A STEP-BY-STEP GUIDE TO
DERIVATIVES PRICING IN C++

C++ FOR QUANTITATIVE FINANCE

Object-oriented C++ for derivatives
using Monte Carlo and Finite Differences.

By Michael L. Halls-Moore

Contents

Chapter 1

Introduction to the Book

1.1 Introduction to QuantStart

QuantStart was founded by Michael Halls-Moore, in 2010, to help junior quantitative analysts (QAs) find jobs in the tough economic climate. It now includes book reviews, quant finance articles and programming tips. Since March 2010, QuantStart has helped over 20,000 visitors to become quantitative analysts. You can always contact QuantStart by sending an email to mike@quantstart.com.

1.2 What is this book?

The C++ For Quantitative Finance book is designed to teach junior/prospective quants with some basic C++ skills to be a professional grade quantitative analyst with advanced C++ skills. The book describes advanced features of C++ such as templates, STL, inheritance, polymorphism and design patterns. In addition, the book applies these features to numerical methods used by quants, such as Finite Differences and Monte Carlo, to numerically determine the price of derivatives. The book is driven by examples and includes many projects to implement the principles and methods taught.

1.3 Who is this book for?

The book has been written for prospective and junior quantitative analysts who have some exposure to programming with C++ and wish to learn how to program in a professional environment. It is designed for those who enjoy self-study and can learn by example. Much of the book is about actual programming and implementation - you can't be a good quant unless you implement your mathematical models!

Senior quants will also find the content useful. Alternative implementations of a particular model or tips and tricks to improve compiler efficiency and software project management will come in useful, even for seasoned financial engineers.

1.4 What are the prerequisites?

You should have a strong mathematical background. Calculus, linear algebra and stochastic calculus will be necessary for you to understand the methods such as Finite Differences and Monte Carlo that we will be implementing.

You should be aware of elementary programming concepts such as variable declaration, flow-control (if-else), looping (for/while) and the basic compilation process for C++ programs. You should be familiar with an IDE or C++ compiler (see below).

You should be familiar with basic quantitative finance concepts such as pricing by binomial trees, stochastic calculus and risk neutrality. You should be aware of the common types of financial derivatives and payoffs, such as vanilla calls and puts and exotic options such as Asian or Barriers.

If you are rusty on this material, or it is new to you, have a look at the QuantStart reading list: http://www.quantstart.com/articles/Quant-Reading-List-Derivative-Pricing and look for "Mathematical Finance".

1.5 Software Requirements

Quantitative finance applications in C++ can be developed in Windows, Mac OSX or Linux. This book is agnostic to the end software, so it is best to use whatever C++ environment you're currently comfortable with.

If you use Windows, you can download Microsoft Visual Studio Express Edition 2012, which comes with Visual C++. There are some great tutorials on how to get started with it, but this book assumes you know how to use an Integrated Development Environment (IDE).

If you use Mac OSX or Linux you can use a text editor such as Vim or Emacs, along with the GCC compiler toolset. On Mac OSX specifically, you can download the XCode Development Tools as an alternative IDE.

1.6 Book Structure

The book is broken down into chapters, each of which will concentrate either on a feature of C++ or on a particular implementation of a pricing model. The first set of chapters concentrate on intermediate features of C++, along with implementation examples of basic quantitative finance concepts. These initial chapters will teach you about the language features, which will be applied to the implementation of quantitative models later on in the book.

You will cover the following topics:

- Object Oriented Programming (OOP)
- Option and PayOff Classes
- Pass-by-Value/Pass-by-Reference
- Const Keyword
- Inheritance
- Abstract Base Classes
- Virtual Destructors
- Function Objects
- Operator Overloading
- Generic/Template Programming
- Standard Template Library (STL)

Later classes will discuss mathematics and statistics. You will look at how to matrices in C++, how to solve matrix equations and how to generate random numbers and statistical classes. These will be essential tools in our C++ quant toolbox.

Then we will discuss options pricing techniques using Monte Carlo Methods (MCM) and Finite Difference Methods (FDM) in detail. We will create basic MCM and FDM solvers for some options and then use them to calculate the "Greeks" (option price sensitivities) and how to hedge. Beyond MCM and FDM, we will study advanced pricing engines. In particular, we will study Exotic/Path-Dependent options, Jump-Diffusion Models and Stochastic Volatility Models.

1.7 What the book does not cover

This is not a beginner book on C++, programming or quantitative finance. It will not teach you about variable declaration, branching, looping or how to compile a program. If you need to brush up on those concepts, take a look at this article on the QuantStart website:

http://www.quantstart.com/articles/Top-5-Essential-Beginner-C-Books-for-Financial-Engineers

The book has not been designed to teach you what a Partial Differential Equation (PDE) is or what a derivative is (mathematical or financial!). It certainly is not an introduction to stochastic calculus. Make sure you come into this book with a good mathematical background (an undergraduate degree in Mathematics, Physics or Engineering is approximately the correct level).

1.8 Where to get help

The best place to look for help is the articles list, found at http://www.quantstart.com/articles or by contacting me at mike@quantstart.com. I've written many articles about basic quantitative finance, so you can brush up by reading some of these. Thanks for pre-ordering the book and helping to support me while I write more content - it is very much appreciated. Good luck with your quant career plans! Now onto some quant finance...

Chapter 2

Introduction to C++

2.1 A Brief History Of C++

C++ was created in 1979 by Bjarne Stroustrup. He provisionally called it "C with Classes", as the C programming language does not possess any real object orientated features. *Although some think C++ was the first "official" object oriented language, Simula holds the distinction instead.*

In 1983 the name changed to *C++*. This was a pun on the increment operator ("++"), hence it referred to an incremental improvement in C. New features were added such as virtual functions and operator overloading. In 1989 version 2.0 was released with multiple inheritance, abstract classes, const members and protected members. Later revisions of C++ in 1990 added templates, exceptions, namespaces and new casts.

In 2011 a fully-updated version ("C++11") was released with extensive multi-threading and generic programming support. This is the version we will be using. However, the reality is that many firms are not using C++11 compliant compilers at this stage. Hence we will not be making heavy use of the new features.

2.2 The Advantages Of C++

As a language C++ is very "close to the hardware". This means that one can allocate and deallocate memory in a custom-defined manner, as well as optimise iteration and looping to a significant degree. This can lead to extremely high performance assuming the programmer has the skills to take advantage of both the hardware and software optimisations available.

C++ allows *multi-paradigm programming*. This means that it is extremely versatile with regards to *how* one should program with it. C++ is often referred to as a "federation of languages". Procedural, object-oriented and functional styles are all supported in various guises.

Generic programming and the Standard Template Library (STL) provide sophisticated type-independent classes and extensive, well-optimised code "out of the box". You can spend more time implementing financial models and less time worrying about how to optimise loops or iteration. This is a fantastic time-saver.

Further, C++ is an ISO standard, so it will be around for a long time. Becoming a very good C++ programmer is certainly a secure career path!

2.3 The Disadvantages Of C++

C++ is not a "beginner" language compared to more modern programming options such as Python, Ruby, MatLab or R. It provides far more control than is often required and thus can be confusing.

It is very easy to make a mistake, create bugs or write poorly optimised code in C++, especially where *pointers* are involved. Despite the fact it has a compilation process, many bugs will slip through at compile-time. Debugging C++ programs is not straightforward especially where memory allocation is concerned.

Compared to languages like Python, MatLab or R, C++ will often require more lines of code (LOC). This means greater maintenance overhead, more testing and thus more chances for bugs to appear. It is not a language to create model "prototypes" with.

The language is so large and feature-heavy that two experienced C++ programmers are able to solve problems in vastly different ways - this leads to interoperability issues on large projects. Standards must be enforced from the outset for effective development.

2.4 The Different Components Of C++

As previously stated, C++ can be seen as a federation of many programming styles. Five fundamental components of C++ are listed here, although there are many more subcomponents, which we will be studying in depth:

- **Procedural, C-Style** - Functions and modules, but no object-orientation or templates
- **Object-Orientation and Classes** - Encapsulation of member data and access
- **Generic Programming and Templates** - Type-independent classes for code re-use

- **Standard Template Library** - Optimised code for containers, algorithms and I/O
- **Boost Libraries** - Well-tested additional third-party utility libraries

We will now look at each of these five components in turn.

2.5 C-Style Procedural Programming

C++ is a superset of the C language. Hence any C program will compile in a C++ compiler (in almost all cases!). In particular, this means that C libraries will still function in your C++ code. Be aware though that the interfaces are likely to differ substantially. C function interfaces generally make heavy use of pointers and function pointers, whereas C++ interfaces will often utilise an OOP approach, making use of templates, references and the STL. For that reason, many popular C libraries have had *wrappers* written for them that provide a C++ interface over older C libraries.

It is possible to program procedurally in C++, without any reference to classes or templates. Functions are often grouped into modules of similar functionality. However, just remember that your interfaces will be constrained in the manner described above. It is often more practical, when programming procedurally in C++, to utilise more advanced libraries from the STL such as the iostream, vector and string libraries.

2.6 Object-Oriented Programming

C++ provides extensive data encapsulation features. Objects can be created to represent entities, via the notion of a *class*. The job of a class is to encapsulate all functionality related to a particular entity, exposing the necessary parts as a *public interface* and hiding any implementation details via a *private interface*. C++ classes encapsulate data and data manipulation through the **public**, **private** and **protected** keywords. This allows detailed specification of interfaces to member data. These terms will be explained in detail when we begin creating our first objects.

C++ classes support virtual functions, operator overloading, multiple inheritance and polymorphism. These concepts allow the modelling of the *is-a* relationship as well as the *has-a* relationship between objects. The latter is known as *composition*. These paradigms promote code-reuse, maintainability and extensibility. We will be making extensive use of these object-oriented techniques for our quantitative finance code.

Best practice *design patterns* have emerged over the years to encourage solutions and implementation procedures for specific object-oriented design problems. Patterns are useful because if the underlying mechanic is understood for a particular model implementation, it is more straightforward for additional quants/programmers to familiarise themselves with the codebase, thus saving extensive amounts of time.

2.7 Generic Programming

Generic programming is a strong feature of C++. It allows the creation of *type-independent classes*. They are a key component of the Standard Template Library (STL).

The fundamental idea is that classes can be created with a set of type parameters, much like a function or method. Upon instantiation of the class, types can be provided. For instance, a Matrix class can be made into a template class. It could be a Matrix of double precision values of a matrix of complex numbers. Code for each specific type implementation is generated at compile time only if such an instantiation is required. Generic programming can save a lot of code duplication, at the expense of larger executable files.

Generic programming is extremely useful in quantitative finance, particularly for sequence container classes. For instance, a template sequence container (such as a vector) could contain double precision numbers or Vanilla Options, with no additional code necessary to support extra objects.

