NY << "," << NT << '\n';

TwoFactorPdeDomain<double> pdeDomain;
CreateAnchorPdeDomain(pdeDomain, xMax, yMax, expiries[t], payoff);
std::vector<double> xmesh = pdeDomain.rx.mesh(NX);
std::vector<double> ymesh = pdeDomain.ry.mesh(NY);
std::vector<double> tmesh = pdeDomain.rt.mesh(NT);
```

The following double loop computes the option price for each expiration and strike:

```
for (std::size_t k = 0; k < strikes.size(); ++k)
{
    // Define payoff as a lambda function
    int cp = -1;
    if (cp > 0)
        payoff = [=](double S, double A) -> double
           { return std::max(S - strikes[k], 0.0); }; // call
```

```
        else
            payoff = [=](double S, double I) -> double
                { return std::max(strikes[k] - S, 0.0); }; // put

        for (std::size_t t = 0; t < expiries.size(); ++t)
        {
            std::cout<<"T: "<<expiries[t]<<", K: "<<strikes[k] << ", ";

            auto pde = CreateAsianPde(strikes[k], expiries[t]);
            SaulyevTwoFactorAsianADESolverVersion2 solver(pde, pdeDomain,
            xmesh, ymesh, tmesh);
            solver.result();

            // Postprocessing: index ranges to analyse the error.
            auto values = FindMeshValues (solver.xmesh, solver.ymesh,
            xHotSpot, yHotSpot);
            cout << "price at hot spot: "
                 << solver.MatNew (std::get<0>(values), std::get<1>(values));
        }
        std::cout << '\n';
    }
```

The output from this code is shown in Table 23.1. The results are similar to those in WLD (2014).

23.8 SUMMARY AND CONCLUSIONS

In this chapter we have applied the ADE method (Barakat–Clark and Saul'yev variants) to model a special kind of path-dependent option with one stochastic factor and one deterministic factor. The resulting PDE is similar to that used when pricing Asian options and in the Cheyette model. We were able to reproduce the results in WLD (2014) that were computed using the *NDSolve* package in *Mathematica*. We have seen that even with a moderate number of mesh points (for example, in the range [200, 400]) we can achieve two- to three-digit accuracy up to 3 years' expiration and beyond. ADE is an efficient scheme (see Pealat and Duffy, 2011) and a possible project might be a feasibility study for calibration.

From a numerical analysis perspective, we have paid some attention to the problem of choosing a stable and accurate finite difference scheme to approximate the first-order hyperbolic part of the PDE defined by equation (23.7). In particular, we showed how to avoid *spurious reflection* at upstream and downstream boundaries caused by using three-point second-order approximations to the first-order derivative with respect to the averaged variable A (or y after we perform domain transformation). Many quants have learned this lesson the hard way as well.

Some other lessons learned were: (1) applying ADE to path-dependent options, (2) an analysis of numerical schemes for first-order hyperbolic PDEs and (3) using a functional programming style in C++ to design a stable software framework for the class of problems discussed in this chapter.

TABLE 23.1 Computed option values from Saul'yev ADE scheme

```
NX, NY, NT (put Saul'yev): 500,500,2000
T: 0.25, K: 60, price at hot spot: 0.000439322
T: 1, K: 60, price at hot spot: 0.169892
T: 3, K: 60, price at hot spot: 0.968049
T: 10, K: 60, price at hot spot: 1.94742

T: 0.25, K: 80, price at hot spot: 0.0431205
T: 1, K: 80, price at hot spot: 0.850115
T: 3, K: 80, price at hot spot: 2.3882
T: 10, K: 80, price at hot spot: 3.57465

T: 0.25, K: 100, price at hot spot: 2.83833
T: 1, K: 100, price at hot spot: 4.6868
T: 3, K: 100, price at hot spot: 6.05442
T: 10, K: 100, price at hot spot: 6.06422

T: 0.25, K: 120, price at hot spot: 18.5937
T: 1, K: 120, price at hot spot: 16.2815
T: 3, K: 120, price at hot spot: 13.7188
T: 10, K: 120, price at hot spot: 9.68991

T: 0.25, K: 140, price at hot spot: 38.2958
T: 1, K: 140, price at hot spot: 33.5593
T: 3, K: 140, price at hot spot: 25.4875
T: 10, K: 140, price at hot spot: 14.6057
```

23.9 EXERCISES AND PROJECTS

1. (Brainstorming Quiz 1)

We wish to determine the best way to model the input arguments to the defining functions of a given PDE. Some possible requirements are:

a) Using arguments of homogeneous and heterogeneous types (for example, some parameters are doubles while others are complex).

b) The ability to create a single PDE class that can be specialised to one-factor, two-factor and three-factor models.

c) Performance: initialising arguments, packing and unpacking them and delivering them to a function call. The challenge is mainly to determine the most suitable data types to hold the arguments and the choice between copy and move semantics, for example.

d) Maintainability: in general, we wish to avoid having to write *boilerplate code* and the dreaded *copy-and-paste* syndrome.

The objective of this exercise is to determine how far each of the following candidate solutions satisfies the above requirements:

i) Variadic functions (see Section 4.2.2).

ii) C++11 variadic parameters.

iii) Fixed-size array `std::array` class to store arguments.

iv) Hard-coded argument lists to model time and space variables (as we discussed in Section 23.2).

v) Using tuples to model input arguments ('packing and unpacking').

vi) Can you use *type traits* (Chapter 6) to promote robustness of the code?

One way to answer these questions is to create a small prototype in C++ and then measure the performance and code quality of the different solutions. For example, our guess is that solution v) has the worst performance and that solution ii) is the most maintainable (and possibly most efficient) of the candidates. In order to scope the problem you should take the heat diffusion equation in one and two dimensions as initial example. Take the diffusion coefficient to be a function of space and time.

2. (Next-Generation *Bridge* Pattern)

In Section 23.2 we implemented the class `TwoFactorPde` using universal function wrappers. In fact, it is the *abstraction* role in the *Bridge* pattern. We no longer need class hierarchies. Study the code in this chapter and then answer the following questions in order to compare the solution from Section 23.2 with those based on *subtype polymorphism* and *CRTP*:

a) Which solution involves the least amount of code writing, code bloat and proliferation of classes?

b) Which solution has the best run-time performance?

c) Which solution is the most flexible in its ability to support a range of programming styles?

3. (PDE Factory Code)

In Section 23.4 we proposed three designs to create instances of the class `TwoFactorAsianPde`. Answer the following questions:

a) Which solution leads to the most maintainable and reliable code?

b) Compare the maintainability with the solutions based on subtype polymorphism and CRTP.

c) Does the solution based on the namespace approach satisfy the single responsibility principle (SRP)? What are the main disadvantages of this approach? Are there any advantages?

d) How easy is it to reuse the code in each case?

e) Create an implementation of the *Factory Method* design pattern for the case of the Barakat–Clark variant of ADE. Test the scheme and compare the accuracy of results with those produced by the Saul'yev method.

4. (Similarity Reduction and Asian Options, Medium-Sized Project)

Closely related to the topics in this chapter is the problem of option pricing in which the payoff depends on the average price of the underlying variable over some prescribed period (Wilmott, 2006). This is the class of *Asian options* and there are several types depending on how averaging is defined and whether monitoring is continuous or discrete.

We reduce the scope by examining continuously sampled arithmetic averaging:

$$I = I(t) = \int_0^t S(\tau)\, d\tau. \tag{23.16}$$

The corresponding PDE is:

$$\frac{\partial V}{\partial t} = \frac{1}{2}\sigma^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} + S\frac{\partial V}{\partial I} - rV = 0. \tag{23.17}$$

We examine the PDE (23.17) and consider the *similarity reduction* technique by defining a new variable $R = I/S$ and a function H (Wilmott, 2006) such that:

$$V(S, R, t) = SH(R, t).$$

You can check that the function H satisfies the following PDE:

$$-\frac{\partial H}{\partial t} + \frac{1}{2}\sigma^2 R^2 \frac{\partial^2 H}{\partial R^2} + (1 - rR)\frac{\partial H}{\partial R} = 0. \tag{23.18}$$

The initial/terminal condition for H is now:

$$H(R, 0) = \frac{V(S, R, 0)}{S(0)} = g(S(0), I(0)) \tag{23.19}$$

where g is some function.
At large values of R the value of H is zero:

$$\lim_{R\to\infty} H(R, t) = 0 \tag{23.20}$$

while when $R = 0$ the PDE degenerates into the first-order hyperbolic PDE:

$$-\frac{\partial H}{\partial t} + \frac{\partial H}{\partial R} = 0. \tag{23.21}$$

The payoff is given by $H(r, 0) = I \max\left(R - \frac{1}{T}, 0\right)$.

The objective of this exercise is to approximate equations (23.18)–(23.21) using the ADE method. Answer the following questions:

a) Use domain truncation in this case as it involves less preprocessing of the PDE than with domain transformation.

b) Discretise equation (23.18) using the Saul'yev and Barakat–Clark schemes.

c) Determine how to discretise the first-order hyperbolic PDE (23.21) and how to integrate it into the full system of discrete equations.

d) Compare the boundary conditions in equations (23.20) and (23.21) with the alternative boundary conditions:

$$H(0, t) = 0,$$
$$H(R, t) \sim R \text{ as } R \to \infty.$$

What does the Fichera theory tell us about the boundary conditions for this problem?

e) Compare the approximate values produced by ADE compared to the Crank–Nicolson method and the approximation by Turnbull and Wakeman (as discussed in Haug, 2007).

f) Compare the approximate values produced by ADE with those produced by the explicit Euler (FTCS) method.

5. (Method of Lines for an Asian Option PDE)

The objective of this exercise is to discretise the PDE (23.7) in the x and y variables only. The resulting *semi-discrete scheme* is a system of ODEs. We used second-order three-point centred difference schemes in the x and y variables. We must be careful when

approximating the PDE at the boundaries $y = 0$ and $y = 1$ when using three-point centred difference schemes for first-order hyperbolic PDES (Vichnevetsky and Bowles, 1982). In particular, we are concerned with avoiding the generation of *spurious (non-physical) reflections* at $y = 0$ and $y = 1$. We avoid these problems by replacing the three-point scheme by *two-point upwind approximations*. To demonstrate, consider PDE (23.10) and its semi-discrete approximation:

$$\frac{du_j}{dt} + a\left(\frac{u_{j+1} - u_{j-1}}{2h}\right) = 0, \quad 1 \leq j \leq J - 1. \tag{23.22}$$

We need to be very careful at the boundary points $j = 0$ and $j = J$. In this case we can use the *upwind approximations*:

$$\begin{aligned} \frac{du_0}{dt} + a\left(\frac{u_1 - u_0}{h}\right) &= 0 \\ \frac{du_J}{dt} + a\left(\frac{u_J - u_{J-1}}{h}\right) &= 0. \end{aligned} \tag{23.23}$$

This scheme has truncation error of order $0(h)$. Despite this low-order accuracy the global order accuracy of the semi-discrete scheme (23.22) is still second order.

Answer the following questions:

a) Consider using the improved second-order treatment of the exit boundary points, for example at $j = J$:

$$\frac{du_J}{dt} + a\left(\frac{3u_J - 4u_{J-1} + u_{J-2}}{2h}\right) = 0. \tag{23.24}$$

b) Investigate the application of the Fichera theory (see Chapter 20, Section 20.5) as a technique to determine the boundary conditions for this problem.

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 24

Ordinary Differential Equations and their Numerical Approximation

24.1 INTRODUCTION AND OBJECTIVES

In this chapter we introduce first-order *ordinary differential equations* (ODEs) and their numerical approximation in C++. In general, it is difficult to find an analytical solution to these problems and for this reason we use the finite difference method to compute an approximate solution. There is a vast literature on this area of numerical analysis. We scope this domain and we attempt to do justice to the following topics:

- A1: What is an ODE, does it have a solution in a given interval and is the solution unique?
- A2: Qualitative properties of the solution of an ODE.
- A3: Linear and nonlinear scalar ODEs and systems of ODEs.
- A4: Numerical solution of ODEs.
- A5: An introduction to the Boost C++ *odeint* library for solving ODEs.
- A6: Some applications of ODEs in computational finance.

We discuss a number of these topics in this chapter. We continue with more advanced cases and applications in Chapter 25. Some relevant literature is Henrici (1962), Brauer and Nohel (1969), Conte and de Boor (1981) and Lambert (1991), to mention just a few. In general, each reader will have her own personal preferences as to which books are most relevant.

24.2 WHAT IS AN ODE?

An ODE is an equation that defines a relationship involving the derivatives of an unknown function y with respect to a single variable. The highest-order derivative in the equation determines the *order* of the equation. An example of a scalar linear nth-order ODE is given by:

$$\sum_{j=0}^{n} a_j(x)\frac{d_y^j}{dx^j} = f(x) \tag{24.1}$$

where $f(x)$ and $a_j(x)$ are functions of $x, j = 0, \ldots, n$ with $a_n(x) \neq 0$.

There are many solutions to this problem in general. In order to specify a possible *unique solution* we give *n initial conditions* at $x = 0$:

$$\frac{d^j y(0)}{dx^j} = \alpha_j, \text{ where } \alpha_j \text{ is a given constant, } j = 0, \ldots, n-1.$$

In general, an ODE is defined in some interval and we will need theorems to ensure the existence of a unique solution of the ODE in this interval. In this chapter we are mainly interested in *first-order equations* ($n = 1$) and to a lesser extent in *second-order equations* ($n = 2$). We can always reduce a scalar second-order ODE to a 2×2 system of first-order ODEs.

We take an example of a linear *n*th-order *homogeneous* ODE called the *Euler–Cauchy equation*:

$$\sum_{j=0}^{n} a_j x^j \frac{d^j y}{dx^j} = 0, \text{ where } \mathrm{a}_j \text{ is a given constant, } j = 0, \ldots, n. \tag{24.2}$$

The substitution $x = e^u$ reduces this equation to a linear ODE in the independent variable with constant coefficients. Alternatively, trial solutions $y = x^m$ (where m is a constant to be determined) may be used to solve these kinds of equations.

24.3 CLASSIFYING ODEs

We can classify ODEs according to various criteria, for example:

- C1: Linear and nonlinear ODEs.
- C2: Scalar ODEs and systems of ODEs.
- C3: First-order/higher-order ODEs.
- C4: Equations with special forms.

Equation (24.1) is an example of a *linear equation* because the coefficients are independent of the unknown solution y, which is called the *dependent variable*, and the derivative terms are linear. The following ODEs are examples of nonlinear scalar ODEs:

$$y' = xy - \sin y, y(0) = 2$$
$$y' = 3y^{2/3}, y(2) = 0$$
$$y' = \frac{2y}{x}, y(x_0) = y_0, \text{ for some } x_0.$$

Systems of ODEs involve two or more dependent variables. For example, consider the scalar second-order ODE with initial condition:

$$a_0 y'' + a_1 y' + by = 0$$
$$y(0) = A, y'(0) = B_0.$$

By a change of variables we see that this ODE can be converted to a 2×2 system of first-order ODEs:

$$a_0 v' + a_1 v + by = 0, \; y' = v$$
$$y(0) = A, \; v(0) = B_0.$$

We conclude this short overview with a discussion of a special class of ODEs. To this end, we say that a function $f(x, y)$ is *homogeneous of degree m* if:

$$f(\lambda x, \lambda y) = \lambda^m f(x, y),\ m \geq 0.$$

Here m is a constant and λ is a real number. The ODE $M(x, y)dx + N(x, y)dy = 0$ is said to be *homogeneous* if $M(x, y)$ and $N(x, y)$ are homogeneous and of the same degree. An example of a homogeneous ODE is $(x^2 - y^2)dx + 2xydy = 0$ (degree 2).

24.4 A PALETTE OF MODEL ODEs

In this section we give a number of typical examples of first-order ODEs. These are of interest in their own right and they occur as part of larger problems in computational finance. Furthermore, understanding them will help you gain insights into other problems and they can also be used as test cases for numerical methods that approximate the solution of ODEs.

24.4.1 The Logistic Function

A *logistic function* (or *logistic curve*) is an S-shaped sigmoid curve defined by the equation:

$$f(x) = \frac{L}{1 + e^{-k(x-x_0)}} \tag{24.3}$$

where:

$$\begin{aligned} x_0 &= x\text{-value of sigmoid's midpoint} \\ L &= \text{curve's maximum value} \\ k &= \text{steepness of the curve.} \end{aligned}$$

A special case is when $k = 1, x_0 = 0, L = 1$ resulting in the *standard logistic function* defined by the equation:

$$f(x) = \frac{1}{1 + e^{-x}}. \tag{24.4}$$

We can verify from equation (24.4) that the logistic function satisfies the *nonlinear initial* value problem:

$$\frac{df}{dx} = f(1 - f), f(0) = \frac{1}{2}. \tag{24.5}$$

The logistic function models processes in a range of fields such as artificial neural networks (learning algorithms where it is called an *activation function*), economics, probability and statistics, to name a few. We recall that the volatility function in Chapter 23 was a kind of logistic function.

A special case of equation (24.4) is the *threshold signal function* defined by:

$$S(x^{k+1}) = \begin{cases} 1 \text{ if } x > T \\ S(x^k) \text{ if } x = T \\ 0 \text{ if } x < T \end{cases} \tag{24.6}$$

for some arbitrary real-valued threshold T and where the index k denotes the discrete time step. This function transduces activations to signals. When the activation reaches the threshold at time $k + 1$, the signal maintains the same value that it had at time k.

24.4.2 Bernoulli Differential Equation

This ODE is named after Jacob Bernoulli. It is special in the sense that it is a nonlinear equation having an exact solution:

$$y' + P(x)y = Q(x)y^n \quad (n \neq 0, n \neq 1). \tag{24.7}$$

In the cases $n = 0$ and $n = 1$ and when P and Q are constants this equation is linear and an exact solution can then be found. We reduce equation (24.7) to a linear one by the substitution $v = y^{1-n}(v' = (1 - n)y^{-n}y')$ to produce the following linear ODE:

$$v' + (1 - n)Pv = (1 - n)Q. \tag{24.8}$$

24.4.3 Riccati Differential Equation

This is a nonlinear ODE of the form:

$$y' = P(x) + Q(x)y + R(x)y^2 + N(x, y). \tag{24.9}$$

This ODE has many applications, for example to interest-rate models (Duffie and Kan, 1996). In some cases a closed-form solution to equation (24.9) is possible, but in this book our focus is on approximating it using the finite difference method.

We now discuss the relationship between the Riccati equation and the pricing of a *zero-coupon bond* $P(t, T)$, which is a contract that offers one dollar at maturity T. By definition, an *affine term structure* model assumes that $P(t, T)$ has the form:

$$P(t, T) = \exp[A(t, T) - B(t, T)r(t)]. \tag{24.10}$$

Let us assume that the short-term interest rate is described by the following SDE:

$$dr = \mu(t, r)dt + \sigma(t, r)dW_t \tag{24.11}$$

where W_t is a standard Brownian motion under the risk-neutral equivalent measure and μ and σ are given functions.

Duffie and Kan proved that $P(t,T)$ is *exponential affine* if and only if the drift μ and volatility σ have the form:

$$\mu(t,r) = \alpha(t)r + \beta(t), \quad \sigma(t,r) = \sqrt{\gamma(t) + \delta(t)} \tag{24.12}$$

where $\alpha(t), \beta(t), \gamma(t)$ and $\delta(t)$ are given functions of t.

Moreover, the coefficients $A(t,T)$ and $B(t,T)$ are determined by the following ODEs:

$$\frac{dB}{dt} = \frac{\gamma(t)}{2}B(t,T)^2 - \alpha(t)B(t,T) - 1, \quad B(T,T) = 0 \tag{24.13}$$

and

$$\frac{dA}{dt} = \beta(t)B(t,T) - \frac{\delta(t)}{2}B(t,T)^2, \quad A(T,T) = 0. \tag{24.14}$$

The first equation (24.13) for $B(t,T)$ is the Riccati equation and the second one (24.14) is solved easily from the first by integration.

24.4.4 Population Growth and Decay

ODEs can be used as simple models of population growth, for example assuming that the rate of reproduction of a population size P is proportional to the existing population and to the amount of available resources. The ODE is:

$$\frac{dP}{dt} = rP\left(1 - \frac{P}{K}\right), \quad P(0) = P_0 \tag{24.15}$$

where r is the *growth rate* and K is the *carrying capacity*. The initial population is P_0. It is easy to check the following identities:

$$P(t) = \frac{KP_O e^{rt}}{K + P_0(e^{rt} - 1)} \text{ and } \lim_{t\to\infty} P(t) = K. \tag{24.16}$$

Transformation of this equation leads to the logistic ODE:

$$\frac{dn}{dr} = n(1 - n) \tag{24.17}$$

where n is the population in units of carrying capacity ($n = P/K$) and r measures time in units of $1/r$. Equation (24.17) has the same form as equation (24.5).

For systems, we can consider the *predator–prey* model in an environment consisting of foxes and rabbits:

$$\begin{aligned} \frac{dr(t)}{dt} &= ar(t)f(t) + br(t) \\ \frac{df(t)}{dt} &= pf(t) + qf(t)r(t) \end{aligned} \tag{24.18}$$

where:

$r(t)$	=	number of rabbits at time t
$f(t)$	=	number of foxes at time t
$br(t)$	=	birth rate of rabbits
$-ar(t)f(t)$	=	death rate of rabbits
b	=	unit birth rate of rabbits
$-pf(t)$	=	death rate of foxes
$qf(t)r(t)$	=	birth rate of foxes
q	=	unit birth rate of foxes.

This ODE system is a model of a closed ecological environment in which foxes and rabbits are the only kinds of animals. Rabbits eat grass (of which there is a constant supply), procreate and are eaten by foxes. All foxes eat rabbits, procreate and die of geriatric diseases.

System (24.18) is sometimes called the *Lotka–Volterra equations*, which are an example of a more general *Kolmogorov model* to model the dynamics of ecological systems with predator–prey interactions, competition, disease and mutualism (Lotka, 1956).

24.5 EXISTENCE AND UNIQUENESS RESULTS

We have given a number of examples of ODEs and we can find analytical or numerical solutions in some cases. We are interested in finding necessary and sufficient conditions for an ODE with an associated initial condition to have a unique solution in a given interval. We first consider linear scalar first-order ODEs because they have analytical solutions and furthermore they can be used as test input to numerical schemes to determine the latter's stability and accuracy properties.

Consider a bounded interval $[0, T]$, where $T > 0$. This interval could represent time or distance, for example, but in most cases we shall view this interval as representing time values. In this interval we define the *initial value problem* (IVP):

$$\left.\begin{aligned} &Lu \equiv u'(t) + a(t)u(t) = f(t), \quad t \in [0, T] \\ &u(0) = A,\ a(t) \geq \alpha > 0, \quad \forall t \in [0, T] \end{aligned}\right\} \tag{24.19}$$

where L is a first-order linear operator involving the derivative of u with respect to the time variable and $a = a(t)$ is a strictly positive function in $[0, T]$. The term $f(t)$ is called the *inhomogeneous forcing* term and it is independent of u. Finally, the solution to the IVP must be specified at some initial point, for example $t = 0$; this is the *initial condition*. In general, the problem (24.19) has a unique solution given by:

$$u(t) = I_1(t) + I_2(t) \tag{24.20}$$

where:

$$I_1(t) = A \exp\left(-\int_0^t a(s)ds\right),\ I_2(t) = \exp\left(-\int_0^t a(s)ds\right)\int_0^t \exp\int_0^x a(s)ds\, f(x)dx.$$

(see Hochstadt, 1964 where the *integration factor method* is used to find a solution).

A special case of system (24.19) is when the right-hand term $f(t)$ is zero and $a(t)$ is constant; in this case the solution becomes an exponential term and it is used when we examine difference schemes to determine their stability properties. In particular, a scheme that behaves badly for this special case will be unsuitable for more general or more complex problems unless some modifications are introduced.

We discuss a number of results that allow us to describe how the solution u of system (24.19) behaves. First, we claim that if the initial value A and inhomogeneous term $f(t)$ are positive, then the solution $u(t)$ should also be positive for any value t in $[0, T]$. This *positivity property* should be reflected in our difference schemes (unfortunately, not all schemes possess this property). Second, we wish to know how the solution $u(t)$ grows or decreases as a function of time. The following lemma and theorem deal with these issues.

Lemma 24.1 (Positivity): Let w be a well-behaved function satisfying the inequalities:

$$Lw(t) \geq 0 \ \forall t \in [0, T] \ \text{ with } \ w(0) \geq 0. \tag{24.21}$$

Then we have the following results:

$$w(t) \geq 0, \ \forall t \in [0, T]. \tag{24.22}$$

This lemma implies that you cannot get a negative solution from positive input. You can verify this result by examining equation (24.20) because all terms are positive.

The following results give bounds on the growth of $u(t)$.

Theorem 24.1 Let $u(t)$ be the solution of system (24.19). Then:

$$|u(t)| \leq N/\alpha + |A|, \ \forall t \in [0, T]$$

where: (24.23)

$$|f(t)| \leq N, \quad \forall t \in [0, T].$$

This result states that the value of the solution is bounded by the input data. We wish to replicate these properties in difference schemes for (24.19).

We now turn our attention to a more general initial value problem for a *nonlinear system* of ODEs:

$$\begin{cases} y' = f(t, y), & t \in \mathbb{R} \\ y(0) = A \end{cases} \tag{24.24}$$

where:

$$y : \mathbb{R} \to \mathbb{R}^n, \quad A \in \mathbb{R}^n, \ f : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$$

and

$$f(t, y) = (f_1(t, y), \ldots, f_n(t, y))^T \quad \text{where } f_j : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}, j = 1, \ldots, n.$$

Let B be a region of $n+1$-dimensional space and let f be continuously differentiable with respect to t and with respect to all the components of y at all points of B. We assume that the following inequalities hold:

$$\left|\frac{\partial f}{\partial y_j}(t,y)\right| \leq K \text{ for some } K > 0 \quad j = 1, \ldots, n \tag{24.25}$$

and

$$|f(t,y)| \leq M \text{ for some } M > 0. \tag{24.26}$$

Theorem 24.2 Let f and $\frac{\partial f}{\partial y_j}(j = 1, \ldots, n)$ be continuous in the box $B = \left\{(t,y) : \left|t - t_0\right| \leq a, |y - \eta| \leq b\right\}$ where a and b are positive numbers, satisfying the bounds (24.25) and (24.26) for (t,y) in B. Let α be the smaller of the numbers a and b/M and define the successive approximations:

$$\begin{aligned} \phi_0(t) &= \eta \\ \phi_n(t) &= \eta + \int_{t_n}^{t} f(s, \phi_{n-1}(s))ds, n \geq 1. \end{aligned} \tag{24.27}$$

Then the sequence $\left\{\phi_j\right\}$ of successive approximations ($j \geq 0$) converges (uniformly) on the interval $\left|t - t_0\right| \leq a$ to a solution $\phi(t)$ of (24.24), satisfying the initial condition $\phi(t_0) = \eta$.

The method (24.27) is called the *Peano iterative method* and it is used to prove existence of the solution of systems of ODEs (24.24). It is mainly of theoretical value as it should not necessarily be seen as a practical way to construct a solution. On the other hand, it is a useful exercise to construct the sequence of iterates in equation (24.27) for some simple (mostly scalar linear) cases.

24.5.1 A Test Case

We take the autonomous nonlinear scalar ODE:

$$y' = f(y) = y^2, y(t_0) = a \tag{24.28}$$

whose solution is given by:

$$y(t) = \frac{a}{1 - a(t - t_0)}. \tag{24.29}$$

We now compute the *Picard iterates* (24.27) for this ODE in order to determine the values of t for which the ODE has a solution. For convenience, let us take $a = 1,\ t_0 = 0$. Some simple integration shows that:

$$\begin{aligned}
\phi_1(t) &= 1 \\
\phi_1(t) &= 1 + \int_0^t f(\phi_0)dt = 1 + t \\
\phi_2(t) &= 1 + \int_0^t f(\phi_1)dt = 1 + t + t^2 + t^3/3 \\
\phi_3(t) &= 1 + t + t^2 + t^3 + \frac{2t^4}{3} + \frac{t^5}{3} + \frac{t^6}{9} + \frac{t^7}{63}.
\end{aligned} \tag{24.30}$$

We can see that the series is beginning to look like $\frac{1}{1-t} = \sum_{j=0}^{\infty} t^j$. We know that this series is convergent for $|t| < 1$. A nice exercise is to compute the Picard iterates in the most general case (that is, $a \neq 1,\ t_0 \neq 0$) and to determine under which circumstances the ODE (24.28) has a solution. In this case we have converted the solution of an ODE to a series and then analysed the latter, for which there are many convergence results such as the *root test* and the *ratio test*.

24.6 OVERVIEW OF NUMERICAL METHODS FOR ODEs: THE BIG PICTURE

We now discuss how to approximate the solution of the initial value problem (24.24) by the finite difference method. The main attention points are (Lambert, 1991):

- A1: One-step and multistep methods.
- A2: Explicit and implicit methods.
- A3: Adaptive methods and step-size control.
- A4: Numerical methods for special problems, for example *stiff ODEs*.

In this chapter we are mainly interested in one-step methods for *non-stiff* ODEs. We also reduce the scope to those numerical methods that use constant time steps, although we shall see how the Boost C++ *odeint* library supports adaptive time stepping.

24.6.1 Mapping Mathematical Functions to C++

Before we discuss numerical methods for ODEs and their implementations in C++ we introduce a number of user-defined data structures when modelling ODEs in the form of equation (24.24). In general, we model *real-valued functions* (map a vector to a `double`) and *vector-valued functions* (map a vector to a vector). In other applications (for example, minimisation and optimisation problems) we may also wish to model *scalar-valued functions* (map a `double` to a double) and possibly *vector functions* (map a `double` to a vector). We modelled these function types in Duffy (2004) using C++03 but since C++11 we can avail of

universal function wrappers to achieve the same end in a more elegant way. We have decided to use `std::vector` as the underlying array structure. In a later version we could take a more generic approach, including one that also supports `std::array`.

The two classes of functions that we support in this chapter are real-valued and vector-valued functions. We reduce the scope by focusing on `double` but it can easily be generalised to other arithmetic types. The data types have aliases to make client code more readable:

```
using RealValuedFunction = std::function
          <double (double x, const std::vector<double>& y)>;

using VectorValuedFunction
       = std::vector< RealValuedFunction >;
```

In other words, `RealValuedFunction` maps arrays to scalars while `VectorValuedFunction` is an array of `RealValuedFunction` instances.

The first example is:

```
auto f = [](double t, const std::vector<double>& y)
          { return t + y[0]*(1.0 - y[1]); };

double t = 1.0;
std::vector<double> y(2); y[0] = 0.5; y[1] = 0.5;
RealValuedFunction rvf = f;
std::cout << "Real-valued function: " << rvf(t, y) << '\n';
```

The second example comes from equation (24.51), as we shall see in Section 24.6.6:

```
double lambda1 = -100.1; double lambda2 = -0.1;
double a = (lambda1 + lambda2) / 2; double b = (lambda1 - lambda2) / 2;
VectorValuedFunction func(2);

// Component functions of fun
RealValuedFunction f1 = [=](double x, std::vector<double> y)
             { return a*y[0] + b*y[1]; };
RealValuedFunction f2 = [=](double x, std::vector<double> y)
             { return b*y[0] + a*y[1]; };
func[0] = f1; func[1] = f2;
```

This approach is similar to that taken in the Boost C++ *odeint* library that we shall discuss in Chapter 25.

We have written some code to compute the *max norm* of an array and the maximum distance between two arrays. These *utility functions* are useful when employing iterative methods such as the predictor–corrector method when we wish to test for convergence, for example:

```
// Utilities
double LInfinityNorm(const std::vector<double>& arr)
{
```

```
    return *(std::max_element(arr.begin(), arr.end()));
}

double LInfinityNorm(const std::vector<double>& arr1,
         const std::vector<double>& arr2)
{

    double maxV = std::abs(arr1[0] - arr2[0]);

    for (std::size_t j = 1; j < arr1.size(); ++j)
    {
       maxV = std::max(maxV, std::abs(arr1[j] - arr2[j]));
    }

    return maxV;
}
```

24.6.2 Runge–Kutta Methods

In this section we introduce a class of one-step methods to approximate the solution of system (24.24).

The first step is to replace continuous time by discrete time. To this end, we divide the interval $[0, T]$ into a number of subintervals. We define $N + 1$ *mesh points* as follows:

$$0 = t_0 < t_1 < \cdots < t_n < t_{n+1} < \cdots < t_N = T.$$

In this case we define a set of subintervals (t_n, t_{n+1}) of size $\Delta t_n \equiv t_{n+1} - t_n, 0 \leq n \leq N - 1$.

In general, we speak of a *non-uniform mesh* when the sizes of the subintervals are not necessarily the same. However, in this book we consider a class of finite difference schemes where the N subintervals have the same length (we then speak of a *uniform mesh*), namely $\Delta t = T/N$. The variable $h = T/N$ is also used to denote the uniform *mesh size*.

In general, we define y_n to be the approximate solution at time t_n and we write the functional dependence of y_{n+1} on t_n, y_n and h by:

$$y_{n+1} - y_n = h\Phi(t_n, y_n, h) \tag{24.31}$$

where Φ is called the *increment function*. For example, in the case of the *explicit Euler method* this function is:

$$\Phi(t, y, h) = f(t, y). \tag{24.32}$$

The increment function represents the increment of the approximate solution. In general, the goal is to produce a formula for Φ that agrees with a certain exact relative increment with an error of $0(h^p)$ where $p > 1$ without making it necessary to compute the derivative of f (Henrici, 1962). A special case of (24.31) is the *fourth-order Runge–Kutta method*:

$$\Phi(t, y, h) = \frac{1}{6}[k_1 + 2k_2 + 2k_3 + k_4] \tag{24.33}$$

where:

$$
\begin{aligned}
k_1 &= f(t, y) \\
k_2 &= f(t + h/2, y + \tfrac{1}{2}hk_1) \\
k_3 &= f(t + h/2, y + \tfrac{1}{2}hk_2) \\
k_4 &= f(x + h, y + hk_3).
\end{aligned}
$$

Other methods are:

Second-order Ralston method:

$$
\begin{aligned}
y_{n+1} &= y_n + \tfrac{1}{4}(k_1 + 3k_2) \\
k_1 &= hf(t_n, y_n) \\
k_2 &= hf(t_n + 2h/3, y_n + 2k_1/3).
\end{aligned}
\tag{24.34}
$$

Heun's (improved Euler) method:

$$
\begin{aligned}
\tilde{y}_{n+1} &= y_n + hf(t_n, y_n) \\
y_{n+1} &= y_n + \tfrac{h}{2}[f(t_n, y_n) + f(t_{n+1}, \tilde{y}_{n+1})].
\end{aligned}
\tag{24.35}
$$

This is a *predictor–corrector method* that also has applications to stochastic differential equations.

In general, explicit Runge–Kutta methods are unsuitable for stiff ODEs.

24.6.3 Richardson Extrapolation Methods

Extrapolation procedures are usually applied to partial differential equations but they can also be applied to Runge–Kutta methods for ODE systems. Let us assume that we are using a Runge–Kutta method to approximate the solution of system (24.24). Then on meshes of size h and $h/2$ we have the following error estimates:

$$
\begin{aligned}
y^h &= y + Mh^{p+1} + 0(h^{p+2}) \\
y^{h/2} &= y + M\left(\frac{h}{2}\right)^{p+1} + 0(h^{p+2}).
\end{aligned}
\tag{24.36}
$$

In (24.36) the constant M is independent of h but depends on y.

Some basic mathematics allows us to improve the accuracy. We define the mesh function $w^{h/2}$ by:

$$
w^{h/2} = \frac{2^{p+1}u^{h/2} - u^h}{2^{p+1} - 1} = u + 0(h^{p+2}).
\tag{24.37}
$$

For example, extrapolation can be applied to the Ralston method (24.34) in the case $\rho = 1$.

24.6.4 Embedded Runge–Kutta Methods

These are Runge–Kutta methods that contain another method and together they produce an estimate of the local truncation error of a single Runge–Kutta step. Popular embedded methods are *Fehlberg*, *Cash–Karp* (Cash and Karp, 1990) and *Dormand–Prince*, which are supported in the Boost C++ *odeint* library. In particular, the Cash–Karp method is suitable for ODEs having solutions with sharp fronts or discontinuities where a low-order solution often has very good accuracy while a higher-order solution has very poor accuracy.

24.6.5 Implicit Runge–Kutta Methods

The Runge–Kutta method (24.33) is explicit in the sense that all unknown terms are on the left-hand side of the equations and all terms on the right-hand side are known. Thus, the solution can be computed without the need to solve a matrix or nonlinear system. Implicit methods do not enjoy this feature.

It can be shown that the solution of system (24.24) satisfies the nonlinear *Volterra integral equation of the second kind* (Haaser and Sullivan, 1991) defined by:

$$y(y) = y(t_0) + \int_{t_0}^{t} f(s, y(s))ds, \quad t \in I \tag{24.38}$$

where I is some interval.

We can use the *mean-value theorem for integrals* (Widder, 1989) to show that there is a point $(\tilde{t}, \tilde{y}),\ t_0 < \tilde{t} < t, y(t_0) < \tilde{y} < y(t)$ such that:

$$y(t) = y(t_0) + hf(\tilde{t}, \tilde{y}) \tag{24.39}$$

where $h = t - t_0$ or in parametric form:

$$y(t) = y(t_0) + hf(t_0 + \theta h, y(t_0) + \psi(y(t - y(t_0)), \quad 0 \leq \theta, \psi \leq 1. \tag{24.40}$$

Some examples are the implicit *Euler method* $\theta = \psi = 1$:

$$y(t) = y(t_0) + hf(t, y(t)),\ h = t - t_0 \tag{24.41}$$

and the *implicit midpoint* rule $\theta = \psi = 1/2$:

$$\begin{aligned} &y(t) = y(t_0) + hk_1 \\ &\text{where:} \\ &k_1 = f\left(t_0 + \tfrac{h}{2}, y(t_0) + \tfrac{h}{2}k_1\right) \end{aligned} \tag{24.42}$$

or in discrete notation $y_{n+1} = y_n + hf(t_n + \frac{h}{2}, \frac{1}{2}(y_n + y_{n+1}))$.

Another approach is the *Radau scheme*, integrating equation (24.38):

$$y(t) - y(t_0) = \int_{t_0}^{t} f(s, y(s))ds \approx \tfrac{h}{4}(f(t_0, y(t_0)) + 3f(t_0 + \tfrac{2}{3}h, y(t_0 + \tfrac{2}{3}h))). \tag{24.43}$$

We use quadratic interpolation to produce the scheme:

$$y(t_0 + \tfrac{2}{3}h) \approx \tfrac{5}{9}y(t_0) + \tfrac{4}{5}y(t) + \tfrac{2}{9}hf(t_0, y(t_0)) \tag{24.44}$$

resulting in the Hammer and Hollingsworth scheme:

$$\begin{aligned} k_1 &= f(t_0, y_0) \\ k_2 &= f(t_0 + \tfrac{2}{3}h, t_0 + \tfrac{h}{3}(k_1 + k_2)) \\ y_1 &= y(t_0) + \tfrac{h}{4}(k_1 + 3k_2). \end{aligned} \tag{24.45}$$

A detailed discussion of implicit and semi-implicit methods can be found in Lambert (1991). These methods are expensive to implement and they cannot rival predictor–corrector or explicit Runge–Kutta methods when the problem to be solved is not stiff.

24.6.6 Stiff ODEs: An Overview

What is a *stiff* ODE? Before we define the term, let us give some examples of *nasty* ODEs; that is, ODEs that cause problems for standard finite difference schemes.

The presence of different time scales in ODEs leads to a number of challenges when approximating them using standard finite difference schemes. In particular, schemes such as explicit and implicit Euler, Crank–Nicolson and predictor–corrector do not approximate these systems well, unless a prohibitively small time step is used. Let us take the example (Dahlquist and Björck, 1974):

$$\frac{dy}{dt} = 100(\sin t - y), y(0) = 0 \tag{24.46}$$

with exact solution:

$$y(t) = \frac{\sin t - 0.01\cos t + 0.01e^{-100t}}{1.0001}. \tag{24.47}$$

This is a stiff problem because of the different time scales, as can be seen by the presence of the exponential term (quickly decaying) in the solution. The other terms do not decay as quickly and they represent long-term behaviour.

We carried out an experiment using the explicit Euler method and we had to divide the interval [0, 3] into 1.2 million subintervals in order to achieve accuracy to three decimal places. The implicit Euler and Crank–Nicolson methods are not much better.

We now take the example of the following 2×2 systems (Lambert, 1991):

$$\begin{aligned} \frac{dy_1}{dt} &= 2y_1 + y_2 + 2\sin x \\ \frac{dy_2}{dt} &= y_1 - 2y_1 + 2(\cos x - \sin x) \\ y_1 &= 2, y_2(0) = 3 \end{aligned} \tag{24.48}$$

and

$$\begin{aligned} \frac{dy_1}{dt} &= -2y_1 + y_2 + 2\sin x \\ \frac{dy_2}{dt} &= 998y_1 - 999y_2 + (\cos x - \sin x) \\ y_1(0) &= 2, y_2(0) = 3. \end{aligned} \tag{24.49}$$

Both problems have the same exact solutions, namely:

$$\begin{aligned} y_1(t) &= 2e^{-t} + \sin t \\ y_2(t) &= 2e^{-t} + \cos t. \end{aligned} \tag{24.50}$$

In this case we say that problem (24.48) is *non-stiff* and problem (24.49) is *stiff*. For the latter case many methods (for example, the fourth-order Runge–Kutta method) can only solve it at the cost of cutting the step size down to an unacceptably small value while these methods have less difficulty with problem (24.48), in which case the methods approximate this ODE system well even using fairly large average step size.

We take another 2×2 system as an example from Stoer and Bulirsch (1980, p. 462):

$$\begin{aligned} \frac{dy_1}{dt} &= \frac{\lambda_1 + \lambda_2}{2} y_1 + \frac{\lambda_1 - \lambda_2}{2} y_2 \\ \frac{du_2}{dt} &= \frac{\lambda_1 - \lambda_2}{2} y_1 + \frac{\lambda_1 + \lambda_2}{2} y_2 \end{aligned} \tag{24.51}$$

with constants λ_1 and λ_2 with $\lambda_1, \lambda_2 < 0$.

The solution of this system is given by:

$$\left.\begin{aligned} y_1(t) &= c_1 e^{\lambda_1 t} + c_2 e^{\lambda_2 t} \\ y_2(t) &= c_1 e^{\lambda_1 t} - c_2 e^{\lambda_2 t} \end{aligned}\right\} t \geq 0 \tag{24.52}$$

where c_1 and c_2 are constants of integration.

It is possible to compute a 'closed solution' based on the explicit Euler method:

$$\begin{aligned} Y_{1j} &= c_1(1 + h\lambda_1)^j + c_2(1 + h\lambda_2)^j \\ Y_{2j} &= c_1(1 + h\lambda_1)^j - c_2(1 + h\lambda_2)^j. \end{aligned} \tag{24.53}$$

The exact solution (24.52) converges to zero for large j. The corresponding discrete solution (24.53) also converges to zero for large values of the index j if $|1 + h\lambda_j| < 1, j = 1, 2$ or equivalently $h < 2/|\lambda_2|$ in the case when $|\lambda_2| >> |\lambda_1|$. For example, if $\lambda_1 = -1,\ \lambda_2 = -1000$, then we must have $h < 0.002$ even though e^{-1000t} contributes almost nothing to the solution.

Summarising, we can characterise *stiffness* in a number of ways:

Significant difficulties can occur when standard numerical techniques are applied to approximate the solution of a differential equation when the exact solution contains terms of the form $e^{\lambda t}$, where λ is a complex number with negative real part.

Problems involving rapidly decaying transient solutions occur naturally in a wide variety of applications, including the study of spring and damping systems, the analysis of control systems, and problems in chemical kinetics. These are all examples of a class of problems called stiff (mathematical stiffness) systems of differential equations, due to their application in analysing the motion of spring and mass systems having large spring constants (physical stiffness).

Other characterisations of stiffness are (Lambert, 1991):

1. A linear constant coefficient system is stiff if all of its eigenvalues have negative real part and the *stiffness ratio* (the quotient of the largest and smallest eigenvalues) is large.
2. Stiffness occurs when stability requirements, rather than those of accuracy, constrain the step length.
3. Stiffness occurs when some components of the solution decay much more rapidly than others.

24.7 CREATING ODE SOLVERS IN C++

We now discuss how to model scalar ODEs, systems of ODEs as well as their numerical approximation in C++. Some of the advantages of writing our own code at this stage are:

- We get practical hands-on experience of ODEs and their numerical approximation. This makes the transition to understanding ODE solvers such as Boost *odeint* all the easier.
- You can test the accuracy of a scheme with minimum fuss (you don't need to install and learn how to use a library).
- We use features in C++11 to write code having a small semantic distance from the mathematical formulation and the numerical algorithms that it realises. In particular, universal function wrappers and lambda functions help bridge the gap.
- More generally, knowing ODE theory and how to approximate ODEs are useful skills that are needed when studying stochastic differential equations (SDEs) and partial differential equations (PDEs).

We now show how we implemented a number of the numerical schemes that we have discussed in this chapter. We have implemented each scheme as a free function. In the scalar case the interface is:

```
// y' = f(x,y)
double AnyMethod(const std::function<double(double x, double y)>& f,
                 double x, double y0,      // Initial x and y0
                 double T,                 // End point of interval [0, T]
                 std::size_t N,            // Number of subdivisions
                 double TOL)               // Prescribed stopping
    { // Some method

    // Code
    }
```

and in the system case it is:

```
// y' = f(x,y)
std::vector<double> AnyMethod(const VectorValuedFunction& f, // y' = f(x,y)
                      double x, std::vector<double> y,     // Initial x and y
                      double T,               // End point of interval [x, T]
                      std::size_t N,          // Number of subdivisions
                      double TOL)             // Prescribed stopping
{ // Some method

    // Code
}
```

In general, the code has been written with readability in mind and no attempt has been made at this stage to optimise its run-time performance.

24.7.1 Explicit Euler Method

This scheme implements equation (24.41) and it is the simplest finite difference scheme to approximate the solution of system (24.24). The code for scalar and systems cases is:

```
// y' = f(x,y)
double Euler(const std::function<double(double x, double y)>& f,
             double x, double y,           // Initial x and y
             double T,                     // End point of interval [x, T]
             std::size_t N,                // Number of subdivisions
             double TOL)                   // Prescribed stopping
{                                          // Explicit Euler method

double h = (T - x) / static_cast<double>(N);

// Variables
double ynP1;   // y(n+1)

for (std::size_t n = 1; n <= N; ++n)
{
    ynP1 = y + h*f(x,y);
    // Go to next level
    x += h;
    y = ynP1;
}

return ynP1;
}

// y' = f(x,y)
std::vector<double> Euler(const VectorValuedFunction& f,
                     double x, std::vector<double>& y,   // Initial x and y
                     double T,               // End point of interval [x, T]
                     std::size_t N,          // Number of subdivisions
                     double TOL)             // Prescribed stopping
```

```
{ // Explicit Euler method

    double h = (T - x) / static_cast<double>(N);

    // Variables
    std::vector<double> ynP1(y.size());   // y(n+1)

    for (std::size_t n = 1; n <= N; ++n)
    {

       for (std::size_t j = 0; j < ynP1.size(); ++j)
       {
          ynP1[j] = y[j] + h*f[j](x, y);
       }

       // Go to next level
       x += h;

       std::copy(ynP1.begin(), ynP1.end(), y.begin());
    }

    return ynP1;
}
```

24.7.2 Runge–Kutta Method

The fourth-order Runge–Kutta method (24.33) has the following implementations in the scalar and systems cases:

```
// y' = f(x,y)
double RK4(const std::function<double(double x, double y)>& f,
       double x, double y,      // Initial x and y
       double T,                // End point of interval [x, T]
       std::size_t N,           // Number of subdivisions
       double TOL)              // Prescribed stopping
{ // Order 4 Runge Kutta method

    double h = (T - x) / static_cast<double>(N);

    // Variables
    double k1, k2, k3, k4;
    double ynP1;   // y(n+1)

    for (std::size_t n = 1; n <= N; ++n)
    {
       k1 = f(x, y);
       k2 = f(x + h / 2, y + h*k1 / 2);
       k3 = f(x + h / 2, y + h*k2 / 2);
       k4 = f(x + h, y + h*k3);
```

```
        ynP1 = y + h*(k1 + 2*k2 + 2*k3 + k4)/6;
        // Go to next level
        x += h;
        y = ynP1;
    }

    return ynP1;

}

// y' = f(x,y)
std::vector<double> RK4(const VectorValuedFunction& f,
        double x, std::vector<double> y,   // Initial x and y
        double T,                          // End point of interval [x, T]
        std::size_t N,                     // Number of subdivisions
        double TOL)                        // Prescribed stopping
{ // Order 4 Runge Kutta method

    double h = (T - x) / static_cast<double>(N);
    std::cout << h << '\n';

    // Variables
    std::vector<double> k1(y.size()), k2(y.size()),
             k3(y.size()), k4(y.size());
    std::vector<double> ynP1(y.size());   // y(n+1)
    std::vector<double> tmp(y.size());   std::vector<double> tmp2(y.size());
    /*
    k1 = f(x, y);
    k2 = f(x + h / 2, y + h*k1 / 2);
    k3 = f(x + h / 2, y + h*k2 / 2);
    k4 = f(x + h, y + h*k3);

    ynP1 = y + h*(k1 + 2*k2 + 2*k3 + k4)/6;
    */
    for (std::size_t n = 1; n <= N; ++n)
    {

       for (std::size_t j = 0; j < k1.size(); ++j)
       {
          k1[j] = f[j](x, y);
          tmp[j] = y[j] + h*k1[j] / 2;
       }

       for (std::size_t j = 0; j < k2.size(); ++j)
       {
          k2[j] = f[j](x + h/2, tmp);
          tmp2[j] = y[j] + h*k2[j] / 2;
       }


       for (std::size_t j = 0; j < k3.size(); ++j)
```

```
        {
            k3[j] = f[j](x + h / 2, tmp2);
            tmp[j] = y[j] + h*k3[j];
        }

        for (std::size_t j = 0; j < k4.size(); ++j)
        {
            k4[j] = f[j](x + h, tmp);
        }

        for (std::size_t j = 0; j < ynP1.size(); ++j)
        {
            ynP1[j] = y[j] + h*(k1[j] + 2*k2[j] + 2*k3[j] + k4[j]) / 6;
        }
        // Go to next level
        x += h;

        std::copy(ynP1.begin(), ynP1.end(), y.begin());
    }

    return ynP1;

}
```

Finally, we give some sample code on creating ODEs and approximating them by the methods in this chapter:

```
auto f = [](double x, double y) { return y*(1.0 - y); };    // (24.5)
double x = 0.0; double y = 0.5;
double T = 2.0;
std::size_t N = 100; double TOL = 1.0e-4;

auto val = PredictorCorrector(f, x, y, T, N, TOL);
std::cout << "Solution PC: " << val << '\n';

auto val2 = RK4(f, x, y, T, N, TOL);
std::cout << "Solution RK4: " << val2 << '\n';

auto val3 = Heun(f, x, y, T, N, TOL);
std::cout << "Solution Heun: " << val3 << ", "
          << 1.0/(1.0 + std::exp(-T)) << '\n';

auto val4 = Ralston2(f, x, y, T, N, TOL);
std::cout << "Solution Ralston2: " << val4 << '\n';

auto val5 = Ralston4(f, x, y, T, N, TOL);
std::cout << "Solution Ralston4: " << val5 << '\n';

// 2X2 system
std::size_t n = 2;
```

```
std::vector<double> y0{ 1.0, 1.0 };

// System (24.48)
VectorValuedFunction func(2);
RealValuedFunction f1 = [](double x, std::vector<double> y)
         { return -2*y[0] + y[1] + 2*std::sin(x); };
RealValuedFunction f2 = [](double x, std::vector<double> y)
   { return -2 * y[1] + 1*y[0] + 2 * (std::cos(x) - std::sin(x)); };

func[0] = f1; func[1] = f2;

auto valSystem = PredictorCorrector(func, x, y0, T, N, TOL);
std::cout << "Solution PC system: " << valSystem[0] << ", "
          << valSystem[1] << ", " <<  '\n';
auto valRK4 = RK4(func, x, y0, T, N, TOL);
std::cout << "Solution RK system: " << valRK4[0] << ", "
          << valRK4[1] << ", " << '\n';

auto valEuler = Euler(func, x, y0, T, N, TOL);
std::cout << "Solution Euler system: " << valEuler[0] << ", "
          << valEuler[1] << ", " << '\n';

auto y1 = 2 * std::exp(-T) + std::sin(T);
auto y2 = 2 * std::exp(-T) + std::cos(T);
std::cout << "Exact solution PC system: " << y1 << ", "
          << y2 << ", " << '\n';
```

24.7.3 Stiff Systems

We conclude this chapter with the numerical approximation of the stiff system (24.51):

```
// 2X2 system
std::size_t n = 2;
std::vector<double> y0{ 2.0, 0.0 };

double T = 1.0; double x = 0.0;
std::size_t N = 1200; double TOL = 1.0e-4;

double lambda1 = -1.0; double lambda2 = -1000;
double a = (lambda1 + lambda2) / 2; double b = (lambda1 - lambda2) / 2;
VectorValuedFunction func(2);

// Component functions of fun
RealValuedFunction f1 = [=](double x, std::vector<double> y)
         { return a*y[0] + b*y[1]; };
RealValuedFunction f2 = [=](double x, std::vector<double> y)
         { return b*y[0] + a*y[1]; };
func[0] = f1; func[1] = f2;
```

```
auto valSystem = PredictorCorrector(func, x, y0, T, N, TOL);
std::cout << "Solution PC system: " << valSystem[0] << ", "
          << valSystem[1] << ", " <<  '\n';
auto valRK4 = RK4(func, x, y0, T, N, TOL);
std::cout << "Solution RK system: " << valRK4[0] << ", "
          << valRK4[1] << ", " << '\n';

auto valEuler = Euler(func, x, y0, T, N, TOL);
std::cout << "Solution Euler system: " << valEuler[0] << ", "
          << valEuler[1] << ", " << '\n';

// Equation (24.52) of book
auto y1 = std::exp(lambda1*T) + std::exp(lambda2*T);
auto y2 = std::exp(lambda1*T) - std::exp(lambda2*T);

std::cout << "Exact solution system: " << y1 << ", "
          << y2 << ", " << '\n';
```

24.8 SUMMARY AND CONCLUSIONS

In this chapter we gave an overview of ODEs and their numerical approximation. We also developed C++ code to implement the methods and we used the code to test some model problems. A practical knowledge of ODEs is useful and probably essential if we wish to understand SDEs and PDEs. In the former case we can use variations of some of the finite difference methods for ODEs that we introduced in this chapter, while in the second case it is possible to apply the Method of Lines to a PDE. This process results in a system of ODEs. We discuss the Method of Lines in Chapter 25.

24.9 EXERCISES AND PROJECTS

1. (Activation Function)

Consider the activation function:

$$f(x) = \tanh x = \frac{2}{1 + e^{-2x}} - 1.$$

Verify that this function satisfies the ODE:

$$\frac{df}{dx} = 1 - f^2.$$

Determine the values of x for which this problem has a unique solution by constructing the Picard iterates. Approximate the ODE by the finite difference schemes that we discussed in this chapter. Carry out the same exercise for the logistic ODE (24.5).

2. (Bernoulli Equation with $n = 5$)

Transform the nonlinear equation $y' - \frac{y}{2x} = 5x^2y^5$ to the linear form $v' + \frac{2}{x}v = -20x^2$ where $v = y^{-4}$. The exact solution is given by $y = \frac{1}{\sqrt[4]{cx^{-2}-4x^3}}$ where c is a constant. For which values of the independent variable x does this problem have a real solution? Approximate the unmodified forms of this ODE using the predictor–corrector and Runge–Kutta methods. Compare your results with the exact solution.

3. (Cauchy–Euler Equation)

The technique that we discuss in this exercise is similar to that used for the Black–Scholes PDE on the positive axis to reduce it to a PDE with constant coefficients.

Consider the ODE:

$$x^2\frac{d^2y}{dx^2} + ax\frac{dy}{dx} + b = 0$$

where a and b are constants.

Define the variable substitution:

$$t = \log(x), y = \phi(\log(x)) = \phi(t).$$

Prove that:

$$\frac{d^2\phi}{dt^2} + (a-1)\frac{d\phi}{dt} + b\phi = 0.$$

Solve this ODE using its *characteristic polynomial* in the case of distinct roots:

$$\phi(t) = c_1e^{\lambda_1 t} + c_2e^{\lambda_2 t}.$$

When λ_1 and λ_2 are roots of the characteristic polynomial $\lambda^2 + (a-1)\lambda + b = 0$ and c_1 and c_2 are constants.

4. (Homogeneous ODE)

Prove that the following ODEs are homogeneous and find the corresponding degrees:

$$(x_2 - y^2)dx + 2xydy = 0$$
$$xdy - (y + \sqrt{x^2 - y^2})dx = 0.$$

Show that:

$$(x^2 - y^2)dx + (x - y)dy = 0$$

is not a homogeneous ODE.

5. (Picard Iteration)

Apply the techniques of Section 24.5.1 to determine the intervals in which the following problems have unique solutions (or not):

$$\frac{dy}{dt} = y(1-y),\ y(0) = 1/2$$
$$\frac{dy}{dt} = 1 + y^2,\ y(0) = 0 \text{ (solution } y(t) = \tan(t))$$
$$\frac{dy}{dt} = \frac{2y}{x},\ y(t_0) = y_0.$$

6. (Runge–Kutta Method)

Answer the following questions:

a) Construct the *Butcher tableau* for the schemes (24.34) (Ralston) and (24.35) (Heun) (see Section 24.10 (Appendix) for a definition).

b) Compare the accuracy of these two methods against each other.

c) Construct and implement the *third-order Runge–Kutta method* defined by the following Butcher tableau:

0	0	0	0
1/2	1/2	0	0
1	−1	2	0
	1/6	2/3	1/6

Write the scheme in algorithmic form.

7. (Coding Exercises)

The goal in this exercise is to implement a number of schemes for (24.24) and to test them on some of the examples in this chapter.

Answer the following questions:

a) Implement the second-order Ralston method (24.34) and its extrapolated fourth-order version (see Section 24.6.3). Compare the accuracy of these methods on scalar stiff and non-stiff ODEs. Carry out the same exercise for systems of ODEs.

b) Implement the Heun method (24.35) and follow the steps as outlined in part a).

c) Test the non-stiff and stiff systems (24.48) and (24.49) using explicit Euler, fourth-order Runge–Kutta and Ralston methods for a range of time steps. Compare the relative accuracy.

24.10 APPENDIX

Runge–Kutta methods are one-step methods for the numerical solution of the ordinary differential equation:

$$\frac{dy}{dt} = f(t, y)$$

which take the form:

$$
\begin{aligned}
y_{n+1} &= y_n + h \sum_{i=1}^{s} b_i k_i \\
k_1 &= f(t_n, y_n), \\
k_2 &= f(t_n + c_2 h, y_n + h(a_{21} k_1)), \\
k_3 &= f(t_n + c_3 h, y_n + h(a_{31} k_1 + a_{32})) \\
&\vdots \\
k_i &= \left(t_n + c_i h, y_n + h \sum_{j=1}^{i-1} a_{ij} k_j \right).
\end{aligned}
$$

Each method listed is defined by its *Butcher tableau*, which puts the coefficients of the method in a table as follows:

$$
\begin{array}{c|cccc}
c_1 & a_{11} & a_{12} & \cdots & a_{1s} \\
c_2 & a_{21} & a_{22} & \cdots & a_{2s} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
c_s & a_{s1} & a_{s2} & \cdots & a_{ss} \\
\hline
 & b_1 & b_2 & \cdots & b_s
\end{array}
$$

A useful exercise is to construct the Butcher tableau for the schemes that we introduced in this chapter.

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 25

Advanced Ordinary Differential Equations and Method of Lines

25.1 INTRODUCTION AND OBJECTIVES

We continue from Chapter 24 with our discussion of the numerical solution of systems of first-order ordinary differential equations. Whereas in Chapter 24 we developed our own code to approximate ODEs, we now use the Boost *odeint* library to do the work for us. We then introduce the *Method of Lines* (MOL) that allows us to approximate the solutions of the time-dependent parabolic partial differential equations as discussed in Chapters 20 to 23. In this sense MOL is a competitor to the handcrafted time discretisation methods such as BTCS and Crank–Nicolson, for example. MOL offers a number of advantages:

- A1: It can be applied to linear time-dependent PDEs with a range of boundary conditions.
- A2: It can be applied to nonlinear PDEs such as UVM (Pealat and Duffy, 2011), CVA (Green, 2016) and early-exercise option pricing using penalty and barrier methods.
- A3: We can choose from a wide range of time-marching schemes, choosing the one that is most suitable to the problem at hand. For example, we may wish to produce highly accurate results or to choose a scheme that is suitable for stiff equations.
- A4: MOL is easy to apply to 2-factor and 3-factor PDEs, for example the two-dimensional heat equation, basket options, the Heston model and the anchoring problem of Chapter 23.
- A5: Run-time performance. MOL is a generic method that is suitable for linear and nonlinear PDEs and in general it will be slower than handcrafted code. However, this may not always be the case, especially for nonlinear PDEs.
- A6: MOL is usually more robust than handcrafted finite difference schemes.

In all cases in this chapter we shall use Boost *odeint* as the ODE solver to approximate the solution of time-dependent convection–diffusion–reaction PDEs. We take illustrative examples and we give several important and challenging exercises on applying MOL and comparing it to the finite difference methods discussed in Chapters 20 to 23.

25.2 AN INTRODUCTION TO THE BOOST *ODEINT* LIBRARY

There are several libraries that support MOL. In *Mathematica*, for example, the *NDSolve* package approximates the solution of a system of ODEs using numerical methods and there is also support in the NAG library. These sources are useful as background references. Furthermore, good discussions of the mathematical and numerical foundations for MOL and related topics can be found in Lambert (1991), Henrici (1962) and Crank (1984).

In this chapter we focus on Boost *odeint*. From the online documentation, the library is described as follows:

> *The main focus of* odeint *is to provide numerical methods implemented in a way where the algorithm is completely independent of the data structure used to represent the state. In doing so,* odeint *is applicable to a broad variety of applications and it can be used with many other libraries. Besides the usual case where the state is defined as a* `std::vector` *or a* `boost::array`*,* odeint *provides support for the following libraries:*
>
> - *Boost.uBLAS.*
> - *Thrust, allowing odeint to run on CUDA devices.*
> - *gsl_vector for compatibility with the many numerical functions in the GSL.*
> - *Boost.Range.*
> - *Boost.Fusion (the state type can be a fusion vector).*
> - *Boost.Units.*
> - *Intel Math Kernel Library for maximum performance.*
> - *VexCL for OpenCL.*
> - *Boost.Graph.*

We do not discuss these features as we are more interested in the *stepper algorithms* and *integrate functions* that *odeint* supports.

25.2.1 Steppers

A *stepper* in *odeint* performs a single step, for example from time level t to time level $t + dt$, where dt is the step size. In other words, it performs a single step of the solution $x(t)$ of an ODE to obtain $x(t + dt)$ using a given step size dt. To this end, each stepper has two overloaded versions of a `do_step` method, namely an *in-place transform* of the current state and an *out-of-place transform*:

- `do_step(sys, inout, t, dt)`
- `do_step(sys, in, t, out, dt)`

respectively, where:

- `sys`: the right-hand of ODE $dy/dt = rhs$.
- `inout`: the solution at time t and which then becomes the solution of time $t + dt$ (the former value is overwritten).
- `in` and `out`: the values at time levels t and $t + dt$, respectively.

In general, you do not need to know about this level of detail because it is taken care of by the various *integrate functions* that we shall discuss in Section 25.2.3. The main stepper specialisations are as follows.

- *Explicit steppers*: the new state of the ODE can be computed directly from the current state without having to solve implicit (usually nonlinear) equations.
- *Symplectic steppers*: these steppers are used for *Hamiltonian systems* which is outside the scope of this book.
- *Implicit steppers*: these are steppers that are suitable for *stiff systems* that typically possess two or more time scales of radically different order. We saw in Chapter 24 that classical Runge–Kutta methods are unsuitable for stiff systems. In general, we use a nonlinear solver (such as the Newton–Raphson method) and hence we need to compute the system *Jacobian*.
- *Controlled steppers*: these are steppers that have the ability to use *adaptive step-size control*. We can decrease or increase the step size depending on whether the current error is less than or greater than a given tolerance. An example is `bulirsch_stoer`.
- *Dense output steppers*: these are steppers that might take larger steps and interpolate the solution between two consecutive points. An example is `euler`.
- *Error stepper*: calculates one step of the solution $x(t)$ of an ODE with step size dt to produce $x(t + dt)$ while also computing an error estimate of the result.

Some specific steppers are:

- `runge_kutta4`
- `euler`
- `runge_kutta_cash_karp54`
- `runge_kutta_dopri5`
- `runge_kutta_fehlberg78`
- `modified_midpoint`
- `rosenbrock4`

and typical C++ code is (for easy reference):

```
// Steppers: a stepper maps the solution x(t) -> x(t+dt) ->
// x(t + 2dt) -> ...
Bode::modified_midpoint<state_type, value_type> mmpStepper;         // O(2)
Bode::bulirsch_stoer<state_type, value_type> bsStepper;             // Variable
Bode::runge_kutta_fehlberg78<state_type, value_type> rkfStepper;    // O(8)
Bode::runge_kutta_dopri5<state_type, value_type> rkdopiStepper;     // O(5)
Bode::euler<state_type, value_type> eulerStepper;                   // O(1)
Bode::runge_kutta_cash_karp54<state_type, value_type> rkccStepper; // O(5)
Bode::runge_kutta4<state_type, value_type> rk4Stepper;              // O(4)
Bode::rosenbrock4<value_type> rosenbrock4Stepper;                   // O(4)
```

The parameters `state_type` and `value_type` are user-definable type as we shall see. For example, the former type could be `std::vector<double>` while the latter type could be `double`.

We now discuss how to use steppers with *integrate functions*. For mathematical and numerical background, see Press et al. (2002), Henrici (1962) and especially Lambert (1991).

25.2.2 Examples of Steppers

Steppers can be seen as the building blocks of *odeint* in the sense that they implement one-step solvers for ODEs. They are similar to the steppers that we created in Chapter 24. It is useful to see how they work, including the ability to estimate the error in a subinterval $[t, t + dt]$. We take an example by defining a specific ODE (the negative exponential function):

```
using value_type = double;
using state_type = std::vector<value_type>;

// Simple ODE system (scalar (vector of length 1))
struct Ode101
{ // du/dt = -u, u(0) = 1; u(t) = exp(-t)

    void operator() (const state_type &x, state_type &dxdt,
                     const value_type t)
    {
       dxdt[0] = -x[0];
    }
};
```

We can then use a stepper to compute an approximate solution. An example using three steppers (fourth-order Runga–Kutta, modified midpoint and Cash–Karp) is:

```
int main()
{
    namespace Bode = boost::numeric::odeint;

    state_type x(1); // state, vector of 1 element (scalar problem)
    x[0] = 1.0;

    state_type y(1);
    state_type z(1);
    state_type err(1);

    // Integration parameters
    value_type t = 0.0;
    value_type dt = 0.1;
    size_t nSteps = 7;

    // Create stepper
    Bode::runge_kutta4<state_type> rk4;
    Bode::modified_midpoint<state_type> mmp;
    Bode::runge_kutta_cash_karp54<state_type> rkck54;
```

```
    Ode101 ode;
    Ode101 ode2;
    Ode101 ode3;

    // Perform integration step by step
    for (std::size_t i = 0; i < nSteps; ++i)
    {
       // adjourn current time
       t += dt;

       // Perform one integration step. Solution x is overwritten
       rk4.do_step(ode, x, t, dt);

       // 2nd option (y is solution at t + dt)
       mmp.do_step(ode2, x, t, y, dt);

       // 3rd option: error stepper
       rkck54.do_step(ode3, x, t, z, dt, err);

       // Display results
       std::cout << '\n' << t << "Exact, approx  :  "
                 << std::setprecision(8)
                 << std::exp(-t) << ", " << x[0] << '\n';
       std::cout << t+dt << "Exact, next approx  :  "
                 << std::setprecision(8)
                 << std::exp(-(t + dt)) << ", " << y[0] << '\n';
       std::cout << t+dt << "Error stepper + error :  "
                 << std::setprecision(8)
                 << std::exp(-(t + dt)) << ", " << z[0]
                 << ", error " << err[0];
    }
}
```

The output from this code is:

```
0.1Exact, approx           :  0.90483742, 0.9048375
0.2Exact, next approx      :  0.81873075, 0.81876483
0.2Error stepper + error   :  0.81873075, 0.81873083, error 2.1926919e-09

0.2Exact, approx           :  0.81873075, 0.8187309
0.3Exact, next approx      :  0.74081822, 0.74084912
0.3Error stepper + error   :  0.74081822, 0.74081835, error 1.9840299e-09

0.3Exact, approx           :  0.74081822, 0.74081842
0.4Exact, next approx      :  0.67032005, 0.67034807
0.4Error stepper + error   :  0.67032005, 0.67032023, error 1.7952247e-09

0.4Exact, approx           :  0.67032005, 0.67032029
0.5Exact, next approx      :  0.60653066, 0.60655607
0.5Error stepper + error   :  0.60653066, 0.60653088, error 1.6243866e-09
```

```
0.5Exact, approx           :  0.60653066, 0.60653093
0.6Exact, next approx      :  0.54881164, 0.54883468
0.6Error stepper + error   :  0.54881164, 0.54881188, error 1.4698059e-09

0.6Exact, approx           :  0.54881164, 0.54881193
0.7Exact, next approx      :  0.4965853, 0.4966062
0.7Error stepper + error   :  0.4965853, 0.49658557, error 1.3299355e-09

0.7Exact, approx           :  0.4965853, 0.49658562
0.8Exact, next approx      :  0.44932896, 0.44934791
0.8Error stepper + error   :  0.44932896, 0.44932925, error 1.2033755e-09
```

You can review the code and this output to help you understand how these steppers work. Finally, the error steppers in the library are:

- `runge_kutta_cash_karp54`
- `runge_kutta_dopri5`
- `runge_kutta_fehlberg78`
- `rosenbrock4`

They compute a new value and they give an estimate of the error.

25.2.3 Integrate Functions and Observers

Integrate functions perform time marching of an ODE from a starting time t_0 to a given end time t_1. The next stage is to use them in your applications starting at state x_0 by the application of calls to a given stepper's `do_step` method. It is also possible to provide two extra pieces of functionality in the form of input arguments to an integrate function:

- An *observer function* that analyses state (typically at the discrete mesh points) during system evolution, for example displaying the current state or storing values in a collection for later use.
- Throwing an exception if too many steps are being taken between observer calls. To this end, we instantiate the lightweight class `max_step_checker` with an integer value representing the maximum number of steps allowed before an exception is thrown. This is in fact an overflow check when no progress is being made.

There are four integrate functions, each one having its own strategy on when and how to call the observer function during integration. Each integrate function (with the exception of `integrate_n_steps`) can be called with any stepper. The integrate functions make use of step-size control or dense output depending on the ability of the stepper:

1. If observer calls at *equidistant* time intervals are needed, then we use `integrate_const` or `integrate_n_steps`. The former function takes the end time as argument while the latter method takes the number of time steps as argument.

2. If observer calls are desired at each time step then `integrate_const` should be used. It is possible to get non-equidistant observer calls in the case of controlled steppers and dense output steppers.
3. We use `integrate_times` if the observer should be called at time points *given by the user*. The times for observer calls are provided as a sequence of time values. This could be a useful method for Bermudan-type option pricing problems.
4. Instead of the above four sophisticated integrate functions we can choose `integrate` if we are only interested in an *entry-level integrator*. It uses a dense output stepper based on `runge_kutta_dopri5` with an error bound 10^{-6} at each step.

Which option to use depends on the context and type of application. We shall give some examples in the next section to show how each integrate function works in the context of the Riccati ODE. The observer and arguments to these functions are optional. The return type of these functions is an unsigned integer representing the number of function evaluations.

25.2.4 Modelling ODEs and their Observers

The first step in using Boost *odeint* is to define the first-order system of equations whose solution we wish to approximate. In most cases the model will be the vector ODE as in equation (24.24) with scalar ODEs as a special case (a scalar can be seen as a vector of length one). Furthermore, the underlying data type is usually `double` but we can discuss ODEs where the underlying data is complex (using `std::complex`). We also discuss *matrix differential equations* in Section 25.4.

We now discuss how to model system (24.24) *in the scalar case*. We first define the necessary function spaces in C++:

```
using value_type = double;
using state_type = std::vector<value_type>;

using FunctionType = std::function<value_type (value_type)>;
using FunctionType2
                   = std::function<value_type (value_type x, value_type y)>;
```

We now define a function object that encapsulates the C++ structure corresponding to system (24.24). In this case the vectors are of length one:

```
// The rhs of y' = f(x,y) defined as a class
class Ode
{
    private:
       FunctionType2 rhs; // dy/dx = rhs
    public:
       Ode(const FunctionType2& RHS) : rhs(RHS) { }

       void operator() (const state_type &x, state_type &dxdt,
                        const value_type t)
       {
```

```
            dxdt[0] = rhs(t,x[0]);
        }
};
```

We also create a related *observer* class that stores relevant state information (the solution at each time level, for example) that can be used later with the Excel driver, for example:

```
class WriteOutput
{
    private:

    public:
       std::vector<value_type>& tValues;
       std::vector<value_type>& yValues;
    public:
        WriteOutput():tValues(std::vector<value_type>()),
                      yValues(std::vector<value_type>())
    {}

    void operator ()( const state_type &x , const value_type t )
    {
       tValues.push_back(t); yValues.push_back(x[0]);
    }

    const std::vector<value_type>& xAxis() const
    {
       return tValues;
    }

    const std::vector<value_type>& yAxis() const
    {

       return yValues;
    }
};
```

In general, we use factory objects to instantiate the class `Ode`. In this case we take a specific case of the *Riccati equation* for motivation that we introduced in Section 24.4.3. We use shared pointers:

```
std::shared_ptr<GeneralisedRiccati> CreateRiccati()
{ // Factory method, specific case

   const double P = 0.0;
   const double Q = -10.8;
   double R = 0.5 * 2.44 * 2.44;
   const double eta = 0.96;
   const double lambda = 0.13;
   double A = 0.42;
```

```
    // Generalised Riccati
    auto p = [=](double x) { return P; };
    auto q = [=](double x) { return Q; };
    auto r = [=](double x) { return R; };
    auto n = [=](double x, double y)
           { return (lambda*y) / (eta - y); };

    return std::shared_ptr<GeneralisedRiccati>
           (new GeneralisedRiccati (p, q, r, n));
}
```

We are now ready to show how to use the integrate functions that we introduced in Section 25.2.3 to model this problem. You can run the code and check the output in Excel:

```
namespace Bode = boost::numeric::odeint;

// Initial condition
state_type x(1);
double A = 0.42;
x[0] = A;

// Interval of integration [0,T]
double L = 0.0;
double U = 1.0;

auto riccatiEquation = CreateRiccati();
//auto riccatiEquation = CreateBernoulli();
Ode ode(*riccatiEquation); WriteOutput wo;

double dt = 0.0001;
std::size_t steps1 = Bode::integrate(ode, x, L, U, dt, wo);
std::cout << "Steps integrate() " << steps1 << ", value " << x[0];
ExcelDriver xl; xl.MakeVisible(true);
xl.CreateChart(wo.xAxis(), wo.yAxis(), std::string("integrate() for
Riccati"));

// Steppers: maps the solution x(t) -> x(t+dt) -> x(t + 2dt) -> ...
Bode::modified_midpoint<state_type, value_type> mmpStepper;
Bode::bulirsch_stoer<state_type, value_type> bsStepper;
Bode::runge_kutta_fehlberg78<state_type, value_type> rkfStepper;
Bode::runge_kutta_dopri5<state_type, value_type> rkdopiStepper;
Bode::euler<state_type, value_type> eulerStepper;
Bode::runge_kutta_cash_karp54<state_type, value_type> rkccStepper;
Bode::runge_kutta4<state_type, value_type> rk4Stepper;
Bode::rosenbrock4<value_type> rosenbrock4Stepper;

dt = 0.01;
x[0] = A;
WriteOutput wo2;
std::size_t steps2
       = Bode::integrate_const(bsStepper, ode, x, L, U, dt, wo2);
```

```
std::cout << "Value from equidistant time interval: " << "steps "
          << steps2 << ", value " << x[0] << std::endl;
xl.CreateChart(wo2.xAxis(), wo2.yAxis(),
          std::string("integrate_const() for Riccati"));

dt = 0.0001;
x[0] = A;
WriteOutput wo3;
std::size_t steps3
       = Bode::integrate_adaptive(rkfStepper,ode,x,L, U, dt, wo3);
std::cout << "integrate_adaptive() for Riccati: " << "steps "
           << steps3 << ", value " << x[0] << std::endl;
xl.CreateChart(wo3.xAxis(), wo3.yAxis(),
          std::string("integrate_adaptive for Riccati"));

WriteOutput wo4;
std::size_t n = 10000;
dt = 0.0001;
x[0] = A;

// Integrate until the time t_0 + n*dt.
std::size_t steps4
   = Bode::integrate_n_steps(rk4Stepper, ode, x, L, dt, n, wo4);
std::cout << "integrate_n_steps() for Riccati: " << "steps "
           << steps4 << ", value " << x[0] << std::endl;
xl.CreateChart(wo4.xAxis(), wo4.yAxis(),
      std::string("integrate_n_steps for Riccati"));


Bode::runge_kutta_dopri5<state_type, double> rkdStepper;
std::size_t N = 10;
auto mesh = CreateMesh(L, U, N);
WriteOutput wo5;

// Call the observer at certain user points
x[0] = A;
std::size_t steps5= Bode::integrate_times(rkdStepper, ode, x,
           boost::begin(mesh), boost::end(mesh), dt, wo5);
std::cout << "integrate_times() for Riccati: " << "steps "
           << steps5 << ", value " << x[0] << std::endl;
xl.CreateChart(wo5.xAxis(), wo5.yAxis(),
         std::string("integrate_times() for Riccati"));

Bode::modified_midpoint<state_type, double> modifiedMPStepper;
std::size_t M = 10;
auto mesh2 = CreateMesh(L, U, M);
WriteOutput wo6;

// Call the observer at certain user points
x[0] = A;
dt = 0.01;
```

```
std::size_t steps6
       = Bode::integrate_times(rkfStepper, ode, x,
          boost::begin(mesh2), boost::end(mesh2), dt, wo6);
std::cout << "integrate_times() II for Riccati: " << "steps "
           << steps6 << ", value " << x[0] << std::endl;
xl.CreateChart(wo6.xAxis(), wo6.yAxis(),
               std::string("integrate_times() II"));
```

25.3 SYSTEMS OF STIFF AND NON-STIFF EQUATIONS

We have given a short introduction to stiff ODEs in Section 24.6.6. Not all solvers are equally efficient for these kinds of problems because of the different time scales. We now discuss stiff ODEs in the context of Boost *odeint*. We focus on specific examples for motivation.

25.3.1 Scalar ODEs

We discuss system (24.46) and its solution (24.47). We have noted that standard finite difference schemes such as Euler and Crank–Nicolson need to employ very small time steps in order to get accurate results. In this section we discuss how to improve performance. To this end, we investigate the Cash–Karp and Bulirsch–Stoer methods. First, we model the ODE as a free function (see equation (24.46)):

```
// Free function to model RHS in dy/dt = RHS(t,y)
void Stiff101 (const state_type &x, state_type &dxdt, const double t)
{
    dxdt[0] = 100.0*(std::sin(t) - x[0]);
}
```

The exact solution is given by (see equation (24.47)):

```
double exact(double t)
{
   return (std::sin(t) - 0.01*cos(t)
         + 0.01*std::exp(-100.0*t)) / 1.0001;
}
```

We use the following data type and state:

```
using value_type = double;
using state_type = std::vector<value_type>;
```

The test program is:

```
int main()
{
   namespace Bode = boost::numeric::odeint;
```

```
    // Initial condition
    value_type L = 0.0;
    value_type T = 20.0;
    value_type dt = 0.001;

    // Initial condition
    state_type x(1);
    value_type A = 0.0;
    x[0] = A;

    std::size_t steps = Bode::integrate(Stiff101, x, L, T, dt);
    std::cout << "Number of steps Cash Karp 54: "
              << std::setprecision(16) << steps
              << ", exact: " << exact(T) << ", approximate: "
              << x[0] << '\n';

    x[0] = A;
    Bode::bulirsch_stoer<state_type, value_type> bsStepper;
    steps = Bode::integrate_const(bsStepper, Stiff101, x, L, T, dt);
    std::cout << "Number of steps Bulirsch-Stoer: " << steps
              << ", exact: " << exact(T) << ", approximate: " << x[0];

    return 0;
}
```

We have found that Cash–Karp gives 4-digit accuracy with 1,200 function evaluations while Bulirsch–Stoer gives 10-digit accuracy with 20,000 function evaluations.

25.3.2 Systems of ODEs

We examine systems (24.48) (non-stiff) and (24.49) (stiff) as good representatives or *exemplars* of how to implement systems of ODEs in Boost *odeint*. We focus mainly on the stiff case and a discussion of suitable methods. We define the *Jacobian matrix J* for system (24.24) whose elements are defined as follows:

$$J_{ij} = \frac{\partial f_i}{\partial y_j},\ 1 \le i,j \le n. \tag{25.1}$$

This formula is used as input to ODE solvers, in which case indexing begins at 0 (C language standard). In the case of system (24.48) the Jacobian is:

$$J = \begin{pmatrix} -2 & 1 \\ 1 & -2 \end{pmatrix} \tag{25.2}$$

and in the case of system (24.49) it is:

$$J = \begin{pmatrix} -2 & 1 \\ 998 & -999 \end{pmatrix}. \tag{25.3}$$

Computing the eigenvalues in each case will show why system (24.49) is stiff.

We now turn our attention to implementing systems (24.49) (and (24.48) commented in the code) in C++. The augmented ODE class is:

```
using value_type = double;
using vector_type = boost::numeric::ublas::vector<value_type>;
using matrix_type = boost::numeric::ublas::matrix<value_type>;

struct StiffSystem
{
    void operator()(const vector_type& x , vector_type& dxdt ,
                    value_type t)
    { // Lambert page 213

        // Non-stiff form
        /*  dxdt[0] = -2.0 * x[0] + 1.0 * x[1] + (2.0*std::sin(t));
            dxdt[1] = 1.0 * x[0] - 2.0 * x[1]
              + (2.0*(std::cos(t) - std::sin(t)));*/

        // Stiff form
        dxdt[0] = -2.0 * x[0] + 1.0 * x[1] + (2.0*std::sin(t));
        dxdt[1] = 998.0 * x[0] - 999.0 * x[1]
                    + (999.0*(std::cos(t) - std::sin(t)));
    }
};


struct StiffSystemJacobi
{
     void operator()(const vector_type& /* x */ , matrix_type& J ,
                      const value_type& t, vector_type& dfdt )
    {
      // Non-stiff form
      /*  J(0,0) = -2.0;
      J(0,1) = 1.0;
      J(1,0) = 1.0;
      J(1,1) = -2.0;*/

      // Stiff form
      J(0,0) = -2.0;
      J(0,1) = 1.0;
      J(1,0) = 998.0;
      J(1,1) = -999.0;

      // Derivative wrt t of RHS
      // Non-stiff
      /*dfdt[0] = 2.0 * std::cos(t);
      dfdt[1] = 2.0 * (-std::sin(t) - std::cos(t));*/

      // Stiff
      dfdt[0] = 2.0 * std::cos(t);
```

```
        dfdt[1] = 999.0 * (-std::sin(t) - std::cos(t));

    }
};
```

You should check the code to ensure that the Jacobian has been calculated correctly!

To aid visualisation and debugging we also create an observer class to store the arrays of computed values that we can display in Excel:

```
class WriteOutput
{
private:

public:
    std::vector<value_type> tValues;
    std::vector<value_type> y1Values;
    std::vector<value_type> y2Values;
public:
    WriteOutput() : tValues(std::vector<value_type>()),
                    y1Values(std::vector<value_type>()),
                    y2Values(std::vector<value_type>()) {}

    void operator ()(const vector_type &x, const value_type t)
    {
        tValues.push_back(t);
        y1Values.push_back(x[0]);
        y2Values.push_back(x[1]);
    }

    const std::vector<value_type>& xAxis() const
    {
        return tValues;
    }

    const std::vector<value_type>& y1Axis() const
    {
        return y1Values;
    }

    const std::vector<value_type>& y2Axis() const
    {
        return y2Values;
    }
};
```

A test program is:

```
// Example based on Lambert "Numerical Methods for Ordinary
// Differential Equations" Wiley 1991. page 213
```

```
// Initial condition
vector_type x(2);
x[0] = 2.0; x[1] = 3.0;

// Error tolerances
value_type absErr = 1.0e-6;
value_type relErr = 1.0e-6;

// Integration range
value_type L = 0.0;
value_type T = 10.0;
value_type dt = 0.001;

WriteOutput wo;
std::size_t steps = Bode::integrate_const
           (Bode::make_dense_output<Bode::rosenbrock4<value_type>>
           (absErr,relErr), std::make_pair(StiffSystem(),
            StiffSystemJacobi()), x , L, T, dt, std::ref(wo));

// std::size_t steps = Bode::integrate(StiffSystem(), x, L, T, dt,
// std::ref(wo));

std::cout << steps << std::endl;
```

We note the use of the *generator function* `make_dense_output` to create a dense output stepper from a simpler stepper. Here we see above how to introduce the *absolute and relative errors* into the stepper.

For completeness, we include the exact solution (24.50) in order to give an estimate of the error:

```
// Post processing: computing error and displaying the values in Excel
value_type f1 = 2.0 * std::exp(-T); value_type f2 = f1;
double exact1 = f1 + std::sin(T);
double exact2 = f2 + std::cos(T);
std::cout << "Exact: [" << std::setprecision(16)
           << exact1 << ", " << exact2 <<  "]\n";
std::cout << "Approximate: [" << x[0] << ", " << x[1] << "]\n";
std::cout << "Errors: " << exact1 - x[0] << ", "
           << exact2 - x[1] << '\n';
```

Finally, we can display the values in Excel:

```
// Display in Excel Driver
ExcelDriver xl; xl.MakeVisible(true);
xl.CreateChart(wo.xAxis(), wo.y1Axis(), std::string("y1 values"));
xl.CreateChart(wo.xAxis(), wo.y2Axis(), std::string("y2 values"));
```

The design and code in this section can be used as a template (*cookie cutter*) when you are designing similar systems.

25.4 MATRIX DIFFERENTIAL EQUATIONS

The Boost *odeint* library can be used to solve matrix ODEs, for example:

$$\frac{dY}{dt} = AY + YB \tag{25.4}$$

where A, B and Y are $n \times n$ matrices.

It can be shown that the solution of system (25.4) can be written as (see Brauer and Nohel, 1969):

$$Y(t) = e^{At} C e^{Bt}, \text{ where } Y(0) = C \text{ and } C \text{ is a given matrix.} \tag{25.5}$$

A special case is when:

$$\begin{aligned} &B = 0 \text{ (zero matrix)} \\ &C = 1 \text{ (identity matrix)} \end{aligned} \tag{25.6}$$

in which case we have the solution which is the *exponential of a matrix*:

$$Y(t) = e^{At}. \tag{25.7}$$

The conclusion is that we can now compute the exponential of a matrix by solving a matrix ODE. We discuss this topic using C++. To this end, we examine the system:

$$\begin{cases} \dfrac{dY}{dt} = AY \\ Y(0) = C \end{cases} \tag{25.8}$$

where C is a given vector.

We model this system with the following C++ class:

```
namespace ublas = boost::numeric::ublas;

using value_type = double;
using state_type = boost::numeric::ublas::matrix<value_type>;

class MatrixOde
{
private:
    // dB/dt = A*B, B(0) = C;
    ublas::matrix<value_type> A_;
    ublas::matrix<value_type> C_;
public:
    MatrixOde(const ublas::matrix<value_type>& A,
                const ublas::matrix<value_type>& IC)
                            : A_(A), C_(IC) {}
```

```
    void operator()(const state_type &x , state_type &dxdt,
             double t ) const
    {

       for( std::size_t i=0 ; i < x.size1();++i )
       {
          for( std::size_t j=0 ; j < x.size2(); ++j )
          {
            dxdt(i, j) = 0.0;
            for (std::size_t k = 0; k < x.size2(); ++k)
            {
                 dxdt(i, j) += A_(i,k)*x(k,j);
            }
          }
       }
    }
};
```

An initial example is:

```
namespace Bode = boost::numeric::odeint;

std::size_t N = 2;
ublas::identity_matrix<value_type> C(N);
ublas::matrix<value_type> A(N,N);
A(0, 0) = 6.0;  A(0, 1) = -1.0; A(1,0) = 9.0; A(1, 1) = 0.0;

ublas::matrix<value_type> B(N,N);

// Initialise the solution matrix B
B = C;

MatrixOde ode(A, C);
double L = 0.0;
double T = 3.0;
double dt = 0.001;

std::size_t steps = Bode::integrate(ode, B, L, T, dt, write_out);

std::cout << "Value at time: " << T << '\n';
std::cout << B << '\n';

// Computing the exact matrix
ublas::matrix<value_type> check(N, N);
double e = std::exp(3.0*T);
check(0, 0) = (1.0 + 3.0*T)*e;
check(0, 1) = -T*e;
check(1, 0) = 9.0*T*e;
check(1, 1) = (1.0 - 3.0*T)*e;
std::cout << check << '\n';
```

The output for cross-checking purposes is:

```
Value at time: 3
[2,2]((81030.9,-24309.3),(218783,-64824.7))
[2,2]((81030.8,-24309.3),(218783,-64824.7))
```

For completeness, we take the simplest example of a matrix ODE, namely a matrix system containing one row and one column (this is called a scalar). We take the ODE that describes the exponential function. We compute the value at $x = 5.0$ and it should be approximately 148.413. We include this example as a kind of sanity check; the code can be used as a template or exemplar for more extensive examples:

```
// Sanity check; 1X1 matrix case dB/t = B on (0,5), B(0) = I
{ // dB/dt = A*B, B(0) = C

    std::size_t N = 1;
    ublas::identity_matrix<value_type> C(N);
    ublas::matrix<value_type> A(N, N);
    A(0, 0) = 1.0;
    ublas::matrix<value_type> B(N, N);

    // Initialise the solution matrix B
    B = C;

    MatrixOde ode(A, C);
    double L = 0.0;
    double T = 5.0;
    double dt = 0.001;

    std::size_t steps = Bode::integrate(ode, B, L, T, dt, write_out);

    std::cout << "Value at time: " << T << '\n';

    std::cout << B << ", " << std::exp(5.0) <<  '\n'; // 148.413
}
```

We recommend that you experiment with larger matrix systems. You can modify the above code in order to get started.

There are many ways to compute the exponential of a matrix (see Moler and Van Loan, 2003), one of which involves the application of ODE solvers. Other methods include:

- S1: Series methods (for example, truncating the infinite Taylor series representation of the exponential).
- S2: Padé approximant. This entails approximating the exponential by a special kind of rational function.
- S3: Polynomial methods using the Cayley–Hamilton method.
- S4: Inverse Laplace transform.
- S5: Matrix decomposition methods.
- S6: Splitting methods.

Finally, we note that the need to compute the exponential of a matrix and to approximate the solution of matrix differential equations is of universal interest in many branches of science, engineering and finance. For example, in credit rating (Wilmott, 2006, Vol. 2, pp. 665–668), estimating *transition matrices* from observed data (Inamura, 2006) and the matrix Riccati equation (Boyle and Guan, 2002). We discuss this equation in short for completeness. Given are the n state variables $X_1, \ldots, X_n$ satisfying the vector SDE:

$$
\begin{aligned}
&dX = \left(K_0 + K_1 X\right) dt + \sigma\left(X\right) dW \\
&\text{where:} \\
&X = \left(X_1, \ldots, X_n\right)^{\top},\ K_0 \in \mathbb{R}^n, K_1 \in \mathbb{R}^{n\times n},\ \sigma : \mathbb{R}^n \to \mathbb{R}^{n\times n} \\
&W\left(t\right) = \left(W_1\left(t\right), \ldots, W_n\left(t\right)\right)^{\top} \ (n\text{-dimensional Brownian motion}).
\end{aligned}
\tag{25.9}
$$

The *short rate* is given by:

$$
r\left(t\right) = \sum_{j=1}^{n} X_j\left(t\right). \tag{25.10}
$$

Then the price of a *zero-coupon bond* at time t that matures at time T is given by:

$$
P\left(t, T\right) = \exp\left(A\left(t,\ T\right) - B\ \left(t, T\right) X\right) \tag{25.11}
$$

The corresponding *matrix Riccati equation* is given by:

$$
\begin{aligned}
&\frac{dB}{dt} = \frac{1}{2} B^{\top} H B - K_1^{\top} B - a \\
&\frac{dA}{dt} = K_0 B \\
&\text{where:} \\
&\begin{cases}
B = B\left(t, T\right),\ A = A\left(t,\ T\right) \\
a = \left(1,\ \ldots,\ 1\right)^{\top} \\
H = \left(H_{ij}\right) \in \mathbb{R}^{n\times n} \text{ and} \\
H_{ij}\left(x\right) = \left(\sigma\left(x\right)\sigma(x)^{\top}\right)_{ij,} \quad 1 \le i, j \le n.
\end{cases}
\end{aligned}
\tag{25.12}
$$

This system of equations can be solved numerically using *odeint*.

25.5 THE METHOD OF LINES: WHAT IS IT AND WHAT ARE ITS ADVANTAGES?

In Chapters 20 to 23 we introduced the finite difference method (FDM) for time-dependent convection–diffusion–reaction PDEs. The approach was to discretise the PDE in space and time simultaneously and then solve the resulting linear or nonlinear system of equations at each time step. This is a generally accepted practice in computational finance. In this section we take a different approach by discretising in the space direction using only standard centred

divided differences to approximate the diffusion and convection terms on uniform or non-uniform meshes. The time variable is still continuous and what we are left with is a system of ODEs. In the finite element method (FEM) this process is called *semi-discretisation* (Strang and Fix, 1973; Topper, 2005; Buchanen, 1994).

We motivate the MOL by examining the initial boundary value problem for the one-dimensional heat equation with Dirichlet boundary conditions.

The problem is:

$$\begin{aligned} &\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}, \quad 0 < x < 1, \quad t > 0 \\ &u(0,\, t) = u(1, t) = 0, \quad t > 0 \\ &u(x,\, 0) = f(x), \quad 0 \le x \le 1. \end{aligned} \tag{25.13}$$

We now partition the x interval $(0, 1)$ into J subintervals and approximate (25.13) by the *semi-discrete scheme*:

$$\begin{aligned} &\frac{du_j}{dt} = h^{-2}\left(u_{j+1} - 2u_j + u_{j-1}\right), \qquad 1 \le j \le J-1 \\ &u_0 = u_j = 0, \quad t > 0 \\ &u_i(0) = f\left(x_i\right),\ 0 \le j \le J. \end{aligned} \tag{25.14}$$

We define the following vectors:

$$\begin{aligned} &U(t) = (u_1(t), \ldots, u_{J-1}(t))^{\top} \\ &U_0\,(f(x_1), \ldots, f(x_{J-1}))^{\top}. \end{aligned}$$

Then we can rewrite system (25.14) as an ODE system:

$$\begin{aligned} &\frac{dU}{dt} = AU, \quad t > 0 \\ &U(0) = U_0 \end{aligned} \tag{25.15}$$

where the matrix A is given by:

$$A = h^{-2}\begin{pmatrix} -2 & 1 & & 0 \\ 1 & \ddots & \ddots & \\ & \ddots & \ddots & 1 \\ 0 & & 1 & -2 \end{pmatrix}.$$

Where do we go from here? There are a number of questions we would like to ask, for example: Q1: Does system (25.15) have a unique solution and what are its qualitative properties? Q2: How accurate is scheme (25.15) as an approximation to the solution of system (25.13)? Q3: How do we discretise (25.15) in time and how accurate is the discretisation?

These questions can be answered by the results from this chapter and Chapter 24. The test case (25.13) can be extended to more general PDEs in one, two and three space dimensions. Some of the advantages of using the MOL are:

1. It can be applied to a wide range of scalar (and vector) time-dependent PDEs, including Dirichlet, Neumann and other kinds of boundary conditions.

2. The resulting system of ODEs is amenable to analysis, including existence and uniqueness results and qualitative properties of the ODE system.
3. High-accuracy, professional and robust open-source and commercial ODE solvers are available. There is no need to create your own home-grown solvers. Most developers are not full-time numerical analysts working on the numerical solution of ODEs, and it is better to outsource this part of the project rather than create your own solvers.
4. The MOL can be applied to nonlinear PDEs and PDEs with discontinuous initial conditions (payoff). In particular, special ODE solvers can be used to solve *stiff* ODEs.
5. The MOL is a competitor to the ADI and 'Soviet Splitting' methods in the sense that splitting an n-dimensional PDE into a sequence of one-dimensional PDEs may be redundant. ADI and splitting methods are more difficult to apply than the MOL to nonlinear and stiff problems.
6. Time-to-market and time-to-develop: it takes less time to write and debug code that uses the MOL and ODE solvers than handcrafted code. This is what we have found compared with the amount of time it took us to write the solvers in Chapters 20 to 23.
7. For PDE novices it is easier to apply the finite difference method by delegating to ODE solvers rather than trying to handcraft, test and debug your own Crank–Nicolson solver, at least in the short term.

We mention that the MOL in the current context is the VMOL (*Vertical MOL*) because discretisation takes place in a vertical direction. This is in contrast to the HMOL (*Horizontal MOL*, also known as *Rothe's method*) where discretisation takes place in a horizontal direction. A discussion of this method can be found in Ladyženskaja, Solonnikov and Ural'ceva (1988) and Meyer (2015). For completeness, we show how to discretise system (25.13) using the HMOL. We discretise in time using the BTCS (implicit Euler) scheme. This leads to a *two-point boundary value problem* at each time level for the function $u^n \equiv u^n(x)$:

$$\begin{aligned} &\frac{u^{n+1} - u^n}{\Delta t} = \frac{d^2 u^{n+1}}{dx^2}, \quad n = 0, \ldots, N-1 \\ &u^{n+1}(0) = u^{n+1}(1) = 0 \\ &u^0(x) = f(x), \ 0 \le x \le 1. \end{aligned} \tag{25.16}$$

We can now approximate the solution of (25.16) using the finite difference method, matrix solvers and interpolators that we introduced in Chapter 13.

25.6 INITIAL FORAY IN COMPUTATIONAL FINANCE: MOL FOR ONE-FACTOR BLACK–SCHOLES PDE

Having discussed Boost *odeint* and the MOL, we now turn our attention to creating our first option pricing code using these techniques. We reduce the scope by pricing a put option with early-exercise features. We deliberately avoid trying to over-engineer the solution and we produce a solution using global data and functions. The goal is to get accurate prices in the short term. For this problem, we can take it that the price P = 6.2995968 is the reference price which the various solvers should emulate.

The main modules in our code are:

- Option data.
- The functions comprising the Black–Scholes PDE (we use lambda functions).
- Modelling the early-exercise constraint using the Courant *quadratic penalty function* (Fiacco and McCormick, 1968).
- A C++ class to model the Black–Scholes PDE and that we use with a Boost *odeint* integrate function (we examine Cash–Karp and Bulirsch–Stoer solvers).
- Solving the ODE system.
- Displaying the results in Excel.

We now describe the code that implements each of these blocks of code. First, the relevant global parameters are:

```
using value_type = double;
// The type of container used to hold the state vector
typedef std::vector<value_type> state_type;

// Option Data
value_type K = 50.0;
value_type T = 1.0;
value_type r = 0.08;
value_type d = 0.0;
value_type sig = 0.40;
```

We model the Black–Scholes PDE as lambda functions:

```
// Functions
auto diffusion =  [] (value_type x, value_type t)
                        {return 0.5*sig*sig*x*x;};
auto convection = [](value_type x, value_type t)
                        { return (r - d)*x; };
auto reaction = [](value_type x, value_type t)
                        { return -r; };


// BC
auto bcl = [](value_type t) // BCL
{
    return K*std::exp(-(r - d)*(t));
};

auto bcr = [](value_type t) // BCR
{
    return 0;
};

auto payoff = [] (value_type x)
{
    return std::max<value_type>(K - x, 0.0);
};
```

Concerning the early-exercise constraint we add a *penalty term* to the PDE. The approach is based on optimisation methods. More generally, consider the *constrained optimisation* problem:

$$\begin{aligned} &\min f(x), f : \mathbb{R}^n \to \mathbb{R}, \quad x \in \mathbb{R}^n \\ &\text{subject to} \\ &g_i(x) \geq 0, \quad j = 1, \ldots, m. \end{aligned} \tag{25.17}$$

The penalty method allows us to introduce a new parameter $\mu \to \infty$ and we get a new *unconstrained optimisation* problem:

$$\min_x \left(f(x) + \mu \sum_{j=1}^{m} \left| g_j(x) \right|^2 \right). \tag{25.18}$$

Moving to the Black–Scholes PDE, the equivalent early-exercise constraint is:

$$V \geq \varphi \text{ where } \varphi = \varphi(x) \text{ is the option payoff and } V \text{ is the option price} \tag{25.19}$$

which can be written in the more generic form to produce a penalty term:

$$g(V, x) = V - \varphi(x) \geq 0 \tag{25.20}$$

where V is the option price. All this mathematics led us to the C++ code for the penalty term:

```
// For American options
value_type penalty(value_type x, value_type u)
{
    // The rationale is that if correction < 0 the
    // constraint is satisfied and the penalty = 0.
    // Otherise, we have to include a non-zero penalty

    value_type correction = payoff(x) - u;

    // Constraint already satisfied
    if (correction <= 0.0)
       return 0.0;

    // A. (Courant) quadratic barrier function
    value_type lambda = 1.0e+12;
    value_type penalty = lambda*correction*correction;
    return penalty;
}
```

In general, we have replaced the *differential inequality*:

$$\frac{\partial V}{\partial t} \geq \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV \tag{25.21}$$

by the *semi-linear* PDE:

$$\frac{\partial V}{\partial t} = \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV + \lambda g(V,S) = 0 \tag{25.22}$$

and this is the equation that we approximate. We choose a large value for λ. In other words, we carry out a *semi-discretisation* in space and the resulting set of equations is encapsulated in a class:

```
class OdeBlackScholes
{ // Basic ODE class for plain Black Scholes


    std::vector<value_type> mesh;
    double h;

public:
    OdeBlackScholes(std::size_t NX, double meshSize)
       :mesh(std::vector<value_type>(NX+1,0.0)), h(meshSize)
    {
       mesh[0] = 0.0;
       for (std::size_t j = 1; j < mesh.size(); ++j)
       {
          mesh[j] = mesh[j - 1] + h;
       }
    }

    void operator() (const state_type& U, state_type& dUdt,
           const value_type t)
    {
       double xval, diff,con;

       value_type h2 = 1.0 / (h*h);
       value_type hm1 = 1.0 / (2.0*h);

       // Boundaries

       // Left boundary (j = 1)
       std::size_t index = 1;
       xval = mesh[index];
       dUdt[index]
          = diffusion(xval,t)*h2*(U[index + 1]-2.0*U[index] + bcl(t))
          + 0.5*convection(xval, t)*(U[index + 1] - bcl(t)) / h
          + reaction(xval, t)*U[index]
          + penalty(mesh[index], U[index]);

       // Right boundary (j = J-1)
       index = U.size() - 2;
```

```
        xval = mesh[index];
        dUdt[index]
           = diffusion(xval,t)*h2*(bcr(t)-2.0*U[index] + U[index - 1])
           + 0.5*convection(xval, t)*(bcr(t) - U[index - 1]) / h
           + reaction(xval, t)*U[index]
           +penalty(mesh[index], U[index]);

        // Interior of domain (1 < j < J-1)
        for (std::size_t index = 2; index < U.size() - 2; ++index)
        {

           xval = mesh[index];
           diff = diffusion(xval, t)*h2;
           con = convection(xval, t)*hm1;
           dUdt[index] =
                       (diff + con) *U[index + 1]
                       + (-2.0*diff + reaction(xval, t))*U[index]
                       + (diff - con) *U[index - 1]
                       + penalty(mesh[index], U[index]);
        }
     }

     void operator () (const state_type& U, const value_type t)
     {
        if (t >= T)
        { // To avoid explosion of output at the moment

           // Your code here

        }
     }
};


void write_out(const state_type& U, const value_type t )
{

   if (t >= T)
   { // To avoid explosion of output at the moment

      std::cout << "\nTime: " << t << std::endl;
      for (std::size_t j = 0; j < U.size(); ++j)
      {
         std::cout << "[" << j << "," << U[j] << "] ";
      }
   }
}
```

We can now configure and start the application:

```
int main()
{
    namespace Bode = boost::numeric::odeint;

    // Input data
    const value_type T_0 = 0.0;
    const value_type dt = 1.0e-1;

    const value_type A = 0.0;
    value_type B = 6 * K;
    const int NX = 20*static_cast<int>(B);
    value_type h = (B - A) / static_cast<value_type>(NX);

    // Initial condition; discretise IC
    state_type U(NX+1); // [0,J]
    U[0] = bcl(0.0); U[U.size()-1] = bcr(0.0); // Compatibility

    value_type x = A + h;
    for (std::size_t j = 1; j < U.size() - 1; ++j)
    {
        U[j] = payoff(x); x += h;
    }

    // Integration_class
    OdeBlackScholes ode(NX, h);

    Bode::bulirsch_stoer<state_type, value_type> myStepper;
    std::size_t steps = Bode::integrate_adaptive
             (myStepper, ode, U, T_0, T, dt, write_out);
    std::cout << "Steps: " << steps << '\n';

    // std::size_t steps = Bode::integrate(ode, U, T_0, T, dt, write_out);
    // std::cout << "Steps: " << steps << '\n';

    ExcelDriver xl; xl.MakeVisible(true);
    xl.CreateChart(ode.mesh, U, "Barrier option");
}
```

25.7 BARRIER OPTIONS

Barrier options are options where the payoff depends on whether the underlying asset's price reaches a given level during a certain period of time before expiration. There are two kinds of barriers:

- *In barrier*: this is reached when the asset price S hits the barrier value H before maturity. In other words, if S never hits H before maturity then the payout is zero.

- *Out barrier*: this is similar to a plain option except that the option is knocked out or becomes worthless if the asset price S hits the barrier H before expiration. This is an option that is knocked out if the underlying asset touches a lower boundary L or upper boundary U, prior to maturity.

The above examples were based on constant values for U and L. In other words, we assume that the values of U and L are time independent. This is a major simplification; in general, U and L are functions of time, $U = U(t)$ and $L = L(t)$. In fact, these functions may even be discontinuous at certain points. For more detailed information, see Haug (2007).

We concentrate on one-factor barrier options described by the following partial differential equation:

$$-\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + r\frac{\partial V}{\partial S} - rV = 0.$$

In contrast to plain options we now need to specify two boundary conditions at finite values of S. For a *double barrier option* two finite boundaries are specified:

$$\begin{aligned} V(A,\, t) &= g_0(t), \quad 0 < t < T \\ V(B,\, t) &= g_1(t), \quad 0 < t < T \end{aligned} \tag{25.23}$$

where g_0 and g_1 are given functions of t.

Here A and B are specific values of the underlying S and we assume that these barriers are constant. More generally, *time-dependent barriers* $L(t)$ and $U(t)$ are defined in boundary conditions as:

$$\begin{aligned} V(L(t),\, t) &= g_0(t), \quad 0 < t < T \\ V(U(t),\, t) &= g_1(t), \quad 0 < t < T. \end{aligned}$$

For single barriers (we are only given one barrier) we have to decide on how to define the other barrier. Given a positive single barrier we can then choose between $S = 0$ or some large value for S, depending on the kind of barrier.

There are a number of scenarios when working with single barrier options. For example, we view a single *up-and-out* barrier as a double barrier option with rebate of value 0 at the down-and-out barrier (that is, when $S = 0$). In this case the company whose stock is being modelled is probably bankrupt and is therefore unable to recover. Another example is a *down-and-out* call option in which case we need to truncate the semi-infinite domain. In this case we take the boundary conditions as follows:

$$V(S_{\max,}\, t) = S_{\max} - Ke^{-r(T-t)} \tag{25.24}$$

where $S_{\max}$ is 'large enough'.

The payoff function is the initial condition and is given by:

$$V(S, 0) = \max(S - K, 0). \tag{25.25}$$

We shall now examine how to approximate barrier option problems by finite difference methods.

25.8 USING EXPONENTIAL FITTING OF BARRIER OPTIONS

The exponentially fitted schemes were developed specifically for boundary layer problems and convection–diffusion equations whose solutions have large gradients in certain regions of the domain of interest (Duffy, 1980). In particular, the schemes are ideal for approximating the solution of PDEs that describe barrier options. We use *exponential fitting* in the space S direction and implicit Euler time marching in the t direction. If needed, we can employ extrapolation techniques in the time direction in order to promote accuracy. For convenience, we write the Black–Scholes equation in the more general and convenient form:

$$LV \equiv -\frac{\partial V}{\partial t} + \sigma(S,t)\frac{\partial^2 V}{\partial S^2} + \mu(S,t)\frac{\partial V}{\partial S} + b(S,t)V \tag{25.26}$$

where:

$$\sigma(S,t) = \frac{1}{2}\sigma^2 S^2,\ \mu(S,t) = rS,\ b(S,t) = -r.$$

The corresponding fitted scheme is now defined as:

$$L_k^h V_j^n \equiv -\frac{V_j^{n+1} - V_j^n}{k} + \rho_j^{n+1} D + D - V_j^{n+1} + \mu_j^{n+1} D_0 V_j^{n+1} + b_j^{n+1} V_j^{n+1},\ 1 \le j \le J-1 \tag{25.27}$$

where:

$$\rho_j^n \equiv \frac{\mu_j^n h}{2} \coth \frac{\mu_j^n h}{2\sigma_j^n}.$$

We define the discrete variants of the initial condition and boundary conditions (25.25) and (25.23) and we realise them as follows:

$$V_j^0 = \max(S_j - K, 0), \quad 1 \le j \le J-1$$

and

$$\left.\begin{array}{l} V_0^n = g_0(t_n) \\ V_J^n = g_1(t_n) \end{array}\right\} \quad 0 \le n \le N.$$

The system can be cast as a linear matrix system:

$$A^n U^{n+1} = F^n,\ n \ge 0 \tag{25.28}$$

with U^0 given and we solve this system using LU decomposition, for example.

25.9 SUMMARY AND CONCLUSIONS

In this chapter we have given a short introduction to the MOL that allows us to transform a time-dependent PDE into a system of ODEs by discretising the former's space variables while

keeping the time variable continuous. We then used the Boost *odeint* library to discretise the ODE system in time. There are several solvers in the library.

We have given both generic examples and applications to computational finance. Finally, we conclude this chapter with a number of exercises and projects that have been discussed in Chapters 20 to 24. What we have done now is to revisit them from the perspective of the MOL.

25.10 EXERCISES AND PROJECTS

1. (ADE versus MOL)

We re-examine the test case of the two-dimensional heat equation with Dirichlet boundary conditions that we introduced in Exercise 22.3 and that we solved using the ADE method. In the current exercise we solve the same problem using the MOL (parts a), b) and c) of Exercise 22.3).

Answer the following questions:

a) Implement the problem using the MOL and `odeint` (use the Cash–Karp integrate function that is called `integrate()`).

b) Which scheme takes more effort to program?

c) Compare the relative accuracy and run-time performance of the two schemes for given mesh sizes in the x and y directions.

d) Run the MOL scheme using the Bulirsch–Stoer and fourth-order Runge–Kutta methods (we discussed the latter method in Section 24.6.2). Again, compare the relative accuracy and run-time performance of the two schemes.

2. (A Generic One-Factor Software Framework Based on Chapter 20)

In Section 25.6 we developed a prototype to price one-factor plain options using `odeint`. No attempt was made at creating a generic solution because the aim was to have a *proof-of-concept* solution to test the accuracy of the *odeint* library. In particular, we used the Bulirsch–Stoer method.

We now create a more generic solution.

Answer the following questions:

a) Identify the separate concerns and create an initial context diagram as discussed in Chapter 9. In particular, identify the major modules, their responsibilities and interfaces.

b) Determine how to model system (20.14)–(20.16) as one or more C++ classes. One choice is the class hierarchy that we introduced in Section 21.6.

c) Generalise the class OdeBlackScholes to accommodate the classes in part b). This now means that we are in a position to model a range of convection–diffusion–reaction PDEs using `odeint`.

d) The code in part c) should support *exponential fitting* as discussed in Sections 20.8 and 25.8 as well as a scheme that does not use exponential fitting.

e) Test the code on the Black–Scholes equation by using the integrate functions and observers from Section 25.2.3.

f) Summarise your findings on the accuracy, efficiency and robustness of your solution.

3. (MOL with Neumann and Other Boundary Conditions)

Some PDE problems can have associated Robin or Neumann boundary conditions (see, for example, equation (20.9)) and we must devise a way to incorporate them

into the MOL. To this end, we examine the Neumann boundary condition at the right boundary:

$$\frac{du}{dx} = g(t) \text{ at } x = B \text{ where } g \text{ is a given smooth function.}$$

We discretise this equation in space to give:

$$\frac{u_J(t) - u_{J-1}(t)}{h} = g(t) \text{ at } x = B \text{ where } J \text{ is the number of steps.}$$

Differentiating this equation with respect to time leads to an ODE:

$$\frac{du_J}{dt} - \frac{du_{J-1}}{dt} = h\frac{dg}{dt} \text{ at } x = B.$$

Another popular choice in computational finance is the *linearity boundary condition* that describes the option price in the far field:

$$\frac{\partial^2 u}{\partial x^2} = 0 \quad \text{at } x = B.$$

Many applications use non-Dirichlet boundary conditions and the above trick is one way to incorporate them into the MOL. We discuss this issue in the current exercise.

Answer the following questions:

a) Apply the MOL to a one-factor heat equation with a Dirichlet boundary condition at one end and a Neumann boundary condition at the other end. Compare the accuracy with that produced by the Crank–Nicolson method as discussed in Chapter 13.

b) Discretise the CIR model (20.35) and in particular the boundary condition (20.38) using the MOL when the Feller condition (20.37) is satisfied. Compare the solution in as many ways as possible with the ADE solution discussed in detail in Exercise 21.2.

4. (The Heston Model (Heston, 1993; Duffy, 2006; Sheppard, 2007))

We discuss the application of the MOL to the Heston PDE model.

Since there are two factors in the Heston model we need two SDEs. First, the spot asset price satisfies the SDE:

$$dS_t = \mu S_t dt + \sqrt{\sqrt{v}(t)}\, S_t dW_t^{(1)}$$

where:

S_t = spot price
$W_t^{(1)}$ = a Wiener process
$v(t)$ = variance
μ = (risk-neutral) drift.

Second, the variance $v(t)$ satisfies an *Ornstein–Uhlenbeck* process defined by the SDE:

$$d\sqrt{v(t) = -\beta\sqrt{v(t)}}dt + \sigma dW_t^2.$$

It can be shown that:

$$dv(t) = \kappa[\theta - v(t)]\,dt + \sigma\sqrt{v(t)}dW_t^{(2)}$$

where:

σ = volatility of the variance
$0 < \theta$ = long-term variance
$\theta < \kappa$ = rate of mean reversion
$W_t^{(2)}$ = a Wiener process and ρ is the correlation value.

The correlation between the two Wiener processes is given by:

$$dW_t^{(1)}\,dW_t^{(2)} = \rho dt.$$

In general, an increase in ρ generates an asymmetry in the distribution while a change of volatility of variance σ results in a higher kurtosis. Finally (as discussed in Heston, 1993) the PDE for a contingent claim U is given by:

$$\frac{\partial U}{\partial t} + L_s U + L_v U + \rho\sigma v S\frac{\partial^2 U}{\partial S\partial v} = 0$$

where:

$$L_S U \equiv \frac{1}{2}vS^2\frac{\partial^2 U}{\partial S^2} + rS\frac{\partial U}{\partial S} - rU = 0$$

$$L_v U \equiv \frac{1}{2}\sigma^2 v\frac{\partial^2 U}{\partial v^2} + \{\kappa[\theta - v(t)] - \lambda(S, v, t)\}\frac{\partial U}{\partial v} = 0$$

and λ is the market price of volatility risk.

The boundary conditions in the case of call options are:

$$U(0, v, t) = 0 \ (S = 0)$$
$$\frac{\partial U}{\partial S}(\infty, v, t) = 1 \quad (S = \infty)$$
$$\frac{\partial U}{\partial t} + rS\frac{\partial U}{\partial S} - rU + \kappa\theta\frac{\partial U}{\partial v} = 0 \ (v = 0)$$
$$U(S, \infty, t) = S \ (v = \infty).$$

Answer the following questions:

a) Implement this PDE model using the MOL. In the first version you can assume that the *correlation is zero*. Compare your answers with the analytic solution or some other accurate approximation. Use centred differencing for both the diffusion and convection terms.

b) Add exponential fitting in both directions in the algorithm in part a) to produce a monotone scheme.

c) We now add correlation to the model. To this end, we use the *Yanenko approximation* to the mixed derivative (Duffy, 2006; Yanenko, 1971). Let $u_{i,j}$ denote the approximate option value. Then:

$$\frac{\partial^2 u}{\partial x \partial y}(x_i, y_j) \sim \frac{1}{4h_x h_y}\left(u_{i+1,j+1} - u_{i+1,j-1} - u_{i-1,j+1} + u_{i-1,j-1}\right)$$

which can be motivated as follows:

$$\begin{aligned}
\Delta_x \Delta_y u_{ij} &= \frac{1}{2h_y}\Delta_x(u_{i,j+1} - u_{i,j-1}) \\
&= \frac{1}{4h_x h_y}[(u_{i+1,j+1} - u_{i-1,j+1}) - (u_{i+1,j-1} - u_{i-1,j-1})] \\
&= \frac{1}{4h_x h_y}(u_{i+1,j+1} - u_{i-1,j+1} - u_{i+1,j-1} + u_{i-1,j-1})
\end{aligned}$$

where:

$$\begin{aligned}
\Delta_x u_{i,j} &= \frac{u_{i+1,j} - u_{i-1,j}}{2h_x} \\
\Delta_y u_{i,j} &= \frac{u_{i,j+1} - u_{i,j-1}}{2h_y}.
\end{aligned}$$

Compute the solution for a range of values of the correlation term.

5. (Accuracy Testing of American Option Pricing; Alternatives to the Brennan–Schwartz Method)

We investigate a number of methods to compute the price of put options with early-exercise features. We assume zero dividends. One of the most popular methods is the *Brennan–Schwartz algorithm* (Brennan and Schwartz, 1977). This approach solves a tridiagonal system of equations as in the case of European options, except that at each time level we must satisfy the constraint in equation (25.20). This is the approach that we took in Chapters 11 and 12, where we priced American options using lattice methods. In particular, we check the constraint at each node of the lattice during the backward induction algorithm. We recall:

```
auto Payoff = [&K] (double S)-> double
                    {return std::max<double>(K-S, 0.0);};
// The early exercise constraint
auto AmericanAdjuster = [&Payoff](double& V, double S)->void
{ // e.g. early exercise

    V = std::max<double>(V, Payoff(S));
};
```

The same approach can be applied to the finite difference method and it is an efficient means of pricing American options. We present some benchmark results based on this latter model:

- The Black–Scholes model with sigma = 0.2, r = 0.1, K = 100, S = 100 and T = 0.25 is used. The truncation boundary is at S = 400.
- The discretisation is performed on non-uniform time–space grids with central finite differences and Rannacher time stepping.
- The resulting LCPs are solved using the Brennan–Schwartz algorithm.
- Numerical results on six different grids are (time steps, space steps and running time):

  {10; 40; 3.0106496; -5.9e-02 0.1}, {18; 80; 3.0553799; -1.5e-02 0.3},

  {34; 160; 3.0663983; -3.7e-03 0.9}, {66; 320; 3.0691617; -9.5e-04 3.6},

  {130; 640; 3.0698666; -2.4e-04 14.2}, {258; 1280; 3.0700463; -6.0e-05 57.2}.

The code for this scheme was handcrafted and optimised for the problem at hand. So, we expect that general solvers will be less efficient.

Answer the following questions:

a) Compare the above results with those produced by the ADE method, that is using the same option data and mesh sizes.

b) Execute the same experiment as in part a) using the MOL (for example, using Cash–Karp and other *integrate functions* in Boost *odeint*). Regarding the early-exercise feature, examine how to embed the Brennan–Schwartz algorithm in the code (probably in an *observer*).

c) Execute the same experiment as in part b) and in this case use the semi-linear PDE defined in equation (25.22). Solve the problem using the MOL in combination with the barrier/penalty method.

6. (Stress Testing American Put Options, Section 25.6)

In this exercise we wish to test the accuracy of the MOL in combination with the Courant quadratic penalty function as shown in equation (25.18) (see Fiacco and McCormick, 1968 for more details). The objective of the exercise is to use the Cash–Karp ODE solver with default parameter values and to take a larger number of steps NS in the underlying until we get the same answer as the following test cases:

Case A: S = K = 50, T = 1, v = 0.4, r = 0.08, d = 0.0; p = 6.299597.

Case A: S = 120, K = 100, T = 3, v = 0.2, r = 0.08, d = 0.08; p = 5.929805.

Answer the following questions:

a) Experiment with different values of NS. Which value of NS is needed to reproduce the put prices in cases A and B? (We have tested these cases with the code from Section 25.6 and we get accurate results; we do not say which value of NS that we used.)

b) In the code we set the penalty parameter value to 1.0e+12. Is this optimal? Do other (smaller, larger) values give better results, for example can we get comparable results with smaller values of NS?

c) Now use the Cash–Karp ODE solver in combination with an observer (see Section 25.2.3). In the observer code implement the Brennan–Schwartz algorithm. How many steps NS do you need in this case to emulate the put prices in cases A and B?

d) Run cases A and B again using the Bulirsch–Stoer method as stepper. Is this approach more efficient or accurate than the method in part a)?

e) How many steps in the binomial method are needed if we wish to get 6-digit accuracy? Apply the lattice methods of Chapters 11 and 12 to this problem. Anecdotal evidence suggests that between two and four million steps are needed using the binomial method.

f) How many steps in the ADE method are needed if we wish to get 6-digit accuracy? Apply the methods of Chapters 20 to 23 to this problem.

As indication, for NS = 2700 we get P = 6.28858, for NS = 5400 we get P = 6.299595 and for NS = 10800 we get P = 6.299596.

7. (Barrier Options)

We have introduced barrier options and their numerical approximation using BTCS (fully implicit scheme) and exponential fitting in Section 25.7 and we produced several tables of values for reference purposes. In this exercise we wish to price barrier options using the MOL.

Answer the following questions:

a) Compare the possible advantages of the MOL with the finite difference schemes in Chapters 20 and 21 with regard to ease of programming, supporting functionality (for example, the eight barrier types, their prices and sensitivities), continuous and discrete monitoring and robustness (for example, behaviour near a barrier close to expiry).

b) Create a software framework using the MOL as a variation of the code in these chapters. Pay particular attention to modelling boundary conditions, rebates and so on.

c) Write code to numerically compute the delta and gamma of barrier options. Display them using the Excel driver as discussed in Chapters 14 and 22 (in particular, Section 22.4).

d) Based on the output from part c), compare values with plain options from a financial viewpoint (see Derman and Kani, 1996).

8. (Compound Options, Project)

The objective of this exercise is to price compound options using the binomial method. A *compound option* is an option on an option. Its payoff involves the value of another option executed either concurrently or in sequence. Thus, a compound option has two expiration dates and two strike prices in general. These options are used for currency and fixed-income markets and to hedge bids for business projects that may or may not be accepted (Mun, 2002). There are two steps in pricing a compound option:

- Price the underlying option.
- Price the compound option.

For example, the payoff of a *call on a call option* with exercise price K_1 for the underlying and K_2 for the compound option is given by:

$$\begin{aligned} F(S) &= \max(S - K_1, 0) \\ G(V_2) &= \max(V_2 - K_2, 0). \end{aligned} \tag{25.33}$$

A compound option gives the owner the right to buy or sell another option. The four basic types are call on call, call on put, put on put and put on call.

We describe compound option pricing using PDEs. The PDE for the underlying option with expiration T is given by:

$$\begin{aligned} &\frac{\partial V_1}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V_1}{\partial S^2} + rS\frac{\partial V_1}{\partial S} - rV_1 = 0 \\ &V_1(S, T_1) = F(S). \end{aligned} \tag{25.34}$$

Let T_2 be the expiration of the compound option. We compute $V(S, T_2)$. The PDE to compute the compound option is given by:

$$\begin{aligned} &\frac{\partial V_2}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V_2}{\partial S^2} + rS\frac{\partial V_2}{\partial S} - rV_2 = 0 \\ &V_2(S, T_2) = G(V_1(S, T_1) - K_2) \text{ where } T_2 < T_1. \end{aligned} \tag{25.35}$$

Compound options can be priced in a number of ways:

- S1: Analytical formulae (Haug, 2007).
- S2: Binomial method (Mun, 2002).
- S3: Using the finite difference method, in particular the MOL which is the focus of this exercise.

We first show the C++ code to price this problem using the binomial method (it is based on Mun, 2002, pp. 185–195). We create three lattices; the first lattice is for the underlying asset, the second lattice is for the equity option while the third lattice corresponds to the compound option. We create one lattice object (for efficiency reasons) to which we apply one forward induction and two backward induction algorithms. In general, we need to do the following (Mun, 2002):

- Choose a spot price and build the *underlying lattice* based on a forward induction algorithm such as the Tian or CRR model.
- Create an *equity lattice* for the *base option* that extends to expiry T_1 using the *base option payoff* as input.
- Create the *option valuation lattice* up to expiry T_2 and perform backtracking using the *compound option payoff* as input.

The code that implements the complete algorithms is based on the functionality from Chapters 11 and 12 (notice that the Tian model is hard-coded in this version):

```
double CompoundOption(const FunctionType& compoundPayoff, double T1,
                  const FunctionType& basePayoff,
                  const OptionData& opt, std::size_t N, double S0)

{

// Steps:
//
// 1. Create underlying lattice.
// 2. Create equity lattice.
// 3. Create compound option lattice.

// 1.
double dt = opt.T / static_cast<double>(N);
std::cout << "dt " << dt << std::endl;

const int TYPEB = 2;

// The function that implements forward and backward induction
LatticeMechanisms::TianLatticeAlgorithms algorithm(opt, dt);
```

```
// Create basic structure for plain options
Lattice<double,TYPEB> lattice(N, 0.0); // init
LatticeMechanisms::ForwardInduction<double>(lattice, algorithm, S0);

// 2.
// Price a plain option
double V0 = LatticeMechanisms::BackwardInduction<double>
       (lattice, algorithm, basePayoff);

// 3.
// a. Find lattice index corresponding to t = T1.
// b. Backtrack from that T1 index.

// a. Find index at t = T1
std::size_t indexT1 = static_cast<std::size_t>(T1/dt);

// b.
double CV0 = LatticeMechanisms::BackwardInduction<double>
          (lattice, algorithm, compoundPayoff, indexT1);

return CV0;
}
```

We now take the example from Haug (2007, p. 136) (put-on-call). The input is:

```
// Option data for 'base' option
OptionData opt;

// Haug 2007 page 136
opt.K = 520.0;
opt.T = 0.5;
opt.r = 0.08;
opt.q = 0.03;
opt.sig = 0.35;
double S0 = 500.0;
std::size_t N = 13000;

// Payoff as a lambda function
double K2 = opt.K;
auto baseCallPayoff = [&K2] (double S)-> double
       {return std::max<double>(S - K2, 0.0);};


double K1 = 50.0;
auto compoundPutPayoff = [&K1] (double V)-> double
       {return std::max<double>(K1 - V, 0.0);};
double T1 = 0.25;

double compoundOptionValue = CompoundOption
           (compoundPutPayoff, T1,  // Compound option
```

```
baseCallPayoff, opt,      // Equity option
N,                        // Lattice size
S0);                      // Underlying value
```

The answer that we get is 21.1959.

Answer the following questions:

a) Price this problem for the call-on-call option based on the following data (Mun, 2002, p. 187):

```
opt.K = 900.0; // Mun 2002 page 187
opt.T = 3.0;
opt.r = 0.077;
opt.q = 0.0;
opt.sig = 0.30;
double S0 = 1000.0;
double K1 = 500.0;
```

The answer that we get is 165.174.

b) Modify the code of the function `CompoundOption()` to include other forward induction algorithms besides the Tian method (as discussed in Chapters 11 and 12). Test the schemes.

9. (Compound Options, PDE Approach)

The objective of this exercise is to price compound options using the finite difference method. We have already explained the background in Exercise 8. The base and compound option payoff PDEs have already been defined by equations (25.33), (25.34) and (25.35).

Answer the following:

a) Write the system (25.33), (25.34) and (25.35) as initial boundary value problems, as in Chapters 20 to 23.

b) Create a software framework for this problem using the MOL. You need to take care of the computed discrete payoff of the base option. Compare the results with those produced in Exercise 8.

c) Solve system (25.33), (25.34) and (25.35) using the ADE method.

10. (Chooser Options)

A *chooser option* allows the purchaser to decide whether a derivative will be a European call or put option. These option types can be priced using lattice methods as discussed in Mun (2002) and Exercise 8 using the binomial method, but in this exercise we use PDE models. The underlying PDEs for the two underlying options are given by:

$$\frac{\partial V_j}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V_j}{\partial S} + rS\frac{\partial V_j}{\partial S} - rV_j = 0, j = 1, 2 \tag{25.36}$$

having strike levels K_1 and K_2, respectively (this is called a *complex chooser option*). The chooser option has the PDE:

$$\frac{\partial C_h}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 C_h}{\partial S^2} + rS\frac{\partial C_h}{\partial S} - rC_h = 0 \tag{25.37}$$

with payoff defined by:

$$C_h(S, T_2) = \max(V_1(S, T_2) - K_1, V_2(S, T_2) - K_2, 0). \tag{25.38}$$

Answer the following questions:

a) Solve this system using the MOL. Test the code using the following data (Haug, 2007, p. 130) using underscores for calls and put:

$$S = 50$$
$$K_c = 55;\ K_p = 48$$
$$T_c = 0.5;\ T_p = 0.5833$$
$$T = 0.25 \text{ (chooser)}$$
$$r = 0.1;\ b = 0.1 - 0.05;\ \sigma = 0.35.$$

b) The price of a *complex chooser option* has an analytic solution that can be written in terms of the bivariate cumulative normal distribution (Haug, 2007; Mun, 2002; Rubinstein, 1991) that we discussed in Chapter 16. Implement this formula and compare the results with those produced in part a).

c) A *simple chooser option* gives the holder the right to choose whether an option is to be a standard call or put after a time t_1 with strike K and time to maturity T_2 ($t_1 < T_2$). Implement the formula as in Haug (2007) and Rubinstein (1991).

11. (Viscous Burgers' Equation)

In this exercise we examine the viscous Burgers' equation that we introduced in Exercise 21.4. It is a nonlinear wave equation with diffusion added. It models fluid flow and is a simple nonlinear scheme for numerical experiments.

The objective of this exercise is to apply the MOL to this problem. Compare the results with those based on the ADE method.

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 26

Random Number Generation and Distributions

26.1 INTRODUCTION AND OBJECTIVES

In this chapter we introduce the *C++ random library*. The ability to generate 'random numbers' is useful in applications such as the simulation of natural and artificial phenomena, sampling, numerical analysis, decision making, cryptography and computer programming (for an account of random numbers, their construction and applications, see for example Knuth, 1997).

Traditionally, C++ has had limited support for the production of random numbers. It adopted the corresponding functionality from the C library, for example the function `rand()` that produces *pseudo-random numbers*. This function has so many shortcomings and it is not recommended for serious applications. Instead, we use *random number engines* that produce unpredictable (random) bits, thus ensuring that the likelihood of producing a 0 bit is the same as the likelihood of producing a 1 bit. Two popular engines are the *Mersenne Twister* and *linear congruential* algorithms. The C++11 engines deliver uniformly distributed numbers but they should not be used as raw data in applications. Instead, they are used as input to functionality in C++11 to transform uniform variates to random variates of a given *statistical distribution*, for example the normal and chi-squared distributions. C++11 supports 20 distributions that are classified into five categories.

The C++ random number library is easy to use and we shall show some examples in this chapter. In Chapters 31 and 32 we shall use it as one of the components in a Monte Carlo software framework option pricer.

More extended functionality can be found in the *Boost Random* C++ library. It contains a number of random number generators (for example, *lagged Fibonacci generators*) and distributions that are not in C++11. In most cases, this functionality should mitigate the need to create home-grown generators or to use proprietary software. Of course, there are always special cases in which the standard methods need to be replaced by specialised ones.

We have already given some discussion and applications of C++ `<random>`. In particular, Section 15.5 of Chapter 15 is relevant.

We also briefly discuss computing correlated vectors of random numbers using Cholesky decomposition in the *Boost uBLAS* and *Eigen* matrix libraries in this chapter.

26.2 WHAT IS A RANDOM NUMBER GENERATOR?

We give an introduction to a number of methods to generate random numbers. It is useful to give some mathematical background to the C++ code that we introduce.

26.2.1 Uniform Random Number Generation

Our starting point is the generation of numbers having a *uniform distribution*. To this end, let us suppose that X is a continuous random variable assuming all values in the closed interval $[a, b]$, where a and b are finite. If the *probability density function* (pdf) of X is given by:

$$f(x) = \begin{cases} \dfrac{1}{b-a}, & a \le x \le b \\ 0, & \text{otherwise} \end{cases} \tag{26.1}$$

then we say that X is *uniformly distributed* over the interval $[a, b]$. A shorthand notation is to say that X is $U(a, b)$. We generate uniformly distributed random numbers by using an algorithm that has been programmed in C or C++.

We now introduce two methods that were popular a number of years ago. We include them here for historical reasons. We also introduce them in the context of the numerical solution of stochastic differential equations (SDEs) in Chapters 31 and 32.

26.2.2 Polar Marsaglia Method

This method uses the fact that if the random variable U is $U(0, 1)$ then the random variable V defined by $V = 2U - 1$ is $U(-1, 1)$. We now choose two variables defined by:

$$V_j = 2U_j - 1 \quad \text{where} \quad U_j \sim U(0, 1), \quad j = 1, 2.$$

Then we define:

$$W = V_1^2 + V_2^2 \le 1, \quad W \sim U(0, 1).$$

We keep trying with different values until this inequality is satisfied, in other words until $W > 0.0$ and $W^2 \le 1$. Continuing, we define the intermediate value:

$$Y = \sqrt{-2 \log(W) / W}.$$

Finally, the pair of values defined by:

$$N_j = V_j Y, \quad j = 1, \ 2 \tag{26.2}$$

constitutes two standard normally (Gaussian) distributed random variables.

26.2.3 Box–Muller Method

This method is based on the observation that if r and φ are two independent $U(0, 1)$ random variables then the variables:

$$\begin{aligned} N_1 &= \sqrt{-2\log r}\cos(2\pi\varphi) \\ N_2 &= \sqrt{-2\log\ r}\sin(2\pi\varphi) \end{aligned} \tag{26.3}$$

are two independent standard Gaussian random variables. In general, the polar Marsaglia method is more efficient than the Box–Muller method.

We see that we can generate standard normal variates based on *standard uniform variates*. How do we generate these latter quantities? A popular method is the *linear congruential pseudo-random number generator* that is defined in recursive form (Kloeden, Platen and Schurz 1997):

$$X_{n+1} = aX_n + b \ (\text{mod}\ c) \ \text{ with } X_0 \text{ given (the seed).} \tag{26.4}$$

This formula generates a sequence of integer values in the range $(0, c-1)$ and these are the remainders when the term $aX_n + b$ is divided by c. Different values of a, b, c and X_0 will lead to different sequences. In practice, the choice depends on a number of factors such as the hardware being used, for example.

The parameters in the *one-step iterative scheme* (26.4) satisfy the following constraints:

$$\begin{aligned} &0 < c \ \text{(modulus)} \\ &0 \le a < c \ \text{(multiplier)} \\ &0 \le b < c \ \text{(increment)} \\ &0 \le X_0 < c \ \text{(the seed)}. \end{aligned} \tag{26.5}$$

Having produced the sequence in equation (26.4) we can now produce a sequence of numbers that seems to be uniformly distributed on the unit interval [0, 1] by defining the numbers:

$$U_n = X_n/c.$$

26.3 WHAT IS A DISTRIBUTION?

We are interested in modelling random variables. There are two categories: first, a *discrete random variable* whose *cumulative distribution function* (cdf) changes only in jumps and is constant between jumps. Associated with the cdf of a discrete variable is the *probability mass function* (pmf). Second, the cdf of a continuous random variable is also continuous and it has a derivative (called the piecewise continuous *probability density function* (pdf)) that exists except at possibly a finite number of points. We assume that the reader is familiar with these concepts (Hsu, 1997).

Each distribution has its own underlying data type, which is either an integer or a real number. It would seem that the C++11 implementation uses the *inverse transform method* to

generate random variables. Let X be a random variable with cumulative probability distribution function $F_X(x)$. Since $F_X(x)$ is a non-decreasing function, the inverse function $F_X^{-1}(y)$ is defined for any value between 0 and 1:

$$F_X^{-1}(y) = \inf\{x : F_X(x) \geq y\}, \quad 0 \leq y \leq 1.$$

In this case the variable y is given.

In order to create a random variate from a given distribution, let U be uniformly distributed on (0, 1). Then:

$$X = F_X^{-1}(U) \tag{26.6}$$

is a *random variate* of the given distribution.

In general, we need to use numerical methods (for example, the Newton–Raphson method or fixed-point iteration) to compute the random variate X in equation (26.6). In special cases it may be possible to give an analytical solution, but this is more the exception than the rule. For completeness, we give some examples.

26.3.1 Analytical Solutions for Random Variate Computations

We examine the one-dimensional *uniform distribution* on the interval (a, b) whose pdf is given by:

$$f_X(x) = \begin{cases} \dfrac{1}{b-a}, & a < x < b \\ 0, & \text{otherwise} \end{cases} \tag{26.7}$$

and the corresponding cdf is:

$$F_X(x) = \int_0^x f_X(y)dy = \begin{cases} 0, & x < a \\ \dfrac{x-a}{b-a}, & a \leq x < b \\ 1, & x \geq b. \end{cases} \tag{26.8}$$

Then, the random variate from the uniform distribution on the interval (a, b) is given by the equation:

$$X = F_X^{-1}(U) = a + (b-a)U. \tag{26.9}$$

An *exponential variate* X with parameter $\beta > 0$ has the density (pdf):

$$f(x) = f_X(x) = \begin{cases} \frac{1}{\beta}e^{-x/\beta}, & 0 \leq x < \infty, \quad \beta > 0 \\ 0, & \text{otherwise} \end{cases} \tag{26.10}$$

and is denoted by $\exp(\beta)$.

The mean and variance are given by:

$$E(X) = \beta$$
$$V(X) = \beta^2$$

respectively. We generate random exponential variates by the *inverse transform method*, as follows:

$$U = F_X(x) = \int_0^x \frac{1}{\beta} e^{-y/\beta} dy = 1 - e^{-x/\beta} \tag{26.11}$$

where U is a standard uniform variate. Then, since $1 - U$ and U have the same distribution, we have:

$$X = -\beta \log(1 - U) \text{ or } X = -\beta \log U. \tag{26.12}$$

26.3.2 Other Methods for Computing Random Variates

The *acceptance–rejection method* is due to John von Neumann and it consists of sampling a random variate from a target distribution by first generating candidates from a more convenient distribution. We then reject a random subset of the generated candidates. The rejection mechanism ensures that the accepted samples are indeed distributed according to the target distribution. The method is also applicable in higher dimensions.

The idea is as follows: suppose that we wish to generate samples from the distribution f in some set A. Let us also suppose that there is a density funtion g that is also defined in A satisfying the following inequality:

$$f(x) \leq cg(x) \quad \forall x \in A \tag{26.13}$$

where c is a known positive constant.

We generate a sample X from g (the *majorising function*) and we accept the sample with probability $f(X)/cg(X)$. We can implement this method by sampling U uniformly in the interval (0, 1) and accepting X if $U \leq f(X)/cg(X)$. If X is rejected, we continue sampling from g until the acceptance test is passed.

The algorithm can be stated as follows:

1. Generate X from distribution g.
2. Generate U as a uniform variate on the interval (0, 1).
3. If $U \leq f(X)/cg(X)$ return X, else go to step 1.

We give an example of finding a function g from which we can generate random variates. To this end, let us define the distribution:

$$f(x) = 3x^2, \quad 0 \leq x \leq 1.$$

We define the distribution g by $g(x) = 1,\ 0 \leq x \leq 1$ (this is a density of a $U(0, 1)$ distribution). Thus:

$$\sup_{0 \leq x \leq 1} \frac{f(x)}{g(x)} = 3 = c$$

and

$$\frac{f(x)}{cg(x)} = x^2.$$

Then the algorithm is:

1. Generate $U_1,\ U_2$ from $U(0,1)$.
2. If $U_2 \leq U_1^2$ accept U_1 as the random variable from f. Else, go to step 1.

This is not a very efficient algorithm as approximating quadratic functions by constants is not very accurate. Exercise 4 discusses a more accurate approach.

The second alternative to the inverse transform method is the *composition method*. We use this method when a distribution function F can be expressed as a linear combination of other distribution functions $F_1,\ F_2, \ldots$ such that:

$$F(x) = \sum_{j=1}^{\infty} p_j F_j(x), \quad p_j \geq 0, \quad \sum_{j=1}^{\infty} p_j = 1. \tag{26.14}$$

This method is useful when it is easier to sample from the component distributions than from the distribution F itself. The general steps to sample from F are:

1. Generate an index I such that $\text{Prob}(I = i) = p_i,\ i = 1, 2, \ldots$
2. Generate a random variate x with distribution function F_I.

We take a simple example:

$$f_X(x) = \frac{5}{12}[1 + (x-1)^4], \quad 0 \leq x \leq 2$$

that we can write as a weighted average:

$$f_X(x) = \frac{5}{6} f_1\ (x) + \frac{1}{6} f_2(x), 0 \leq x \leq 2$$

where:

$$f_1(x) = 1 = 2,\ f_2(x) = \frac{5}{2}(x-\ 1)^4,\ 0 \leq x \leq\ 2.$$

Some simple algebra shows that:

$$x = \begin{cases} 2U, \text{ if } U < 5/6 \\ 1 + \sqrt[5]{2U}, \text{ if } U \geq 5/6. \end{cases}$$

26.4 SOME INITIAL EXAMPLES

Having discussed the mathematical background to the generation of random variates we show how C++11 implements this functionality. The objective is to become acquainted with the functionality and to take some examples.

The first example shows how to create real-valued uniform random variates on an interval $[A, B]$ by choosing the *default engine* to generate random numbers. We then use this engine to produce variates of the desired distribution:

```
#include <random>

std::default_random_engine rng;

// Generate uniform random variates in interval [A, B]
double A = 0.0;
double B = 1.0;
std::uniform_real_distribution<double> dist(A, B);

int nTrials = 30;
for (int i = 1; i <= nTrials; ++i)
{ // Produce uniform variates

    std::cout << dist(rng) << ", ";
}
```

The following code creates a vector and we use the `shuffle()` algorithm to shuffle the order of its elements. We provide the engine with a seed value to produce unpredictable random values (if an engine does not receive a seed then it will produce the same results each time the vector is shuffled, in other words the values will be *replicated*; incidentally, this may be a desirable feature in some applications):

```
// Create and shuffle an array
std::vector<double> v = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };

std::default_random_engine rng;

// Seed, initialise just once
unsigned int mySeed = static_cast<unsigned int>(time(0));
eng.seed(mySeed);
std::shuffle(v.begin(), v.end(), eng);

for (std::size_t i = 0; i < v.size(); ++i)
{
    std::cout << v[i] << ", ";
}
```

Finally, another way to compute a seed for an engine is to use a *random device engine* that should guarantee randomness. An example is:

```
// Random device generates uniform integer random variables
std::random_device rd;
```

```
for (std::size_t i = 0; i < v.size(); ++i)
{
    std::cout << rd() << ", ";
}
```

In the next section we shall see how to use the random device engine to produce a single seed for another engine. In general, it is advisable to use it for this purpose and not for generating arrays of random numbers.

26.4.1 Calculating the Area of a Circle

Consider the square and quarter circle in Figure 26.1. We know their areas, namely r^2 and $\frac{1}{4}\pi r^2$, respectively. By dividing one area by the other, we see that we can estimate π. Now, we imagine throwing many darts at the areas in Figure 26.1. We count the number of times a dart falls in the quarter circle compared with the total number of darts thrown (we assume that all darts fall somewhere in the square). The quotient of these two numbers will be an estimate for $\frac{1}{4}\pi$. We now create a loop and generate two standard uniform variates that will represent the x and y coordinates of the position where the dart has fallen, respectively. We then determine whether the darts have fallen in the quarter circle.

The code in C++11 is adapted from the code that used the *Boost Random* library in Demming and Duffy (2010):

```
std::default_random_engine rng;

std::random_device rd;
rng.seed(rd());
std::uniform_real_distribution<double> uni(0.0,1.0);

// Choose the desired accuracy
std::cout << "How many darts to throw? "; long N; std::cin >> N;

// Throw the dart; does it fall in the circle or outside
// Start throwing darts and count where they land
long hits = 0;
double x, y, distance;
for (long n = 1; n <= N; ++n)
```

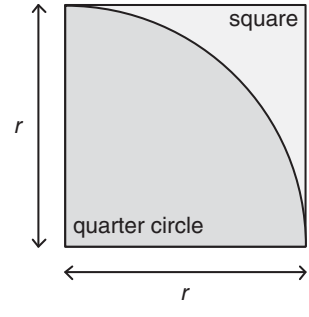


FIGURE 26.1 Calculating π

```
{
    x = uni(rng); y = uni(rng);
    distance = std::sqrt(x*x + y*y);

    if (distance <= 1.0)
    {
        hits++;
    }
}

std::cout << "#hits, PI is: " << hits << ", " << 4.0 * double(hits) /
double (N);
```

26.5 ENGINES IN DETAIL

The C++ random library supports 16 engines to generate random numbers. In general, an engine is a stateful source of randomness. Each engine is a function object in the sense that it implements the *function call operator* `()` to yield a new random value. The internal state is changed to yield a new random value.

The following engine categories and classes (in the `std` namespace) are supported:

- Basic engines that provide algorithms to generate random values:

  `linear_congruential_engine`

  `mersenne_twister_engine`

  `subtract_with_carry_engine`

- Engine adapters are classes that can be initialised by a (basic) engine:

  `discard_block_engine`: adapt an engine by discarding a given number of generated values each time;

  `independent_bits_engine`: adapt an engine to produce random values with a specified number of bits;

  `shuffle_order_engine`: adapt an engine by permutation of the order of their generated values.

- Adapters with predefined parameters – these are user-friendly classes with instantiated template parameters:

  `minstd_rand0`

  `minstd_rand`

  `mt19937`

  `mt19937_64`

  `ranlux24_base`

  `ranlux48_base`

```
ranlux24
ranlux48
knuth_b
```

According to the C++ standard, these engines are implementation independent. They are guaranteed to generate identical value sequences on different platforms.

A discussion of the efficiency and accuracy of the above engines is outside the scope of this chapter. It would seem that the C++ random library has been motivated by the *Boost Random* library.

26.5.1 Seeding an Engine

In order to generate a different random sequence when using a random number engine we have to initialise the engine with a seed that will be the initial value of the linear (or nonlinear) congruential generator that implements it. Thus, once we create an engine we could run the risk of generating the same sequence of values each time we use it. In order to avoid this problem we compute a seed based on a fine-grained system clock or by using `std::random device`, as already discussed. In other words, we wish to avoid replicating the engine's seed from run to run. On the other hand, *replication* might be a requirement, for example when debugging a program or reproducing results. To this end, we take an example to create a number of engines using a combination of default and user-defined seeds. Here we see some of the possibilities, including saving and restoring the state of an engine:

```
// Lines of code to show the ability to seed an engine

// Default ctor and default initial state
std::mt19937_64 eng1;

// Engine with seeds
std::mt19937_64 eng2(1234);

unsigned int mySeed = static_cast<unsigned int>(time(0));
std::mt19937_64 eng3(mySeed);

std::mt19937_64 eng4;
eng4.seed();

// Random device for uniformly integer random variables
std::random_device rd;

std::mt19937_64 eng5(rd());

// Save and restore state of engine
std::stringstream engState;
engState << eng5;           // Save state

eng5.seed(); // Go back to default state

engState >> eng5;           // Restore state
```

```
// Advance to nth next state (similar to n calls to eng())
std::mt19937_64 eng6;
int n = 10;
eng6.discard(n);

// Now produce the next random value and advance state
std::cout << "eng1: " << eng1() << std::endl;
std::cout << "eng2: " << eng2() << std::endl;
std::cout << "eng3: " << eng3() << std::endl;
std::cout << "eng4: " << eng4() << std::endl;
std::cout << "eng5: " << eng5() << std::endl;
std::cout << "eng6: " << eng6() << std::endl;
```

We can also see how to use seeds for random number engines with *replication/no replication* scenarios.

26.5.2 Seeding a Collection of Random Number Engines

In some cases we may wish to seed a large number of random number engines. To this end, the class `std::seed_seq` consumes a sequence of integer-valued data and produces a requested number of unsigned integer values i in the range $0 \leq i \leq 2^{32}$ based on the consumed data. It has a constructor and it can be seeded. The member function `generate()` computes an array of bias-eliminated, evenly distributed 32-bit values.

The first example creates a seeding sequence and then computes a seeding array of length 10:

```
std::seed_seq seq{ 1, 2, 3, 4, 5 };

std::vector <std::uint32_t> seeds(10);
seq.generate(seeds.begin(), seeds.end());

for (auto n : seeds)
{ // The same sequence will always be generated

    std::cout << n << ",";
}
```

The second example uses a seeding sequence based on clock time and produces a different value on each run:

```
std::seed_seq seq2{ static_cast<unsigned int>(time(0)) };
std::vector <std::uint32_t> seeds2(10);
seq2.generate(seeds2.begin(), seeds2.end());

for (auto n : seeds2)
{ //  A different sequence will be generated for each run

    std::cout << n << ",";
}
```

The last example uses the random device to generate the seeds:

```
std::random_device rd;
std::seed_seq seq3{ rd() };
std::vector <std::uint32_t> seeds3(10);
seq3.generate(seeds3.begin(), seeds3.end());

for (auto n : seeds2)
{ //  A different sequence will be generated for each run

    std::cout << n << ",";
}
```

You can run this code and examine the output. The above option could be useful in parallel programming applications in which each thread has its own random number generator. This is really an open research topic. Of course, we must avoid *race conditions* as well as the risk of generating correlated numbers. This topic is outside the scope of this book.

26.6 DISTRIBUTIONS IN C++: THE LIST

We are interested in the five categories of discrete and continuous distributions that are supported in C++, namely:

- Uniform distributions

```
uniform_int_distribution
uniform_real_distribution
```

- Bernoulli distributions

```
bernoulli_distribution
binomial_distribution
geometric_distribution
negative_binomial_distribution
```

- Poisson distributions

```
poisson_distribution
exponential_distribution
extreme_value_distribution
gamma_distribution
weibull_distribution
```

- Normal distributions

```
normal_distribution
cauchy_distribution
chi-squared_distribution
fisher_f_distribution
lognormal_distribution
student_t_distribution
```

- Sampling distributions

```
discrete_distribution
piecewise_constant_distribution
piecewise_linear_distribution
```

A detailed discussion of all these distributions is outside the current scope. However, we have a number of exercises in this chapter that are concerned with some of the above distributions. In general, your particular application will determine which distributions to use.

26.7 BACK TO THE FUTURE: C-STYLE PSEUDO-RANDOM NUMBER GENERATION

In the distant (and not-so-distant) past the quality of random number generators from mainframes to personal computers was low. Anecdotal historical evidence is given in Press et al. (2002, Chapter 7). In this section we give an overview of the C function `rand()` that produces a *pseudo-random number* in the range (0, RAND_MAX), `RAND_MAX` being a constant defined in `<cstdlib>`. It is typically a 16-bit integer and will have a value 32,767. This means that random numbers will be repeated after a finite number of iterations in the (linear) congruential generator that implements `rand()`. More anecdotal evidence suggests that the random numbers are repeated after 20 million iterations. This will be disastrous when we generate random numbers for Monte Carlo simulation. In short, we should avoid this function if at all possible. However, we include a discussion here because there may be legacy systems that use this function and that wish to use the new generators in C++.

As a quick summary, we have:

- `rand()`: generates a pseudo-random number.
- `RAND_MAX`: the maximum possible value generated by `rand()`.
- `srand()`: seeds the pseudo-random number generator.

We take typical examples of use. We first include some library files:

```
#include <cstdlib>        // for rand, srand
#include <ctime>          // time()
#include <iostream>
#include <limits>         // numeric limits
#include <vector>
```

The first example generates a single random number:

```
// Seed, initialise just once
unsigned int mySeed = static_cast<unsigned int>(time(0));
srand(mySeed);

int N = 10;
```

```
// Generate a random number in the range [0,N-1]
int val = rand() % N;
std::cout << "Random number: " << val << std::endl;
```

The second example generates a random number that uses higher-order bits which are more random than the lower-order bits:

```
// Random number between 1 and 10
int val2 = 1 + int(10.0*rand()/(RAND_MAX + 1.0));
std::cout << "Random number: " << val2 << std::endl;
```

Finally, we execute a simulation to compute the frequency of each generated number in the closed range $[0, N-1]$ (in this case $N = 10$). We expect the frequencies to be the same for a good random number generator:

```
// Test: generate many random numbers; each number in [0,N-1] should
// occur with equal frequency.
std::vector<std::size_t> histogram(N,0);

std::size_t M = std::numeric_limits<std::size_t>::max();
for (std::size_t i = 1; i <= M; ++i)
{ // Generate random number in [0,N-1] and place in bucket

    val = rand() % N;
    histogram[val]++;
}

for (std::size_t i = 0; i < histogram.size(); ++i)
{ // Print the histogram

   std::cout << "Bucket number: " << i << ", frequency: "
             << histogram[i] << std::endl;
}
```

Typical output in the case of 100 million runs is:

```
Bucket number: 0, frequency: 9994999
Bucket number: 1, frequency: 9996847
Bucket number: 2, frequency: 9998909
Bucket number: 3, frequency: 9999474
Bucket number: 4, frequency: 10001231
Bucket number: 5, frequency: 10005179
Bucket number: 6, frequency: 10004794
Bucket number: 7, frequency: 9998955
Bucket number: 8, frequency: 10000133
Bucket number: 9, frequency: 9999479
```

This completes our discussion of this random number generator. We do not recommend it, especially in conjunction with concurrent calls to `rand()` or `srand()` in multithreaded

applications because *data races* can occur. These functions access and modify the internal state of objects. This is a general problem when we use random number generators with internal state that can be non-deterministically modified by threads.

26.8 CRYPTOGRAPHIC GENERATORS

In the interest of completeness, we give a brief overview of random generators that are used in *cryptography*. This is the study of techniques to ensure data security and secure communications in the presence of eavesdroppers and adversaries. To this end, we need to generate random numbers for key generation, SSL (*Secure Sockets Layer*) and challenge/response algorithms. We note that pseudo-random number generators are not suitable for cryptographic applications because the requirements for *statistical randomness* differ from those for *cryptographic randomness*. A sequence of numbers can be statistically random but cryptographically insecure if an attacker can predict the sequence of the generated numbers by understanding the algorithm and the random seed used. This means that data will not be secure. We need *cryptographic random numbers*. For a sequence of random numbers to be deemed *cryptographically secure* it must be computationally infeasible to regenerate the same sequence of random numbers. In the case of pseudo-random numbers all we need to know is the pseudo-random number generator and the seed. It is intuitively clear that it takes more time to create a cryptographically secure random number than a pseudo-random number. In the case of the C# language, for instance, it can be up to eight times slower.

26.9 MATRIX DECOMPOSITION METHODS

The generation of correlated and uncorrelated standard normal variates is important in several areas in finance, such as Monte Carlo option pricing and multidimensional diffusions (Kienitz and Wetterau, 2012). We introduce a number of algorithms that are used in these applications. The algorithms are concerned with operations on matrices whose elements are real valued or complex valued. This is a major area of research (Golub and van Loan, 1996) and we focus on a number of methods that are directly applicable in the current book.

We introduce some notation. The *Hermitian transpose* of a square matrix A of size n with complex coefficients (denoted by A^H) is the complex conjugate transpose of A, that is $A^H = \bar{A}^\top$ or $\left(A^H\right)_{ij} = \overline{A}_{ji},\ 1 \leq i, j \leq n$, where in general $\bar{z}$ denotes the complex conjugate of the complex number z, that is $\bar{z} = x - \mathrm{i}y$ where $z = x + \mathrm{i}y$.

An example is:

$$
\begin{aligned}
A &= \begin{pmatrix} 1 & 3\mathrm{i} & 2-\mathrm{i} \\ \mathrm{i} & 1 & -\mathrm{i} \\ 7 & 1+\mathrm{i} & \mathrm{i} \end{pmatrix}, \quad \mathrm{i} = \sqrt{-1} \\
A^H &= \begin{pmatrix} 1 & -\mathrm{i} & 7 \\ -3\mathrm{i} & 1 & 1-\mathrm{i} \\ 2+\mathrm{i} & \mathrm{i} & -\mathrm{i} \end{pmatrix}.
\end{aligned}
\tag{26.15}
$$

In other words, we mirror the corresponding elements on each side of the main diagonal and we then take the complex conjugate of each mirrored element. A special case is when the elements are real valued and we then speak of the *transpose* of a matrix. An example is:

$$\begin{aligned} A &= \begin{pmatrix} 1 & 0.9 & 0.7 \\ 0.5 & 1 & 0.6 \\ 2.0 & -3.0 & 1 \end{pmatrix} \\ A^{\mathrm{T}} &= \begin{pmatrix} 1 & 0.5 & 2.0 \\ 0.9 & 1 & -3.0 \\ 0.7 & 0.6 & 1 \end{pmatrix}. \end{aligned} \tag{26.16}$$

This is called transposing the matrix.

A matrix A is called *Hermitian* if it is equal to its own Hermitian transpose, that is $A = A^H$. We say that a matrix is *normal* if $AA^H = A^H A$. A matrix is *symmetric* if it is its own transpose, that is $A = A^{\top}$.

Let x and y be two complex-valued n-dimensional vectors $x = (x_1, \ldots, x_n)^{\top}$, $y = (y_1, \ldots, y_n)^{\top}$. Then the *Euclidean inner product* is defined as:

$$(x, y) = \sum_{i=1}^{n} x_j \overline{y_j},\ x_j, y_j \in \mathbb{C},\ j = 1, \ldots, n \tag{26.17}$$

where in general $\bar{a}$ is the complex conjugate of $a \in \mathbb{C}$. We now say that a Hermitian matrix A is *positive definite* if:

$$(Ax, x) > 0 \tag{26.18}$$

for all nonzero n-dimensional vectors x, where Ax denotes matrix–vector multiplication of A and x. Furthermore, the matrix A is called *positive semi-definite* if:

$$(Ax, x) \geq 0. \tag{26.19}$$

A *lower-triangular matrix* L is one whose elements above the main diagonal are zero while an *upper-triangular matrix* U is one whose elements below the main diagonal are zero. An example is given by:

$$\begin{aligned} L &= \begin{pmatrix} 1 & 0 & 0 \\ 7 & 2 & 0 \\ 8 & 9 & 4 \end{pmatrix} \\ U &= \begin{pmatrix} 1 & 7 & 8 \\ 0 & 2 & 9 \\ 0 & 0 & 4 \end{pmatrix}. \end{aligned}$$

A matrix U is called *unitary* if its inverse equals its Hermitian transpose, that is:

$$U^{-1} = U^H = \overline{U}^{\top}. \tag{26.20}$$

An *orthogonal matrix* P is a unitary matrix whose elements are all real and that satisfies:

$$P^{-1} = P^{\top}. \tag{26.21}$$

26.9.1 Cholesky (Square-Root) Decomposition

This is a method to factor a positive definite matrix A into the product of a lower-triangular matrix L and its Hermitian transpose L^H:

$$A = LL^H. \tag{26.22}$$

In this section we use the *Boost uBLAS* library that supports vectors and matrices. We mention that this library does not have operations for numerical linear algebra. We already discussed this library in Chapter 10. See also Demming and Duffy (2012).

For symmetric positive definite matrices the Cholesky decomposition algorithm takes on a particularly simple form:

$$A = LL^{\top}.$$

The algorithm to compute the matrix L is given by (see Dahlquist and Björck, 1974):

$$l_{jj} = \left(a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2 \right)^{1/2}, \quad 1 \leq j \leq n$$

$$l_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk} \right) / l_{jj}, \quad i = j+1, \ldots, n.$$

The C++ code that implements this algorithm is:

```
// Cholesky decomposition.
void Cholesky(const ublas::matrix<double>& A,
              ublas::triangular_matrix<double, ublas::lower>& L)
{
    double sum;
    size_t N = A.size1();

    for (size_t j = 0; j < N; ++j)    // Loop over columns.
    {
        sum = 0.0;
        for (size_t k = 0; k < j; ++k)
        {
            sum += L(j,k)*L(j,k);
        }
        L(j,j) = sqrt(A(j,j) - sum);
```

```
            for (size_t i = j+1; i < N; ++i) // Rows.
            {
                sum = 0.0;
                for (size_t k = 0; k < j; ++k)
                {
                    sum += L(i,k)*L(j,k);
                }
                L(i,j) = (A(i,j) - sum) / L(j,j);
            }
        }
    }
```

Finally, an example of how to use the Cholesky algorithm (notice we used a *dense matrix* in Boost uBLAs) is:

```
// Correlation matrix.
unsigned D = 4;
u::matrix<double> Corr(D,D);
Corr(0,0) = Corr(1,1) = Corr(2,2) = Corr(3,3) = 1.0;
Corr(0,1) = Corr(1,0) = 0.5;
Corr(0,2) = Corr(2,0) = 0.2;
Corr(0,3) = Corr(3,0) = 0.01;
Corr(1,2) = Corr(2,1) = 0.01;
Corr(1,3) = Corr(3,1) = 0.30;
Corr(3,2) = Corr(2,3) = 0.30;
cout << "Original matrix:" << endl << Corr << endl << endl;
```

A test case is:

```
// Cholesky decomposition.
cout << endl << "--- Do Cholesky decomposition ---" << endl << endl;
ublas::triangular_matrix<double, ublas::lower> L(D,D);
Cholesky(Corr, L);
cout << "L:" << endl << L << endl << endl;
```

Now we remark on testing the Cholesky method. In particular, we wish to double-check our algorithm while at the same time using some of the functionality of *uBLAS*. We try to *recover* the original matrix that we used in the algorithm by multiplying the constructed lower-triangular matrix by its transpose:

```
// Recover the original matrix; a kind of check.
ublas::triangular_matrix<double, ublas::upper> U = LowerToUpper(L);
cout << "U:" << endl << U << endl << endl;
cout << "L*U (same as orignal matrix):" << endl << L*U << endl << endl;
```

where `LowerToUpper()` is a user-defined free function that converts a lower-triangular matrix to an upper-triangular matrix:

```
// Transpose a lower triangular matrix to form an upper triangular matrix.
ublas::triangular_matrix<double, ublas::upper> LowerToUpper
                (const ublas::triangular_matrix<double, ublas::lower>& L)
```

```
{
    ublas::triangular_matrix<double, ublas::upper> U(L.size2(), L.size1());

    for (size_t i = 0; i < U.size1(); ++i)
    {
        for (size_t j = i; j < U.size2(); ++j)
        {
            U(i,j) = L(j,i);
        }
    }

    return U;
}
```

We conclude this section by showing how to recover the original matrix by multiplying the upper- and lower-triangular matrices:

```
// Create LU matrices.
cout << "--- Recover original matrix from LD & LD ---" << endl << endl;
ublas::triangular_matrix<double, ublas::lower> LD(D,D);
ublas::triangular_matrix<double, ublas::upper> UD(D,D);
InitLU(Corr, LD, UD);       // Documented in section 26.9.2 This factor
                            // decomposes the
                            // matrix Corr into product of LD and UD
cout << "LD: " << endl << LD << endl << endl;
cout << "UD: " << endl << UD << endl << endl;

// Recover original matrix in different ways.
cout << "LD*UD (same as orignal matrix): " << endl << LD*UD << endl;
cout << "UD*LD (same as orignal matrix): " << endl << UD*LD << endl;
```

where we have created user-defined operators for multiplying lower- and upper-triangular matrices:

```
// Compute Lower*Upper.
ublas::matrix<double> operator * (
             const ublas::triangular_matrix<double, ublas::lower>& L,
             const ublas::triangular_matrix<double, ublas::upper>& U)
{
    ublas::matrix<double> result(L.size1(), L.size2());

    double sum; size_t r;

    for (size_t i = 0; i < result.size1(); ++i)
    {
        for (size_t j = 0; j < result.size2(); ++j)
        {
            sum = 0.0;
            r = std::min(i,j);
            for (size_t p = 0; p <= r; ++p)
```

```
            {
                sum += L(i,p)*U(p,j);
            }

            result(i,j) = sum;
        }
    }

    return result;
}

// Compute Upper*Lower.
ublas::matrix<double> operator * (
             const ublas::triangular_matrix<double, ublas::upper>& U,
             const ublas::triangular_matrix<double, ublas::lower>& L)
{
    // Rough and Ready, but it works; it is a test of uBLAS::transpose.
    ublas::triangular_matrix<double, ublas::upper> Upper = trans(L);
    ublas::triangular_matrix<double, ublas::lower> Lower = trans(U);

    return Lower*Upper;
}
```

We can run this code and see that the results are the same in all cases. Finally, another test is to subtract the original matrix from both products and then take the corresponding *max norm*:

```
// Look at matrix norms.
ublas::matrix<double> m1 = Corr - UD*LD;
ublas::matrix<double> m2 = Corr - LD*UD;
cout << "A - U*L max error: " << ublas::norm_inf(m1) << endl;
cout << "A - L*U max error: " << ublas::norm_inf(m2) << endl << endl;
```

These values should be very small.

An example of use is:

```
// Correlation matrix,
// This is the example in MC book page 145 where we use Cholesky
// decomposition.

unsigned D = 4;
u::matrix<double> Corr (D,D);
Corr(0,0) = Corr(1,1) = Corr(2,2) = Corr(3,3) = 1.0;

Corr(0,1) = Corr(1,0) = 0.5;
Corr(0,2) = Corr(2,0) = 0.2;
Corr(0,3) = Corr(3,0) = 0.01;

Corr(1,2) = Corr(2,1) = 0.01;
Corr(1,3) = Corr(3,1) = 0.30;
```

```
Corr(3,2) = Corr(2,3) = 0.30;
std::cout << "Original matrix: " << Corr << std::endl;

u::matrix<double> Backup(Corr);

u::triangular_matrix<double, u::lower> LD(D,D);
u::triangular_matrix<double, u::upper> UD(D,D);

InitLU(Corr, LD, UD);

// Recover original matrix in different ways
Corr = LD * UD;
std::cout << "LD*UD: " << Corr << std::endl;

Corr = UD * LD;
std::cout << "UD*LD: " << Corr << std::endl;

// Look at matrix norms
u::matrix<double> m1 = Backup - UD*LD;
std::cout << "A - U*L max error: " << u::norm_inf(m1) << std::endl;

u::matrix<double> m2 = Backup - LD*UD;
std::cout << "A - L*U max error: " << u::norm_inf(m2) << std::endl;

// Now test Cholesky directly
u::triangular_matrix<double, u::lower> L(D,D);
Cholesky(Backup, L);

u::matrix<double> Product  = L * trans(L);
std::cout << "Cholesky, L*trans(L): " << Product << std::endl;
std::cout << "A - L*trans(L) max error: "
           << u::norm_inf(Backup - Product) << std::endl;

Product  = trans(L)*L;
std::cout << "Cholesky, trans(L)*L: " << Product << std::endl;
std::cout << "A - trans(L)*L max error: "
           << u::norm_inf(Backup - Product) << std::endl;
```

The Cholesky decomposition is also applicable to matrices with complex coefficients; however, we need to introduce a modification to the code (we need to take the complex conjugate of the transposed matrix elements):

```
template <typename T>
    void CholeskyComplex(const u::matrix<std::complex<T>>& A,
                         u::triangular_matrix<std::complex<T>, u::lower>& L)
{ // Cholesky decomposition

    std::complex<T> sum;
    unsigned N = A.size1();
```

```
    for (unsigned j = 0; j < N; ++j)  // Loop over columns
    {

        sum = 0.0;
        for (unsigned k = 0; k < j; ++k)
        {
            // Modification for complex numbers, use conjugate
            sum += L(j, k)*std::conj(L(j, k));
        }
        L(j, j) = std::sqrt(A(j, j) - sum);

        for (unsigned i = j + 1; i < N; ++i) // Rows
        {
            sum = 0.0;
            for (unsigned k = 0; k < j; ++k)
            {
                // Modification for complex numbers
                sum += L(i, k)*std::conj(L(j, k));
            }
            L(i, j) = (A(i, j) - sum) / L(j, j);
        }
    }
}
```

26.9.2 LU Decomposition

This algorithm partitions a general matrix *A* into the product of a lower-triangular matrix *L* and an upper-triangular matrix *U*. The algorithm is documented in Dahlquist and Björck (1974) and in Isaacson and Keller (1966), for example. The C++ code in combination with uBLAS is given as a free function:

```
// LU decomposition: A -> L*U
void InitLU(const ublas::matrix<double>& A,
            ublas::triangular_matrix<double, ublas::lower>& L,
            ublas::triangular_matrix<double, ublas::upper>& U)
{
    double sum;
    unsigned N = A.size1();

    // Common to make all diagonal elements == 1.0
    for (size_t k = 0; k < N; ++k)
    {
        L(k,k) = 1.0;
    }

    for (size_t j = 0; j < N; ++j)        // Loop over columns.
    {
        // U
        for (size_t i= 0; i <= j; ++i)    // Columns.
```

```
        {
            sum = 0.0;
            for (size_t k = 0; k < i; ++k)
            {
                sum += L(i,k)*U(k,j);
            }
            U(i,j) = A(i,j) - sum;
        }
        // L
        for (size_t i = j+1; i < N; ++i)    // Rows.
        {
            sum = 0.0;
            for (size_t k = 0; k < j; ++k)
            {
               sum += L(i,k)*U(k,j);
            }
            L(i,j) = (A(i,j) - sum) / U(j,j);
        }
    }
}
```

Having computed the matrices L and U we can then solve a matrix system in two steps; first we decompose the matrix A and then we solve the lower-triangular and upper-triangular systems in sequence:

$$Ax = b$$
$$(LU)\,x = b \text{ and } Ly = b \text{ and } Ux = y.$$

The code for this sequence of steps is:

```
// Solve Ax = b
ublas::vector<double> SolveLU(const ublas::vector<double>& b,
             const ublas::triangular_matrix<double, ublas::lower>& L,
             const ublas::triangular_matrix<double, ublas::upper>& U)
{
    size_t N = b.size();
    ublas::vector<double> result(N);

    double sum;

    // Forward sweep Ly = b
    result[0] = b[0] / L(0,0);
    for (size_t i = 1; i < N; ++i)
    {
        sum = 0.0;
        for (size_t k = 0; k < i; ++k)
        {
            sum += L(i,k)*result[k];
        }
        result[i] = (b[i] - sum) / L(i,i);
    }
```

```
    // Backward sweep Ux = y
    result[N-1] = result[N-1]/U(N-1, N-1);
    for (size_t i = N-2; i >= 0 && i < N; --i)
    {
        sum = 0.0;
        for (size_t k = i+1; k < N; ++k)
        {
            sum += U(i,k)*result[k];
        }
        result[i] = (result[i] - sum) / U(i,i);
    }

    return result;
}
```

A simple example is:

```
unsigned N = 4;

// Input for Ax = b
ublas::matrix<double> A(N, N);
ublas::vector<double> b(N);
for (size_t i = 0; i < A.size1(); ++i)
{
    b[i] = 1.0;
    for (size_t j = 0; j < A.size2(); ++j)
    {
        A(i, j) = 0.0;
    }
    A(i,i) = 1.0;
}
std::cout << "Initial matrix A: " << A << std::endl;
cout << "Initial vector b: " << b << endl;

// Determine the L and U matrices.
ublas::triangular_matrix<double, ublas::lower> L(N,N); Init(L, 0.0);
ublas::triangular_matrix<double, ublas::upper> U(N,N); Init(U, 0.0);
InitLU(A, L, U);
cout << "Matrix L: " << L << endl;
cout << "Matrix U: " << U << endl;

// Solve 'Ax=b'.
ublas::vector<double> result = SolveLU(b, L, U);
cout << "Result vector Ax=b: " << result << endl;
```

26.9.3 QR Decomposition

This algorithm factors a matrix A into the product of a unitary matrix Q and an upper-triangular matrix R:

$$A = QR. \tag{26.23}$$

The algorithm is based on the *modified Gram–Schmidt method* that we now describe as applied to the linearly independent columns of A. To this end, let $\{X_1, X_2, \ldots, X_n\}$ be a set of linearly independent vectors. The objective is to convert these vectors into a set of orthonormal vectors $\{Q_1, \ldots, Q_n\}$ (an *orthonormal vector* $v = (v_1, \ldots, v_n)^\top$ has length one, that is $\|v\|^2 \equiv (v, v) = \sum_{j=1}^{n} v_j^2 = 1$ (Euclidean norm)) such that each vector Q_k $(k = 1, \ldots, n)$ is a linear combination of $X_1, \ldots, X_{k-1}$.

The algorithm is:

$$\begin{aligned}
&\text{Step 1: Set } r_k = \|X_k\| \text{ and } Q_k = (1/r_k)\, X_k.\\
&\text{Step 1: For } j = k+1, \ldots, n \text{ set } r_{kj} = (X_J, Q_k).\\
&\text{Step 3: For } j = k+1, \ldots, n \text{ replace } X_j \text{ by } X_j - r_{kj} Q_k.
\end{aligned} \tag{26.24}$$

In step 2, we used the vector inner product as defined in equation (26.17).

The C++ code implementing algorithm (26.24) is given by:

```
namespace u = boost::numeric::ublas;

std::tuple<u::matrix<double>, u::triangular_matrix<double, u::upper>>
                    QR(u::matrix<double>& A)
{ // QR decomposition into orthogonal matrix Q and upper triangular
  // matrix R.