Generic programming also leads to an advanced paradigm known as *Template Metaprogramming*. This is a coding concept that allows *compile-time execution* and thus potential optimisation in certain circumstances. However we will not be looking at this, as it is quite an advanced concept and beyond the scope of this book!

2.8 The Standard Template Library (STL)

The STL is a collection of optimised template libraries for common data structures and algorithms applied to those structures. It contains many subcomponents:

- **Sequence Containers**, e.g. Vectors, Lists, Stacks, Queues and Deques (double-ended queues).
- **Associative Containers**, e.g Sets, Maps, Multi-Maps
- bf Algorithms, e.g. Sort, Copy, Reverse, Min, Max
- **Iterators** - Abstracted objects that allow iteration across ("stepping through") a container
- **Input/Output** - iostream, fstream, string

We will discuss components of the STL related to quantitative finance in significant depth throughout the course.

2.9 The Boost Libraries

Boost is a set of third-party libraries that include many additional features for C++ development. Many of the libraries have made it into the "TR1" C++ standard and have appeared in C++11. It includes support for many "day to day" activities such as File System interface abstraction, regular expressions (regex), threading, smart pointers and networking. It is well-tested and is used in a significant number of high-end production applications.

As quantitative analysts, we are particularly interested in the mathematics and statistics libraries, which we will discuss in later lessons.

Visit the site: www.boost.org

2.10 C++ In Quantitative Analysis

C++ has a long history within quantitative finance. The majority of investment banks and many hedge funds make use of C++ for their derivative pricing libraries and quantitative trading applications, due to its speed and extensibility. It is an ISO standard so will be around for a long time.

QuantLib, the C++ quantitative finance library, makes extensive use of the third-party Boost libraries. It contains a wealth of well-tested derivatives pricing code. It can be somewhat daunting for the beginner quant as it contains a substantial amount of code.

One question we need to answer is why learn C++ over Python, MatLab or R? The short answer is that most of the quantitative analysis positions in bulge bracket banks will almost certainly see it as a requirement. It will definitely set you ahead of the pack if you know it inside-and-out, especially its use in quantitative finance. The longer answer is that it will teach you to be an extremely competent programmer and other quantitative languages such as Python, MatLab or R will be easy to pick up afterwards.

What C++ gains in flexibility and speed of execution, it loses in added complexity. It is a very comprehensive tool, but also one that can be difficult to use - especially for the beginner.

2.11 C++ In This Course

I have already mentioned in the previous chapter that this book has been written to teach you the intermediate/advanced features of the language in a quantitative finance setting, via extensive implementation and examples. We will be making use of many of the aforementioned paradigms when designing our derivatives pricing engines. In particular, we will be using:

- **Procedural/C-Style** - Functionality modules, independent of encapsulation
- **STL** - Matrix Classes, Matrix Solvers, Asset Price Paths
- **Object Orientation** - MCM/FDM solvers, PayOff hierarchies, function objects
- **Generic Programming** - Matrix/Vector operations on differing function types
- **Boost Libraries** - Smart Pointers for memory allocation and object instantiation

Chapter 3

Your First Quantitative Finance C++ Program

3.1 Components Of An Object-Oriented C++ Program

Our goal for this chapter is to create a basic VanillaOption class, which will encapsulate the necessary data around a European vanilla call option derivative. We will make use of C++ object-oriented programming (OOP) techniques to implement this class. In particular, we will use *private member data* to encode the option parameters and expose access to this data via *accessor* methods.

In C++, each class is generally specified by two separate files that describe it in full. The first is a *declaration* file, also known as the *header* file. Header files have a file extension denoted by .h or .hpp. The second file is the *implementation* file, also known as the *source* file. Source files have a file extension denoted by .cpp.

The header file specifies the declaration and the interface of the class. It does not include any implementation, unless some of the functions or methods are declared *inline*. The source file describes the implementation of the class. In particular, the mechanics of all methods and any *static functions* are described here. The separation of these two files aids readability and maintainability over a large codebase.

Note: A good question to ask at this point is "Why are the header and source file separated?". This is a legacy issue from C. When you ***#include*** *a file in C++ the preprocessor step carries out a direct text substitution of the header file. Hence, if all of the functions and methods were declared inline, the preprocessor would be pulling all of this implementation into the compilation* *step, which is unnecessary. In languages such as Java and C#, forward declaration occurs and so there is no need, beyond style/readability reasons, to have separate files.*

3.2 What Are Classes?

We will now discuss what classes *really* are. A C++ class is designed to *encapsulate* data and methods (class functions) acting upon this data. A class hides details of its operations from other program components that do not need access to, or need to be aware of, those underlying details.

A class in C++ consists of member data, which can possess three levels of access:

- **Private** member data can only be accessed by methods within the class
- **Public** member data can be accessed by any other aspect or component of the whole program
- **Protected** member data can only be accessed by inherited classes of this class (more on this later)

Class methods are separated into four categories: *Constructors*, *destructor*, *selectors* and *modifiers*, which we will now proceed to describe.

3.3 Constructors And Destructors

A *constructor* is a special method on a C++ class. It is the first method to be called when a C++ class object is instantiated (created!). There are three types of constructor in C++: the *default* constructor, the *parameter* constructor and the *copy* constructor.

A default constructor allows a class object to be instantiated without the need to specify any input data. Default constructors do not have to be implemented in a class. However, if you wish to use a class in a sequence container, for instance, such as a vector of VanillaOption classes (e.g. vector<VanillaOption>), then the class must provide a default constructor.

A parameter constructor requires one or more parameters in order to be called. Usually these parameters are tied into the initialisation of member data. For instance, we may wish to store a strike price as member data and thus the parameter constructor would contain a strike value as a parameter. In some situations the member data can be initialised directly from the body of the constructor. In other instances a *member initialisation list* is necessary. The latter situation is necessary when the member data is not a simple type (and thus a separate class) and requires its own parameters for construction.

The final type of constructor is the copy constructor. A copy constructor allows an object to be "cloned" from another, by providing a reference to the first as the only parameter in the copy constructor signature. It is only necessary to specify a copy constructor explcitly when you wish to override the default provided by the compiler. This is the case when there are special memory allocation needs, such as when allocating memory via the *new* operator.

A closely related method, although not technically a constructor, is the *assignment operator*. It allows one class instance to be set to another via the = operator. One only needs to explicitly define an assignment operator in times of explicit memory allocation, as with the copy constructor. Otherwise the compiler will generate one.

A destructor provides the implementation on how to deallocate any memory allocated upon object creation. They are necessary methods because if memory is allocated by the class upon instantiation, it needs to be "freed up" (i.e. returned to the operating system) when the object *goes out of scope*. Since all objects are defined with regards to a particular "block of code", i.e. a *scope*, as soon as the code path leaves that scope, the object ceases to exist. Any reference to that object subsequently or prior to entering this scope will cause a compilation error as the object is essentially undefined at that point because either it doesn't exist yet or has been destroyed.

3.4 Selectors And Modifiers

C++ provides very fine-grained control of how member data can be accessed by other components of a program. This means that an *interface* to data can be extremely well specified. In quantitative finance, this means that we can make the end user's life (known as the "client") much easier by only exposing to them aspects of the object that they will need to see and interact with. For instance, if we create an Option model, we may expose methods to modify the option parameters and calculate Greeks/sensitivities, but we do not need to allow the user to interact with the underlying pricing engine, such as Monte Carlo or Finite Differences. This is the essence of encapsulation.

Selector methods have the ability to read member data and provide calculations based upon that data. Crucially, they are unable to modify any of the member data. An example of a selector method would be a *getter*, which is a method used to return a value of some member data. Another example would be a price calculation method that has no need to modify the data, only the need to read it and calculate with it.

Modifier methods are able to read member data as well as change that data. They can accomplish everything a selector can do, with additional privileges. An example of a selector mehtod is a *setter*, which is a method used to set directly a value of some member data. We need to be careful to distinguish between methods that cannot modify their parameters and methods that cannot modify anything at all. This is where the **const** keyword comes in. It greatly aids readability for our interface definitions.

The **const** keyword, as well as the mechanisms of *pass by value* and *pass by reference* (all discussed later) determine whether a function/method is a selector or modifier.

3.5 European Vanilla Option Specification

We have now reached the point where we wish to implement some C++ code for a simple European option! Before we even open our favourite C++ coding environment, we must determine our class specification. This is a very good habit to form as it sets expectations between other *clients* of our code. It also allows you to consider flow-paths, edge cases and other issues that could crop up before you set finger to keyboard.

Note: "Client" in this instance could be another quant on your team, a trader who uses your code via an Excel plugin or a quant developer who needs to optimise this code.

We know that the price of a European vanilla option, at least in a Black-Scholes world, is characterised by five parameters. In our VanillaOption class, we will implement these values as private member data. This is because once they have been set, via construction of the class, we do not wish to modify them further. Later on we will consider class interfaces where we may wish to modify this data at code run-time. The five parameters of interest are:

- Strike price, **K**
- Interest Rate ("risk free rate"), **r**
- Expiry time of the option, **T**
- Underlying asset price, **S**
- Volatility of the underlying, σ

In addition, European vanilla call and put options are specified by their pay-off functions, f:

- Call: $f_C(S) = \max(S - K, 0)$
- Put: $f_P(S) = \max(K - S, 0)$

We know that the price of such options have analytical solutions. We can implement these as selector methods of our VanillaOption class, since calculating a price does not need to modify any underlying member data.

3.6 VanillaOption Header File

Since this our first C++ file, we will explain everything in depth so there's no confusion!

At the top and bottom of this header file you will notice a set of *preprocessor directives*. The preprocessor step is carried out prior to code compilation. Essentially, it modifies the textual basis of the code that is sent to the compiler. It is a legacy feature from the days of C programming. Preprocessor directives are always preceeded by a # symbol and only take up one line, without the use of a semi-colon, unlike the C++ code itself.

```
#ifndef __VANILLA_OPTION_H
#define __VANILLA_OPTION_H
..
..
#endif
```

In this instance we are using a *conditional directive* that states if the __VANILLA_OPTION_H identifier has not previously been defined, then define it and add the code until the #ENDIF directive. This is a mechanism that stops us importing the same code twice for the compiler, which we obviously do not wish to do! It is helpful to use an indentifier that references the specific file (notice I have included the _H at the end of the identifier) as we wish to carry this step out for both the header and source files.

The next section of the code *declares* the VanillaOption class. It is telling any other files about the member data and methods that exist within the class, without detailing the implementation, which occurs separately in the source file. It is customary to capitalise class names and to avoid the use of an underscore separator, which are used instead for functions and methods. This helps programmers rapidly distinguish them as classes as opposed to functions, methods or variables.

There are two sections within the class that declare the **private** and **public** members and methods:

```
class VanillaOption {
private:
  ..
public:
  ..
};
```

The **private** section contains two methods: init and copy. These are helper methods. Helper methods are not called directly by the client, but are used by constructors and assignment operators to assign member data upon class initialisation and copying. Notice that the copy method has **const** VanillaOption& rhs as its parameter. This is known as a *pass-by-reference-to-const*. We will explain this in depth later on, but for now you can think of it as saying to the compiler “do not copy the rhs parameter and do not modify it either”, when the parameter is passed.

Note: rhs is an abbreviation for “right hand side”, a common style idiom for copy constructors and assignment operators.

Beneath the private methods are the double precision member data values for each of the five previously discussed European vanilla option parameters. Not that both calls and puts have exactly the same parameters, it is only their pay-off functions that differ.

```
private:
  void init();
  void copy(const VanillaOption& rhs);

  double K;        // Strike price
  double r;        // Risk-free rate
  double T;        // Maturity time
  double S;        // Underlying asset price
  double sigma;    // Volatility of underlying asset
```

The **public** section contains two constructors - one default and one parameter-based - an assignment operator and a destructor. The destructor is prepended via the **virtual** keyword. We will go into a substantial amount of depth about virtual methods and virtual destructors in particular later on in the book. For now you should note that a virtual destructor is necessary for correct memory deallocation to occur when we use the process of *class inheritance*. It is a “best practice” to almost always set your destructors to be virtual.

Notice that the copy constructor and assignment operator both use the mechanism of pass-by-reference-to-const, which we briefly discussed above. This is because the copy constructor and assignment operator do not need to modify the object they are copying from, nor do they wish to directly copy it before using it in the method, as copying can be an *expensive* operation in terms of processor usage and memory allocation.

The assignment operator returns a reference to a VanillaOption&, not a VanillaOption directly. Although this is not strictly necessary (you can return void), by returning a reference to a VanillaOption, it allows what is known as chained assignment. Chained assignment allows you to write code such as option1 = option2 = option3 = option4.

```
public:
  VanillaOption();    // Default constructor - has no parameters
  VanillaOption(const double& _K, const double& _r,
                const double& _T,  const double& _S,
                const double& _sigma);     // Parameter constructor
  VanillaOption(const VanillaOption& rhs);               // Copy constructor
  VanillaOption& operator=(const VanillaOption& rhs);   // Assignment
      operator
  virtual ~VanillaOption();                              // Destructor is
      virtual

  ..
```

The next set of methods in the public section are selectors (getters) used to access the private member data. The public block also contains two calculation (selector) methods for the call and put prices, respectively. All of these selector methods return a non-void value (in fact they return a double precision value).

All of the selector methods are post-marked with **const**. This means that these methods are unable to modify member data or anything else! Note that the **const** keyword appears at the end of the method prototype. If it was placed at the beginning it would mean that the method was returning a **const double** value, which is entirely different! A **const double** value cannot be modified once created, which is not the intended behaviour here.

```
public:
  ..

  // Selector ("getter") methods for our option parameters
  double getK() const;
  double getr() const;
  double getT() const;
  double getS() const;
  double getsigma() const;
```

```
// Option price calculation methods
double calc_call_price() const;
double calc_put_price() const;
```

One lesson to learn here is that the const keyword has many different meanings and it is extremely sensitive to placement. One could argue that because of these semantic issues it would be easier not to use it anywhere in the code. This is extremely bad practice, because it states to clients of the code that all functions/methods are capable of modification and that any object can be copied. Do not fall into this seemingly beneficial trap!

3.7 VanillaOption Source File

As with the header file describe above, I have included two preprocessor directives to handle potential code duplication. The logic is identical to that of the header file:

```
#ifndef __VANILLA_OPTION_CPP
#define __VANILLA_OPTION_CPP
..
..
#endif
```

The source file is used to specify the implementation of the previously declared class in the header file. Hence it is necessary to import the header file so the compiler understands the declarations. This is the job of the preprocessor, which is why the **#include** statement is prefixed via a # symbol. We have also imported the cmath library containing the mathematical functions we need such as *sqrt* (square root), *exp* (exponential function) and *log* (natural logarithm).

Notice that vanillaoption.h has been imported with quotes (""), while cmath has been imported with delimiters (<>). The quotes tell the compiler to look in the current directory for vanillaoption.h, while the delimiters let it know that it should search its own installed location to import the mathematical library, as it is part of the C++ standard library.

```
#include "vanilla_option.h"
#include <cmath>
```

After the import preprocessor directives, the source file begins by implementing all of the methods previously described in the header file. The first implemented method is init, which possesses a return type of **void**. Initialisation of data does not need to return any value.

The init() method signature is prepended with VanillaOption::. The double colon ("::") is known as the *scope resolution operator*. It states that init is only defined while within the scope of VanillaOption and hence it is meaningless to call the init method as a naked function, i.e. unless it is attached to a VanillaOption instance.

init is a helper method used by the default constructor. The initialisation of the member data is separated this way, as init is used by other methods to initialise member data and hence it stops repeated code. This pattern is known as Do-not Repeat Yourself (DRY).

In the following code we have implemented some reasonable defaults for our option parameters:

```
// Initialises the member data
void VanillaOption::init() {
    K = 100.0;
    r = 0.05;       // 5% interest rate
    T = 1.0;        // One year until maturity
    S = 100.0;      // Option is "at the money" as spot equals the strike.
    sigma = 0.2;    // Volatility is 20%
}
```

The next implemented method is copy, again with a return type of void, because it also does not need to return any values. The copy method is another helper method used by the copy constructor and assignment operator. It is once again designed to stop repeated code, via the DRY pattern. Notice that the member data of the right-hand side (rhs) VanillaOption is obtained via getter/selector methods, using the "." operator.

The copy method uses the *pass-by-reference-to-const* pattern to once again reduce copying and restrict modification, for performance reasons:

```
// Copies the member data
void VanillaOption::copy(const VanillaOption& rhs) {
    K = rhs.getK();
    r = rhs.getr();
    T = rhs.getT();
    S = rhs.getS();
    sigma = rhs.getsigma();
}
```

Following init and copy are the constructors. Constructors are special methods for classes, straightforwardly identified by their lack of return type (even void). In our code there are two constructors: Default and parameter-based.

The default constructor simply calls the init helper method and does nothing else. The parameter constructor takes in all five parameters necessary to price a European option and then initialises the underlying member data. Notice the underscores (_) prepended to the parameters. This is to avoid having to have alternative names for the option parameters, but still keeping them unique from the member data. The parameters are once again *passed-by-ref-to-const*. This is not strictly necessary here as they are simple types. Copying them would not generate unnecessary overhead.

```cpp
VanillaOption::VanillaOption() { init(); }

VanillaOption::VanillaOption(const double& _K, const double& _r,
                             const double& _T, const double& _S,
                             const double& _sigma) {
  K = _K;
  r = _r;
  T = _T;
  S = _S;
  sigma = _sigma;
}
```

Following the default and parameter constructors are the copy constructor and assignment operator. Once again, as the copy constructor IS a constructor it does not possess a return type. All it does it call the copy helper method. In fact both methods make use of the copy method, ensuring DRY. We have described copy above.

Assignment operators (**operator**=) can be tricky to understand at first glance. The following code states that if the "=" operator is used to assign an object to itself (i.e. code such as my_option = my_option;) then it should not perform the copying (because this is a waste of resources), instead it should just return the original object. However, if they are not equal then it should perform a proper ("deep") copy and then return the new copied object.

The mechanism through which this occurs is via a *pointer* known as **this**. **this** always points to the underlying object, so it is only available within the scope of a class. In order to return a *reference* to the underlying object (VanillaOption&), the pointer must be *dereferenced* with the dereference (*) operator. Hence ∗**this**, when coded within a class method, means "get me a reference to the underlying object which this method belongs to".

Pointers and references are very confusing topics, particularly for beginner C++ programmers. If you are somewhat unsure of pointers and references the following article at QuantStart.com will provide a list of helpful books to help you get up to scratch:

http://www.quantstart.com/articles/Top-5-Essential-Beginner-C-Books-for-Financial-Engineers

```
// Copy constructor
VanillaOption::VanillaOption(const VanillaOption& rhs) {
    copy(rhs);
}

// Assignment operator
VanillaOption& VanillaOption::operator=(const VanillaOption& rhs) {
    if (this == &rhs) return *this;
    copy(rhs);
    return *this;
}
```

In this class the destructor is extremely simple. It contains no implementation! This does NOT mean that the compiled program ends up with no implementation, however. Instead the compiler is smart enough to provide one for us. The VanillaOption class only contains private member data based on simple types (double precision values). Hence the compiler will deallocate the memory for this data upon object destruction - specifically, when the object goes out of scope. There is no need to add in specific code to handle more sophisticated memory deallocation, as might be the case when using the **new** operator. Notice also that the destructor has no return type (not even void), as with constructors.

The destructor does not have to be declared virtual again in the source file, as there can only ever be one destructor per class and hence there is no ambiguity with the header declaration file. This is not the case with constructors, where adding **const**, for instance, actually implements a new constructor, from one that does not possess the **const** keyword.

```
// Destructor
VanillaOption::~VanillaOption() {
    // Empty, as the compiler does the work of cleaning up the simple types
        for us
}
```

Following the destructor are the getter methods used to publicly obtain the values for the private member data. A legitimate question to ask at this stage is “Why are we declaring the data private and then exposing access via a getter method?” In this instance of our class specification it makes the underlying member data read-only. There is no means of modifying the member data once set via the initial construction of the object, in our interface. In other class designs we might want to use setters, which are modifier methods used to set member data. It will all depend upon the client requirements and how the code will fit into the grander scheme of other objects.

I will only display one method here (in this instance the retrieval of the strike price, K) as the rest are extremely similar.

```
// Public access for the strike price, K
double VanillaOption::getK() const { return K; }
```

The final methods to implement are `calc_call_price` and `calc_put_price`. They are both **public** and **const**. This forces them to be selector methods. This is what we want because a calculation should not modify the underlying member data of the class instance. These methods produce the analytical price of calls and puts based on the necessary five parameters.

The only undefined component of this method is the N(.) function. This is the *cumulative distribution function of the normal distribution*. It has not been discussed here, but it is implemented in the original header file for completeness. I have done this because I don't think it is instructive in aiding our object oriented example of a VanillaOption, rather it is a necessary function for calculating the analytical price.

I won't dwell on the specifics of the following formulae. They can be obtained from any introductory quantitative analysis textbook. I would hope that you recognise them! If you are unfamiliar with these functions, then I suggest taking a look at Joshi[12], Wilmott[26] or Hull[10] to brush up on your derivatives pricing theory. The main issue for us is to study the implementation.

```
double VanillaOption::calc_call_price() const {
  double sigma_sqrt_T = sigma * sqrt(T);
  double d_1 = ( log(S/K) + (r + sigma * sigma * 0.5 ) * T ) / sigma_sqrt_T
      ;
  double d_2 = d_1 - sigma_sqrt_T;
  return S * N(d_1) - K * exp(-r*T) * N(d_2);
}

double VanillaOption::calc_put_price() const {
  double sigma_sqrt_T = sigma * sqrt(T);
  double d_1 = ( log(S/K) + (r + sigma * sigma * 0.5 ) * T ) / sigma_sqrt_T
      ;
  double d_2 = d_1 - sigma_sqrt_T;
  return K * exp(-r*T) * N(-d_2) - S * N(-d_1);
}
```

3.8 Passing By Reference Or Value

Previously we have mentioned the concepts of *passing by value* and *passing by reference*. These concepts refer to how a parameter is passed to a function or method. When passing by value, the type or object being passed is copied. Any modification or access to that object, within the function, occurs on the copy and not on the original passed object. If the type has a small memory footprint, such as an **int** or **double** this copying process is rapid.