    // Components Q and R
    const std::size_t N = A.size1(); const std::size_t M = A.size2();
    u::matrix<double> Q(N, N);
    u::triangular_matrix<double, u::upper> R(N, M);

    // Create the first normalised vector

    for (std::size_t k = 0; k < M; ++k)
    { // Process each column of the input matrix A

        // Define column #k
        u::matrix_column<u::matrix<double>> Ac(A, k);    // Column of A

        // Orthonormal vector component of Q
        u::matrix_column<u::matrix<double>> Qc(Q, k);

        R(k,k) = u::norm_2(Ac); Qc = (1.0/R(k,k))*Ac;

        for (std::size_t j = k+1; j < M; ++j)
        {
           u::matrix_column<u::matrix<double>> Xc(A, j);
           R(k,j) = u::inner_prod(Xc, Qc);
           Xc -= R(k,j) * Qc;
        }
    }
```

```
    return std::tuple<u::matrix<double>,
          u::triangular_matrix<double, u::upper>>   (Q, R);
}
```

We remark that this code uses the useful function `u::matrix_column` in *Boost uBLAS* to access an individual column of a matrix. Finally, it is possible to check if the algorithm has worked by using the properties of the matrices in equation (26.24):

```
void QRPostProcessing(const u::matrix<double>& A,
                      const u::matrix<double>& Q,
                      const u::triangular_matrix<double, u::upper>& R)
{ // Testing the accuracy of the QR decomposition

    cout << "\n QR Postprocessing results\n";

    std::size_t N = A.size1();

    // Special constant matrices
    u::identity_matrix<double> I(N,N);

    // 1. A - QR
    std::cout << "\nDifferences of A and Q*R: "
              << u::norm_inf(A - u::prod(Q,R)) << std::endl;

    // 2. Q^T*Q - I
    std::cout << "Differences of trans(Q)*Q and identity I: "
              << u::norm_inf(u::prod(u::trans(Q),Q) - I) << std::endl;

    // 3. Q^T*A - R
    std::cout << "Differences of trans(Q)*A and R: "
              << u::norm_inf(u::prod(u::trans(Q),A) - R) << std::endl;
}
```

We take an example:

```
const int N = 3;
u::matrix<double> A(N, N);

A(0, 0) = -4.0; A(0, 1) = 2.0; A(0, 2) = 2.0;
A(1, 0) = 3.0; A(1, 1) = -3.0; A(1, 2) = 3.0;
A(2, 0) = 6.0; A(2, 1) = 6.0; A(2, 2) = 0.0;
u::matrix<double> ACpy(A);

auto qr = QR(A);
auto Q = std::get<0>(qr); auto R = std::get<1>(qr);
std::cout << std::get<0>(qr) << '\n';
std::cout << std::get<1>(qr) << '\n';

QRPostProcessing(ACpy, Q, R);
```

The output is:

```
[3,3]((-0.512148,0.494524,0.702247),
    (0.384111,-0.599423,0.702247),
    (0.768221,0.629394,0.117041))

[3,3]((7.81025,2.4327,0.128037),
    (0,6.56369,-0.809221),
    (0,0,3.51123))

QR Postprocessing results

Differences of A and Q*R: 2.22045e-16
Differences of trans(Q)*Q and identity I: 1.38778e-16
Differences of trans(Q)*A and R: 1.9984e-15
```

For larger matrices the console output is difficult to read and we can then use the Excel driver that we introduced in Chapter 14:

```
// Start Excel
using NumericMatrix = boost::numeric::ublas::matrix<double>;
ExcelDriver& excel = ExcelDriver::Instance();
excel.MakeVisible(true);           // Default is INVISIBLE!

std::string sheetName("Test Case Q Matrix");
long row = 4; long col = 2;
excel.AddMatrix<NumericMatrix>(Q, sheetName, row, col);

std::string sheetName2("Test Case R Matrix");
excel.AddMatrix<NumericMatrix>(R, sheetName2, row, col);
```

This option is useful not only for visualisation purposes, but it also allows us to export matrix and vector data to Excel for further processing in VBA or C#, for example. It is useful to run the code and visualise the output.

26.10 GENERATING RANDOM NUMBERS

A *covariance matrix* (also known as a *dispersion matrix* or *variance–covariance matrix*) is a matrix whose element in the (i, j) position is the covariance between the ith and jth elements of a random vector.

A *multivariate normal distribution* $N(\mu, \Sigma)$ is specified by its mean vector μ and covariance matrix Σ. The covariance matrix may be specified implicitly through its diagonal entries σ_i^2 and correlations ρ_{ij} using $\rho_{ij} = \frac{\Sigma_{ij}}{\sigma_i \sigma_j}$ in matrix form:

$$\sum = \begin{pmatrix} \sigma_1 & & & \\ & \sigma_2 & & 0 \\ & & \ddots & \\ 0 & & & \sigma_d \end{pmatrix} \begin{pmatrix} \rho_{11} & \rho_{12} & \cdots & \rho_{1d} \\ \rho_{12} & \rho_{22} & & \rho_{2d} \\ \vdots & & \ddots & \vdots \\ \rho_{1d} & \rho_{2d} & \cdots & \rho_{dd} \end{pmatrix} \begin{pmatrix} \sigma_1 & & & \\ & \sigma_2 & & 0 \\ & & \ddots & \\ 0 & & & \sigma_d \end{pmatrix}.$$

In the two-dimensional case the covariance matrix is:

$$\sum = \begin{pmatrix} \sigma_1^2 & \sigma_1\sigma_2\rho \\ \sigma_1\sigma_2\rho & \sigma_2^2 \end{pmatrix} \tag{26.25}$$

where:

$$\sigma_1, \sigma_2 > 0 \text{ and } -1 \leq \rho \leq 1.$$

In this case the Cholesky decomposition becomes:

$$\sum = LL^{\top} \tag{26.26}$$

where:

$$L = \begin{pmatrix} \sigma_1 & 0 \\ \rho\sigma_2 & \sqrt{1-\rho^2\sigma_2} \end{pmatrix}.$$

Thus, in order to sample from a *bivariate normal distribution* $N(\mu, \Sigma)$ with $\left(\mu = \mu_1,\ \mu_2\right)^{\top}$ we use the formula:

$$\begin{aligned} X_1 &= \mu_1 + \sigma_1 Z_1 \\ X_2 &= \mu_2 + \sigma_2 Z_1 + \sigma_2\sqrt{1-\rho^2} Z_2. \end{aligned} \tag{26.27}$$

Finally, the algorithm to generate arrays of correlated random numbers is:

a) Generate an uncorrelated $N(0, 1)$ vector Z.
b) Calculate the Cholesky lower-triangular matrix C of the matrix Σ.
c) Compute the new correlated random vector $X = CZ + m$ where m is the mean of X.

See also Exercise 8.

26.10.1 Appendix: Overview of the *Eigen* Matrix Library

Eigen is a high-level C++ library of template headers for linear algebra, matrix and vector operations, geometrical transformations, numerical solvers and related algorithms. *Eigen* is an open-source library and it is implemented using the *expression templates metaprogramming* techniques; it builds expression trees at compile-time and generates custom code to evaluate them. Using expression templates and a cost model of floating-point operations, the library performs its own *loop unrolling* and *vectorisation*.

A summary of the main features in *Eigen* is:

- Matrix and array classes and basic linear algebra; array manipulation.
- LU decomposition.
- LLT and LDLT Cholesky decomposition.

- Singular value decomposition (SVD) and least-squares solver.
- QR decomposition (Householder).
- Eigenvalue and eigenvector decompositions.
- Sparse matrix storage and related basic linear algebra.

One way to learn *Eigen* and apply it to applications in computational finance is to copy exemplar code and then modify it to suit your current application. If you know numerical linear algebra and possibly another C++ matrix library then learning *Eigen* should be easy.

As a final example, we discuss how to solve an *overdetermined linear system*: given an $m \times n$ matrix A where $m \geq n$ and an $m \times 1$ vector b, find a vector x such that Ax is the 'best' approximation to b.

The system $Ax = b$ is overdetermined so it cannot be solved exactly. But it does have a *least-squares solution*:

$$A^{\top}(b - Ax) = 0 \text{ or } \left(A^{\top}A\right)x = A^{\top}b \tag{26.28}$$

satisfying:

$$\|b - Ax\| \leq \|k - Ay\| \, \forall y \in \mathbb{R}^n.$$

System (26.28) is called the normal equations (Dahlquist and Björck, 1974, p. 197).

The code is as follows:

```
// Create overdetermined system
// From Dahlquist's bookpage 199: answer is (5/4, 7/4, 3)
int M = 6; int N = 3;
Eigen::MatrixXd A(M, N);
A(0, 0) = 1.0; A(0, 1) = 0.0; A(0, 2) = 0.0;
A(1, 0) = 0.0; A(1, 1) = 1.0; A(1, 2) = 0.0;
A(2, 0) = 0.0; A(2, 1) = 0.0; A(2, 2) = 1.0;
A(3, 0) = -1.0; A(3, 1) = 1.0; A(3, 2) = 0.0;
A(4, 0) = 0.0; A(4, 1) = -1.0; A(4, 2) = 1.0;
A(5, 0) = -1.0; A(5, 1) = 0.0; A(5, 2) = 1.0;

Eigen::VectorXd b(M);
b[0] = 1.0; b[1] = 2.0; b[2] = 3.0; b[3] = 1.0; b[4] = 2.0; b[5] = 1.0;

// Set up and solve overdetermined system
auto AT = A.transpose();
auto ATb = AT*b;
auto ATA = AT*A;
Eigen::VectorXd x = ATA.ldlt().solve(ATb);
std::cout << "Solution(5/4, 7/4, 3): " <<x[0]<<","<<x[1]<< "," << x[2];

// Compute the residual
auto r = AT*A*x - AT*b;
std::cout << "Residual size: " << r.norm() << std::endl;
```

Finally, we apply the Cholesky method to the example in Kienitz and Wetterau (2012, p. 138; we get the same output):

```
unsigned D = 4;
Eigen::MatrixXd corr(D, D);

corr(0, 0) = corr(1, 1) = corr(2, 2) = corr(3, 3) = 1.0;

corr(0, 1) = corr(1, 0) = 0.748733508;
corr(0, 2) = corr(2, 0) = 0.888524896;
corr(0, 3) = corr(3, 0) = 0.821526502;

corr(1, 2) = corr(2, 1) = 0.726662801;
corr(1, 3) = corr(3, 1) = 0.683150423;

corr(3, 2) = corr(2, 3) = 0.878266274;
std::cout << "\nThe correlated matrix is" << std::endl << corr;

// Compute the Cholesky decomposition of matrix corr
Eigen::LLT<Eigen::MatrixXd> lltOfA(corr);
// Retrieve factor L  in the decomposition
Eigen::MatrixXd L = lltOfA.matrixL();
// Previous two lines can also be written as "L = A.llt().matrixL()"

std::cout << "\nThe Cholesky factor L is" << std::endl << L;
std::cout << "\nTo check this, let us compute L * L.transpose()" ;
std::cout << L * L.transpose() << std::endl;
std::cout << "\nThis should equal the matrix corr matrix" << std::endl;

// Sanity check
auto Diff = corr - L * L.transpose();
std::cout << "\nThis should equal zero? " << Diff << std::endl;
```

26.11 SUMMARY AND CONCLUSIONS

We have given an overview of random number generation and statistical distributions in C++. We first gave a short introduction to the mathematical foundations of random number generation and some of the associated use cases, such as generating uniform random variates and variates from a given *univariate* distribution. There are several basic engines, engine adapters and adapters with predefined parameters and furthermore 20 distributions classified into five major categories.

It will be interesting to see the uptake of this library in the future. One advantage is that we no longer need to rely on proprietary libraries. On the other hand, the functionality in standard C++ may not be suitable for all applications in computational finance and in these cases we may need to resort to more specialised libraries.

Chapters 31 and 32 apply the results of this chapter to computational finance.

26.12 EXERCISES AND PROJECTS

1. (Legacy Code Migration)

Some legacy systems may still use the *polar Marsaglia* (equation (26.2)) or *Box–Muller* (equation (26.3)) methods based on `std::rand()`, for example. The objective of this exercise is to write functions for these methods using the engines in C++11. The return type is a tuple of values, each field being a standard normal random variate. Test your code. Examine the possible performance issues associated with the creation and unpacking of tuples, especially if performed in a loop. Consider the interface that returns a single value.

2. (Random Variates)

Generate a random variable with pdf:

$$f_X(x) = \begin{cases} 2x, & 0 \le x \le 1 \\ 0, & \text{otherwise.} \end{cases}$$

Show that the cdf is given by:

$$f_X(x) = \begin{cases} 0, & x < 0 \\ x^2, & 0 \le x \le 1 \\ 1, & x > 1 \end{cases}$$

and that:

$$X = F_X^{-1}(U) = \sqrt{U}, \quad 0 \le U \le 1.$$

In other words, in order to generate a variate from this distribution we generate a uniform variate U on (0, 1) (for example, using a C++ engine) and then take the square root of U.

3. (Distributions Not Supported in C++11)

In some cases we may wish to generate random variates from distributions that are not supported in C++11. What do we do in such cases? There are basically two choices; first, an exact formula may be known or failing that, we resort to numerical methods. In the first case we take the example of the *Rayleigh distribution* which has many applications, for example when we condition a standard Brownian motion starting at the origin to have the value b at time 1 (Glasserman, 2004, p. 56). Then the maximum over the closed interval [0, 1] is given by:

$$F(x) = 1 - e^{2x(x-b)}, \quad x \ge b.$$

If we solve the equation $F(X) = U$ with $U \in (0, 1)$ we get the roots:

$$X = \frac{b}{2} \pm \frac{\sqrt{b^2 - 2\log(1-U)}}{2}.$$

We take the larger of the roots since the maximum of the Brownian path must be at least b. Since U and $1-U$ have the same distribution, we arrive at the value:

$$X = \frac{b}{2} + \frac{\sqrt{b^2 - 2\log(U)}}{2}.$$

Implement this formula as a C/C++ function using one of the engines in C++11. Finally, implement the same functionality using an iterative solver using the inverse transform method (26.6) in combination with the *Boost C++ Math Toolkit* (see also Chapter 19) that contains a class for the Rayleigh distribution.

4. (Acceptance–Rejection Method)

The efficiency of the algorithm in Section 26.3.2 can be improved because the *majorising function g* is not very accurate. We propose a better function to reduce the rejection rate. To this end, let us consider the majorising function:

$$g(x) = 2x, \quad 0 \leq x \leq 1.$$

We can check that the cdf is given by $F(x) = x^2$ and hence the inverse transform equation (26.6) tells us that $X = F_X^{-1}(U) = \sqrt{U}$. Show that the test for acceptance becomes $U_2 \leq \frac{3}{2}\sqrt{U_1}$ and we then accept U_1 as the random variable from *f*.

Test the efficiency of this procedure compared with the solution given in Section 26.3.2.

5. (Weibull Distribution)

Consider the *Weibull distribution* with *shape parameter* $k > 0$ and *scale parameter* $\lambda > 0$. Show that the cdf is given by:

$$F(x) = \begin{cases} 1 - e^{-\left(\frac{x}{\lambda}\right)^k} & x \geq 0 \\ 0, & x < 0. \end{cases}$$

Show that a random variate of this distribution is given by:

$$X = (\lambda - \log U)^{1/k}$$

where U is a uniform variate on the inverval (0, 1). We note that C++ supports the Weibull distribution.

6. (Multinormal Distribution)

A random vector $X = \left(X_1, \ldots, X_n\right)$ has a *multinormal distribution* if its pdf is given by:

$$f_X(x) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left[-\frac{1}{2}(x-\mu)^\top \Sigma^{-1}(x-\mu)\right]$$

where $|\Sigma|$ is the determinant of the $n \times n$ *covariance matrix* defined by:

$$\sum = \begin{pmatrix} \sigma_{11} & \sigma_{12} & \cdots & \sigma_{1n} \\ \sigma_{21} & \sigma_{22} & \cdots & \sigma_{2n} \\ \vdots & \vdots & & \vdots \\ \sigma_{n1} & \sigma_{n2} & \cdots & \sigma_{nn} \end{pmatrix}.$$

Here Σ^{-1} is the inverse of Σ. Furthermore, x and μ are n-dimensional vectors.

A special case is the standard n-dimensional normal $N(0, I_n)$ having the form $\frac{1}{(2\pi)^{n/2}} \exp\left(-\frac{1}{2}x^\top x\right)$ where $x^\top$ is the transpose of the vector x. Here I_n is the $n \times n$ identity matrix.

We note that the symmetric matrix Σ is positive definite, that is $x^\top \Sigma x > 0 \; \forall x \in \mathbb{R}^n$ and we can express it in Cholesky decomposition form, namely $\Sigma = CC^\top$ where C is a lower-triangular matrix.

The objective of this exercise is to generate a correlated vector X. We describe the steps to execute the algorithm.

Answer the following questions:

a) Generate an uncorrelated $N(0, 1)$ vector Z.

b) Calculate the Cholesky lower-triangular matrix C.

c) Compute the new correlated random vector $X = CZ + m$ where m is the mean of X.

Implement a solution in C++11 in combination with a matrix library, such as *Eigen*, *Boost uBLAS* or your own favourite library.

7. (Laplace Distribution)

Even though C++ does not support all univariate distributions that does not mean all is lost! In some cases we may be able to generate random numbers from a given distribution if we have an analytic form for its cdf, for example. More specifically, we may be able to invert the cdf and then we are done or failing that we can use the Newton–Raphson method or the fixed-point method to iteratively compute the solution. In this exercise we concentrate on the analytic approach by examining the *Laplace* and *Extreme Value* distributions, the latter being supported in C++11.

The *Laplace distribution* (also called the *double exponential distribution*) can be seen as the difference between two independent identically distributed exponential random variables. It depends on a *location parameter* μ and a *scale (diversity) parameter* $b \geq 0$. In the case $\mu = 0$ and $b = 1$ we get an exponential distribution on the positive half-line scaled by a factor $1/2$. The pdf and cdf for the Laplace distribution are given respectively by:

$$f(x) = \frac{1}{2b} \exp\left(-\frac{|x-u|}{b}\right)$$

$$F(x) = \frac{1}{2} + \frac{1}{2}\text{sgn}(x-\mu)\left(1 - \exp\left(-\frac{|x-\mu|}{b}\right)\right).$$

Prove that a random variate of this distribution is given by:

$$X = \mu - b\text{sgn}(U)\log(1 - 2|U|)$$

where U is drawn from a uniform distribution in the interval $(-1/2, 1/2)$.

The second example is the *Extreme Value distribution* whose pdf and cdf are given respectively by:

$$f(x) = \sigma^{-1} \exp\left(-\frac{x-u}{\sigma}\right) \exp\left(-\exp\left(-\frac{x-\mu}{\sigma}\right)\right), \; x \in \mathbb{R}$$
$$F(x) = \exp\left(-\exp\left(-\frac{x-\mu}{\sigma}\right)\right), \; x \in \mathbb{R}.$$

Show that:

$$X = -\sigma \log(-\log(U)) + \mu$$

where U is drawn from a uniform distribution in the interval $(0, 1)$. In the case of the extreme value distribution compare your results with those from C++11.

Write functions in C++ to compute random variates from these two distributions.

8. (A Reusable Framework for Random Number Generation)

The goal of this exercise is to create a software framework containing functionality that is needed for random number generation in computational finance. We focus on producing arrays of correlated random numbers. The main requirements are efficiency and accuracy in the short term and then extendibility in a second increment of the framework. You should use the generic class `RNGenerator` from Section 15.5.2 as it contains functionality that you can use in the framework.

Answer the following questions:

a) Extend the functionality of `RNGenerator` for the creation of matrices with random values.

b) Create free functions to generate two correlated standard normal variates having a given correlation.

c) Create two separate functions to generate an array of correlated random numbers using Cholesky decomposition; one function uses the code presented in Section 26.9.1 while the second function uses the *Eigen* library. Test these functions on examples to compare efficiency and accuracy. Does *Eigen* support Cholesky decomposition on matrices with complex coefficients?

d) Stress test: create a loop and in the loop create a random covariance matrix. Apply the functions from part c). Compute the maximum error between the two solutions.

e) Modify the code so that it can switch between different implementations of the Cholesky algorithm (we are typically thinking of a *Strategy* design pattern (GOF, 1995) or a *mixin*). Implement this code using the CRTP with `std::function` as embedded algorithm.

f) Can you discover new requirements and features that you would like to add to the software framework?

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 27

Microsoft .Net, C# and C++11 Interoperability

27.1 INTRODUCTION AND OBJECTIVES

In this chapter we introduce the C++/CLI (*Common Language Infrastructure*) language. It is a .NET programming language similar to C# and VB.NET in terms of efficiency and functionality but with the added advantage that it is possible to write applications containing a mixture of C++/CLI, C# and *native C++ code*.

Before we discuss the C++/CLI language we give an overview of the programming knowledge and skills that are needed in order to become comfortable with the language. In particular, it is especially important to understand the similarities and differences between native C++ and C++/CLI (and to a lesser extent between C# and C++/CLI). Ideally, a good working knowledge of C++ is a prerequisite in order to understand C++/CLI, including pointers, memory management, classes and templates. C++/CLI code can be mixed with native C++ code. It is important to distinguish between the two languages and this is achieved by having a good knowledge of each one. Furthermore, C++/CLI is relatively easy to learn for developers who already program in C#. Since C# and C++/CLI are both .NET languages they share a common syntax. Programs that have been written in C# are easily understood by developers who know C++/CLI and it is not too difficult for C# developers to fathom C++/CLI.

We summarise the goals and content in this chapter:

- A global overview of the .NET language C++/CLI.
- Memory management in .NET.
- The various ways to create multi-language (C#, C++ and C++/CLI) applications.
- Using C++ libraries (for example, *Boost* and *Quantlib*) in .NET applications.
- Next-generation *Observer* design pattern (using .NET *Events*) and using it in native C++.
- Integration with legacy C, native C++ and COM code.
- .NET *assemblies* and their integration with our software framework as introduced in Chapter 9.

We are unable to discuss everything that we would like to discuss due to lack of space. However, we are working on a follow-up text in which we expand on these interoperability topics.

27.2 THE BIG PICTURE

C++/CLI differs from C++ in a number of ways. Some of the differences are related to the fact that C++/CLI is a .NET language and it enjoys many features that are not supported in C++:

a) It is a dynamic language and it has support for *introspection* (*Reflection*), dynamic loading of assemblies (dlls) and dynamic object creation.
b) Access to .NET libraries for databases, serialisation, networking, graphical user interfaces and web services, for example.
c) Support for multithreading and parallel programming.
d) Automatic memory management and garbage collection support.
e) Support for interfaces.
f) Support for delegates and events.
g) Support for properties (standardised getters and setters for member data).

Part of the challenge when learning C++/CLI is to know which features in native C++ do and do not work in that language and how the two languages cooperate together. They have much in common but there are differences.

C++/CLI is a set of extensions to C++ to benefit from the services that an implementation of the CLI offers. It is a .NET language and you will find it easy to learn if you know (*native*) C++ and some C#. In particular, you can freely mix C++/CLI and C++ code in a project and in this sense we say that C++/CLI is *bilingual* because it can talk to both .NET code and native C++ code. A summary of the language from Wikipedia reads:

> *C++/CLI (C++ modified for CLI) is a language specification created by Microsoft and intended to supersede Managed Extensions for C++. It is a complete revision that aims to simplify the older Managed C++ syntax, which is now deprecated. C++/CLI was standardised by Ecma as ECMA-372. It is currently available in Visual Studio 2005, 2008, 2010, 2012, 2013, 2015 and 2017, including the Express editions.*

As already mentioned, the syntax of C++/CLI will be easy to learn if you already know C#. An added advantage is knowledge of (native) C++. There are a number of differences between C++ and C++/CLI, however, one of which is the memory model in .NET. Objects are of course instances of classes and they are *always* created on the heap in .NET (unless we define *value types*). This is different from C++ where objects can also be created on the stack. Objects are automatically garbage collected in C++/CLI. In contrast, value types' values live on the stack and are not garbage collected. It is important to distinguish between *value types* and *reference types* in .NET as these can be a source of confusion. By default, all classes are derived from `System::Object`, which is the lowest common denominator of almost all .NET types. This base class can refer to any kind of object or value and its data lives on the heap. We need to discuss how to convert values to objects (*boxing*) and convert objects to values (*unboxing*). A simple example is:

```
int i = 42;
Object^ obj = (Object^)i;                              // boxing
Console::WriteLine("Values: {0}", obj->ToString());   // 42
```

```
// Unbox using a safe cast
int j = safe_cast<int>(obj);                          // 42
long k = safe_cast<long>(obj) + 10;                   // 52

Console::WriteLine("Values: {0},{1}", j, k);
```

We now jump into C++/CLI code to create a class hierarchy, create instances of derived classes and perform casting in the hierarchy. The syntax is documented in the code and the main highlights are:

- The mechanics of inheritance (`public` inheritance is default).
- Static constructor, default constructor, copy constructor and other constructors.
- Static member data.
- Finalisers (look like a C++ destructor; not needed for managed classes).
- *Property* syntax to set and get member data in a standard way.
- The use of abstract classes and abstract (no body) methods.
- Virtual (polymorphic) methods and overriding them.

This list feels like a *mini-course* but readers should have little difficulty understanding the code. As an example, we define the base class:

```
#pragma once                        // Prevent multiple inclusion

public ref class Shape abstract     // ref => on heap
{
public:
   Shape() { }                      // Default constructor

   Shape(Shape^ source) { }         // Copy constructor

   // Finaliser
   !Shape() { }

   // Return descriptive string
   virtual String^ ToString() override
   {
      return "Shape";
   }

   // The draw function is abstract because
   // Shape is not concrete. It can't be drawn.
   virtual void Draw() abstract;
};
```

The derived class `Point` is:

```
#pragma once   // Prevent multiple inclusion

#include "Shape.hpp"
```

```
public ref class Point: Shape                  // Point class derived from
                                               // Shape
{
private:
   double x;                                   // Space for x-coordinate
   double y;                                   // Space for y-coordinate

   static int numPoints = 0;                   // Number of points created
   static Point^ origin=gcnew Point(0.0, 0.0); // Origin point
   static int test;
public:

   /* Constructors */

   // Static constructor
   static Point()
   { // Ctor called prior to when the class is first initialised

      test = 42;
   }

   // Default constructor
   Point(): Shape(), x(0.0), y(0.0)
   {
      numPoints++;
   }

   // Copy constructor
   Point(Point^ source): Shape(source), x(source->x), y(source->y)
   {
      numPoints++;
   }

   // Constructor with coordinates
   Point(double x, double y): Shape(), x(x), y(y)
   {
      numPoints++;
   }

   // Finaliser invoked just before object is garbage collected
   !Point()
   {
      numPoints--;   // Decrease counter
   }

   // Return the number of created points
   // Note, in static members you can't use 'this'
   static int GetPoints()
   {
      return numPoints;
   }
```

```
    /* Properties */

    // Access the x-coordinate
    property double X
    {
       double get()
       {
          return x;
       }

       void set(double x)
       {
          this->x=x;
       }
    }

    // Access the y-coordinate
    property double Y
    {
       double get()
       {
          return y;
       }

       void set(double y)
       {
          this->y=y;
       }
    }

    // Read only property to access origin point
    static property Point^ Origin
    {
       Point^ get()
       {
          return origin;
       }
    }

    // Return descriptive string
    virtual String^ ToString() override
    {
       return Shape::ToString()
             + String::Format(": Point({0}, {1})", x, y);
    }

    // Draw point, emulated with printing text
    virtual void Draw() override
    {
       System::Console::WriteLine("Draw Point({0}, {1})", x, y);
    }
};
```

A test program is:

```
using namespace System;

#include "Point.hpp"

int main()
{
   Point^ p=gcnew Point(1.0, 3.0);     // Create Point
   // Shape^ s=gcnew Shape();          // can't be created
   Shape^ sp=gcnew Point(4.5, 3.1);

   p->Draw();                          // Prints 'Draw Point(1.0, 3.0)'
   sp->Draw();                         // Prints 'Draw Point(4.5, 3.1)'

   // Casting
   Shape^ sp2 = gcnew Point(1.0, 2.0);
   try
   {
      Point^ pt = safe_cast<Point^>(sp2);
      Console::WriteLine(L"Coordinates:{0},{1} ", pt->X, pt-> Y);
   }
   catch (System::InvalidCastException^ e)
   {
      Console::WriteLine("Cast failed");
   }

   return 0;
}
```

This code can be studied, run and generalised to other problems.

27.3 TYPES

The *Common Type System* (CTS) is a key component in the *Common Language Runtime* (CLR). It allows developers to create multi-language applications. In particular, it has the following functions:

a) An object-oriented data model to support the data types of all .NET programming languages.
b) A set of constraints that the data types of a .NET programming language must satisfy in order to interact with other .NET languages.
c) A framework to allow .NET languages to interoperate and to ensure data type safety.

The two main data types are the value type and the reference type. *Value types* (such as `int`, `float` and `double`) as well as user-defined value types are stored as the reference type. *Reference types* (such as arrays, classes and handles) are stored on the managed heap as references to the location of the data type.

The CTS types in C++/CLI are similar to the types in C#, with some additions:

- Built-in value types that represent integers, reals, Booleans and characters.
- Single and multidimensional arrays.
- Reference class types: these are user-defined types.
- Delegates: a *delegate* is a type that holds a reference to a method.
- Interface types: similar to abstract classes except that an interface can only contain public, pure virtual methods and it may not contain data.
- Handler type: this is a reference to a type that is similar to a native C++ pointer except that handles to data cannot be manipulated, in other words it is not possible to add or subtract offsets to handles, in contrast to native C++ pointers.
- User-defined value types: user-defined extensions to the standard (primitive) value types.
- Enumeration: a list of named integer constants.
- Boxed value type: a temporary reference to a value type that allows it to be placed on the heap.

27.4 MEMORY MANAGEMENT

Developers will be familiar with memory management. For example, the .NET Framework has a *garbage collector* that is responsible for the removal of objects from the managed heap. Thus, the developer does not have to worry about removing objects from the CLI (managed) heap when they are no longer being used. Native C++ uses an unmanaged run-time (CRT) heap and in this case it is the responsibility of the programmer to explicitly remove objects from heap memory. In other words, native C++ uses (unsafe) pointers while C++/CLI supports both unsafe pointers and handles. A *handle* is a reference to a type.

.NET garbage collection is automatically taken care of by the CLR. In some cases the frequency of garbage collection may need to be modified and to this end we can use the class `System::GC` to trigger the garbage collection process. Since the garbage collector only works with managed memory and has no knowledge of unmanaged memory, it underestimates the urgency of scheduling garbage collection, as can be seen with a managed object that allocates a large amount of unmanaged memory, for example. We thus need a mechanism to inform the garbage collector after having allocated unmanaged memory and after having released this unmanaged memory. To this end, the class `System::GC` has the following static methods:

- `GC::AddMemoryPressure()`: informs the run-time that a large allocation of unmanaged memory should be taken into account when scheduling garbage collection.
- `GC::RemoveMemoryPressure()`: informs the runtime that a large allocation of unmanaged memory has been released and no longer needs to be taken into account when garbage collection takes place.

An example of use is:

```
// Allocate some unmanaged memory
long long N = 1e8;
double* darr = new double[N];
```

```
// Inform runtime that unmanaged memory has been allocated
GC::AddMemoryPressure(N);

// Release unmanaged memory and inform runtime
delete [] darr;
GC::RemoveMemoryPressure(N);
```

This technique should be useful when a mixture of C++/CLI and native C++ code is being used to manage large data structures such as arrays, vectors and matrices, for example.

Some facts relating to garbage collection are useful to summarise at this stage because of the subtle issues involved when working with the CLI (managed) and CRT (C/C++ Runtime) (unmanaged) heaps:

1. The garbage collector tracks and reclaims objects that have been allocated in managed memory. It does not recognise references to objects from unmanaged code.
2. The garbage collector only frees an object if it concludes that there are no references to that object.
3. The garbage collector does not maintain information about resources (for example, file handles or database connections) held by an object. This issue is resolved by implementing a *finaliser* in a type that uses unmanaged resources. These unmanaged resources must be released before instances of the type are reclaimed.
4. The garbage collector supports *object aging* using generations. A *generation* is a unit of measure of the relative age of objects in memory. It is possible to force garbage collection of objects in a range of generations.

The goal of this section was to give an impression of how garbage collection works in .NET and the issues that arise when developing code in a mixed C++ environment.

There are two kinds of *destructor* in C++/CLI. The first kind is concerned with the deallocation of previously allocated memory while the second type is concerned with the deallocation of managed or unmanaged resources.

An object that is allocated on the CLI heap using `gcnew` needs to be deallocated. The choices are:

- Call the *delete operator* on the handle of a `ref` class object. Then the managed memory will be immediately deallocated in the reverse order to which it was allocated. In this case the programmer has full control of when objects are finally cleaned up.
- In some cases the programmer does not care when memory is reclaimed. Then it is not necessary to call the *delete* operator. The CLR will detect that an object is no longer needed and it will garbage collect the memory in due time. Thus, in most cases the use of the *delete* operator is not needed.

When we call the *delete* operator on an object handle the corresponding destructor function (if it exists) is called. In the body of the destructor we normally call the *delete* operator for any object that the `ref` class needs to clean up. Finally, if a `ref` class does not allocate memory then there is no need to create a destructor and in these cases a default destructor will be called. We shall see some examples of use when we create .NET wrappers for native C++ objects.

27.5 AN INTRODUCTION TO NATIVE CLASSES

A *native class* (also known as an *unmanaged class* or *unsafe class*) is a standard compiler-independent C++ class. Its instances are placed on the CRT heap and not on the CLI heap. This implies that unmanaged memory is not maintained by the .NET garbage collector. Thus, it is the responsibility of the programmer to manage memory in code (something that C++ developers know). Some points to remember when creating native classes are:

- When no explicit base class is specified then the class is an independent root, in contrast to managed classes that are derived from `System::Object` by default (if no explicit base class has been specified).
- Native classes support multiple inheritance; managed classes do not support multiple inheritance but this is compensated for by the fact that C++/CLI supports *interfaces*.
- Native classes support friends whereas managed classes do not.
- Native classes can only inherit from native classes and managed classes can only inherit from managed classes.
- A native class can contain data members of type pointer to native classes but it cannot contain a handle to a managed class. In other words, we cannot embed a C++/CLI class in a native C++ class.
- A managed class can contain data members of type pointer to native classes as well as a handle to a managed class.

We thus see that managed classes may be composed of native classes (and hence *delegate* to them) but a native class cannot delegate to a managed class. We shall see how to use managed classes as wrappers to extend the life of native C++ (legacy) code in .NET. We also see that it is not possible to call .NET code from native C++, at least not directly. However, it is possible to use .NET objects in COM (*Component Object Model*) by creating a special CCW (*COM-Callable Wrapper*) object that bridges the COM and .NET worlds.

27.6 INTERFACES AND ABSTRACT CLASSES

An *interface* is a collection of abstract methods and properties placed in a single cohesive unit. It may not have (must not have) any data members. Thus, an interface contains no implementation whatsoever (not even data) and the concept is well established in languages such as C# and Java. Classes implement an interface by providing definitions (body) for each of its abstract methods and properties. An interface defines a *contract*.

An *abstract* `ref` class in C++/CLI is by definition a class that contains at least one pure virtual member or abstract method. It can have methods, properties, constructors and destructor but the key point to note is that it cannot be instantiated.

We note that an interface is a specification of *pure behaviour* (it is a non-structural description) while an abstract class may have structural components. A useful design pattern is when an abstract class implements some of the methods and properties of an interface while leaving the implementation of the other interface methods and properties to derived classes of the abstract class.

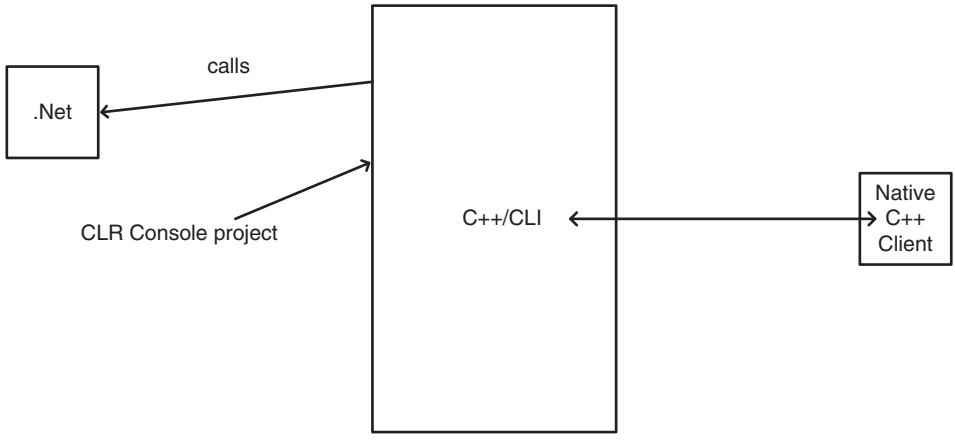


FIGURE 27.1 C++/CLI driver

27.7 USE CASE: C++/CLI AS 'MAIN LANGUAGE'

We now commence with our discussion of how to write applications that use a combination of C++/CLI, native C++ and C#. In general, we create some kind of Visual Studio project (for example, a CLR Console project) and we can add references (assemblies/dlls) to it that have been created using C# or C++/CLI. In this section we discuss the use case as shown in Figure 27.1.

In this case we create a *CLR Console project* which contains a mixture of C++/CLI code and native C++ code. Since this is a .NET project we can use the functionality from .NET as well as the functionality from other sources. We take a specific example for motivation. In this case we discuss different ways of using both C++/CLI and C++ to create vectors and sum their elements. Furthermore, we give a simple example of parsing regular expressions by calling functions from the .NET *Regex* library (we could have used regular expression functionality in C++11 or *Boost* but then the development team may not be using C++11 just yet and installing the full *Boost* library may be overkill in this case; using .NET is easier). We also give an example of vectors in the STL/CLR library since templates are supported in C++/CLI. First, we create an STL vector, a .NET array and a native C++ array and we compute the sum of their elements. The code is:

```
#include <vector>
#include <algorithm>
#include <numeric>
#include <iostream>

// STL/CLR
#include <cliext/vector>

using namespace System;
using namespace System::Collections::Generic;
using namespace System::Text::RegularExpressions;
```

```
int main(array<System::String ^> ^args)
{
   Console::WriteLine(L"Hello World");

   // C++/STL Arrays
   int N = 10;
   std::vector<double> v(N);
   std::iota(std::begin(v), std::end(v), 1);

   // .NET array
   cli::array<double>^ v2 = gcnew cli::array<double>(N);
   double v3[] = { 0,-1,-2,-3,-4,-5,-6,-7,-8,-9,-10 };
   for (int j = 0; j < v2->Length; ++j)
   {
      v2[j] = v[j];
      v3[j] = v[j];
   }

   // Sum the values in the arrays
   double initVal = 0.0;
   double acc = std::accumulate (std::begin(v), std::end(v), initVal);
   std::cout << "STL sum: " << acc << '\n';

   double acc2 = v2[0];
   double acc3 = v3[0];
   for (int j = 1; j < v2->Length; ++j)
   {
      acc2 += v2[j]; acc3 += v3[j];
   }
   Console::WriteLine(L".NET sum: {0},{1} ", acc2, acc3);

   // STL/CLR examples
   cliext::vector<int> v4(10);
   for (std::size_t j = 0; j < v4.size(); ++j)
   {
      v4[j] = j + 1;
   }

   for (std::size_t j = 0; j < v4.size(); ++j)
   {
      std::cout << v4[j] << ", ";
   }

   return 0;
}
```

Here we see .NET code and native code and they can be freely mixed to suit our needs. Other examples, test cases and applications can be conjured up to show how easy it is to mix the two languages. Finally, we show how to use regular expressions:

```
// Calling the .Net Regular Expression library
cli::array<String^>^ sentence =
{
   "Daisy, Daisy",
   "Give me your answer do",
   "I'm half crazy",
   "All for the love of you"
};

String^ matchStr = "Daisy";
for (int i = 0; i<sentence->Length; i++)
{
   Console::Write("*{0}", sentence[i]);
   if (Regex::IsMatch(sentence[i], matchStr, RegexOptions::IgnoreCase))
   {
      Console::WriteLine
         ("(match for '{0}' found)", matchStr);
   }
   else
   {
      Console::WriteLine("");
   }
}
```

This set of examples constitutes what we could call *basic training* in C++/CLI. We are unable to jump into more syntax in this chapter because we wish to discuss applications. For more background information, see Heege (2007), Hogenson (2008) and Templeman (2013). In general, Heege (2007) would be a suitable reference for C++ programmers in our opinion.

27.8 USE CASE: CREATING PROXIES, ADAPTERS AND WRAPPERS FOR LEGACY C++ APPLICATIONS

We discuss how to create applications in a given language (call it A) by using functionality in the same or other languages (call it B). In general, we first need to adapt the interface of B to suit the needs of A and second we need to control access to B because we are only interested in exporting or exposing a subset of the functionality of B. The design patterns to use are *Adapter*, *Façade* and *Proxy* (GOF, 1995).

> *In software engineering, the* Adapter *pattern is a software design pattern (also known as* Wrapper*, an alternative naming shared with the* Decorator *pattern) that allows the interface of an existing class to be used as another interface. It is often used to make existing classes work with other classes without modifying their source code.*

The *Façade* pattern (also called *Facade*) is a software design pattern commonly used in object-oriented programming. The name is based on the analogy to an architectural façade.

A façade is an object that provides a simplified interface to a larger body of code, such as a class library. A façade can:

- S1: Make a software library easier to use, understand and test, since the façade has *convenience methods* for common tasks.

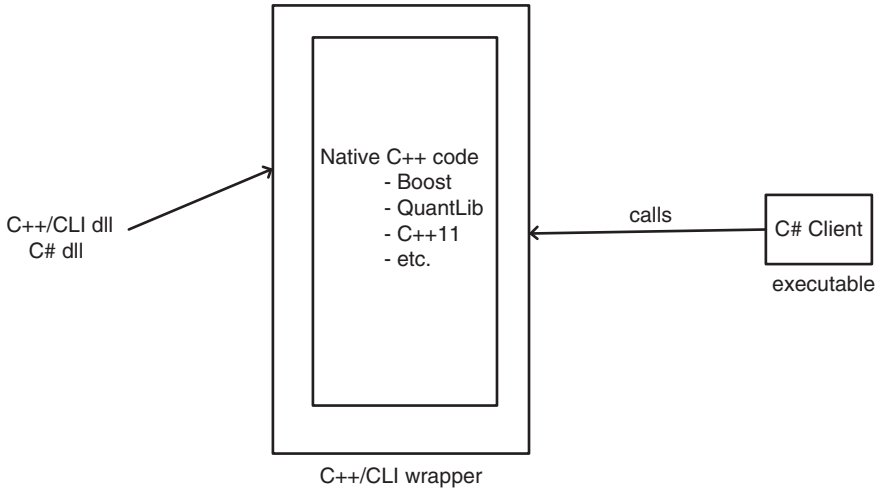


FIGURE 27.2 Wrapping legacy C++ code

- S2: Make the library more readable.
- S3: Reduce dependencies between client code and the inner workings of a library, since most code uses the façade, thus allowing more flexibility in developing the system.
- S4: Wrap a poorly designed collection of APIs with a single well-designed API.

The third design pattern is the *Proxy*:

A proxy, in its most general form, is a class functioning as an interface to something else. The proxy could interface to anything: a network connection, a large object in memory, a file, or some other resource that is expensive or impossible to duplicate. In short, a proxy is a wrapper or agent object that is being called by the client to access the real object behind the scenes. Use of the proxy can simply be forwarding to the real object, or can provide additional logic. In the proxy, extra functionality can be provided, for example caching when operations on the real object are resource intensive, or checking preconditions before operations on the real object are invoked. For the client, usage of a proxy object is similar to using the real object, because both objects implement the same interface.

Our first application of these patterns is sketched in Figure 27.2. In this case we create a C++/CLI class that is composed of a native C++ class and whose member functions delegate to those of the latter class. In this case we are interested in wrapping functions from both *Boost* and *Quantlib*. We first create a *C++/CLI Class Library dll* and then a *C# Console project* that references this dll. To this end, the wrapper class for *Boost* and *Quantlib* (we do not show the headers here; they can be found on the distribution medium) is:

```
#include "stdafx.h"

#include "BoostMath.hpp"
#include "BVNWrapper.hpp"
```

```
namespace Wrapper
{

   // Cdf

   double BoostMath::Cdf(UniformDistribution^ distribution, double x)
   {
      return boost::math::cdf(*distribution->GetNative(), x);
   }

   double BoostMath::Cdf(BernoulliDistribution^ distribution, double x)
   {
      return boost::math::cdf(*distribution->GetNative(), x);
   }

   double BoostMath::Cdf(ChiSquaredDistribution^ distribution, double x)
   {
      return boost::math::cdf(*distribution->GetNative(), x);
   }

   double BoostMath::Cdf(NonCentralChiSquaredDistribution^ distribution,
            double x)
   {
      return boost::math::cdf(*distribution->GetNative(), x);
   }


   // Pdf

   double BoostMath::Pdf(UniformDistribution^ distribution, double x)
   {
      return boost::math::pdf(*distribution->GetNative(), x);
   }

   double BoostMath::Pdf(BernoulliDistribution^ distribution, double x)
   {
      return boost::math::pdf(*distribution->GetNative(), x);
   }

   double BoostMath::Pdf(ChiSquaredDistribution^ distribution, double x)
   {
      return boost::math::pdf(*distribution->GetNative(), x);
   }

   double BoostMath::Pdf(NonCentralChiSquaredDistribution^ distribution,
                           double x)
   {
      return boost::math::pdf(*distribution->GetNative(), x);
   }
```

```
    // Quantile

    double BoostMath::Quantile(UniformDistribution^ distribution, double x)
    {
       return boost::math::quantile(*distribution->GetNative(), x);
    }
    double BoostMath::Quantile(BernoulliDistribution^ distribution,
                    double x)
    {
       return boost::math::quantile(*distribution->GetNative(), x);
    }

    double BoostMath::Quantile(ChiSquaredDistribution^ distribution,
                   double x)
    {
       return boost::math::quantile(*distribution->GetNative(), x);
    }

    double BoostMath::Quantile
          (NonCentralChiSquaredDistribution^ distribution, double x)
    {
       return boost::math::quantile(*distribution->GetNative(), x);
    }
}
```

These are static member functions that accept C++/CLI classes as input arguments. For the uniform distribution class (the others are the same) the wrapper class is:

```
#include "stdafx.h"

#include "UniformDistribution.hpp"

namespace Wrapper
{
   // Default constructor
   UniformDistribution::UniformDistribution()
   {
      m_distribution=new boost::math::uniform_distribution<>();
   }

   // Constructor with lower and upper value
   UniformDistribution::UniformDistribution(double lower, double upper)
   {
      m_distribution=new boost::math::uniform_distribution<>
                                                  (lower, upper);
   }

   // Finaliser (called by garbage collector or destructor)
   UniformDistribution::!UniformDistribution()
```

```
    {
       delete m_distribution;
    }

    // Destructor (Dispose)
    UniformDistribution::~UniformDistribution()
    {
       // Call finaliser
       this->!UniformDistribution();
    }

    // Get the lower value
    double UniformDistribution::Lower()
    {
       return m_distribution->lower();
    }

    // Get the upper value
    double UniformDistribution::Upper()
    {
       return m_distribution->upper();
    }

    // Get the native object
    boost::math::uniform_distribution<>* UniformDistribution::GetNative()
    {
       return m_distribution;
    }

}
```

The wrapper class for *Quantlib* is:

```
#include <string>
#include <memory>

#include <ql/quantlib.hpp>
#include <ql/math/distributions/bivariatenormaldistribution.cpp>
using NativeClass = QuantLib::BivariateCumulativeNormalDistributionWe04DP;

using namespace System;

namespace Wrapper
{

    // ManagedWrapper has similar interface to native that it wraps
    public ref class ManagedWrapper
    {
    private:
       NativeClass* m_nativeClass;   // The embedded native class
    // std::unique_ptr<NativeClass> m_nativeClass;   // not possible!
       double _rho;
```

```
    public:
        // Constructor with data
        ManagedWrapper(double rho) : _rho(rho)
        {
            // Create native class instance with data
            m_nativeClass = new NativeClass(_rho);
        }

        double Compute(double a, double b)
        {
            return (*m_nativeClass)(a, b);
        }

        // Get- and set data as property
        property double Correlation
        {
            double get() { return _rho; }
            void set(double value)
            {
                _rho = value;

                delete m_nativeClass;
                m_nativeClass = new NativeClass(_rho);
            }
        }

        !ManagedWrapper()
        {
            delete m_nativeClass;
        }

        ~ManagedWrapper()
        {
            this->!ManagedWrapper();
        }
    };
}
```

The class `Native class` contains the related *Quantlib* functionality.

We have placed both sets of source code in the same file for convenience. In the interest of maintenance we would prefer to create a separate dll for each library. In that case the class library project should reference to header and library files, if applicable.

How do we use the new wrapped code? Referring to Figure 27.2 again, we create a C# Console project and we add relevant dlls as reference:

```
using System;

using Wrapper;

class MainClass
```

```
{
   static void Main(string[] args)
   {
      double x=0.25;
      int k=0;

      // Uniform distributions
      UniformDistribution myUniform=new UniformDistribution(0.0, 1.0);
      Console.WriteLine("Lower value: {0}, Upper value: {1}",
                        myUniform.Lower(), myUniform.Upper());

      Console.WriteLine("pdf of Uniform: {0}", BoostMath.Pdf
      (myUniform, x));
      Console.WriteLine("cdf of Uniform: {0}", BoostMath.Cdf
      (myUniform, x));
      Console.WriteLine();

      // Bernoulli distributions
      BernoulliDistribution myBernoulli=new BernoulliDistribution(0.4);
      Console.WriteLine("Probability of success: {0}",
                           myBernoulli.SuccessFraction());

      Console.WriteLine("pdf of Bernoulli: {0}", BoostMath.Pdf
      (myBernoulli, k));
      Console.WriteLine("cdf of Bernoulli: {0}", BoostMath.Cdf
      (myBernoulli, k));
      Console.WriteLine();

      // Chi squared distributions
      ChiSquaredDistribution myChiSquared =new
      ChiSquaredDistribution(0.4);

      Console.WriteLine("pdf of ChiSquared: {0}", BoostMath.Pdf
      (myChiSquared, 2));
      Console.WriteLine("cdf of ChiSquared: {0}", BoostMath.Cdf
      (myChiSquared, 2));
      Console.WriteLine();

      // Non central chi squared distributions
      NonCentralChiSquaredDistribution myNonCentralChiSquared
         = new NonCentralChiSquaredDistribution(2, 2);

      Console.WriteLine("pdf of NonCentralChiSquared: {0}",
                           BoostMath.Pdf(myNonCentralChiSquared, 2));
      Console.WriteLine("cdf of NonCentralChiSquared: {0}",
                           BoostMath.Cdf(myNonCentralChiSquared, 2));
      Console.WriteLine();

      // C# code here: Check output data from chapter 16, section 16.4
      ManagedWrapper mw = new ManagedWrapper(0.948175);
      Console.WriteLine("rho {0}", mw.Correlation);
```

```
        Console.WriteLine("value {0}",
                            mw.Compute(-4.7346574, 3.0501621));

        mw.Correlation = 0.17220467;
        Console.WriteLine("rho {0}", mw.Correlation);
        Console.WriteLine("value {0}", mw.Compute(-2.726639,4.9965896));

        mw.Correlation = 0.054292586;
        Console.WriteLine("rho {0}", mw.Correlation);
        Console.WriteLine("value {0}", mw.Compute(-1.946981,3.1876528));

    }
}
```

The output is:

```
Lower value: 0, Upper value: 1
pdf of Uniform: 1
cdf of Uniform: 0.25

Probability of success: 0.4
pdf of Bernoulli: 0.6
cdf of Bernoulli: 0.6

pdf of ChiSquared: 0.0400666483385111
cdf of ChiSquared: 0.947619568720922

pdf of NonCentralChiSquared: 0.154254161276836
cdf of NonCentralChiSquared: 0.345745838723165

rho 0.948175
value 1.09712714868593E-06
rho 0.17220467
value 0.00319914983444984
rho 0.054292586
value 0.0257568370067285
```

The output is consistent with that produced in Section 16.4. This approach can be applied when you wish to use native C++ code from C# without having to port that C++ code to C#. This promotes code reusability.

27.8.1 Alternative: SWIG (Simplified Wrapper and Interface Generator)

SWIG is based on a well-established *interface definition language* (IDL) concept (Rogerson, 1997). The IDL is based on the *Open Software Foundation* (OSF) *Distributed Computing Environment* (DCE). The syntax resembles that of C and C++ and it is able to describe the interfaces and data shared by the client and the component whose services the client wishes to access. Having described interfaces and components in IDL the next step is to run them through

some kind of *IDL compiler* (for example, Microsoft's MIDL or the SWIG conversion tools). The compiler takes an IDL file as input and it generates code in some language, for example C#. In the particular case of SWIG, we call native C++ functions from C#. The SWIG tools create source code and this is the glue between C++ and the target language. We note that SWIG cannot be used for calling functions in the target language from native C++. We note that *Quantlib* supports SWIG.

27.9 'BACK TO THE FUTURE' USE CASE: CALLING C# CODE FROM C++11

We now turn to the use case as sketched in Figure 27.3. In this case we wish to use code that has been written in C# and we call it from native C++. One reason is that the C# code exists, has been tested and documented and we do not have the resources to rewrite the code again in C++. For this reason we create a CLR dll and we include the C# code in it. Then we create a CLR Console project and we call the code in the dll. Done! We do not have to create wrappers.

We take an example of a data interpolation library that we have written in C# for fixed-income applications (the full details are discussed in Duffy and Germani, 2013). Popular interpolators are:

- Akima
- Hyman 1989 filter
- Hagan and West
- Linear
- Cubic spline.

In this section we create a dll and we place code for the Akima and Hyman methods in it. We also put the C# version of the Excel driver code (Chapter 14 discusses the C++ version) into the dll for convenience. The source code is in Duffy and Germani (2013) and we do not

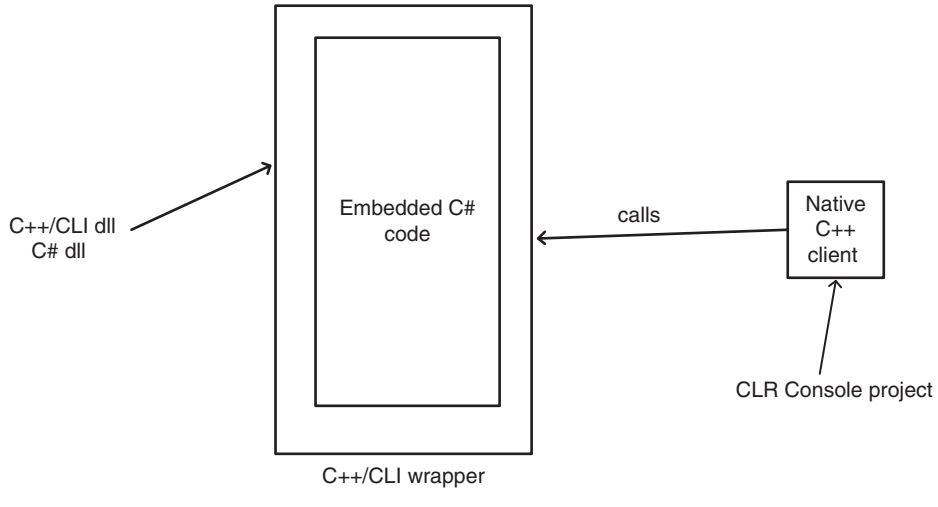


FIGURE 27.3 Value-added C++

include it here due to lack of space. We ported the test code (but not the library code that remains in C#) in C# to C++/CLI. A test program is:

```
// Author Andrea Germani
// Copyright (C) 2012. All right reserved
//
// Ported to C++/CLI Daniel J. Duffy
//
// (C) Datasim Education BV 2018
//
// ------------------------------------------------------------
using namespace System;
using namespace System::Collections::Generic;
//using namespace System::Linq;
using namespace System::Text;

#include <vector>

// Based on Datasim interpolators
public ref class InterpolatorExample101
{
public:

   // 13.12.1   The 101 Example, from A to Z
   static void Example101()
   {
      // My excel mechanism
      ExcelMechanisms^ exl = gcnew ExcelMechanisms();
      // I Create initial t and r arrays.
      std::vector<double> v = { 0.1, 1.0, 4.0, 9.0, 20.0, 30.0 };
      Vector<double>^ t = gcnew Vector<double>(6, 0);
      for (int j = t->MinIndex; j <= t->MaxIndex; j++)
      {
         t[j] = v[j];
      }

      // (gcnew double() { 0.1, 1, 4, 9, 20, 30 }, 0);
      Vector<double>^ r = gcnew Vector<double>(6, 0);
      r[0] = 0.081; r[1] = 0.07; r[2] = 0.044;
      r[3] = 0.07; r[4] = 0.04; r[5] = 0.03;


      // Compute log df
         Vector<double>^ logDF = gcnew Vector<double>(r->Size,
         r->MinIndex);
         for (int n = logDF->MinIndex; n <= logDF->MaxIndex; ++n)
         {
           logDF[n] = Math::Log(Math::Exp(-t[n] * r[n]));
         }
         exl->printOneExcel<double> (t, logDF, "logDF",
                                          "time", "logDF", "logDF");
```

```
// II Hyman interpolator
HymanHermiteInterpolator_V4^ myInterpolatorH
      = gcnew HymanHermiteInterpolator_V4(t, logDF);

// Create the abscissa values f (hard-coded for the moment)
int M = 299;
Vector<double>^ term = gcnew Vector<double>(M, 0);
term[term->MinIndex] = 0.1;
double step = 0.1;
for (int j = term->MinIndex + 1; j <= term->MaxIndex; j++)
{
   term[j] = term[j - 1] + step;
}

// III Compute interpolated values
Vector<double>^ interpolatedlogDFH = myInterpolatorH->Curve(term);
exl->printOneExcel<double>(term, interpolatedlogDFH,
             "Hyman cubic", "time", "int logDF", "int logDF");

Akima1970Interpolator ^ myInterpolatorA
      = gcnew Akima1970Interpolator(t, logDF);

// IV Compute continuously compounded risk free rate from the
// ZCB Z(0,t),using equation (3)Hagan and West (2008).
Vector<double>^ rCompounded = gcnew Vector<double>
              (interpolatedlogDFH->Size, interpolatedlogDFH->
              MinIndex);

for (int j = rCompounded->MinIndex; j <= rCompounded->MaxIndex; j++)
{
   rCompounded[j] = -interpolatedlogDFH[j] / term[j];
}
exl->printOneExcel<double>(term, rCompounded,
      "RCompound Hyman Cubic", "time",
         "r continuously comp.", "r cont");

// Akima
Vector<double>^ interpolatedlogDFA = myInterpolatorA->Curve(term);
exl->printOneExcel<double>(term, interpolatedlogDFA,
   "Akima cubic", "time", "int logDF", "int logDF");

Vector<double>^ rCompounded2 = gcnew
Vector<double>(interpolatedlogDFH->Size,
              interpolatedlogDFH->MinIndex);

for (int j = rCompounded->MinIndex;j<=rCompounded->MaxIndex; j++)
{
   rCompounded[j] = -interpolatedlogDFH[j] / term[j];
}
exl->printOneExcel<double>(term, rCompounded,
```

```
            "RCompound Hyman Cubic", "time", "r continuously comp.",
               "r cont");

         // Compute discrete forward rates equation (6) Hagan and West (2008)
         Vector<double>^ f = gcnew Vector<double>
                (rCompounded->Size, rCompounded->MinIndex);
         f[f->MinIndex] = 0.081;

         for (int j = f->MinIndex + 1; j <= rCompounded->MaxIndex; j++)
         {
            f[j] = (rCompounded[j] * term[j] - rCompounded[j - 1]
               * term[j - 1]) / (term[j] - term[j - 1]);
         }
         exl->printOneExcel<double>(term, f, "Hyman Cubic", "time",
                        "discrete forward", "dis fwd");
      }


   };
```

A simple test program is:

```
int main()
{
   InterpolatorExample101::Example101();

   return 0;
}
```

Part of the output is shown in Figure 27.4. This reflects the results in Duffy and Germani (2013).

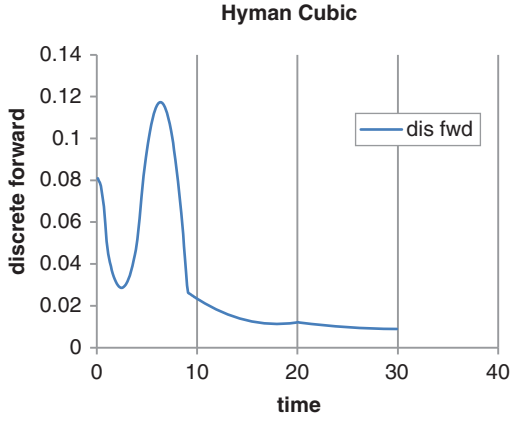


FIGURE 27.4 Hyman interpolation applied to discrete forward rate

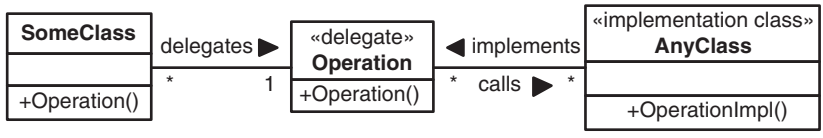


FIGURE 27.5 Delegate as proxy (intermediary)

27.10 MODELLING EVENT-DRIVEN APPLICATIONS WITH DELEGATES

In this section we introduce *Delegates* in the .NET Framework. From a design perspective delegates are a realisation of the *Proxy* pattern; in this case a delegate plays the role of the proxy and it has a reference to a *target method* having the same signature as that of the proxy interface. Clients communicate with the delegate without knowing (or having to know) the details of the target method 'behind' the delegate. It is possible to replace one target method by another one at run-time. Client code communicates directly with the delegate and not with any target method. A UML diagram showing the structural relationships between these entities is given in Figure 27.5.

A *delegate type* is a specification of a *function signature*, that is it has a return type, a name and zero or more input arguments. In this sense the delegate type represents a protocol or *contract*. We note that a delegate type has no body. In a sense it is similar to an abstract method in that it must be instantiated so that clients can use it. In native C++, a delegate is similar to the *universal function wrapper* `std::function`. To this end, we introduce the concept of a *delegate instance*, which is a specific method with a body whose signature conforms to that of the delegate type.

We take our first example in C#. Consider the delegate type that specifies scalar-valued functions of a scalar variable:

```
delegate double ComputableFunction(double x);   // The protocol
```

This type can now be assigned to specific delegate instances (called *target methods*) if these instances have the same signature as that of the type. For convenience, we define some instances as static methods:

```
class Delegates

   // Some delegate instances
   public static double Square(double x)
   {
      return x * x;
   }

   public static double Cube(double x)
   {
      return x * x * x;
   }
```

```
    public static double ModifiedExponential(double x)
    {
       return x * Math.Exp(x);
    }

    //

}

// end of class
```

Clients can then assign the given type to any of these instances, for example:

```
// Basic usage of delegates
ComputableFunction fun = Delegates.Square;
double val= 2.0;
Console.WriteLine("Square of {0} is {1}", val, fun(val));

fun = Delegates.Cube;
Console.WriteLine("Cube of {0} is {1}", val, fun(val));

fun = Delegates.ModifiedExponential;
Console.WriteLine("Mod. exponential of {0} is {1}", val, fun(val));
```

We see that the client code is shielded from specific implementations because it uses the variable `fun` which is a delegate type that can be assigned to any *conforming* target method. In this case the client is responsible for both the creation of delegate instances and assigning the delegate type to them. In a more general setup the client would not create these instances directly but it would instead outsource their creation to dedicated factory objects (in keeping with the SRP). In general, we can say that a delegate is similar to a *callback* and to a C *function pointer*.

27.10.1 Next-Generation Strategy (Plug-in) Patterns

One of the goals of software design patterns is to create code that can be modified to suit new requirements. There are many corresponding use cases and solutions and in this section we discuss the particular case of being able to customise the body of a method. To this end, we abstract the body of the method away by using a delegate having a signature consistent with that of the method. Instead of hard-coding the method (which is inflexible) we use a delegate in *server code* that can be assigned to a target method by the *client*. But where do we define the delegate and how is it used in the server? There are two options:

- *State based*: the server is constructed with an embedded delegate.
- *Stateless*: the *plug-in method* is called by providing a delegate as input parameter.

We take an example of a class that generates arrays of numbers. We transform an array by applying a function to each of its elements. We consider both the state-based and stateless solutions in a single class for convenience. We define:

```
delegate double ComputableFunction(double x);   // The protocol
```

The C# class to generate numbers is:

```
public class ArrayGenerator
{ // Create an array of values based on a customisable mathematical
  // function that is implemented using delegates

   private ComputableFunction func;      // Embedded Strategy algo
                                         // (plugin method)

   public ArrayGenerator(ComputableFunction myFunction)
   {
      func = myFunction;
   }


   // State-based delegate form
   public double[] ComputeArray(double[] array)
   { // Compute new arrays using the plugin method as data member

      double[] result = new double[array.Length];

      for (int j = 0; j < result.Length; j++)
      {
            result[j] = func(array[j]);
      }
      return result;
   }

   // Stateless delegate form
   static public void ComputeArray(ref double[] array,
   ComputableFunction plugin)
   { // Compute new arrays using a plugin method as input argument

      for (int j = 0; j < array.Length; j++)
      {
         array[j] = plugin(array[j]);
      }
   }

   static public void Print(double[] array)
   {
      Console.WriteLine('\n');
      for (int j = 0; j < array.Length; j++)
      {
         Console.Write("{0},", array[j]);
      }
   }
}
```

This class is a simple factory to transform an input array to an output array using both state-based and stateless delegates. To test the code, we take an example of an operation that squares

the elements of an array. We then take the square root of the elements of the transformed array. The following code uses C# lambda functions (for readability) to define the transformations. The C# code is:

```
// Basic input array
int M = 2;
double[] array = new double[M]; array[0] = 2.0; array[1] = 2.0;

ComputableFunction SquareRoot = x => Math.Sqrt(x);
ComputableFunction Square = x => x*x;

ArrayGenerator generator = new ArrayGenerator(M, Square);
double[] transformedArray = generator.ComputeArray(array);
                                                    // Square elements
ArrayGenerator.Print(transformedArray);             // 4,4

ArrayGenerator.ComputeArray(ref transformedArray,
SquareRoot);                                        // Square root
ArrayGenerator.Print(transformedArray);             // 2,2
```

The above code is essentially the *Strategy* pattern using delegates. It is more flexible than *Strategy* because the amount of coupling between server and client is reduced when compared with the object-oriented approach (we do not need pointers to the base class). *The focus is on function signature rather than on subtype polymorphism*. Furthermore, it is not necessary to create a class hierarchy as is the case with the *Strategy* pattern.

A generalisation of this approach is when the algorithm is more complicated. For example, the algorithm may have a well-defined structure consisting of several customisable steps, each step representing some part of the algorithm. In this case we can implement each step using a combination of stateless and state-based delegates and this corresponds to the *Template Method* pattern discussed in GOF (1995). The C++/CLI code (that was parted from the corresponding C# code) for this example is:

```
// TestStrategy.cpp
//
// Not: Showing the use of Delegates and lambda functions.
// Showing how to achieve the same effect with interfaces.
//
// Ported from C#
//
// (C) Datasim Education BV 2010-2018
//

using namespace System;
using namespace System::Collections::Generic;
using namespace System::Text;
using namespace System::Text::RegularExpressions;

public delegate double ComputableFunction(double x);   // The protocol
```

```
/* interface class ICompute
{
   double ComputableFunction(double x);
};*/

public ref class ArrayGenerator
{ // Create an array of values based on a customisable mathematical
  // function that is implemented using delegates

private:
   int N;                    // Size of array
   ComputableFunction^ func;

public:
   ArrayGenerator(int size, ComputableFunction^ myFunction)
   {
      N = size;
      func = myFunction;
   }

   List<double>^ ComputeArray()
   {
      List<double>^ result = gcnew List<double>();

      for (int j = 0; j < N; j++)
      {
         result->Add(func(Convert::ToDouble(j)));
      }
      return result;
   }

   void ComputeAndPrint()
   {

        List<double>^ arr = ComputeArray();
        Console::WriteLine('\n');
        for (int j = 0; j < N; j++)
        {
          Console::Write(", {0}", arr[j]);
        }
        Console::WriteLine('\n');
   }
};

public ref class Delegates
{
public:
   static double Square(double x)
   {
      return x * x;
   }
```

```
    double Cube(double x)
    {
       return x * x * x;
    }
};
```

A test program in C++/CLI is:

```
int main()
{
   int N = 4;
   ComputableFunction^ cFun = gcnew ComputableFunction
   (Delegates::Square);

   Delegates^ del = gcnew Delegates();
   ComputableFunction^ cFun2 = gcnew ComputableFunction
   (del, &Delegates::Cube);
   ArrayGenerator^ a1 = gcnew ArrayGenerator(N, cFun);
   ArrayGenerator^ a2 = gcnew ArrayGenerator(N, cFun2);

   a1->ComputeAndPrint();
   a2->ComputeAndPrint();

   return 0;
}
```

27.10.2 Events and Multicast Delegates

The *.NET Framework* has a pattern for defining *multicast events*. Before we discuss the details we give an overview of some issues when we investigate the *basis of communication* between modules in software systems (see Shaw and Garlan, 1996). In particular, we classify systems according to whether communication between modules depends upon shared state, events or both. To this end, an *event* is a transfer of information occurring at a discrete time. This classification is:

a) *Events*: no shared state; all communication between modules relies on events.
b) *Pure state*: communication is solely by means of shared state; the *receiver* (client) must repeatedly inspect or *poll* the state variables to detect changes.
c) *State with hints*: communication is by means of shared state but the receiver is actively notified of changes by an event mechanism. Polling of the state is not needed. The receiver may choose to ignore events and reconstruct all necessary information from the shared state. In this case the events are *hints* to the receiver.
d) *State and events*: both shared state and events are used. The events are crucial because they provide information that is not available from monitoring of the state.

The *Standard Event Pattern* is an implementation of cases a), c) and d) above. To this end, the predefined class `System.EventArgs` (or `System::EventArgs` in C++/CLI) is a base class that contains information pertaining to an event. Subclasses of `System.EventArgs`

expose data as properties or as read-only fields. We also need to define the delegate for the event. The rules are:

- The delegate must have `void` as return type.
- The delegate must have two arguments. The first argument is of type *object* and it represents the sender of the event while the second argument contains the extra information to convey to the receiver.
- The name of the delegate must end in *EventHandler*.

In general, we are interested in the generic delegate `System.EventHandler<>` that satisfies these rules and whose generic parameter corresponds to the class whose data (or a subset of the data) is conveyed from sender to receiver. A UML class diagram to show the relationships between the entities is shown in Figure 27.6. It is similar to the *Observer* (*Publisher–Subscriber*) pattern:

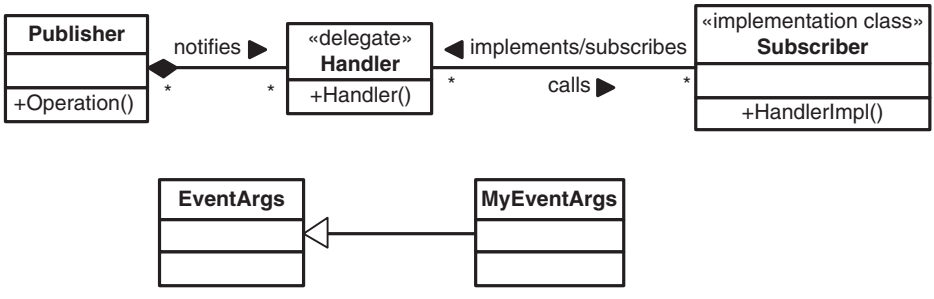


FIGURE 27.6 Model for the Standard Event Pattern

We take an example. In this case, changes in the value of an object result in a notification in connected subscribers. The new C# (or C++/CLI) code is similar to how we would implement the *Observer* pattern. The solution has now been standardised. The model for the data that is conveyed from *sender* to *receiver* is:

```
// V1: public delegate void ChangeHandler(double oldValue,
// double newValue);
public class ValueChangedEventArgs : EventArgs
{ // The class containing the conveyed data from sender to receiver

   public readonly double oldValue;
   public readonly double newValue;

   public ValueChangedEventArgs(double o, double n)

   {
      oldValue = o;
      newValue = n;
   }
}
```

Next, the class that plays the role of *broadcaster* is:

```
public class Subject
{ // The Broadcaster

   private double val;

   public Subject(double myValue) { val = myValue; }

   // V1: public event ChangeHandler ch;
   public event EventHandler<ValueChangedEventArgs> valueChanged;

   protected virtual void OnValueChanged(ValueChangedEventArgs e)
   {
      if (valueChanged != null) valueChanged(this, e);
   }


   public double Value
   {
      get { return val; }
      set
      {
         if (val == value) return;

         double oldValue = val;
         val = value;

         // Update the subscriber
         OnValueChanged(new ValueChangedEventArgs(oldValue, val));
      }
   }
}
```

All the subscribers will be updated each time the data in the broadcaster changes. For example, a particular case is when there is more than a 10% relative increase in the old value:

```
public static void MyChange(object sender, ValueChangedEventArgs e)
{
   if ((e.newValue - e.oldValue) / e.oldValue > 0.1)
   {
      Console.WriteLine("Alert, more than 10% rise");
   }
      else
   {
      Console.WriteLine("No news, good news");
   }
}
```

Finally, an example of use is:

```
static void Main()
{
   // Define broadcaster with empty event
   Subject s = new Subject(3.90);

   // Define target method for broadcaster's event
   s.valueChanged += MyChange;

   // Change propagation method
   s.Value = 24.0;
}
```

This particular example is interesting because it shows how to apply event notification patterns in C#. The code can be adapted to more general situations such as the *Observer* pattern and applications involving events in graphical user interfaces (GUIs) and database applications.

We now take the final example of the *Observer* pattern to monitor the passing of time. We implement the code (which we ported from C#) in C++/CLI. To this end, we create a clock class with an embedded changeable time:

```
using namespace System;
using namespace System::Threading;

public ref class TimeChangeEventArgs : EventArgs
{
public:
   DateTime dt;

   TimeChangeEventArgs(DateTime dt)
   { // Constructor

      this->dt = dt;
   }
};


public ref class Clock
{
public:
// Define event delegate
   delegate void TimeChangeEventHandler(Object^ sender,
   TimeChangeEventArgs^ e);

   // Event variable to store subscribers. Note, it also works
   // without the "event" keyword but with "event".
   event TimeChangeEventHandler^ OnTimeChange;

   void Run()
   {
```

```
        for (;; Thread::Sleep(1000)) // Infinite loop, sleeps
        {
           // Get the current time
           TimeChangeEventArgs^ args
              = gcnew TimeChangeEventArgs(DateTime::Now);

           // Raise event and call event methods
           // C# and C++/CLI versions
           // if (OnTimeChange != nullptr) OnTimeChange(this, args);
           OnTimeChange(this, args);// Not necessary to check for null
        }
     }
};
```

We now create a subscriber class with functionality to display clock information in different ways:

```
public ref class ClockSubscriber
{
public:
   static void DisplayTime1(Object^ sender,
                    TimeChangeEventArgs^ args)
   { // TimeChangeEventHandler 1

      Console::WriteLine("DisplayTime 1: {0}", args->dt);
   }

   static void DisplayTime2(Object^ sender, TimeChangeEventArgs^ args)
   { // TimeChangeEventHandler 2

      // Display time with an offset
      int h = 24; int d = 0; int m = 0;
      TimeSpan ts(h, d, m);

      Console::WriteLine("Offset, DisplayTime 2: {0}", args->dt + ts);
   }
};
```

Finally, a test program is:

```
int main()
{
   // Create clock instance
   Clock^ clock = gcnew Clock();

   // Subscribe event handlers for Clock.TimeChangeEvent
   clock->OnTimeChange += gcnew
                    Clock::TimeChangeEventHandler(ClockSubscriber::
                    DisplayTime1);
```

```
    clock->OnTimeChange += gcnew
                    Clock::TimeChangeEventHandler(ClockSubscriber::
                    DisplayTime2);

    // Start the clock
    clock->Run();

    return 0;
}
```

In this example, we see how easy it is to create *one-to-many relationships* between event sources (*publishers*) and event sinks (*subscribers*).

27.11 USE CASE: INTERFACING WITH LEGACY CODE

We conclude this chapter with a discussion of how to interface .NET code with legacy C and COM (*Component Object Model*).

27.11.1 Legacy DLLs

In some cases we may wish to call Win32 library functions (for example, from .NET) that have no equivalent in .NET or functions that have not yet been converted to .NET. To this end, we use DLL functions using *PInvoke* (*Platform Invocation Services*). We import functions using the `DllImport` attribute and we declare them as being `extern`. The .NET Framework handles data type conversions. A typical example of use is:

```
using namespace System;
using namespace System::Text;
using namespace System::Runtime::InteropServices;

// Import "kernel32.dll" function: BOOL Beep(DWORD dwFreq, DWORD
// dwDuration)
[DllImport("kernel32.dll")] extern bool Beep
             (int frequency, int duration);

int main()
{

    // Play sound
    for (int i=37; i<=5000; i+=20) Beep(i, 10);
    for (int i=5000; i>=37; i-=20) Beep(i, 10);

    return 0;
}
```

In this case we created a CLR Console project.

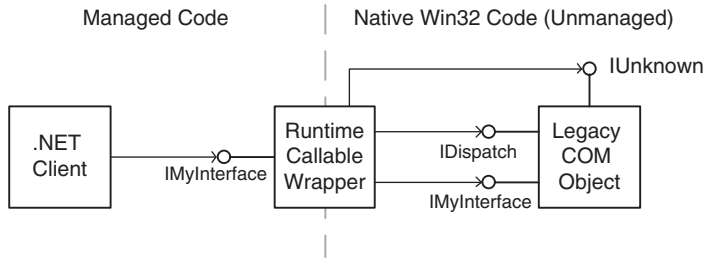


FIGURE 27.7 Runtime Callable Wrapper (RCW)

27.11.2 Runtime Callable Wrapper (RCW)

These wrappers can be used when we wish to use existing ActiveX controls or to use Office products such as *Word* and *Excel*. The general design is shown in Figure 27.7. We see that there is an *RCW proxy* between the .NET client code and the COM component. The runtime creates both the COM object being called and a wrapper for that object. Furthermore:

- The runtime maintains a single RCW object per process for each object.
- It maintains a cache of interface pointers on the COM object.
- It releases its reference on the COM object when it is no longer needed.
- The runtime performs garbage collection on the RCW.
- The RCW marshals data between managed and unmanaged code on behalf of the wrapped object for both method arguments and return values when client and server have different representations of the data between them.

We give an example of a .NET client that creates a Word document and places some text in it:

```
using namespace System;
namespace Word=Microsoft::Office::Interop::Word;


void main()
{
   // Object to pass as missing argument. Not the same as 'null'
   Object^ objMissing=System::Reflection::Missing::Value;

   try
   {
      // Create instance of Word and show it
      Word::Application^ app=gcnew Word::ApplicationClass();
      app->Visible=true;

      // Add document
      Word::Document^ doc=app->Documents->Add
          (objMissing, objMissing, objMissing, objMissing);
```

```
        // Set text of the document
        Word::Range^ range=doc->Content;
        range->default="Datasim Education BV";
    }
    catch(System::Runtime::InteropServices::COMException^ ex)
    {
        Console::WriteLine("HRESULT: {0}\n{1}",
        ex->ErrorCode, ex->Message);
    }
```

27.11.3 COM Callable Wrapper (CCW)

For some applications we may wish to use .NET components from non .NET applications, for example creating Office add-ins in .NET or accessing .NET code from VBA. To this end, the runtime automatically creates a COM proxy class around a .NET class. The design is shown in Figure 27.8. Some salient features of this design are:

- The runtime creates exactly one CCW for a managed object irrespective of the number of COM clients requesting its services.
- Multiple COM clients hold a reference to the CCW that exposes an interface.
- The CCW holds a single reference that implements the interface. It is garbage collected.
- CCWs are invisible to other classes running in the .NET Framework.
- CCWs marshal calls between managed and unmanaged code. They also manage the object identity and object lifetime of the managed objects that they wrap.

For more details on the topics in this section, see Bovey et al. (2009).

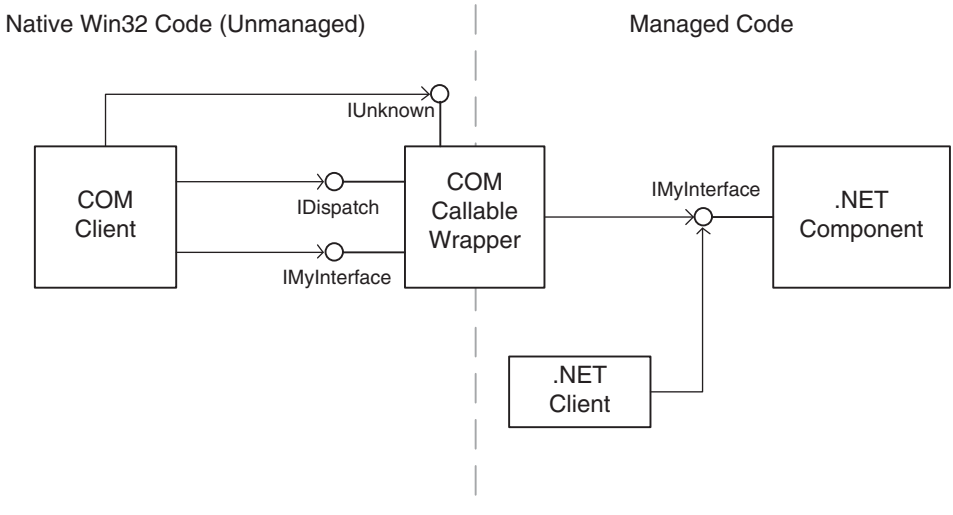


FIGURE 27.8 COM Callable Wrapper (CCW)

27.12 ASSEMBLIES AND NAMESPACES FOR C++/CLI

When developing applications in native C++ we have to decide how to distribute the code to clients. There are several options. First, the supplier can deliver the code as a collection of header-only files (as in the majority of libraries in *Boost*, for example) or as a combination of source files and (static) library files (as with *Quantlib*, for example). We have already discussed how to create static libraries in Chapter 8.

We now create applications in .NET by deploying assemblies. An *assembly* is the basic *unit of deployment* in .NET. It contains compiled types, *Intermediate Language* (IL) code, runtime resources and information relating to versioning, security and the referencing of other assemblies. At file level, an application is an assembly with a *.exe* extension while a reusable library is an assembly with a *.dll* extension.

An assembly contains the following entities:

- *Assembly manifest*: this is a description of the contents of the assembly such as the list of types, resources and files that make up the assembly, the list of assemblies referenced by this assembly and the required permissions to run the code in the assembly. It also contains the name of the assembly, its version and *culture* (a culture represents a particular human language; .NET supports the RFC 1766 standard that represents cultures and subcultures as two-letter codes. For example, the codes for English and German cultures are `en` and `de`, respectively, while Australian English and Austrian German have the codes `en-AU` and `de-AT`, respectively).
- *Application manifest*: this provides information to the operating system on how the assembly should be deployed. The application manifest is an XML file that imparts information concerning the assembly to the operating system.
- *Compiled code*: the IL code and metadata of the types and methods in the assembly. The presence of metadata implies that the assembly is *self-describing*.
- *Resources*: additional data and non-executable code such as strings, bitmaps and localisable text.

27.12.1 Assembly Types

Assemblies are similar to traditional DLLs because both are units of deployment. But in contrast to DLLs, assemblies do not need to define *Windows Registry* settings. Furthermore, assemblies support version control while DLLs do not and multiple assembly versions can be installed in applications.

The two major assembly types are called *private* and *shared assemblies*. A *private assembly* is one that can only be used by a single application. The corresponding *.dll* file is stored in the application directory. To create an assembly DLL we create a *Class Library* project in Visual Studio, add relevant classes to the project and then we compile and build the project. Having done that, we use this assembly DLL as part of an application by adding it *as a reference* to a Windows or Console application project. We note that the assembly DLL must be in the same directory as, or in a sub-directory of, the directory in which the main *.exe* file resides. A typical example of a private assembly is one that contains classes for dates, day count conventions and calendars. It can be used in many fixed-income applications.

From a software design viewpoint, we can design a software system as a set of loosely coupled subsystems. We package each subsystem as a separate assembly and each assembly can be designed and tested independently of the other assemblies. Having done that, we can deploy these assemblies as reusable components in various applications and they can be separately versioned. Finally, assemblies can be loaded at run-time, which improves performance and interoperability.

A *shared assembly* is one that can be used by many applications. It must be stored in a special directory called the *Global Assembly Cache* (GAC), which is a central repository for centralising version-based assemblies. As developer, it is possible to create shared assemblies and place them in the GAC but a discussion of this topic is outside the scope of this book. However, we do need to understand the structure of the GAC and how assembly versioning and security are realised. To this end, we give a short introduction to *strong names* that uniquely identify an assembly in the GAC.

A *strong name* is generated from the following components:

- The assembly 'simple' name (this is a string).
- The version number (of the form `a.b.c.d` containing numeric values).
- The culture.
- The assembly's public key.

Each assembly has a version number of the form `a.b.c.d` where `a` represents a major release, `b` a minor release, `c` is a build number and `d` is a revision. The default value is 0.0.0.0. The GAC can contain multiple versions of a shared assembly and different assembly versions can be instantiated at the same time. The client decides which version to use.

We can view the contents of the GAC by examining the contents of the directory `C:\Windows\assembly`. When creating Visual Studio projects you can add an assembly from the GAC by using the *Add Reference* option. We can choose between .NET and .COM components. For example, we use the GAC to include *Excel interop* into an application, thus allowing us to access the Excel object model.

27.12.2 Specifying Assembly Attributes in `AssemblyInfo.cs`

It is possible to control the contents of an assembly to a large extent by setting the values of the attributes in a file called `AssemblyInfo.cs` (or `AssemblyInfo.cpp`) that is automatically created with each new C# or C++/CLI project. It contains default values which you can change to suit your needs.

Some relevant attributes are:

- `AssemblyTitle`: a friendly name for the assembly.
- `AssemblyDescription`: a description of the assembly.
- `AssemblyConfiguration`: specifies if the assembly is for debug or release mode.
- `AssemblyCompany`: specifies the creator of the assembly.
- `AssemblyProduct`: specifies the name of the product.
- `AssemblyCopyright`: a copyright information string.

- `AssemblyTradenark`: specifies trademark information.
- `AssemblyCulture`: this attribute specifies the supported language; we leave it empty if the assembly is language neutral.

An example of the assembly file is:

```
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

// General Information about an assembly is controlled through the
// following set of attributes. Change these attribute values to modify
// the information associated with an assembly.
[assembly: AssemblyTitle("LoadDateLibrary")]
[assembly: AssemblyDescription("Dates and Day Counts")]
[assembly: AssemblyConfiguration("release")]
[assembly: AssemblyCompany("Datasim Education BV")]
[assembly: AssemblyProduct("Germani and Duffy")]
[assembly: AssemblyCopyright("Copyright © Datasim Education BV 2010")]
[assembly: AssemblyTrademark("TM Duffy Germani")]
[assembly: AssemblyCulture("")]

// Setting ComVisible to false makes the types in this assembly not
// visible to COM components. If you need to access a type in this
// assembly from COM, set the ComVisible attribute to true on that type.
[assembly: ComVisible(false)]

// The following GUID is for the ID of the typelib if this
// project is exposed to COM
[assembly: Guid("ec444bce-76db-4935-9017-78f5aed1e693")]

[assembly: AssemblyVersion("2.0.0.0")]
[assembly: AssemblyFileVersion("2.0.0.0")]
```

27.12.3 An Example: Dynamically Loading Algorithms from an Assembly

We have seen how to create C++ static libraries in Section 8.9. They resolve the problem of having to provide source code to clients. This solution is not very flexible as static libraries tend to be large monolithic beasts and they cannot be replaced by other libraries at run-time. The .NET Framework on the other hand supports *configurable assemblies*. To show how effective the use of assemblies is we take a model example from CAD (computer aided design) in which we create geometrical objects. In particular, we create flexible algorithms (*Strategy* design pattern) to calculate the distance between two points.

The UML class diagram is shown in Figure 27.9, which was originally discussed in Duffy and Germani (2013) and programmed in C#. What we do is to reuse the C# code from that book which implements the distance algorithms and we test them by encapsulating them in an

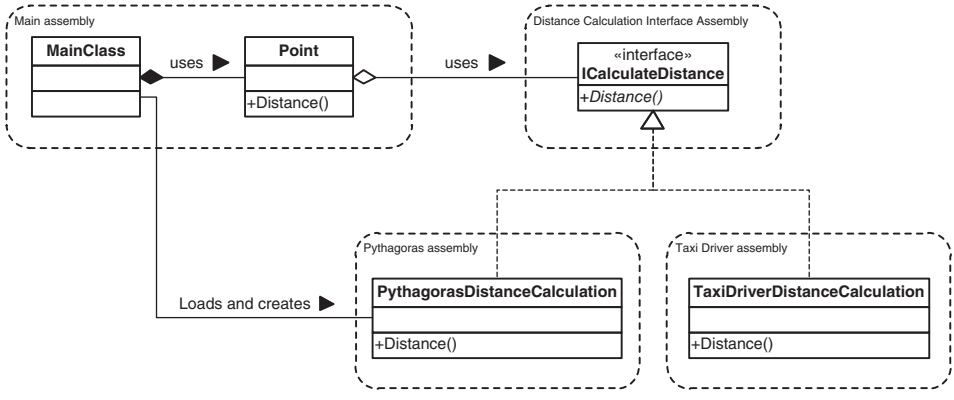


FIGURE 27.9 Dynamic assembly loading

assembly and then referencing the assembly in a CLR Console project containing C++/CLI code. For completeness, the C# code for the algorithms is:

```
public interface ICalculateDistance
{
   // Calculate the distance. We can't pass a point because then we
   // need to know about point but the point must know about this
   // interface. Then we get circular dependencies.
   double Distance(double x1, double y1, double x2, double y2);
}


public class PythagorasDistanceCalculation: ICalculateDistance
{
   public double Distance(double x1, double y1,
                  double x2, double y2)
   {
      double dx=x1-x2;
      double dy=y1-y2;

      return Math.Sqrt(dx*dx+dy*dy);
   }
}

public class TaxiDriverDistanceCalculation: ICalculateDistance
{
   public double Distance(double x1, double y1,
                    double x2, double y2)
   {
      double dx=x1-x2;
      double dy=y1-y2;

      return Math.Abs(dx) + Math.Abs(dy);
   }
}
```

We also implement a C# class implementing two-dimensional points containing an embedded global distance algorithm:

```
public class Point
{

   // The distance algorithm to use
   private static ICalculateDistance s_distanceAlgorithm;
      // Access the distance calculation algorithm
   public static ICalculateDistance DistanceAlgorithm
   {
      get { return s_distanceAlgorithm; }
      set { s_distanceAlgorithm=value; }
   }

   // Data members.
   private double m_x;
   private double m_y;

   // Default constructor
   public Point()
   {
      m_x=0.0;
      m_y=0.0;
   }

   // Constructor.
   public Point(double x, double y)
   {
      m_x=x;
      m_y=y;
   }

   // X property.
   public double X
   {
      get { return m_x; }
      set { m_x=value; }
   }


   public double Y
   {
      get { return m_y; }
      set { m_y=value; }
   }

   // Calculate distance between two points
   public double Distance(Point p)
   {
```

```
        // Is there an algorithm installed
        if (s_distanceAlgorithm==null) throw new
        ApplicationException
              ("No distance calc algorithm installed.");

        // Calculate the distance using the installed algorithm
        return s_distanceAlgorithm.Distance
              (m_x, m_y, p.m_x, p.m_y);
    }

    // String conversion.
    public override string ToString()
    {
        return String.Format("Point({0}, {1})", m_x, m_y);
    }
}
```

Finally, the client code is C++/CLI and it uses the above C# code for `Point` and algorithms:

```
using namespace System;

int main()
{
    try
    {
        // Load the algorithm, later
        //Point.DistanceAlgorithm=LoadDistanceAlgorithm(args[0]);

        Point::DistanceAlgorithm =
              gcnew PythagorasDistanceCalculation();
        // Point::DistanceAlgorithm =
              // gcnew TaxiDriverDistanceCalculation();

        Point^ p1=gcnew Point(0.0, 0.0);
        Point^ p2=gcnew Point(1.0, 1.0);

        Console::WriteLine("p1.Distance(p2): {0}",
                           p1->Distance(p2));
        }
        catch (Exception^ e)
        {
           Console::WriteLine("Error: {0}", e->Message);
        }
    }
```

The choice of algorithm is hard-coded in the above example. It could be made more flexible by allowing the user to make a choice as to which specific algorithm to use. Unfortunately, if we wish to use another algorithm we will need to modify the source code and build a new executable file. A more flexible and elegant solution is to employ the .NET *Reflection* to open

an assembly in an application and then extract the algorithm from it by instantiating the class that implements `ICalculateDistance`. In particular, we use the following functions:

- `LoadFrom` and `LoadFile` methods: load an assembly from a file.
- `Activator` class: has a method called `CreateInstance` that accepts a type and optional parameters that get passed to a constructor.

An example to instantiate a distance algorithm is:

```
// Load distance algorithm
private static ICalculateDistance LoadDistanceAlgorithm
                          (string assemblyFile)
{
   // Load the specified assembly
   Assembly ass=Assembly.LoadFrom(assemblyFile);

   // Get all the types from the assembly and
   // find the class that implements our interface
   foreach (Type t in ass.GetExportedTypes())
   {
      if (t.IsClass)
      {
         foreach (Type i in t.GetInterfaces())
         {
                  // Class found so create instance
                  if (i==typeof(ICalculateDistance))
                  return (ICalculateDistance)Activator.CreateInstance(t);
         }
      }
   }

   // Not found, in version 2 use exception handling
   return null;
}
```

We can use this function in the main method as follows:

```
Point.DistanceAlgorithm=LoadDistanceAlgorithm(args[0]);
```

This simple example is a prototype for larger and more complex applications. We are unable to elaborate on this topic but see Exercise 4 for a discussion of some interesting use cases.

27.13 SUMMARY AND CONCLUSIONS

In this chapter we introduced the C++/CLI and C# languages in the .NET Framework and we showed how it is possible to enhance the functionality of native C++ by the ability to create multi-language applications. Some important use cases were:

- Using C++/CLI as language of choice (instead of C#).
- Creating wrappers for C++ classes (for example, *Boost*, *Quantlib*, C++11 libraries) and using them in C# and C++/CLI applications.
- Creating wrappers for C# and C++/CLI code and then using this code in native C++ applications.

The main advantages are interoperability, reusability and maintainability.

27.14 EXERCISES AND PROJECTS

1. (Using *Façade* Design Pattern)

The *Façade* design pattern is often used when a system is very complex or difficult to understand because the system has a large number of interdependent classes. This pattern hides the complexities of the larger system and provides a simpler interface to the client. It typically involves a single wrapper class that contains a set of members required by the client. These members access the system on behalf of the *Façade* client and hide the implementation details.

Answer the following questions:

a) In Section 27.8 we enumerated the use cases whose presence in an application might persuade us to apply the *Façade* pattern. Which ones are most important in the general context of C++, C# and C++/CLI interoperability? For example, which features in the languages help in the construction of this pattern?

b) We now consider the use cases S1, S2, S3 and S4 in Section 27.8 when we wish to create user-friendly wrappers for the financial models in *Quantlib* for C# developers who do not necessarily have C++ knowledge. Which of the scenarios S1, S2, S3 and S4 are important in this case?

c) Now consider a (native) C++ developer who wishes to access .NET functionality. Which of the scenarios S1, S2, S3 and S4 are important in this case?

d) In GOF (1995) some of the consequences of using this pattern are:

- It shields clients from subsystem components.
- It promotes weak coupling between the subsystems and clients, especially for subsystems with tightly coupled components, for example poorly designed object-oriented code.
- The pattern does not prevent clients from using the subsystem classes directly if they wish to do so. Thus, clients can choose between ease of use and generality. New users can use the simpler interface before moving to the internals.

Given these features, how would you integrate *Façade* into the design philosophy that we introduced in Chapter 9?

e) What are the advantages of defining standardised interfaces between a client and a façade? How can the *Bridge* pattern be applied in order to create interchangeable façades?

2. (Database Coupling and Integration, Brainstorming)

C++ is primarily a systems language and there is no native support for *persistence*. In general, objects do not survive the life of the program in which they are created (these are called *transient* objects). A *persistent object store* is a type of computer storage system that records and retrieves complete objects, or provides the illusion of doing so. Simple

examples store the serialised object in binary format (zeros and ones). More complex examples include object databases or object-relational mapping systems, which combine a database system with support for storing objects. Many modern operating systems provide some sort of support for 'persisting' objects in flat files, often based on XML or JSON.

ADO.NET is a set of classes that expose data access services for .NET Framework programmers. ADO.NET provides a set of components for creating distributed, data-sharing applications. It is an integral part of the .NET Framework, providing access to relational, XML and application data. ADO.NET supports a variety of development needs, including the creation of front-end database clients and middle-tier business objects used by applications, tools, languages or Internet browsers.

Examples of database systems that support ADO.NET are Oracle, SQL Server, MySQL and Microsoft Access (for example, there is an interface to Excel, allowing it to be used as a database).

We reduce the scope by concentrating on using Excel as a database system for storing computed data in computational finance applications.

Answer the following questions:

a) Is there a requirement to support persistent objects in your applications?

b) Compare the approach to persistency to that taken using Excel add-ins.

c) The *Unified Software Design* approach in Chapter 9 is silent on the issue of supporting persistent objects. How would you modify the design to support databases such as Oracle, SQL Server, MySQL and Excel?

3. (Cross-Language Structural Design Patterns, I)

The original software design patterns in GOF (1995) (implicitly) assume that all objects have been created using the same language, for example C++. At the moment of writing we are able to create objects using a mixture of languages such as C#, C++ and C++/CLI. In general, the possible mismatches between client and server objects are compounded by language differences, for example:

- Memory model.
- Generic classes versus template classes.
- How built-in data types are represented and exchanged.
- Incompatible source (and possibly binary) code.

We now wish to formalise what we have produced. In order to reduce the scope we focus on the use case of Section 27.8 (calling native C++ code from .NET). The goal is to determine the applicability of and possible modifications to the following structural patterns: *Adapter*, *Bridge*, *Proxy* and *Decorator*. Possible use cases in this *multi-language development environment* are:

- U1: For a given use case, which design patterns can be used to realise it?
- U2: For a given pattern, to which use cases and applications can it be applied?

Answer the following questions:

a) What are the essential features of and differences between these patterns in general? What kinds of forces does each pattern resolve? They look similar but they do not serve the same purpose.

b) In Section 27.8 which of the above patterns was used to create wrappers for the statistical distributions in Boost?

c) Under which circumstances is the application of the *Bridge* pattern useful?

d) How can these structural patterns be integrated into the *Unified Software Design* approach as discussed in Chapter 9? In particular, focus initially on the interfaces

between the systems in the context diagram. In a later design phase we may see opportunities to use these design patterns at the detailed class level.

4. (Building an Application by Composing Assemblies, Brainstorming)

In .NET it is possible to compose an application by building it from precompiled self-aware binary assemblies as described in Section 27.12. In principle, this feature allows us to define a new generation of software design patterns and opportunities to extend the design blueprints that we introduced in Chapter 9.

The goal of this exercise is to pose a number of questions to trigger discussion on how to integrate assemblies into the design philosophy of Chapter 9 and how to extend the traditional software design patterns (GOF, 1995) to incorporate *Reflection* (Albahari and Albahari, 2015). We focus on creational design patterns. To reduce the scope we ask questions concerning the *canonical context diagram* in Figure 9.4 of Chapter 9. The goal is to configure the application by referencing assemblies, each of which is implemented as a dll.

Answer the following questions:

a) Determine the modifications and extensions to the design blueprints in Chapter 9 in order to support assemblies. Define a strategy to ensure that there are no function or class name collisions between the various assemblies in the application. Consider using namespaces and the many-to-many relationship between them and assemblies.

b) Test your answers from part a) by implementing the '101' application in Section 9.3.4 (`Input`, `Processing`, `Output` and SUD components). Create assemblies for each of `Input`, `Processing` and `Output`. Write code to allow the loading of assemblies at run-time.

c) Adapt the *Builder* pattern (and other creational patterns) to accommodate assemblies. In particular, we wish to create objects from code that is embedded in the application as well as being able to load the code from an assembly and using `Activator.CreateInstance` to create the objects. Clients should not (have to) know which software component is responsible for instantiation.

d) Investigate the impact of integrating assemblies with the *Strategy*, *Command* and *Visitor* patterns. In particular, we do not wish to embed source code in the executable file but instead we load functionality from an external assembly as needed. For example, a major issue with the traditional *Command* pattern is that many (sometimes hundreds of) command classes and instances are needed in applications. The code must be linked into the application. We avoid this problem by using assemblies. This approach reduces the executable file's footprint.

In general, the answers to the exercises go some way to realising the software architectures that are discussed in Leavens and Sitarman (2000).

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 28

C++ Concurrency, Part I Threads

28.1 INTRODUCTION AND OBJECTIVES

In this chapter we introduce the *C++ Concurrency* library by first focusing on how it supports multithreading, that is creating programs that are executed by multiple independent threads of control. The main goal of this chapter is to show how the library supports multithreading by introducing each syntax feature in the library, creating examples and then using these features to write simple and complete multithreaded programs. We also give initial examples and exercises from computational finance based on some of the numerical methods and option pricing models as already discussed in previous chapters.

This chapter is useful for the following reader groups:

- Developers who have little or no experience of writing multithreading programs. The chapter is structured in such a way that the code samples can be run and extended in order to get hands-on experience with the library.
- Developers with experience of proprietary or open-source multithreading libraries who wish to compare their functionality with that offered by C++ *Concurrency*.
- C++ developers who already use C++ *Concurrency*. This chapter could then function as a compact summary of the functionality and the examples and exercises hopefully provide insights into simple multithreaded code for a number of the numerical algorithms that we have discussed in previous chapters of this book.

In Chapter 29 we discuss how to write parallel programs using tasks. A *task* is a sequential operation that works together with other tasks to perform a larger operation (see Campbell and Miller, 2011 for an excellent discussion of parallel programming in Visual C++). We decompose a problem into large-grained components each of which can be assigned to a task. The use of tasks frees us from many of the low-level details that confront us if we decide to use threads.

In Chapter 30 we introduce the Microsoft *Parallel Patterns Library* (PPL) that implements a number of *parallel design patterns* in C++ (see Campbell and Miller, 2011; Mattson, Sanders and Massingill, 2006).

A number of the examples in this chapter contain code that was originally written using the *Boost Thread* library (Demming and Duffy, 2010). In most cases we were able to recompile and run the code just by replacing the namespace `boost` by the namespace `std`.

A useful piece of C++ syntax to determine how many processors there are on your system is:

```
// Display the number of processors/cores
std::cout<<std::thread::hardware_concurrency();
```

A discussion of *thread notification* is outside the scope of this chapter.

28.2 THREAD FUNDAMENTALS

In C++ the way to create and run threads is to instantiate the class `std::thread`. It has a number of constructors:

- Default constructor (not a thread).
- Move constructor.
- Creating a thread with a *callable object* as input argument.
- (Copy constructor is not supported.)

We create a thread using two stored lambda functions as *thread function* (we shall give more examples of callable objects in Section 28.3). The first function has no input arguments while the second function has two input arguments. The code is:

```
int main()
{
   // Create a thread + thread function
   auto f = [] () { std::cout << "Hello world\n"; };
   std::thread t(f);
   // Wait for thread to finish (mandatory)
   t.join();

   // Thread function having input arguments
   auto f2 = [](int n, std::string& s)
            { std::cout << n << " Hello world " << s; };
   std::thread t2(f2, 42, std::string("and beyond\n"));
   // Wait for thread to finish (mandatory)
   t2.join();
}
```

In this case the main thread creates (*forks*) two threads that start executing immediately. The main thread waits for the other threads to complete by calling `join()` on each one.

It is possible to query some properties of a thread, for example if it is still running, the thread's ID, getting a POSIX-compliant thread handle and querying the number of concurrent threads supported by the implementation:

```
void ThreadStatus(std::thread& t)
{
   std::cout << "Still running?"<<std::boolalpha
             <<  t.joinable() << '\n';
   std::cout << "Thread ID:  " << t.get_id() << '\n';
```

```
    // POSIX thread implementation
    std::cout << "Native handle:  " << t.native_handle() << '\n';
    std::cout << "Number of processors:  "
              << t.hardware_concurrency() << '\n';
}
```

We continue with another example that is more closely related to computational finance. In this case we look at STL algorithms, which we discussed in detail in Chapters 17 and 18, in particular numerical algorithms although the same principles can be applied to other algorithm categories. We take the example of computing the sum and product of the elements of a read-only vector. In this case there is no *data race* (we introduce the topic of data races in Section 28.7). To this end, we define two lambda functions that encapsulate data and STL algorithms. These functions are then used as input parameters to threads:

```
void STLinThreads(std::size_t n)
{ // Run two algorithms in parallel

   std::vector<double> v(n);
   std::iota(std::begin(v), std::end(v), 1);

   // Sum and product of elements of vector
   // Define lambda functions
   auto sum = [&]()
   { // For vector of length n -> n(n+1)/2

      double initVal = 0.0;   // Must be initialised to a value
      // Add initVal to all elements
      double acc = std::accumulate(v.begin(), v.end(), initVal);
      std::cout << "Sum: " << acc << std::endl;
   };

   auto product = [&]()
   {
      double initVal = 1.0;
      double acc = std::accumulate(v.begin(), v.end(), initVal,
      std::multiplies<double>());
      std::cout << "Product: " << acc << std::endl;
   };

   std::thread t1(sum); std::thread t2(product);
   t1.join(); t2.join();
}
```

This code is just one way to improve the speedup of sequential computations. For small arrays the above parallel code may even be slower than the corresponding sequential code and for large arrays the use of *reduction variables* in OpenMP may be more efficient (or using loop parallelism in PPL, as we shall see in Chapter 30).

28.2.1 A Small Digression into the World of OpenMP

We produce code in OpenMP similar to that above to compute the sum and product of the elements in a vector. In fact, learning some OpenMP is a good way to gain insight into multithreading. It is easy to write code and see what happens. We first show how to query the hardware environment and to set the number of threads to use:

```
// Maximum # of threads in current team, here == 1 , master thread
std::cout << "Maximum number of threads: " << omp_get_max_threads();
std::cout << "Number of available processors: " << omp_get_num_procs();

// Set the number of threads to use
omp_set_num_threads(4);      // Number of threads = 4
```

We now give three solutions to computing sum and product:

- Using *shared variables* in combination with a *critical section* (which ensures that only one thread can update them). This is a very inefficient solution.
- Using *reduction variables*. This is an efficient solution.
- A multithreaded solution without any locks on the variables. This is an efficient solution but it gives the wrong answers because a data race occurs.

The code corresponding to these three cases is:

```
// Using reduction-like variables
double sum = 0.0;
double product = 1.0;

long long N = 4;
std::vector<double>v(N, 0.0);
for (std::size_t j = 0; j < v.size(); ++j)
       { v[j] = 1.0 / static_cast<double>(j + 1); }

#pragma omp parallel for shared(N,v, sum, product)
for (long long j = 0; j < v.size(); ++j)
{

   #pragma omp critical
   { // Slow but thread-safe

      sum +=v[j];
      product *=v[j];
   }
}

std::cout << "Reduction, I: " << sum << ", " << product << '\n';

sum = 0.0;
product = 1.0;
#pragma omp parallel for reduction(+: sum), reduction(*: product)
```

```
for (long long j = 0; j < v.size(); ++j)
{
   sum +=v[j];
   product *=v[j];
}

std::cout << "Reduction, II: " << sum << ", " << product << '\n';

// Non thread-safe
{
   double sum = 0.0;
   double product = 1.0;
   #pragma omp parallel for //reduction(+: sum), reduction(*: product)
   for (long long j = 0; j < v.size(); ++j)
   {
      sum += v[j];
      product *= v[j];
   }
   std::cout << "Reduction, III: " << sum << ", " << product;
}
```

C++11 does not support reduction variables. A further discussion of OpenMP is unfortunately outside the scope of this book.

28.3 SIX WAYS TO CREATE A THREAD

Having introduced `std::thread` and given some examples of how to use it to write multi-threaded code using simple thread functions (stored lambda functions) we now discuss other ways to accommodate various kinds of *callable objects*, some of which were introduced in Chapters 2 and 3 of this book:

1. Thread with global/free function.
2. Thread with static functions.
3. Thread with callable type.
4. Thread with member function (no arguments).
5. Thread with member function (with arguments).
6. Thread with lambda function.

We take generic examples to show how the code works. It will then be possible to extend and generalise the code to more extensive examples and applications in computational finance. To this end, we create a function to return an instance of `std::thread` depending on a choice:

```
// Function to create a thread with different styles of functions
std::thread CreateThread(int i)
{
   // Create a thread using the requested function
   switch (i)
```

```
{
   default:
   case 1:
   {
      // Create thread with a function pointer to a
      // global function (& can be left out)
      return std::thread(&GlobalFunction);
   }

   case 2:
   {
      // Create thread with a function pointer to a
      // static function (& can be left out)
      return std::thread(&CallableClass::StaticFunction);
   }

   case 3:
   {
      // Ask the number of iterations
      std::cout<<"Number of iterations: ";
      int iterations;
      std::cin>>iterations;

      // Create thread with a callable class (has operator() )
      // CallableClass object is copied to the thread so its
      // lifetime is the same as that of the thread
      CallableClass c(iterations);
      return std::thread(c);
   }

   case 4:
   {
      // Ask the number of iterations
      std::cout << "Number of iterations: ";
      int iterations;
      std::cin >> iterations;

      // Create thread with a callable class (has operator() )
      // CallableClass object is copied to the thread so
      // its lifetime is the same as that of the thread
      CallableClass c(iterations);
      return std::thread(&CallableClass::Algorithm, &c);

   }
   case 5:
   {
      // Ask the number of iterations
      std::cout << "Number of iterations: ";
      int iterations;
      std::cin >> iterations;
```

```
            CallableClass c(iterations);
            return std::thread(&CallableClass::AlgorithmII, &c, 20);
         }
         case 6:
         {
            // Create thread with a lambda function
            return std::thread([]()
               { std::cout << "lambda called\n"; });
         }
      }
   }
```

The code for the free (global) function is:

```
// Global function called by thread
void GlobalFunction()
{
   for (int i=0; i<10; i++)
   {
      std::cout<< i
               <<" - Do something in parallel of main "<<'\n';
      std::this_thread::yield();
   }
}
```

The class `CallableClass` used in this code has the interface and body:

```
// Class that is a callable type (has operator() )
class CallableClass
{
private:
   // Number of iterations
   int m_iterations;

public:

   // Default constructor
   CallableClass()
   {
      m_iterations=10;
   }

   // Constructor with number of iterations
   CallableClass(int iterations)
   {
      m_iterations=iterations;
   }

   // Copy constructor
   CallableClass(const CallableClass& source)
```

```
{
   m_iterations=source.m_iterations;
}

// Destructor
~CallableClass()
{
}

// Assignment operator
CallableClass& operator = (const CallableClass& source)
{
   m_iterations=source.m_iterations;
   return *this;
}

// Static function called by thread
static void StaticFunction()
{
   for (int i=0; i<10; i++)
   {
      std::cout<<i<<
         " - Do something in parallel static"<<'\n';
         std::this_thread::yield();
   }
}

// Operator() called by the thread
void operator () ()

   for (int i=0; i<m_iterations; i++)
   {
      std::cout<<i<<"Do with main method ()"<<'\n';
      std::this_thread::yield();
   }
}

void Algorithm ()
{
   for (int i = 0; i<m_iterations; i++)
   {
      std::cout << i << " - Cooperative processing";
      std::this_thread::yield();
   }
}

void AlgorithmII(int count)
{
   for (int i = 0; i<count; i++)
   {
      std::cout << i << "Algorithm with parameter, value " << count;
```

```
            std::this_thread::yield();
        }
    }

};
```

A simple test program is:

```
// Entry point of the program.
int main()
{
    // Display the number of processors/cores
    std::cout<<std::thread::hardware_concurrency()
             <<" processors/cores detected."<<'\n'<<'\n';

    int choice;

    // Do till exit
    do
    {
        // Display menu
        std::cout<<"1. Thread with global/free function"<<'\n';
        std::cout<<"2. Thread with static functions"<<'\n';
        std::cout<<"3. Thread with callable type"<<'\n';
        std::cout <<"4.Thread with member func(no args)"<< '\n';
        std::cout << "5. Thread with member func(+ arg)" << '\n';
        std::cout << "6. Thread with lambda function" << '\n';
        std::cout<<"0. Exit"<<'\n';
        std::cout<<"Your choice: ";

        // Read input
        std::cin>>choice;

        if (choice>0 && choice<7)
        {
            // Create and start thread
            std::thread t(CreateThread(choice));

            // Do something
            for (int i=0; i<10; i++)
            {
                std::cout<<i<<" - Do something in main."<<'\n';
                std::this_thread::yield();
            }

            // Wait till thread is finished
            t.join();
        }

        // Empty line
        std::cout<<'\n';
```

```
    }
    while (choice!=0);   // Exit if 0 was chosen

    return 0;
}
```

The added value of class `CallableClass` is that it can be used as an *exemplar* for more complicated code that you create and that will run in multithreaded mode. It can function as a reminder and refresher on precise syntax details.

28.3.1 Detaching a Thread

We have seen from the above examples that the calling thread must wait on its child thread by invoking a `join()` member function. There is an option, however, whereby a child thread can be detached from the calling thread. It separates the thread of execution from the thread object, allowing execution to continue independently of the parent. Any allocated resources will be freed once the thread exits. An example of use is:

```
// Detached thread;  independent execution
std::size_t n = 1'00;
std::vector<double> v(n);
std::iota(std::begin(v), std::end(v), 1);

{
   auto EvenOdd = [&]()
   {
      auto Even = [&](int n) { return n % 2 == 0; };
      auto Odd = [&](int n) { return !Even(n); };
      std::cout << "Number of even values: "
                << std::count_if(std::begin(v), std::end(v), Even);
      std::cout << "Number of odd values: "
                << std::count_if(std::begin(v), std::end(v), Odd);
   };
   std::thread t3(EvenOdd); t3.detach();

   // Put a break in order to view output
   std::cout << "Press a key ..";
   int t; std::cin >> t;
}
```

In a sense, we 'fire and forget' detached threads. The calling thread does not have to wait on detached threads. The usage is similar to a command being executed on an object.

28.3.2 Cooperative Tasking with Threads

Multithreading is *pre-emptive* in the sense that the scheduler allocates a fixed amount of time (a *quantum*) to each thread, after which time the thread reverts to *sleep* or *wait-to-join* mode. For example, a thread can sleep for some random time, for example:

```
// Wait a while
std::default_random_engine dre(42);
std::uniform_int_distribution<int> delay(0, 1000);
int duration = delay(dre);
std::this_thread::sleep_for(std::chrono::milliseconds(duration));
```

This feature is useful in applications in which an operation (such as a backup or checking the availability of a resource) is executed by a thread every so often and then goes back to sleep, thus freeing up resources. Another case is when a thread gives other threads a chance to run. In other words, the thread relinquishes control of the CPU. A code snippet is:

```
std::this_thread::yield();   // Allow other threads to run
```

This feature is useful in applications in which a possibly long and compute-intensive operation is temporarily suspended in order to give other threads a chance to run. This improves the general *responsiveness* of programs that use multithreading, for example computing the powers of a number or of a matrix, as the following snippet of code shows:

```
// Calculate m^n. Supposes n>=0.
void operator () ()
{
   result=m;                        // Start with m^1
   for (int i=1; i<n; i++)
   {
      result*=m;                    // result=result*m
      std::this_thread::yield();    // Allow other threads to run
   }
   if (n==0) result=1;              // m^0 is always 1
}
```

The full context and code is to be found in the source code provided by the author.

Summarising: we use `yield()` to give other threads a chance to run or get the processor back as soon as possible while we use `sleep()` to give other threads a chance to run, when we do not need the processor for a period of time or when we perform some work at regular intervals.

28.4 INTERMEZZO: PARALLELISING THE BINOMIAL METHOD

Having given an introduction to threads in C++ we now discuss how to use them to improve the speedup of compute-intensive applications. To this end, we discuss how to parallelise the binomial method (see Chapters 11 and 12) in a particular case as discussed in Section 11.7 in which we price four options (call and put, European and American variants) using the binomial method. We create a single lattice that holds the underlying (stock) data and a lattice that holds option data, one lattice for each option type. Thus, for the purposes of this test we will have five lattices in memory. We are not yet interested in creating a generic framework, nor in the most memory-efficient solution, but we wish to show how to use threads to improve program speedup. We discuss the following solutions and we can compare their run-time performance and accuracy.

We now describe the steps and code for the proposed solution. We first define option-related data and a number of time steps (we take $N = 7000$ for stress-testing purposes):

```
// Option data
OptionData opt;

opt.K = 65.0;
opt.T = 0.25;
opt.r = 0.08;
opt.q = 0.0;
opt.sig = 0.3;

int N = 7000;
double dt = opt.T / static_cast<double>(N);
std::cout << "Stress test, number of time steps: " << N << '\n';
```

Next we build the *underlying lattice* of stock prices using one of the popular ways of choosing the binomial parameters (as discussed in Exercise 7 in Chapter 11, in Haug, 2007 and in Clewlow and Strickland, 1998), for example: Cox–Rubinstein–Ross, Tian, Leisen–Reimer, Rendleman Bartter, Jarrow–Rudd and Trigeorgis:

```
// Choice of latice parameters
// LatticeMechanisms::CRRLatticeAlgorithms algo(opt, dt);
LatticeMechanisms::TianLatticeAlgorithms algo(opt, dt);
// LatticeMechanisms::JRLatticeAlgorithms algorithm(opt, dt);
// LatticeMechanisms::ChangPalmerLatticeAlgorithms
//                    algorithm(opt,dt,rootVal,100);
// LatticeMechanisms::RandomWalkLatticeAlgorithms algorithm(opt, dt);

// Create basic asset lattice
Lattice<double,2> asset(N, 0.0); // init
double rootVal = 60.0;      // S(0)
LatticeMechanisms::ForwardInduction<double>(asset, algo, rootVal);
```

We support put and call options (see Exercise 9 for other kinds of payoffs):

```
// Kinds of payoff as lambda functions
double K = opt.K;
auto PutPayoff = [&K] (double S)-> double
          {return std::max<double>(K - S, 0.0);};
auto CallPayoff = [&K] (double S)-> double
          {return std::max<double>(S - K, 0.0);};

// American early exercise constraint
auto AmericanPutAdjuster = [&PutPayoff](double& V, double S)->void
{ // e.g. early exercise

   V = std::max<double>(V, PutPayoff(S));
};

auto AmericanCallAdjuster = [&CallPayoff](double& V, double S)->void
{ // e.g. early exercise
```

```
    V = std::max<double>(V, CallPayoff(S));
};

auto EmptyAdjuster = [](double& V, double S)->void
{ // No action executed at a node

    // Do nothing, no code generated in client code
};
```

We note the presence of handy lambda functions that are applied to the nodes of the lattice in the backward induction algorithm that accepts a universal function as an argument. This approach results in very flexible code.

We now create four option lattices and we price the corresponding options sequentially:

```
// Create various 'derivative lattices' and compute the option prices
// This is a kind of demultiplexing: one signal/source 'fanned' out to
// multiple option pricers.
StopWatch<> sw;
sw.Start();

Lattice<double,2> euroPut(N, 0.0);
LatticeMechanisms::ForwardInduction<double>(euroPut, algo, rootVal);
Lattice<double,2> euroCall(N, 0.0);
LatticeMechanisms::ForwardInduction<double>(euroCall, algo, rootVal);
Lattice<double,2> earlyPut(N, 0.0);
LatticeMechanisms::ForwardInduction<double>(earlyPut, algo, rootVal);
Lattice<double,2> earlyCall(N, 0.0);
LatticeMechanisms::ForwardInduction<double>(earlyCall, algo, rootVal);

// Compute option prices
using namespace LatticeMechanisms;
double euroPutPrice = BackwardInduction<double>
          (asset, euroPut, algo, PutPayoff, EmptyAdjuster);
std::cout << "Euro put: " << euroPutPrice << std::endl;

double euroCallPrice = BackwardInduction<double>
          (asset, euroCall, algo, CallPayoff, EmptyAdjuster);
std::cout << "Euro call: " << euroCallPrice << std::endl;

// Price an early exercise option
double earlyPutPrice = BackwardInduction<double>
       (asset, earlyPut, algo, PutPayoff,AmericanPutAdjuster);
std::cout << "Early put: " << earlyPutPrice << std::endl;
double earlyCallPrice = BackwardInduction<double>
       (asset, earlyCall, algo, CallPayoff,AmericanCallAdjuster);
std::cout << "Early call: " << earlyCallPrice << std::endl;

sw.Stop();
std::cout<<"Elapsed time (seconds) sequential code:"<<sw.GetTime();
```

We now generalise and improve this code by encapsulating the above functionality with lambda functions which will be used in parallel code:

```
auto fn1 = [&]()
{
   LatticeMechanisms::ForwardInduction<double>
            (euroPut, algo, rootVal);
   std::cout << "Euro put: " << BackwardInduction<double>
      (asset, euroPut, algo, PutPayoff, EmptyAdjuster) << '\n';
};


auto fn2 = [&]()
{
   LatticeMechanisms::ForwardInduction<double>
            (euroCall, algo, rootVal);
   std::cout << "Euro call: " << BackwardInduction<double>
      (asset, euroCall, algo, CallPayoff, EmptyAdjuster) << '\n';
};

auto fn3 = [&]()
{
   LatticeMechanisms::ForwardInduction<double>
            (earlyPut, algo, rootVal);
   std::cout << "Early put: " << BackwardInduction<double>
      (asset, earlyPut, algo, PutPayoff, AmericanPutAdjuster);
};

auto fn4 = [&]()
{
   LatticeMechanisms::ForwardInduction<double>
            (earlyCall, algo, rootVal);
   std::cout << "Early call: " << BackwardInduction<double>
      (asset, earlyCall, algo, CallPayoff, AmericanCallAdjuster);
};
```

We now discuss a number of ways to parallelise the above code. Each solution is based on the *fork-and-join* model in which a master thread creates a team of *child threads*, gives them work to do and waits for them to finish:

- S1: Using C++ threads.
- S2: Using an array of C++ threads.
- S3: Using Boost *thread group*.
- S4: Using OpenMP *parallel loops*.
- S5: Using C++ *futures* (*asynchronous tasks*) (to be discussed in Chapter 29).

The solutions in this case are easy to program because the shared data is read-only and each thread operates on its own *thread-local data*. Thus, in this case we do not have to employ *locking mechanisms* in order to avoid data races which will have an adverse impact on run-time performance. We do not need to lock shared data in the current case.

Solution S1 is a straightforward application of C++ threads:

```
sw.Start();
std::cout << '\n';

std::thread t1(fn1);
std::thread t2(fn2);
std::thread t3(fn3);
std::thread t4(fn4);

// No shared data so we define 1 barrier/rendezvous point
t1.join();
t2.join();
t3.join();
t4.join();

sw.Stop();
std::cout << "Elapsed time C++11 threads in seconds: " << sw.GetTime();
```

Solutions S2 and S3 are variations on solution S1 by embedding threads in an array and using Boost thread groups, respectively:

```
// Thread array
sw.Start();
std::vector<std::shared_ptr<std::thread>> threadGroup;

std::vector <std::function<void()>>
   tGroupFunctions = { fn1, fn2, fn3, fn4 };

std::cout << '\n';
for (std::size_t i = 0; i < tGroupFunctions.size(); ++i)
{
   threadGroup.emplace_back(std::shared_ptr<std::thread>
         (new std::thread(tGroupFunctions[i])));
}
for (std::size_t i = 0; i < tGroupFunctions.size(); ++i)
{
   if (threadGroup[i]->joinable())
   {
      threadGroup[i]->join();
   }
}
sw.Stop();
std::cout << "Elapsed time C++ thread array in seconds: "
           << sw.GetTime() << '\n';
```

and

```
// Boost thread group
sw.Start();
std::cout << '\n';
```

```
boost::thread_group threads;
for (std::size_t i = 0; i< tGroupFunctions.size(); ++i)
{
   threads.create_thread(tGroupFunctions[i]);
}
std::cout << "Thread group size: " << threads.size() << '\n';

threads.join_all();

sw.Stop();
std::cout <<"Elapsed time BOOST thread groups in seconds:"
          << sw.GetTime() << '\n';
```

Solution S4 has the same form as solution S2 but in this case we do not have to explicitly create threads. In this case we use a *pragma* to announce that a group of threads should be created. Each thread performs a subset of the work. There is a default *barrier* after the loop body where each thread waits for all the other threads to arrive. Only when all threads have arrived at this barrier does the code continue execution (the *barrier directive*). In other words, a barrier synchronises all the threads executing within the *parallel region*:

```
sw.Start();
std::cout << '\n';
#pragma omp parallel for
for (long i = 0; i < tGroupFunctions.size(); ++i)
{
   tGroupFunctions[i]();
}
sw.Stop();
std::cout << "Elapsed time OpenMP thread groups in seconds: "
          << sw.GetTime() << '\n';
```

Again, race conditions will not occur because each loop iteration is executed by a single thread and furthermore, each thread operates on its own *thread-local data*.

Finally, we give a preview of how C++ supports multitasking and allows developers to create parallel code using higher-level syntax (solution S5). In particular, we run *asynchronous tasks* and we wait for them to finish:

```
sw.Start();
std::cout << '\n';
// Start future by running the functions asynchronously
std::future<void> fut1(std::async(fn1));
std::future<void> fut2(std::async(fn2));
std::future<void> fut3(std::async(fn3));
std::future<void> fut4(std::async(fn4));

// Wait for results to become available
fut1.wait();
fut2.wait();
fut3.wait();
fut4.wait();
```

```
// No shared data so we define 1 barrier/rendevouz point
fut1.get();
fut2.get();
fut3.get();
fut4.get();

sw.Stop();
std::cout << "Elapsed time C++11 futures in seconds: "<< sw.GetTime();
```

Finally, the output from the above code is:

```
Stress test, number of time steps: 7000
Interest: 0.08
Vol: 0.3
Strike: 65
Expiration: 0.25
Dividend: 0
4 processors/cores detected.

Euro put: 5.8464
Euro call: 2.13349
Early put: 6.12361
Early call: 2.13349
Elapsed time (seconds) sequential code: 1.68254

Euro call: 2.13349
Early call: 2.13349
Euro put: 5.8464
Early put: 6.12361
Elapsed time C++11 threads in seconds: 0.714551

Euro call: 2.13349
Early call: 2.13349
Euro put: 5.8464
Early put: 6.12361
Elapsed time C++ thread array in seconds: 0.714546

Thread group size: 4
Euro call: 2.13349
Early call: 2.13349
Euro put: 5.8464
Early put: 6.12361
Elapsed time BOOST thread groups in seconds: 0.719553

Euro call: 2.13349
Early call: 2.13349
Euro put: 5.8464
Early put: 6.12361
Elapsed time OpenMP thread groups in seconds: 0.702057
```

```
Euro call: 2.13349
Early call: 2.13349
Euro put: 5.8464
Early put: 6.12361
Elapsed time C++11 futures in seconds: 0.707558
```

We formally discuss multitasking in Chapters 29 and 30. The code in this section can be used as a template (or *cooky cutter*) for other similar kinds of applications that we would like to parallelise, as we shall see in Section 32.2 when we compute option prices using the Monte Carlo method.

28.5 ATOMICS

One of the most far-reaching features in C++11 is that it supports a new *multithreading-aware memory model* (see Williams, 2012 for a detailed discussion). We do not go into details just yet but instead introduce a number of related concepts. An *atomic operation* is an indivisible operation. It cannot be half-done; it must be executed with success or not done at all. In database applications it is similar to a *transaction* that executes completely or not at all. In the current chapter we wish to ensure that only one thread can access an object, resource or piece of memory at any moment in time. Not all operations are atomic. For example, an assignment statement of 64-bit variables may not be atomic because one thread may be updating the high-order 32-bit part while another thread can be simultaneously updating the low-order 32-bit part. In general, an atomic operation at the hardware level is uninterruptible and it must be guaranteed to terminate, for example it can never get stuck in an infinite loop.

An *atomic type* is one for which all atomic operations are atomic and only operations on these types are atomic in the sense of the language. Atomic types are supported in C++ by the template struct `std::atomic`. It has support for standard types such as `char`, `int`, `long` (both signed and unsigned versions) as well as *fast integer types* for better performance. The member functions are:

- Constructors: default constructor and constructor taking a value (latter is not atomic).
- Is the atomic object *lock-free* (can more than one thread access the object concurrently)? This allows the developer to determine whether operations on a given type are executed directly with atomic instructions or executed by using a lock internal to the compiler and library.
- Store: atomically replace the value of the atomic object with a non-atomic argument.
- Load: atomically obtain the value of the atomic object.
- Load a value from an atomic object.
- Exchange: atomically replace the values of the atomic objects and obtain the value held previously.

Furthermore, there are specialised member functions for atomic addition and subtraction as well as for bitwise AND, OR and XOR. Finally, there are operators for atomic increment and decrement.

We now give some examples of using `std::atomic` based on the above discussion:

```
#include <atomic>
#include <iostream>

int main()
{
   // Full specification std:atomic<long long>
   std::atomic_llong v1(std::numeric_limits<long long>::max());
   std::atomic_llong v2;

   // Lock-free ADT more than 1 thread can access it concurrently
   std::cout << "Lock-free? " << std::boolalpha
             << v1.is_lock_free();

   std::cout << "Value v1, v2: "<<v1 << ", " <<v2; // max, random value

   // Atomically replace the value of v2 by the value of v1
   v2.store(v1);
   std::cout << "Value v1, v2: "<<v1<<", " << v2; // max, max

   // Automatically load and return current value
   std::cout << "Current values v1, v2: " << v1.load() << ", "
             << v2.load() << std::endl; // max, max

   v2.store(0);
   std::cout << "Exchanged: " << v1.exchange(v2)
             << ", " << v2 << std::endl; // max, 0
   std::cout << "Current values v1, v2: " << v1.load() << ", "
             << v2.load() << std::endl; // 0, 0

   // Increment/decrement
   std::atomic_int_fast16_t v3(1);
   //std::atomic_int_fast16_t v4(v3); NO

   std::atomic_int_fast16_t v4;
   v4.store(10);

   std::cout << "Current values v3, v4: " << v3.load() << ", "
             << v4.load() << std::endl; // 1, 10

   v3++; v4--;
   std::cout << "Current values v3, v4: " << v3.load() << ", "
             << v4.load() << std::endl; // 2, 9

   // Addition and substration
   std::atomic<long long> v5(0);
   std::atomic<int> v6(10);

   v5.fetch_add(10); v6.fetch_sub(20);
   std::cout << "Current values v5, v6: " << v5.load() << ", "
             << v6.load() << std::endl; // 10, -10
```

```
    // Bitwise operations
    std::atomic<int> v7(0);
    std::cout << "Current values v7: " << v7.load() << '\n'; // 0

    v7 += 20; v7 -= 30; v7 &= 40; v7 ^= 10;
    std::cout << "Current values v7: " << v7.load() << '\n'; // 0

    return 0;
}
```

We recommend that you study and run this code.

28.5.1 The C++ Memory Model

Various memory models have been proposed for shared-memory parallel programming, but the central issue from the viewpoint of the programmer is that new values of shared data become available to those threads that did not perform the update. The writing thread should either write new values back into memory or invalidate other copies of data to enable their update. It is clear that we need to regulate the order in which updates should be performed. In general, we are interested in *correctness* and there are several related conditions, for example *sequential consistency*:

> *A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order and the operations of each individual processor appear in this sequence in the order specified by its program.*

This model requires that values of shared data be *uniformly consistent* among the executing threads. In other words, if one thread creates a new value and this value is used immediately in a subsequent statement, then the thread executing that statement must use the new value. The consequences are:

- Good for the programmer who does not have to worry about coordinating access to data.
- Bad for performance: memory update after each operation that modifies a shared variable and potentially before each use of a shared variable.
- Other *relaxed consistency models* have been proposed (better performance, programming is harder).

In C++, the enum `std::memory_order` specifies how regular, non-atomic memory accesses are to be ordered around an atomic operation. The default ordering in the library provides for *sequentially consistent* ordering. This default can affect performance but it is possible to specify the exact constraints. Without any constraints the values are non-deterministic (we do not discuss `memory_order_consume` or `memory_order_acq_rel`):

- `memory_order_relaxed`: no synchronisation or ordering constraints; atomicity is required.
- `memory_order_acquire`: load operation; ensures writes in other threads are visible in current thread.

- `memory_order_release`: store operation; all writes in current thread are visible in other threads.
- `memory_order_seq_cst`: single *total order* exists in which all threads observe all modifications in the same order.

We take a simple example in which we create a number of threads that increment a counter atomically. We use a sequential model so that all threads see the modifications:

```
#include <vector>
#include <iostream>
#include <thread>
#include <atomic>
#include <chrono>
#include <ctime>

std::atomic<long long> cnt = { 0 };

void f()
{
   int arg = 10;
   int repetitions = 10000;
   // No synchronisation or ordering constraints; atomicity required
   for (int n = 0; n < repetitions; ++n)
   {
      // increment cnt by arg
      cnt.fetch_add(arg, std::memory_order_relaxed);

   }
}

void f1()
{
   int repetitions = 10;
   // No synchronisation or ordering constraints; atomicity required
   for (int n = 0; n < repetitions; ++n)
   {
      cnt++;
   }
}

void f2()
{
   int arg = 1;
   int repetitions = 1000'000'0;
   for (int n = 0; n < repetitions; ++n)
   {
   // cnt.fetch_sub(arg, std::memory_order_relaxed);
      cnt.fetch_sub(arg, std::memory_order_seq_cst);

   }
```

```
}

void f3()
{
   int arg = 10;
   int repetitions = 10000;
   for (int n = 0; n < repetitions; ++n)
   {
      cnt += arg;
   }
}
```

We take an example in the case of function `f21`;

```
int main()
{

   std::chrono::time_point<std::chrono::system_clock> start, end;
   start = std::chrono::system_clock::now();

   int numThreads = 30;
   std::vector<std::thread> v;
   for (int n = 0; n < numThreads; ++n)
   {
      v.emplace_back(f2);
   }

   for (auto& t : v) {t.join();}

   end = std::chrono::system_clock::now();

   std::chrono::duration<double> elapsed_seconds = end - start;
   std::time_t end_time = std::chrono::system_clock::to_time_t(end);

   std::cout << "finished computation at " << std::ctime(&end_time)
             << "elapsed time: " << elapsed_seconds.count();

   std::cout << "Final counter value is " << cnt << '\n';
}
```

28.5.2 Atomic Flags

The class `std::atomic_flag` is an atomic Boolean type. It is guaranteed to be lock free but it does not provide load or store operations. This class is useful when locking is needed, for example when implementing *spinlock mutexes*. It has the following member functions:

- Default constructor: initialises the class to an unspecified state. It is also possible to initialise the class to a clear (false) state:

```
std::atomic_flag ready = ATOMIC_FLAG_INIT;
```

- The assignment operator is not supported (it is *deleted*).
- Clear: change the state of an atomic flag to `false`.
- Atomically change the state of an atomic flag to *set* (`true`) and return the value it held before.

We give a simple example of creating an array of threads and creating spinlock code. The main method is:

```
int main()
{
   using ThreadGroup = std::vector<std::thread>;
   ThreadGroup v;

   // Append a new element to the end of the container
   for (int n = 0; n < 10; ++n){ v.emplace_back(func, n);}

   // Join all threads
   for (auto& t : v) {t.join();}


   std::cout << "All threads finished, thread number and data: "
             << this_id << ", " << data << std::endl;

   return 0;
```

while the thread function `func()` is:

```
#include <thread>
#include <vector>
#include <atomic>
#include <iostream>

// Common data structure
double data = 0.0;

// Put flag in clear state
std::atomic_flag lock = ATOMIC_FLAG_INIT;

std::thread::id this_id;

void func(int n)
{ // Locking using atomic flag

   this_id = std::this_thread::get_id();

   for (int i = 0; i < 1; ++i)
   {
      while (lock.test_and_set(std::memory_order_acquire))
      {// all *writes* from threads visible in current thread
```

```
            std::cout << "\nAcquiring lock on thread "
                      << n << std::endl;
            data = static_cast<double>(n);

        };
        std::cout << "Thread number, lock released "
                  << n << std::endl;

        // Atomically changes the state of atomic flag to
        // clear (false)store ensures all *writes* in
        // current thread are visible in other threads
        lock.clear(std::memory_order_release);
    }
}
```

Sample output is:

```
Thread number, lock released 0

Acquiring lock on thread 1
Thread number, lock released 1
Thread number, lock released 2
Thread number, lock released 3
Thread number, lock released 4
Thread number, lock released 5
Thread number, lock released 6
Thread number, lock released 7
Thread number, lock released 8
Thread number, lock released 9
All threads finished, thread number and data: 7384, 1
```

28.5.3 Simple Producer-Consumer Example

We give an example of how to use atomics to create a simple example based on the *Producer–Consumer pattern*. In this case both the producer and the consumer run in their own threads. The single producer delivers data to consumers (which wait on the producer). When the user presses the *Enter* key the atomic Boolean is set to `true` and the consumers can process the data. Then the program ends.

The shared data, atomic Boolean as well as the code for the producer and consumer entities are:

```
// Common data structure
double data;
std::atomic<bool> ready(false);

void Producer()
{
    // Wait until user types a character
    std::cout << "Type <return> ";  std::cin.get();
```

```
   // Set the data and set the boolean to true (i.e. ready)
   data = 3.1415;

   ready.store(true);
}


void Consumer(int n)
{
   // Hang around wait waiting for the data to be ready
   while (!ready.load())
   {
      std::cout.put('.').flush();

      // Go back to sleep
      std::this_thread::sleep_for
         (std::chrono::milliseconds(100));
   }

   std::cout << "\nValue of data is from consumer: "
             << n << ", " << data << std::endl;
}
```

A test program using threads and C++ tasks (as discussed in Chapter 29) is:

```
#include <thread>
#include <future>
#include <atomic>
#include <chrono>
#include <iostream>


int main()
{
   std::thread producer (Producer);
   std::thread consumer1 (Consumer, 1);
   std::thread consumer2(Consumer, 2);

   producer.join();
   consumer1.join();
   consumer2.join();

   std::cout << "\n\nSolution using tasks\n";
   // Same using tasks and async()
   auto p = std::async(std::launch::async, Producer);
   auto c1 = std::async(std::launch::async, Consumer, 1);
   auto c2 = std::async(std::launch::async, Consumer, 2);

   return 0;
}
```

This is useful code because it can be generalised to more complex applications. The main advantage is that we use atomics as a notification mechanism.

28.6 SMART POINTERS AND THE THREAD-SAFE POINTER INTERFACE

Shared pointers are not thread safe in general. Reading a pointer's use-count by one thread while another thread modifies the data does not cause a race condition; however, the value may not be up to date. Data races occur if multiple threads access a non-constant member function of `std::shared_ptr<T>`. This is an undesirable feature and for this reason we use overloads of the corresponding access functions for load, store and exchange, called the *high-level interface*. We take an example to show how these functions work. We examine the following class and define aliases for readability reasons:

```
struct X
{
   double val;

   X() : val(0.0) {}
   void operator ()()
   {
      std::cout << "An X " << val << std::endl;
   }
};


template <typename T>
   using GenericPointerType = std::shared_ptr<T>;

using PointerType = GenericPointerType<X>;
```

The next step is to create smarter points of instance of struct `X`:

```
// Two shared ptrs s and s2
PointerType s(new X);
s->val = 3.14;
std::cout << "Use count s:" << s.use_count() << '\n';

// False
std::cout << "Atomic interface is lock free " << std::boolalpha
          << std::atomic_is_lock_free(&s) << std::endl;

PointerType s2 = nullptr;
std::atomic_store(&s2, s);
std::cout << "Use count s2,s: " << s2.use_count() << ","
          << s.use_count() << '\n';
std::cout << "Value in stored pointer s2: "
          << s2->val << std::endl; // 3.14
```

```
s->val = 2.71;
std::cout << "Value in stored pointer s2 now: "
          << s2->val << std::endl; // 2.71

// Load s2 into memory
std::atomic_load(&s2);
std::cout << "s2 val loaded " << s2->val << std::endl;

// Exchange s2 and s and return *previous* value of s2
s->val = 148.333;
std::atomic_exchange(&s2, s);
std::cout << "s2 val after exchange: " << s2->val << std::endl;

// Load s2 and s
std::atomic_load(&s);
std::atomic_load(&s2);
std::cout << "Use count s2,s: " << s2.use_count()
          << "," << s.use_count() << '\n';
std::cout << "Values s2,s: " << s2->val << "," << s->val << '\n';

// Now s3
PointerType s3(new X);
s3->val = 42.0;
std::atomic_store(&s, s3);
std::atomic_store(&s2, s3);
std::cout << "Use count s2,s from s3: " << s2.use_count()
          << "," << s.use_count() << '\n';
std::cout << "Values s2,s from s3: " << s2->val
          << "," << s->val << '\n';

// End Two shared ptrs s and s2

PointerType p1(new X);
PointerType p2(new X); p2->val = 42.0;

std::cout << "p1 val " << p1->val << std::endl;          // 0
std::cout << "p2 val " << p2->val << std::endl;          // 42
auto former = std::atomic_exchange(&p1, p2);
std::cout << "former " << former->val << std::endl;      // 0

std::cout << "p1 val " << p1->val << std::endl;          // 42
std::cout << "p2 val " << p2->val << std::endl;          // 42

std::atomic_load(&p2);
std::cout << "p2 val loaded " << p2->val << std::endl;   // 42

p2 = std::atomic_exchange(&p1, p2);
std::cout << "p2 val " << p2->val << std::endl;          // 42
```

Finally, we give an example of non-deterministic behaviour in which several threads modify shared data. We get a different result each time that we run the program because there is no predefined order in which the threads are fired and executed:

```
std::mutex displayMutex;

void Modify(PointerType& p, double newVal)
{
   // Wait a while, long enough to trigger the race
   std::default_random_engine dre(42);
   std::uniform_int_distribution<int> delay(0, 10000);
   std::this_thread::sleep_for
   (std::chrono::milliseconds(delay(dre)));

   p->val = newVal;

   displayMutex.lock();
   std::cout << "Thread ID, std::thread::id "
             << std::this_thread::get_id() << std::endl;
   displayMutex.unlock();
}

std::cout << "Smart races\n";
PointerType x(new X);
x->val = 2.71;

double d1 = 1.0;
double d2 = 2.0;
double d3 = 3.0;
double d4 = 4.0;


// New, easier syntax
std::thread t4(Modify, x, d1);
std::thread t2(Modify, x, d2);
std::thread t3(Modify, x, d3);
std::thread t1(Modify, x, d4);

t1.join(); t2.join();
t3.join(); t4.join();

std::cout << "Value: " << x->val << std::endl;
```

28.7 THREAD SYNCHRONISATION

One of the attention points when writing multithreaded code is to determine how to organise threads in such a way that access to shared data is done in a controlled manner. This is because the order in which threads access data is non-deterministic and this can lead to inconsistent results called *race conditions.* A classic example is when two threads attempt to withdraw

TABLE 28.1 Thread synchronisation

Thread 1	Thread 2	balance
if (70>balance)		100
	if (90>balance)	100
balance−=70		30
	balance−=90	−60

funds from an account at the same time. The steps in a sequential program to perform this transaction are:

1. Check the balance (are there enough funds in the account?).
2. Give the amount of money to be withdrawn.
3. Commit the transaction and update the account.

When there are two threads involved then steps 1, 2 and 3 may be *interleaved*, which means that the threads can update data in a non-deterministic way. For example, the scenario in Table 28.1 shows that after withdrawing 70 and 90 money units the balance is −60 money units, which destroys the *invariant condition* that states, in this case, that the balance may never become negative. Why did this transaction go wrong?

The solution is to ensure that steps 1, 2 and 3 constitute an *atomic transaction*, by which we mean that they are locked by a single thread at any one moment in time. *Boost.Thread* has a number of classes for thread synchronisation and C++11 has similar constructs (the code is the same; we just have to use namespace *std* instead of namespace *boost*). The first class is called `mutex` (*mutual exclusion*) and it allows us to define a lock on a code block and release the lock when the thread has finished executing the code block. To do this, we create an `Account` class containing an embedded mutex:

```
class Account
{
private:

    // The mutex to synchronise on
    boost::mutex m_mutex;

    // more...
};
```

We now give the code for withdrawing funds from an account. Notice the *thread-unsafe version* (which can lead to race conditions) and the *thread-safe version* using `mutex`:

```
// Withdraw an amount (not synchronised). Scary!
void Withdraw(int amount)
{
   if (m_balance-amount>=0)
   {
         // For testing we now give other threads a chance to run
         boost::this_thread::sleep(boost::posix_time::seconds(1));
```

```
            m_balance-=amount;
        }
        else throw NoFundsException();
    }

    // Withdraw an amount (locking using mutex object)
    void WithdrawSynchronized(int amount)
    {
        // Acquire lock on mutex.
        // If lock already locked, it waits till unlocked
        m_mutex.lock();

        if (m_balance-amount>=0)
        {
           // For testing we now give other threads a chance to run
           boost::this_thread::sleep(boost::posix_time::seconds(1));

           m_balance-=amount;
        }
        else
        {
           // Release lock on mutex. Forget this and it will hang
           m_mutex.unlock();
           throw NoFundsException();
        }

        // Release lock on mutex. Forget this and it will hang
        m_mutex.unlock();
    }
```

Only one thread has the lock at any time. If another thread tries to lock a mutex that is already locked it will enter the *SleepWaitJoin* state until the lock is released by the original thread. Summarising, only one thread can hold a lock on a mutex and the code following the call to `mutex.lock()` can only be executed by one thread at a given time.

A major disadvantage of using mutex is that the system will *deadlock* ('hang') if you forget to call `mutex.unlock()`. For this reason we use the `unique_lock<Lockable>` adapter class that locks a mutex in its constructor and unlocks a `mutex` in its destructor. The new version of the withdraw member function will be:

```
// Withdraw an amount (locking using unique_lock)
void WithdrawSynchronized2(int amount)
{
     // Acquire lock on mutex. Will be automatically unlocked
     // when lock is destroyed at the end of the function
     boost::unique_lock<boost::mutex> lock(m_mutex);
     if (m_balance-amount>=0)
     {
        // For testing we now give other threads a change to run
        boost::this_thread::sleep(boost::posix_time::seconds(1));
```

```
            m_balance-=amount;
        }
        else throw NoFundsException();
    }    // Mutex automatically unlocked here
```

Note that it is not necessary to explicitly unlock the mutex in this case.

28.8 WHEN SHOULD WE USE THREADS?

Designing parallel applications that use C++ threads can result in code with good run-time performance but in general reaping the potential benefits can come at a price. Writing correct and efficient multithreaded C++ code is very difficult (even for experienced programmers). Some of the challenges (in no particular order) are:

- Debugging multithreaded code is difficult because of the presence of bugs, many of which are difficult to reproduce. More perniciously, the semantics of a parallel program may be fundamentally different to its sequential equivalent. In other words, realising *sequential consistency* is a major challenge. Other *troubleshooting* issues are discussed in Chapman, Jost and Van der Pas (2008).
- Using traditional *object-oriented technology* (OOT) with C++ threads is somewhat of a mismatch. Objects encapsulate data and in many cases we may wish to share this data between threads. Of course, we can lock data by using mutexes but this will have an impact on the efficiency and reliability of the resulting code. In particular, migrating legacy C++ libraries to a multithreaded version will be a non-trivial task and maybe even impossible.
- Learning how to create and use C+ threads is an important skill. Even if we do not use threads directly, it is still necessary to understand multithreading foundations. This approach, combined with a project to learn OpenMP, is a good way to get hands-on experience.
- In Chapters 29 and 30 we shall introduce an alternative approach to multithreading. We decompose a system into concurrent tasks. For many applications in computational finance this is a good approach and we shall give several examples for the Monte Carlo method.

28.9 SUMMARY AND CONCLUSIONS

In this chapter we gave an introduction to the *C++ Concurrency library*. We discussed what a thread is and how it can be created by providing it with a *callable object* which plays the role of the algorithm or *thread function*. In general, we decompose a problem into subproblems and then assign each subproblem to a specific thread. We must realise that the developer is responsible for coordinating threads, especially when they modify shared data. By default, access to data is non-deterministic and we use locking mechanisms to ensure that only one thread can modify shared data at any moment in time. A special class of objects are *atomic variables* whose operations are guaranteed to be thread safe. We note that smart pointers are not thread safe in general, but C++ provides several atomic operations for them.

As a relevant application of C++ threads, we return to the lattice option pricing models that we discussed in Chapters 11 and 12. We introduced a number of use cases and we

implemented one of them using C++ threads. The underlying design can be applied to other compute-intensive applications.

The best way to learn about threading is to write multithreaded code. We recommend implementing the accompanying exercises. In Chapter 29 we introduce C++ multitasking, which is easier to learn than multithreading.

28.10 EXERCISES AND PROJECTS

1. (Creating a Thread)

We examine the code in Section 28.2. Answer the following questions:

a) What happens if your forget to call `join()` on a thread?

b) Any return value from a thread function is ignored. Is there some way to use a lambda function to somehow circumvent this problem in order to access this value? Is this a robust solution? (In Chapter 29 we shall present a structural solution to this problem.)

c) Some thread functions may loop forever or fail to terminate for some reason. Consider the following code:

```
// Thread function having input arguments
auto f2 = [](int n, std::string& s)
         { std::cout << n << " Hello world " << s; };
std::thread t2(f2, 42, std::string("and beyond\n"));
// Wait for thread to finish (mandatory)
t2.join();
```

What happens if you run this code without joining?

d) If a thread cannot be started then a `std::system_error` exception will be thrown. The exception may represent the error condition `std::errc::resource_unavailable_try_again` or another implementation-specific error condition. Can you reproduce these conditions?

e) It is possible to *swap* two threads using a free function or a member function of `std::thread`. Create code to perform thread swap. Test the code by examining the thread IDs before and after the swap.

f) We use the system console to display the results of the computations in Section 28.2. The output is garbled because two threads are concurrently writing to the console and non-deterministic behaviour can occur. Determine how to resolve this problem by the introduction of mutexes and locks.

2. (STL Algorithms, Simple Case of a *MapReduce* Pattern)

This is an example of writing multithreaded code using *C++ Concurrency* to perform compute-intensive operations on large lists L1, L2 and L3 (for example, 10 million elements each). The three lists will be modified in different ways and then finally merged. A requirement is that STL algorithms be used. The operations are:

- Sort L1 in ascending order; do a random shuffle of L1; final output is L1.
- Sort L2 in descending order; do a random shuffle of L2; final output is L2.
- Reverse the elements of L3; final output is L3.
- Merge L1, L2 and L3.

Answer the following questions:

a) Write sequential code (*single threaded*) to carry out these operations. Measure the processing time.

b) Draw a *data dependency graph* in which the tasks are STL algorithms and the arcs represent the output of each algorithm. Which tasks can be run in parallel and which tasks must be run sequentially (data dependency graphs are introduced in Chapter 29)?
c) Design the graph from part b) using two threads and then with four threads. What is the speedup?
d) If you have OpenMP experience then design the graph from part b) using OpenMP *sections*. What is the speedup?

All code should use the algorithm in Chapters 17 and 18.

3. (Quiz)

Examine the following code. What happens when you run it and how is the problem resolved?