If the object being passed by value is a large matrix containing double values, for instance, it can be extremely expensive to copy as all values within the matrix need to be duplicated and stored. We usually want to avoid situations like this. Following is an example of a function prototype passing a double precision type by value:

```
double my_function(double strike_price);
```

The following is an example of an object (in this case a Matrix) being passed by reference into a matrix norm method. The norm returns a double precision value:

```
double norm(Matrix& mat);
```

When an object is passed by reference, the function can modify the underlying object. We need to be careful that this is the intended behaviour, otherwise we can end up modifying an object in place. The compiler would not stop us from doing so, even if we intended not to. This can lead to bugs!

What if we want to restrict copying for performance reasons (and hence pass by reference), but do not want the function to ever modify the passed object data? We use the **const** keyword and *pass-by-reference-to-const*. The compiler will catch any attempts to modify our Matrix, mat, within the function. This is why getting into the habit of using **const** liberally in your class definitions is so important. It not only helps reduce bugs, it provides a "contract" with clients that lets them know what functions/methods are able to do to their data. Following is an example of *pass-by-reference-to-const*:

```
double norm(const Matrix& mat);
```

We can also append the **const** keyword to the function prototype. This stops the function modifying any data whatsoever. In this case, it might be data which resides in a larger scope than the function itself. A norm function should not really be modifying any data at all so we postfix it with **const**:

```
double norm(const Matrix& mat) const;
```

In summary, this states that norm accepts a non-modifiable, non-copied Matrix object and does not have the ability to modify anything else while returning a double precision value - exactly what we want!

This concludes our first quantitative finance C++ program. In the next chapter we will look at how to deal with option pay-off functions in a maintainable, reusable and efficient way.

Chapter 4

Option Pay-Off Hierarchies and Inheritance

4.1 Inheritance In Quantitative Finance

Inheritance is a powerful concept in object-oriented programming (OOP). It allows us to model a type of relationship between objects known as *is-a*. For instance, we know that an American Call Option *is a* Call Option. Hence an American Call Option should *inherit* all properties that a Call Option possesses, such as a strike price and an underlying asset price.

Inheritance is extremely useful in quantitative finance, as there are many is-a relationships we can model:

- American, Bermudan and European Options are all Options
- Parabolic, Elliptic and Hyperbolic PDE are all PDE
- Dense, Sparse, Banded and Block-Banded Matrices are all Matrices

The key benefit is that inheritance allow us to create interfaces that accept *superclasses* as parameters, but can also subsequently handle *subclasses*. As an example, we could create a matrix transpose function, which is able to accept dense, sparse, banded or block-banded matrix objects as parameters, when it has been defined as accepting base matrix objects.

4.2 Option Pay-off Hierarchies

Let us consider a problem: What if we wish to modify our VanillaOption class from the previous chapter to separate out the pricing of the option from the option class itself? Ideally we would like to create a second object "OptionSolver" or similar, which can accept an "Option" class and provide a price. This option solver would likely make use of Monte Carlo or Finite Difference methods (which we will describe in full later). At the moment we only have analytical prices for calls and puts, but we would like to allow for additional option types such as digital options, double digital options or power options.

This is not as straightforward as it seems, for the following reasons:

- Other types of options may not have straightforward analytical pricing functions, so we need to use numerical methods.
- We would need to create new, extra selector methods for calculation of these prices and thus must modify our interface and inform our clients.
- We must add extra parameters to the Option class constructor. For instance, a double digital option requires two barrier values, while a power option requires the value of the power as well as spot.
- Every time we add a new pricing function we must re-edit and re-compile both the header and source file for every modification, which could be an expensive operation in a large codebase.

Is there a better way? Yes! It involves separating the *pay-off* for a type of option from the Option class itself. Our plan will be to create a new class, called PayOff. We will then create further subclasses, which *inherit* properties from the PayOff class. For instance, we could create PayOffCall, PayOffPut or PayOffDoubleDigital. All of these inherited subclasses will contain the properties and interface of PayOff, but will be able to extend and override it in their specific cases. This is the essence of object-oriented inheritance. Further, our Option class can accept a *base class* PayOff as a constructor parameter and we are still able to provide a subclass as a parameter, since the expected interface will still be present. This means that Option will not actually know what type of PayOff it will receive, since as far as it is concerned it is dealing with the superclass interface.

4.3 Implenting The PayOff Base Class

The first step towards implementation of a pay-off hierarchy in C++ classes is determining the interface to our base class, PayOff.

We want to restrict the base class itself from ever being instantiated as there is no "default" behaviour for an option pay-off. This means that PayOff will become an *abstract base class*. An abstract base class is a class that has at least one *pure virtual* method. A pure virtual method is a method that does not possess an implementation in the base class itself, but is implemented in an inherited subclass. The compiler restricts the client from instantiating an abstract base class if it detects a pure virtual method within its definition, so it is really just defining an interface.

An inherited subclass of PayOff will provide a virtual public method, which takes an underlying asset spot price as a parameter and provides the pay-off price at expiry (for European options). That is the most general interface we will support at this stage in the book. It is up to each subclass to provide specific implementation details for their own pay-off functions.

Rather than generate a specific method called pay_off(**const double**& spot_price), or similar, we will use another C++ feature known as *operator overloading*. We will provide our own "overriden" implementation of the **operator**() method. This is a special operator that allows us to call the class instance as if it were a function. Since the PayOff class and its subclasses are really just "wrappers" for the pay-off function, it makes sense to specify the interface in this manner. It is clearer for clients and describes the intent of the class hierarchy with more clarity. These types of classes are known as *function objects* or *functors*. We will have more to say about functors later on in the book.

4.4 PayOff Header File

The first line in the PayOff header file, beyond the preprocessor directives (which we won't discuss), is the inclusion of the algorithm library. This is necessary as we need to make use of the max function to determine the larger of two parameters in our pay-off functions:

```
#include <algorithm>
```

The PayOff base class has no private members or methods, as it is never going to be instantiated. It is only designed to provide an interface, which will be carried through in its subclasses. It does possess a single (default) constructor and virtual destructor.

The pure virtual overloaded method, **operator**(), defines PayOff as an (abstract) function object. It takes the underlying asset spot price, S, and provides the value of the pay-off. As stated above this method is not implemented within this class directly. Instead an inherited subclass provides the implementation, specific to that subclass. To specify a virtual method as pure virtual, we need to append "= 0" to the end of the function prototype.

We have already mentioned that **operator**() allows the class to be called like a function. This is an alternative to using C-style function pointers. We will discuss function pointers and function objects in detail later in the book.

```
class PayOff {
public:
    PayOff() {};                // Default (no parameter) constructor
    virtual ~PayOff() {};    // Virtual destructor

    // Overloaded operator(), turns the PayOff into an abstract function
        object
    virtual double operator() (const double S) const = 0;     // Pure
        virtual method
};
```

4.5 PayOffCall Header

The first type of option we wish to create is a European vanilla call option. We will name our class PayOffCall. It inherits from the PayOff base class.

PayOffCall is an inherited subclass of PayOff. It requires greater detail than provided in the abstract base class. In particular we need to implement the pay-off function for a vanilla European call option. We inherit PayOffCall from PayOff using the **class** PayOffCall : **public** PayOff syntax.

The PayOffCall subclass requires a single strike value K in its constructor. It also requires the implementation of **operator**() to provide the pay-off function itself.

Note: We will not discuss PayOffPut. It is almost identical to the code below.

```
class PayOffCall : public PayOff {
private:
    double K;      // Strike price

public:
    PayOffCall(const double K_) {};      // No default constructor
    virtual ~PayOffCall() {};      // Destructor virtual for further
        inheritance
    // Virtual function is now over-ridden (not pure-virtual anymore)
    virtual double operator() (const double S) const;  // Pay-off is max(S-
        K,0)
};
```

4.6 PayOffDoubleDigital Header

We have created two base classes at this point: PayOffCall and PayOffPut. We are also going to create PayOffDigitalCall and PayOffDigitalPut, which represent digital call and put options. We won't describe their declaration here, but for completeness, we will add them to the header and source files.

We have only described PayOffCall in any detail at this stage. Now we will consider a double digital option. If you are unfamiliar with these options, then they are very similar to a digital/binary option except that they have two "strike" prices. If the spot price ends up between the two strike values at expiry, then the value of 1 unit of currency is paid in the respective denominating currency. If spot falls outside of these values at expiry, then the option pays nothing. Our double digital subclass will be denoted PayOffDoubleDigital.

As PayOffDoubleDigital is also a subclass of PayOff, it requires greater detail than provided in the abstract base class, PayOff. Specifically, a double digital option has two strike barriers, U and D, representing the upper and lower barrier, with $U > D$. **operator**() also needs implementation, by specification of the double digital pay-off function:

```
class PayOffDoubleDigital : public PayOff {
private:
    double U;    // Upper strike price
    double D;    // Lower strike price

public:
    // Two strike parameters
    PayOffDoubleDigital(const double U_, const double D_);
    virtual ~PayOffDoubleDigital();

    // Pay-off is 1 if spot within strike barriers, 0 otherwise
    virtual double operator() (const double S) const;
};
```

4.7 PayOffCall Source File

Please note that PayOff is an abstract base class and hence does not need an implementation in the source file, as it is never actually instantiated. However, the remaining pay-off subclasses do require implementation. We will begin with PayOffCall.

The constructor of PayOffCall is straightforward. It assigns the strike parameter to the strike member data, which thus makes it a parameter-based constructor. The destructor does not require the **virtual** keyword in the implementation source file, only in the header, so we omit it.

For PayOffCall, **operator**() simply returns the pay-off function for a European call option. It is marked **const**, which states to the compiler that the method cannot modify any member data (or any other data for that matter!). The parameter S is the underlying asset spot price, which we have modelled as a double precision value:

```
// Constructor with single strike parameter
PayOffCall::PayOffCall(const double _K) { K = _K; }

// Destructor (no need to use virtual keyword in source file)
PayOffCall::~PayOffCall() {}

// Over-ridden operator() method, which turns PayOffCall into a function
    object
double PayOffCall::operator() (const double S) const {
    return std::max(S-K, 0.0); // Standard European call pay-off
}
```

4.8 PayOffDoubleDigital Source File

As with the header file, we won't delve too deeply into the put option versions of these pay-off classes as they are almost identical to the call versions. They are specified in the source file itself, so if you wish to study them, you can look at the file directly.