```
auto threadfunc = []() { std::cout << "whoopsy-daisy\n"; };

std::thread t1(threadfunc);
auto t2 = std::move(t1);

t2.join();
t1.join();
```

4. Which of the following statements best describe *threads*?
a) They are independent units of execution.
b) Threads communicate by message passing.
c) Threads usually share memory.
d) A sequential program has one thread.

5. Give two reasons for writing multithreaded code:
a) It always runs faster than sequential code.
b) It is used to improve the performance of long-running algorithms.
c) It leads to code that is easier to maintain than sequential code.
d) It can make GUI applications appear to be more responsive.

6. The possible disadvantages of multithreaded code are:
a) The output may not be the same as in the serial/sequential case.
b) Unpredictable spurious results may be produced in running code.
c) A multithreaded application may perform less well than the equivalent sequential code.
d) A thread can cause another thread to abort.

7. Which of the following statements concerning C++11 threads are true?
a) A thread has a *thread function* that can be created using any *callable object*.
b) A thread starts executing as soon as it is created.
c) The return type of a thread is the same as that of its thread function.
d) A thread is deemed to have run its course when its thread function has completed processing.

8. What is *thread synchronisation* and how is it implemented in C++11?
a) It is a mechanism to allow threads to notify each other of changes in thread state.
b) It is needed in order to avoid non-deterministic *race conditions*.
c) C++11 uses *mutexes* to realise thread synchronisation.
d) C++11 uses *condition variables* to realise thread synchronisation.

9. (Parallelising Binomial with Various Payoffs)

In Section 28.4 we developed multithreaded code to price European and American put and call options. Modify the code to support the following kinds of payoff:

```
// Define variables needed by payoffs and pricers
double r = opt.r; double q = opt.q;
double T = opt.T;
double K = opt.K;
double s = opt.sig;
double S0 = 100.0;
double Q = 10.0;

// Define payoffs
double d1 = (std::log(S0/K)
          + (r -q + 0.5*s*s)*T) / (s*std::sqrt(T));
double d2 = d1 - s*std::sqrt(T);

// Cash-or-nothing options
auto CashOrNothingCallPayoff = [=] (double S)-> double
{
   if (S <= K) return 0.0;
   return Q;
};

auto CashOrNothingPutPayoff = [=] (double S)-> double
{
   if (S >= K) return 0.0;
   return Q;
};

auto CashOrNothingCallPrice = [=] (double S)-> double
{
   return Q*std::exp(-r*T)*NCdf(d2);
};

auto CashOrNothingPutPrice = [&] (double S)-> double
{
   return Q*std::exp(-r*T)*NCdf(-d2);
};

// Exercise to test this


// Asset-or-nothing options
auto AssetOrNothingCallPayoff = [=] (double S)-> double
{
   if (S <= K) return 0.0;
   return S;
};

auto AssetOrNothingPutPayoff = [=] (double S)-> double
{
   if (S >= K) return 0.0;
   return S;
};
```

```
auto AssetOrNothingCallPrice = [=] (double S)-> double
{
   return S*std::exp(-q*T)*NCdf(d1);
};

auto AssetOrNothingPutPrice = [=] (double S)-> double
{
   return S*std::exp(-q*T)*NCdf(-d1);
};


double NCdf(double x)
{ // Normal cumulative distribution

   boost::math::normal_distribution<> myNormal(0.0, 1.0);
   return boost::math::cdf(myNormal, x);
}
```

10. (Parallelising the Generalised Binomial Method)

The objective of this exercise is to parallelise the algorithms to compute call and put option prices for a range of payoffs in equations (12.5) and (12.6). In all cases we return the prices in a 2-tuple.

Answer the following questions:

a) Implement equations (12.5) and (12.6) by Bernoulli paths using sequential C++ (a single thread). Test the code for standard calls and puts as well as the following two option types:

Standard power contract:

$$g(S) = \max(K - S^p, 0), \text{ (put) } p > 0$$
$$g(S) = \max(S^p - K, 0), \text{ (call) } p > 0$$

Power contract:

$$g(S) = (S/K)^p, \ p > 0.$$

In the case of a standard power contract we use the following data: $S = 555$, $K = 550$, $r = 0.06$, volatility $= 0.15$, dividend yield $= 0.04$, $T = 0.5$. For $p = 0.96$ we get $C = 0.17614$ and for $p = 1.05$ we get $C = 213.01648$.

b) We wish to parallelise the code in part a) using C++ threads. To this end, we divide the index range $[0, n]$ in equations (12.5) and (12.6) into four subranges. We then assign each subrange to a single thread and each thread computes a part of the price. Then the partial sums are added together to give the final answers. Compare the prices with those computed in part a).

c) Compare the run-time performance of the solutions in parts a) and b).

d) (After you have read Sections 30.2.4 and 30.4 of Chapter 30.) The objective now is to apply the *parallel reduction* technique and *Aggregation/Reduction pattern* from the PPL to achieve the same result as in part b).

e) Solve the current problem using OpenMP's *reduction* clause. A simple example to motivate your solution is:

```
int N = 10000;
double vec1[N];
double vec2[N];
double sum = 0.0;
int i;
#pragma omp parallel for reduction (+:sum)
for (i = 0; i < N; ++i)
   sum += vec1[i]*vec2[i];
```

11. (Parallelising the Finite Difference Method)

In Chapters 20 to 23 we discussed how to approximate the solution of partial differential equations using the finite difference method. We proposed a number of schemes and we give some examples (see Section 21.8 for more details):

```
CNIBVP fdmCN(currentImp, earlyExercise,N, J);
ImplicitEulerIBVP fdmIE(currentImp, earlyExercise,N, J);
ExtrapolatedImplicitEulerIBVP fdmEIE (currentImp,earlyExercise,N, J);
ADE fdmADE(currentImp, earlyExercise,N, J);
ADE_BC_CRTP fdmADE2(currentImp, earlyExercise,N, J);
ADE_Larkin_RW_CRTP fdmADE3(currentImp, earlyExercise,N, J);
ADEExtrapolated fdmADE4(currentImp, earlyExercise, N, J);
```

In general, we can think of several use cases and viewpoints to take in order to test finite difference schemes for accuracy and performance:

- UC1: Compare the accuracy of a new finite difference scheme with an accurate baseline scheme or exact solution.
- UC2: Stress test a scheme for a range of input parameters to the PDE. The parameters are randomly generated using the designs that we used in Chapter 16 (in particular, Section 16.4). The focus is to determine the robustness and accuracy of a scheme for a range of input parameters.
- UC3: Compute option prices for an input data set.

We focus on questions related to case UC3.

Answer the following questions:

a) Adapt the code in Section 28.4 to test the Crank–Nicolson, ADE and MOL against each other. Each scheme will be run in its own thread.

b) Compute the option price in each case and compare the prices regarding accuracy. Concentrate on the cases where the differences are greater than a given tolerance.

c) Extend the code in parts a) and b) to the case where the input data is generated using the techniques we used in Section 16.4. Create a loop with a given number of iterations and compute the global maximum error between the solutions from all runs of random generation of the input data.

The advantage of this approach is that you can run the program in batch mode as it were, and when you come back from lunch it will give output on the accuracy of the scheme that you are testing.

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 29

C++ Concurrency, Part II Tasks

29.1 INTRODUCTION AND OBJECTIVES

In this chapter we discuss how C++ supports the creation of programs and algorithms by decomposing them into components that can potentially run in parallel. In general terms, *potential or exploitable concurrency* involves being able to structure code to permit a problem's subproblems to run on multiple processors. Each subproblem is implemented by a task. A *task* (Quinn, 2004) is a program in local memory in combination with a collection of I/O ports. Tasks send data to other tasks through their *output ports* and they receive data from other tasks through their *input ports*. It is obvious from this discussion that tasks can depend on other tasks because they need the data that is produced by other tasks. We model these dependencies by a *data dependency graph* or *task graph.* This is a directed graph where each vertex represents a task to be completed and an edge from vertex *a* to vertex *b* means that task *a* must complete before task *b* begins. The absence of an edge between two vertices means that there is no dependency between them and hence the related tasks can be run in parallel.

The goals of this chapter are to:

1. Analyse a program for potential parallelism. Create a data dependency graph to identify the main tasks and their dependencies.
2. Apply task and data *decomposition patterns* to break a program into pieces that can execute concurrently (Mattson, Sanders and Massingill, 2005).
3. Choose an *algorithm structure pattern* that is effective, simple and portable (Mattson, Sanders and Massingill, 2005).
4. Map the products from step 3 to multitasking language features in C++.

From this list we note that no mention has been made of the object-oriented programming model or threads. In the former case we do not see object technology as the main driver of parallel programming design and in the latter case we shall see tasking as being easier to apply and less error-prone than the use of threads. In short, we wish to have a one-to-one relationship between the tasks and their implementation in C++. To this end, we discuss the following language features:

- Implementing asynchronous operations.
- Using *futures* and *promises* to access the results of asynchronous operations.
- Running a function asynchronously by calling `std::async`.

- Using *shared futures* to access the results of asynchronous operations. These are similar to 'normal' futures except that multiple threads are allowed to wait for the same shared state in the former case.
- Creating a *packaged task* to wrap a *callable object* (for example, free function, lambda function, bind expression or function object) so that it can be invoked asynchronously.
- Exception handling in multitasking applications.

Summarising, we trace the design trajectory from initial requirements to implementing a parallel program in C++. We take simpler examples for motivation and we examine the applicability of the *C++ Concurrency* library to a number of the applications in this book, for example the Monte Carlo and finite difference methods. These design techniques can also be applied to other problems in computational finance.

29.2 FINDING CONCURRENCY: MOTIVATION

Concurrency refers to the existence of multiple threads of execution, each one receiving a slice of time to execute before being pre-empted by another thread. Concurrency is a requirement for a program that must react to external events and to stimuli resulting in data transfer from user input devices and sensors, for example. Typical examples of concurrent programs are operating systems and computer games. We note that a program can be concurrent even on a one-core machine. *Parallelism* involves several threads executing at the same time on multiple cores. The main goal of *parallel programming* is to improve the performance of compute-intensive applications that should not be interrupted when running on multiple cores. A task is the smallest exploitable unit of concurrency.

We are interested in applications that consist of concurrent tasks running on multiple processors. The question is how to design a system that makes optimal use of available hardware. In short, we wish to improve the *speedup*. Speedup is a scalar value that indicates how many times faster a parallel program executes than its sequential counterpart. It is intuitively obvious that we must find some way to decompose a system into tasks with each one running on a single processor. A task should not be too *coarse grained* or too *fine grained*.

It is not clear to the author how traditional object-oriented technology can be used as the driver of a system decomposition into concurrent tasks. Even in the case of a system that will run in single-threaded mode we use system decomposition methods as introduced in Chapter 9 rather than initially looking for objects and classes. On a related note we can imagine that porting single-threaded C++ applications to parallel code will be very time-consuming at best and impossible at worst. It is a dilemma and the possible choices are to do nothing to the application or to redesign it from scratch.

We now embark on a discussion of how to analyse and design parallel programs. In all cases and at each level of abstraction we are looking for ways to break a system into (quasi-) independent concurrent tasks. At some stage objects and classes will play a role, but only when we are reasonably sure that the *design blueprints* are stable.

29.2.1 Data and Task Parallelism

The two main decomposition patterns to break a problem into subproblems that can execute concurrently are based on data and tasks. *Task parallelism* (also known as *function parallelism*

and *control parallelism*) is based on the model that a problem is a stream of instructions that can be broken down into a sequence of tasks that run simultaneously. This approach is similar to the way we decomposed systems in Chapter 9. Task parallelism is concerned with running different tasks on the same or different data at the same time. Communication between the tasks takes place by passing data from one task to the next one as part of a *workflow*. An example of task parallelism is the *pipelining model* in which single data is moved between a series of tasks where each task can execute independently of the others. Each stage or *filter* in the pipeline can handle a different unit of input at a time. This is similar to an assembly line and we shall discuss this model in Chapter 30. A special case is the *Producer–Consumer* model. *Data decomposition* involves decomposing data into *chunks* and these chunks are input to tasks. Ideally, it should be possible to operate on the chunks relatively independently of each other. An example of where data decomposition can be used is in the addition and multiplication of matrices. In this case we decompose a matrix into blocks (submatrices) in such a way that each block fits into cache memory. Another example is *loop-level parallelisation*, in which an array is broken into chunks and a single operation is applied to each chunk. Examples of data parallel programming environments are *Message Passing Interface* (MPI) and *Open Multi Processing* (OpenMP) (see Gropp, Lusk and Skjellum, 1997 and Chapman, Jost and Van der Pas, 2008, respectively). This data model is used extensively in GPU applications.

The distinction between data and task decomposition is somewhat artificial and most programs fall on a continuum between these two extremes. They can be combined in the same application.

29.3 TASKS AND TASK DECOMPOSITION

In this section we introduce some design techniques to ease the transition to writing multitasking code in C++.

29.3.1 Data Dependency Graph: First Example

We already mentioned that a *data dependency graph* or *task graph* is a directed graph in which each vertex represents a task to be completed and an edge from vertex *a* to vertex *b* means that task *a* must complete before task *b* begins. We are now in a position to draw these graphs as we know what tasks are. To help us get started, we view the graph as a data flow diagram that processes *primary input data* and transforms it in some way to produce *major output data*. We take an initial example in Figure 29.1 in which functions receive data and produce data. At this stage we are assuming that there is no shared data, in which case the functions `F1`, `F2`, `F3` and `F4` are considered to be *pure functions*, that is:

- The functions always evaluate the same result given the same argument value(s). The function result cannot depend on any hidden information or state that may change while program execution proceeds or between different executions of the program, nor can it depend on any external input from I/O devices.
- Evaluation of the result does not cause any semantically observable *side-effects* or output, such as mutation of mutable objects or output to I/O devices.

Assuming that Figure 29.1 is an accurate description of the problem that we wish to design we then decide to create a first sequential implementation. We run this program and observe

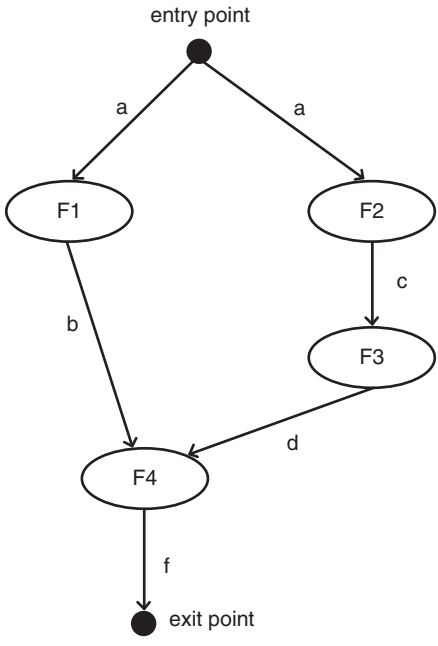


FIGURE 29.1 My first data dependency graph

the output that it produces and the total running time. We compare these results with those produced by the task-based solutions. First, the simple functions `F1`, `F2`, `F3` and `F4` are defined by:

```
double F1(double a)
{
   std::cout << "F1\n";
   return a;
}


double F2(double x)
{
   std::cout << "F2\n";
   return x - 12.0;
}


double F3(double x)
{
   std::cout << "F3\n";
   return x + 10.0;
}

double F4(double x, double y)
```

```
{
   std::cout << "F4\n";
   return x + y;
}
```

The C++ sequential code to implement the directed graph in Figure 29.1 is:

```
double SequentialTaskGraph(double a)
{ // Single thread of control

   std::cout << "Sequential \n";
   double b = F1(a);
   double c = F2(a);
   double d = F3(c);

   double f = b + d;

   return f;
}
```

A simple test case is:

```
int main()
{
   double a = 22.0;
   std::cout << SequentialTaskGraph(a) << std::endl;

   return 0;
}
```

The final output is the value 42. This code runs but it does not take advantage of the potential parallelism in the graph because `F1` and `F2` can run in parallel, as can `F1` and `F3`. We also note that there are no side-effects in this code because all the functions are pure. We shall see how to parallelise this code using *C++ Concurrency* in Section 29.4.

29.3.2 Data Dependency Graph: Generalisations

We give some more examples of data dependency graphs to help us gain insights into discovering parallelism before we start programming. Most graphs fall into one of two categories:

- *Data flow graph*: data is passed along the graph's edges. Vertices receive data messages, transform them and then pass them on to other vertices.
- *Data dependence graph*: vertices operate directly on shared data and data is not passed along edges.

The example in Figure 29.2 is an example of data parallelism in which operation *B* is performed with three different inputs from operation *A*. Here we see the same operation *B* being performed on different subsets of the same data. The output from the *B*s are merged in operation *C*. Figure 29.3 is an example of task parallelism in which operations *B*, *C* and *D* may be performed concurrently. Finally, Figure 29.4 is a sequential dependency graph.

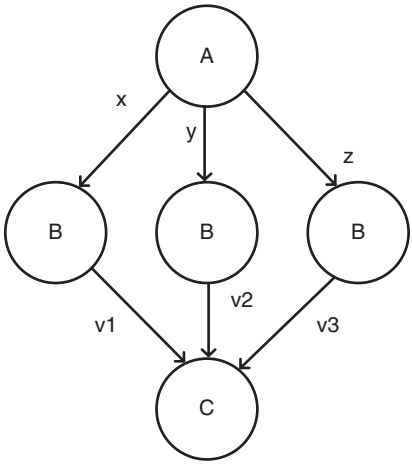


FIGURE 29.2 Data parallelism

We shall discuss how to implement data dependency graphs using Microsoft's *Parallel Patterns Library* (PPL) in Chapter 30. We note that Intel's *Threading Building Blocks* (TBB) has support for these data structures, although a discussion of TBB is outside the scope of this book. In Chapter 30 we shall also discuss task-based message passing and dataflow models.

29.3.3 Steps to Parallelisation

It is clear that we need a process to analyse, design and implement programs that execute on multiprocessor hardware. The code must give accurate results and it must be optimised for performance. To this end, a summary of the main steps is:

1. *Decomposition*: break the program into tasks.
2. *Assignment*: assign tasks to processes.

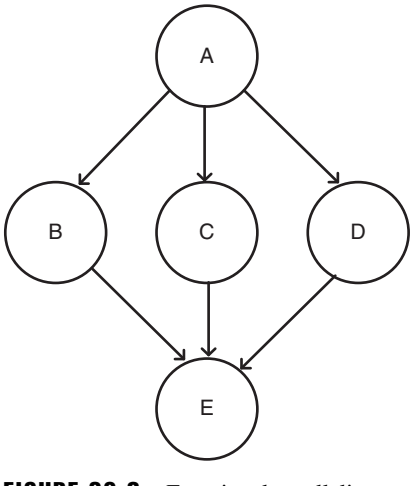


FIGURE 29.3 Functional parallelism

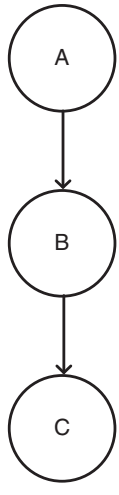


FIGURE 29.4 Sequential computation

3. *Orchestration*: data access, communication and synchronisation of processes.
4. *Mapping*: bind processes to processors.

This is an iterative (sometimes combined with trial-and-error experimentation) process and we can spend much time making the program correct, free of race conditions and improving its speedup.

29.4 FUTURES AND PROMISES

These terms originated in the late 1970s. Futures and promises are constructs for synchronising program execution in programming languages. They describe an object that acts as a proxy for a result that is initially unknown (the terms *delayed* and *deferred* are also used). The result is unknown because the computation of the value has not yet completed.

More precisely, a *future* represents a read-only placeholder view of a variable while a *promise* is a writable, single-assignment entity that sets the value of the future. In short, the future is the value and a promise is the (asynchronous) function that sets the value. We can define a future without specifying which specific promise will set its value. Different possible promises may set the future's value; however, *the value can only be set once*. It is possible to associate a future and a promise, as we shall see.

There is quite a bit of functionality to introduce. To this end, the class template `std::future` has functionality for the following:

1. Create a future to hold the result of an asynchronous operation that is created using `std::async` , `std::packaged_task` or `std::promise`.
2. We can use several methods to query, wait for or extract a value from the `std::future`.
3. The asynchronous operation can modify *shared state* (which is linked to the created `std::future`) when it is ready to send the result to the creator.

The class template `std::promise` provides a facility to store a result/value or an exception that is later acquired asynchronously via an `std::future` instance. It has functionality for the following:

1. *Make ready*: store the result or exception in the shared state. The state is marked as ready.
2. *Release*: the promise gives up its reference to the shared state.
3. *Abandon*: the promise stores the exception of type with error code `std::future_errc::broken_promise`, makes the shared state *ready* and then *releases* it.

The class template `std::async` runs a function asynchronously (possibly in a separate thread) and it returns an `std::future` instance that will eventually hold the result of the function call. The class template `std::packaged_task` wraps a *callable object* (free function, lambda expression, bind expression or function object) so that it can be invoked asynchronously. Its return value or exception is stored in a shared state that can be accessed through `std::future` instances.

29.4.1 Examples of Futures and Promises in C++

We now give some examples on how to define asynchronous functions. Once we understand how the functionality is used in these simple cases we can then generalise the results to larger problems (dependency graphs) as well as more complex algorithms and applications involving writable shared data. In particular, it is interesting to brainstorm at this stage on how to use parallel programming code in the designs that we introduced in Chapter 9.

The approach is to create a simple function that simulates a real-life algorithm. In this case we introduce a *sleep operation* that mirrors *load balancing* effects and delays:

```
double compute(double x)
{
   // Wait a while; simulates delays and load balancing effects
   std::default_random_engine dre(42);
   std::uniform_int_distribution<int> delay(0, 4000);
   std::this_thread::sleep_for (std::chrono::milliseconds(delay(dre)));

   return x*x;
}
```

We also have a variant when using a promise as input argument:

```
void computePromise(double x, std::promise<double> &p)
{
   // Wait a while
   std::default_random_engine dre(42);
   std::uniform_int_distribution<int> delay(0, 10);
   std::this_thread::sleep_for (std::chrono::milliseconds(delay(dre)));

   p.set_value(x*x);
}
```

In this latter case see how the promise stores a value that will be acquired at a later stage by a future.

We are now ready to discuss the different ways to run the algorithm. The first option runs asynchronously:

```
double x = std::sqrt(42.0);

// A. 'Direct' tasks
std::future<double> fut = std::async(compute, x);

// Get the shared data
double result = fut.get();
std::cout << "Result async : " << result << '\n';
```

The second option is more complicated because we are creating a promise and associating a future with it. We explicitly create a thread that runs the algorithm and the promise stores the result of the computation. The future waits for the algorithm to complete before it gets the value. The caller also waits on the thread to complete (notice the use of move semantics):

```
// B. Using a promise
std::promise<double> pr;
std::future<double> futP = pr.get_future();

std::thread t(computePromise, x, std::move(pr));
futP.wait();
std::cout << "Result promise : " << futP.get() << '\n';
t.join();
```

The third option is to use a *packaged task* that is run in the same thread as the caller:

```
// C. (Delayed) packaged task
std::packaged_task<double(double)> task(compute);
std::future<double> fut2 = task.get_future();

task(x);
// Get the shared data
std::cout << "Starting packaged task: " << fut2.get() << '\n';
```

The fourth and final option is when we run the task in its own thread:

```
// D. Packaged task in a separate thread
{
   std::packaged_task<double(double)> task(compute);
   std::future<double> fut2 = task.get_future();

   std::thread t(std::move(task), x);
   t.join();

   // Get the shared data
   std::cout << "Starting packaged task from a separate thread: "
             << fut2.get() << '\n';
}
```

These test cases and corresponding code can be seen as exemplars or templates for other applications. We notice the use of move semantics in the fourth option. Finally, in all cases the output is the magic number 42.

Futures are used for specific *one-off events*. A client can perform other tasks until it needs the event to have occurred and before it can proceed. It waits for the future to be *ready*. *Once this event has ocurred the future cannot be reset*. In other words, there is a one-to-one correspondence between the future and the event. In conclusion, a future provides an easy way to return a value directly from a task, in contrast to threads where this desirable feature is absent.

29.4.2 Mapping Dependency Graphs to C++

Having discussed what promises and futures are and having given some initial examples we now discuss how they scale to larger problems, for example, by applying them to the task graph in Figure 29.1. There are different ways to parallelise this graph and the options are given below:

```
void F1Promise(std::promise<double>& p)
{
   std::cout << "F1Promise\n";
   p.set_value(22.0);
}


void F2Promise(std::promise<double>& p, double x)
{
   std::cout << "F2Promise\n";
   p.set_value(x - 12.0);
}


double ParallelTaskGraphI(double a)
{ // Parallel with futures

   std::cout << "Parallel I \n";
// double b = F1(a);
   std::future<double> b(std::async(F1, a));
   double c = F2(a);
   double d = F3(c);

   double f = b.get() + d;

   return f;
}

double ParallelTaskGraphII(double a)
{ // Parallel with futures

   std::cout << "Parallel II \n";
   //   double b = F1(a);
```

```
    std::future<double> b(std::async(F1, a));
    //double c = F2(a);
    std::future<double> c(std::async(F2, a));
    double d = F3(c.get());

    double f = b.get() + d;

    return f;
}


double ParallelThreadPromises(double a)
{ // Parallel with threads and promises

    // We create 2 'child' threads, one of which is detached
    //
    // !! creating two detached threads does not seem to work

    std::cout << "Parallel Promises \n";


    std::promise<double> b;
    std::thread t(F1Promise, std::ref(b));
// t.detach();
    std::future<double> futB(b.get_future());

    std::promise<double> c;
    std::thread t2(F2Promise, std::ref(c), a);
    std::future<double> futC(c.get_future());

    t.join();
    t2.join();

    double d = F3(futC.get());

    // Fulfill promise; synchronise with getting the future
    double f = futB.get() + d;

    return f;
}
```

These solutions can be used as templates for more complex systems. It is worthwhile to study the code and check that each of the above functions is an implementation of the task graph in Figure 29.1.

29.5 SHARED FUTURES

We introduced what could be called *unique futures* in Section 29.4. They are modelled on `std::unique_ptr` . The class template `std::shared_future` provides a mechanism to

access the result of asynchronous operations. Multiple threads are allowed to wait for the same shared state. Unlike `std::future`, which is only moveable (so only one instance can refer to any particular asynchronous result), `std::shared_future` is copyable and multiple shared future objects may refer to the same shared state. Access to the same shared state from multiple threads is safe if each thread does it through its own copy of a shared future object.

We can create a shared future using constructors:

- Default constructor: create an empty shared future (does not refer to a shared state).
- Construct a shared future that refers to another shared future.
- Transfer ownership from a future or a shared future using move semantics.

Furthermore, `std::future` has a member function `share()` to transfer the state to a new shared state.

We now show some examples of creating shared futures. The code is self-explanatory:

```
// Shared future 101
std::future<int> fut = std::async(GetNumber);
std::shared_future<int> sfut = fut.share();        // move
std::shared_future<int> sfut2 = fut.share();      // Fut is empty
std::cout << "sfut " << sfut.get() << std::endl;
std::cout << "sfut*2 " << sfut.get() * 2 << std::endl;

if (sfut2.valid())
{ // Check for empty state

   std::cout << "sfut2 " << sfut2.get() << std::endl;
}
else
{
   std::cout << "No valid state\n";
}

// Other ways to create a shared future
std::future<int> fut3 = std::async(GetNumber);
std::shared_future<int> sfut3(std::move(fut3));
std::shared_future<int> sfut4(sfut3); // Both refer same shared state
std::cout << "sfut3 " << sfut3.get() << std::endl;
std::cout << "sfut4 " << sfut4.get() << std::endl;

std::future<int> fut4 = std::async(GetNumber);
std::shared_future<int> sfut5(std::move(fut4));

// End of 101 code
```

We conclude our discussion of shared futures with an extended example in which we launch three threads. First, the thread function that all three threads fire is:

```
void DoIt(char c, std::shared_future<int> f)
{ // Thread function that uses a shared future whose state can be
```

```
    // referenced by multiple futures.

     try
     {
        // Wait for number of characters to print
        int num = f.get();

        for (int i = 0; i < num; ++i)
        {
           std::this_thread::sleep_for
              (std::chrono::milliseconds(100));

           // Locking just to avoid a mess on the console
           std::mutex mut;
           mut.lock();
           std::cout.put(c).flush();
           mut.unlock();
        }
     }
     catch (const std::exception& e)
     {
        std::cerr << "Exception in thread "
                  << std::this_thread::get_id()
                  << ", " << e.what() << std::endl;
     }
}
```

We used a lock in the try block to avoid non-deterministic race conditions when writing to the console. Next, we create a shared future:

```
std::shared_future<int> f = std::async(GetNumber);

int GetNumber()
{
   return 40;
}
```

Finally, we launch three threads:

```
// Launch 3 threads that all write to Console
char c = '.';
auto fA = std::async(std::launch::async, DoIt, c, f);

c = '+';
auto fB = std::async(std::launch::async, DoIt, c, f);

c = '*';
auto fC = std::async(std::launch::async, DoIt, c, f);

// Now return the values
fA.get();fB.get(); fC.get();
```

Typical output from all the code from the beginning of this section is:

```
sfut 40
sfut*2 80
No valid state
sfut3 40
sfut4 40

*+.*.+*.+*.+*+..+**+.+.*+.**.+*.+.+**+..+**.++.*.+**+..+*.*+*.++*.*+.*
+..+*.*+.+**.++.**.+*.++*.+*.*+.+.**+.*.+*+.*+.*+.
```

29.6 WAITING ON TASKS TO COMPLETE

A typical application implements some kind of task graph as shown in Figures 29.1 to 29.4. We remark that tasks and threads need to wait on other tasks and threads to complete processing. We recall that in the case of threads we have a number of options:

- The waiting thread can continuously check a flag in shared data. Once the 'called' thread has completed it can set the flag.
- The waiting thread can sleep for small periods and then wake up to check if the flag has been set.
- The third (and preferred) option is to use a *condition variable* to wait on an event. One or more threads can wait on a *condition* to be satisfied. When one of the waiting threads has determined that the condition has been satisfied it will notify one or more threads waiting on the condition variable and then wake them up so that they can continue processing.

The first two of the above choices is not recommended in general as they are wasteful of resources. In this section we discuss how tasks wait on other tasks. To this end, the class templates for futures and shared futures have three member functions:

- `wait`: blocks waiting task until a resource becomes available.
- `wait_for`: blocks until a specified *duration* has elapsed or the result becomes available (whichever one comes first). The state of the result is returned.
- `wait_until`: waits for the result to become available. The task blocks until a *specified time* has been reached or the result becomes available. The return value indicates why `wait_until` returned.

We give examples of computing the ubiquitous *Fibonacci numbers*. We introduced a delay into the computations in order to simulate timeouts. You can modify the values and run the code to see what output is produced. The function to be launched and the corresponding futures are:

```
#include <iostream>
#include <future>
#include <thread>
#include <chrono>

int fib(int n)
{
```

```
   if (n < 3) return 1;
   else return fib(n - 1) + fib(n - 2);
}

std::future<int> f1 = std::async(std::launch::async, []()
{
   return fib(20);
});

std::future<int> f2 = std::async(std::launch::async, []()
{
   return fib(25);
});

std::cout << "Wait until result becomes available...\n";
f1.wait();
f2.wait();

std::cout << "f1: " << f1.get() << '\n';
std::cout << "f2: " << f2.get() << '\n';
```

The second wait style using `wait_for()` is:

```
// Wait 1) for a duration, 2) for a specified amount of time

// 1
std::future<int> f3 = std::async(std::launch::async, []()
{
   std::this_thread::sleep_for(std::chrono::seconds(10));
   return fib(20);
});

std::future_status status;
do
{
   status = f3.wait_for(std::chrono::seconds(5));
   if (status == std::future_status::deferred)
   {
      std::cout << "deferred wait for\n";
   }
   else if (status == std::future_status::timeout)
   {
      std::cout << "timeout wait for\n";
   }
   else if (status == std::future_status::ready)
   {
      std::cout << "ready for wait for!\n";
   }
} while (status != std::future_status::ready);

std::cout << "Result of wait for is " << f3.get() << '\n';
```

The third wait style using `wait_until()` is:

```
// 2
std::chrono::system_clock::time_point two_seconds_passed
   = std::chrono::system_clock::now() + std::chrono::seconds(2);

std::future<int> f4 = std::async(std::launch::async, []()
{
   std::this_thread::sleep_for(std::chrono::seconds(1));
   return fib(25);
});

std::future_status status2;
do
{
   status2 = f4.wait_until(two_seconds_passed);

   if (status == std::future_status::deferred)
   {
      std::cout << "deferred wait until\n";
   }
   else if (status == std::future_status::timeout)
   {
      std::cout << "timeout wait until\n";
   }
   else if (status == std::future_status::ready)
   {
      std::cout << "ready wait until!\n";
   }
} while (status != std::future_status::ready);

std::cout << "Result of wait until is " << f4.get() << '\n';
```

In the above code we see that the return value of `wait_for` and `wait_until` can be one of the following:

- `future_status::deferred`: the function to calculate the result has not been started yet.
- `future_status::ready`: the result is ready.
- `future_status::timeout`: the timeout has expired.

29.7 CONTINUATIONS AND FUTURES IN BOOST

A common pattern in asynchronous programming is when the output of one operation becomes the input to another operation in a pipeline. This is similar to *function composition* in mathematics. A continuation can be implemented by a *callback* that executes when an operation has completed. In general, a continuation will not begin until the preceding (*antecedent*) task

has completed. If an exception is thrown then the succeeding continuation can handle it in a try-catch block.

We take two examples. The first example is an implementation of the graph in Figure 29.1. The implementation is not exactly in one-to-one correspondence with the graph but it is clear what the code is doing (and we still get the value 42):

```
double ParallelTaskContinuation(double a)
{ // Parallel with futures in BOOST

   std::cout << "Parallel with future continuations \n";
   //   double b = F1(a);
   boost::future<double> b = boost::async([=]() { return F1(a); });

   double val = 10.0;
   boost::future<double> c = b.then([&val](boost::future<double> f)
                              {return f.get() + val; });

   double f = F3(c.get());

   return f; // a = 22 => f == 42
}
```

The second example is an application of function composition in mathematics:

```
double ChainingContinuation()
{ // Parallel with futures in BOOST, composing functions
  // exp(sqrt(x*x));

   double x = 5.0;

   // Functions to be composed in a pipeline
   auto f1 = [](double x) { return x*x; };
   auto f2 = [](double x) { return std::sqrt(x); };
   auto f3 = [](double x) { return std::exp(x); };

   std::cout << "Parallel with future chaining continuations \n";
   boost::future<double> fut1 = boost::async([=]() { return f1(x); });

   boost::future<double> fut2 = fut1.then([=](boost::future<double> f)
            {return f2(x); });
   boost::future<double> fut3 = fut2.then([=](boost::future<double> f)
            {return f3(x); });

   return fut3.get(); // should be 148.413
}
```

This code can be described as implementing a simple data flow program.

This design approach can easily be extended and generalised to more complex cases. Continuations registered using the `.then` function avoid blocking waits or wasting threads on

polling, thus greatly improving the responsiveness and scalability of an application. The construct `future.then()` provides the ability to sequentially compose two futures by declaring one to be the *continuation* of another. With `then()` the antecedent future must be ready (has a value or exception stored in the shared state) before the continuation starts. We note that C++ does not support continuations at the moment of writing.

29.8 PURE FUNCTIONS

The concept of a *pure function* is pervasive in functional programming. The following statements hold for these kinds of functions:

- The function always evaluates the same result for the same input. It neither contains hidden information (state) nor can it modify any external state.
- Evaluation of the result does not cause any semantically observable *side-effects* or output, such as mutation of mutable objects.

In other words, a pure function need not depend on any or all of the argument values but it may only depend on a subset of them and on nothing else. A function will be *impure* if an argument is passed by reference, for example. In this case any parameter mutation will alter the value of the argument outside the function.

Examples of pure functions are:

- The C++ math functions, for example `std::cos(x)`.
- 3+3, 148.413*4.

Examples of impure functions are:

- Lambda functions with captured variables as modified references in their closure.
- Functions that return the current date or time.
- Many random number generators, for example `std::rand()` that uses and updates a global 'seed' state.
- The C function `printf()` is impure because it causes output to an I/O device.

One of the disadvantages of impure functions is the fact that bugs can be caused by unanticipated side-effects. Furthermore, functions may return different results for the same input. This situation is compounded by parallel and multithreaded code as it becomes more difficult to manage global side-effects. In short, the use of global data can lead to *race conditions*.

Ideally, we can implement tasks as pure functions, in which case we design systems using data dependency graphs (as shown in Figure 29.1, for example) without having to worry too much about synchronisation of shared state.

Summarising, pure functions are characterised by:

- They always produce the same result when given the same input parameters.
- They never have side-effects.
- They never alter state.

29.9 TASKS VERSUS THREADS

In Chapter 28 we discussed the use of *logical* (software) threads to create multithreaded code. The alternative is to employ asynchronous tasks as introduced in this chapter. The question now is to decide which choice is 'better' in some sense. In general, we are interested in performance, scalability and maintainability of applications. Logical threads are mapped to physical threads, usually in a one-to-one correspondence for higher efficiency. Mismatches and lower efficiency can occur due to *undersubscription* in which case there are not enough logical threads to keep the hardware busy while *oversubscription* occurs when there are more logical threads than physical threads. Oversubscription leads to *time-sliced* execution of logical threads, thus causing overheads.

We give a summary of some of the advantages of tasks:

1. Improved *load balancing*. The scheduler uses the correct number of threads and it distributes work evenly across those threads. A good task decomposition algorithm will help the scheduler assign tasks to threads and hence balance the load. On the other hand, it is the responsibility of the developer to manually perform load balancing when writing multithreaded code.
2. Faster task startup and shutdown. Tasks are lighter than logical threads. They can be anywhere from 18 to 100 times faster to start and stop than the corresponding thread operations.
3. More efficient evaluation order. Thread schedulers are democratic in the sense that they typically distribute time slices to threads in a *round-robin* fashion. Thus, each logical thread gets a fair share of the allocated time. Task schedulers, however, do have some higher-level information and they can delay starting a task until it can make useful progress.
4. We can concentrate on the logical dependencies between tasks (using dependency graphs, for example) and we can let the scheduler do what it does best, namely task scheduling. Thread notification and synchronisation is more complicated.
5. For novice (and experienced) developers, tasks are easier to understand and to implement than threads. Multithreaded code can be difficult to debug (are there race conditions and performance issues?) and difficult to scale when more hardware comes on board.

29.10 PARALLEL DESIGN PATTERNS

It is now generally accepted that some kind of top-down system decomposition approach is needed when designing concurrent software systems. We look for *exploitable concurrency* and one way to achieve this is to decompose a problem or system into tasks that can execute concurrently. To this end, one technique is to use the data dependency graph to depict tasks, applying different operations to different data elements. Tasks can be coarse grained or fine grained. This is sometimes called *functional parallelism*.

Some examples are:

- Fine-grained functional parallelism embedded in a sequential algorithm.
- Coarse-grained tasks to load, scale, filter and display images in an image-processing application.
- Parallel matrix multiplication.

Being able to create a task dependency graph for a problem increases the opportunity of finding potential concurrency and creating software with a good speedup. Furthermore, we determine the most appropriate *architectural style* for the problem at hand. Some high-level common styles and patterns are (Mattson, Sanders and Massingill, 2005; POSA, 1996; Shaw and Garlan, 1996):

- S1, *Task Parallelism*: Decompose a problem into a collection of tasks that run concurrently. A parallel algorithm is a collection of concurrent tasks. A complication is when these tasks read and write data that is shared between them.
- S2, *Divide and Conquer (recursive decomposition)*: Break a problem into a number of smaller subproblems, solve each one independently and then merge the subproblems into a solution for the full problem. This is a well-known technique when implementing sequential algorithms. The objective of this parallel pattern is to do the same thing using concurrent tasks. In other words, this is a form of *dynamic task parallelism* for applications that use data structures such as trees and graphs.
- S3, *Geometric Decomposition*: Organise an algorithm around a data structure that has been decomposed into concurrently updatable chunks. In this case the core data structure is known or readily found and the general case is when independent tasks apply the same (or different) operations to different parts or subsets of a data structure.
- S4, *SPMD (Single Program, Multiple Data)*: A single task operates on separate (semi-) independent data chunks.
- S5, *Pipeline*: Perform a sequence of calculations on many sets of data in which the calculations can be viewed in terms of data flowing through a sequence of *stages*. This model assumes (a) a long stream of input, (b) a series of suboperations (stages/filters) through which every unit of input must be processed and (c) each processing stage can handle a different unit of input at a given time. A good analogy is a car manufacturing assembly line.
- S6, *Producer–Consumer* (Demming and Duffy, 2010): This is a special case of the *Pipeline* pattern. In this case a *producer agent* (data source) sends messages to a message block and a *consumer agent* (data sink) reads messages from this block. In other words, the producer and consumer agents are concurrent tasks that communicate by means of *inter-process communication*.
- S7, *Master/Worker* (see also Nichols, Buttlar and Farrell, 1996): This pattern consists of two main entities; first, the *master* that initiates the computation and one or more *workers*. The master creates a bag of tasks and it waits for all tasks to complete. A worker takes a task from the bag, executes it and then fetches the next task. This is a form of *work stealing*.
- S8, *Loop Parallelism*: This pattern is concerned with parallelising computationally intensive loops. This is a common issue in many numerical applications that make extensive use of vector and matrix data structures.
- S9, *Shared Queue*: This pattern describes the steps to be taken when multiple tasks or threads share a queue data structure in a safe manner. This is a special case of a more general problem, namely creating concurrent containers and concurrent objects.
- S10, *Parallel Reduction/Parallel Aggregation*: This pattern is closely related to the *Loop Parallelism* pattern. In this case multiple inputs are combined into a single output. Formally, a *reduction* is an operation that uses a binary associative operator combining a set

of values into a single value. A good example is summing the values of a large array to produce a single value.

- S11, *Message Passing Paradigm*: This is a ubiquitous paradigm and a key component in models of concurrency and object-oriented programming. Using messages is an alternative to shared data structures in combination with *locks* and *semaphores* in order to avoid race conditions. There is no shared data between tasks that exchange information using messages. In this case we speak of an *isolated state*. An example of a library using this paradigm is the *Asynchronous Agents Library* that we discuss in Chapters 30 and 32.

These 11 patterns represent proven ways of analysing and designing concurrent systems. In order to determine which pattern (or combination of patterns) to use, we can deploy data or task decomposition to find potential parallelism rather than using traditional object-oriented decomposition. We shall elaborate on these topics in later chapters, when we examine extended examples and applications. The examples highlight essential features and functionality before proceeding to more advanced topics.

29.11 SUMMARY AND CONCLUSIONS

In this chapter we have introduced multitasking and parallel programming using functionality from the *C++ Concurrency* library. In contrast to low-level threads (introduced in Chapter 28) we took a top-down approach by decomposing a problem into concurrent tasks with the help of data dependency graphs. We trace the full software problem from an initial data dependency graph to an implementation using C++ futures.

We continue in Chapter 30 with an introduction to Microsoft's PPL, which supports multitasking. In Chapter 32 we show how to apply *C++ Concurrency* and PPL to improve the speedup of Monte Carlo applications.

29.12 QUIZZES, EXERCISES AND PROJECTS

1. (Concurrency versus Parallelism, Quiz)

Which of the following statements are applicable to concurrency or parallelism (or both)?

a) Use of shared resources.
b) Goals are correctness, performance (throughput) and robustness.
c) Reduce latency (*latency* is the fixed cost of servicing a request).
d) Prevent *thread starvation* (never allow a program to become idle).
e) Concurrency is an optimisation, parallelism is a functional requirement.

2. (Data Parallelism versus Task Parallelism, Quiz)

Which of the following statements accurately describe data parallelism best and which accurately describe task parallelism?

a) Synchronous/asynchronous computation.
b) Optimum load balancing/load balancing depending on hardware availability and scheduling algorithms.
c) Different operations performed on the same data.
d) Degree of parallelisation depends on number of independent tasks/input data size.
e) Fewer/more speedup use cases.

3. (Black–Scholes Gone Parallel, Useful to Get Hands-on Experience)

We adapt the code in Section 29.4.1 to make it more relevant to computational finance. Some of the *use cases* that could be investigated are:

- Running multiple instances of lattice, finite difference and Monte Carlo algorithms as we saw in Chapter 28. We can imagine a combination of *shared data* and *thread-local data*.
- Stress testing numerical algorithms, for example running the ADE and Crank–Nicolson methods a million times using randomly generated data and comparing the generated output.
- Richardson extrapolation: running the BTCS schemes with step sizes *dt* and *dt*/2 concurrently.
- Comparing the accuracy of the binomial method against that of the finite difference method.

In this exercise we take a simpler case which we can then extend. The objective is to compute call and put option prices using the exact Black–Scholes formula and the ADE method. Answer the following questions:

a) Use two stored lambda functions to encapsulate functionality to price options using the above two algorithms. Specify the signature (input–output) of each function.

b) Implement these algorithms using each of the options in Section 29.4.1. The output should be the same in all cases.

c) Can you see opportunities to modify the code to test some of the above use cases, for example, running Crank–Nicolson and ADE methods in their own threads?

4. (Tasks 101: Running Functions Asynchronously)

For this exercise you need to understand C++11 *futures*, `std::async` and thread/task launching policies.

This exercise involves running synchronous and asynchronous functions based on various launch policies. The functions should have both `void` and non-`void` return types. The objective is in showing how the tasks execute and exchange information.

Answer the following questions:

a) Create two functions having the following signatures:

```
void func1()
{

}

double func2(double a, double b)
{

}
```

You may insert any code you see fit into the body of these functions for the purpose of this exercise.

b) Use `std::async` (default settings) to launch `func1` and `func2`. Get the results of the computations and print them when applicable. Check the validity of the associated future before and after getting the result.

c) What happens if you try to get the result of an `std::future` more than once?

d) Now test the same code using the launch parameter `std::launch::async`. Do you notice any differences?

e) We now wish to asynchronously call a function at some time later in the client code (*deferred/lazy evaluation*). Get the result of the function and check that it is the same as before.

5. (Waiting on Tasks to Complete)

In this exercise the client thread fires up four asynchronous functions. The signature of the function is:

```
double Computation(int n);
```

The client waits (a kind of *barrier*) on the four associated futures. Then compute the sum and average of the four results. Compare the *speedup* of this program by comparing the processing time with that of the equivalent sequential program.

(This example is a simple case of SPMD, *Single Program, Multiple Data*.)

6. (Shared Futures 101)

We process the outcome of a concurrent computation more than once, especially when multiple threads are running. To this end, we use `std::shared_future` so that we can make multiple calls to `get()`. Answer the following questions:

a) Create the following *shared future* by calling the appropriate constructor:
- Default instance.
- As a shared future that shares the same state as another shared future.
- Transfer the shared state from a 'normal' future to a shared future.
- Transfer the shared state from a shared future to a shared future.

b) Check that the member functions in `std::future` are also applicable to `std::shared_future`.

c) Test what happens when you call `get()` twice on a shared future.

d) Create a shared future that waits for an infinite loop to finish (which it never does). To this end, use `wait_for` and `wait_until` to trigger a timeout.

7. (C++11 Promises 101)

A *promise* is the counterpart of a future. Both are able to temporarily hold a shared state. Thus, a promise is a general mechanism to allow values and exceptions to be passed *out of threads*. A promise is the 'push' end of the promise–future communication channel.

Answer the following questions:

a) Create a default promise, a promise with an empty shared state and a promise based on the move constructor.

b) Create a promise with `double` as stored value. Then create a future that is associated with the promise.

c) Start a thread with the new future from part b). Create a thread function that uses the value of the shared data.

d) Use the promise to set the value of the shared data.

8. (Packaged Tasks)

The added value of using a *packaged task* is that we can create a background task without starting it immediately. In particular, the task is typically started in a separate thread. Answer the following questions:

a) Consider the thread function:

```
// Thread function
double compute(double x, double y)
```

```
{
      // Wait a while
      std::default_random_engine dre(42);
      std::uniform_int_distribution<int> delay(0, 1000);
      std::this_thread::sleep_for (std::chrono::milliseconds(delay(dre)));

      return std::cos(x) * std::exp(y);
}
```

Write code to start a task:

```
double x = 0.0; double y = 2.71;

// A. 'Direct' tasks
std::future<double> fut = std::async(compute, x, y);

// Get the shared data
double result = fut.get();
std::cout << "Result: " << result << '\n';
```

b) Rewrite/port the code in order to use a packaged task and delayed execution.
c) Create a queue of packaged tasks, dequeue each task and execute it in the queue.

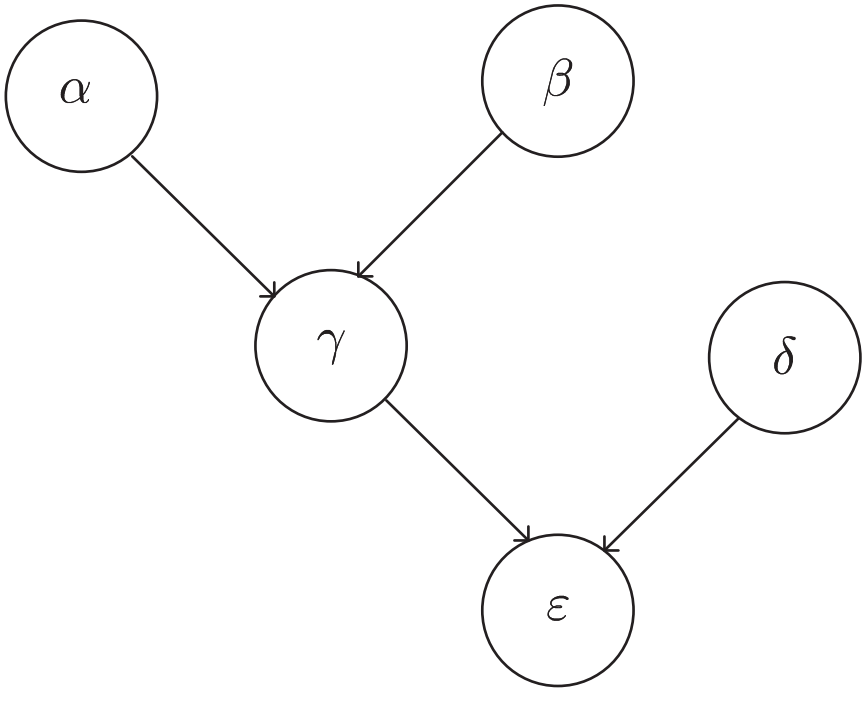


FIGURE 29.5 Alternative dependency graph

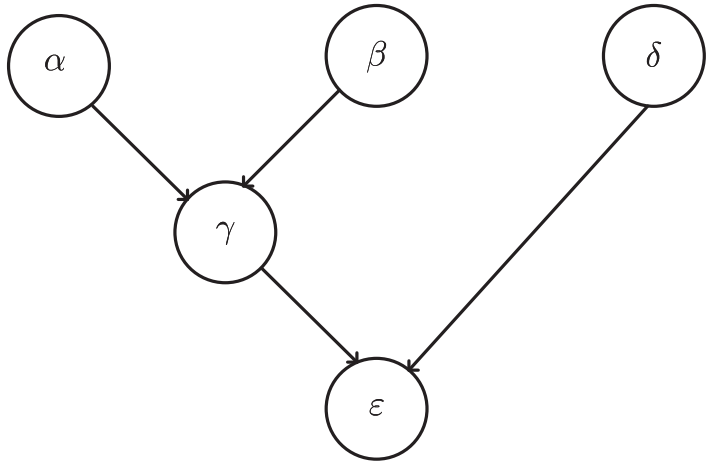


FIGURE 29.6 Alternative dependency graph, version 2

9. (Task Graph)

We give two examples of dependency graphs in Figures 29.5 and 29.6 in which nodes are implemented by the following functions (we use fancy Greek names to distinguish functions from variables):

$$
\begin{aligned}
v &= \alpha() \\
w &= \beta() \\
x &= \gamma(v, w) \\
y &= \delta() \\
z &= \varepsilon(x, y).
\end{aligned}
$$

These functions are unspecified and you may choose the types of the variables v, w, x, y and z.

Answer the following questions:

a) Implement these graphs using the C++ functionality similar to that in Sections 29.4.1 and 29.4.2. Let the variables v, w, x, y and z be `double`. Make sure that the final output is the same in both cases.

b) Now choose other, more heavyweight types such as matrices and vectors in order to emulate a non-trivial numerical algorithm. As before, make sure that the final output is the same in both cases.

c) Measure the performance of each solution in part b).

d) How would you implement the task groups in Figures 29.5 and 29.6 using *OpenMP*?

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 30

Parallel Patterns Language (PPL)

30.1 INTRODUCTION AND OBJECTIVES

In this chapter we introduce the *Parallel Patterns Library* (PPL) that implements an imperative programming model to promote scalability and ease of use when developing concurrent applications. PPL is a Microsoft product and has been supported since Visual C++ 2010. It is not part of the C++ standard and there are other libraries (such as Intel's *Threading Building Blocks* (TBB)) that offer similar functionality. Prediction is difficult but our guess is that future versions of the C++ standard might contain functionality that is not unlike that of PPL and TBB. PPL is easy to learn and to use in applications. The syntax is similar to that of STL and it supports C++11. The main challenge now is knowing which design pattern to use in a given context.

We assume the following prerequisites on the part of the reader (these topics have been discussed in previous chapters):

- Working knowledge of lambda functions, stored lambda functions, captured variables and closures.
- Knowing what *parallel design patterns* are and how to identify them in applications.
- Knowing the reasons for using parallel libraries and what the resulting consequences are.

Before we discuss PPL in detail we first give an overview of the lower-level component that is called the *Concurrency Runtime*. In a sense it abstracts away low-level infrastructure details related to concurrency. Its two main components are:

- *Task Scheduler*: this component can be seen as a special kind of *thread pool* which is used to optimise large numbers of fine-grained work requests. It does not automatically create a new worker thread when a request arrives but instead it queues the new task and executes it when processor resources become available.
- *Resource Manager*: this component allocates processor cores among an application's task schedulers. It ensures that related tasks execute optimally in the sense of accessing hardware cache. It also provides dynamic resource management to adjust the level of concurrency across components based on core utilisation.

We do not discuss the *Task Scheduler* and *Resource Manager* in this book. The *Concurrency Runtime* provides a default scheduler.

The PPL provides the following features:

- *Task parallelism*: a mechanism that works on top of the *Windows ThreadPool* to execute several work items (tasks) in parallel.
- *Parallel algorithms*: generic algorithms that work on top of the *Concurrency Runtime* to act on collections of data in parallel.
- *Parallel containers and objects*: generic container types that provide safe concurrent access to their elements and state.

We now introduce the main features in PPL. We hope that you find it interesting and you can find applications for it.

30.2 PARALLEL ALGORITHMS

In Chapters 17 and 18 we discussed STL algorithms in detail but they are not thread safe. PPL does provide a number of parallel algorithms that avoid the manual labour of having to devise means for dividing work between threads.

The parallel algorithms (in namespace `concurrency`) in PPL are:

- `parallel_for`: repeatedly perform the same task in parallel. Each task is parametrised by an iteration value. This algorithm is useful when the loop body does not share resources among iterations of the loop.
- `parallel_for_each`: this is similar to the corresponding `std::for_each` except that now the task is executed in parallel.
- `parallel_invoke`: executes a set of tasks in parallel. It returns when each task has finished. This algorithm is useful when we have several independent tasks that must execute in parallel.
- `parallel_transform`: this is the parallel version of `std::transform`. However, the operation order is not deterministic because the computation is done in parallel.
- `parallel_reduce`: this is the parallel version of `std::accumulate`. However, the operation order is not deterministic because the computation is executed in parallel.

In order to achieve good performance with `parallel_transform` and `parallel_reduce` we need to use large arrays. For arrays whose size is below a certain threshold we see that the corresponding STL algorithms will perform better in general.

- `parallel_sort`, `parallel_buffered_sort`, `parallel_radixsort`: these algorithms are useful for large datasets that can benefit from being sorted in parallel. These algorithms sort the elements *in place*, by which we mean that we produce an output in the same memory space that contains the input by successively transforming that data until the output is produced. This avoids the need to use twice the amount of storage – one array for the input and an equal-sized array for the output.

30.2.1 Parallel For

Sequential loops need no introduction. Most C++ applications in computational finance use single and double loops. We take a simple example of a loop that prints the elements of a vector on the console:

```
// Sequential loop
std::size_t n = 5; int start = 42;
std::vector<int> v(n);
std::iota(std::begin(v), std::end(v), start);

for (std::size_t i = 0; i < v.size(); ++i)
{
   std::cout << v[i] << ",";
}
```

Since there is only one thread of control and there is no contention when accessing the console, we can run this code and get the predictable output {42,43,44,45,46,}. In other words, we iterate sequentially in the vector and we print each of its elements in turn. Looking at the code we see that it can be parallelised (in principle at least) because of the independence between the operations in the body of the loop. To this end, PPL supports the `parallel_for` operation in the `concurrency` namespace. In this case the parallel code is:

```
// Parallel for uses a lambda function in closed-open range [first, last)
concurrency::parallel_for(std::size_t(0), v.size(), [&](std::size_t i)
{
   std::cout << v[i] << "*";
});
```

We can run this code a number of times; each time the output will be different. Some typical output is {464542*43*44***} and {46*44*45*42*43*}, which is rather chaotic and non-deterministic. What is going on? First, multiple tasks are now responsible for parts of the processing and there is no guarantee regarding the order in which the vector's elements are processed. Second, each task prints data on the console which can lead to a data race (hence the chaotic output). This latter problem can be resolved by locking the code but it will have severe consequences for performance (in such cases it can be much slower than the corresponding sequential code):

```
// Parallel code and locking the console
std::mutex my_mutex;
concurrency::parallel_for(std::size_t(0), v.size(), [&](std::size_t i)
{
   std::lock_guard<std::mutex> guard(my_mutex);
   std::cout << v[i] << ":";
});
```

In this case the console output will no longer be garbled; however, the elements will be printed in a random order. Typical output is {46:44:42:43:45:} and {46:42:43:45:44}.

30.2.2 Parallel `for_each`

This algorithm is the parallel version of the corresponding `for_each` algorithm in STL. It is very flexible because it allows us to access and modify the elements of a container in different ways. Specifically, we can use a lambda function, function object or stored lambda function to access or modify the individual elements of the container. We take a simple example to square the elements of a vector of longs using a lambda function:

```
// 0. Sequential code to square each element
std::for_each(std::begin(values),std::end(values), [](long& n) {n *= n; });
```

To take advantage of multiple cores, PPL supports the following code:

```
concurrency::parallel_for_each(std::begin(values), std::end(values),
                               [](long& n) {n *= n; });
```

Of course, instead of a lambda function we can use a function object and a generic lambda function and call the algorithm again:

```
template<class Ty>
      class SquareFunctor
{
public:
      void operator()(Ty& n) const
      {
         n *= n;
      }
};

// Use a function object (functor) to square each element.
concurrency::parallel_for_each(std::begin(values), std::end(values),
                               SquareFunctor<long>());

// Use a generic lambda C++14 function pointer to square each element.
auto sf = [](auto& n) {n *= n; };
concurrency::parallel_for_each(std::begin(values), std::end(values), sf);
```

In general, we can expect a speedup of (slightly) less than two (taking parallel overheads into account) when we run the parallel code on a dual-core machine. There are no data races because there is no shared data.

There is a difference between the parallel and sequential variants of the `for_each` algorithm, namely the former does not guarantee the order of execution.

This algorithm can be seen as a very simple case of SPMD (*Single Program, Multiple Data*). It can be applied to more computationally intensive algorithms operating on complex data structures.

30.2.3 Parallel Invoke and Task Groups

These operations allow us to define a collection of tasks as a unit or group and run all the tasks. There is no predefined execution order.

We first discuss the *parallel invoke* functionality. The first example involves two work functions `f1()` and `f2()` that we run in parallel:

```
void f1()
{
   std::cout << "first function\n";
}

void f2()
{
   std::cout << "second function\n";
   int f = 2;
}

// Parallel Invoke using lambda functions
concurrency::parallel_invoke(

   [] () { f1(); },
   [] () { f2(); }
);
```

The second example shows the use of calling two work functions that have input arguments:

```
void Compute(double x, std::vector<double>& result)
{
   for (std::size_t i = 0; i < result.size(); ++i)
   {
      result[i] =
         static_cast<double>(i) * std::cos(x) * std::sin(x);
   }

   std::cout << "\nFinished Compute \n";
}

void Print(double x)
{
   std::cout << "\nFinished Print, " << x << '\n';
}
```

We now execute these functions in parallel:

```
// Parallel Invoke using lambda functions
double x = 3.14; long n = 1'0000'000;
std::vector<double> v(n);

concurrency::parallel_invoke(

   [&] () { Compute(x, v); },
   [&] () { Print(v[v.size()-1]); }
);
```

If an exception is thrown by one of the work functions then it will be deferred and rethrown when all tasks finish. In the case of multiple exceptions being thrown the runtime chooses one of the exceptions to be rethrown. *The remaining exceptions will not be externally observed*. We simulate an application of tasks in which exceptions are thrown:

```
// Functions throwing exceptions
void exception1()
{
   throw -1;
}

void exception2()
{
   throw 'Z';
}

// Parallel invoke with exceptions
try
{
   concurrency::parallel_invoke(
      [&]() { exception1(); },
      []() { f1(); },
      [&]() { exception2(); }
   );
}
catch (int& e)
{
   std::cout << "\ninteger exception " << e << '\n';
}
catch (char& e)
{
   std::cout << "\ncharacter exception " << e << '\n';
}
```

You can run this code and see what happens.

We can reproduce the functionality of `parallel_invoke` by creating a task group and calling its `run` and `wait` methods. The first example is:

```
// Parallel task using task group
concurrency::task_group tg;

tg.run([]() {f2(); });
tg.run([]() {f1(); });
tg.wait();
```

It is also possible to combine the `run` and `wait` methods into a single operation:

```
// Parallel task using task group
concurrency::task_group tg2;
```

```
std::cout << "* Run and wait combined\n";
tg2.run([]() {f2(); });
tg2.run_and_wait([]() {f1(); });
```

The `run_and_wait` method acts as a hint to the scheduler that it can reuse the current context to execute the new task.

30.2.4 Parallel Transform and Parallel Reduction

These useful algorithms are the parallel versions of the STL algorithms `std::transform` and `std::accumulate`, respectively. They support random access, bidirectional and forward iterators.

We give examples and for completeness we also give the code in the sequential case:

```
std::vector<double> v1 = { 1.0, 2.0, 3.0, 4.0 };

// Sequential transform
// Modifier, predefined function objects
std::transform(v1.begin(), v1.end(), v1.begin(),
                std::negate<double>());
print(v1, std::string("Negated values: "));

std::transform(v1.begin(), v1.end(), v1.begin(),
                [](double d) { return -d; });
print(v1, std::string("Negated values: "));

// Parallel transform
auto MyNegate = [](double d) { return - d; };
concurrency::parallel_transform(std::begin(v1), std::end(v1),
                                std::begin(v1), MyNegate);
print(v1, std::string("Negated values: "));

// Reduce
double initVal = 0.0;
std::cout << concurrency::parallel_reduce(std::begin(v1),
                                          std::end(v1), initVal) << '\n';

double acc1 = std::accumulate(std::begin(v1), std::end(v1), initVal);
std::cout << "Sum , classic sum: " << acc1 << std::endl;
```

30.3 PARTITIONING WORK

In general, we improve the performance of algorithms that operate on a container by partitioning its index space into ranges and then assigning each range to a thread. In other words, we distribute consecutive chunks of data between threads. We can manually perform this partitioning but the end result is not scalable. Nonetheless, it is instructive to write user-defined code to get a feeling for how PPL works. Furthermore, you will need to hand-craft a solution because C++ does not support *automatic partitioning* at the moment of writing. To this end, we discuss an example of the *Divide and Conquer* parallel design pattern (Mattson, Sanders

and Massingill, 2005) applied to the sorting of a sequential container. We split the problem into a number of smaller subproblems, solve each one independently and then merge the subsolutions into a solution for the whole problem. We discuss a user-defined *quicksort* algorithm with three implementations, namely sequential C++, C++ threads and PPL's `parallel_invoke`. First, the sequential version is:

```
template <typename ForwardIt>
   void QuickSort(ForwardIt first, ForwardIt last)
{ // Sequential recursive implementation of Hoare's Quicksort algorithm

   // Base case for recursion
   if (first == last) return;

   // Get value at given distance std::distance(first, last) / 2
                                                              // from first
   auto pivot = *std::next(first, std::distance(first, last) / 2);

   // Move elements in [first,last] to front for t < pivot value
   ForwardIt middle1
      = std::partition(first, last, [pivot](const auto& t)
            { return t < pivot; });
   // Move elements in [middle,last] to front for t >= pivot value
   ForwardIt middle2
      = std::partition(middle1, last, [pivot](const auto& t)
            { return pivot >= t; });

   QuickSort(first, middle1);
   QuickSort(middle2, last);
}
```

For the C++ case we create two threads and then merge their separate results:

```
template <typename T>
   void MergeSort(std::list<T>& con)
{ // Parallel mergesort

     // 1. Split container into two pieces
     // 2. Quicksort the pieces
     // 3. Merge the pieces


   auto first = std::begin(con); auto last = std::end(con);
   if (first == last) return;

   auto pivot = std::next(first, std::distance(first, last) / 2);

   auto fn = [](auto f, auto s) {QuickSort(f, s); };

   std::thread t1(fn,first, pivot);
   std::thread t2(fn, pivot, last);
```

```
    // Wait on t1 and t2
    t1.join();
    t2.join();

    // Clumsy code; slows things down somewhat
    std::list<T> result;
    std::merge(first, pivot, pivot, last, std::back_inserter(result));
    con = std::move(result);
}
```

The PPL solution is:

```
template <typename T>
   void MergeSortPPL(std::list<T>& con)
{ // Parallel mergesort

    // 1. Split container into two pieces
    // 2. Quicksort the pieces
    // 3. Merge the pieces


    auto first = std::begin(con); auto last = std::end(con);
    if (first == last) return;

    auto pivot = std::next(first, std::distance(first, last) / 2);

    concurrency::parallel_invoke(
       [&first, &pivot]() {QuickSort(first, pivot); },
       [&pivot, &last]() {QuickSort(pivot, last); }
    );


    // Clumsy code; slows things down somewhat
    std::list<T> result;
    std::merge(first, pivot, pivot, last, std::back_inserter(result));
    con = std::move(result);
}
```

The reason for creating this code is to show how to partition a problem into subproblems and solve each subproblem. In general, we (and probably most developers) do not have the time to build production software for these kinds of problems and prefer to use a library such as PPL or TBB. In particular, PPL's `parallel_for`, `parallel_for_each` and `parallel_transform` algorithms provide overloaded versions that take an extra parameter called a *partitioner* that defines how work is divided. The default partitioning mechanism creates an initial workload and then uses a *work-stealing algorithm* and *range-stealing algorithm* to balance the partitions when the workloads become unbalanced.

There are four partitioning options:

- *Static Partitioner*: this partitioner divides work into a fixed number of ranges. It does not employ work stealing and hence has less overhead. It is optimal to use when each iteration of a parallel loop performs a fixed and uniform amount of work.

- *Affinity Partitioner*: this partitioner divides work into a fixed number of ranges. It is similar to a static partitioner but it improves *cache affinity* (moves the threads of a workload closer together) because of the way it maps ranges to worker threads.
- *Auto Partitioner*: this partitioner divides work into an initial number of ranges. The runtime uses this number as default. Each range can be divided into subranges, thus allowing load balancing to occur. The runtime redistributes subranges of work from other threads to that thread.
- *Simple Partitioner*: this partitioner divides work into ranges such that each range has at least the number of iterations that are specified by a given *chunk size*. This choice supports load balancing. However, the runtime does not divide ranges into subranges.

We give an example:

```
// Partitioners
auto negate = [](double d) { return -d; };

concurrency::static_partitioner stap;
concurrency::simple_partitioner simp(4); // Chunk size
concurrency::auto_partitioner autp;
concurrency::affinity_partitioner affp;

std::vector<double> v(10'000'000);
concurrency::parallel_transform(v.begin(), v.end(), v.begin(), negate, stap);
concurrency::parallel_transform(v.begin(), v.end(), v.begin(), negate, simp);
concurrency::parallel_transform(v.begin(), v.end(), v.begin(), negate, autp);
concurrency::parallel_transform(v.begin(), v.end(), v.begin(), negate, affp);
```

We conclude this section with the remark that the type of application and its workload will determine which partitioner to use in a given context. It is recommended that you conduct experiments to determine which option is most suitable for your application.

30.3.1 Parallel Sort

PPL has support for three sorting algorithms:

- *Parallel Sort*: this is a general *compare-based* algorithm meaning that elements are compared by value. It has no additional memory requirements and is suitable for general-purpose sorting.
- *Parallel Buffered Sort*: this is a compare-based algorithm. It performs better than the general parallel sort algorithm but requires an additional $O(N)$ space.
- *Radix Sort*: this is a *hash-based* algorithm because it uses keys to sort elements. We can compute the destination of an element instead of having to use comparison operations. It requires an additional $O(N)$ space.

An example is:

```
void ParallelSort()
{ // Sorting in PPL
```

```
    std::vector<double> v(25'000'0);
    std::generate(std::begin(v), std::end(v), std::mt19937(42));

    // General-purpose sort
    concurrency::parallel_sort(std::begin(v), std::end(v));

    auto DESCENDING = [](double d1, double d2) { return (d1 > d2); };
    concurrency::parallel_sort(std::begin(v), std::end(v), DESCENDING);

    // Faster general-purpose sort
    concurrency::parallel_buffered_sort(std::begin(v), std::end(v));

    // Integer key-based sort
    auto metric = [](double d)->std::size_t
             { return std::hash<double>{}(d); };
    concurrency::parallel_radixsort(std::begin(v), std::end(v), metric);
    }
```

For completeness, we show for comparison purposes how a container is sorted using STL algorithms (see Chapters 17 and 18):

```
void SequentialSort()
{
    std::cout << "\n**** Block I, Full sort ****\n";
    std::vector<double> v = { 1.0, -1.0, 2.0, -2.0, 3.0, -3.0 };
    print(v, "input vector: ");

    // Sort using default <
    std::sort(std::begin(v), std::end(v));
    print(v, "sorted vector using operator <: ");

    auto DESCENDING = [](int d1, int d2) { return (d1 > d2); };
    std::sort(std::begin(v), std::end(v), DESCENDING);
    print(v, "sorted vector using lambda op: ");

    // Stable sort: avoids worst-case performance
    std::stable_sort(std::begin(v), std::end(v));
    print(v, "stable sorted vector using operator <: ");

    std::sort(std::begin(v), std::end(v), DESCENDING);
    print(v, "stable sorted vector using lambda op: ");

    auto ASCENDING = [](int d1, int d2) { return (d1 < d2); };
    std::sort(std::begin(v), std::end(v), ASCENDING);
    print(v, "stable sorted vector using lambda op: ");
}
```

30.4 THE AGGREGATION/REDUCTION PATTERN IN PPL

In Section 30.3 we saw how to use parallel techniques that apply the same *independent* operation to many input values. In some cases, however, parallel loops have bodies that do not

execute independently and are in danger of producing data races. For example, let us consider the case of sequential code that sums the elements of a vector in two different ways:

```
long M = 1'000'000'0;
std::vector<int> vec(M, 1);
int startValue = 0;
int sum = std::accumulate(std::begin(vec), std::end(vec),
                          startValue, std::plus<int>());
std::cout << "Sequential sum: " << sum << '\n';

sum = 0;
for (std::size_t i = 0; i < vec.size(); ++i)
{
   sum += vec[i];
}
std::cout << "Sequential sum: " << sum << '\n';
```

In this case the correct output in both cases is produced because the algorithm is sequential and there is no contention for the shared state in the loop body. We now consider what would happen if we naively use PPL:

```
sum = 0;
concurrency::parallel_for(std::size_t(0), vec.size(),
                          [&sum, &vec](std::size_t i)
{
      sum += vec[i];
});
std::cout << "Parallel sum, no mutex: " << sum << '\n';
```

Even though speedup is improved compared with the sequential case (if the array is large enough) the result will be incorrect because the loop contains an unprotected shared variable, namely `sum`. We can remedy this problem by using a mutex but the speedup will be worse than in the sequential case:

```
std::mutex sum_mutex;
sum = 0;
concurrency::parallel_for(std::size_t(0), vec.size(),
                                &sum, &sum_mutex, &vec](std::size_t i)
{
   std::lock_guard<std::mutex> guard(sum_mutex);
   sum += vec[i];
});
std::cout << "Parallel sum, with mutex: " << sum << '\n';
```

So, what can be done to achieve good speedup and avoid data races at the same time? There is hope because the *PPL Aggregation (Reduction) pattern* comes to the rescue. It is an example of the *Master/Worker* pattern discussed in Section 29.10. It combines multiple inputs into a single output efficiently and safely. To this end, PPL has a special template data structure called `combinable<T>` that allows us to create per-task local results in parallel and

then merge these results as a final sequential step. In other words, it performs *lock-free* thread-local subcomputations during the execution of a parallel algorithm. A data structure is said to be lock free if more than one thread must be able to access the data structure concurrently. It can be used instead of a shared variable and it results in improved performance precisely because there is no contention for the shared variable. This class has default and copy constructors as well as a constructor with a callable object as argument. It also has member functions `local()` that returns a reference to the thread-private subcomputation and `combine()` that computes a final value from the set of thread-local computations, as the following code to compute the sum of the elements in a vector shows:

```
// Define the local combiner sum
concurrency::combinable<int> sum3 ([]() { return 0; });

// Compute each local sum
concurrency::parallel_for_each(std::begin(vec),
                               std::end(vec),[&sum3, &vec](std::size_t i)
{
   // Each 'local' is an int
   sum3.local() += vec[i];
});

// Add up local contributions and print it
std::cout << "Parallel PPL (combiner) sum: " <<
                sum3.combine(std::plus<int>()) << '\n';
```

In this case the correct aggregated value will be produced as before. For completeness, we mention how to call other constructors:

```
concurrency::combinable<int> sum4;
concurrency::combinable<int> sum5(sum3);
```

We now discuss the use of PPL aggregation to *non-scalar data types*. In particular, it can be applied to *map/reduce design patterns* (see Miner and Shook, 2012) and data analytics applications.

30.4.1 An Extended Example: Computing Prime Numbers

We discuss how to compute prime numbers and the distribution of primes as a subset of the positive integers. We discuss both sequential and parallel computation using a combination of PPL, STL, the Boost *dynamic bitset* library and the C++ `std::bitset<T>` class. We compare the solutions based on accuracy and performance. We first give a short mathematical introduction to number theory. A *prime number* (or prime) is a natural number that has no positive divisors other than itself and 1. For example, 5 is prime while 8 is not prime because 1, 2, 4 and 8 are its divisors. A number that is not a prime is called a *composite*. The property of being prime is called *primality*. There are several efficient (and not so efficient) algorithms to test for primality. The method that we use here to test the primality of an integer n is called *trial division* and it consists of dividing n by each integer between 1 and less than or equal to the

square root of n. Then n is not a prime if any of these divisions is an integer; otherwise it is a prime. The code is:

```
bool IsPrime(int n)
{

   if (n < 2)
      return false;

   for (int i = 2; i*i <= n; ++i)
   { // simple loop

      if ((n % i) == 0)
      {
         return false;
      }
   }
   return true;
}
```

There are other algorithms to test for primality, for example the *sieve of Eratosthenes* and the *sieve of Atkin*, but a discussion is outside the scope of this book.

We now discuss the algorithm to find the number of primes in a given integral range `[0,n]`. The focus at the moment is more on showing how to use PPL rather than creating highly optimised code. To this end, we use the data structure `std::bitset<n>` to record information, specifically a bit position j is set if the corresponding integer is a prime and j is set to 0 otherwise. The code to generate the thread-local contributions is:

```
// Create the thread-local result localPrimes
std::bitset<n> result;
concurrency::combinable<std::bitset<n>> localPrimes;
concurrency::parallel_for(0, n, [&](int i)
{
   if (IsPrime(i))
      localPrimes.local().set(i);
});
```

The reduce/aggregation part of the algorithm is now:

```
// Merge each thread-local computation into the final result.
localPrimes.combine_each([&](std::bitset<n>& local)
{
     result |= local;
});
```

When generating very many primes we can also give rough estimates on the distribution of primes based on the *Prime Number Theorem*. To this end, let $\pi(x)$ denote the number of primes

that do not exceed x (this is called the *prime-counting function*). Then the *Prime Number Theorem* (Hunter, 1964) states that:

$$\lim_{x \to \infty} \frac{\pi(x)}{\frac{x}{\log x}} = 1. \tag{30.1}$$

We can also compare the processing time using PPL with the corresponding sequential code. An implementation using Boost dynamic bit sets is:

```
const int n = 20;
std::set<int> mySet2;
std::cout << "Add elememts to set ...: " << '\n';

for (std::size_t i = 0; i < n; ++i)
{
   mySet2.insert(i);
}

boost::dynamic_bitset<unsigned int> bs(n);

for (auto i = std::begin(mySet2); i != std::end(mySet2); ++i)
{
   if (IsPrime(*i))
   {
      bs.flip(static_cast<unsigned int>(*i));
   }
}
```

Finally, we can check that the numbers produced are indeed primes since for any two primes their greatest common divisor is one and their least common multiple is their product. To this end, we apply the functionality in Boost *Math Common Factor* library:

```
// Using unsigned long (compile time)
unsigned int p1 = 167; unsigned int p2 = 661;

// GCD(p1,p2) = 1 LCM(p1,p2) = p1*p2 for two primes p1 and p2
std::cout << "GCD and LCM of " << p1 << " and " << p2 << " are "
            << boost::math::gcd<unsigned int>(p1, p2) << " and "
            << boost::math::lcm<unsigned int>(p1, p2) << ", p1*p2: "
            << p1*p2 << '\n';
```

Number theory was once considered an area with no real applications but in recent years prime numbers have been used as the basis for public key cryptography algorithms. They are also used for hash tables and pseudo-random number generators.

30.4.2 An Extended Example: Merging and Filtering Sets

In this section we give an example that can be seen as a simple case of *MapReduce* (see Miner and Shook, 2012). *MapReduce* is a programming model and implementation for processing

and generating large datasets with a parallel, distributed algorithm on a cluster. A *MapReduce* program is composed of a *Map( )* method that performs filtering and sorting and a *Reduce( )* method that performs a summary operation. The model is a specialisation of the *split–apply–combine* strategy for data analysis.

As a first example of this pattern we consider creating a collection of sets, computing their pairwise union and then combining these sets into a single set which is their union. In this way we avoid duplicates and this reduces the amount of data to be stored. First, we create the sets:

```
std::set<int> ms1;
for (int n = 1; n <= 5; ++n) { ms1.insert(n); }
std::set<int> ms2;
for (int n = 1; n <= 10; ++n) { ms2.insert(n); }
std::set<int> ms3;
for (int n = 11; n <= 50; ++n) { ms3.insert(n); }
std::set<int> ms4;
for (int n = 41; n <= 100; ++n) { ms3.insert(n); }

std::vector<std::set<int>> input{ ms1, ms2, ms3, ms4 };
```

The next step is to define the thread-local sets that are computed from the elements of the input collection:

```
// Define and compute the local combiner sum (in parallel)
concurrency::combinable<std::set<int>> localSet([]()
   { return std::set<int>(); });

// Compute each local sum
concurrency::parallel_for(std::size_t(0), input.size(),
   [&](std::size_t i)
{
   // Each 'local' is a set<int>. Take the union of the current set and
   // thread-local set. 'Union' is a handy wrapper function for STL
   // std::set_union()
   Union<int>(input[i], localSet.local());
});
```

Finally, we aggregate the thread-local sets to produce the final result and we are done!

```
std::set<int> result;
// Add up local contributions and print it
// Merge each thread-local computation into the final result.
localSet.combine_each([&](std::set<int>& local)
{
   Union<int>(local, result);
});

// Print the result
for (const auto& elem : result)
{
   std::cout << elem << ",";
}
```

For completeness we show the code for `Union()`. It is a wrapper for the corresponding STL algorithm:

```
template <typename T>
      void Union(const std::set<int>& s1, std::set<int>& s2)
{ // Wrapper function for STL std::set_union()

      std::set<int>::iterator it = s2.begin();
      std::insert_iterator<std::set<int> > insertiter(s2, it);
      std::set_union(s1.begin(), s1.end(), s2.begin(), s2.end(),
                     insertiter);
}
```

This is a relatively simple example but it can be generalised and solved in other ways, and it is a good example to show how to apply PPL to more complex applications.

30.5 CONCURRENT CONTAINERS

PPL supports a number of containers and objects that provide thread-safe access to their state (in the `concurrency` namespace). These *concurrent containers* provide concurrency safe access to the most important operations of a container. The functionality of these containers resembles that in the STL. We use concurrent containers when we have parallel code that requires both *concurrency safe read and write access* to the same container.

There are some differences between PPL containers and their STL equivalents and a full discussion is outside the scope of this book. We note that converting code from STL to PPL can cause the nature of the application to change.

- *Concurrent vector*: provides concurrency safe append and access to its elements. Append operations do not invalidate existing pointers or iterators. Iterator access and traversal operations are also concurrency safe. Concurrent vectors do not use move semantics when appending nor do they store their elements contiguously in memory.
- *Concurrent queue*: provides concurrency safe *enqueue* and *dequeue* operations. This means that we can use this container as a *synchronising queue* in the *Producer–Consumer* pattern with no additional code having to be written (in contrast to doing the same exercise using threads in C++). The methods `empty`, `push` and `try_pop` are concurrency safe, while iterators are not concurrency safe.
- *Concurrent unordered map and multimap*: unordered STL containers are new in C++ but they are not concurrency safe. PPL unordered maps and multimaps enable concurrency safe insert and element-access operations.
- *Concurrent unordered set and multiset*.

It is safe to perform insertions in parallel with these containers.

We give an example for motivational purposes and reference. In this case we sum the elements of a vector:

```
using Vector = concurrency::concurrent_vector<int>;
//using Vector =std::vector<int>;
```

```
void InsertValues(Vector& v, std::size_t L, std::size_t U)
{

   for (std::size_t i = L; i < U; ++i)
   {
      std::this_thread::sleep_for(std::chrono::milliseconds(2000));
      v.push_back(i);
   }
}

int main()
{
   Vector v;

   std::thread t1(InsertValues, std::ref(v), 0, 50);
   std::thread t2(InsertValues, std::ref(v), 50, 100);
   std::thread t3(InsertValues, std::ref(v), 100, 201);

   t1.join(); t2.join(); t3.join();
   std::cout << "Size: " << v.size() << '\n';

   // Combine the results.
   concurrency::combinable<int> sums;

   for (auto i = std::begin(v); i != std::end(v); ++i)
   {
      sums.local() += *i;
   }

   // Sum of 1st n integers = n(n+1)/2
   std::cout << "sum = " << sums.combine(std::plus<int>()) << '\n';
}
```

A *concurrent object* is shared concurrently among components. A process that computes the state of a concurrent object in parallel produces the same result as another process that computes the same state serially. The `concurrency::combinable` class (introduced in Sections 30.4.1 and 30.4.2) is an example of a concurrent object type. The *combinable class* lets you perform computations in parallel, and then combine those computations into a final result. We use concurrent objects when we would otherwise use a synchronisation mechanism, for example a mutex to synchronise access to a shared variable or resource.

30.6 AN INTRODUCTION TO THE ASYNCHRONOUS AGENTS LIBRARY AND EVENT-BASED SYSTEMS

In this section we give a short overview of a model of concurrency that can be seen as an alternative to models based on shared state. The model is based on dataflow between *independent components* that communicate with one another by sending and receiving messages. An underlying communication pattern is called *implicit invocation* (also known as *notification* and *publish-subscribe* idiom (see GOF, 1995)). In this case we do not invoke a procedure directly

but rather a component announces or broadcasts one or more events. Other components in the system can *register* (*subscribe to*) interest in an event by associating a procedure with it. When an event is announced, the system itself invokes all of the procedures that have been registered for the event. Thus, the event announcement 'implicitly' causes the invocation of procedures in other modules (Shaw and Garlan, 1996). In short, we speak of *event-based* or *event-driven systems* and these systems have their origins in packet-switched networks. In particular, there is no global clock and models of time have been motivated by quantum mechanics and special relativity. Each agent has its own local time.

There are many ways to implement event-based systems. Some examples are:

- Using the object-oriented *Observer* pattern (GOF, 1995).
- Using the Boost *signals2* library (*signals* and *slots*).
- Using the *Asynchronous Agents Library* (as discussed in this section).

A discussion of the *Observer* pattern and Boost *signals2* is given in Demming and Duffy (2010).

The *Asynchronous Agents Library* (or *Agents Library* for short) provides both an *actor-based* programming model (see Agha, 1990) and *message passing interfaces* for coarse-grained dataflow and pipelining applications. Asynchronous agents enable applications to make use of latency by performing work while other components wait for data. This feature reduces the deadlock that we experience with threads.

30.6.1 *Agents Library* Overview

An *asynchronous agent* (or *agent*) is an application component that communicates with other agents to solve larger computing tasks. The *Concurrency Runtime* task scheduler provides an efficient mechanism to enable agents to block and yield *cooperatively* without requiring pre-emption.

The lifecycle of an agent is shown in Figure 30.1. An agent finds itself during its lifetime in various states:

- *created*: the agent has been scheduled for execution.
- *runnable*: the runtime is scheduling the agent for execution.
- *started*: the agent has started and is running.
- *done*: the agent has finished processing.
- *cancelled*: the agent has been cancelled before it entered the started state.

The *Asynchronous Agents Library* includes the following types of messaging blocks which are useful in a variety of situations (it is also possible to implement user-defined messaging blocks):

- *unbounded_buffer*: a concurrent queue of unbounded size.
- *overwrite_buffer*: a single value that can be updated many times.
- *single_assignment*: a single value that can be set just once. Further updates are ignored.
- *call*: a function that is invoked whenever a value is added to the messaging block. It is a *callback*.
- *transformer*: a function that is invoked whenever a value is added to the messaging block; the function's return value is added to an output messaging block.

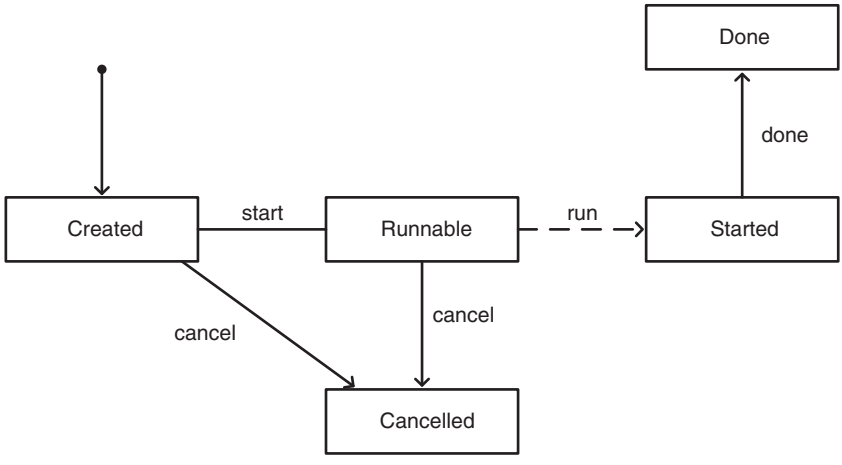


FIGURE 30.1 Agent lifecycle

- *choice*: selects a message from a set of sources.
- *join*: waits for more than one source before proceeding.
- *multitype_join*: same as join, except that inputs may have multiple message types.
- *timer*: produces messages based on time intervals.

There is more functionality in this library than described here and unfortunately a discussion is outside the scope of the current book. We focus on a number of examples to motivate how to use the library. In Chapter 32 we shall use the library in a Monte Carlo simulation for option pricing.

30.6.2 Initial Examples and Essential Syntax

In this section we give some examples of using message block types. We take the simple case of sending and receiving three characters that we conveniently encapsulate in a number of functions to reduce code duplication:

```
template <typename Buffer>
   void Send(Buffer& buffer)
{
   concurrency::send(buffer, 'a');
   concurrency::send(buffer, 'b');
   concurrency::send(buffer, 'c');
}

template <typename Buffer>
   void Receive(Buffer& buffer)
{
   std::cout << concurrency::receive(buffer) << ',';
   std::cout << concurrency::receive(buffer) << ',';
   std::cout << concurrency::receive(buffer) << '\n';
}
```

```
template <typename Buffer>
   void SendAndReceive(Buffer& buffer)
{
   Send(buffer);
   Receive(buffer);
}
```

We now show the use of the three most common message block types:

```
// A message buffer that is shared by the agents.
concurrency::unbounded_buffer<char> buffer1;
concurrency::overwrite_buffer<char> buffer2;
concurrency::single_assignment<char> buffer3;

SendAndReceive<concurrency::unbounded_buffer<char>>(buffer1);
// a,b,c

SendAndReceive<concurrency::overwrite_buffer<char>>(buffer2);
// c,c,c

SendAndReceive<concurrency::single_assignment<char>>(buffer3);
// a,a,a
```

The next example is to show how to create a callback using the `call` class. The callback is executed when an event fires:

```
// 'call' class. Perform a work function when data is received
// 'event' is a synchronisation object for flow of execution
concurrency::event receivedAll;

int receiveCount = 0; int maxReceiveCount = 3;
concurrency::call<char> callback = [&](char c)
{
   std::cout << c << ",";
   if (++receiveCount == maxReceiveCount)
   {
      receivedAll.set(); // set event to signalled state
   }
};

Send(callback);     // Notice that callback is a message buffer
// Wait for call object to process all items
receivedAll.wait();  // a,b,c
```

The next example shows how to transform input to output (using the `transformer` class, the input and output can be of different types; in this case input is `char` and output is `int`):

```
// Transformer on different input and output types
std::cout << '\n';
concurrency::transformer<char,int> converter = [](char c)
```

```
{
   return int(c); // simple test
};

SendAndReceive(converter);         // Converter is a message buffer

long relation (long n)
{
   if (n < 2) return n;

   return relation(n - 1) + relation(n - 2);
}
```

We conclude this section by introducing the `choice` and `join` classes. Both options are concerned with the processing of a set of messages. In the first case the first function to finish is selected (select among competing tasks) while in the second case all functions are selected:

```
// Choice: select the first available messages from a set of sources
concurrency::overwrite_buffer<long> rel1;
concurrency::overwrite_buffer<long> rel2;
concurrency::single_assignment<long> rel3;

auto selectOne = concurrency::make_choice(&rel1, &rel2, &rel3);

concurrency::parallel_invoke(
   [&rel1] {concurrency::send(rel1, relation(22)); },
   [&rel2] {concurrency::send(rel2, relation(14)); },
   [&rel3] {concurrency::send(rel3, relation(13)); }
);

// See who responds first
switch (concurrency::receive(selectOne))
{
   case 0:
      std::cout << "rel1 " << concurrency::receive(rel1) << '\n';
      break;
   case 1:
      std::cout << "rel2 " << concurrency::receive(rel2) << '\n';
      break;
   case 2:
      std::cout << "rel3 " << concurrency::receive(rel3) << '\n';
      break;
   default:
      std::cout << "oops\n";
      break;
}

// Run multiple tasks
concurrency::single_assignment<long> relA;
concurrency::single_assignment<long> relB;
concurrency::single_assignment<long> relC;

auto joinAll = concurrency::make_join(&relA, &relB, &relC);
```

```
concurrency::parallel_invoke(
                          [&relA] {concurrency::send(relA, relation(6)); },
                          [&relB] {concurrency::send(relB, relation(7)); },
                          [&relC] {concurrency::send(relC, relation(10)); }
);

auto result = concurrency::receive(joinAll);

std::cout << "relA " << std::get<0>(result)
           << ", relB " << std::get<1>(result)
           << ", relC " << std::get<2>(result) <<'\n';
```

The output from the complete code in this section is:

```
a,b,c
c,c,c
a,a,a
a,b,c,
97,98,99
rel3 233
relA 8, relB 13, relC 55
```

30.6.3 Simulating Stock Quotes Work Flow

The *Agents* library can be used to implement the *Producer–Consumer* pattern more easily than is possible by using threads in combination with synchronising queues (see Demming and Duffy, 2010). In the current case we do not have to worry about locking shared data (because there is no shared data) and agents communicate solely using message passing. Agents' state is said to be *isolated*.

The following example simulates a stock quotes application. Stock values are read from an external device and each item in the array is sent to a consumer to be displayed, for example in Excel, on the console or some other output device. Both agent classes are derived from the abstract base class `concurrency::agent` having the pure virtual member function `run()`. Derived classes must implement this function, of course. We use `ITarget` and `ISource` to denote the *communication endpoints* for producer and consumer, respectively. The producer class is:

```
class Producer : public concurrency::agent
{
private:
   // The target buffer to write to.
   concurrency::ITarget<double>& _target;
public:
   explicit Producer(concurrency::ITarget<double>& target)
      : _target(target) {}

   void run()
   {
      // For illustration, create a predefined array of stock quotes.
      // A real-world application reads these quotes from an external
      // source,
```

```
      // such as a network connection or a database.
      std::array<double, 6> quotes =
                        { 24.44, 24.65, 24.99, 23.76, 22.30, 25.89 };

      // Send each quote to the target buffer.
      std::for_each(begin(quotes), end(quotes), [&](double quote)
      {

         send(_target, quote);

         // Pause before sending the next quote.
         concurrency::wait(20);
      });
      // Send a negative (sentinel) value to indicate the end of
      // processing.
      concurrency::send(_target, -1.0);

      // Set the agent to the finished state.
      done();
   }
};
```

The consumer class is:

```
class Consumer : public concurrency::agent
{
private:
   // The source buffer to read from.
   concurrency::ISource<double>& _source;
public:
   explicit Consumer(concurrency::ISource<double>& source)
      : _source(source){}

   void run()
   {
      // Read quotes from the source buffer until we receive
      // a negative value.
      double quote;

      // Poll to see if any new values have arrrived
      while ((quote = concurrency::receive(_source)) >= 0.0)
      {
         // Print the quote.
         std::cout.setf(std::ios::fixed);
         std::cout.precision(2);
         std::cout << "Current quote is " << quote << '\n';

         // Pause before reading the next quote.
         concurrency::wait(10);
      }
```

```
        // Set the agent to the finished state.
        done();
    }

};
```

Finally, the test program is:

```
int main()
{
    // A message buffer that is shared by the agents.
    concurrency::overwrite_buffer<double> buffer;

    // Create and start the producer and consumer agents.
    Producer producer(buffer);
    Consumer consumer(buffer);
    producer.start();
    consumer.start();

    // Wait for the agents to finish.
    concurrency::agent::wait(&producer);
    concurrency::agent::wait(&consumer);
}
```

Notice the presence of the *sentinel value* in the producer.

30.6.4 Monte Carlo Option Pricing Using Agents

This is an example that we shall discuss in Chapter 32. It is a producer–consumer application in which the producer is a path evolver based on the Euler finite difference method while the consumers are one or more option pricers.

30.6.5 Conclusions and Epilogue

We decided to introduce the *Agents* library as it fills a gap in the design landscape. In this case we are interested in event-based distributed systems. From the examples already given we can see that the design philosophy is not incompatible with the design approach taken in Chapter 9 and we shall show why in Chapter 32. The library can be used in applications that apply the traditional and rigid *Observer* pattern or the more flexible Boost *signals2* library. The major difference is that we do not call object methods; agents send and receive data using message passing.

We conclude this section with a list of features to take into account when programming with agents and tips to improve efficiency:

- Asynchronous agents are effective because they isolate their internal state from other agents. This helps reduce contention on shared memory. *State isolation* reduces the chance of deadlock and race conditions because components do not have to synchronise access to shared data.

- Some buffer types (for example, `concurrency::unbounded_buffer<char>`) can hold an unlimited number of messages, at least in theory. A consequence is that an application can enter a low-memory or out-of-memory state if the producer sends messages to a data pipeline faster than the consumer can process them. We can limit the number of active messages that are concurrently active in the data pipeline by the introduction of a *throttling mechanism* (for example, a *semaphore*).
- The work in a data pipeline application should be *coarse grained*. This is because message passing involves more overheads than *fine-grained* work (which PPL task groups are more suitable for). We can combine the two approaches by defining a coarse-grained data network that uses fine-grained parallelism at (and within) each processing stage.
- Avoid passing large message payloads by value. For example, `overwrite_buffer<char>` offers a copy of every message that it receives to each of its targets. We can use pointers or references to improve memory performance when we pass messages that have a large payload.
- In some cases it is not clear when or where to deallocate memory as messages travel in a data pipeline. In particular, we should use `std::shared_ptr`.
- Each actor runs in a single thread by default. It executes each message sequentially.

We create a small Monte Carlo framework in Section 32.5 using agents.

30.7 A DESIGN PLAN TO IMPLEMENT A FRAMEWORK USING MESSAGE PASSING AND OTHER APPROACHES

The approach that we adopt in this book (in particular, Chapter 9) is to use system decomposition techniques to break the problem into loosely coupled *logical components* whose black-box interface specifications we wish to identify as early as possible. In a sense we wish to postpone having to make implementation decisions for as long as possible. Having created a *logical model* we can then investigate the different scenarios for a *detailed design* of the problem. Some choices and special cases are:

- S1: The traditional object-oriented solution based on inheritance and subtype polymorphism.
- S2: The same approach as in case S1 except we use static polymorphism and the *Curiously Recurring Template Pattern* (CRTP).
- S3: Using universal function wrappers (C++11 `std::function`) to implement polymorphic functions without the need to create class hierarchies.
- S4: Creating a parallel solution using threads, tasks or message passing (Campbell and Miller, 2011; Williams, 2012).
- S5: Using the *actor model* and message passing.
- S6: A combination of the cases S1 to S4.

In all of the above cases (with the exception of S5) the connections are from object to object (we call this an *object connection architecture*). In other words, one object in the network calls a member function of some other object. A major disadvantage is that modules (for example, classes and class hierarchies) must be built before the architecture is defined (Leavens and Sitarman, 2000). Another major problem is that object-oriented interfaces

specify only the features *provided* by modules. An *Architectural Definition Language* (ADL) augments object-oriented technology by defining those features that modules *require* from their environment. An ADL interface expresses the dependencies that a conforming module has on it environment. We say that the interface is *contextualised*. In simple terms, we model an object's *events* which are supported in the Boost *signals2* library. To motivate required interfaces, we can create a *signal* with a given signature and attach signature-compatible *slots* to it. We take an example of a class that models sensors. It has an embedded signal which is triggered when an event in the sensor takes place. Then all connected slots will be updated. We use a *connection object* because it knows which signals and slots are connected. The class containing the signal is:

```
#include <iostream>
#include <boost/signals2.hpp>
#include <cmath>


template <typename T>
   class Sensor
{ // Subject only, generates events and simulates hardware

private:
   boost::signals2::signal<void (const T& t)> sig;
   boost::signals2::connection conn;
public:
   Sensor():   sig() {}

   template <typename Slot>
      void connect (Slot& sensor)
   {
      conn = sig.connect(std::ref(sensor));
   }

   template <typename Slot>
      void disconnect(Slot& sensor)
   {
      conn.disconnect();
   }

   void changeEvent(const T& t)
   {
      T t2 = std::exp(t);
      sig(t2);
   }

   ~Sensor() { std::cout << "Sensor bye\n"; }

};
```

This class plays the role of the *subject* in the *Observer* pattern (see GOF, 1995) while slots correspond to *observers*.

We show how event notification works in Boost. First, we create a signal and we connect two slots to it:

```
using Hardware = Sensor<double>;
Hardware sensor;                    // Subject/signal

auto slot
   = [](const double& d) { std::cout << "First: " << d << '\n'; };
auto slot2
   = [](const double& d) { std::cout << "Second: " << d/2 <<'\n'; };

sensor.connect(slot);
sensor.connect(slot2);
```

We note that the signatures of signals and slots must be the same for event notification to take place. When an external event takes place we say that the signal *emits* and it notifies its connected slots:

```
// Change propagation mechanism
double d = 5.0;
sensor.changeEvent(d);
```

Finally, we can disconnect the slots from the signal as follows:

```
// Break off the connection
sensor.disconnect(slot);
sensor.disconnect(slot2);
```

We are unable to discuss Boost *signals2* in this book due to space restrictions. We have included the above example to show that there is more to life than object-oriented technology. We are interested not only in *provided* interfaces but also in *required* interfaces (see Figure 30.2), the latter being realised by signals. In this sense the concepts are closely related to events and control flow between objects. Data can be exchanged between objects in a number of ways (this aspect is related to control flow, communication and synchronisation issues as discussed in Shaw and Garlan, 1996, p. 108), depending on how modules communicate. There are several options:

- C1 (*Events*): There is no shared state; all communication relies on events (for example, using signals or message passing).

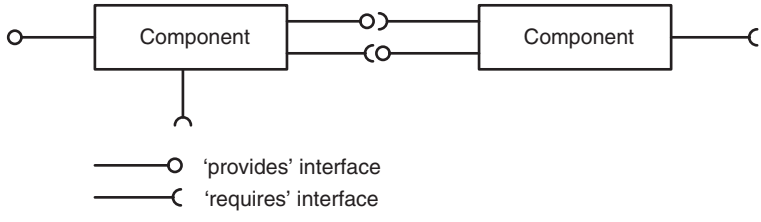


FIGURE 30.2 Augmented object interface types

- C2 (*Pure state*): Communication is solely by means of shared state; the *caller* (*receiver of data*) must repeatedly inspect the state variables to detect changes. In other words, the recipient must *poll* the shared state for changes.
- C3 (*State with hints*): Communication is by means of shared state but the recipient is actively informed of changes using an event mechanism. This means that polling of the state is not required. The hints are efficiency hints rather than essential information and the recipient can ignore the hints and reconstruct all necessary information from the shared state.
- C4 (*State and events*): Both shared state and events are used: events are needed because they contain essential information not available from state monitoring.

These choices need to be resolved at detailed design level. In this sense the activities in this section augment the system decomposition methods of Chapter 9.

We conclude this section with a discussion of how to assimilate the *Agents* library into the software process in Chapter 9. The library is based on a dataflow model and less on an event model. What becomes important is to determine and specify the services that agents deliver and expect (as discussed in Section 9.3.3). In other words, we must know the type of data that agents send and receive. We will need to know for which data an agent is a source and for which data an agent is a target. Finally, we need to design the use cases for starting and shutting down a system

One final remark, we discuss two different styles of programming (DeRemer and Kron, 1975); *programming in the large* can involve programming by larger groups of people. Emphasis is placed on partitioning work into modules with precisely specified interactions. Programming in the large requires abstraction-creating skills. Until a module becomes implemented it remains an abstraction. Taken together, the abstractions should create an architecture that are unlikely to need change. They should define interactions that have precision and demonstrable correctness.

On the other hand, *programming in the small* describes the activity of writing small programs. They are small in terms of source code size, are easy to specify, quick to code and typically perform one or more tasks well. Programming in the small can involve programming by individuals or small groups over short time periods and may involve less formal practices (for instance, less emphasis on documentation or testing), tools and programming languages. Programming in the small can also describe an approach to creating a prototype software or where rapid application development is more important than stability or correctness.

30.8 SUMMARY AND CONCLUSIONS

This chapter is a continuation of Chapter 29 in which we introduce *Parallel Patterns Library* (*PPL*) that allows us to create task-based concurrent software systems. This library supports a range of parallel design patterns such as aggregation/reduction, task groups and pipeline models. We also discussed *work-partitioning algorithms* to help improve system performance. We introduced the *Asynchronous Agents Library* that supports the creation of actor-based applications based on message passing.

We shall give some applications of *C++ Concurrency*, PPL and the *Asynchronous Agents Library* to option pricing using the Monte Carlo method in Chapter 32.

30.9 EXERCISES AND PROJECTS

1. (Quiz)

The questions in this exercise are best answered by writing code, testing assumptions and examining output. To this end, we give a number of assertions and questions and you need to determine if they are true or false:

a) Forgetting to call `task_group::wait` causes a runtime error.

b) The body of a *Parallel Invoke* may contain a combination of lambda functions and stored lambda functions.

c) Only one exception is visible when using *Parallel Invoke* even if multiple exceptions occur.

d) The order of execution of tasks in a task group is non-deterministic (there is no prescribed order of execution).

e) Task coordination is achieved using *cooperative blocking*.

2. (Lattice Models Revisited)

We examine the models, designs and code from Chapters 11 and 12 (that dealt with applying the binomial method to one-factor European and American option pricing problems). We focus here on using parallel C++ to improve the *speedup* of the code. Some of the forces and issues are:

a) Is it worthwhile parallelising the code? For example, the program's *serial fraction* may be more than 10% and Amdahl's law will give a pessimistic upper bound on the speedup on a multicore CPU.

b) Parallel code may give different results compared to serial code because of non-deterministic round-off behaviour or worse, the parallel code is not thread safe. A program that yields identical results (except for round-off errors) when executed with one thread or multiple threads is said to be *sequentially* (*serially*) equivalent.

c) How far do we need to restructure a serial program to gain parallel performance? The answer to this question becomes even more difficult when the code is undocumented and unstructured. In the common case of loops, for example, we can turn a sequential program into a parallel program by a series of transformations on the loops. In particular, we may wish to eliminate *loop-carried dependencies* and leave the overall program semantics unchanged. These are called *semantically neutral transformations*. It is important that the transformations do not 'break' the program in the sense that it starts producing different output.

d) Just because we are using parallel C++ does not automatically guarantee good speedup. The problem must be computationally intensive enough to warrant it.

e) Deciding on whether to use threads or tasks and then which parallel design pattern(s) is most suitable for the job at hand.

Answer the following questions based on Chapters 11 and 12:

1. Examine the code in Section 11.4. Determine if it is worth parallelising and if so how you would go about the job. In particular, re-engineer the code and create a task graph; implement it using a combination of futures and parallel loops.

2. In Section 11.8 we discuss the merging of two lattices. Determine how to parallelise the code and then generalise it to merging four lattices.

3. In Section 11.4.2 we discuss lattice creation. Can the corresponding code be parallelised? If yes, how can it be effected and if not why not?

4. In Section 12.3 we have given a formula to compute option prices using *Bernstein polynomials*. Determine under which circumstances it is advantageous to write parallel code

for equations (12.5) and (12.6). Consider the use of reduction variables. Consider how to use tasks and task groups to price the following option set: {plain call, plain put, cash-or-nothing, supershare call, supershare put}. You can use lambda functions and captured variables to implement the payoff functions, for example:

```
// Supershare option (Haug 2007 page 176)
double KL = 90.0;
double KH = 110.0;

auto Payoff = [&](double S)-> double
   {if (S < KH && S >= KL) return S/KL; return 0.0; };
```

5. Write parallel code to compute vega, theta and rho as given in equation (12.14). It might be a good idea to first draw a task graph to find sources of potential parallelism. Then decide on which parallel design pattern to apply. Measure speedup. Are the sequential and parallel results the same?

When we say 'parallel C++' we refer to both the functionality in *C++ Concurrency* and in PPL.

3. (Parallelising Finite Difference Schemes)

The objective of this exercise is to parallelise the process of comparing the accuracy of finite difference schemes (as discussed in Chapters 20 to 22) using task groups and `parallel_for`. In order to scope the problem we compare the option price, delta and gamma produced by the Crank–Nicolson and ADE schemes with regard to accuracy. Some requirements are:

- R1: Compute option price, delta and gamma by the Crank–Nicolson and ADE schemes.
- R2: Produce an error report showing how the two methods differ from each other with respect to accuracy.
- R3: In this version, use a fixed set of option data.
- R4: Compare the schemes for a range of step sizes in the underlying space and in time directions.

Answer the following questions:

a) Create a context diagram for this problem and describe the data flow from source input data to desired output.

b) Use the results in part a) to find potential concurrency and then create a task dependency graph consisting of tasks and the data flow between them.

c) Implement and test the task dependency graph.

d) We relax the requirement R3 in order to support randomly generated input data. What impact has this new requirement on the task dependency graph and its implementation?

4. (Parallel Aggregation)

PPL supports `concurrency::parallel_reduce` that computes the sum of all elements in a given range by computing successive partial sums. It can also compute the result of successive partial results obtained from using an arbitrary specified binary operator. It is semantically similar to `std::accumulate` but it requires the binary operation to be *associative* and an *identity value* is needed instead of an initial value. It has three overloaded versions based on the number of input arguments and we discuss a specific one. The parameters are:

- The begin and end iterators (STL style) of the range to be reduced, in other words the iteration bounds.

- The value of the reduction's identity element (0 for addition, 1 for multiplication; for aggregate set union the identity element is the empty set).
- The function object that can be applied on a subrange of an iterator range to produce a local partial aggregation. The return value is the local partial result from the first phase of the *Parallel Aggregation* pattern.
- The reduction function that combines the partial results that each subrange has computed.

Answer the following questions:

a) Apply this algorithm to the examples in Section 30.4 (for example) and compare it with other aggregation algorithms in that section with regards to efficiency and understandability.

b) Determine what the following piece of code is doing:

```
std::vector<long> vec2(2000, 4);
int sumV = 0;
vec2[600] = 4;
auto d = concurrency::parallel_reduce(std::begin(vec2),
std::end(vec2), sumV); std::cout << "Parallel_reduce PPL sum : "
                                  << d << '\n';
```

How does its performance compare to other aggregation algorithms?

c) Compare the performance of PPL aggregation algorithms with that of *OpenMP*:

```
// OpenMP and reduction variable
M = 8;
std::vector<int> vecA(M, 2);
int sum2 = 0;
int prod2 = 1;
int i;
#pragma omp parallel for private(i), reduction (+ : sum2),
reduction (* : prod2)
for (long i = 0; i < vecA.size(); ++i)
{ // vec is some vector

   sum2 += vecA[i];
   prod2 *= vecA[i];
}
std::cout << "Parallel OpenMP sum and product: "
           << sum2 << ", " << prod2 << '\n';
```

In the case of floating-point data the order in which thread-specific values are combined is non-deterministic in general and hence the final result may not be precisely the same as when the reduction is performed serially. This is caused by the fact that floating-point operations can introduce *roundoff errors*. This could be a cause of concern in computational finance applications where reproducible and accurate results are needed. We may even decide not to parallelise the specific loop if the error resulting from roundoff becomes unacceptable.

Investigate this anomaly using PPL and OpenMP. Take examples to exhibit the effect of roundoff error.

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 31

Monte Carlo Simulation, Part I

Analyse a little, design a little, program a little.

—Grady Booch

Why are software process models important? Primarily because they provide guidance on the order (phases, increments, prototypes, validation tasks) in which a project should carry out its major tasks. Many software projects have come to grief because they pursued their various development and evolution phases in the wrong order.

—Barry Boehm

31.1 INTRODUCTION AND OBJECTIVES

In this chapter and in Chapter 32 we initiate a project to design and implement an option pricing software framework using the Monte Carlo method. In order to reduce the scope and keep the presentation manageable we focus on the pricing of a range of one-factor options whose underlying variable is described by stochastic differential equations (SDEs). In general, analytical solutions are not available and we resort to numerical methods, in particular the finite difference method (FDM).

Some of the goals of this chapter are:

- To develop a process to design the software framework that produces a series of software prototypes with each prototype containing more functionality than its predecessor. In this case we attempt to simulate the lifecycle of real software projects (for example, market-driven products or in-house development projects).
- To show how the techniques that we introduced in Chapter 9 can be applied to the design of flexible and extendible code. In this chapter we show how to achieve this goal in a bottom-up manner by creating a series of software prototypes while in Chapter 32 we formalise the design when we present a *predefined architecture* that can be instantiated to reproduce the results in this chapter as a special case.
- Rather than using only the object-oriented style to design the software framework (as in Duffy and Kienitz, 2009), we augment this style with generic and functional styles that are now supported in C++.

We introduce and describe the following software prototypes:

- P1: A *monolithic program* ('ball of mud') that prices one-factor plain call and put options using the explicit Euler method. This prototype functions as a requirements document and *feature list* (in the sense of the discussion in Section 9.3.2) that can be extended in later prototypes.
- P2: A solution based on policy-based design (PBD) and templates as discussed in Section 9.3.4.
- P3: This is a variation of solution P2 in which class hierarchies based on subtype/dynamic polymorphism are replaced by the *Curiously Recurring Template Pattern* (CRTP). CRTP is described in Chapter 21 (in particular, Section 21.7.2) where we apply it to PDE models.
- P4: A design based on the approach that we introduced in Section 6.8 that will be presented in Chapter 32. In this case we try to play down the role of classes and objects and focus more on system decomposition and the implementation of inter-component interface contracts using universal function wrappers (the class `std::function`).

There is no unique way to choose which prototypes to develop or in which order. However, there is a wrong way and a right way. In practice, customer and market requirements will determine how a software project evolves. We now discuss this topic.

31.1.1 Software Product and Process Management

From a software process viewpoint, the approach in this chapter is similar to that in the *Spiral Model* (Boehm, 1986). The spiral model is a risk-driven process for software projects. Based on the unique risk patterns of a given project, the spiral model guides a team to adopt elements of one or more process models, such as incremental, waterfall or evolutionary prototyping. The basic principles are (www.wikipedia.org):

a) Focus is on risk assessment and on minimising project risk by breaking a project into smaller segments and providing more ease of change during the development process, as well as providing the opportunity to evaluate risks and weigh consideration of project continuation throughout the life cycle.
b) Each cycle involves a progression through the same sequence of steps, for each part of the product and for each of its levels of elaboration, from an overall concept-of-operation document down to the coding of each individual program.
c) Each trip around the spiral traverses four basic quadrants: (1) determine objectives, alternatives, and constraints of the iteration, (2) evaluate alternatives; identify and resolve risks, (3) develop and verify deliverables from the iteration and (4) plan the next iteration. We begin each cycle with an identification of stakeholders and their "win conditions", and we end each cycle with review and commitment.

We see Boehm's *Spiral Model* as complementing the approach taken in Chapter 9. It is a *process driver* that helps deliver more functionality after the completion of each cycle. It is not the only model around and it may not be the best one in all cases. A discussion of the relative merits of the competing software development processes (such as *Agile* and *Waterfall*) is outside the scope of this chapter. See Exercise 1 where we discuss the spiral model from a project management viewpoint.

31.1.2 Who can Benefit from this Chapter?

We discuss a number of topics that we integrate to form readable, efficient and maintainable code. Instead of creating small code snippets to show how to implement an algorithm we try to apply the steps that György Pólya describes when solving mathematical problems, as already discussed in Chapter 9 (Pólya, 1990):

1. First, you have to *understand the problem*.
2. After understanding, then *make a plan*.
3. *Carry out the plan*.
4. *Look back* at your work. How could it be better?

In the current context we see step 1 as the preliminary research work that we need to carry out (in particular, studying the financial and mathematical literature), step 2 can be translated into the design techniques that we discussed in Chapter 9. Step 3 constitutes implementing the *design blueprints* from step 2. Finally, step 4 is concerned with debugging, acceptance testing and preparing for the next cycle in the *Spiral Model*, for example. Unfortunately, some software systems begin life at step 3 in a rush to produce results without first understanding the problem. The main disadvantage is that the resulting software tends to become unmaintainable and we end up with *spaghetti code* or even worse, a 'ball of mud'.

Given the above tips and guidelines we mention several reader groups for whom this book and chapter are relevant:

- Quant developers who work in computational finance and who wish to write maintainable programs. This book provides a defined process to help them achieve this end.
- MSc and MFE students who wish to work in finance. C++ is a de-facto standard in computational finance and is seen as an important skill to have.
- Software developers who do not necessarily have a background in computational finance and who would like to learn more about the subject by studying the C++ code in this book.

In general, the best advice is '*analyse a little, design a little, program a little*', as we have already mentioned.

31.2 THE BOOST PARAMETERS LIBRARY FOR THE IMPATIENT

We give a short introduction to the topic of defining parameters that are used as input arguments to functions. This is a common use case and we can choose between calling a function with a list of arguments or by first aggregating the arguments somehow into a structure and then using this structure as input to the function. The various options are:

- A1: 'Flat' list of uncoupled arguments. This option could include the case of variadic parameters.
- A2: Place the arguments in a collection. The collection could be fixed size (for example, `std::array`) or dynamic (for example, `std::vector`).
- A3: Place arguments in a C++ `struct`.
- A4: Place arguments in a C++ `std::tuple`.
- A5: Use the Boost *Parameter* library.

The ultimate choice in a given application depends on the requirements, for example:

- R1: Efficiency considerations, how expensive is it to pack and unpack function arguments?
- R2: Is the data homogeneous or heterogeneous?
- R3: Do nested data structures need to be supported?
- R4: Supporting configuration data and run-time data.

We discuss each of these options in this section, starting with option A5. Boost *Parameter* supports *named arguments*. Since named arguments can be passed in any order, they can be employed when a function or template has more than one parameter with a useful default value. This library associates each parameter name with a keyword. Now users can identify arguments by name rather than by position.

In this section we take the initial example of a class that models points in three-dimensional space:

```
struct Point
{ // 3d points

   double x;
   double y;
   double z;

   // ...
};
```

The first step is to define the keywords that will be used in client code:

```
namespace PointParams
{
   BOOST_PARAMETER_KEYWORD(Tag, X)
   BOOST_PARAMETER_KEYWORD(Tag, Y)
   BOOST_PARAMETER_KEYWORD(Tag, Z)
}
```

We need to create a special constructor as follows:

```
template <typename ArgPack> Point(const ArgPack& args)
{ // Boost Parameter

   x = args[PointParams::X];
   y = args[PointParams::Y];
   z = args[PointParams::Z];
}
```

Finally, we can create points as follows:

```
Point p1((PointParams::X = 1.0, PointParams::Y = 2.0,
            PointParams::Z = 2.0));
p1.print();
```

This simple idea can be extended to examples in computational finance.

31.2.1 Other Ways to Initialise Data

We show the code that implements options A1 to A5 in the case of the class that models three-dimensional points. The complete code is easy to follow and is given by:

```
#include <boost/parameter.hpp>
#include <iostream>
#include <tuple>
#include <array>
#include <vector>

namespace PointParams
{
   BOOST_PARAMETER_KEYWORD(Tag, X)
   BOOST_PARAMETER_KEYWORD(Tag, Y)
   BOOST_PARAMETER_KEYWORD(Tag, Z)
}


struct Point
{ // 3d points

   double x;
   double y;
   double z;

   Point() : x(0.0), y(0.0), z(0.0) {}

   explicit constexpr Point(double xVal, double yVal, double zVal)
      : x(xVal), y(yVal), z(zVal) {}

   explicit Point(std::tuple<double, double, double>& tup)
   {
      std::tie(x,y,z) = tup;
   }

   explicit Point(std::array<double, 3>& arr)
   {
      x = arr[0]; y = arr[1]; z = arr[2];
   }

   explicit Point(std::vector<double>& arr)
   {
      x = arr[0]; y = arr[1]; z = arr[2];
   }

   template <typename ArgPack> Point(const ArgPack& args)
   { // Boost Parameter

      x = args[PointParams::X];
      y = args[PointParams::Y];
```

```
      z = args[PointParams::Z];
   }

   void print() const
   {
      std::cout << "(" << x << "," << y << "," << z << ")\n";
   }
};
```

A simple test program is:

```
#include "Point.hpp"

int main()
{
   // Boost parameter
   Point p1((PointParams::X = 1.0, PointParams::Y = 2.0,
   PointParams::Z = 2.0));
   p1.print();

   using namespace PointParams;
   Point p2((X = -1.0, Y = -2.0, Z = -2.0));
   p2.print();

   // Initialise member data directly
   Point p3;
   p3.print();
   p3.x = p3.y = p3.z = std::numeric_limits<double>::max();
   p3.print();

   // Uniform initialisation
   Point p4 { 2.71, 3.14, 22.0 / 7.0 };
   p4.print();

   // Create point from a tuple
   auto tup = std::make_tuple(-3.14, -3.14, -3.1415);
   Point p5(tup);
   p5.print();

   // From dynamic and fixed-sized arrays
   std::array<double, 3> arr = { 5.2, 3.3, 2.3 };
   Point p6(arr);
   p6.print();

   std::vector<double> arr2 = { -5.2, -3.3, -2.3 };
   Point p7(arr2);
   p7.print();

   return 0;
}
```

The output from this program is:

```
(1,2,2)
(-1,-2,-2)
(0,0,0)
(1.79769e+308,1.79769e+308,1.79769e+308)
(2.71,3.14,3.14286)
(-3.14,-3.14,-3.1415)
(5.2,3.3,2.3)
(-5.2,-3.3,-2.3)
```

The code in this section is a template or *exemplar* when dealing with more extended examples. See Exercise 2.

31.2.2 Boost *Parameter* and Option Data

We continue our discussion of Boost *Parameter* with an application to create the parameters needed by the Monte Carlo method for one-factor option pricing applications. We define these parameters as keywords:

```
#include <boost/parameter.hpp>

namespace OptionParams
{
   BOOST_PARAMETER_KEYWORD(Tag, strike)
   BOOST_PARAMETER_KEYWORD(Tag, expiration)
   BOOST_PARAMETER_KEYWORD(Tag, interestRate)
   BOOST_PARAMETER_KEYWORD(Tag, volatility)
   BOOST_PARAMETER_KEYWORD(Tag, dividend)
   BOOST_PARAMETER_KEYWORD(Tag, optionType)
}
```

We encapsulate option data in a struct that has appropriate constructors:

```
// Encapsulate all data in one place
struct OptionData
{ // Option data + behaviour

   double K;
   double T;
   double r;
   double sig;

   // Extra data
   double D;       // dividend

   int type;       // 1 == call, -1 == put
```

```
    explicit constexpr OptionData(double strike, double expiration,
                                  double interestRate,double volatility,
                                  double dividend, int PC): K(strike),
                                  T(expiration), r(interestRate),
                                  sig(volatility), D(dividend), type(PC) {}

    template <typename ArgPack> OptionData(const ArgPack& args)
    {
       K = args[OptionParams::strike];
       T = args[OptionParams::expiration];
       r = args[OptionParams::interestRate];
       sig = args[OptionParams::volatility];
       D = args[OptionParams::dividend];
       type = args[OptionParams::optionType];

    }

    // ..

};
```

An example of use is:

```
// Init via named Boost variables
OptionData myOption((OptionParams::strike = 65.0,
OptionParams::expiration = 0.25,
OptionParams::volatility = 0.3,
OptionParams::dividend = 0.0,
OptionParams::optionType = -1,
OptionParams::interestRate = 0.08));
```

31.3 MONTE CARLO VERSION 1: THE MONOLITH PROGRAM ('BALL OF MUD')

With all software projects a natural reaction is to write code to show that the project is feasible. In the current case we employ Pólya's heuristic H10 (*Specialisation*) (see Chapter 9) to solve a problem that is more specific than the current problem. In this case we could imagine writing code to price a range of multifactor options using the Monte Carlo method and its generalisations. It is not possible to design such a system without first scoping the problem. Thus, we discuss the simplest problem as a baseline case:

- The underlying process is described by *Geometric Brownian Motion* (GBM).
- Price one-factor plain call and put options.
- Use C++11 `<random>` to generate standard normal variates.
- The algorithm uses a double loop.

The C++ code is easy to follow:

```
class SDE
{ // Defines drift + diffusion + data
```

```
private:
   std::shared_ptr<OptionData> data;     // The data for the option
public:
   SDE(const OptionData& optionData)
      : data(new OptionData(optionData)) {}

   double drift(double t, double S)
   { // Drift term

      return (data->r - data->D)*S; // r - D
   }


   double diffusion(double t, double S)
   { // Diffusion term

      return data->sig * S;
   }

};
```

A monolithic test program is:

```
int main()
{
   std::cout <<  "1 factor MC with explicit Euler\n";

   // Init via arg positon

   // Init via named Boost variables
   OptionData myOption((OptionParams::strike = 65.0,
   OptionParams::expiration = 0.25,
   OptionParams::volatility = 0.3,
   OptionParams::dividend = 0.0,
   OptionParams::optionType = -1,
   OptionParams::interestRate = 0.08));

   SDE sde(myOption);

   // Initial value of SDE
   double S_0 = 60;

   // Variables for underlying stock
   double x;
   double VOld = S_0;
   double VNew;

   long NT = 100;
   std::cout << "Number of time steps: ";
   std::cin >> NT;

   long NSIM = 50000;
```

```
std::cout << "Number of simulations: ";
std::cin >> NSIM;
double M = static_cast<double>(NSIM);

double dt = myOption.T / static_cast<double> (NT);
double sqrdt = std::sqrt(dt);

// Normal random number
double dW;
double price = 0.0;     // Option price
double payoffT;
double avgPayoffT = 0.0;
double squaredPayoff = 0.0;
double sumPriceT = 0.0;

// Normal (0,1) rng
std::default_random_engine dre;
std::normal_distribution<double> nor(0.0, 1.0);

// Create a random number
dW = nor(dre);
long coun = 0; // Number of times S hits origin

StopWatch<> sw;
sw.Start();

for (long i = 1; i <= M; ++i)
{ // Calculate a path at each iteration

   if ((i/100'000) * 100'000 == i)
   {// Give status after each 10000th iteration

      std::cout << i << ", ";
   }

   VOld = S_0;
   x = 0.0;
   for (long index = 0; index <= NT; ++index)
   {

      // Create a random number
      dW = nor(dre);

      // The FDM (in this case explicit Euler)
      VNew = VOld + (dt * sde.drift(x, VOld))
             +(sqrdt * sde.diffusion(x, VOld) * dW);

      VOld = VNew;
      x += dt;
   }
```

```
        // Assemble quantities (postprocessing)
        payoffT = myOption.myPayOffFunction(VNew);
        sumPriceT += payoffT;
        avgPayoffT += payoffT/M;
        avgPayoffT *= avgPayoffT;

        squaredPayoff += (payoffT*payoffT);
    }

    // Finally, discounting the average price
    price = std::exp(-myOption.r * myOption.T) * sumPriceT/M;

    std::cout << "Price, after discounting: "
              << price << ", " << std::endl;
    double SD = std::sqrt((squaredPayoff / M)
             - sumPriceT*sumPriceT/(M*M));
    std::cout << "Standard Deviation: " << SD << ", " << std::endl;

    double SE = SD / std::sqrt(M);
    std::cout << "Standard Error: " << SE << ", " << std::endl;

    sw.Stop();
    std::cout << "Elapsed time in seconds: " << sw.GetTime() << '\n';

    return 0;
}
```

This code will not get us a Nobel Prize but we use it as an initial software prototype that we improve in an incremental fashion. At this stage we wish to get accurate results and an idea of the processing time. Of course, the design is a monolith.

The advantage of having this initial prototype is that we can trace the *data flow* in the system from major input to major output. In other words, we are able to visualise the individual *activities* in the *core process*. We may even discover some missing activities that we need to include in a future prototype.

31.4 POLICY-BASED DESIGN: DYNAMIC POLYMORPHISM

The initial prototype in the previous section solves a specific problem and it would be a major undertaking to adapt it in its current form to satisfy new requirements. It is tempting to try to adapt the code to suit new requirements, but this will lead to sorrow. We see it as a *throw-away prototype* and we prefer to start again by applying the system decomposition techniques from Chapter 9 to discover cohesive and loosely coupled components that cooperate to satisfy current and future requirements. In the second version we wish to support a range of SDEs, random number generators and finite difference schemes. We still focus on one-factor plain options. We reduce the risk by not trying to do too much in this cycle as it would entail having to write and debug a lot of code, even though we do not yet have a stable design.

The goal now is to decompose the system into subsystems and in this case we follow the approach taken in Section 9.3.4 (Policy-based design using templates and private inheritance).

In the current case the mediator/SUD class is composed of its template parameters (using shared pointers). The *mediator class* has the form:

```
template <typename Sde, typename Pricer, typename Fdm, typename Rng>
   class SUD : private Sde, private Pricer, private Fdm, private Rng
{ // System under discussion

private:
   // Four main components
   std::shared_ptr<Sde> sde;
   std::shared_ptr<Fdm> fdm;
   std::shared_ptr<Rng> rng;
   std::shared_ptr<Pricer> pricer;

private:
   // Other MC-related data

   int NSim;
   std::vector<double> res;       // Generated path per simulation

public:
   SUD() {}
   SUD(const std::shared_ptr<Sde>& s,
   const std::shared_ptr<Pricer>& p,
   const std::shared_ptr<Fdm>& f,
   const std::shared_ptr<Rng>& r,
   int numberSimulations, int NT)
            : sde(s), pricer(p), fdm(f), rng(r)
   {
      NSim = numberSimulations;
      res = std::vector<double>(NT + 1);
   }

   void start()
   { // Main event loop for path generation

      double VOld, VNew;

      for (int i = 1; i <= NSim; ++i)
      { // Calculate a path at each iteration

      if ((i / 100000) * 100000 == i)
      { // Give status after a given numbers of iterations

        std::cout << i << ",";
      }

      VOld = sde->InitialCondition(); res[0] = VOld;

      for (std::size_t n = 1; n < res.size(); n++)
```

```
        { // Compute the solution at level n+1

            VNew = fdm->advance(VOld, fdm->x[n - 1], fdm->k,
                        rng->GenerateRn(), rng->GenerateRn());
            res[n] = VNew; VOld = VNew;
        }

        // Send path data to the Pricer(s)
        // This step can be optimised
        pricer->ProcessPath(res);
        pricer->PostProcess();
    }

}

};
```

We notice the presence of a `start()` method that implements the Monte Carlo path evolver.

In this case we have modelled the classes `Pricer` and `Rng` using subtype polymorphism:

```
class Pricer
{
public:
    // Create a single path
    virtual void ProcessPath(const std::vector<double>& arr) = 0;

    // Notify end of simulation
    virtual void PostProcess() = 0;

    // Discounting, should be a delegate/signal
    virtual double DiscountFactor() = 0;

    // Option price
    virtual double Price() = 0;

    std::function<double (double)> m_payoff;
    std::function<double()> m_discounter;

    Pricer() = default;
    Pricer(const std::function<double(double)>& payoff,
           const std::function<double()>& discounter)
    {
        m_payoff = payoff;
        m_discounter = discounter;
    }
};

// Pricing Engines
class EuropeanPricer : public Pricer
```

```
{
private:
   double price;
   double sum;
   int NSim;

public:
   EuropeanPricer() {}
   EuropeanPricer(const std::function<double(double)>& payoff,
            const std::function<double()>& discounter)
            : Pricer(payoff, discounter)
   {
      price = sum = 0.0; NSim = 0;
   }

   void ProcessPath(const std::vector<double>& arr) override
   { // A path for each simulation/draw

   // Sum of option values at terminal time T
      sum += m_payoff(arr[arr.size() - 1]); NSim++;
   }

   double DiscountFactor() override
   { // Discounting

      return m_discounter();
   }

   void PostProcess() override
   {
      price = DiscountFactor() * sum / NSim;
   }

   double Price() override
   {
      return price;
   }

};
```

and

```
#include <random>

class Rng
{
public:
   virtual double GenerateRn() = 0;
};
```

```
class PolarMarsaglia: public Rng
{ // Only for educational purposes

private:
   std::default_random_engine dre;
   std::uniform_real_distribution<double> unif;
public:
   PolarMarsaglia (): dre(std::default_random_engine()),
      unif(std::uniform_real_distribution<double>(0.0, 1.0)) {}

   double GenerateRn() override
   {

      double u, v, S;

      do
      {
         u = 2.0 * unif(dre) - 1.0;
         v = 2.0 * unif(dre) - 1.0;
         S = u * u + v * v;
      } while (S > 1.0 || S <= 0.0);

      double fac = std::sqrt(-2.0 * std::log(S) / S);
      return u * fac;
   }
};

class CPPRng : public Rng
{ // C++11 versions

private:
   // Normal (0,1) rng
   std::default_random_engine dre;
   std::normal_distribution<double> nor;
public:
   CPPRng() : dre(std::default_random_engine()),
   nor(std::normal_distribution<double>(0.0, 1.0)) {}

   double GenerateRn() override
   {
      return nor(dre);
   }
};
```

In a later version we would need to modify the code to allow seeding of these random number generators.

Subtype polymorphism allows us to define various kinds of pricers and random number generators. The classes for stochastic differential equations and finite difference methods are hard-coded in this prototype. The `Pricer` hierarchy is set up in such a way that it can be incorporated into a design in which `SUD` sends messages to multiple pricers in a transparent manner using Boost *signals2* or .NET *delegates*, for example.

The SDE and FDM classes are hard-coded and are not derived from a base class in this version:

```
// Instance Systems
class GBM
{ // Simple SDE
private:
   double mu;     // Drift
   double vol;    // Constant volatility
   double d;      // Constant dividend yield
   double ic;     // Initial condition
   double exp;    // Expiry

public:
   GBM() = default;
   GBM(double driftCoefficient, double diffusionCoefficient,
        double dividendYield, double initialCondition,
        double expiry)
   {
      mu = driftCoefficient;
      vol = diffusionCoefficient;
      d = dividendYield;
      ic = initialCondition;
      exp = expiry;
   }

   double Drift(double x, double t) { return (mu - d) * x; }
   double Diffusion(double x, double t) { return vol * x; }

   double DriftCorrected(double x, double t, double B)
   {
      return Drift(x, t) - B * Diffusion(x, t)*DiffusionDerivative(x, t);
   }

   double DiffusionDerivative(double x, double t)
   {
      return vol;
   }

   // Property to set/get initial condition
   double InitialCondition() const
   {
      return ic;
   }


   // Property to set/get time T
   double Expiry() const
   {
      return exp;
   }

};
```

and

```
template <typename Sde>
   class EulerFdm
{
private:
   std::shared_ptr<Sde> sde;
   int NT;
public:
   std::vector<double> x;      // The mesh array
   double k;                   // Mesh size

   double dtSqrt;
public:
   EulerFdm() = default;
   EulerFdm(const std::shared_ptr<Sde>& stochasticEquation,
            int numSubdivisions)
            : sde(stochasticEquation), NT(numSubdivisions)
   {

      NT = numSubdivisions;
      k = sde->Expiry() / static_cast<double>(NT);
      dtSqrt = std::sqrt(k);
      x = std::vector<double>(NT + 1);

      // Create the mesh array
      x[0] = 0.0;
      for (std::size_t n = 1; n < x.size(); ++n)
      {
         x[n] = x[n - 1] + k;
      }
   }

   double advance(double xn, double tn, double dt,
            double  normalVar, double  normalVar2) const
   {
      return xn + sde->Drift(xn, tn) * dt
                + sde->Diffusion(xn, tn) * dtSqrt * normalVar;
   }
};
```

In principle, we could have created SDE and FDM class hierarchies in this version but we have resisted the temptation for the moment due to lack of project time.

We give a test case of pricing a call option and a put option on the same underlying data. One pricer uses the polar Marsaglia algorithm for random number generation while the other algorithm uses functionality from the C++11 `<random>` library:

```
int main()
{
   int NSim = 1'000'000;
   int NT = 500;

   double driftCoefficient = 0.08;
```

```
double diffusionCoefficient = 0.3;
double dividendYield = 0.0; double initialCondition = 60.0;
double expiry = 0.25;

auto sde = std::shared_ptr<GBM>
            (new GBM(driftCoefficient, diffusionCoefficient,
            dividendYield, initialCondition, expiry));

double K = 65.0;

// Factories for objects in context diagram
std::function<double(double)> payoffPut = [&K](double x)
              {return std::max<double>(0.0, K - x); };
std::function<double(double)> payoffCall = [&K](double x)
              {return std::max<double>(0.0, x - K); };
double r = 0.08; double T = 0.25;
std::function<double()> discounter = [&r, &T]()
              { return std::exp(-r * T); };

auto pricerCall = std::shared_ptr<Pricer>
              (new EuropeanPricer(payoffCall, discounter));
auto pricerPut = std::shared_ptr<Pricer>
              (new EuropeanPricer(payoffPut, discounter));

auto fdm = std::shared_ptr<EulerFdm<GBM>>
         (new EulerFdm<GBM>(sde, NT));

auto rngPM = std::shared_ptr<Rng>(new PolarMarsaglia());
auto rngCPP = std::shared_ptr<Rng>(new CPPRng());

StopWatch<> sw;
sw.Start();

SUD<GBM, EuropeanPricer, EulerFdm<GBM>, CPPRng>
      s(sde, pricerPut, fdm,rngPM, NSim, NT);

s.start();

std::cout << "\n C++11 Rng: " << pricerPut->Price() << '\n';

sw.Stop();
std::cout << "Elapsed time in seconds: " << sw.GetTime() << '\n';

sw.Start();
SUD<GBM, EuropeanPricer, EulerFdm<GBM>, CPPRng>
      s2(sde, pricerCall, fdm, rngCPP, NSim, NT);

s2.start();

std::cout << "\n Polar Marsaglia: " << pricerCall->Price();
```

```
    sw.Stop();
    std::cout << "Elapsed time in seconds: " << sw.GetTime() << '\n';

    return 0;
}
```

You can run this code and examine the output. In Chapter 32 we shall run the code in parallel mode.

31.5 POLICY-BASED DESIGN APPROACH: CRTP AND STATIC POLYMORPHISM

We have already shown how to apply CRTP to PDEs in Chapter 21 (in particular, Section 21.7.2). We now apply CRTP to the code in Section 31.4 by transforming the class hierarchies `Pricer` and `Rng` to CRTP form. We should always flag our reasons for using this pattern, for example efficiency, reliability and maintainability. In the case of CRTP, efficiency is certainly an important reason for using it because there is no run-time overhead. The two class hierarchies in CRTP form are now:

```
template <typename D>
   class Pricer
{
public:
   void ProcessPath(const std::vector<double>& arr)
   { // Create a single path

      return static_cast<D*>(this)->ProcessPath(arr);
   }

   void PostProcess()
   { // Notify end of simulation

      return static_cast<D*>(this)->PostProcess();
   }

   virtual double DiscountFactor()
   { // Discounting, should be a delegate

      return static_cast<D*>(this)->DiscountFactor();
   }

   virtual double Price()
   { // Option price

      return static_cast<D*>(this)->Price();
   }


   std::function<double (double)> m_payoff;
```

```
    std::function<double()> m_discounter;

    Pricer() = default;
    Pricer(const std::function<double(double)>& payoff,
           const std::function<double()>& discounter)
    {
       m_payoff = payoff;
       m_discounter = discounter;
    }
};

// Pricing Engines
class EuropeanPricer : public Pricer< EuropeanPricer>
{
private:
   // ...

public:

   // Code same as in section 31.4
};
```

and

```
template <typename D> class Rng
{
public:
   double GenerateRn()
   {
      return static_cast<D*>(this)->GenerateRn();

   }
};

class PolarMarsaglia : public Rng<PolarMarsaglia>
{ // Only for educational purposes

   private:
      std::default_random_engine dre;
      std::uniform_real_distribution<double> unif;
      // ...
   public:
      // ...
};
```

The code to test this new hierarchy is similar to that in Section 31.4 and we see the emergence of instantiated pricer and random number classes:

```
auto pricerCall = std::shared_ptr<Pricer<EuropeanPricer>>
                     (new EuropeanPricer(payoffCall, discounter));
```

```
auto pricerPut = std::shared_ptr<Pricer<EuropeanPricer>>
                    (new EuropeanPricer(payoffPut, discounter));

auto fdm = std::shared_ptr<EulerFdm<GBM>>(new EulerFdm<GBM>(sde, NT));

auto rngPM = std::shared_ptr<Rng<PolarMarsaglia>>
                     (new PolarMarsaglia());
auto rngCPP = std::shared_ptr<Rng<CPPRng>>(new CPPRng());

StopWatch<> sw;
sw.Start();

SUD<GBM, EuropeanPricer, EulerFdm<GBM>, CPPRng>
                       s(sde, pricerPut, fdm, rngCPP, NSim, NT);
s.start();
```

31.6 *BUILDERS* AND THEIR SUBCONTRACTORS (*FACTORY METHOD PATTERN*)

The PDB approach promotes loose coupling between components but we have to create these components first. This was done in `main()` in Sections 31.4 and 31.5. The impending maintainability issue is that this function does not satisfy the *Single Responsibility Principle* (SRP) because it first creates the components (including the mediator/SUD) and then it runs the application. We separate out the code to create objects and we place this responsibility in a new *Builder* class. This class may in its turn delegate to more specialised *factory objects*. A factory class creates an object of a specific type. This new requirement necessitates a major modification of the design and code that we created in Sections 31.4 and 31.5. Fortunately, the only component that needs to be modified is the mediator. The main steps are:

1. Create a factory class that creates an instance of one of the components that communicates with the mediator. In a sense, these factories are *subcontactors*.
2. Create a builder class that is composed of the factory objects in part 1. It has an interface consisting of a tuple having components that the mediator communicates with.
3. `main()` plays the role of the *director* in the *Builder* pattern (as described in GOF, 1995). It chooses which builder to use, calls its interface function to create the components and then delivers them to the mediator.
4. The director calls the mediator's *start/run* method to compute the option price.

These steps can be executed in phases. First, we split the code in `main()` into a free function that creates a tuple of components that is used in `main()` and then we run the application. Having done that and checked that the code works as before we can then redesign the builder component as an aggregation of dedicated factory objects. Some modifications to the code are needed. For example, the mediator accepts a tuple of components:

```
SUD(const std::tuple<std::shared_ptr<Sde>, std::shared_ptr<Pricer>,
   std::shared_ptr<Fdm>, std::shared_ptr<Rng>>& parts,
   int numberSimulations, int NT)
```

```
        : sde(std::get<0>(parts)), pricer(std::get<1>(parts)),
          fdm(std::get<2>(parts)), rng(std::get<3>(parts))
{
   NSim = numberSimulations;
   res = std::vector<double>(NT + 1);
}
```

Clients then need to create components, encapsulate them in a tuple and then call this constructor, for example in the case of a CEV option:

```
{ // CEV
   double K = 110.0;
   double r = 0.05; double sig = 0.2; double T = 0.5;

   // Payoff function factories
   std::function<double(double)> payoff = [=](double x)
      {return std::max<double>(0.0, x - K); };

   std::function<double()> discounter = [=]()
      { return std::exp(-r * T); };
   auto pricer = std::shared_ptr<Pricer>
         (new EuropeanPricer(payoff, discounter));

   auto cevSde = ChooseSde<double>(2);


   auto fdmPCMid = std::shared_ptr<EulerFdm>
      (new EulerFdm(cevSde, NT));

   SUD sPCMid(cevSde, pricer, fdmPCMid, rng, NSim, NT);
   sPCMid.start();
   std::cout << "\nMidpoint Normal CEV 32.10:  "
             << std::setprecision(16) << sPCMid.Price() << '\n';
}
```

There are many ways to extend the code to make it more flexible and we delegate these issues to the exercises here and to the exercises in Chapter 32.

31.7 PRACTICAL ISSUE: STRUCTURING THE PROJECT DIRECTORY AND FILE CONTENTS

As in all software projects, maintainability will play an important role as more functionality is added to the software framework. The software tends to become more difficult to modify, mainly due to the coupling between the modules in the object network. We should try to ensure that each module satisfies the SRP and that there is loose coupling between modules. One possible approach to achieve the latter objective is to apply the *Layers* design pattern that we discussed in Chapters 11 and 12. In this section we discuss the practical issue on how to set up the directory and file structure that reflects the layered design shown in Figure 31.1. Each box can be a function or class in its own file. In most cases we adhere to the principles of SRP,

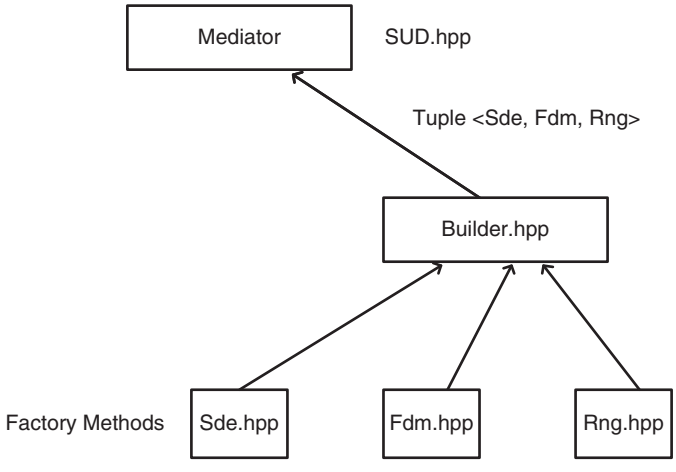


FIGURE 31.1 Layered design

although in some cases we may decide to place the code for a class and its related factories in a single file for convenience. We can also decide to create libraries and DLLs and place models in them as discussed in Chapter 8.

For example, we can write code to implement the *Factory Method* design pattern (GOF, 1995), in this case an interface to create an SDE. A specific example is:

```
template <typename T>
   using ISde = std::tuple<FunctionType<T>, FunctionType<T>>;

template <typename T>
   std::shared_ptr<Sde<T>> ChooseSde(int choice)
{ // Simple factory method

   // Initial condition for SDE
   T S0 = 60.0;
   T expiry = 0.25;

   if (1 == choice)
   {
      std::cout << "GBM\n";
      // Second test case; GBM dS = a S dt + sig S dW
      T r = 0.08; T sig = 0.3;
      auto Drift = [=](T t, T S) { return r * S; };
      auto gbmDiffusion = [=](T t, T S) { return sig * S; };

      ISde<T> gbmFunctions =
            std::make_tuple(Drift, gbmDiffusion);

      // Create Sde
      return std::shared_ptr<Sde<T>>
         (new Sde<T>(gbmFunctions, S0, expiry));
   }
```

```
    else
    {
       std::cout << "CEV\n";
       double r = 0.08; double sig = 0.3;
       auto Drift = [=](double t, double S) { return r * S; };
       auto gbmDiffusion = [=](double t, double S) { return sig * S; };

       double beta = 0.5;
       auto cevDiffusion =
                        [=](double t, double S) {return sig *
                        std::pow(S, beta); };

       ISde<double> cevFunctions
          = std::make_tuple(Drift, cevDiffusion);

       // Create Sde
       return std::shared_ptr<Sde<T>>
             (new Sde<T>(cevFunctions, S0, expiry));
    }
}
```

In a later version we would place this code in a dedicated file, for example `Sde Factory.hpp`. An example of use is:

```
// Using factories
auto sde = ChooseSde<double>(1);
double S = 100.0; double t = 0.5;
std::cout << "Drift, diffusion: " << sde->drift(t,S) << ", "
          << sde->diffusion(t,S) << '\n';
```

31.8 SUMMARY AND CONCLUSIONS

In this chapter we applied the system decomposition methods that we introduced in Chapter 9 to the creation of a software framework to price one-factor plain options using the Monte Carlo method. We know what the final design should look like but instead of implementing the design blueprints based on Figure 9.4 (for example) we embarked on a process of *bottom-up discovery* in which we produced a series of software prototypes, each one having more structure and functionality than the one preceding it. In this way we can apply a risk-driven software process to determine which functionality to add for each new prototype.

This chapter gives an indication of how to develop a software system in an incremental fashion while at the same time attempting to keep the code maintainable.

Some of the open issues that could be used as requirements for a new cycle of the *Spiral Model* are:

- R1: More general SDEs, for example support for *corrected drift functions* and calculating the derivative of the diffusion term.
- R2: Support for a range of finite difference schemes, for example the predictor–corrector and Milstein methods.

- R3: More support for random number generation based on the C++ <*random*> library.
- R4: Improving speedup by using multithreading and multitasking code (for example, using *C++ Concurrency*).
- R5: Creating a mediator class that prices several option types in a seamless manner (using *events* that are supported by .NET *delegates* or by the Boost *signals2* library). The prices use the same path information delivered by the path evolver.
- R6: Developing flexible modules (for example, builders and factories) to configure the software framework while keeping the code maintainable.

We discuss these new requirements in Chapter 32. The code can always be improved.

31.9 EXERCISES AND PROJECTS

1. (Spiral Model, Mini-Project)

The goal of this exercise is to apply a *risk-driven* approach to the software development process. You play the role of *software project manager* as it were. We base the questions here on the *Spiral Model* (Boehm, 1986) and more generally on the approach taken in Sommerville and Sawyer (1997). In general, the goal in this exercise is to design and implement features R1 to R6 in Section 31.8 accurately, on time and within budget. We need to understand the mathematical and financial background to these requirements. Then we map these requirements to C++ code and we integrate this code into the software framework as presented in this chapter.

Answer the following questions:

a) Determine the main risks associated with features R1 to R6. For example, if either the mathematical or the financial background is not yet known, there may be stability, process, performance or other kinds of risks (Sommerville and Sawyer, 1997). In particular, rank each feature according to its importance to the success of the project and its risk of not being realised.

b) Estimate the expected time to complete each of the requirements R1 to R6.

We now discuss the problem of estimating *activity times*. In general, estimating how long it takes to complete an activity is non-trivial as it demands insight, preparation and experience. The *Critical Path Method* (CPM) network method assumes that a single value estimate of the time required for each activity is available. The *Project Evolution and Review Technique* (PERT), on the other hand, includes an explicit recognition of the uncertainty associated with activity time estimates. We model activity duration as a *triangular probability distribution* having three *uncertainty parameters*:

- Variable a = the *most optimistic (shortest) time* to complete the project or activity. We assume that everything proceeds better than is normally expected.
- Variable b = the *most likely (modal) time*. We assume that everything proceeds as expected.
- Variable c = the *most pessimistic (longest) time*. We assume that everything goes wrong (but at the same time avoiding major catastrophes).

We note that Boost supports the *triangular distribution* and its statistical properties as documented in the *Math Toolkit* library and the *Random* library that allows us to generate random variates of the triangular distribution. Continuing, based on the above

three estimates we compute the *expected time* or the *best estimate* to complete an activity by the formula:

$$T_e \equiv (a + 4b + c)/6.$$

In general, we apply this formula to each activity in the project. This expected time is analogous to the single value estimate in the CPM method.

c) Determine how many cycles to use in the current case. Identify areas of uncertainty for each cycle, determine the objectives, consider alternative means of implementing parts of the cycle (buy/reuse/design), consider cost, schedule and interface constraints.

d) Having completed a cycle, how would you review the progress to date?

e) The *Spiral Model* is risk driven. What are its advantages when compared to *document-driven or code-driven* software process models? How does it compare with your current practices?

f) How would you start a project to price *n*-factor problems using the Monte Carlo method?

2. (Function Arguments)

In Section 31.2 we discussed options A1 to A5 to package parameters as input arguments to functions as well as requirements R1 to R4 that these options may or may not have to satisfy in a given situation.

Answer the following questions:

a) Given a requirement (say R1), which of A1, …, A5 is a good option in order to satisfy it? Which ones have a negative impact? Consider both one-off configuration data and run-time data that is continuously or regularly updated.

b) Consider the code in Section 31.3. Instead of generating a random number each time one is needed (in the *inner loop*) we create an array of random numbers for each *draw* of the simulation (in the *outer loop*):

```
// Normal random number array
std::vector<double> dW(NT + 1);
for (long i = 1; i <= M; ++i)
{ // Calculate a path at each iteration

   // ...

   // Precompute array of random variates
   generateRN(dW);

   for (long index = 0; index < dW.size(); ++index)
   {

      // The FDM (in this case explicit Euler)
      VNew = VOld + (k * sde.drift(x, VOld))
                  +(sqrk * sde.diffusion(x, VOld) * dW[index]);

      VOld = VNew;
      x += k;
   }
   // ...

}
```

Are there advantages to *precomputing* these arrays of random numbers compared to the approach taken in Section 31.3? Anecdotal evidence seems to suggest that there is no negative performance impact when using this option in a single-threaded program.

c) Determine the efficiency of using tuples to hold run-time data. For example, consider the Box–Muller and polar Marsaglia algorithms, each of which computes a tuple/pair of standard normal variates. Clients only need one of their elements. Investigate the efficiency when calling the algorithms a large number of times. The interface is along the lines of the following code:

```
// This function is the interface to generate normal
// variates in interval (L,U).

using RngDelegate = std::function<double(double L , double U)>;

class Rng
{
public:
   virtual std::tuple<double, double> GenerateRn() = 0;
};


class PolarMarsaglia :public Rng {
private:
   RngDelegate randu;
public:
   PolarMarsaglia(RngDelegate src);
   std::tuple<double, double> GenerateRn();
};


class BoxMuller : public Rng {
private:
   RngDelegate randu;
public:
   BoxMuller(RngDelegate src);
   std::tuple<double, double> GenerateRn();
};
```

d) Which of the options A1, …, A5 would you use for the following problems?

- The data to describe two-factor basket options.
- The input arguments of the coefficients of a PDE.
- Function arguments used in optimisation (for example, Levenberg–Marquardt, differential evolution).

3. (Constant Elasticity of Variance (CEV) Model, Project)

This is a popular model because it is able to reproduce an *implied volatility skew* for plain options. It is used in a range of applications such as equity, commodity and interest rate modelling. In the case of equities, we define the SDE as:

$$dS = (r - q)\,Sdt + \sigma S^{\beta} dW \tag{31.1}$$

where:

r = risk-free rate
q = constant dividend
σ = volatility scale parameter
β = volatility elasticity factor (a constant)
W = Wiener process.

This SDE subsumes some well-known models as special cases; for $\beta = 1$ the process reduces to geometric Brownian motion and when $\beta = 0$ we get an *Ornstein–Uhlenbeck process*. For $\beta = 0.5$ we obtain the *square-root process*.

How do we interpret the CEV model? When $\beta < 1$ the local volatility increases as the stock price decreases. This implies a heavy left tail and a less heavy right tail, as witnessed with equity options. Empirical evidence suggests that values $\beta = -3$ and $\beta = -4$ are appropriate for the S&P 500 index. When $\beta > 1$ local volatility increases as the stock price increases. This implies a heavy right tail and a less heavy left tail, as witnessed with options on futures. In general, the CEV parameters are estimated by a *calibration process*.

We now discuss how to price call and put options based on the process in equation (31.1). We write it in the form:

$$dS = (r - q)\,Sdt + (\sigma S^{\beta-1})SdW. \tag{31.2}$$

We now give a number of datasets and option values based on the Monte Carlo method. These sets are taken from Wong and Jing (2008); the test values are $r = 0.05,\ q = 0.0, \sigma = 0.20,\ S_0 = 100.0,\ T = 0.5$. The parameters K and β are variable and we concentrate on the put price. The exact values for European options are:

$K = 90, \beta = 0,\ P = 1.4680$
$K = 110, \beta = 0, P = 9.9552$
$K = 90, \beta = 2 = 3, P = 1.3380$
$K = 110, \beta = 2 = 3, P = 10.1099$
$K = 90, \beta = -3, P = 2.2210$
$K = 110, \beta = -3, P = 9.3486.$

The Monte Carlo method gives good results in general, except in the case of $\beta = -3$, where the value is hovering around 8.83 irrespective of the number of time steps, number of simulations and method used. This is an attention point.

Answer the following questions:

a) Model the SDE (31.1)/(31.2) in C++ and add it to the software framework. Does this new feature have an impact on the stability of the code, that is, how much code needs to be modified?

b) Compute the put price using the Euler and Milstein methods.

We can also formulate the problem as a partial differential equation (PDE):

$$-\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \beta \frac{\partial^2 V}{\partial S^2} + (r - q)S\frac{\partial V}{\partial S} - rV = 0, \quad S \geq 0, 0 < t \leq T. \tag{31.3}$$

This equation will have a unique, well-defined solution if we prescribe the *payoff function* (also known as the *initial condition*) where we assume that we price a *put* option:

$$V(S,0) = V_0(S) = \max(K - S, 0) \tag{31.4}$$

and the *boundary conditions* for a plain put option:

$$V(0,t) = Ke^{-(r-q)t}, \quad 0 < t \leq T \tag{31.5}$$

$$V(S_{\max}, t) = 0. \tag{31.6}$$

The values $S = 0$ and $S = S_{\max}$ are called the *near-* and *far-field boundaries*, respectively.

Model this PDE using one of the finite difference schemes in this book (Chapters 20 to 23). Compare the results with the exact values. In particular, compare the relative accuracy and efficiency of the PDE and Monte Carlo approaches.

4. (Milstein Method, Project)

In this exercise we implement the Milstein scheme and a variation of it in which we do not need to compute derivatives.

These schemes are based on truncated Taylor expansions. The simplest *strong Taylor scheme* is the Euler method and its *strong order of convergence* is 0.5. It is useful to use it when the drift and diffusion terms are nearly constant but the method is unsuitable in more complex cases and when we wish to achieve higher accuracy. The Milstein scheme has strong order of convergence equal to 1.0. It is the Euler scheme with an extra added term. To this end, consider the SDE:

$$dX = \mu(t,X)dt + \sigma(t,X)dW. \tag{31.7}$$

Then the Milstein scheme is:

$$X_{n+1} = X_n + \mu_n \Delta t + \sigma_n \Delta W_n + \tfrac{1}{2}\sigma_n \sigma_n' \left\{ \left(\Delta W_n\right)^2 - \Delta t \right\} \tag{31.8}$$

where:

$$\mu_n = \mu(t_n, X_n), \quad \sigma_n = \sigma(t_n, X_n), \quad \sigma' = \frac{\partial \sigma}{\partial X}.$$

Finally, the following Ito–Taylor scheme has strong convergence order equal to 1.5 (Kloeden and Platen, 1992, p. 351):

$$\begin{aligned} X_{n+1} = X_n &+ \mu \Delta t + \sigma \Delta W_n + \tfrac{1}{2}\sigma\sigma' \left\{ \left(\Delta W_n\right)^2 - \Delta t \right\} \\ &+ \mu' \sigma \Delta Z_n + \tfrac{1}{2}\left(\mu\mu' + \tfrac{1}{2}\sigma^2\mu''\right)\Delta t^2 \\ &+ \left(\mu\sigma' + \tfrac{1}{2}\sigma^2\sigma''\right)\left\{\Delta W_n \Delta t - \Delta Z_n\right\} \\ &+ \tfrac{1}{2}\left(\sigma'' + \left(\sigma'\right)^2\right)\left\{\tfrac{1}{3}\left(\Delta W_n\right)^2 - \Delta t\right\}\Delta W_n \end{aligned} \tag{31.9}$$

where we use the shorthand notation:

$$\mu = \mu\left(t_n, X_n\right), \sigma = \sigma\left(t_n, X_n\right), \sigma'' = \frac{\partial^2 \sigma}{\partial X^2}, \mu' = \frac{\partial \mu}{\partial X}, \mu'' = \frac{\partial^2 \mu}{\partial X^2}$$

and ΔZ_n is $N(0,1)$ normally distributed.

We calculate the pair $(\Delta W, \Delta Z)$ as follows:

$$\begin{aligned} \Delta W &= U_1 \sqrt{\Delta t} \\ \Delta Z &= \tfrac{1}{2} \Delta t^{3/2} \left(U_1 + \tfrac{1}{\sqrt{3}} U_2 \right) \end{aligned} \tag{31.10}$$

where U_1 and U_2 are two independent $N(0,1)$-distributed random variables.

The strong Taylor approximations have a number of disadvantages, such as the necessity of calculating derivatives of the drift and diffusion functions (which may not be continuous). The performance may not be optimal due to many function evaluations and the generation of pairs of standard normal random variates. Some of these drawbacks can be mitigated by employing schemes that avoid the use of derivatives. To motivate this class of schemes, we examine equation (31.7) again. We consider the following *derivative-free* method (a variation of the Milstein scheme (31.8)):

$$\begin{aligned} X_{n+1} &= X_n + \mu \Delta t + \sigma \Delta W_n + \tfrac{1}{2} \sqrt{\Delta t} \left\{ \sigma\left(t_n, \overline{X}_n\right) - \sigma \right\} \left\{ \left(\Delta W_n\right)^2 - \Delta t \right\} \\ \overline{X}_n &= X_n + \mu \Delta t + \sigma \sqrt{\Delta t} \end{aligned} \tag{31.11}$$

where we use the same notation as in equation (31.9). We see that this scheme is similar to scheme (31.8) when we realise that the term:

$$\frac{1}{\sqrt{\Delta t}} \left\{ \sigma\left(t_n, X_n + \mu \Delta t + \sigma \sqrt{\Delta t}\right) - \sigma\left(t_n, X_n\right) \right\} \tag{31.12}$$

is an approximation to $\sigma \sigma'$ at (t_n, X_n). This is an explicit order 1.0 strong scheme.

Answer the following questions:

a) Implement schemes (31.8) and (31.11) and add them to the software framework. Test these schemes on the GBM and CEV models.

b) Compare the accuracy and efficiency of these schemes with those delivered by the Euler method.

c) Implement a finite difference scheme of your choice and execute parts a) and b). Compare its accuracy with that of the other schemes that we introduced in this chapter.

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

CHAPTER 32

Monte Carlo Simulation, Part II

32.1 INTRODUCTION AND OBJECTIVES

In this chapter we continue with the analysis of the Monte Carlo method already discussed in Chapter 31, in which we designed a software framework for one-factor plain options. We discovered a number of new features that we would like to address here:

- R1: More general SDEs, for example support for *corrected drift functions* and the derivative of the diffusion term.
- R2: Support for a range of finite difference schemes, for example the predictor–corrector and Milstein methods.
- R3: More support for random number generation based on the C++ <*random*> and Boost *Random* libraries.
- R4: Improving speedup by using multithreading and multitasking code (for example, *C++ Concurrency* and PPL).
- R5: Creating a mediator class that prices several option types in a seamless manner (using *events* that are supported by *Agents Library* and the Boost *signals2* library).
- R6: Developing flexible modules (for example, builders and factories) to configure the software framework while keeping the code maintainable.

We address these requirements by creating C++ code to realise them.

32.2 PARALLEL PROCESSING AND MONTE CARLO SIMULATION

In this section we discuss feature R4. In general, the Monte Carlo method has a reputation for being slow, especially when we simulate many paths (Glasserman, 2004). (Incidentally, it is a good idea to run your Monte Carlo application in *release mode.*) We can start thinking about how to improve the speedup by using parallel programming techniques. However, there are a number of caveats:

- C1: We need guidelines to tell us which parts of the code to parallelise. We can choose between *fine-grained* and *coarse-grained parallelisation* (Campbell and Miller, 2011;

Mattson, Sanders and Massingill, 2005). In the latter case we can employ the *Geometric Decomposition* pattern to decompose a data structure into concurrently updatable 'chunks'.

- C2: Parallelisation may not produce the results that we were hoping for. For example, the speedup may be poor or the accuracy may not be optimal. In some applications we may demand *sequential equivalence*, which states that the results from a parallel program should be the same as those from a single-threaded program. This demand may be compromised by subtle race conditions or due to the differences in round-off errors depending on the number of threads used.
- C3: The application that we are trying to parallelise is too small to be parallelised and a sequential solution will perform better.
- C4: Some applications are inherently sequential and there is little point trying to parallelise them. To this end, the *serial fraction* γ is the fraction of a program's execution time taken up by parts that must be executed serially. Then, *Amdahl's Law* states that the maximum speedup $S(P)$ which can be attained by executing an algorithm on P processors is given by the formula:

$$S(P) = \frac{T(1)}{\gamma + \frac{1-\gamma}{P}} \tag{32.1}$$

where:

$T(P)$ = total execution time running on P processors
$S(P)$ = speedup on P processors
γ = serial fraction.

More generally, consider a program consisting of a setup section, a compute section and a finalisation section. The total running time is given by:

$$T_{total} = T_{setup} + T_{compute} + T_{finalisation}. \tag{32.2}$$

We assume that the setup and finalisation sections cannot be run concurrently with any other activities but that the compute section is amenable to a parallel implementation. Then, an estimate of the serial fraction is given by:

$$\gamma = \frac{T_{setup} + T_{finalisation}}{T_{total}(1)}. \tag{32.3}$$

- C5: In the case of Monte Carlo simulation, we have the extra challenges when using random number generators. A given generator must be thread safe and re-entrant if we wish to get accurate and unbiased results. We also need to ensure that simulated paths are independent.

We now discuss a number of ways to parallelise Monte Carlo code using OpenMP, PPL tasks as well as C++ threads and futures. More generally, the presented code can be used as a reference for other applications.

32.2.1 Some Random Number Generators

Until recently, there were few standardised (standard) C and C++ libraries for random number generation and statistical distributions. But as discussed in Chapter 26, we see that C++ has support for 16 random number engines and 30 distributions. Furthermore, the *Boost Random library* has support for 30 random number engines and 28 distributions. It subsumes the functionality in C++ `<random>` and it contains a number of useful functions not found in C++ at the moment of writing.

In Chapter 31 we defined several random number generators as C++ classes that we can use in the software framework. We now define two more classes based on the 64-bit versions of the Mersenne Twister and Boost lagged Fibonacci algorithms, respectively. The code is based on the *CRTP* pattern:

```
class MTRng64 : public Rng<MTRng64>
{ // C++11 versions

private:
   // Normal (0,1) rng
   std::mt19937_64 dre;
   std::normal_distribution<double> nor;
public:
   MTRng64() : dre(std::mt19937_64()),
   nor(std::normal_distribution<double>(0.0, 1.0)) {}

   double GenerateRn()
   {
      return nor(dre);
   }
};
```

and

```
using FibGenerator = boost::variate_generator
   <boost::lagged_fibonacci607, boost::normal_distribution<double>>;

class FibonacciRng : public Rng<FibonacciRng>
{ // C++11 versions

private:
   // Normal (0,1) rng
   boost::lagged_fibonacci607 dre;
   boost::normal_distribution<double> nor;

   FibGenerator gen;

public:
   FibonacciRng() : dre(boost::lagged_fibonacci607()),
         nor(boost::normal_distribution<double>(0.0, 1.0)),
         gen (FibGenerator(dre, nor)){}
```

```
    double GenerateRn()
    {
       return gen();
    }
};
```

In general, it is easy to create new classes by copying the code, changing the name of the class and replacing the random number engine and distribution by one of your choice. For example, this is how we designed the classes `MTRng64` and `FibonacciRng`. See also Exercise 1.

32.2.2 A Test Case

We take an example (for motivation) in which we price four options. Each option will be priced in its own thread. We avoid race conditions because first, shared data is read-only and second, each thread uses its own random number engine. In more advanced examples we may not be in this enviable position and we then need to create locks on writeable shared data. This will have a negative impact on the speedup, however.

The various multithreaded implementations that we present here are based on the *fork/join* pattern (Mattson, Sanders and Massingill, 2005; Nichols, Buttlar and Farrell, 1996). We depict the pattern in Figure 32.1. The program starts as a single thread of execution. Then a team of threads is created (forked) and fired at the beginning of a *parallel region*. The threads are joined at a synchronisation point or *barrier* when each thread has run its course. In this case we can speak of a *parent thread* (or task) and *child threads* (or tasks). We note that threads can be dynamically created during the course of program execution. The *fork/join* pattern is a core model in OpenMP (Chapman, Jost and Van der Pas, 2008).

The task graph for this test case is easy to draw and it is shown in Figure 32.1. Each task corresponds to an option pricer based on a particular random number engine. The computed option prices are noted as the variables a, b, c and d.

Before we show multithreaded code we create the thread functions and each one will run in its own thread. Notice that we create four random number generators and four threads. In this way we avoid discussions on thread-safe and re-entrant code for the moment:

```
int NSim = 1'000'000;
int NT = 500;

double driftCoefficient = 0.08; double diffusionCoefficient = 0.3;
double dividendYield = 0.0; double initialCondition = 60.0;
double expiry = 0.25;
auto sde = std::shared_ptr<GBM>
          (new GBM(driftCoefficient, diffusionCoefficient, dividendYield,
          initialCondition, expiry));

double K = 65.0;

// Factories for objects in context diagram
std::function<double(double)> payoffPut
   = [&K](double x) {return std::max<double>(0.0, K - x); };
```

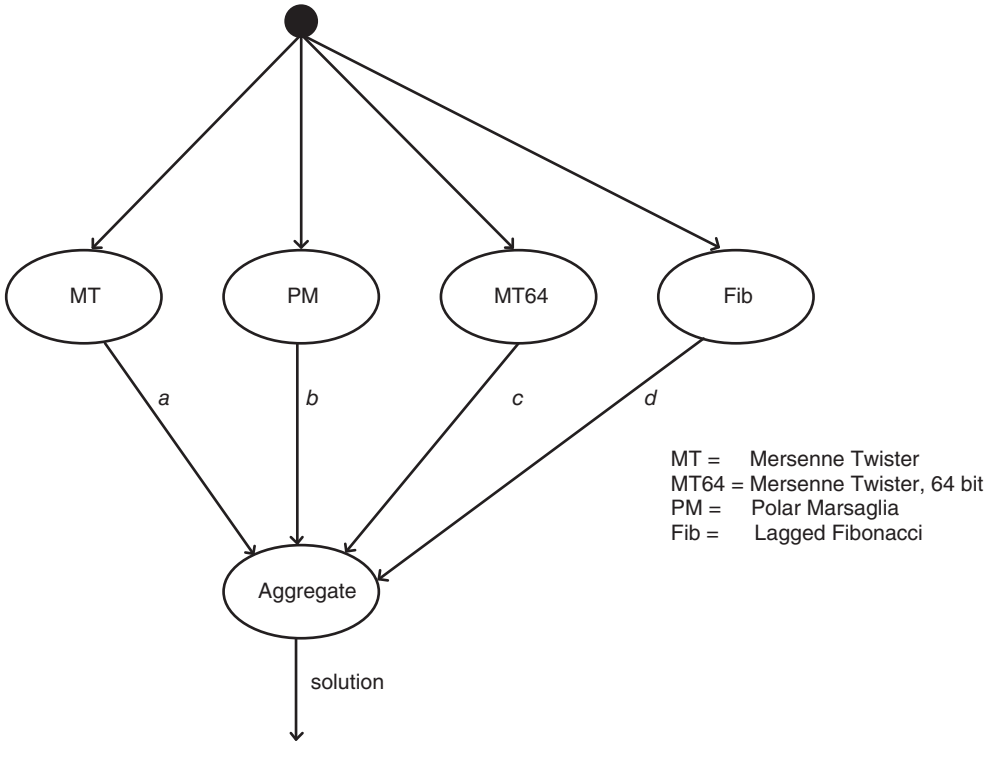


FIGURE 32.1 Task graph for test case

```
std::function<double(double)> payoffCall
   = [&K](double x) {return std::max<double>(0.0, x - K); };
double r = 0.08; double T = 0.25;
std::function<double()> discounter = [&r, &T]()
             { return std::exp(-r * T); };

auto pricerCall = std::shared_ptr<Pricer<EuropeanPricer>>
             (new EuropeanPricer(payoffCall, discounter));
auto pricerPut = std::shared_ptr<Pricer<EuropeanPricer>>
             (new EuropeanPricer(payoffPut, discounter));

auto fdm = std::shared_ptr<EulerFdm<GBM>>(new EulerFdm<GBM>(sde, NT));

// Random number generators
auto rng1 = std::shared_ptr<Rng<PolarMarsaglia>>(new PolarMarsaglia());
auto rng2 = std::shared_ptr<Rng<MTRng>>(new MTRng());
auto rng3 = std::shared_ptr<Rng<MTRng64>>(new MTRng64());
auto rng4 = std::shared_ptr<Rng<FibonacciRng>>(new FibonacciRng());


SUD<GBM, EuropeanPricer, EulerFdm<GBM>, PolarMarsaglia>
             s(sde, pricerPut, fdm, rng1, NSim, NT);
```

```
SUD<GBM, EuropeanPricer, EulerFdm<GBM>, MTRng>
             s2(sde, pricerPut, fdm, rng2, NSim, NT);
SUD<GBM, EuropeanPricer, EulerFdm<GBM>, MTRng64>
             s3(sde, pricerCall, fdm, rng3, NSim, NT);
SUD<GBM, EuropeanPricer, EulerFdm<GBM>, FibonacciRng>
             s4(sde, pricerCall, fdm, rng4, NSim, NT);


auto fn1 = [&]()
    { s.start();std::cout << "\n Polar Marsaglia: " << s.Price(); };
auto fn2 = [&]()
    { s2.start(); std::cout << "\n MT: " << s2.Price(); };
auto fn3 = [&]()
       {s3.start(); std::cout << "\n MT64:" << s3.Price(); };
auto fn4 = [&]()
    { s4.start(); std::cout << "\n Fibonacci: " << s4.Price(); };
```

The tests are (we discuss them in later sections):

```
int main()
{
   SequentialMonteCarlo();
   MonteCarloPPLParallel_Invoke();
   MonteCarloCPP11_Futures();
   MonteCarloCPP11_Threads();
   MonteCarloCPP11_OpenMP();

   return 0;
}
```

The output and run-time performance of each test is:

- Sequential

```
Polar Marsaglia: 5.85298
MT: 5.84891
MT 64: 2.12642
Fibonacci: 2.13177

Elapsed time sequential in seconds: 585.127
```

- PPL tasks

```
Fibonacci: 2.13151
MT 64: 2.13177
MT: 5.84637
Polar Marsaglia: 5.84895

Elapsed time PPL tasks in seconds: 299.583
```

- C++11 futures

```
Fibonacci: 2.13216
MT: 5.84738
MT 64: 2.13177
Polar Marsaglia: 5.84893

Elapsed time C++11 futures in seconds: 305.51
```

- C++11 threads

```
Fibonacci: 2.13207
MT 64: 2.13183
MT: 5.84703
Polar Marsaglia: 5.84892

Elapsed time C++11 threads in seconds: 305.284

Fibonacci: 2.13099
MT: 5.84623
MT 64: 2.13085
Polar Marsaglia: 5.84544

Elapsed time C++11 thread groups in seconds: 308.631
```

- OpenMP, parallel loops

```
 Fibonacci: 2.13195
 MT 64: 2.13177
 MT: 5.84646
 Polar Marsaglia: 5.8489

Elapsed time OpenMP, parallel loops in seconds: 310.269
```

The exact values are P = 5.8462822, C = 2.1333684.

32.2.3 C++ Threads

We now explain the code behind the output. This is a straightforward mapping of the ubiquitous *fork/join pattern* as shown in Figure 32.2; we encapsulate each pricing choice in a dedicated thread. For convenience, we are assuming that there is no shared state or that the shared state is read-only. Furthermore, any return value from the thread function is ignored. We have already discussed the thread functions `fn1`, `fn2`, `fn3` and `fn4` in Section 32.2.2.