PayOffDoubleDigital has a slightly modified interface to the PayOffCall, as it requires two strike barriers, rather than a single strike parameter. Hence we modify the constructor to take an upper value, U and a lower value, D. The (parameter) constructor simply assigns these parameters to the respective member data. Later on in the book we will use a more sophisticated mechanism, known as the member initialisation list, to carry out this initialisation procedure.

operator() still takes a spot price S. This is because the *calling* interface for all pay-off classes in our hierarchy is identical. It is only the member data and constructors which change. For a double digital option the pay-off function returns 1.0 if the spot lies between the barriers and zero otherwise, at maturity:

```
// Constructor with two strike parameters, upper and lower barrier
PayOffDoubleDigital::PayOffDoubleDigital(const double _U, const double _D)
    {
    U = _U;
    D = _D;
}

..

// Over-ridden operator() method, which turns
// PayOffDoubleDigital into a function object
double PayOffDoubleDigital::operator() (const double S) const {
    if (S >= D && S <= U) {
        return 1.0;
    } else {
        return 0.0;
    }
}
```

This concludes our overview of the PayOff files. We will now discuss what a *virtual destructor* is and why you would need to prefix a destructor with the **virtual** keyword.

4.9 Virtual Destructors

All of our PayOff class and subclass destructors have so far been set to virtual, using the prefixed **virtual** keyword. Why are we doing this and what happens if we do not use this keyword?

In simple terms, a virtual destructor ensures that when derived subclasses go *out of scope* or are deleted the order of destruction of each class in a hierarchy is carried out correctly. If the destruction order of the class objects is incorrect, in can lead to what is known as a *memory leak*. This is when memory is allocated by the C++ program but is never deallocated upon program termination. This is undesirable behaviour as the operating system has no mechanism to regain the lost memory (because it does not have any references to its location!). Since memory is a finite resource, if this leak persists over continued program usage, eventually there will be no available RAM (random access memory) to carry out other programs.

For instance, consider a pointer to a base class (such as PayOff) being assigned to a derived class object address via a reference. If the object that the pointer is pointing to is deleted, and the destructor is not set to **virtual**, then the base class destructor will be called instead of the derived class destructor. This can lead to a memory leak. Consider the following code:

```
class Base {
public:
 Base();
 ~Base();
};

class Derived : public Base {
private:
  double val;
public:
 Derived(const double _val);
 ~Derived();
}

void do_something() {
 Base* p = new Derived;
 delete p;   // Derived destructor not called!!
}
```

What is happening here? Firstly, we create a base class called Base and a subclass called Derived. The destructors are NOT set to virtual. In our do_something() function, a pointer p to a Base class is created and a reference to a new Derived class is assigned to it. This is legal as Derived *is a* Base.

However, when we delete p the compiler only knows to call Base's destructor as the pointer is pointing to a Base class. The destructor associated with Derived is not called and val is not deallocated. A memory leak occurs!

Now consider the amended code below. The **virtual** keyword has been added to the destructors:

```
class Base {
public:
 Base();
 virtual ~Base();
};

class Derived : public Base {
private:
  double val;
public:
 Derived(const double _val);
 virtual ~Derived();
}

void do_something() {
 Base* p = new Derived;
 delete p;  // Derived destructor is called
}
```

What happens now? Once do_something() is called, **delete** is invoked on the pointer p. At code execution-time, the correct destructor is looked up in an object known as a *vtable*. Hence the destructor associated with Derived will be called prior to a further call to the destructor associated with Base. This is the behaviour we originally desired. val will be correctly deallocated. No memory leak this time!

Chapter 5

Generic Programming and Template Classes

5.1 Why Generic Programming?

In 1990 templates were added to C++. This enabled a new paradigm of programming known as *generic programming.*

In order to explain how generic programming differs from object-oriented programming (OOP) we need to review how classes work. Recall that classes allow encapsulation of data. They provide a declaration of an interface, which determines how an external client accesses that data. Instance objects are created from these classes. You can think of classes as a "mould" or "pattern" for creating objects. Normal classes are bound to the data types specified for their member data. This means that if we want to store values or objects within that class, we must specify the type of data upfront.

5.2 Template Classes

Template classes behave in a different way. They allow us to define classes without the need to specify the types of data that the classes will utilise upfront. This is a very powerful feature as it allows us to create *generic* classes that aren't restricted to a particular set of data types.

This is quite a complicated abstract concept to grasp at first. So let's consider an example, which is a common pattern within the C++ *standard template library* (STL).

5.3 A Matrix Template Class

Consider a mathematical matrix object that uses double precision types to store its values. This class could be implemented as a single array of double values or even an array of arrays of double values. Irrespective of how we implement this class our matrix implementation is likely to make use of pointers to double values.

Let's imagine that our application now requires matrices to store complex numbers. This is a very common situation in computational physics, particularly in quantum mechanics. *I realise I'm deviating from quantitative finance here, but bear with me!* This brings up many questions:

- How we will we implement such a class? What effect will it have on our code duplication and external client code?
- Will we replace our double precision matrix code to make use of the C++ complex library? This would require modifying all of the implementation code to store complex types, instead of doubles. Further, any client code that makes use of this matrix class will need to be modified to handle complex numbers.
- How would we store a real-numbered matrix? Would we set all imaginary parts of each complex component to zero? This would seem like an awful waste of memory!

To create a new matrix object that stores complex numbers, the entire matrix code would need duplication and replacement with the complex type. This is poor practice. Perhaps there is a better way?

Templates allow code to be written once and then parametrised with a concrete type such as **int**, **double** or complex. This means that we can write the code for our matrix object once and then supply various concrete types at the point we wish to use such a matrix.

Templates can thought of as moulds or patterns for creating classes, in exactly the same way that classes can be thought of as moulds or patterns for creating instance objects.

5.4 Template Syntax and Declaration

Let's start gently by studying the syntax of declaring a template, with the aforementioned matrix class example.

To create a template class we prefix the usual class declaration syntax with the **template** keyword, using <> delimiters to pass data types as arguments via the **typename** keyword:

*Note: You can use **class** in place of **typename** in the following declaration, as the syntax is synonymous. I am using **typename** because I find it less confusing!*

```
template <typename T> class Matrix {
  // All private, protected and public members and methods
};
```

Here we have created a template class representing a matrix and are passing it a single type, T. In other areas of this code we could instantiate a template class by passing concrete types (such as **double** and **int**) into the constructor syntax, as with the following examples:

A matrix parametrised with a double precision type:

```
Matrix<double> double_matrix(...);
```

A matrix parameterised with an integer type:

```
Matrix<int> int_matrix(...);
```

Templates can also be nested. That is, we can use one template class (which requires its own type) within the typename of another. For example, consider the aforementioned matrix of complex numbers. If we wanted to create such a class with our template matrix, we would need to not only let the matrix know that it should use the complex type, but we also need to provide the complex type with an underlying storage mechanism (in this case double precision):

```
#include <complex>  // Needed for complex number classes

// You must put a space between end delimiters (> >)
Matrix<complex<double> > complex_matrix(...);
```

Note that you must place an additional space (" ") between the two right hand delimiters (> >) otherwise the compiler will think you are trying to invoke the bit shift >> operator, which is not the desired behaviour.

5.5 Default Types in Templates

In quantitative finance, nearly all values required of any data structure are real numbers. These are often stored as **float** (single precision) or **double** (double precision) types. Writing out extensive type parameter lists when declaring classes can make syntax unwieldy. Thankfully, C++ allows default template values, which provide an antidote to this problem. The default type is simply appended to the type name keyword with the equals sign:

```
template <typename T = double> class Matrix {
  // All private, protected and public members and methods
};
```

Note: Since templates support multiple type parameters, if one parameter has been specified as a default all subsequent parameters listed must also have defaults.

5.6 SimpleMatrix Declaration

We will now flesh out the Matrix class described above as a first example for generic programming with C++. At this stage we will simply use it to store values. It will not have any additional functionality, such as matrix multiplication, at the moment. We will add this functionality in a later chapter.

So what does our Matrix need to do? In its simplest form we must be able to define the type of data being stored, the number of rows and columns as well as provide access to the values in a simplified manner.

Here is the listing for SimpleMatrix.hpp:

```
#IFNDEF __SIMPLE_MATRIX_H
#DEFINE __SIMPLE_MATRIX_H

#include <vector>    // Need this to store matrix values

template <typename Type = double> class SimpleMatrix {
private:
  vector<vector<Type> > mat;   // Use a "vector of vectors" to store the
      values

public:
  SimpleMatrix();   // Default constructor

  // Constructor specifiying rows, columns and a default value
  SimpleMatrix(const int& rows, const int& cols, const Type& val);

  // Copy constructor
  SimpleMatrix(const SimpleMatrix<Type>& _rhs);

  // Assignment operator overloaded
  SimpleMatrix<Type>& operator= (const SimpleMatrix<Type>& _rhs);

  virtual ~SimpleMatrix();  // Destructor

  // Access to the matrix values directly, via row and column indices
  vector<vector<Type> > get_mat() const;
  Type& value(const int& row, const int& col);
};

#ENDIF
```

We won't discuss the pre-processor macros, since we've discussed them at length in previous chapters.

This is our first encounter with the vector component of the standard template library. The vector class is itself a template class. We will be using a vector as the underlying storage mechanism for our types. Thus we need to include the vector library:

```
#include <vector>
```

Recall the syntax for creating a template class with a default type:

```
template <typename Type = double> class SimpleMatrix { ... };
```

The following line requires some explanation. This code is telling the compiler that we wish to create a private data member element called mat, which is a vector. This vector requires an underlying type (since a vector is a template class). The underlying type for mat is a vector< Type>, i.e. a vector of our as-yet-unspecified Types. This means that mat is really a set of rows, storing columns of Types. Thus is it is a "vector of vectors" and allows us to store our matrix values:

```
private:
  vector<vector<Type> > mat;
```

The default constructor is unremarkable, but the parameter constructor requires a bit of discussion. We have three parameters, the first two of which are integer values and the latter is a Type. The two integer values tell the compiler how many rows and columns the matrix will have, while the third Type parameter tells the compiler the quantity to fill the matrix values with:

```
public:
  SimpleMatrix();   // Default constructor
  SimpleMatrix(const int& rows, const int& cols, const Type& val);
  ..
```

Notice that in the copy constructor we also have to specify the type in the <> delimiters, otherwise the compiler will not know the concrete object to accept as a parameter when copying the SimpleMatrix. The overloaded assignment operator returns a reference to a SimpleMatrix< Type>, as once again, the compiler must have a concrete type when implementing this code. In addition, it takes the same type (with **const**) as a parameter:

```
public:
  ..
  SimpleMatrix(const SimpleMatrix<Type>& _rhs);
  SimpleMatrix<Type>& operator= (const SimpleMatrix<Type>& _rhs);
  ..
```

We won't discuss the destructor, other than to mention that it is set to be **virtual**, which is a good practice that we have outlined in previous chapters.