```
std::thread t1(fn1);
std::thread t2(fn2);
std::thread t3(fn3);
std::thread t4(fn4);
```

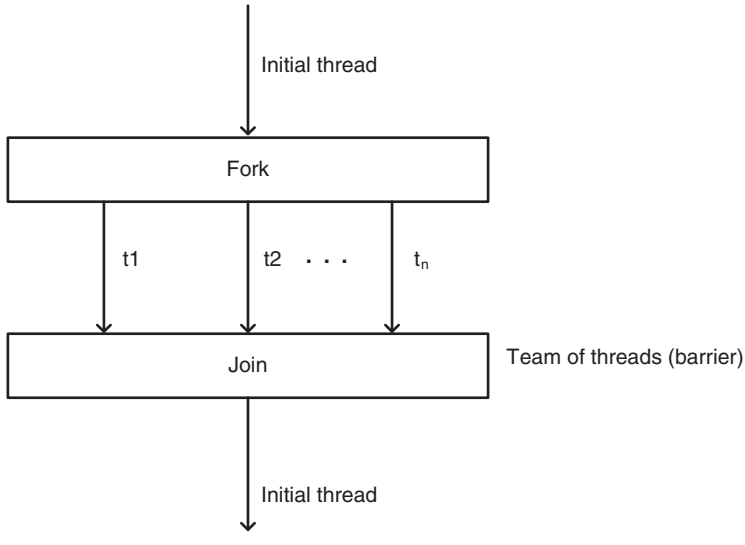


FIGURE 32.2 Fork/join pattern

```
// No shared data so we define 1 barrier/rendezvous point
t1.join();
t2.join();
t3.join();
t4.join();
```

This code does not scale well. A possible workaround is to define a (dynamic) array of threads that we fire and wait to complete:

```
std::vector<std::shared_ptr<std::thread>> threadGroup;
std::vector <std::function<void()>>
      tGroupFunctions = { fn1, fn2, fn3, fn4 };

for (std::size_t i = 0; i < tGroupFunctions.size(); ++i)
{
   threadGroup.emplace_back(std::shared_ptr<std::thread>
                            (new std::thread(tGroupFunctions[i])));
}

for (std::size_t i = 0; i < tGroupFunctions.size(); ++i)
{
   if (threadGroup[i]->joinable())
   {
      threadGroup[i]->join();
   }
}
```

An even more efficient solution will be to create a *thread pool*. The threads could be reycled instead of being started when they are needed and stopped.

32.2.4 C++ Futures

We recall that a C++ `std::future` is a mechanism to access the result of an asynchronous operation. It has a return type that can be used in subsequent code (in contrast to threads). The creator of a future can use a variety of methods to query, wait for or extract a value from the `std::future`. In our case the return type is `void`, the code blocks (by calling `wait()`) until the results become available and then we get:

```
std::future<void> fut1(std::async(fn1));
std::future<void> fut2(std::async(fn2));
std::future<void> fut3(std::async(fn3));
std::future<void> fut4(std::async(fn4));

// Wait for results to become available
fut1.wait();
fut2.wait();
fut3.wait();
fut4.wait();

fut1.get();
fut2.get();
fut3.get();
fut4.get();
```

32.2.5 PPL Parallel Tasks

PPL supports *tasks* that hide low-level details that we experience when using threads. Unlike threads, new tasks do not necessarily begin to execute immediately. Instead, they run when the associated task scheduler removes them from the work queue. The program's performance scales with the number of cores.

A simple case of a PPL parallel task is the `parallel_invoke` function:

```
// Start the work functions
concurrency::parallel_invoke
(
   fn1,
   fn2,
   fn3,
   fn4
);
```

There are overloaded versions of the function `parallel_invoke` that accept up to nine arguments, each argument corresponding to a function to be run in parallel with the other functions. The functions to be run can either complete normally or finish by throwing an exception.

We can reproduce the functionality of `parallel_invoke` by creating a *task group object* and calling its `run` and `wait` methods, for example:

```
// Parallel task using task group
concurrency::task_group tg;
```

```
tg.run(fn1);
tg.run(fn2);
tg.run(fn3);
tg.run(fn4);

tg.wait();
```

A variation that can make better use of threads is to combine the `run` and `wait` steps into a single operation:

```
tg.run(fn1);
tg.run(fn2);
tg.run(fn3);
tg.wait();
tg.run_and_wait(fn4);
```

32.2.6 OpenMP Parallel Loops

OpenMP supports the *Loop Parallelism* pattern (Chapman, Jost and Van der Pas, 2008; Mattson, Sanders and Massingill, 2005). We create an array of computationally intensive functions that we run in parallel:

```
std::vector <std::function<void()>>
         tGroupFunctions = { fn1, fn2, fn3, fn4 };

#pragma omp parallel for
   for (int i = 0; i < tGroupFunctions.size(); ++i)
   {
      tGroupFunctions[i]();
   }
```

We must ensure that the parallel program yields identical results when executed using one thread or using several threads. This is called *sequential equivalence*. This requirement may not always be possible due to round-off errors, for example.

Loop parallelism in *OpenMP* is very easy to program and it has good performance properties.

32.2.7 Boost Thread Group

C++ Concurrency does not have functionality to model groups of threads as one entity. However, we emulate the functionality as an array of threads, as already seen in Section 32.2.3. We recall:

```
std::vector<std::shared_ptr<std::thread>> threadGroup;
std::vector <std::function<void()>>
      tGroupFunctions = { fn1, fn2, fn3, fn4 };
for (std::size_t i = 0; i < tGroupFunctions.size(); ++i)
```

```
{
   threadGroup.emplace_back(std::shared_ptr<std::thread>
         (new std::thread(tGroupFunctions[i])));
}

for (std::size_t i = 0; i < tGroupFunctions.size(); ++i)
{
   if (threadGroup[i]->joinable())
   {
      threadGroup[i]->join();
   }
}
```

Boost.Thread contains the class `thread_group` that supports the creation and management of a group of threads as one entity. The threads in the group are related in some way. The functionality is:

- Create a new thread group with no threads.
- Delete all threads in the group.
- Create a new thread and add it to the group.
- Remove a thread from the group without deleting the thread.
- `join_all()`: call `join()` on each thread in the group.
- `size()`: give the number of threads in the group.

The corresponding code (using the variables from Section 32.2.3) is:

```
boost::thread_group threads;
for (std::size_t i = 0; i< tGroupFunctions.size(); ++i)
{
   threads.create_thread(tGroupFunctions[i]);
}
std::cout << "Thread group size: " << threads.size() << '\n';

threads.join_all();
```

It is possible to create your own class that emulates Boost thread groups if you prefer not to use Boost libraries for this problem.

32.3 A FAMILY OF PREDICTOR–CORRECTOR SCHEMES

We now discuss a scheme to be used for one-factor and multifactor SDEs. It is called the *predictor–corrector* method and is a generalisation of a similar scheme that is used to solve ordinary differential equations (Lambert, 1991). In order to apply it to SDEs we examine the nonlinear problem given by equation (31.7). If we discretise this SDE using the trapezoidal

rule we will get a nonlinear system of equations at each time level that we need to solve using Newton's or Steffensen's method, for example:

$$X_{n+1} = X_n + \{\sigma\mu(X_{n+1}) + (1-\alpha)\mu(X_n)\}\Delta t + \{\beta\sigma(X_{n+1}) + (1-\beta)\sigma(X_n)\}\Delta W_n, \quad n \geq 0$$
where: (32.4)
$$0 \leq \alpha \leq 1, \quad 0 \leq \beta \leq 1.$$

Instead of solving this equation we attempt to linearise it by replacing the 'offending' unknown solution at time level $n+1$ by another known quantity (called the *predictor*) that is hopefully close to it. To this end, we compute the explicit Euler predictor as:

$$\tilde{X}_{n+1} = X_n + \mu(X_n)\Delta t + \sigma(X_n)\Delta W_n, \quad n \geq 0 \tag{32.5}$$

which we subsequently use in the *corrector equation*:

$$\begin{aligned} X_{n+1} = X_n &+ \{\alpha\mu(X_{n+1}) + (1-\alpha)\mu(X_n)\}\Delta t + \{\beta\sigma(\tilde{X}_{n+1}) \\ &+ (1-\beta)\sigma(X_n)\}\Delta W_n, \quad n \geq 0. \end{aligned} \tag{32.6}$$

However, we modify equation (31.7) by using the *corrected drift function*:

$$\overline{\mu}_\beta(x) = \mu(x) - \beta\sigma(x)\frac{\partial\sigma}{\partial x}(x). \tag{32.7}$$

The new corrector equation is given by:

$$\begin{aligned} X_{n+1} = X_n &+ \{\alpha\overline{\mu}_\beta(\tilde{X}_{n+1}) + (1-\alpha)\overline{\mu}_\beta(X_n)\}\Delta t \\ &+ \{\beta\sigma(\tilde{X}_{n+1}) + (1-\beta)\sigma(X_n)\}\Delta W_{n,} \, n \geq 0. \end{aligned} \tag{32.8}$$

The solution of this scheme can be found without the need to use a nonlinear solver. Furthermore, you can customise the scheme to support different levels of implicitness and explicitness in the drift and diffusion terms applied to equations (32.4) and (32.8):

A. Fully explicit ($\alpha = \beta = 0$)
B. Fully implicit ($\alpha = \beta = 1$) (32.9)
C. Implicit in drift explicit in diffusion ($\alpha = 1, \beta = 0$)
D. Symmetric ($\alpha = \beta = {}^1\!/_2$).

We need to determine which combination of parameters results in stable schemes. We have seen that reducing the time steps (which is the same as increasing the number of subdivisions of the interval $[0, T]$) does not always increase the accuracy and may actually lead to serious inaccuracies. This phenomenon is not common when discretising deterministic equations, where convergence tends to be monotonic with respect to the step size.

In order to model the schemes in C++ we need to extend the SDE functionality to include the derivative of the diffusion term and the corrected drift function as defined by equation (32.7) (see also Kloeden, Platen and Schurz, 1997, p. 198). To this end, we propose the class that is essentially an extension of the corresponding class in Chapter 31:

```
// Functions of arity 2 (two input arguments)
template <typename T>
   using FunctionType = std::function<T (const T& arg1, const T& arg2)>;

// Interface to simulate any SDE with Drift/Diffusion
// Use a tuple to aggregate two functions into what is similar to an
// interface in C# or Java.
template <typename T>
   using ISde = std::tuple<FunctionType<T>, FunctionType<T>>;

template <typename T = double> class Sde
{
private:
   FunctionType<T> dr_;
   FunctionType<T> diff_;

   FunctionType<T> drCorr_;
   FunctionType<T> diffDeriv_;

   T ic;   // Initial condition
   T exp_; // Expiry

public:
   Sde() = default;
   Sde(const Sde<T> & sde2, const T& initialcCondition, const T& expiry)
      : dr_(sde2.dr_), diff_(sde2.diff_), ic(initialcCondition),
      drCorr_(sde2.drCorr_),diffDeriv_(sde2.diffDeriv_),exp_(expiry) {}
      Sde(const ISde<T> & functions, const FunctionType<T>&
      driftCorrected,
      const FunctionType<T>& diffusionDerivative,
      const T& initialcCondition,const T& expiry)
      : dr_(std::get<0>(functions)),diff_(std::get<1>(functions)),
         drCorr_(driftCorrected), diffDeriv_(diffusionDerivative),
         ic(initialcCondition), exp_(expiry) {}

   T Drift(const T& t, const T& S)
   {
      return dr_(t,S);
   }

   T Diffusion(const T& t, const T& S)
   {
      return diff_(t,S);
   }

   double DriftCorrected(double t, double x, double B)
   {
      return Drift(t,x)
         - B * Diffusion(t,x) * DiffusionDerivative(t,x);
   }
```

```
    double DiffusionDerivative(double t, double x)
    {
       return diffDeriv_(t, x);
    }


    // Property to set/get initial condition
    double InitialCondition() const
    {
       return ic;
    }

    // Property to set/get time T
    double Expiry() const
    {
       return exp_;
    }
};
```

We see that there are two extra data members compared to the SDE class in Chapter 31. Continuing, we note that the corrector step in equation (32.8) takes the averages of the values at the endpoints of the subinterval in question. This is similar to the approach taken by the *trapezoid rule* (Conte and de Boor, 1981). We now propose a variation of equation (32.8) based on taking values at the midpoint of the subinterval in question (the *midpoint rule*; Conte and de Boor, 1981):

$$X_{n+1} = X_n + \overline{\mu}_\beta(\alpha\tilde{X}_{n+1} + (1-\alpha)X_n)\Delta t + \sigma(\beta\tilde{X}_{n+1} + (1-\beta)X_n)\Delta W_{n,} \quad n \geq 0. \quad (32.10)$$

Summarising, schemes (32.8) and (32.10) use corrected drift based on the trapezoid and midpoint rules, respectively. In general, they are more accurate than the Euler method. Furthermore, scheme (32.8) tends to be somewhat more accurate than scheme (32.10), although hard-and-fast rules unfortunately cannot be given. Mathematical and numerical analysis should be supplemented by extensive numerical testing to ensure that a given method is robust and accurate.

We note that equation (32.6) has the same form as equation (32.8), except that we use the drift term rather than the corrected drift term. Likewise, the midpoint equivalent of scheme (32.10) for the case of normal drift is:

$$X_{n=1} = X_n + \mu(\alpha\tilde{X}_{n+1} + (1-\alpha)X_n)\Delta t + \sigma(\beta\tilde{X}_{n+1} + (1-\beta)X_n)\Delta W_{n,} \quad n \geq 0. \quad (32.11)$$

We have found schemes (32.6) and (32.11) to be less accurate than schemes (32.8) and (32.10). We now show the code for scheme (32.8), which can be integrated into the software framework (the other schemes are similar in structure):

```
class PredictorCorrectorTrapezoidCorrected : public IFdm
{ // 32.8

private:
   double A;
   double B;
```

```
public:
   PredictorCorrectorTrapezoidCorrected() :IFdm() {}

   PredictorCorrectorTrapezoidCorrected
      (const std::shared_ptr<Sde<double>>& stochasticEquation,
      int numSubdivisions, double alpha = 0.5, double beta = 0.5)
      : IFdm(stochasticEquation, numSubdivisions), A(alpha), B(beta) {}

   double advance(double xn, double tn, double dt,
                  double normalVar, double normalVar2) const override
   {
      double Wincr = dtSqrt*normalVar;
      double adjDriftTerm, diffusionTerm;

      // Predictor part; Euler with 'normal' drift function
      double VMid = xn + k * sde->Drift(tn, xn)
                  + sde->Diffusion(tn, xn)* Wincr;

      // Corrector part
      adjDriftTerm = (A*sde->DriftCorrected(tn + dt, VMid, B)
                  + (1.0 - A)*sde->DriftCorrected(tn,xn, B))*k;
      diffusionTerm = (B * sde->Diffusion(tn + dt, VMid)
                  + (1.0 - B)*sde->Diffusion(tn, xn))*Wincr;

      return xn + adjDriftTerm + diffusionTerm;
   }
};
```

We note that this scheme supports the range of schemes as shown in equation (32.9). An example of using this scheme and others is:

```
int NSim = 10'000'000;
int NT = 1000;

double r = 0.2; double sig = 0.3; double T = 1.0;
double ic = 60.0;
auto drift = [=](double t, double S) { return r * S; };
auto gbmDiffusion = [=](double t, double S) { return sig * S; };
auto gbmDiffusionDerivative = [=](double t, double S) { return sig; };
auto driftCorrected = [=](double t, double S)
{
   return drift(t, S)
      - 0.5 * gbmDiffusion(t, S)*gbmDiffusionDerivative(t, S);
};

ISde<double> gbmFunctions = std::make_tuple(drift, gbmDiffusion);

// Create Sdes
auto sdeGbm = std::shared_ptr<Sde<double>>
```

```
          (new Sde<double>(gbmFunctions, driftCorrected,
double K = 110.0;

std::function<double(double)> payoff
   = [=](double x) {return std::max<double>(0.0, x - K); };
std::function<double()> discounter
   = [=]() { return std::exp(-r * T); };
auto pricer = std::shared_ptr<Pricer>
          (new EuropeanPricer(payoff, discounter));
auto fdmPCTrap = std::shared_ptr<PredictorCorrectorTrapezoid>
       (new PredictorCorrectorTrapezoid(sdeGbm, NT));

SUD sPCTrap(sdeGbm, pricer, fdmPCTrap, rng, NSim, NT);
sPCTrap.start();
std::cout << "\nTrapezoid Normal 32.6 " << sPCTrap.Price() << '\n';
```

32.4 AN EXAMPLE (CEV MODEL)

We introduced the *Constant Elasticity of Variance* (CEV) model in Exercise 3 in Chapter 31. In particular, we describe its corresponding SDE by equation (31.2) and its PDE by equation (31.3). The CEV model is an extension of the Black–Scholes model but volatility is no longer constant (Cox and Ross, 1976). It is able to explain the *volatility smile* which we observe in the market, namely that the implied volatility embedded in market option prices is dependent on exercise price and expiration. The assumed volatility function is given by:

$$\tilde{\sigma} = \sigma S^{\beta-1}. \tag{32.12}$$

The model reverts to Black–Scholes when $\beta = 1$, to Cox–Ingersoll–Ross (CIR) when $\beta = 1/2$ and to Vasicek when $\beta = 0$. In general, β is computed by calibration to market data. When $\beta < 0$ the volatility tends to infinity as the asset price goes to zero (β is called the *skew parameter*).

In general, equation (31.1) does not have an analytic solution and we thus resort to numerical methods.

Some of these methods are:

a) Monte Carlo simulation as discussed in this chapter and in Chapter 31. In particular, we examine the effectiveness of the Euler and predictor–corrector methods applied to the CEV model.

b) Closed-form solution for European puts and calls based on the *non-central chi-squared distribution* (Cox and Ross, 1976; Emanuel and MacBeth, 1982).

c) Using the finite difference method to price one-factor and two-factor CEV problems using the *Alternating Direction Explicit* (ADE) method and the *Method of Lines* (MOL) (see Handoko, 2014; Hung Tran, 2015).

We focus on European options in this section. The corresponding formulae are:

$$
\begin{aligned}
&\text{for } \beta < 1\\
&P = Ke^{-rT}[1 - X^2(d, c, d)] - S_0 e - q^T \chi^2(b, c + 2, d)\\
&C = Se^{-qT}[1 - X^2(b; c + 2, d)] - Ke^{-vT}\chi^2(d;\ c, b)\\
&\text{for } \beta > 1\\
&P = Ke^{-rT}[1 - X^2(b, 2 - c, d)] - S_0 e^{-qT}\chi^2(d, -c, b)\\
&C + Se^{-qT}[1 - X^2(d; -c, d)] - Ke^{-rT}\chi^2(b; 2 - c, d)
\end{aligned}
$$

where:

$$
\begin{aligned}
&\mathrm{b} = \frac{[Ke^{-(r-q)T}]^{2(1-\beta)}}{(1-\beta)^2\omega}, \quad c = \frac{1}{1-\beta}\\
&d = \frac{S_0^{2(1-\beta)}}{(1-\beta)^2 w}
\end{aligned}
\tag{32.13}
$$

and

$$
\omega = \frac{\sigma^2}{2(r-q)(\beta-1)}[e^{2(r-q)(\beta-1)^T} - 1].
$$

The generic SDE C++ class in Section 32.3 is composed of four universal function wrappers that will be initialised by specific target functions relating to the CEV model. In other words, we do not need to create a class but instead we instantiate the SDE class. In order to keep things cohesive, we encapsulate the creational process in a factory class. In this case we show the code for both GBM and CEV models:

```
template <typename T>
      std::shared_ptr<Sde<T>> ChooseSde(int choice)
{ // Simple factory method

   // Initial condition for SDE
   if (1 == choice)
   {
      T S0 = 1.0;
      T expiry = 0.5;

      std::cout << "GBM\n";
      // Second test case; GBM dS = a S dt + sig S dW
      T r = 0.08; T sig = 0.3;
      auto Drift = [=](T t, T S) { return r * S; };
      auto gbmDiffusion = [=](T t, T S) { return sig * S; };

      ISde<T> gbmFunctions
            = std::make_tuple(Drift, gbmDiffusion);

      // Create Sde
      return std::shared_ptr<Sde<T>>(new Sde<T>
            (gbmFunctions, S0, expiry));
   }
```

```
    else
    {
       std::cout << "CEV\n";
       T S0 = 100.0;
       T expiry = 0.5;

       double r = 0.05; double sigA = 0.2;
       double beta = 4;
       double sig = sigA*std::pow(S0, 1.0 - beta);
       auto drift = [=](double t, double S) { return r * S; };

       auto cevDiffusion = [=](double t, double S)
             { return sig * std::pow(S, beta); };

       ISde<double> cevFunctions
             = std::make_tuple(drift, cevDiffusion);

       auto cevDiffusionDerivative = [=](double t, double S)
             { return beta*sig*std::pow(S, beta-1); };

       // Create Sde
       return std::shared_ptr<Sde<T>>(new Sde<T>
          (cevFunctions, cevDiffusionDerivative, S0, expiry));
    }
}
```

We show how to call this factory method and price the CEV option:

```
K = 110.0;
double r = 0.05; double sig = 0.2; double T = 0.5;

// Payoff function factories
std::function<double(double)> payoff = [=](double x)
             {return std::max<double>(0.0, x - K); };
//   std::function<double(double)> payoff = [=](double x)
// {return std::max<double>(0.0, K - x); };

std::function<double()> discounter = [=]()
             { return std::exp(-r * T); };
auto pricer = std::shared_ptr<Pricer>
             (new EuropeanPricer(payoff, discounter));

auto cevSde = ChooseSde<double>(2);

// FDM factories
// auto fdmPCMid = std::shared_ptr<PredictorCorrectorMidpointCorrected>
//(new PredictorCorrectorMidpointCorrected(cevSde, NT));

auto fdmPCMid = std::shared_ptr<EulerFdm>(new EulerFdm(cevSde, NT));

// auto fdmPCMid =
```

```
// std::shared_ptr<PredictorCorrectorTrapezoidCorrected>
// (new PredictorCorrectorTrapezoidCorrected(cevSde, NT));

SUD sPCMid(cevSde, pricer, fdmPCMid, rng, NSim, NT);
sPCMid.start();
std::cout << "\nMidpoint Normal CEV 32.10: " << sPCMid.Price();
```

You can run and test the code.

32.5 IMPLEMENTING THE MONTE CARLO METHOD USING THE *ASYNCHRONOUS AGENTS LIBRARY*

In Chapter 30 we gave an introduction to the *Agents Library* and we discussed some examples and benefits of using the library. We now investigate its applicability in combination with the system decomposition methods of Chapter 9 to design a *new style* of Monte Carlo option pricer. We are particularly interested in answering the following questions:

- Q1: Can we easily implement the design blueprints from Chapter 9 as loosely coupled agents that communicate via *message passing*?
- Q2: Are there advantages in knowing that agents have *isolated state* and hence cannot interfere with the internal state of other agents? In particular, do they resolve the issues with race conditions that can arise with random number generators?
- Q3: How does data flow in the system? How do we use *control flow* to start and stop the system?
- Q4: How easy is it to configure the application with various path evolvers, option pricers and other systems in the context diagram? For example, can we add or remove components at run-time? How maintainable is a software system using agents and the *Agents* library?
- Q5: How does the performance of an agent-based solution compare with other designs that we have discussed in this chapter (for example, Section 32.2.2)?
- Q6: How do we configure a system based on agents?

In contrast to the Monte Carlo simulators in which we created a dedicated path for each specific pricer we now use a single agent that creates path information. Then multiple pricers can receive this path as input to produce an option price. The design is shown in Figure 32.3, where we have designed three agents (in this version the agent `MIS` gathers statistics on the running simulator, for example by displaying *percentage complete* information). The agents communicate using message passing. The two types of data are:

- Numeric data `d1` (path information) and `d2` (percentage complete data).
- Control data `c1` (a bool) and `c2` (a bool) to start and stop the application.

We connect the agents in Figure 32.3 by defining *sources* and *targets*. We focus on `PathEvolver` and `Pricer` agents:

```
class PathEvolver : public concurrency::agent
{
```

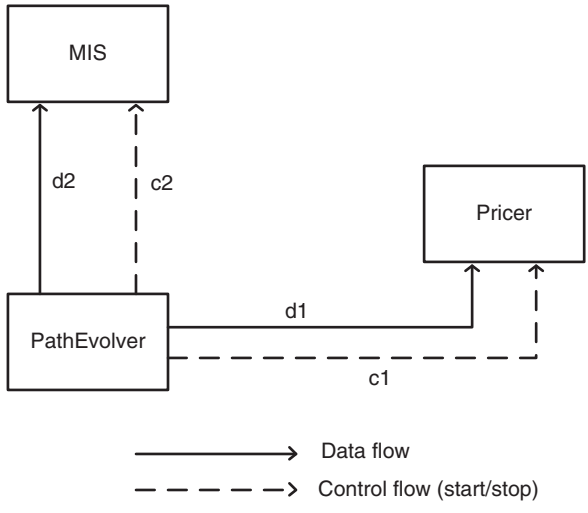


FIGURE 32.3 Message passing for Monte Carlo

```
private:
   // The target buffer to write to.
   concurrency::ITarget<double>& _target;
   // The sentinel value
   concurrency::ITarget<bool>& _stop;

   // MIS
   concurrency::ITarget<std::size_t>& _mis;

   int NT;
   int NSIM;

   std::shared_ptr<Sde> sde;
   double S0;

   // ...
};

class OptionPricer : public concurrency::agent
{
private:
   // The source buffer to read from.
   concurrency::ISource<double>& _source;

   // Send sentinel value to target
   concurrency::ISource<bool>& _stop;

   // ...
};
```

The class `PathEvolver` starts and stops the application. The other agents get their orders from it, as it were. We can see how the agents communicate by examining their `run()` methods:

```
void run()
{
   // Path Evolver
   ThreadSafePrint("Starting producer");
   concurrency::send(_stop, false);
   std::default_random_engine dre;
   std::random_device rd;
   dre.seed(rd());
   std::normal_distribution<double> nor(0.0, 1.0);

   double k = sde->expiration() / static_cast<double>(NT);
   double sqrk = std::sqrt(k);
   double v;

   for (std::size_t n = 0; n < NSIM; ++n)
   {
      concurrency::send(_mis, n);

      double VOld = S0; double VNew;
      double x = 0.0;

      for (long index = 0; index < NT; ++index)
      {
         v = nor(dre);
         // FDM (explicit Euler), equation (9.2) from the text
         VNew = VOld + (k * sde->drift(x, VOld))
                + (sqrk * sde->diffusion(x, VOld) * v);

         VOld = VNew;
         x += k;
      }
      concurrency::send(_target, VNew);
   }
   concurrency::send(_stop, true);

   // Set the agent to the finished state.
   done();
   ThreadSafePrint("Exit producer");
}
```

and

```
// Option Pricer
void run()
{
```

```
    double payoffT;
    double avgPayoffT = 0.0;
    double squaredPayoff = 0.0;
    double sumPriceT = 0.0;
    int NSIM = 0;
    double VNew;

    bool stop;
    while ((stop = concurrency::receive(_stop)) == false)
    {
       // Assemble quantities (postprocessing)
       VNew = concurrency::receive(_source);
       payoffT = myOption.myPayOffFunction(VNew);

       sumPriceT += payoffT;
       avgPayoffT += payoffT / NSIM;
       avgPayoffT *= avgPayoffT;

       squaredPayoff += (payoffT*payoffT);
       NSIM++;
    }


    // Finally, discounting the average price

    price = std::exp(-myOption.r * myOption.T) * sumPriceT
          / static_cast<double>(NSIM);

    done();
    ThreadSafePrint("Exit consumer");
}
```

The management system receives progress information from the path evolver:

```
class MIS : public concurrency::agent
{
private:
    concurrency::ISource<std::size_t>& _mis;
    // Send sentinel value to target
    concurrency::ISource<bool>& _stop;

    std::size_t freq;

public:
    explicit MIS(concurrency::ISource<std::size_t>& mis,
    concurrency::ISource<bool>& stop,std::size_t frequency)
          : _mis(mis), _stop(stop), freq(frequency) {}

    void run()
    {
```

```
        bool stop;
        std::size_t f;
        while ((stop = concurrency::receive(_stop)) == false)
        {
           f = concurrency::receive(_mis);
           if ((f/freq)*freq == f)
           {
              std::cout << f << ",";
           }
        }

        done();
        ThreadSafePrint("Exit MIS");
     }
  };
```

For completeness, we show the utility function to print text on the console in a thread-safe way:

```
void ThreadSafePrint(const std::string& s)
{ // Function to avoid garbled output on the console

   std::mutex my_mutex;
   std::lock_guard<std::mutex> guard(my_mutex);
   std::cout << s << '\n';
}
```

We now discuss the steps that are needed to configure a Monte Carlo option calculator based on the actor model. The model is based on the design decision that the path evolver delivers data to several pricing agents and it can also send progress information to several management and watchdog systems. We represent these requirements in a UML diagram in Figure 32.4. In other words, this diagram gives insight on how to configure an application. In particular, we note the 1:*N* multiplicity relationships between the agents. We discuss one example. First, we create *message buffers* and option-related data:

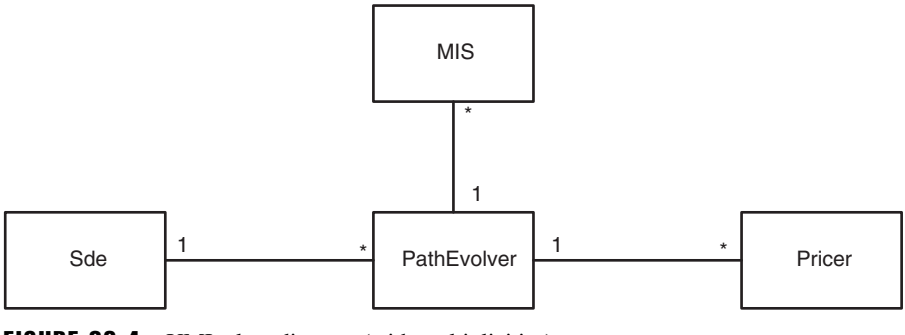


FIGURE 32.4 UML class diagram (with multiplicities)

```
// A message buffer that is shared by the agents.
concurrency::overwrite_buffer<double> buffer;
concurrency::overwrite_buffer<bool> sentinel;

// Communication with MIS agent (draw count)
concurrency::overwrite_buffer<std::size_t> misCount;

// Create and start the producer and consumer agents.
OptionData myOption((
   OptionParams::strike = 65.0, OptionParams::expiration = 0.25,
   OptionParams::volatility = 0.3, OptionParams::dividend = 0.0,
   OptionParams::optionType = 1,OptionParams::interestRate = 0.08));

OptionData myOption2((
   OptionParams::strike = 65.0, OptionParams::expiration = 0.25,
   OptionParams::volatility = 0.3, OptionParams::dividend = 0.0,
   OptionParams::optionType = -1,OptionParams::interestRate= 0.08));

// ATM
OptionData myOption3((
   OptionParams::strike = 60.0, OptionParams::expiration = 0.25,
   OptionParams::volatility = 0.3, OptionParams::dividend = 0.0,
   OptionParams::optionType = -1,OptionParams::interestRate= 0.08));
```

We now create one path evolver and four pricers (which are consumers of path information):

```
int NT = 500;
int NSIM = 1'000'000;

StopWatch<> sw;
sw.Start();

// Create SDE
auto sde1 = std::shared_ptr<Sde>(new Sde(myOption));
double S0 = 60.0;
PathEvolver producer(buffer, sentinel, misCount, NT, NSIM, sde1, S0);

// Define pricers
OptionPricer consumer(buffer, sentinel,myOption);
OptionPricer consumer2(buffer, sentinel, myOption2);
OptionPricer consumer3(buffer, sentinel, myOption);
OptionPricer consumer4(buffer, sentinel, myOption3);
MIS mis(misCount, sentinel, 100'000);
```

Finally, we start the agents, wait for them to complete and then we calculate the option prices:

```
producer.start();
consumer.start();
```

```
consumer2.start();
consumer3.start();
consumer4.start();
mis.start();

// Wait for the agents to finish.
concurrency::agent::wait(&producer);
concurrency::agent::wait(&consumer);
concurrency::agent::wait(&consumer2);
concurrency::agent::wait(&consumer3);
concurrency::agent::wait(&consumer4);
concurrency::agent::wait(&mis);

sw.Stop();
std::cout << "Elapsed time, Raw creation: " << sw.GetTime() << '\n';

std::cout << '\n' << consumer.Price() << '\n';
std::cout << '\n' << consumer2.Price() << '\n';
std::cout << '\n' << consumer3.Price() << '\n';
std::cout << '\n' << consumer4.Price() << '\n';
```

This example shows the essential steps to take when using the *Agents* library in the Monte Carlo option pricing simulator.

32.6 SUMMARY AND CONCLUSIONS

32.6.1 Appendix: C++ for Closed-Form Solution of CEV Option Prices

We tested the finite difference schemes to approximate the CEV option prices against the closed-form solutions in equation (32.13) . We discuss how we coded these latter formulae. We compute put and call prices in the cases when the skew parameter β is less than one and greater than one. When it is identically one we get the Black–Scholes price and the current formula should not be used because we get a divide by zero exception.

We thus have four formulae to compute the CEV option price and we describe how we structured the code based on equation (32.13) . First, the main function is:

```
std::tuple<double, double> CEV(const OptionData& opt, double S, int beta)
{ // Main function to compute exact CEV puts and calls.

   if (beta < 1)
   {
      return CEV_I(opt, S, beta);
   }
   else
   {
      return CEV_II(opt, S, beta);
   }
}
```

We see that put and call option prices are simultaneously computed and returned as a tuple. The cases for the two beta value regimes are:

```
std::tuple<double, double> CEV_I(const OptionData& opt, double S,
                                 int beta)
{ // beta < 1, dof == degrees of freedom, w == noncentrality parameter

   std::cout << "CEV I\n";
   double val1 = 0.0;
   double val2 = 0.0;
   ChiSquaredValues1(opt, S, beta, val1, val2);

   double P = opt.K*std::exp(-opt.r*opt.T) * (1.0 - val1)
            - S*std::exp(-opt.D*opt.T)*val2;
   double C = S*std::exp(-opt.D*opt.T)*(1.0 - val2)
            - opt.K*std::exp(-opt.r*opt.T) * val1;

   auto res =std::make_tuple(P, C);
   return res;
}

std::tuple<double, double> CEV_II(const OptionData& opt, double S,
                                  int beta)
{ // beta > 1, dof == degrees of freedom, w == noncentrality parameter

   std::cout << "CEV II\n";
   double val1 = 0.0;
   double val2 = 0.0;
   ChiSquaredValues2(opt, S, beta, val1, val2);

   double P = opt.K*std::exp(-opt.r*opt.T) * (1.0 - val2)
         - S*std::exp(-opt.D*opt.T)*val1;
   double C = S*std::exp(-opt.D*opt.T)*(1.0 - val1)
         - opt.K*std::exp(-opt.r*opt.T) * val2;

   auto res = std::make_tuple(P, C);
   return res;
}
```

The helper variables b, c, d and w in equation (32.13) are coded as follows:

```
#include <boost/math/distributions.hpp>

void ChiSquaredValues1(const OptionData& opt, double S, int beta,
             double &val1, double &val2)
{

   double r = opt.r; double K = opt.K; double q = opt.D;
   double T = opt.T; double sig = opt.sig;
```

```
    double c = 1.0 / (1.0 - beta);
    double w = 0.5*sig*sig*
    (std::exp(2.0*(r - q)*(beta - 1.0)*T) - 1.0)/((r - q)*(beta - 1.0));
    double b = std::pow(K*std::exp(-(r - q)*T), 2.0*(1.0 - beta))* c *c/ w;
    double d = std::pow(S, 2.0*(1.0 - beta)) * c * c / w;

    boost::math::non_central_chi_squared_distribution<double> dist1(c, b);
    val1 = boost::math::cdf(dist1, d);
    boost::math::non_central_chi_squared_distribution<double>
                                          dist2(c+2, d);

    val2 = boost::math::cdf(dist2, b);
}

void ChiSquaredValues2(const OptionData& opt, double S, int beta,
                           double &val1, double &val2)
{

    double r = opt.r; double K = opt.K; double q = opt.D;
    double T = opt.T; double sig = opt.sig;

    double c = 1.0 / (1.0 - beta);
    double w = 0.5*sig*sig*
    (std::exp(2.0*(r - q)*(beta - 1.0)*T) - 1.0) / ((r - q)*(beta - 1.0));
    double b = std::pow(K*std::exp(-(r - q)*T), 2.0*(1.0 - beta))* c * c/w;
    double d = std::pow(S, 2.0*(1.0 - beta)) * c * c / w;

    boost::math::non_central_chi_squared_distribution<double> dist1(-c, b);
    val1 = boost::math::cdf(dist1, d);
    boost::math::non_central_chi_squared_distribution<double>
                                          dist2(2-c, d);
    val2 = boost::math::cdf(dist2, b);
}
```

As a sanity check we create a function that tests *put–call parity* (just in case there is a bug in the code or the formulae break down for some reason):

```
void PutCallParity(const OptionData& opt, double S, int beta)
{
    // Calculate prices
    auto tup = CEV(opt, S, beta);

    double P = std::get<0>(tup);
    double C = std::get<1>(tup);

    double lhs = C + opt.K*std::exp(-opt.r*opt.T);
    double rhs = P + S;

    std::cout << "put call parity " << lhs - rhs << '\n';
}
```

Finally, we show how to use this code to compute CEV option prices:

```
// Using Boost Parameter
OptionData   opt((OptionParams::strike=110.0,OptionParams::expiration=0.5,
OptionParams::volatility = 0.2, OptionParams::dividend = 0.0,
OptionParams::optionType = -1, OptionParams::interestRate = 0.05));

double beta = 4.0;
double S = 100.0;
double sig = opt.sig*std::pow(S, 1.0 - beta);
opt.sig = sig;
auto res = CEV(opt, S, beta);
std::cout << std::get<0>(res) << ", " << std::get<1>(res) << '\n';

PutCallParity(opt, S, beta);
```

We conclude this appendix with some remarks on the *non-central chi-squared distribution*. First, we create such a distribution as follows:

```
// Initial test
double dof = 3.0; double lambda = 1.5;
boost::math::non_central_chi_squared_distribution<double>
       dist(dof, lambda);

double x = 9.0;
std::cout << boost::math::cdf(dist, x) << '\n';
```

Furthermore, we use the functionality to produce random variates (based on Algorithm 3.5, p. 124 of Glasserman, 2004) to generate variates of the non-central chi-squared distribution which can be used to generate paths of the CIR model by sampling from the transition density (Glasserman, 2004). A test case is:

```
#include <boost/random.hpp>
// Simple 101 example to show how to use
double k = 2; double lambda = 2.0;
std::random_device rd;

std::default_random_engine eng(rd());
boost::random::non_central_chi_squared_distribution<> dist(k, lambda);

std::cout << "Variate: " << dist(eng) << '\n';
```

We note that we have discussed other related properties of this distribution in Section 15.7.1. We note that C++ does not support the non-central chi-squared distribution.

32.7 EXERCISES AND PROJECTS

1. (Making Random Number Classes More Generic)

In Chapter 31 and Section 32.2.1 we created a class hierarchy consisting of classes that encapsulate specific random number engines from C++ `<random>` and Boost *Random*. This is an inelegant solution in the long term because of code duplication. The objective of this exercise is to apply a signature-based approach in which we create a class containing a customisable function object (or possibly two) that can be instantiated by functionality in the aforementioned libraries.

What are the advantages of this approach?

2. (C++ Futures with Non-`void` Return Types)

Modify the code in Section 32.2.3 to create futures that return a `double` value corresponding to an option price. Test the new code. Furthermore, investigate the consequences of not calling `wait()` for each future. Do you get the same results as before?

3. (Data Feeds and Computation, Project)

The goal of this exercise is to design a special kind of pipeline pattern using the *Asynchronous Agents Library*. We sketch a simple case of a *datafeed* application that occurs in many business activities, for example statistical analysis, profit and loss calculations and processing workflow management (Fayad, Schmidt and Johnson, 1999). In general, we speak of a *pipeline model* (POSA, 1996) consisting of:

- A long stream of input.
- A series of suboperations (known as stages or *filters*) through which each unit of input must be processed.
- Each filter can handle a different unit of input at a given time.

We reduce the scope by taking the example in Figure 32.5 in which we identify three filters:

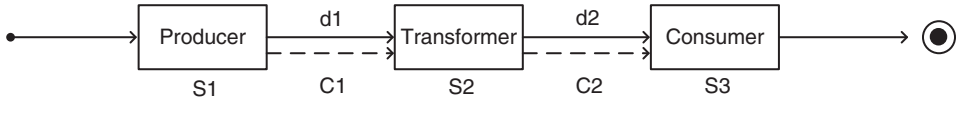
- S1 (Producer of data).
- S2 (Transformer of data).
- S3 (Consumer of data).

In general, S1 produces blocks of data. Each block is sent to S2 which is transformed to another format. The transformed data is sent to S3 (for example, it displays the data on a user screen). The application is started and stopped by system S1.

Answer the following questions:

a) Design the system depicted in Figure 32.5 using the *Asynchronous Agents Library*. Base your design on the approach in Section 32.5. Pay particular attention to using the most suitable message block types that the filters share in order to model both the data flow and the control flow in the system.

b) Consider the case of a single instance of systems S1 and S2 and multiple instances of system S3. How would you configure the application in this case?



C1, C2: start/stop (control flow)
D1, D2: data transfer (data flow)

FIGURE 32.5 Workflow example

c) Specialise the design to price batches of options using the finite difference method (as in Chapter 21). In this case system S1 produces a continuous stream of option data, system S2 computes each option price and system S3 displays all option prices on the console.

d) How would you design the application in Figure 32.5 using the design approach taken in Chapter 9 in combination with multithreading?

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

APPENDIX 1

Multiple-Precision Arithmetic

A1.1 INTRODUCTION AND OBJECTIVES

The goal of this appendix is to give an introduction to *multiple-precision arithmetic* (sometimes called *arbitrary-precision arithmetic* or *bignum arithmetic*) for calculations on numbers whose digits of precision are only limited by the available memory of the host system. In particular, we introduce the *Boost C++ multiprecision library* that supports high-precision decimal and integer types beyond the 64 bits of precision in C++11.

Multiple precision is useful in applications in which the speed of arithmetic is not a hard requirement. Multiple-precision computations are slower than computations using floating-point arithmetic. So, why and when would we choose a multiple-precision solution? Some scenarios are:

a) Avoiding overflow in computations which can plague fixed-precision computations. Overflow can occur for many reasons, for example when the result of a computation is a large number that does not fit into a floating-point value or when dividing a number by a very small number.
b) In arithmetic computations (using addition, for example) in which the input is floating point but the result exceeds the machine word size. In this case we could decide to define the result as a multiple-precision data type.
c) Many mathematical and numerical methods use large numbers such as factorials and the result of computing the confluent hypergeometric function, to mention just two examples.
d) Applications in which artificial limits and overflows are not relevant.
e) Computing fundamental mathematical constants to a million (or more) digits. Probably the most famous example is computing π to a million digits accuracy. These are examples that appeal to the popular imagination.
f) Public-key cryptography and data security. In this domain special algorithms employ arithmetic with integer keys having hundreds of digits.

Cases a) to d) would probably be the ones of most relevance to computational finance. We discuss them in some detail in the coming sections.

We mentioned that multiple-precision arithmetic is slower than floating-point arithmetic. Multiple-precision arithmetic is usually carried out in software. Floating-point arithmetic is

carried out in hardware. For this reason we must decide when to use multiple-precision arithmetic in applications.

The main objective of this appendix is to acquaint the reader with multiprecision computations in Boost.

A1.2 MULTIPRECISION DATA TYPES

The library has support for a number of numeric data types having extended precision:

- Fixed-precision and arbitrary-precision integer types (each type can be signed or unsigned), for example with 128-, 256-, 512- and 1024-bit precision.
- Arbitrary-precision rational numbers.
- Support for dynamic memory allocation; we can store integer values in dynamic memory.
- Checking (or not checking) for errors and inconsistent computation, for example deciding whether to throw a range error when attempting to perform a bitwise operation on a negative value or just returning the value and not the bit pattern when checking is not in force.
- Import and export of raw bits of high-precision integers from and to external storage.

The template classes are defined as follows and we shall give examples to show the intent of their template parameters:

```
typedef unspecified-type limb_type;

enum cpp_integer_type { signed_magnitude, unsigned_magnitude };
enum cpp_int_check_type { checked, unchecked };

template <unsigned MinBits = 0,unsigned MaxBits = 0,
      cpp_integer_type SignType = signed_magnitude,
      cpp_int_check_type Checked = unchecked,
      class Allocator = std::allocator<limb_type> >
class cpp_int_backend;

typedef number<cpp_int_backend<> > cpp_int;
typedef rational_adaptor<cpp_int_backend<> > cpp_rational_backend;
typedef number<cpp_rational_backend> cpp_rational;

// Fixed precision unsigned types
typedef number<cpp_int_backend<128, 128, unsigned_magnitude,
                        unchecked, void> >  uint128_t;
typedef number<cpp_int_backend<256, 256, unsigned_magnitude,
                        unchecked, void> >  uint256_t;
// and many more...
```

The library has support for high-precision floating-point numbers based on a given number of decimal digits. In general, the corresponding classes allocate no memory; however,

arithmetic operations become very expensive as the number of digits grows. Some features and properties are:

- Analogues of the IEEE single, double and quad float data types as well as the Intel extended double type.
- Specialisations of `std::numeric_limits` for these types.
- The ability to define *base 2* and *base 10* digit types.
- The types support infinities (an infinity is generated when a result would overflow) and NaNs (which are generated for any mathematically undefined operation).

The library supports both *radix 2* and *radix 10* types or number systems. Number systems use a finite number of distinct symbols. Each symbol is called a *digit* and it denotes a quantity. The number of these distinct symbols is known as the *base* or *radix* of the number system. The *binary numbers system* (base 2) has the digit symbols {0, 1}, the *decimal number system* (base 10) has the digit symbols {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} while the *hexadecimal number system* (base 16) has digit symbols {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E}. *Numbers* are formed by juxtaposing digits.

The template classes are defined as:

```
enum digit_base_type
{
   digit_base_2 = 2,
   digit_base_10 = 10
};

template <unsigned Digits, digit_base_type base = digit_base_10,
            class Allocator = void, class Exponent = int,
            ExponentMin = 0, ExponentMax = 0>
   class cpp_bin_float;

typedef number<cpp_bin_float<50> > cpp_bin_float_50;
typedef number<cpp_bin_float<100> > cpp_bin_float_100;

typedef number<backends::cpp_bin_float<24,
            backends::digit_base_2,void,boost::int16_t,-126,127>, et_off>
cpp_bin_float_single;

typedef number<backends::cpp_bin_float<53,
         backends::digit_base_2,
         void, boost::int16_t, -1022, 1023>, et_off>
cpp_bin_float_double;

typedef number<backends::cpp_bin_float<64,
         backends::digit_base_2,
         void, boost::int16_t, -16382, 16383>, et_off>
cpp_bin_float_double_extended;

typedef number<backends::cpp_bin_float<113,
         backends::digit_base_2,
         void, boost::int16_t, -16382, 16383>, et_off>
cpp_bin_float_quad;
```

There are also classes in the library to provide arithmetic types at 50 and 100 decimal digits accuracy, respectively:

```
template <unsigned Digits10, class ExponentType=boost::int32_t, class
Allocator=void>
class cpp_dec_float;

typedef number<cpp_dec_float<50> > cpp_dec_float_50;
typedef number<cpp_dec_float<100> > cpp_dec_float_100;
```

In general, Boost multiprecision will be used in situations in which we wish to avoid overflow or where we need to compute very large numbers, either permanently or for use in other applications.

Some typical questions are:

- S1: Do we wish to have arbitrary precision or fixed precision?
- S2: Do we wish to use binary or decimal number systems?
- S3: Choosing between integers and floating-point numbers.
- S4: Determining what the performance and accuracy requirements of your application are.

We now introduce the library by way of some easy examples to help us get used to the syntax.

A1.3 INITIAL EXAMPLES AND APPLICATIONS

In this section we introduce the multiple-precision data types and how to use them in numerical applications.

A1.3.1 Computing the Area of a Circle

No book would be complete if there was not at least a mention of how to compute π to a given accuracy. This number corresponds to the area of a circle of radius 1:

```
template<typename T>
 T CircleArea(const T& r)
{
   using boost::math::constants::pi;
   return pi<T>() * r * r;
}

// radius == 1, so area = pi
const int r = 1;
float r_f = r;
float a_f = CircleArea(r_f);

double r_d = static_cast<double>(r);
double a_d = CircleArea(r_d);
```

```
cpp_dec_float_50 r_mp = static_cast<cpp_dec_float_50>(r);
cpp_dec_float_50 a_mp = CircleArea(r_mp);

cpp_dec_float_100 r_mp100 = static_cast<cpp_dec_float_100>(r);
cpp_dec_float_100 a_mp100 = CircleArea(r_mp100);

std::cout << std::setprecision(std::numeric_limits<float>::digits10)
          << a_f << std::endl;

std::cout << std::setprecision(std::numeric_limits<double>::digits10)
          << a_d << std::endl;

std::cout
<< std::setprecision(std::numeric_limits<cpp_dec_float_50>::digits10)
                    << a_mp << std::endl;

std::cout
<< std::setprecision(std::numeric_limits<cpp_dec_float_100>::digits10)
           << a_mp100 << std::endl;
```

The output from this program is:

```
3.14159
3.14159265358979
3.14159265358979323846264338327950288419716939937510
3.14159265358979323846264338327950288419716939937510582097494459230781640
628620899862803482534211706 68
```

The exact value is:

```
3.14159265358979323846264338327950288419716939937510582097494459230781
640628620899862803482534211706798
```

The above code was written for didactical reasons to show how to define and display multiprecision data. We now discuss how Archimedes of Syracuse actually computed π (see Dörrie, 1965 for the details). Archimedes approximated the circumference of a circle of radius r (we take $r = \frac{1}{2}$) by the perimeter a of the circumscribed polygon and the perimeter b of the inscribed polygon. The algorithm is based on the fact that the *harmonic mean* of two numbers x and y is smaller than their *geometric mean*:

$$\frac{2xy}{x+y} < \sqrt{xy}.$$

The Archimedes recursion formulae for the perimeters a and b are:

$$a_{n+1} = \frac{2a_n b_n}{a_n + b_n}, \quad b_{n+1} = \sqrt{b_n a_{n+1}}, n \neq 0$$

with the initial values:

$$a_0 = 2\sqrt{3},\ b_0 = 3 \text{ (radius} = 1/2).$$

The code for this algorithm is:

```
template <typename T>
   void ArchimedesPi(int N)
{ // N == number of iterations

   T an, anp1, bn, bnp1;
   an = T(2.0)*boost::multiprecision::sqrt(T(3.0)); bn = T(3.0);

   for (int n = 1; n <= N; ++n)
   {
      anp1 = T(2.0)*an*bn / (an + bn);
      bnp1 = boost::multiprecision::sqrt(bn*anp1);

      an = anp1; bn = bnp1;
   }


   std::cout << "N, value: "
      << std::setprecision(std::numeric_limits<T>::digits10)
      << N << "/" << bn << "\n\n";

   std::cout << "Error: " << std::setprecision(8)
             << boost::multiprecision::abs(bn - an) << "\n\n";

}
```

We call this function a number of times and we display a subset of the output:

```
int N = 200;

for (int n = 0; n <= N; n += 10)
{
   ArchimedesPi< cpp_dec_float_100>(n);
}
```

The output is:

```
N, value:
20/3.14159265358966268270170617109077471998597937915933465741202108437728760536295088473800185954626223 5

Error: 3.9166728e-13

N, value:
40/3.14159265358979323846264326453973002586817544027216332053216522252735651164593188018436438521444669 2
```

```
Error: 3.5621932e-25

N, value:
60/3.14159265358979323846264338327950288408917620507073465384865924366109969287718976336684872480311458

Error: 3.2397958e-37

N, value:
80/3.14159265358979323846264338327950288419716939937500760173689723236079737032492766828422638159104739

Error: 2.9465771e-49

N, value:
100/3.14159265358979323846264338327950288419716939937510582097494450297793979003832755187039605537355163 1

Error: 2.6798963e-61

N, value:
120/3.14159265358979323846264338327950288419716939937510582097494459230781640620496394913552798253765045 5

Error: 2.4373515e-73

N, value:
140/3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862796093340005835 4

Error: 2.2167583e-85

N, value:
160/3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211700 1

Error: 2.0161299e-97

N, value:
180/3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706 8

Error: 1.833314e-109
```

We thus see that the approximation improves as we increase the number of edges of the polygons.

A1.3.2 Numerical Differentiation

In Chapter 22 (Section 22.3) we discussed several techniques to approximate the first and second derivatives of a univariate function using divided differences and cubic splines. We also

discuss *catastrophic cancellation* that occurs when we subtract two nearly equal numbers. In this case this operation increases relative accuracy substantially more than it increases absolute error. This phenomenon is also called *loss of significance*. In the case of approximating derivatives we have seen in Section 22.3.5 that there is a critical value of the step size below which the approximation will give disastrous results. We computed this critical value (approximately 0.003) for the exponential function using floating-point arithmetic.

We use the following generic divided difference formulae to approximate a function's derivatives:

```
template <typename T>
   T firstDividedDifference(const FunctionType<T>& f, T xval, T h)
{ // 2nd order approximation h^2/6

   return (f(xval+h) - f(xval-h)) / (T(2) * h);
}

template <typename T>
   T RichardsonfirstDividedDifference(const FunctionType<T>& f,
                                      T xval, T h)
{ // 4th order approximation to df/dx using Richardson extrapolation

   T h2 = h * T(0.5);
   T f1 = 4*firstDividedDifference(f, xval, h2);
   T f2 = firstDividedDifference(f, xval, h);

   return (f1 - f2) / 3;

}

template <typename T>
   T firstDividedDifferenceOrder4(const FunctionType<T>& f, T xval, T h)
{ // 4th order approximation h^4/30

   return (f(xval -2*h) - 8*f(xval -h) + 8*f(xval + h)
          - f(xval + 2*h)) / (12 * h);
}

template <typename T>
   T secondDividedDifference(const FunctionType<T>& f, T xval, T h)
{ // 2nd order h^2/12

   return (f(xval+h) - (2 * f(xval)) + f(xval-h)) / (h * h);
}

template <typename T>
   T secondDividedDifferenceOrder4(const FunctionType<T>& f, T xval, T h)
{ // 4th order h^4/90

   return (-f(xval - 2*h) + 16*f(xval - h) - 30*f(xval)
           + 16*f(xval + h) - f(xval + 2*h)) / (12 * h * h);
}
```

We can use these functions with a range of functions and data types, for example:

```
template <typename T>
   using FunctionType = std::function<T(const T& t)>;

using value_type = boost::multiprecision::cpp_dec_float_100;

value_type h = 0.001;

FunctionType<value_type> fun = [](value_type x) -> value_type
             { return exp(x); };


std::cout << std::setprecision(20) <<  "1st divided difference "
           << firstDividedDifference <value_type> (fun, x, h) << ", "
           << firstDividedDifferenceOrder4 <value_type>(fun, x, h)
           << ",  "
           << RichardsonfirstDividedDifference<value_type>(fun, x, h)
           << "\n2nd divided difference "
           <<  secondDividedDifference<value_type>(fun,x,h)
           << ","
           << secondDividedDifferenceOrder4<value_type>(fun,x,h);
```

In general, we can avoid catastrophic cancellation if we employ a stable algorithm to solve well-posed problems. You can test this code using multiprecision data types.

A1.3.3 `std:numeric_limits<>` Constants and Functions

In general, most of the constants and functions in this class template are meaningful for multiprecision types (there are some exceptions, for example *infinity* is meaningless for arbitrary-precision arithmetic backends). The following functions print some of these constants and functions:

```
template <typename T>
   void ExamineType(const std::string& s)
{
   std::cout << "\nRadix (integer) base: "
             << std::numeric_limits<T>::radix << "\n";
   std::cout << "Type: " << s
             << ", number of representable base-10 digits: "
             << std::numeric_limits<T>::digits10 << "\n";
   std::cout << "Rounding style: "
             << std::numeric_limits<T>::round_style << "\n";
   std::cout << "Largest finite value: "
             << std::numeric_limits<T>::max() << "\n";
   std::cout << "Smallest finite value: "
             << std::numeric_limits<T>::min() << "\n";
   std::cout << "Infinity: "
             << std::numeric_limits<T>::infinity() << "\n";
   std::cout << "Epsilon: "
             << std::numeric_limits<T>::epsilon() << "\n";
```

```
    std::cout << "Return quite NaN: "
              << std::numeric_limits<T>::quiet_NaN() << "\n";
    std::cout << "Max rounding error: "
              << std::numeric_limits<T>::round_error() << "\n\n";
}
```

Some examples are:

```
ExamineType<float>(std::string("float"));
ExamineType<double>(std::string("double"));
ExamineType<boost::multiprecision::cpp_dec_float_50>
              (std::string("cpp_dec_float_50"));
ExamineType<boost::multiprecision::cpp_dec_float_100>
              ( std::string("cpp_dec_float_100"));
```

The output is:

```
Radix (integer) base: 2
Type: float, number of representable base-10 digits: 6
Rounding style: 1
Largest finite value: 3.40282e+38
Smallest finite value: 1.17549e-38
Infinity: inf
Epsilon: 1.19209e-07
Return quite NaN: nan
Max rounding error: 0.5


Radix (integer) base: 2
Type: double, number of representable base-10 digits: 15
Rounding style: 1
Largest finite value: 1.79769e+308
Smallest finite value: 2.22507e-308
Infinity: inf
Epsilon: 2.22045e-16
Return quite NaN: nan
Max rounding error: 0.5


Radix (integer) base: 10
Type: cpp_dec_float_50, number of representable base-10 digits: 50
Rounding style: -1
Largest finite value: 1e+67108864
Smallest finite value: 1e-67108864
Infinity: inf
Epsilon: 1e-49
Return quite NaN: nan
Max rounding error: 0.5


Radix (integer) base: 10
Type: cpp_dec_float_100, number of representable base-10 digits: 100
Rounding style: -1
Largest finite value: 1e+67108864
Smallest finite value: 1e-67108864
```

```
Infinity: inf
Epsilon: 1e-99
Return quite NaN: nan
Max rounding error: 0.5
```

Floating-point types cannot store all real values (those in the set $\mathbb{R}$) exactly. For example, 0.5 can be stored exactly in a binary floating point but not 0.1. What is stored is the nearest representable real value. Fixed-point types (usually decimal) are also defined as exact, in that they only store a fixed precision. The results of computations are rounded up or down.

A1.4 MULTIPLE PRECISION AND SPECIAL FUNCTIONS

Many numerical applications involve computation with mathematical functions, series summation and testing for equality of floating-point types. When using 32-bit or 64-bit types we can experience overflow, underflow and catastrophic cancellation which can be resolved in many cases by using multiprecision data types.

We take an example of computing Legendre polynomials. The code is:

```
int main()
{
   namespace mp = boost::multiprecision;
   using namespace boost::math;

   std::cout << "Degree n of Legendre polynomial: ";
   int n; std::cin >> n;
   std::cout << "Degree m for associated Legendre polynomial: ";
   int m; std::cin >> m;

   // '101' test
   double x = 0.5;
   std::cout<<std::setprecision(16);
   try
   {
      // Legendre 1st and 2nd kinds
      std::cout << legendre_p<double>(n, x) << std::endl;
      std::cout << legendre_q<double>(n, x) << std::endl;

      // Associated Legendre polynomials
      std::cout << legendre_p<double>(n, m, x) << std::endl;
   }
   catch (std::exception& e)        // x in closed range [-1,1]
   {
      std::cout << e.what();
   }

   {
   using value_type = boost::multiprecision::cpp_dec_float_50;
   value_type z = 0.5;
   std::cout << std::setprecision(50);
```

```
    try
    {
       std::cout << legendre_p<value_type>(n, z);
       std::cout << legendre_q<value_type>(n, z);

       // Associated Legendre polynomials
       std::cout << legendre_p<value_type>(n, m, z);
    }
    catch (std::exception& e)       // x in closed range [-1,1]
    {
       std::cout << e.what();
    }
}

{

   using value_type
         = boost::multiprecision::cpp_dec_float_100;
   value_type z = 0.5;
   std::cout << std::setprecision(100);

   try
   {
      std::cout << legendre_p<value_type>(n, z);
      std::cout << legendre_q<value_type>(n, z);

      // Associated Legendre polynomials
      std::cout << legendre_p<value_type>(n, m, z);
   }
   catch (std::exception& e)   // x in closed range [-1,1]
   {
      std::cout << e.what();
   }
}

return 0;
}
```

See also Exercises 4 and 5.

A1.5 NUMERICAL SOLUTION OF ORDINARY DIFFERENTIAL EQUATIONS

In Chapters 24 and 25 we discussed the numerical solution of ODEs. In Chapter 25 we discussed the application of the Boost `odeint` library to such problems. We note that this library supports multiprecision data types. This is a useful approach when we wish to get more accurate results or when we wish to avoid round-off errors in ill-conditioned or stiff systems.

As an example we take the case of computing *Fresnel integrals* as discussed in Exercise 5 of Chapter 8. These integrals cannot be evaluated in closed form in terms of elementary functions except in special cases. For this reason we resort to numerical methods, as we have

seen in Exercise 8.5, in this case as the solution of an ODE. The integrand oscillates but the magnitude of the oscillations decreases for larger values of the independent variable.

The ODE class using multiprecision data in this case is:

```
#include <iostream>
#include <vector>
#include <cmath>

#include <boost/numeric/odeint.hpp>
#include <boost/numeric/odeint/stepper/bulirsch_stoer.hpp>
#include <boost/multiprecision/cpp_dec_float.hpp>
#include <boost/multiprecision/cpp_int.hpp>

// The type of container used to hold the state vector

// C++ types
//using value_type = double;
//using value_type = float;

// Boost types
//using value_type = boost::multiprecision::cpp_dec_float_50;
using value_type = boost::multiprecision::cpp_dec_float_100;

using state_type = std::vector<value_type>;

namespace Bode = boost::numeric::odeint;
using namespace boost::numeric::odeint;

// The rhs of x' = f(x) defined as a class
const value_type piD2
      = static_cast<value_type>(3.14159265358979323846 * 0.5);

class OdeExp
{
public:

public:
   OdeExp() { }

   void operator() (const state_type &x, state_type &dxdt,
             const value_type t)
   {
      // C++ standard
      //   dxdt[0] = std::cos(piD2*t*t);
      //   dxdt[0] = std::sin(piD2*t*t);

      // Boost Multiprecision
      // dxdt[0] = boost::multiprecision::sin(piD2*t*t);
      dxdt[0] = boost::multiprecision::cos(piD2*t*t);
   }
};
```

```
class WriteOutput
{
public:

public:
   WriteOutput() { }

   void operator ()( const state_type &x , const value_type t )
   {
      std::cout << "Time and value (FO): " << t << " "
                << x[0] << std::endl;
   }
};
```

A test program is:

```
int main()
{
   namespace Bode = boost::numeric::odeint;

   // Initial condition
   state_type x(1);
   value_type A =value_type(0.0);
   x[0] = A;


   // [L,U]
   value_type L = value_type(0.000000);
   value_type U = value_type(6.0);
   value_type dt = value_type(1.0e-15);

   WriteOutput wo;
   OdeExp ode;

   // std::size_t steps
   //    = Bode::integrate(ode, x, L, U, dt, std::ref(wo));

   // Bode::runge_kutta_dopri5<state_type, value_type> myStepper;
   Bode::bulirsch_stoer<state_type, value_type> myStepper;

   std::size_t steps = Bode::integrate_adaptive (myStepper,ode,x,L,U,dt,
   std::ref(wo));

   std::cout.precision(100);
   std::cout << "Approximate answer, steps: " << x[0] << ", "
             << steps << std::endl;

   return 0;
}
```

You can experiment with this example by choosing different steppers, mesh sizes and integrate functions. A stress test is to determine how to get a good approximation to the limit cases:

$$\int_0^\infty \cos\frac{\pi}{2}t^2 dt = \int_0^\infty \sin\frac{\pi}{2}t^2 dt = 0.5.$$

We also note that the standard mathematical functions do not work with multiprecision types. There is, however, support for these functions in the namespace `boost::multiprecision`. This means that if you wish to write portable code then these functions should be wrapped in some user-defined namespace, for example:

```
template <typename T>
   T cos(T x)
{
   return boost::multiprecision::cos(x);
}
```

Fresnel integrals have applications in optics and engineering (Ditchburn, 1991).

The technique of transforming an integral equation to an ODE has many applications and using Boost *odeint* is a useful tool to compute the numerical solution of such an ODE.

A1.6 GENERATING RANDOM NUMBERS

We now investigate the possibility of using multiprecision data types with the C++ *Random Number* library, for example generating random integers with large bit counts. The example taken here is based on the Boost *Multiprecision* library online documentation. As mentioned in that documentation, this is a research topic.

We discuss two random number generators:

- `generate_canonical`: generates a random floating number in the range [0, 1]. This algorithm supports the ability to increase *entropy*, which is a better measure of true randomness than random numbers produced in software.
- *`independent_bits_engine`*: a random number engine adapter that produces random numbers with a different number of bits than that of the wrapped engine.

The algorithm `generate_canonical` has input parameters for the type of the wrapped engine, the number of bits that the generated numbers should have and the type of the generated random numbers (should be unsigned integers). An example is:

```
template <typename T>
   void CanonicalGenerator(int number, int precision)
{
   std::random_device rd;
   std::mt19937 gen(rd());
   const std::size_t bits = std::numeric_limits<T>::digits;
```

```
    std::cout << "\n\n";
    std::cout << std::setprecision(precision);
    for (int n = 0; n<number; ++n)
    {
       std::cout << std::generate_canonical<T, bits>(gen) << ", ";
    }
    std::cout << "\n\n";
}
```

A test program is:

```
// Generate canonical
int number = 4;
CanonicalGenerator<double>(number, 16);

using boost::multiprecision::cpp_dec_float_50;
CanonicalGenerator<double>(number, 50);
```

Typical output is:

```
0.6443861122559016, 0.9267760353530851, 0.8461941290458814, 0.929694695848
9013,

0.80979306562688702086916237021796405315399169921875,
0.86959590730995528495839153038104996085166931152344,
0.61351228154489578781038972010719589889049530029297,
0.24403308304902970871097522831405512988567352294922,
```

The algorithm *independent_bits_engine* has input parameters for the type of the wrapped engine, the number of bits that the generated numbers should have and the type of the generated random numbers (should be unsigned integers). An example is:

```
std::random_device rd;
const std::size_t W = 32;
std::independent_bits_engine<std::mt19937, W, std::uint_fast64_t>
            generator(rd());

std::cout << "\n\n";
std::cout << std::setprecision(precision);
for (int n = 0; n<number; ++n)
{
   std::cout << generator() << ", ";
}
std::cout << "\n\n";

// Construct the underlying engine with a seed sequence
std::seed_seq seedSeq = { -1,0, 1 };
std::independent_bits_engine<std::mt19937, W, std::uint_fast64_t>
          generator2(rd());
std::cout << "\n\n";
```

```
for (int n = 0; n<number; ++n)
{
   std::cout << generator2() << ", ";
}
std::cout << "\n\n";
```

We got the following output when we ran this code:

```
3224128699, 1790518196, 367697637, 3964670867,

1545172431, 2759616371, 502877303, 3738674815,
```

For completeness, we give some examples of `std::seed_seq` producing values that are distributed over an entire 32-bit range. This provides a way to seed a large number of random number engines or to seed a generator that requires a lot of entropy, given a small seed or a poorly distributed initial seed sequence:

```
void SeedSequence()
{
   // Initial seed sequence of length 0
   std::seed_seq s1;

   // Iterate over a range
   int a[4] = { 1,2,3,4 };
   std::seed_seq s2(a, a + 4);

   // Initialiser list
   std::seed_seq s3 = { -1,0, 1 };
}
```

A1.7 SUMMARY AND CONCLUSIONS

In this appendix we gave an overview of the *Boost Multiprecision* library. It has support for high-precision integral and floating-point data types. It is possible to carry out the usual arithmetic operations on these types and the common mathematical functions in the namespace `std` (the `<cmath>` library) have their equivalents in the namespace `boost::multiprecision`. We also discussed how to integrate multiprecision data types with a number of C++ language features and libraries that we discussed in previous chapters.

In most applications it will probably not be necessary to use multiprecision data types unless you are experiencing underflow or overflow problems. The price for the added robustness is a reduction in performance in general.

A1.8 EXERCISES AND PROJECTS

1. (Stiff ODEs and Multiprecision)

We have seen in Chapter 24 that explicit finite schemes (such as the Euler and Runge–Kutta methods) perform badly for ODEs with *several time scales*. The objective

of this exercise is to determine if the use of multiprecision data types can remedy this problem. Answer the following questions:

a) Apply the explicit Euler method to the ODE (24.46) using multiprecision data types. Is this an improvement to using `double`?

b) Now apply Cash–Karp with multiprecision data types to ODE (24.46) and investigate the potential improvements.

c) Depending on the findings from parts a) and b), consider applying Boost *odeint* to the system (24.51).

Having completed this exercise you should be in a position to give an opinion on the usefulness (or not) of multiprecision data types when computing the numerical solutions of stiff ODEs.

2. (Boost *odeint* for Complex State Types)

Differential equations whose coefficients are *complex numbers* occur in a number of applications, for example Fourier transforms and the transformation of PDEs to ODEs.

There are two main ways to solve ODEs with complex-valued coefficients. The first approach is to use Boost *odeint* for complex numbers, as the following simple example shows:

```
// C++ types
using value_type = double;
using state_type = std::complex<value_type>;

namespace Bode = boost::numeric::odeint;

// Global parameters of ODE
std::complex<double> a(2,3);    // 2 + 3i
std::complex<double> A(10,0);   // 10

class OdeExp
{
public:

public:
   OdeExp() { }

   void operator() (const state_type &x, state_type &dxdt,
         const value_type t)
   {
      // C++ standard
      dxdt = -a * x;
   }
};

int main()
{
   namespace Bode = boost::numeric::odeint;

   // Initial condition
   state_type x;
   x = A;
```

```
    // [L,U]
    value_type L = value_type(0.000000);
    value_type U = value_type(1.0);
    value_type dt = value_type(1.0e-15);