The final set of public methods are get_mat() and value (...) . They allow direct access to both the matrix itself and the values at a particular location, specified by the row and column index, starting from 0 rather than 1.

get_mat() returns a vector<vector<Type> >, not a reference to this object! Hence mat will be copied whenever this method is called - a rather inefficient process if our matrix is large. We have implemented it this way only to highlight the syntax of creating a template class, not to show ideal matrix optimisation (which will come later on in the course).

value (...) returns a reference to Type as this allows direct access to the values within the matrix (for modification as well as access):

```
public:
  ..
  vector<vector<Type> > get_mat() const;
  Type& value(const int& row, const int& col);
```

At this stage our SimpleMatrix is not very exciting. We can create it, store values within it and then access/change those values via row/column indices. However, what it does have going for it is that it can handle multiple types! We could use **int**, **double** or complex types within our code, thus saving ourselves a lot of duplicated coding.

5.7 SimpleMatrix Implementation

Now that we've discussed the declaration of the template matrix class, we need to code up the implementation. Here is the listing for *simplematrix.cpp* in full:

```
#IFNDEF __SIMPLE_MATRIX_CPP
#DEFINE __SIMPLE_MATRIX_CPP

#include "simplematrix.h"

// Default constructor
template <typename Type>
SimpleMatrix<Type>::SimpleMatrix() {
  // No need for implementation, as the vector "mat"
  // will create the necessary storage
}

// Constructor with row/col specification and default values
template <typename Type>
SimpleMatrix<Type>::SimpleMatrix(const int& rows, const int& cols,
                                 const Type& val) {
  for (int i=0; i<rows; i++) {
    std::vector<Type> col_vec (cols, val);
    mat.push_back(col_vec);
  }
}

// Copy constructor
template <typename Type>
SimpleMatrix<Type>::SimpleMatrix(const SimpleMatrix<Type>& _rhs) {
  mat = _rhs.get_mat();
}

// Overloaded assignment operator
template <typename Type>
SimpleMatrix<Type>& SimpleMatrix<Type>::operator= (const SimpleMatrix<Type
    >& _rhs)
{
  if (this == &_rhs) return *this;  // Handling assignment to self
  mat = _rhs.get_mat();
  return *this;
}

// Destructor
template <typename Type>
SimpleMatrix<Type>::~SimpleMatrix() {
  // No need for implementation, as there is no
  // manual dynamic memory allocation
}

// Matrix access method, via copying
template <typename Type>
SimpleMatrix<Type> SimpleMatrix<Type>::get_mat() const {
  return mat;
}

// Matrix access method, via row and column index
template <typename Type>
Type& SimpleMatrix<Type>::value(const int& row, const int& col) {
  return mat[row][col];
}

#ENDIF
```

As with previous source files we are using pre-processor macros and have included the respective header file. We will begin by discussing the default constructor syntax:

```
// Default constructor
template <typename Type>
SimpleMatrix<Type>::SimpleMatrix() {
  // No need for implementation, as the vector "mat"
  // will create the necessary storage
}
```

Beyond the comment, the first line states to the compiler that we are defining a *function template* with a single typename, Type. The following line uses the scope resolution operator

(::) to specify the implementation of the default constructor. The key point to note here is that although the class itself is referred to as SimpleMatrix<Type>, the constructor does not include the type and is simply written as SimpleMatrix(). Because the scope of the class already includes the type it is not necessary to repeat it within the constructor signature itself.

The next method to be defined is the parameter constructor, necessary to specify the number of rows, columns and an initial value for all matrix cells:

```
// Constructor with row/col specification and default values
template <typename Type>
SimpleMatrix<Type>::SimpleMatrix(const int& rows, const int& cols,
                                 const Type& val) {
  for (int i=0; i<rows; i++) {
    std::vector<Type> col_vec (cols, val);
    mat.push_back(col_vec);
  }
}
```

As with the default constructor, we need to specify the class scope by including the type, but not in the constructor name itself. The constructor takes three parameters, which are respectively the number of rows, the number of columns and an initial value to fill all matrix cells with.

The implementation loops over the number of rows and adds a new vector of length cols, with value val, to each element of mat. Thus mat ends up as a *vector of vectors*.

Subsequent to the parameter constructor is the implementation of the copy constructor:

```
// Copy constructor
template <typename Type>
SimpleMatrix<Type>::SimpleMatrix(const SimpleMatrix<Type>& _rhs) {
  mat = _rhs.get_mat();
}
```

This is a straightforward implementation. It simple states that the new mat instance should be copied from the _rhs ("right hand side") reference SimpleMatrix object, via the get_mat() function. Recall that get_mat() is an expensive method to call, so in later lessons we will implement a more efficient matrix class.

After the copy constructor we implement the overloaded assignment operator:

```
// Overloaded assignment operator
template <typename Type>
```

```
SimpleMatrix<Type>& SimpleMatrix<Type>::operator=(const SimpleMatrix<Type>&
    _rhs)
{
  if (this == &_rhs) return *this;  // Handling assignment to self
  mat = _rhs.get_mat();
  return *this;
}
```

The assignment operator implementation is not too dissimilar to that of the copy constructor. It basically states that if we try and assign the object to itself (i.e. my_matrix = my_matrix;) then it should return a dereferenced pointer to **this**, the reference object to which the method is being called on. If a non-self assignment occurs, then copy the matrix as in the copy constructor and then return a dereferenced pointer to **this**.

The destructor does not have an implementation as there is no dynamic memory to deallocate:

```
// Destructor
template <typename Type>
SimpleMatrix<Type>::~SimpleMatrix() {
  // No need for implementation, as there is no
  // manual dynamic memory allocation
}
```

The last two public methods involve access to the underlying matrix data. get_mat() returns a copy of the matrix mat and, as stated above, is an inefficient method, but is still useful for describing template syntax. Notice also that it is a **const** method, since it does not modify anything:

```
// Matrix access method, via copying
template <typename Type>
SimpleMatrix<Type> SimpleMatrix<Type>::get_mat() const {
  return mat;
}
```

The final method, value (...) returns a direct reference to the underlying data type at a particular row/column index. Hence it is not marked as **const**, because it is feasible (and intended) that the data can be modified via this method. The method makes use of the [] operator, provided by the vector class in order to ease access to the underlying data:

```
// Matrix access method, via row and column index
template <typename Type>
```

```
Type& SimpleMatrix<Type>::value(const int& row, const int& col) {
  return mat[row][col];
}
```

This completes the listing for the SimpleMatrix object. We will extend our matrix class in later lessons to be far more efficient and useful.

5.8 Generic Programming vs OOP

One common question that arises when discussing templates is “Why use templates over normal object orientation?”. Answers include:

- Generally more errors caught at compile time and less at run-time. Thus, there isn’t as much need for try-catch exception blocks in your code.
- When using *container* classes with iteration. Modelling a Time Series or Binomial Lattice, for instance. Generic programming offers a lot of flexibility here.
- The STL itself is dependent upon generic programming and so it is necessary to be familiar with it.

However, templates themselves can lead to extremely cryptic compiler error messages, which can take a substantial amount of time to debug. More often than not, these errors are due to simple syntax errors. This is probably why generic programming has not been taken up as much as the object oriented paradigm.

Chapter 6

Introduction to the Standard Template Library

The Standard Template Library (STL) is a cross-platform standard of efficient data structures and algorithms. It is one of the most important features of C++ and will form the basis of all ongoing quantitative finance programs within this book. The STL is often confusing to novice programmers because it makes use of the *generic programming paradigm*, as opposed to the *object oriented paradigm*. In fact, it could be said that the STL is the opposite approach, in that rather than encapsulate the data and algorithms acting upon it, the STL actually separates these functions through a common *iterator* interface.

The STL is extremely important in quantitative finance because many of the quant data structures and algorithms rely on more elementary concepts such as sorting, maxima/minima, copying, mathematical function modelling and searching. Rather than reimplement these basic algorithms from scratch, C++ provides them "out of the box". This allows you to spend more time testing your own algorithms, rather then "reinventing the wheel".

The STL is a large library. It contains a wealth of data structures and algorithms. In this chapter I will attempt to give a broad overview of the STL and highlight those aspects which are relevant to quantitative finance.

6.1 Components of the STL

The STL can be broadly split into four separate components:

- **Containers** - Data structures designed to efficiently store information across a variety of generic types.
- **Iterators** - Methods by which to traverse/access the data within the containers.
- **Algorithms** - Basic algorithmic capabilities, such as sort, find, max, min, replace. Makes use of iterators to allow "container independent" code.
- **Function Objects** - a.k.a. *functors*. Used to make templates classes "callable" to allow modelling of mathematical functions.

There are extra smaller components, but these are the four that will interest us for quantitative finance work.

6.2 Containers

Containers allow certain types of data to be grouped into logical structures. The STL provides *generic containers* so that the same data structure can handle multiple different types without any additional required coding. For instance, we could create a `std::vector<double>` or a `std::vector<int>`. The former would be a vector of double precision values, while the latter is a vector of integer precision values. Notice that we are not required to code anything else to let the vector know we are using different types!

6.2.1 Sequence Containers

Sequence containers are extremely common within quantitative finance applications. They are often used to store numerical data sequentially. For instance we may want to store components of an element of a vector field or matrix of complex numbers. There are three main types of sequential containers that tend to be used the most:

- **Lists** - The two types of list are the *singly-linked list* and the *doubly-linked list*. A singly-linked list only has forward pointers to the next element, whereas a doubly-linked list has forward and backward pointers. They do not allow *random access* to elements, but do allow fast inserts and removals.
- **Vectors** - Vectors allow fast random access to elements, but are slow for inserts and removals within the vector (as all elements need to be copied and moved). They allow fast appends and "pops" (i.e. removals from the end).

- **Deques** - i.e. *double-ended queues*. Deques are similar to vectors except that they allow fast appends/removals to both ends of the queue.

There is also the valarray sequence container, which is (supposedly) optimised for numerical work. However, in practice valarray is harder to use and many modern compilers do not implement the optimisations well enough to warrant using them as an alternative to vectors.

6.2.2 Associative Containers

Associative containers are different to sequential containers in that there is no direct linear sequence to the elements within the container. Instead, they are *sorted* based on a comparison criterion. This criterion is often determined via the context. For instance, in a `std::set<int>`, the comparison operator is the *less than* (<) operator. There are four main types of associative container, which differ in terms of duplicate elements and whether the *key* data and *value* data coincide:

- **Set** - A set provides a container that does not allow duplicate elements and where the data is itself the *key* and *value*. Sets can be *sorted* according to comparison functions. Here is an example of a set of integers: $\{1, 2, 3, 4\}$.