    OdeExp ode;
    std::size_t steps = Bode::integrate(ode, x, L, U, dt);

    std::cout.precision(16);
    std::cout << "Approximate answer, steps: " << x << ", "
              << steps << std::endl;
    std::cout << "Exact answer: " << A*std::exp(-a*U)
              << ", " << std::endl;

    return 0;
}
```

The second approach is to recast a complex ODE as a pair of real ODEs. To this end we consider the ODE:

$$\frac{du}{dt} + \lambda u = 0, \quad u(0) = A(\lambda,\ A \in \mathbb{C})$$

$$u : [0, T] \to \mathbb{C}$$
$$u(t) = v(t) + iw(t)$$

$$\lambda = \alpha + i\beta$$
$$A = a + ib, \quad (i = \sqrt{-1}).$$

Then this ODE can be decoupled into a pair of ODEs (just group real and imaginary parts):

$$\frac{dv}{dt} + \alpha v - \beta w = 0$$

$$\frac{dw}{dt} + \alpha w + \beta v = 0$$

$$v(0) = a,\ w(0) = b$$
$$u(t) = v(t) + iw(t).$$

Answer the following questions:

a) Implement the system of equations using Boost `odeint` and compare the solution with the first solution given in this exercise.

b) Generalise both of the solutions in this exercise to one-factor convection–diffusion–reaction PDEs as discussed in Chapters 20, 21 and 25. In this case both the solution and the PDE coefficients can be complex valued.

c) In Section A1.5 we computed Fresnel integrals as the numerical solution of two real ODEs. It is possible to write such integrals as a single integral with complex-valued integrand:

$$\psi(x) = \int_0^x e^{\frac{i\pi}{2}t^2} dt.$$

This formula is found in diffraction applications. We can now differentiate this integral to produce the complex ODE:

$$\frac{d\psi}{dx} = e^{\frac{i\pi}{2}x^2}, \quad x > 0$$
$$\psi(0) = 0.$$

Implement this ODE using the same code as in the above introductory code.

3. (Quadratic Equations)

Consider computing the roots of the quadratic equation:

$$94906265.625x^2 - 189812534x + 94906268.375 = 0.$$

Its roots are:

$$x_1 = 1.00000002897958$$
$$x_2 = 1.000000000000000$$

while using IEEE 754 double-precision arithmetic corresponding to 15 and 17 significant digits of accuracy the computed roots are:

$$x_1 = 1.000000014487979$$
$$x_2 = 1.000000014487979$$

Compute the roots of this quadratic equation using Boost multiprecision data types.

4. (Heat Equation and Separation of Variables)

Consider the heat equation on a bounded domain with Dirichlet boundary conditions:

$$\frac{\partial u}{\partial x} = \frac{\partial^2 u}{\partial x^2}, \quad 0 < x < 1, \ t > 0$$

$$u(0,t) = u(1,t) = 0$$

$$u(x,0) = 2x, 0 \le x \le \frac{1}{2}$$
$$u(x,0) = 2(1-x), \ \frac{1}{2} \le x \le 1.$$

The analytical solution can be found by the method of *Separation of Variables* (Smith, 1978):

$$u(x,t) = \frac{8}{\pi^2} \sum_{n=1}^{\infty} \frac{1}{n^2} \left(\sin \frac{1}{2} n\pi \right) (\sin\ n\pi x) e^{-n^2 \pi^2 t}.$$

Answer the following questions:

a) Implement the above solution using double and multiprecision data types. In particular, determine how many terms need to be taken in the series to achieve a given accuracy.

b) Parallelise the code in part a) by using the parallel reduction and aggregation code in Chapter 30.

c) Approximate the solution of the heat equation using the finite difference methods, starting in Chapter 13.

5. (Confluent Hypergeometric Function)

This function is defined as follows (Abramowitz and Stegun, 1965):

$$M(\alpha,\ \gamma, z) = \sum_{n=0}^{\infty} \frac{\Gamma(\alpha)\,\Gamma(\alpha + n)}{\Gamma(\alpha)\,\Gamma(\gamma + n)} \frac{z^n}{n!}$$

where $\Gamma(z)$ is the gamma function, that is $\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt$.

Answer the following questions:

a) Implement this formula as a sum of terms, each of which is the following mix of C++ and Boost code:

```
template <typename T>
   std::complex<T> CHFTerm(T alpha, T gamma, int n,
                   const std::complex<T>& z)
{
   using boost::math::tgamma;
   using boost::math::factorial;

   T nt = static_cast<T>(n);
   T coeff = (tgamma(gamma) * tgamma(alpha + nt))
            / (tgamma(alpha) * tgamma(gamma + nt));

   // Write z^n in polar form
   T x = z.real(); T y = z.imag();
   T r = boost::multiprecision::sqrt(x*x + y*y);
   T rn   = boost::multiprecision::pow(static_cast<T>(r),
             static_cast<T>(n));
   std::complex<T> inTheta(static_cast<T>(0.0),
             static_cast<T>(n)*std::arg(z));
   std::complex<T> zn = rn*std::exp<T>(inTheta);
   T fac = 1.0 / factorial<T>(n);

   return coeff * zn * fac;
}
```

b) Test the code using the following cases:

$$M(0.3, 0.2, -0.1) = 0.8578490$$
$$M(-1.3,\ 1.2,\ .1) = 0.8924108$$
$$M(17, 16, 1) = 2.8881744$$
$$M'(-0.7, -0.6,\ 0.5) = 1.724128 \text{ where } M'(a, b, z) = \frac{a}{b}M(a + 1.b + 1, z)$$
$$M(a, b, z) = e^z M(b - a,\ b, -z).$$

c) Now implement the algorithm using multiprecision data types. Do you get better results than in the 32-bit and 64-bit cases?

d) Parallelise the code.

We remark that the confluent hypergeometric function can be used to compute an analytic approximation to the partial differential equations that describe certain classes of options. See, for example, Lewis (2016) and Linetsky (2004) where it is used to compute hitting times for the CIR diffusion process.

6. (Factorials and Gamma Function)

We have seen in Chapters 11 and 12 how to compute factorials:

$$n! = n\,(n-1)\ldots 3.2.1. \tag{A1.1}$$

For larger values of the integer arguments we employ the gamma function:

$$\Gamma(Z) = \int_0^\infty x^{z-1}e^{-x}dx,\ z \in \mathbb{C}$$
$$\Gamma(n) = (n-1)! \text{ if } n \text{ is a positive integer.} \tag{A1.2}$$

The objective of this exercise is to compare the relative efficiency, robustness and accuracy of these two algorithms to compute factorials.

Answer the following questions:

a) Implement the algorithms for a range of integral values, for example $N = 10^a$ where a is an integer. For which values of a do we get overflow for both algorithms?

b) Add code from the type traits library to test for infinite numbers, NaN and otherwise possible overflow events.

c) Use the Boost multiprecision library to create the equivalents of the algorithms in parts a) and b) using extended data types, for example:

```
boost::multiprecision::cpp_dec_float_100 input(10'000'001);
std::cout << boost::math::tgamma(input) << '\n';
```

d) Test the relative run-time performance of all the algorithms in parts a) to c).

Financial Instrument Pricing Using C++, 2e
By Daniel J. Duffy

APPENDIX 2

Computing Implied Volatility

> … the source of all great mathematics is the special case, the concrete example. It is frequent in mathematics that every instance of a concept of generality is, in essence, the same as a small and concrete special case.
>
> —Paul Halmos

A2.1 INTRODUCTION AND OBJECTIVES

This appendix serves a number of purposes. First, it elaborates on the topics in Chapter 19 by focusing on a specific problem (computing implied volatility) using the various methods discussed in that chapter. Second, it uses some of the concepts, models and methods that we discussed in various chapters of this book, for example:

- Computing implied volatility.
- Numerical methods for root finding and univariate optimisation.
- Homotopy methods and the solution of nonlinear equations.
- Creating a software framework for function optimisation based on the design principles that we introduced in Chapter 9 (see also Figure 19.1).
- Doing mathematics in C++; higher-order functions. This is a small research topic.
- Unconstrained multivariate optimisation.

Finally, we shall discuss a number of new topics (such as higher-order mathematical functions, Differential Evolution and multivariate optimisation) in more detail in a later work.

A2.2 IMPLIED VOLATILITY BY LEAST-SQUARES OPTIMISATION

In the Black–Scholes model, the theoretical value of a vanilla option is a monotonic increasing function of the volatility of the underlying asset. This means that it is usually possible to compute a unique implied volatility from a given market price for an option. This implied volatility

is best regarded as a rescaling of option prices which makes comparisons between different strikes, expirations and underlyings easier and more intuitive. Based on these observations we formulate the following *nonlinear least-squares problem* that we write as:

$$\begin{aligned} &f = \text{market price} - \text{option price}\ (\sigma) \\ &\text{optimise}\ z = \min_{\sigma} f^2(\sigma). \end{aligned} \tag{A2.1}$$

More generally, in n dimensions, the nonlinear system of equations:

$$f(x) = 0,\ x \in \mathbb{R}^n \tag{A2.2}$$

can be written as a *minimisation problem*:

$$\min_{x \in \mathbb{R}^n} F(x) \tag{A2.3}$$

where:

$$F(x) = \sum_{j=1}^{n} f_j^2(x) = f^{\top}(x) f(x).$$

Another way to view implied volatility is to think of it as a price, not as a measure of future stock moves. We view it simply as a more convenient way than currency to communicate option prices (Natenberg, 2007).

We now discuss how to compute implied volatility using the Golden Search, three-point interval search and Brent methods (we use the Brent method from Boost). It is our intention to describe each of the participating classes in the context diagram in Figure 19.1. The code for each component is placed in its own file in the interests of maintainability (we have not created builders in this version):

```
#include "Interfaces.hpp"
#include "Termination.hpp"
#include "Solvers.hpp"
#include "MIS.hpp"
#include "Reporting.hpp"
#include "Mediator.hpp"
#include "Pricers.hpp"
```

The code for some of these classes has been given in Section 19.7 and is not repeated here. The current problem is to compute implied volatility from call (or put) prices. We encapsulate these requirements in a class that implements equations (A2.1) to (A2.3) above:

```
// Standardised interface
using IPricingFunction = std::function<double(double sig)>;

class ImpliedVolatilityEstimator
{ // Computes iv from call or put prices; we are using least squares
  // such Golden mean, Brent, 3-point interval, Differential Evolution
```

```
private:
   double mp;                    // Market price
   IPricingFunction pricer;      // Call or put price
public:
   ImpliedVolatilityEstimator
      (double marketPrice, const IPricingFunction& optionPricer)
      : mp(marketPrice), pricer(optionPricer) {}

   double operator () (double sig) const
   { // Compute current residual


      double calculatedValue = pricer(sig);
      double d = -mp + calculatedValue;

      // Least-squares form
      return d*d;
   }
};
```

In general, we use this class with well-known pricers:

```
class Pricer
{ // Special adapter class to work with implied volatility
private:
   OptionData opt;
public:
   Pricer(const OptionData& optionData) : opt(optionData) {}

   double PutPrice(double sig) const
   {
      double tmp = sig * std::sqrt(opt.T);

      double d1 = (std::log(opt.S / opt.K)
                  + (opt.b + (sig*sig)*0.5) * opt.T) / tmp;
      double d2 = d1 - tmp;

      return opt.K * std::exp(-opt.r * opt.T)* N(-d2)
             - opt.S* std::exp((opt.b - opt.r)*opt.T) * N(-d1);
   }


   double CallPrice(double sig) const
   {
      double tmp = sig * std::sqrt(opt.T);

      double d1 = (std::log(opt.S / opt.K)
                  + (opt.b + (sig*sig)*0.5) * opt.T) / tmp;
      double d2 = d1 - tmp;
```

```
        return (opt.S * std::exp((opt.b - opt.r)*opt.T) * N(d1))
              - (opt.K * std::exp(-opt.r * opt.T)* N(d2));
    }


    double Vega(double sig) const
    {
        double tmp = sig * std::sqrt(opt.T);
        double d1 = (log(opt.S / opt.K)
           + (opt.b + (sig*sig)*0.5) * opt.T) / tmp;

        return opt.S * std::exp((opt.b - opt.r)*opt.T) * n(d1)
              * sqrt(opt.T);
    }

    double DVegaDVol(double sig) const
    { // Needed in Halley's and Schroeder's methods

        double tmp = sig * std::sqrt(opt.T);
        double d1 = (log(opt.S / opt.K)
                    + (opt.b + (sig*sig)*0.5) * opt.T) / tmp;
        double d2 = d1 - tmp;

        return (Vega(sig)*d1*d2) / sig;
    }

};
```

The next step is to configure this class as well as the other classes in Figure 19.1. We take some specific examples:

```
OptionData opt;
opt.K = 65.0;
opt.T = 0.25;
opt.r = 0.08;
opt.b = opt.r;
opt.sig = 0.3;
opt.S = 60.0;

// sig = 0.3
value_type putMarketPrice = 5.8462'8220'986;
value_type callMarketPrice = 2.1333'6844'496;

// We create 3 optimisers (3 point, Golden Section, Brent)
auto pricer1
   = std::bind(&Pricer::PutPrice, option, std::placeholders::_1);
ImpliedVolatilityEstimator ivp1(putMarketPrice, pricer1);

auto pricer2
   = [&option](value_type sig) { return option.CallPrice(sig); };
```

```
ImpliedVolatilityEstimator ivp2(callMarketPrice, pricer2);

//   auto callPricer = std::bind(PutPrice, std::placeholders::_1);

// Bounds for implied volatility
value_type sigL = 0.0023; value_type sigH = 0.9;
value_type fL = 0.0;
value_type fH = 0.0;

TerminatorI<value_type> stopping(1.0e-12);
ReportingDefault<value_type> report;
MISDefault<int> mis;

ThreePointIntervalSolver<value_type> solver1(ivp1, sigL, sigH);
GoldenSectionIntervalSolver<value_type> solver2(ivp2, sigL, sigH);
BrentIntervalSolver<value_type> solver3(ivp2, sigL, sigH);
```

Finally, we create and run the mediators:

```
Mediator<value_type> med1(ivp1, sigL, sigH, stopping,
                          solver1, mis, report);
med1.run();

Mediator<value_type> med2(ivp2, sigL, sigH, stopping,
                          solver2, mis, report);
med2.run();

Mediator<value_type> med3(ivp2, sigL, sigH, stopping,
                          solver3, mis, report);
med3.run();
```

It is also possible to create an instantiation of the classes in Figure 19.1 by using lightweight lambda functions (we do not need to create classes as with the traditional object-oriented style):

```
ITerminationType<value_type> brentTerm
   = [](value_type a, value_type b) { return true; };

IMISType<value_type> mis4
   = [](value_type a, value_type b, value_type fa,value_type fb) {};

auto report4
      = [](value_type a, value_type b, value_type fa, value_type fb)
           {std::cout<<std::setprecision(16)<<"Brent" << (a + b)/2; };

GoldenSectionIntervalSolver<value_type> brentSolver(ivp1, sigL, sigH);

Mediator<value_type> med4
       (ivp1, sigL, sigH, stopping, brentSolver, mis4, report4);
med4.run();
```

The output from running these four mediators is:

```
** Final report **
Bracket + avg Value: [0.3,0.3] 0.3
Averaged function value (~ 0?): 2.1486703e-23

** Final report **
Bracket + avg Value: [0.3,0.3] 0.3
Averaged function value (~ 0?): 1.5536833e-23

** Final report **
Bracket + avg Value: [0.3,0.3] 0.3
Averaged function value (~ 0?): 1.6535216e-19

Brent 0.3000000000362369
```

For completeness, we have created an *adapter* class (GOF, 1995) for the Brent algorithm in Boost so that it can be integrated into our software framework:

```
template <typename T>
   class BrentIntervalSolver
{
private:
   IFunctionType<T> f;     // Function to be minimised
   T a, b;
   boost::uintmax_t maxIter;
public:
   BrentIntervalSolver (const IFunctionType<T>& function, T A, T B,
                        int maxIerations = 1000)
      : f(function), a(A), b(B), maxIter(maxIerations){}

      void operator () (T& a, T& b, T& fa, T& fb)
      {
         int bits = std::numeric_limits<double>::digits;

         std::pair<T,T> result = boost::math::tools::brent_find_minima
            (f, a, b, bits, maxIter);

         a = b = result.first;
         fa = fb = result.second;
      }

      T value() const
      {
         return 0.5*(a + b);
      }

      int NumberIterations() const
      {
         return maxIter;
      }
   };
```

A2.3 IMPLIED VOLATILITY BY FIXED-POINT ITERATION

We have discussed fixed-point iteration in Sections 19.5, 19.6 and in the appendix to that chapter (Section 19.12). The original problem must be posed as a fixed-point problem as follows:

$$f(x) = 0 \Leftrightarrow x = g(x) \tag{A2.4}$$

where:

$$g(x) = x - \frac{f(x)}{m}$$

and m is chosen in order to make $g(x)$ a contraction. In general, the function $g(x)$ will be a contraction if:

$$g'(x) = \frac{f'(x)}{m} = \frac{\partial C}{\partial \sigma} / m < 1 \text{ or } m > \frac{\partial C}{\partial \sigma}. \tag{A2.5}$$

The C++ code is:

```
// Fixed point iteration
value_type callMarketPrice = 2.1333'6844'496;
auto callPricer = [&](value_type x, value_type mktValue)
{
   return -mktValue + option.CallPrice(x);
};


auto callPricerFP = [&](value_type x)
{
   value_type tmp = opt.sig * std::sqrt(opt.T);
   value_type d1 = (log(opt.S / opt.K)
      + (opt.b + (opt.sig*opt.sig)*0.5) * opt.T) / tmp;

   value_type m = opt.S* sqrt(opt.T);

   value_type x0 = opt.sig;

   return x - (callPricer(x, callMarketPrice) / m);

};
```

The Banach fixed-point theorem is stated as Theorem 1 in Section 19.12. We have implemented the algorithm as follows:

```
template <typename T>
   class BanachFixedPointSolver
{ // 1st order fixed point iteration
```

```
private:
   IFunctionType<T> f;    // Function to be minimised
   T x0;                  // Initial guess
   ITerminationType<T> tol;
   T a, b;
   int incr;
public:
   BanachFixedPointSolver
      (const IFunctionType<T>& function, T A, T B, T initialGuess,
       const ITerminationType<T>& termination)
      : f(function), a(A), b(B), x0(initialGuess), tol(termination), incr(0)
   {

   }

   void operator () ()
   {

      double x1, x2;

      do
      {
         x1 = f(x0);
         x2 = f(x1);

         x0 = x2;
         ++incr;

      } while (!tol(x2, x1));

      a = x1;
      b = x2;

   }

   T value() const
   {
      return 0.5*(a + b);
   }

   int NumberIterations() const
   {
      return incr;
   }
};
```

and the class implementing the Aitken algorithm is given by:

```
template <typename T>
   class AitkenFixedPointSolver
```

```
{
private:
   IFunctionType<T> f;     // Function to be minimised
   T x0;                   // Initial guess
   ITerminationType<T> tol;
   T a, b;
   int incr;
public:
   AitkenFixedPointSolver (const IFunctionType<T>& function, T A, T B,
                           T initialGuess, const ITerminationType<T>&
                           termination)
      : f(function), a(A), b(B), x0(initialGuess), tol(termination), incr(0)
   {}

   //void operator () (T& a, T& b, T& fa, T& fb)
   void operator () ()
   {

      double x1, x2, ait;


      // Kick-off the first 3 values.
      do
      {
         x1 = f(x0);
         x2 = f(x1);

         double den = (x2 - x1) - (x1 - x0);

         double num = x2 - x1;
         ait = x2 - (num*num) / den;

         x0 = ait;
         ++incr;

      } while (!tol(x2, ait));

      a = b = ait;

   }

   T value() const
   {
      return 0.5*(a + b);
   }

   int NumberIterations() const
   {
      return incr;
   }
};
```

A test case is given by:

```
value_type guess = 0.0002;
BanachFixedPointSolver<value_type> solver4
      (callPricerFP, sigL, sigH, guess, stopping);
solver4();
std::cout << "\nBanach FPT + #iterations: " << solver4.value()
          << ", " << solver4.NumberIterations() << '\n';

AitkenFixedPointSolver<value_type> solver5
       (callPricerFP, sigL, sigH, guess, stopping);
solver5();
std::cout << "\nAitken fpt + #iterations: " << solver5.value()
          << ", " << solver5.NumberIterations() << '\n';
```

The output is:

```
Banach FPT + #iterations: 0.3000000000027625, 25

Aitken fpt + #iterations: 0.3000000000038584, 4
```

We discuss fixed-point algorithms for a number of reasons. First, they can be used as tools to prove the existence (and uniqueness) of solutions to a wide range of linear and nonlinear equations (Smart, 1974). Second, they can easily be described in algorithmic form and then coded in C++. Finally, they can be used instead of other derivative-based solvers such as the Newton–Raphson method which demands that the initial guess be close to the exact solution and for which we must compute derivatives, either analytically or numerically. We do not have these issues with fixed-point algorithms in general.

It is possible to combine the above two fixed-point algorithms; in the first phase we use the Banach algorithm to *bracket* the solution to a given accuracy (say 1/10) and then use this computed value as the initial guess for the Aitken method. Equation (19.25) gives an indication of how many iterations to take in the Banach algorithm before switching to Aitken. We thus achieve algorithmic robustness and efficiency.

A2.4 VOLATILITY BY HOMOTOPY AND ODE SOLVERS

We have already defined the concept of homotopy in Section 6.6.2. We now show how to solve a nonlinear equation by embedding it in a homotopy mapping and then transforming this mapping to an ODE. To this end, consider again the nonlinear equation:

$$f(x) = 0 \tag{A2.6}$$

that we embed as:

$$H(x,t) = f(x) - (1-t)f(x_0), \quad 0 \le t \le 1. \tag{A2.7}$$

(Here, x_0 is an arbitrary guess.)

Using rules for differentiation:

$$\begin{aligned}\frac{\partial H}{\partial x} &= \frac{\partial f}{\partial x} = f'(x)\\ \frac{\partial H}{\partial t} &= f(x_0)\end{aligned} \tag{A2.8}$$

allows us to write the following ODE:

$$\begin{aligned}\frac{dx}{dt} &= \frac{\partial x}{\partial H}\frac{\partial H}{\partial t} = f(x_0)/f'(x)\\ x(0) &= 0.\end{aligned} \tag{A2.9}$$

We solve this ODE numerically for $x(t)$ up to $t = 1$ and hence the solution of the original problem (A2.6) will be given precisely by $x(1)$. Transforming problem (A2.6) to ODE system (A2.9) is often called *Davidenko's* method.

We solve system (A2.9) using the Boost *odeint* library that we discussed in Chapters 24 and 25. First, the C++ class and the needed functions are:

```
// Call parameters, needs to generalised
const double S = 60.0;
const double K = 65.0;
const double r = 0.08;
const double b = r;
const double sig = 0.3;
const double T = 0.25;

// Hard-coded here
value_type marketValue = 2.1333'6844'496;

value_type Function (state_type x)
{ // Call price, x == sig

   value_type tmp = x[0] * std::sqrt(T);

   value_type d1 = (std::log(S / K) + (b + (x[0]*x[0])*0.5) * T)
            / tmp;
   value_type d2 = d1 - tmp;
   return (S * std::exp((b - r)*T)* N(d1))
      - (K * std::exp(-r * T)* N(d2)) -marketValue;
}

value_type FunctionDeriv (state_type x)
{ // x == sig

   // Vega = dC/dsig

   value_type tmp = x[0] * std::sqrt(T);
   value_type d1 = (std::log(S / K) + (b + (x[0]*x[0])*0.5) * T)
            / tmp;
```

```
   return S * std::exp((b-r)*T)*n(d1) * sqrt(T);
}


// The rhs of x' = f(x) defined as a class
class Davidenko
{
private:
   state_type x0;

public:

   Davidenko(state_type guess) : x0(guess){}

   void operator() (const state_type& x, state_type& dxdt,
                    const value_type t)
   {
      dxdt[0] = -Function(x0) / FunctionDeriv(x);
   }
};
```

A test program is:

```
typedef double value_type;
typedef std::vector<value_type> state_type;

int main()
{
   namespace Bode = boost::numeric::odeint;

   // Initial condition
   state_type x(1, 0.0); // vector
   x[0] = 0.113;

   Davidenko ode(x);

   value_type L = 0.0;
   value_type U = 1.0; // upper limit
   value_type dt = 1.0e-9;

   std::size_t steps = Bode::integrate(ode, x, L, U, dt);
   std::cout << "Value from 101 integrator: "
             << std::setprecision(8)   << x[0] << std::endl;

   Bode::runge_kutta_dopri5<state_type, value_type> myStepper;

   x[0] = 0.113;
   dt = 1.0e-1;
   int n = static_cast<int>((U - L)/dt);
```

```
    steps = Bode::integrate_n_steps(myStepper, ode, x, L, dt, n);
    std::cout << "Value from n steps: " << x[0] << std::endl;

    return 0;
}
```

The output is:

```
Value from 101 integrator: 0.30000007
Value from n steps: 0.29999998
```

The homotopy approach can be generalised to systems of equations, but a discussion is outside the scope of this book. The above code can be adopted to your own applications.

A2.5 A VECTOR SPACE OF FUNCTIONS IN C++

The functional programming model is supported in C++ to a certain extent and we have seen some of its applications in previous chapters, for example:

- Chapter 3: lambda functions, universal function wrappers and `std::bind`.
- Section 6.8: using universal function wrappers to create flexible SDE classes.
- Sections 22.6 and 22.7: multiparadigm design patterns.

The book is peppered with examples and applications of this programming model. In this section we implement the mathematical concept of a vector space that we introduced in Section 6.4.1. In the current case the elements of the vector space are functions that map a *domain to a range* (or *co-domain*). We reduce the scope by focusing on functions that map scalars to scalars:

```
template <typename T>
    using FType = std::function<T (T)>;
```

We create a minimalist set of C++ functions to mimic the corresponding mathematical properties. We now introduce the main topic.

A2.5.1 Function Algebra and Initial Examples

We discuss how we implemented the higher-order C++ functions that describe the corresponding mathematical functions. The crucial point to note is that the functions return a function and not a scalar value! This *extra level of indirection* is achieved by embedded lambda functions, as the following prototypical example to add two functions shows:

```
template <typename T>
    FType<T> operator + (const FType<T>& f, const FType<T>& g)
{ // Addition

    return [=](T x)
    {
        return f(x) + g(x);
    };
}
```

Some examples of use are:

```
double G(double x)
{
   return x+1;
}

double H(double x, double y)
{
   return x + y;
}

FType<double> e = [](double x) { return 5.0; };
FType<double> f = [](double x) { return G(x); };
FType<double> g = [](double x) { return x*x; };

FType<double> bindFun = std::bind(H, 1.0, std::placeholders::_1);
auto func1 = g + bindFun;
std::cout << "13? " << func1(3.0) << "\n";

auto h1 = f + g;
FType<double> h2 = f + exp(e);
FType<double> h3 = g + f;
```

We see that this manner of coding is very close to the way we do mathematics. In the same vein as above we define the following set of functions:

```
template <typename T>
   FType<T> operator + (const FType<T>& f, T a)
{ // Scalar addition

   return [=](T x)
   {
      return f(x) + a;
   };
}

template <typename T>
   FType<T> operator * (T a, const FType<T>& f)
{ // Scalar multiplication

   return [=](T x)
      {
      return a*f(x);
   };
}


template <typename T>
   FType<T> operator - (const FType<T>& f, FType<T>& g)
```

```
{ // Subtraction

   return [=](T x)
   {
      return f(x) - g(x);
   };
}

template <typename T>
   FType<T> operator - (const FType<T>& f)
{ // Unary negation

   return [=](T x)
   {
      return -f(x);
   };
}

template <typename T>
   FType<T> operator + (const FType<T>& f)
{ // Unary plus

   return [=](T x)
   {
      return +f(x);
   };
}

template <typename T>
   FType<T> operator << (FType<T>& f, FType<T>& g)
{ // Composition of functions

   return [=](T x)
   {
      return f(g(x));
   };
}

template <typename T>
   FType<T> assign(T constant)
{ // Create a function from a scalar constant

   return [=](T x)
   {
      return constant;
   };
}

template <typename T>
   FType<T> exp(const FType<T>& f)
{ // exp
```

```
    return [=](T x)
    {
        return std::exp(f(x));
    };
}


template <typename T>
    FType<T> log(const FType<T>& f)
{ // log

    return [=](T x)
    {
        return std::log(f(x));
    };
}

template <typename T>
    FType<T> pow(const FType<T>& f, int n)
{ // powers of a function f(x)*...*f(x) (n times)

    return [=](T x)
    {
        return std::pow(f(x), n);
    };
}
```

This list is by no means exhaustive (nor is it meant to be) but it is easy to define your own functions and variations including algebra for vector functions, vector-valued functions and specialisations of the underlying data types such as complex numbers, for example.

We now give more examples of using this new functionality. The code compiles in all cases and is easy to understand. The first example computes the fourth power of a function:

```
FType<double> f = [](double x) { return x*x; };
int n = 4;
auto prodFun = pow(f, n);

double x = 2.0;
std::cout << "Power: " << prodFun(x) << "\n";
```

Strictly speaking, we should place these functions in a namespace to avoid confusing them with standard C++ functions such as `std::pow`, for example (we leave this as an exercise). The second example is a kind of *sanity check* to see how the code with unary minus and unary plus gives the expected result (the code compiles with flying colours):

```
{ // Unary minus and plus

    auto f = assign(5.0);
    auto f1 = exp(f);
    auto f2 = +exp(f);
```

```
    auto f3 = -exp(f);
    auto f4 = -(-exp(f));
    auto f5 = +(-exp(f));
    auto f6 = -(+exp(f));

    double x = 5.0;
    std::cout << "exp stuff: " << f1(x) << ", " << f2(x) << ", "
              << f3(x) << ", " << f4(x) << ", " << f5(x)
              << ", " << f6(x) << '\n'; // {+, -} 148.413
}
```

We can concoct many more cases but they will all follow the same pattern as above. We now move on to create complex functions from simpler ones.

A2.5.2 Assembling Functions, Piece by Piece

Since we have created an algebra of functions we are now in a position to construct a complex function from a library of simpler functions. As a first example we examine the n-dimensional *Ackley function*, which is one multimodal test function for optimisation algorithms (Price, Storn and Lampinen, 1996):

$$f(x) = -20\exp\left(-0.2\sqrt{\frac{1}{n}\sum_{j=0}^{n-1} x_j^2}\right) - \exp\left(\frac{1}{n}\sum_{j=0}^{n-1}\cos(2\pi x_j)\right) + 20 + e \qquad \text{(A2.10)}$$
$$-30 \le x_j \le 30, \quad j = 0, \ldots, n-1.$$

We reduce the scope by taking the one-dimensional case and implementing it as a free function:

```
double AckleyGlobalFunction(double x)
{
   double a = 20.0; double b = 0.2; double c = 2.0*3.14159;

   return -a*std::exp(-b*x) - std::exp(std::cos(c*x))
          + a + std::exp(1.0);
}
```

We can also choose the function in equation (A2.10) as the sum of three functions, just to show how our function algebra works:

```
double AckleyGlobalFunctionII(double x)
{ // Building the functions in a HOF assembly process

   double a = 20.0; double b = 0.2; double c = 2.0*3.14159;

   // Bit unwieldy; used to show function vector spaces.
   FType<double> f11 = [&](double x) {return -b*x; };
   FType<double> f1 = -a*exp(f11);
```

```
    FType<double> f22 = [&](double x) {return std::cos(c*x); };
    FType<double> f2 = -exp(f22);
    FType<double> f3 = assign(a + std::exp(1.0));
    FType<double> Ackley = f1 + f2 + f3;

    return Ackley(x);
}
```

We can test if these functions give the same output by running a simple test program:

```
double x = -1.0;
while (x < 1.0)
{
    std::cout << AckleyGlobalFunction(x)-AckleyGlobalFunctionII(x);
    x += 0.1;
}
```

The output should be (almost) zero in all cases.

As a final example we take the function:

$$x^2 + x^4 + x^8, \quad -10 \le x \le 10. \tag{A2.11}$$

A possible implementation is:

```
double TestFunction(double x)
{ // To show how function algebra works


    // Bit unwieldy; used to show function vector spaces.
    FType<double> f1 = [](double x) {return x*x; };
    FType<double> f2 = [&](double x) {return f1(x)*f1(x); };
    FType<double> f3 = [&](double x) {return f2(x)*f2(x); };

    FType<double> sumFunc = f1 + f2 + f3;

    return sumFunc(x);
}
```

We can check this code with a straightforward implementation using the power function as follows:

```
double x = 2.710;
std::cout << "Test vs exact: " << TestFunction(x) << ", "
          << std::pow(x, 2.0) + std::pow(x, 4.0) + std::pow(x, 8.0);
```

The output is:

```
Test vs exact: 2970.35, 2970.35
```

A2.5.3 Higher-Order Functions and Optimisation

We conclude with a discussion of using *Brent's method* with functions that have been created as above. We define a convenience function:

```
template <typename T>
   void BrentMinimum(const FType<T>& fun, T minV, T maxV,
                     const std::string& s)
{
   std::cout << "Function " << s << ", ";
   int bits = 50;
   boost::uintmax_t maxIter = 10000;

   std::pair<double, double> result
      = boost::math::tools::brent_find_minima
            (fun, minV, maxV, bits, maxIter);
   std::cout << "x and y values: " << result.first
             << ", " << result.second << std::endl;
}
```

We apply this optimiser to equations (A2.10) and (A2.11), the Ackley function, the sphere function as well as the one-dimensional case of the *Rastrigin function*:

$$f(x) = \sum_{j=0}^{n-1} \left(x_j^2 - 10\cos(2\pi x_j) + 10 \right) \qquad -5.12 \le x_j \le 5.12, \quad j = 0, \ldots, n-1. \tag{A2.12}$$

The code is:

```
// Define interval [min, max] where search will take place
double minV = -30.0; double maxV = -minV;
BrentMinimum<double>(AckleyGlobalFunctionII, minV, maxV, "Ackley");

minV = -5.12; maxV = -minV;
BrentMinimum<double>(RastriginGlobalFunction, minV, maxV, "Rastrigin");

minV = -1000.0; maxV = -minV;
BrentMinimum<double>(SphereGlobalFunction, minV, maxV, "Sphere");

minV = -10.0; maxV = -minV;
BrentMinimum<double>(TestFunction, minV, maxV, "Test");
```

where we have used the functions:

```
double RastriginGlobalFunction(double x)
{
   double A = 10.0;

   return A + x*x - A*std::cos(2.0*3.14159);
}
```

```
double SphereGlobalFunction(double x)
{
   return x*x;
}
```

The output is:

```
Function Ackley, x and y values: -30, -8048.57
Function Rastrigin, x and y values: 2.87297e-08, 1.40831e-10
Function Sphere, x and y values: 2.84217e-14, 8.07794e-28
Function Test, x and y values: 8.09975e-21, 6.5606e-41
```

In general, many benchmark multivariate optimisation examples in the literature have been designed to produce the easily checked results:

$$f(x^*) = 0, \quad \text{where } x^* = \left(x_0^*, \ldots, x_{n-1}^*\right), \quad x_j^* = 0, \quad j = 0, \ldots, n-1. \tag{A2.13}$$

A2.6 MULTIVARIATE OPTIMISATION

We conclude this appendix with a short overview of *multivariate optimisation*, which is a generalisation of the one-dimensional methods to n independent variables. We focus on the *steepest ascent (descent) method*, set up a basic software framework and give some examples for two-factor problems.

A2.7 NONLINEAR PROGRAMMING AND MULTIVARIABLE OPTIMISATION

In this section we introduce the mathematical formulation of unconstrained multivariable optimisation problems (minimise or maximise a given objective function).

Consider the scalar function $F : \mathbb{R}^n \to \mathbb{R}^1, \quad n \geq 1$ that maps n-dimensional Euclidean space into the real line. In general, we wish to find a point x in Euclidean space that *minimises or maximises* this function. The general *optimisation problem* is to find a real number z such that:

$$optimise \quad z = F(x), \quad x = (x_1, \ldots, x_n)^\top. \tag{A2.14}$$

Here F is called the *objective function* and the problem is *unconstrained* because there are no restrictions placed on x. We note that maximisation of $F(x)$ is the same as the minimisation of its additive inverse $-F(x)$. In the main, we are usually interested in minimisation problems.

We now introduce some terms. The *gradient function* ∇F associated with $F = F(x_1, \ldots, x_n)$ is a vector defined by:

$$\nabla F = \left(\frac{\partial F}{\partial x_1}, \ldots, \frac{\partial F}{\partial x_n}\right)^\top. \tag{A2.15}$$

The *Hessian matrix* H_F associated with F is defined as:

$$H_F = \left(\frac{\partial^2 F}{\partial x_i \partial x_j} \right), \quad i,j = 1, \ldots, n. \tag{A2.16}$$

We take a simple example and to this end we consider the case $n = 2$. Let $F(x) = x_1^2 + x_2^2$. Then:

$$\nabla F(x) = (2x_1,\ 2x_2)^\top$$
$$H_F = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}.$$

We discuss two methods to solve problem (A2.14). First, the *method of steepest descent* defines an *iterative scheme* to compute a sequence of values:

$$x_{k+1} = x_k + \lambda_k^* \nabla F(x_k), \quad k = 0, 1, 2, \ldots \tag{A2.17}$$

where x_0 is an initial vector and λ_k^* is a positive scalar that minimises $F(x_k + \lambda \nabla F(x_k))$. This single-variable problem can be solved using *Brent's method* or a sequential search technique, for example, and in this case finding a local minimum is acceptable. The iterative process in equation (A2.17) terminates if and when the difference (in some norm) between the values of the solution at two successive iterates is smaller than a prescribed tolerance. The last computed x-vector is taken as the final approximation to the solution x^* of problem (A2.14).

Another approximation is the *Newton–Raphson method*. Again, we choose an initial vector x_0. We then generate a sequence of vectors by the following iterative scheme:

$$x_{k+1} = x_k + H_F(x_k)^{-1} \nabla F(x_k), \quad k = 0, 1, 2, \ldots \tag{A2.18}$$

where H_F is defined in equation (A2.16).

The stopping criterion is the same as in scheme (A2.17). The scheme (A2.18) converges to a local minimum if the Hessian matrix is positive definite. If the initial vector is not chosen correctly then scheme (A2.18) may not converge to a local minimum or it may not even converge at all.

We wish to extend problem (A2.14) by discussing multivariable optimisation problems with constraints. There are several kinds of constraints than can be used, for example:

- *Equality constraints*: $g(x) = 0.$ (A2.19)
- *Inequality constraints*: $g(x) \geq 0.$ (A2.20)
- *Linear constraints*: $\left(g(x) = Ax - b = 0(g_1(x), \ldots, g_m(x)^\top\right)$ (A2.21)

where A is a constant $n \times m$ matrix, b is an m-dimensional vector and the symbol $\top$ denotes the transpose of a vector or of a matrix. The function g is given.

We now consider problem (A2.14) subject to the constraints (A2.19) and we assume that $m < n$ (that is, problem (A2.14) has fewer constraints than variables). In order to solve this problem, we form the *Lagrangian function*:

$$L(x_1, \dots, x_n, \lambda_1, \dots, \lambda_m) \equiv F(x) - \sum_{j=1}^{m} \lambda_j g_j(x) \tag{A2.22}$$

where $\lambda_j (j = 1, \dots, m)$ are unknown constants called *Lagrange multipliers*. We then solve the system of equations:

$$\frac{\partial L}{\partial x_j} = 0, \quad j = 1, \dots, n \tag{A2.23}$$

and

$$\frac{\partial L}{\partial \lambda_j} = 0, \quad j = 1, \dots, m. \tag{A2.24}$$

In general, the Lagrange multiplier method is equivalent to using the constraint equations to eliminate some of the x-variables from the objective function and then solving an unconstrained minimisation problem in the remaining variables.

It is possible to solve system (A2.22) using the Newton–Raphson method. To this end, we define the extended vector $z = (x_1, \dots, x_n, \lambda_1, \dots, \lambda_m)$ and then we define the scheme:

$$z_{k+1} = z_k + H_L(z_k)^{-1} \nabla L(z_k), \quad k = 0, 1, 2, \dots \tag{A2.25}$$

The problem with this scheme is that it is difficult to find an initial vector and if it is incorrect then the Newton–Raphson method will not converge to the correct value or the method may even diverge. Other methods for solving system (A2.22) are discussed in Scales (1985) and Bronson and Naadimuthu (1997).

A2.8 NONLINEAR LEAST SQUARES

We now discuss a special class of objective function (A2.14), namely when the function $F(x)$ is the sum of squares of other nonlinear functions:

$$F(x) = \sum_{j=1}^{m} f_j^2(x) \tag{A2.26}$$

where:

$$x \in \mathbb{R}^n \text{ and } f(x) = \big(f_1(x), \dots, f_m(x)\big)^\top.$$

The minimisation of functions of this kind is called *nonlinear least squares*. We see that the objective function can never take negative values for these kinds of problems. We can write equation (A2.14) in the equivalent form:

$$F(x) = f^{\top}(x)f(x) \tag{A2.27}$$

when $f^{\top}$ is the transpose of f.

A2.8.1 Basic Algorithm and C++ Code

We now show how we implemented the method of steepest descent as described by equation (A2.17). We mention that steepest descent finds a function minimum while steepest ascent computes a function maximum (you can turn the latter problem into the former problem by simply changing the sign of the objective function).

We describe what we need to do in order to implement the algorithm for equation (A2.17). First, we define uniform function wrappers for functions that map a vector to a scalar (*real-valued functions*) and functions that map vectors into vectors (*vector-valued functions*):

```
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/io.hpp>

namespace u = boost::numeric::ublas;

// Function categories used in optimisation
using FunctionType
   = std::function<double (const u::vector<double>&)>;
using GradientType
   = std::function<u::vector<double> (const u::vector<double>&)>;
```

The next step is to design a C++ class to implement the steepest descent method. The members and constructor of an *initial version* are:

```
class SteeepestDescent
{ // Classic steepest descent method

private:

   FunctionType func;              // Function + gradient

   GradientType grad;

   u::vector<double> Xk;           // Value at iteration k
   u::vector<double> X0;           // Initial seed
   u::vector<double> XkP1;         // Value at iteration k+1
   u::vector<double> gXk;          // Gradient at iteration k

   double minV, maxV;              // Bracket for Brent method
   double TOL;                     // Tolerance
```

```
   // Statistics
   double error;                    // Final error
   std::vector<double> errors;      // Errors at each iteration

public:
   SteeepestDescent(FunctionType function, GradientType gradient,
                    const u::vector<double>& x_0,         // Initial guess
                    double tolerance,                     // Desired
                                                          // accuracy
                    double minimumV, double maximumV)     // Bracket for
                                                          // Brent

      : func(function), grad(gradient), Xk(x_0), X0(x_0),
            XkP1(x_0.size()),TOL(tolerance), minV(minimumV),
            maxV(maximumV), gXk(x_0.size()),
            errors(std::vector<double>()) {}


   // more ...

};
```

The actual algorithm is:

```
void run()
{ // Coordinating function (loop)

   int bits = 50;
   boost::uintmax_t maxIter = 1000;

   int coun = 0;
   std::pair<double, double> result;

   do
   {
      gXk = grad(Xk);
      result = boost::math::tools::brent_find_minima
                    (*this, minV, maxV, bits, maxIter);
      XkP1 = Xk + ((result.first) * gXk);

      // Test for convergence
      error = u::norm_inf(XkP1 - Xk);
      Xk = XkP1;

      // Statistics
      errors.push_back(error);

   } while (error > TOL);

   std::cout << "maxiters " << maxIter << std::endl;
}
```

And it uses the following member functions:

```
const u::vector<double>& result() const
{ // Updated value

   return XkP1;
}


double operator () (double x) const
{ // Function call operator (needed for Brent)

   return func(Xk + (x * gXk));
}
```

Finally, information pertaining to the efficiency and accuracy of the algorithm is given by a reporting function:

```
void Statistics() const
{ // Both strategic and operational reporting

   std::cout << "**** Steepest descent statistics ****" << "\n\n";
   std::cout << "*TOL and [min, max] for Brent: " << TOL
             << ", [" << minV << ", " << maxV << "]" << '\n';
   std::cout << "*Error and number of iterations needed: "
             << error << ", " << errors.size() << "\n\n";

   int n = 11;
   for (auto error : errors)
   {
      std::cout << "(" << n++ << "," << error << ")";
   }

   std::cout << "\nInitial guess: " << X0 << '\n';
   std::cout << "Final result: " << result() << '\n';

   std::cout << "Function value: " << func(result()) << '\n';

   std::cout << '\n';
}
```

A2.8.2 Examples

We give a single example to show how to use the algorithm:

$$x = F(x, y) = \sin(x + y), \quad y = G(x, y) = \cos(x - y) \tag{A2.28}$$

or in least squares form:

$$\min_{(x,y)} \left((x - \sin(x + y))^2 + (y - \cos(x - y))^2 \right). \tag{A2.29}$$

We need to provide the following input describing the initial guess, the function and its gradient:

```
namespace Schaum329
{ // Objective function + gradient; solution is x = .935, y = .998

   double minV = 0.8;
   double maxV = 0.4;

   u::vector<double> Seed()
   {
      u::vector<double> result(2);

      result[0] = 0.5;
      result[1] = 0.5;

      return result;
   }

   double MyFunction(const u::vector<double>& a)
   { // Objective function in NL minimax solvers

      double x = a[0]; double y = a[1];
      double sp = std::sin(x+y); double cm = std::cos(x-y);

      double f = (x-sp);
      double g = (y-cm);

      return f*f +g*g;
   }

   u::vector<double>MyGradient(const u::vector<double>& a)
   { // Gradient function in NL minimax solvers

      double x = a[0]; double y = a[1];
      double sp = std::sin(x+y); double cm = std::cos(x-y);
      double sm = std::sin(x-y); double cp = std::cos(x+y);

      u::vector<double>result(2);
      double fx = 1.0 - cp;
      double gx = sm;
      double fy = cp;
      double gy =   1.0 - sm;

      double f =  2.0*(x-sp);
      double g =  2.0*(y-cm);
      result[0] = f*fx + g*gx;
      result[1] = f*fy + g*gy;

      return result;
   }

} // End of namespace
```

A test program and its output is:

```
using namespace Schaum329;              // Checked

u::vector<double>Xk(Seed());            // Copy seed
u::vector<double>XkP1(Xk.size());

// Local bracket
double minBrent = -3.0;
double maxBrent = 3.0;

double TOL = 1.0e-9;
SteeepestDescent sa
      (MyFunction, MyGradient, Xk, TOL, minBrent, maxBrent);

sa.run();

std::cout << sa.result() << ", "
      << MyFunction(sa.result()) <<std::endl;
sa.Statistics();

**** Steepest descent statistics ****

*TOL and [min, max] for Brent: 1e-09, [-3, 3]
*Error and number of iterations needed: 3.08748e-10, 15

(1,0.511685)(2,0.322599)(3,0.067237)(4,0.02977)(5,0.000885976)
(6,0.000644228)(7,5.4778e-05)(8,2.09649e-05)(9,5.78459e-06)(10,4.73895e-
07)(11,3.47721e-07)(12,1.66439e-08)(13,9.4501e-09)(14,1.6198e-
09)(15,3.08748e-10)

Initial guess: [2](0.5,0.5)
Final result: [2](0.935082,0.99802)
Function value: 3.58129e-20
```

You can apply this approach to other problems.

A2.8.3 What's Next?

We have discussed unconstrained univariate and multivariate derivative-based optimisation. In many cases we need to solve constrained optimisation problems as seen in equations (A2.19) to (A2.24). A discussion is outside the scope of this book. Furthermore, the steepest descent method has a number of issues that should be addressed:

1. It may not always converge if the initial guess is not in a neighbourhood of the exact solution.
2. A solution does not always converge to a *global minimum*; in many cases it converges to a *local minimum*.
3. We need to calculate the gradient, either analytically or numerically.

4. For some methods we may need to compute the Jacobian and Hessian, thereby introducing new numerical challenges.
5. These methods are difficult to parallelise.
6. Slow convergence.

There are many algorithms that address these problems, one of which is called *Differential Evolution* (DE) (see Price, Storn and Lampinen, 1996). This method does not need an accurate seed, it does not use derivatives and it can be parallelised. We hope to discuss DE and some of its applications in a later work. Incidentally, we have found DE in combination with a least-squares approach to be the most robust method for computing implied volatility.

A2.9 SUMMARY AND CONCLUSIONS

In this appendix we expanded on the topics introduced in Chapter 19. In particular, we discussed design and code details to make the mathematical content and design approach more tractable. We take the example of computing implied volatility using a variety of methods. We extended the discussion to multivariate optimisation using the *steepest descent* method. Finally, we gave an introduction to simulating mathematical functions and operations using C++.

A2.10 EXERCISES AND PROJECTS

1. (Implementing Context Diagram 19.1)

This exercise examines the design and code in Sections A2.2 and A2.3.

a) Run the solvers from Sections A2.2 and A2.3 for the following changes to option data:

```
// sig = 0.25, K = 60
value_type putMarketPrice = 2.4042'3949'490;
value_type callMarketPrice = 3.5923'1909'650;
```

Experiment with different instances of the classes in Figure 19.1.

b) Stress tests: experiment with ATM, ITM and OTM options. For which range of values do the numerical formulae work?

c) Describe in your own words the advantages of the design approach taken in Section 19.7, in particular, black-box interface specification and how it adds to the readability, maintainability and efficiency of the code.

d) Run the code in parallel as already discussed in this book, for example using C++ threads and tasks.

e) How sensitive are the Banach fixed-point and Aitken algorithms to the choice of initial guess (seed)? In the former case, convergence is assured for any starting value but what can be said of the latter method?

f) Consider making the code for the Banach fixed-point and Aitken algorithms more robust, for example stopping if convergence has not been met within a certain number of iterations or when the denominator in the Aitken algorithm becomes very small.

2. (Computing Square Roots)

Use the Banach fixed-point theorem to find the solutions of the following nonlinear equations:

a) $x^2 = 2, \quad x_{n+1} = \frac{1}{2}\left(x_n + \frac{2}{x_n}\right), \quad n \geq 0 \quad x_0 = 1.$

Can you motivate how we arrived at this iteration?

b) $x = \sin(x + y)$
$y = \cos(x - y)$
$x_0 = y_0 = 0.5$.
Now use the Aitken algorithm to accelerate convergence.

3. (Fixed Points and Polynomials)

We solve the equation $x = f(x)$ by the recursion $x_n = f(x_{n-1}),\ n \geq 0$, for a given x_0. The recursion converges to a root if: $|f'(x)| \leq L < 1$ and the error e_n is given by $e_n \sim f'(r)e_{n-1}$.

Apply this method to solving the equation:

$$F(x) = x^3 + 2x^2 + 10x - 20 = 0$$

by writing it in the form:

$$x = 20/(x^2 + 2x + 10)$$

(the approximate solution is $x = 1.368808107$ and it was originally calculated by Leonardo of Pisa (Fibonacci) in 1225).

4. (Homotopy and ODEs)

The questions in this exercise refer to Section A2.4:

a) Generalise the code so that the class `Davidenko` can be used with any call or put option.

b) Test the software in the same way as in Exercise 1.

c) Investigate the relative accuracy of the following steppers:

```
//   Bode::runge_kutta4<state_type, value_type> myStepper;
     Bode::runge_kutta_dopri5<state_type, value_type> myStepper;
//   Bode::euler<state_type, value_type> myStepper;
//   Bode::bulirsch_stoer<state_type, value_type> myStepper;
//   Bode::runge_kutta4<state_type, value_type> myStepper;
//   Bode::modified_midpoint<state_type, value_type> myStepper;
```

d) Write an observer class to store the solution's intermediate values and corresponding mesh points by modifying the following class and using it in the ODE solver:

```
class WriteOutput
{ // User-defined observer

public:

public:
   WriteOutput() { }

   void operator ()(const state_type &x ,const value_type t)
   {
      // TBD
   }

};
```

5. (Yield Computation, Project)

The goal of this *medium-sized project* is to apply the methods of Chapter 19 and this appendix to computing yield. It is recommended because of the hands-on experience of numerical analysis that you will gain.

We test the bisection and Newton methods from the Boost library on a problem from a fixed-income application. In this case we calculate the yield measure of a bond and hence its relative attractiveness. The yield in this case is the interest rate that will make the present value of the cash flows from the investment equal to the price (or cost) of the investment. The yield y satisfies the equation:

$$P = \sum_{t=1}^{n} \frac{CF_t}{(1+y)^t} \tag{A2.30}$$

where:

P = present value
CF_t = cash flow in year t
n = number of years.

We write this in the form that is suitable for computation:

$$f(y) = P - \sum_{t=1}^{n} \frac{CF_t}{(1+y)^t} = 0. \tag{A2.31}$$

We now have a function of the unknown variable y and in fact we wish to find a root of equation (A2.31).

Answer the following questions:

a) The interval (a,b) in which the solution is to be found must be known.

b) In the case of the bisection method we must have $f(a)f(b) < 0$; the values $a = 0$ (zero interest rate) and $b = 0.25$ (25% interest rate) might be good choices but you may have to experiment with some sample values.

c) In the case of Newton's method we need to calculate the derivative of f with respect to y, that is:

$$\frac{df}{dy} = \sum_{t=1}^{n} \frac{tCF_t}{(1+y)^{t+1}}. \tag{A2.32}$$

d) A common challenge when employing the Newton method is the choice of initial guess (seed). If it is far from the exact solution then the method will fail to converge, as we have discovered in an MBS application based on the Black, Derman and Toy (BDT) model for the short rate. Under certain circumstances we found that the Newton method failed to converge. We resolved the problem by first bracketing the solution to an accuracy of let's say 1/10 using the bisection method to find a rough approximation that is then used as a seed for the Newton method.

e) Apply this hybrid approach to the current problem. Is this method more robust or more efficient than the bisection and Newton methods?

f) Solve the problem using homotopy in combination with the Boost *odeint* library. Do you notice improvements compared to the Newton method, for example insensitivity to the choice of initial guess (seed) as discussed in part d)?

g) Now pose the problem in a (fixed-point) form that is suitable for application of the Banach fixed-point theorem and the Aitken method. Does the latter method converge for all seeds? Now apply the hybrid approach as introduced in part d) in which we first apply the Banach fixed-point method to find a good seed to be used as input to the Aitken method.

h) Now pose the problem as a nonlinear least-squares problem and solve it using Brent's method.

i) Now run parts a) to g) using continuous compounding:

$$P = \sum_{j=1}^{n} C_j e^{-yT_j}$$

where:

P = bond price
y = bond yield continuous compounding
C_j = cash flow (coupon) at time T_j $j = 1, \ldots, n$.

j) Test all the methods on the test case (and other more complex ones of your choosing):

```
double func13(double x)
{ // Bond yield, Hull p. 81 (y = 6.76%)

   double a = 3.0;
   double b = 103.0;

   return a*std::exp(-0.5*x) + a*std::exp(-1.0*x)
         + a*std::exp(-1.5*x) + b*std::exp(-2.0*x)
         - 98.39;

}
```

6. (Mathematical Functions in C++)

In this exercise you build a simple *vector space of functions*. The basic objective is to show how to create *higher-order functions* in C++. It shows what is possible with lambda functions and their applications to mathematics, engineering and numeric computation in general.

We scope the problem drastically; we consider only functions of a single variable returning a scalar value. All underlying types are `double`.

First, create function classes as follows:

```
// Function maps Domain to Range
template <typename R, typename D>
         using FunctionType = std::function<R (const D x)>;

// Working function type
using ScalarFunction = FunctionType<double, double>;
```

Answer the following questions:

a) Create functions to add, multiply and subtract two functions. Create the unary additive inverse of a function and a function representing scalar multiplication. A typical example is:

```
template <typename R, typename D>
      FunctionType<R,D> operator + (const FunctionType<R,D>& f,
                                    const FunctionType<R,D>& g)
{ // Addition of functions

      return [=] (const D& x)
      {

         return f(x) + g(x);
      };
}
```

Typical functionality that you need to create is:

```
// Scalar functions: double to double
   ScalarFunction f = [](double x) {return x*x; };
   ScalarFunction g = [=](double x) { return x; };
   std::cout << g(2) << ", " << g(8) << "*";

   // Set I: Arithmetic functions
   double scale = 2.0;
   auto fA = 2.0*(f + g);
   auto fM = f - g;
   auto fD = f / g;
   auto fMul = f * g;
   auto fC = g << g << 4.0*g;   // Compose
   auto fMinus = -(f + g);
   double x = 5.0;
   std::cout << fA(x) << ", " << fM(x) << ", " << fD(x) << ", "
             << fMul(x)<<", compose: "<< fC(x)
             << ", " << fMinus(x);
```

b) Create trigonometric and other functions such as those in `<cmath>`. For example:

```
template <typename R, typename D>
      FunctionType<R,D> exp(const FunctionType<R,D>& f)
{ // The exponential function

      return [=](const D& x)
      {
         return std::exp(f(x));
      };
}
```

Typical functionality that you need to create is:

```
auto ftmp = log(g); auto ftmp2 = cos(f);
auto f2 = (abs(exp(g))*log(g) + sqrt(ftmp)) / ftmp2;
std::cout << f2(x) << std::endl;
```

c) Finally, produce code that allows you to do the following:

```
auto h1 = min(f, g);
auto h2 = max(f, g);
std::cout << "min(f,g): " << h1(2.0) << '\n';
std::cout << "max(f,g): " << h2(2.0) << '\n';

auto h3 = min(h1, max(2 * f, 3 * g));
auto h4 = max(h2, max(-2 * f, 3 * g));
```

7. (Iterative Method)

We solved system (A2.28) as a least-squares problem in Section A2.8.2 of the current appendix. Now solve this problem by the following iterative scheme:

$$\begin{aligned} x_n &= F(x_{n-1},\ y_{n-1}) \\ y_n &= G(x_{n-1},\ y_{n-1}) \text{ with } x_0 = y_0 = 0.5. \end{aligned} \tag{A2.33}$$

The exact solution is:

$$x \sim 0.934,\ y \sim 0.998. \tag{A2.34}$$

Compare this solution with the least-squares solution in terms of efficiency and accuracy.

8. (New Version of Software)

The code in this appendix and Chapter 19 needs to be made more flexible in order to accommodate other nonlinear solvers such as *stochastic gradient descent* and *simulated annealing* (Goodfellow, Bengio and Courville, 2016).

Answer the following questions:

a) Re-engineer existing code by identifying the equivalents of systems SUD as well as S1 to S5 in Figure 19.1. The existing code should be placed in the appropriate systems. Your new design should not be too dissimilar to the design in Figure 19.1.

b) Test the re-engineered code with examples of your choosing.

c) Formalise and standardise the inter-system interfaces so that other solvers such as DE, Levenberg–Marquardt, simulated annealing and SQP can be accommodated with minimal disruption (Kienitz and Wetterau, 2012).

d) Consider how to incorporate constraints into the framework.

References

Abrahams, D. and Gurtovoy, A. (2005) *C++ Template Metaprogramming*. New York: Addison-Wesley.

Abramowitz, M. and Stegun, I. A. (1965) *Handbook of Mathematical Functions*. New York: Dover.

Agha, G. (1990) *Actors, A Model of Concurrent Computation in Distributed Systems*. Boston, MA: MIT Press.

Ahlberg, J. H., Nilson, E. N. and Walsh, J. L. (1967) *The Theory of Splines and Their Applications*. New York: Academic Press.

Albahari, J. and Albahari, B. (2016) *C# 6.0 in a Nutshell*. Sebastopol, CA: O'Reilly.

Alexander, A. (1979) *The Timeless Way of Building*. Oxford: Oxford University Press.

Alexandrescu, A. (2001) *Modern C++ Design*. New York: Addison-Wesley.

Allgower, E. L. and Georg, K. (2003) *Introduction to Numerical Continuation Method*. Philadelphia, PA: SIAM Publications.

Aziz, A. K. and Hubbard, B. E. (1964) Bounds on the truncation error by finite differences for the Goursat problem. *Mathematics of Computation*, 18(85), 19–35.

Barakat, H. Z. and Clark, J. A. (1966) On the solution of diffusion equation by numerical methods. *Journal Heat Transfer*, 88, 421–427.

Beckers, S. (1980) The constant elasticity of variance model and its implications for option pricing, *Journal of Finance*, 35(3), 661–673.

Berman, K. A. and Paul, J. L. (2005) *Algorithms*. Stamford, CT: Thomson Course Technology.

Bharucha-Reid, A. T. (1972) *Random Integral Equations*. New York: Academic Press.

Bishop, E. (1967) *Foundations of Constructive Analysis*. New York: McGraw-Hill.

Bjerksund, P. and Stensland, G. (2002) Closed-form valuation of American options. Working Paper, NHH.

Boehm, B. (1986) A spiral model of software development and enhancement, ACM SIGSOFT Software Engineering Notes, Vol. 11, Issue 4.

Bovey, R., Wallentin, D., Bullen, S. and Green, J. (2009) *Professional Excel Development*. New York: Addison-Wesley.

Boyle, P. and Guan, F. (2002) The Riccati equation in mathematical finance, *Journal of Symbolic Computation*, 33(3), 343–355.

Brauer, F. and Nohel, J. A. (1969) *The Qualitative Theory of Ordinary Differential Equations*. New York: Dover.

Brennan, M. and Schwartz, E. (1977) The valuation of American put options, *Journal of Finance*, 32(2), 449–462.

Bronson, R. and Naadimuthu, G. (1997) *Operations Research*. Schaum's Outline Series. New York: McGraw-Hill.

Brooks, F. (1995) *The Mythical Man-Month*. New York: Addison-Wesley.

Buchanen, G. R. (1994) *Finite Element Analysis*. Schaum's Outline Series. New York: McGraw-Hill.

Buchova, Z., Ehrhardt, M. and Guenther, M. (2015) Alternating direction explicit methods for convection diffusion equations, *Acta Mathematica Universitatis Comenianae*, 84(2), 309–325.

Campbell, C. and Miller, A. (2011) *Parallel Programming with Microsoft Visual C++*. London: Microsoft Press.

Campbell, L. J. and Yin, B. (2006) *On the Stability of Alternating-Direction Explicit Methods for Advection-Diffusion Equations*. Hoboken, NJ: Wiley Interscience.

Cash, J. R. and Karp, A. H. (1990) A variable order Runge–Kutta method for initial value problems with rapidly varying right-hand sides, *ACM Transactions on Mathematical Software*, 16(3), 201–222.

Chapman, B., Jost, G. and Van der Pas, R. (2008) *Using OpenMP*. Boston, MA: MIT Press.

Clewlow, L. and Strickland, C. (1998) *Implementing Derivatives Models*. Chichester: Wiley.

Conte, S. D. and de Boor, C. (1981) *Elementary Numerical Analysis*. New York: McGraw-Hill.

Cornish, E. A. (1954) The multivariate t-distribution associated with a set of normal sample deviates, *Australian Journal of Physics*, 7(4), 531–542.

Corrado, C. J. and Miller, T. W. (1996) A note on a simple, accurate formula to compute implied standard deviations, *Journal of Banking and Finance*, 20(3), 595–603.

Courant, R. and Hilbert, D. (1968) *Methoden der Mathematischen Physik II*. Berlin: Springer.

Cox, J. C., Ingersoll, J. E. and Ross, S. A. (1985) An intertemporal general equilibrium model of asset prices, *Econometrica*, 53, 363–384.

Cox, J. C. and Ross, S. A. (1976) The valuation of option for alternative stochastic processes, *Journal of Financial Economics*, 3, 145–166.

Cox, J. C. and Rubinstein, M. (1985) *Options Markets*. Englewood Cliffs, NJ: Prentice Hall.

Crank, J. (1984) *Free and Moving Boundary Problems*. Oxford: Clarendon Press.

Dahlquist, G. and Björck, A. (1974) *Numerical Methods*. New York: Dover.

Date, C. J. (1981) *An Introduction to Database Systems*. New York: Addison-Wesley.

De Marco, T. (1978) *Structured Analysis and System Specification*. Boston, MA: Yourdon Press.

Demming, R. and Duffy, D. (2010) *Introduction to the Boost C++ Libraries, Volume I*. Koedijk: Datasim Press.

Demming, R. and Duffy, D. (2012) *Introduction to the Boost C++ Libraries, Volume II*. Koedijk: Datasim Press.

DeRemer, F. and Kron, H. (1975) *Programming-in-the-Large versus Programming-in-the-Small, Proceedings of the International Conference on Reliable Software*. Los Angeles, CA: Association for Computing Machinery, pp. 114–121.

Derman, E. and Kani, I. (1996) The ins and outs of barrier options: Part 1. *Derivatives Quarterly*, Winter, 55–67.

Ditchburn, R. W. (1991) *Light*. New York: Dover.

Dörrie, H. (1965) *100 Great Problems of Elementary Mathematics*. New York: Dover.

Doughtery, R. L., Edelman, A. and Hyman, J. M. (1989) Nonnegativity-, monotonicity-, or convexity-preserving cubic and quintic hermite interpolation, *Mathematics of Computation*, 52(186), 471–476.

Drezner, Z. (1978) Computation of the bivariate normal integral, *Mathematics of Computation*, 32(141), 277–279.

Duffie, D. and Kan, R. (1996) A yield-factor model of interest rates, *Mathematical Finance*, 6, 379–406.

Duffy, D. J. (1980) Uniformly convergent difference schemes for problems with a small parameter in the leading derivative. PhD thesis, Trinity College Dublin.

Duffy, D. J. (2004) *Domain Architectures*. Chichester: Wiley.

Duffy, D. J. (2004A) A critique of the Crank–Nicolson scheme, strengths and weaknesses for financial engineering, *Wilmott Magazine*, July, 68–76.

Duffy, D. J. (2004B) *Financial Instrument Pricing Using C++*, First Edition. Chichester: Wiley.

Duffy, D. J. (2006) *Finite Difference Methods in Financial Engineering*. Chichester: Wiley.

Duffy, D. J. (2006A) *Introduction to C++ for Financial Engineers*. Chichester: Wiley.

Duffy, D. (2009) Unconditionally stable and second-order accurate explicit finite difference schemes using domain transformation: Part I One-factor equity problems. Available at SSRN: https://ssrn.com/abstract=1552926.

Duffy, D. and Germani, A. J. (2013) *C# for Financial Markets*. Chichester: Wiley.

Duffy, D. and Kienitz, J. (2009) *Monte Carlo Frameworks*. Chichester: Wiley.

Emanuel, D. and Macbeth, J. (1982) Further results on the constant elasticity of variance call option pricing model, *Journal of Financial and Quantitative Analysis*, 17(4), 533–554.

Ezust, A. and Ezust, P. (2006) *An Introduction to Design Patterns in C++ with QT4*. Englewood Cliffs, NJ: Prentice Hall.

Fayad, M. E., Schmidt, D. C. and Johnson, R. E. (1999) *Implementing Application Frameworks*. New York: Wiley.

Fiacco, A. V. and McCormick, G. P. (1968) *Nonlinear Programming: Sequential Unconstrained Minimization Techniques*. New York: Wiley.

Fichera, G. (1956) Sullle equazioni differenziali lineari ellittico-paraboliche del secondo ordine, *Atti Accad. Naz. Lincei. Mem. CI. Sci. Fis. Mat. Nat. Sez.* I(8), 5, 1–30. MR 19, 658; 1432.

Friedman, A. (1992) *Partial Differential Equations of Parabolic Type*. New York: Dover.

Gao, S., Zhang, Z. and Cao, C. (2011) Differentiation and numerical integral of the cubic spline interpolation, *Journal of Computers*, 6(10), 2037–2044.

Genz, A. (2004) Numerical computation of rectangular bivariate and trivariate normal and t probabilities, *Statistics and Computing*, 14, 251–260.

Glasserman, P. (2004) *Monte Carlo Methods in Financial Engineering*. New York: Springer.

Godounov, S. K. and Riabenki, V. S. (1977) *Schémas aux Différences*. Paris: Dunod.

GOF (1995) [Gamma, E., Helm, R., Johnson, R. and Vlissides, J.] *Design Patterns*. New York: Addison-Wesley.

Golub, G. H. and van Loan, C. F. (1996) *Matrix Computations*. Baltimore, MD: Johns Hopkins University Press.

Goodfellow, I., Bengio, Y. and Courville, A. (2016) *Deep Learning*. Boston, MA: MIT Press.

Green, A. (2016) *XVA, Credit, Funding and Capital Valuation Adjustments*. Chichester: Wiley.

Gropp, W., Lusk, E. and Skjellum, A. (1997) *Using MPI*. Boston, MA: MIT Press.

Grunspan, C. (2011) Asymptotic expansions of the lognormal implied volatility: A model free approach. Available at SSRN: https://ssrn.com/abstract=1965977.

Haaser, N. B. and Sullivan, J. A. (1991) *Real Analysis*. New York: Dover.

Hagan, P. S. and West, G. (2006) Interpolation methods for curve construction, *Applied Mathematical Finance*, 13(2), 89–129.

Hagan, P. S. and West, G. (2008) Methods for constructing a yield curve, *Wilmott Magazine*, May, 70–81.

Hageman, L. A. and Young, D. M. (1981) *Applied Iterative Methods*. New York: Dover.

Halter, S. (2002) *JavaSpaces Example by Example*. Englewood Cliffs, NJ: Prentice Hall.

Handoko, A. (2014) *One and Two Factors Basket American Option Pricing with Constant Elasticity Variance using Method of Lines*. MSc in Mathematical Finance, University of Birmingham.

Harel, D. and Politi, M. (2000) *Modeling Reactive Systems with Statecharts*. New York: McGraw-Hill.

Hatley, D. J. and Pirbhai, I. M. (1988) *Strategies for Real-Time System Specification*. New York: Dorset House.

Haug, E. (2007) *The Complete Guide to Option Pricing Formulas*. New York: McGraw-Hill.

Heege, M. (2007) *Expert C++/CLI*. New York: Apress.

Henrici, P. (1962) *Discrete Variable Methods in Ordinary Differential Equations*. New York: Wiley.

Heston, S. L. (1993) A closed-form solution for options with stochastic volatility with applications to bond and currency options, *Review of Financial Studies*, 6(2), 327–343.

Hilderbrand, F. B. (1974) *Introduction to Numerical Analysis*. New York: Dover.

Hochstadt, H. (1964) *Differential Equations*. New York: Dover.

Hocking, J. G. and Young, G. S. (1961) *Topology*. New York: Dover.

Hogenson, G. (2008) *Foundations of C++/CLI*. New York: Apress.

Hsu, H. (1997) *Probability, Random Variables and Random Processes*. Schaum's Outline Series. New York: McGraw-Hill.

Hughes, W. F. and Brighton, J. A. (1967) *Fluid Dynamics*. Schaum's Outline Series. New York: McGraw-Hill.

Hull, J. (2006) *Options, Futures and other Derivatives*. Englewood Cliffs, NJ: Prentice Hall.

Hull, J. and White, A. (1996) *Hull–White on Derivatives*. London: Risk Publications.
Hung Tran, T. (2015) *GPUs on Multilevel Monte Carlo and ADE*. MSc in Mathematical Finance, University of Birmingham.
Hunter, J. (1964) *Number Theory*. Edinburgh: Oliver & Boyd.
Il'in, A. M., Kalashnikov, A. S. and Oleinik, O. A. (1962) Linear equations of the second order of parabolic type (translation), *Russian Mathematical Surveys*, 17(3), 1.
Imai, M. (1986) *Kaizen: The Key to Japan's Competitive Success*. New York: McGraw-Hill.
Inamura, Y. (2006) Estimating continuous time transition matrices from discretely observed data. Bank of Japan Working Paper Series No. 06-E07-April.
Isaacson, E. and Keller, H. B. (1966) *Analysis of Numerical Methods*. New York: Dover.
Jackson, M. (2001) *Problem Frames*. New York: Addison-Wesley.
Johnson, N. and Kotz, S. (1972) *Continuous Multivariate Distributions*. New York: Wiley.
Josuttis, N. (2012) *The C++ Standard Library*. New York: Pearson Education.
Joyce, J. (1939) *Finnegans Wake*. London: Faber & Faber.
Kangro, R. and Nicolaides, R. (2000) Far field boundary conditions for Black–Scholes equations, *SIAM Journal of Numerical Analysis*, 38(4), 1357–1368.
Keller, H. B. (1968) *Numerical Methods for Two-Point Boundary-Value Problems*. Waltham, MA: Blaisdell.
Kernighan, B. and Ritchie, D. (1988) *The C Programming Language*. Englewood Cliffs, NJ: Prentice Hall.
Kienitz, J. and Wetterau, D. (2012) *Financial Modelling*. Chichester: Wiley.
Kloeden, P. E. and Platen, E. (1995) *Numerical Solution of Stochastic Differential Equations*. New York: Springer.
Kloeden, P. E., Platen, E. and Schurz, H. (1997) *Numerical Solution of Stochastic Differential Equations*. New York: Springer.
Knuth, D. E. (1997) *The Art of Computer Programming*, Vols 1–3. New York: Addison-Wesley.
Kotz, S., Balakrishnan, N. and Johnson, N. (2000) *Continuous Multivariate Distributions*. New York: Wiley.
Kreider, D. L., Kuller, R. G., Ostberg, D. R. and Perkins, F. W. (1966) *An Introduction to Linear Analysis*. New York: Addison-Wesley.
Kyle, A. S. and Obizhaeva, A. A. (2016) Dimensional analysis and market microstructure invariance. Available at SSRN: https://ssrn.com/abstract=2823630.
Ladyženskaja, O. A., Solonnikov, V. A. and Ural'ceva, N. N. (1988) *Linear and Quasi-linear Equations of Parabolic Type*. Providence, RI: American Mathematical Society.
Lambert, J. D. (1991) *Numerical Methods for Ordinary Differential Equations*. New York: Wiley.
Larkin, B. M. (1964) Some stable explicit difference approximations to the diffusion equation, *Mathematics of Computation*, 18, 196–202.
Lawson, J. D. and Morris, J. L. (1978) The extrapolation of first order methods for parabolic partial differential equations, *SIAM Journal of Numerical Analysis*, 15(6), 1212–1224.
Leavens, G. T. and Sitarman, M. (2000) *Foundations of Component-based Systems*. Cambridge: Cambridge University Press.
Leisen, D. P. J. and Reimer, M. (1996) Binomial models for option valuation – examining and improving convergence, *Applied Mathematical Finance*, 3(4), 319–346.
Lewis, A. L. (2016) *Option Valuation under Stochastic Volatility II*. Newport Beach, CA: Finance Press.
Linetsky, V. (2004) Computing hitting time densities for CIR and OU diffusions: Applications to mean reverting models, *Journal of Computational Finance*, 7(4), 1–22.
Lipschutz, S. and Lipson, M. (1997) *Discrete Mathematics*. Schaum's Outline Series. New York: McGraw-Hill.
Lorenz, M. and Kidd, J. (1994) *Object-Oriented Software Metrics*. Englewood Cliffs, NJ: Prentice Hall.
Lotka, A. J. (1956) *Elements of Mathematical Biology*. New York: Dover.

Mattson, T. G., Sanders, B. A. and Massingill, B. L. (2005) *Patterns for Parallel Programming*. New York: Addison-Wesley.
McConnell, S. (2004) *Code Complete*. London: Microsoft Press.
Meyer, B. (1997) *Object-Oriented Software Construction*. Englewood Cliffs, NJ: Prentice Hall.
Meyer, G. H. (2015) *The Time-Discrete Method of Lines for Options and Bonds*. Singapore: World Scientific.
Miller, G. A. (1956) The magical number seven, plus or minus two, some limits on our capacity for processing information, *Psychological Review*, 63(2), 81–97.
Miner, D. and Shook, A. (2012) *MapReduce Design Patterns*. Sebastopol, CA: O'Reilly.
Moler, C. and Van Loan, C. (2003) Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later, *SIAM Review*, 45(1), 3–49.
Morton, K. W. and Mayers, D. F. (1994) *Numerical Solution of Partial Differential Equations – An Introduction*. Cambridge: Cambridge University Press.
Müller, S. (2009) The binomial approach to option valuation. Thesis, Technical University Kaiserslautern.
Mun, J. (2002) *Real Options Analysis*. New York: Wiley.
Nadarajah, S. and Kotz, S. (2005) Probability integrals of the multivariate t distribution, *Canadian Applied Mathematics Quarterly*, 13(1), 43–87.
Natenberg, S. (2007) *Option Volatility Trading Strategies*. Chichester: Wiley.
Nelson, D. B. and Ramaswamy, K. (1990) Simple binomial processes as diffusion approximations in financial models, *Review of Financial Studies*, 3(3), 393–430.
Nichols, B., Buttlar, D. and Farrell, J. (1996) *Pthreads Programming*. Sebastopol, CA: O'Reilly.
Oleinik, O. A. and Radkevic, E. V. (1973) *Second Order Equations with Nonnegative Characteristic Form*. Translated from the Russian by Paul C. Fife. Providence, RI: American Mathematical Society.
Parnas, D. (1972) On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(5), 330–336.
Patie, P. and Winter, C. (2008) First exit time probability for multidimensional diffusions: A PDE-based approach, *Journal of Computational and Applied Mathematics*, 222, 42–53.
Pealat, G. and Duffy, D. (2011) The alternating direction explicit (ADE) method for one-factor problems, *Wilmott Magazine*, July, 54–60.
Pólya, G. (1990) *How to Solve It*. Harmondsworth: Penguin Books.
Pólya, G. (1990A) *Mathematics and Plausible Reasoning*. Princeton, NJ: Princeton University Press.
POSA (1996) [Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M.) *Pattern-Oriented Software Architecture: A System of Patterns*. Chichester: Wiley.
Press, W. H., Teukolsky, S. A., Vetterling, W. T. and Flannery, B. P. (2002) *Numerical Recipes in C++*. Cambridge: Cambridge University Press.
Price, V. P., Storn, R. M. and Lampinen, J. A. (1996) *Differential Evolution*. New York: Springer.
Quinn, M. J. (2004) *Parallel Programming in C with MPI and OpenMP*. New York: McGraw-Hill.
Rannacher, R. (1984) Finite element solution of diffusion problems with irregular data, *Numerische Mathematik*, 43, 309–327.
Roache, P. J. (1998) *Fundamentals of Fluid Dynamics*. Albuquerque, NM: Hermosa.
Rogerson, D. (1997) *Inside COM*. London: Microsoft Press.
Ron, U. (2000) A practical guide to swap curve construction. Working Paper, Bank of Canada.
Rothe, E. (1930) Zweidimensionale parabolische Randwertaufgaben als Grenzfall eindimensionaler Randwertaufgaben, *Mathematische Annalen*, 102, 650–670.
Rubinstein, M. (1991) Options for the undecided, *Risk Magazine*, 4(4), 43.
Saul'yev, V. K. (1964) *Integration of Equations of Parabolic Type by the Method of Nets*. Oxford: Pergamon Press.
Scales, L. E. (1985) *Introduction to Non-linear Optimization*. Basingstoke: Macmillan.
Sharp, N. J. (2006) Advances in mortgage valuation: An option-theoretic approach. PhD thesis, Manchester University.

Shaw, M. and Garlan, D. (1996) *Software Architecture: Perspectives on an Emerging Discipline*. Englewood Cliffs, NJ: Prentice Hall.
Sheppard, R. (2007) Pricing equity derivatives under stochastic volatility: A partial differential equation approach. MSc thesis, University of the Witwatersrand.
Shilov, G. E. (1977) *Linear Algebra*. New York: Dover.
Smart, D. R. (1974) *Fixed Point Theorems*. Cambridge: Cambridge University Press.
Smith, G. D. (1978) *Numerical Solution of Partial Differential Equations: Finite Difference Methods*. Oxford: Oxford University Press.
Sommerville, I. and Sawyer, P. (1997) *Requirements Engineering*. Chichester: Wiley.
Stakgold, I. (1998) *Green's Functions and Boundary Value Problems*. Hoboken, NJ: Wiley Interscience.
Stoer, J. and Bulirsch, R. (1980) *An Introduction to Numerical Analysis*. New York: Springer.
Strang, G. and Fix, G. (1993) *An Analysis of the Finite Element Method*. Englewood Cliffs, NJ: Prentice Hall.
Stroud, A. H. (1974) *Numerical Quadrature and Solution of Ordinary Differential Equations*. New York: Springer.
Stroud, A. H. and Secrest, D. (1966) *Gaussian Quadrature Formulae*. Englewood Cliffs, NJ: Prentice Hall.
Takahasi, H. and Mori, M. (1974) Double exponential formulas for numerical integration, *Publications of the Research Institute for Mathematical Science*, 9(3), 721–741.
Tannehill, J. C., Anderson, D. A. and Pletcher, R. H. (1997) *Computational Fluid Mechanics and Heat Transfer*. Abingdon: Taylor and Francis.
Tavella, D. and Randall, C. (2000) *Pricing Financial Instruments, The Finite Difference Method*. New York: Wiley.
Templeman, J. (2013) *Microsoft Visual C++/CLI*. London: Microsoft Press.
Thomas, J. W. (1995) *Numerical Partial Differential Equations, Volume I: Finite Difference Methods*. New York: Springer.
Thomas, L. H. (1949) *Elliptic Problems in Linear Difference Equations Over a Network*. Technical Report, Columbia University.
Tian, Y. S. (1999) A flexible binomial option pricing model, *Journal of Future Markets*, 19, 817–843.
Tolstov, G. P. (1962) *Fourier Series*. New York: Dover.
Topper, J. (2005) *Financial Engineering with Finite Elements*. Chichester: Wiley.
Trefethan, L. N. (2011) Six myths of polynomial interpolation and quadrature, *Mathematics Today*, 47(4), 184–188.
Tsokos, C. P. and Padgett, W. J. (1974) *Random Integral Equations*. New York: Academic Press.
Varga, R. S. (1962) *Matrix Iterative Analysis*. Englewood Cliffs, NJ: Prentice Hall.
Vichnevetsky, R. and Bowles, J. B. (1982) *Fourier Analysis of Numerical Approximations of Hyperbolic Equations*. Philadelphia, PA: SIAM Publications.
Walsh, J. B. (2003) The rate of convergence of the binomial tree scheme, *Finance and Stochastics*, 7(3), 337–361.
West, G. (2004) *Better Approximation to Cumulative Normal Functions*. Report of the Financial Modelling Agency.
Widder, D. V. (1989) *Advanced Calculus*. New York: Dover.
Wijngaarden, A. van and Scheen, W. L. (1949) *Tables of Fresnel Integrals*. Amsterdam: North Holland.
Wildi, T. (1995) *Metric Units and Conversion Charts*. New York: IEEE Press.
Williams, A. (2012) *C++ Concurrency in Action*. New York: Manning.
Wilmott, P. (2006) *Paul Wilmott on Quantitative Finance, Vols 1–3*. Chichester: Wiley.
Wilmott, P., Howison, S. and Dewynne, J. (1995) *The Mathematics of Financial Derivatives, A Student Introduction*. Cambridge: Cambridge University Press.
Wilmott, P., Lewis, A. and Duffy, D. (2014) Modelling volatility and valuing derivatives under anchoring, *Wilmott Magazine*, 73, 48–57.

Wirfs-Brock, R., Wilkerson, B. and Wiener, L. (1990) *Designing Object-Oriented Software*. Englewood Cliffs, NJ: Prentice Hall.

Wong, H. Y. and Jing, Z. (2008) An artificial boundary method for American option pricing under the CEV model, *SIAM Journal of Numerical Analysis*, 46(4), 2183–2209.

Yanenko, N. N. (1971) *The Method of Fractional Steps*. New York: Springer.

Index