- **Multiset** - A multiset is very similar to a set except that it can possess multiple duplicate values. Here is an example of a multiset of integers: $\{1, 2, 3, 3, 4\}$. Notice the duplicate 3s.

- **Map** - A map contains pairs of elements, the two components of which are known as the *key* and the *value*. The key is used to "look up" the value data. Maps do not allow duplicate keys, but they do provide a great deal of flexibility in what types of data can represent the keys and values. An example of a map would be a set of date keys, each of which possessed a price value, without duplicate dates.

- **Multimap** - A multimap is similar to a map except that duplicate keys are allowed. This is possible because the values can be iterated over in a sorted fashion, using the aforementioned comparison function.

Although it might initially seem that associative containers are less appropriate for handling numerical data within quantitative finance, this is not actually the case. A map is ideal for holding the values of sparse matrices (such as tridiagonal or banded), which crop up in finite difference methods, for instance.

6.2.3 Container Adaptors

Although not strictly containers in themselves *container adaptors* are a very useful component of the STL. Container adaptors are designed to adapt the behaviour of the aforementioned containers above such that their behaviour is restricted or modified in some fashion. The three container adaptors that are of interest to us are as follows:

- **Stacks** - A *stack* is a common data structure within computer science as well in day-to-day life. It represents the Last-In-First-Out (LIFO) concept, where elements are placed upon each other and the first one to be removed was the last one to be added.
- **Queues** -A *queue* differs from a stack in that it represents the First-In-First-Out (FIFO) concept, where the first element added to the queue becomes the first to be removed. Queues are often used to represent task lists of "background jobs" that might need to be carried out, such as heavy-duty numerical processing.
- **Priority Queues** -A *priority queue* is similar to a queue, except that elements are removed from it based on a priority comparison function. This is particularly common in operating system design where many tasks are constantly being added to a queue, but some are more essential than others and so will be executed first, even if they were added at a later point.

6.3 Iterators

Now that we've discussed STL Containers, it is time to turn our attention to **iterators**. Iterators are objects that can navigate or *iterate* over elements in a container. They are essentially a generalisation of pointers and provide similar, but more advanced, behaviour. Their main benefit is that they allow the decoupling of the implementation of a container with an algorithm that can iterate over it.

Iterators are often tricky for beginning C++ programmers to get to grips with as there are many different *categories* and *adaptors* that can be applied to modify their behaviour. We're are going to describe the taxonomy that exists to help you choose the correct iterator type for your quantitative finance algorithm implementations.

6.3.1 Iterator Categories

Iterators are grouped according to their *category*. There are five separate iterators categories in C++: Input, Output, Forward, Bidirectional and Random Access:

- **Input** - An input iterator is a *single-pass read-only* iterator. In order to traverse elements, the ++ increment operator is used. However, elements can only be read once. Input iterators do not support the $--$ decrement operator, which is why they are termed single-pass. Crucially, an input iterator is unable to modify elements. To access an element, the $*$ pointer dereference operator is utilised.

- **Output** - An output iterator is very similar to an input iterator with the major exception that the iterator writes values instead of reading them. As with input iterators, writes can only occur once and there is no means of stepping backwards. Also, it is only possible to assign a value once to any individual element.

- **Forward** - A forward iterator is a combination of an input and output iterator. As before the increment operator (++) allows forward stepping, but there is no decrement operator support. To read or write to an element the pointer derefernce operator ($*$) must be used. However, unlike the previous two iterator categories, an element can be read or written to multiple times.

- **Bidirectional** - A bidirectional iterator is similar to a forward iterator except that it supports backward stepping via the decrement operator ($--$).

- **Random Access** - A random access iterator is the most versatile iterator category and is very much like a traditional C-style pointer. It possesses all the abilities of a bidirectional iterator, with the addition of being able to access any index via the [] subscript operator. Random access iterators also support *pointer arithmetic* so integer stepping is allowed.

Iterators are divided into these categories mainly for performance reasons. Certain iterators are not supported with certain containers when C++ deems it likely that iteration will lead to poor performance. For instance, random access iteration is not supported on the std::list as access to a random element would potentially require traversal of the entire linked-list used to store the data. This is in contrast to a std::vector where *pointer arithmetic* can be used to step directly to the desired item.

6.3.2 Iterator Adaptors

Iterator adaptors allow iterators to be modified to allow special functionality. In particular, iterators can be reversed, they can be modified to insert rather than overwrite and can be adapted to work with streams.

Reverse Iterators

Reverse iterator adaptors essentially ”swap” the increment (++) and decrement (−−) operators so that any algorithms making use of these iterators will be carried out in reverse. Every STL container allows reverse iteration. It is possible to convert standard iterators to reverse iterators, although any converted iterator must support both increment and decrement (i.e. it must be bidirectional).

Insertion Iterators

Insertion iterators adapt normal output iterators to perform insertion instead of overwriting. They only allow writing, as with normal output iterators. There are three separate insertion iterators in the STL: **back**, **front** and **general**. They all behave differently with regards to the position of the inserted element:

- **Back inserters** utilise the push_back method of the container it is acting upon and so will append values at the end. Hence they only work on a subset of containers that support the method (strings, lists, deques and vectors).
- **Front inserters** utilise the push_front method of the container and is only available for two containers in the STL - the deque and the list.
- **General inserters** utilise the insert method of the container being acted upon, which is supported by all STL containers.

Stream Iterators

Stream iterators are adapted iterators that permit reading and writing to and from input and output streams. **istream** iterators are used to read elements from an input stream, making use of the >> operator. **ostream** iterators are the writing counterpart and can write to an output stream via the << operator.

6.3.3 Const Iterators

Containers allow both iterator and const_iterator types. const_iterators are returned by certain container methods (such as begin or end) when the container itself is **const**.

One must be careful to distinguish between const_iterator and **const** iterator (notice the lack of underscore!). The former is a non-const object with iterator type, so that it returns objects which cannot be modified. The latter is a constant iterator (i.e. it cannot be incremented!). It is possible to convert an iterator to a const_iterator, but going the other way is not possible.

const_iterators are preferred (if possible) because they aid readability. They let programmers know that the iterator is only iterating over the container and not modifying it.

6.3.4 Iterator Traits

A common requirement is to be able to use pointers in place of iterators when carrying out an algorithm. Since they have nearly identical behaviour, it makes sense to allow this. The way this is achieved is through *iterator traits*. The iterators_traits class template provided by the STL is designed to allow a common interface for any type of iterator.

When creating algorithms that iterate over a container it is possible to make use of the *iterator traits* to obtain the underlying type being referred to by the presented iterator (or pointer), without the need to know whether an iterator or pointer is actually being presented. This makes the code more maintainable and versatile.

6.4 Algorithms

We're now ready to discuss STL algorithms, which are operations that act on the containers via the aforementioned STL Iterators. STL algorithms are extremely useful because they reduce or eliminate the need to 'reinvent the wheel' when implementing common algorithmic functionality. In quantitative finance coding they can save a significant amount of time.

While some of the algorithms are designed to modify the individual values within *ranges* (certain STL containers or arrays of values), they do not modify the containers themselves - i.e. there is no reallocation or size modification to the containers.

6.4.1 Algorithm Categories

To use these algorithms it is necessary to include the <algorithm> header file. For the numeric algorithms, it is necessary to include the <numeric> header file. As with containers and iterators, algorithms are categorised according to their behaviour and application:

- **Nonmodifying algorithms** - Nonmodifying algorithms do not change the value of any element, nor do they modify the order in which the elements appear. They can be called for all of the standard STL container objects, as they make use of the simpler forward iterators.

- **Modifying algorithms -** Modifying algorithms are designed to alter the value of elements within a container. This can either be done on the container directly or via a copy into another container range. Some algorithms classed as 'modifying' algorithms can change the order of elements (in particular copy_backward()), but most do not.

- **Removal algorithms -** Removal algorithms are, by definition, modifying algorithms, but they are designed to remove elements in a container or when copying into another container.

- **Mutating algorithms -** Once again, mutating algorithms are modifying algorithms, but they are designed specifically to modify the order of elements (e.g. a random shuffle or rotation).

- **Sorting algorithms -** Sorting algorithms are modifying algorithms specifically designed for efficient sorting of elements in a container (or into a range container).

- **Sorted range algorithms -** Sorted range algorithms are special sorting algorithms designed to function on a container which is already sorted according to a particular sorting criterion. This allows for greater efficiency.

- **Numeric algorithms -** Numeric algorithms are designed to work on numerical data in some fashion. The principal algorithm in this category is accumulate(), which allows mathematical operators to be applied to all elements in a container.

We'll now take a look at these categories in depth. If you would like to find out more about each subsequent algorithm, you can click on the name of any algorithm below.

6.4.2 Nonmodifying Algorithms

- for_each() - This is possibly the most important algorithm in this section, as it allows any unary function (i.e. a function of one argument) to be applied to each element in a range/container. Note that this function can actually also be modifying (hence why it is included below). It is often better to use a more specific algorithm, if one exists, than to use this, as specialist implementations will be more efficient.

- count() - This returns the number of elements in a range or container.

- count_if() - This counts how many elements in a range or container much a particular criterion.

- min_element() - Returns the element that has the smallest value, making use of the ¡ relation to perform comparison. It can accept a custom binary function to perform the comparison instead.
- max_element() - Returns the element that has the largest value, making use of the ¿ relation to perform comparison. It can accept a custom binary function to perform the comparison instead.
- find() - Finds the first element in a range or container that equals a passed value.
- find_if () - Finds the first element in a range or container that matches a particular criterion, rather than a passed value.
- search_n() - This is like find AND find_if except that it looks for the first N occurances of such a value OR the first N occurances where a relational predicate is met.
- search() - This searches for the first occurance of a subrange within a range/container and can do so either by first/last value of the subrange or via a predicate matching all the values of the desired first/last subrange.
- find_end() - Similar to search, except that it finds the last occurance of such a subrange.
- find_first_of () - Finds the first element in a subrange of a range or container. Can make use of a binary predicate function, otherwise uses direct value.
- adjacent_find() - Returns the location (an iterator) to the first matching consecutive pair of values in a range/container. Can also match via a binary predicate.
- equal() - Compares two ranges to see if they are equal.
- mismatch() - Compares two ranges and returns a pair of iterators containing the points at which the ranges differ.
- lexicographical_compare() - The lexicographical comparison is used to sort elements in a manner similar to how words are ordered in a dictionary. It can either use **operator**< or make use of a binary predicate function to perform the comparison.

6.4.3 Modifying Algorithms

- for_each() - This is the same as the for_each we discussed above, but I have included it in the Modifying section to reinforce that it can be used this way too!

- copy() - Copies a range/container of elements into another range.
- copy_backward() - Copy a range/container of elements into another range, starting from the last element and working backwards.
- transform() - Transform is quite a flexible algorithm. It works in two ways. A unary operation can be applied to the source range, on a per element basis, which ouputs the results in the destination range. A binary operation can be applied to both elements in the source and destination range, subsequently overwriting elements in the destination range.
- merge() - Merge is intended to take two sorted ranges and combine them to produce a merged sorted range. However, it is possible to utilise unsorted ranges as arguments but this then leads to an unsorted merge! For this reason I've decided to include it in the modifying category, rather than the category for sorted ranges, below.
- swap_ranges() - This swaps the elements of two ranges.
- fill () - This replaces each element in a range with a specific value.
- fill_n () - Similar to fill , but replaces the first N elements in a range with a specific value.
- generate() - This replaces each element in a range with the result of an operation of a generator function.
- generate_n() - Similar to generate, but replaces the first N elements in a range with the result of an operation of a generator function.
- replace() - This replaces elements matching a specific value with another specific value.
- replace_if () - This replaces elements matching a specific criterion with another specific value.
- replace_copy() - Similar to replace, except that the result is copied into another range.
- replace_copy_if () - Similar to replace_if , except that the result is copied into another range.

6.4.4 Removal Algorithms

- remove() - This removes elements from a range that match a specific value.
- remove_if() - This removes elements from a range that match a specific criterion, as determined via a unary predicate.

- remove_copy() - Similar to remove, except that elements are copied into another range.
- remove_copy_if() - Similar to remove_if, except that elements are copied into another range.
- unique() - This is quite a useful algorithm. It removes adjacent duplicate elements, i.e. consecutive elements with specific values.
- unique_copy() - Similar to unique, except that it copies the elements into another range.

6.4.5 Mutating Algorithms

- reverse() - This simply reverses the order of the elements in a range or container.
- reverse_copy() - Similar to reverse, except that the results of the reversal are copied into another range.
- rotate() - By choosing a 'middle' element in a range, this algorithm will *cyclically rotate* the elements such that the middle element becomes the first.
- rotate_copy() - Similar to rotate, except that the result is copied into another range.
- next_permutation() - This rearranges the elements in a range to produce the next lexicographically higher permutation, using **operator**<. It is also possible to use a binary predicate comparison function instead of **operator**<.
- prev_permutation() - Similar to next_permutation, except that it rearranges to produce the next lexicographically lower permutation.
- random_shuffle() - Rearranges the list of elements in a range in a random fashion. The source of randomness can be supplied as a random number generator argument.
- partition() - Rearranges a range/container such that the elements matching a predicate are at the front. Does NOT guarantee relative ordering from the original range.
- stable_partition() - Similar to partition, except that it does guarantee relative ordering from the original range.

6.4.6 Sorting Algorithms

- sort() - Sorts the elements into ascending order, using **operator**< or another supplied comparison function.

- stable_sort () - This is similar to sort. It is used when you need to ensure that elements remain in the same order when they are "tied" for the same position. This often comes up when dealing with priorities of tasks. Note also that the performance guarantee is different.

- partial_sort () - Similar to sort, except that it only sorts the first N elements and terminates after they're sorted.

- partial_sort_copy () - Similar to partial_sort except that it copies the results into a new range.

- nth_element() - This allows you to ensure that an element at position n is in the correct position, were the rest of the list to be sorted. It only guarantees that the elements preceeding n are less than in value (in the sense of **operator**<) and that proceeding elements are greater than in value.

- partition () - See above for partition.

- stable_partition () - See above for stable_partition.

- make_heap() - Rearranges the elements in a range such that they form a *heap*, i.e. allowing fast retrieval of elements of the highest value and fast insertion of new elements.

- push_heap() - This adds an element to a heap.

- pop_heap() - This removes an element from a heap.

- sort_heap() - This sorts the elements in a heap, with the caveat that the range is no longer a heap subsequent to the function call.

6.4.7 Sorted Range Algorithms

- binary_search() - Searches the range for any matches of a specific value.

- includes() - Determines whether each element in one range is also an element in another range.

- lower_bound() - Searches for the first element in a range which does not compare less than a specific value. Can also use a custom comparison function.

- upper_bound() - Searches for the first element in a range which does not compare greater than a specific value. Can also use a custom comparison function.

- equal_range() - Finds a subrange within a range which contains values equal to a specific value.
- merge() - See above for merge.
- set_union() - Creates a set union of two sorted ranges. Thus the destination range will include elements from either or both of the source ranges. Since this is a set operation, duplicates are eliminated.
- set_intersection () - Creates a set intersection of two sorted ranges. Thus the destination range will include elements that exist only in both of the source ranges.
- set_difference () - Creates a set difference of two sorted ranges. Thus the destation range will include elements from the first range that are not in the second range.
- set_symmetric_difference() - This is the symmetric version of set_difference . It creates a set symmetric difference, which is formed by the elements that are found in one of the sets, but not in the other.
- inplace_merge() - This combines two sorted ranges into a destination range, which is also sorted. It is also *stable* as it will preserve ordering for subranges.

6.4.8 Numeric Algorithms

- accumulate() - This is an extremely useful algorithm. It allows all elements of a container to be combined by a particular operation, such as via a sum, product etc.
- inner_product() - This combines all elements of two ranges by summing the multiples of each 'component'. It is possible to override both the 'sum' and 'multiply' binary operations.
- adjacent_difference () - Assigns every element the value of the difference between the current value and the prior value, except in the first instance where the current value is simply used.
- partial_sum() - Assigns the result of the partial sum of the current element and its predecessors, i.e. $y_i = \sum_{k=1}^{i} x_i$, for y the result and x the source.

Not only do these algorithms save us valuable development time, but we are able to ascertain their performance characteristics from the C++ Reference on Algorithms. Further, they have all been highly optimised via their respective compilers and so our programs will likely run significantly faster than if we construct our own.

6.5 C++11 STL

This section introduces, in a reference fashion, some of the new algorithms and containers that are now part of the C++11 STL. Highlights include a hash object (unordered_map), a singly-linked list (forward_list) and many algorithms which attempt to "fill in the missing pieces" left by C++03, which we have described above.

6.5.1 Containers

The new containers have essentially been introduced for performance reasons. C++ now has a hash table, a singly-linked list and an array object as part of the STL standard.

- unordered_set - As with a set, the unordered_set contains at most one of each value and allow fast retrieveal of elements. Allows forward iterators.
- unordered_multiset - As with a multiset, the unordered_multiset can contain multiple copies of the same value and allows fast retrieval of elements. Allows forward iterators.
- unordered_map - A hash table! As with a map, the unordered_map can contain as most one of each key with fast retrieval. Allows forward iterators.
- unordered_multimap - As with multimap, the unordered_multimap can contain multiple copes of the same key. Allows forward iterators.
- forward_list - This is a singly-linked list. It provides constant time inserts and erases, but no random access to elements.
- array - C++11 now has an STL array. It stores n elements of type T contiguously in memory. It differs from a vector as it cannot be resized once created.

6.5.2 Algorithms

C++11 has introduced some new algorithms to "fill in the blanks" left by the previous C++03 standard. Other algorithms exist to simplify common programming tasks, either by using a simpler interace or better parameter lists. As of 2013, compiler support for these algorithms is relatively good, but you should consult the documentation of your compiler's STL implementation.

- all_of - This returns true if all the values in the range satisfy a predicate, or the range is empty.

- any_of - This returns true if any of the values in the range satisfy a predicate, or the range is empty.
- none_of - This returns true if none of the values in the range satisfy a predicate, or the range is empty.
- find_if_not - This returns an iterator to the first value that causes a predicate to be false (similar to partition_point). This uses a linear search mechanism.
- copy_if - This copies all elements that satisfy a predicate into another range.
- copy_n - This copies n elements from a range into another range.
- unitialized_copy_n - Similar to unitialized_copy, except that it works for n elements.
- move - This moves elements from one range into another.
- move_backward - This moves elements from one range into another, reversing the order of the move.
- is_partitioned - This returns true if all of the elements in a range that satisfy a predicate are before all of those that do not. It also returns true if the range is empty.
- partition_copy - This copies elements from a source range into two separate destination ranges based on whether the elements satisfy a predicate or not.
- partition_point - This returns an iterator to the first value that causes a predicate to be false (similar to find_if_not). This uses a binary search mechanism.
- partial_sort_copy - This copies all sorted elements from a source to a range. The number of elements copied is determined by the smaller of the sorted source range and the result range. Can also use an optional sorting comparison operator.
- is_sorted - This returns true if the range is sorted. Can also use an optional sorting comparison operator.
- is_sorted_until - This returns an iterator to the last position for which the range is sorted. Can also use an optional sorting comparison operator.
- is_heap - This returns true if the range is a heap, i.e. the first element is the largest. Can also use an optional sorting comparison operator.
- is_heap_until - This returns an iterator to the last position for which the range is a heap. Can also use an optional sorting comparison operator.

- min - Finds the smallest value in the parameter list. Can also use an optional sorting comparison operator.
-
- max - Finds the largest value in the parameter list. Can also use an optional sorting comparison operator.
- minmax - This returns a pair of the smallest and largest elements. Can also use an optional sorting comparison operator.
- minmax_element - This returns two iterators, one pointing to the smallest element in the range and the other pointing to the largest element in the range. Can also use an optional sorting comparison operator.
- iota - This creates a range of sequentially increasing values, making use of the pre-increment operator (++i) to create the sequence.

This concludes our tour of the C++ Standard Template Library. In the subsequent chapters we will make heavy use of the STL to carry out quantitative finance work.

Chapter 7

Function Objects

Many of the concepts within quantitative finance are represented by functions or groups of functions. The types of these functions can vary considerably. Not all of the functions we consider are real-valued, for instance. Many functions take other functions as argument or are vector-valued.

Our goal in this chapter is to develop some intuition about basic mathematical functions and attempt to model their behaviour in C++. Because C++ is a "federation of languages" and supports multi-paradigm programming, there are many options available to us. There is no "right answer" as to how to model functions in C++, rather there exist different methods, some of which are more optimal in certain situations.

There are plenty of examples of functions from within mathematics, and quantitative finance in particular. Some concrete examples from quantitative finance include:

- **Pay-off functions** - These take a real-valued asset spot and strike price, generally, to provide a real-valued option value at expiry.

- **Differential Equation Coefficients** - ODEs and PDEs possess coefficients dependent upon various parameters, which can potentially be real-valued functions.

- **Matrices** - Linear Algebra tells us that matrices are in fact linear maps, i.e. functions between vector spaces.

We will strike a balance between re-use, efficiency and maintainability when modelling functions in C++, so as to improve productivity without creating complicated code. C++ presents us with a number of alternatives for modelling functions. In particular:

- **Function Pointers** - These are a feature of the C language and so form part of the C++ standard. A function pointer allows a pointer to a function to be passed as a parameter to another function.
- **Function Objects (Functors)** - C++ allows the function call **operator**() to be overloaded, such that an object instantiated from a class can be "called" like a function.
- **STL Functions** - The Standard Template Library (STL) provides three types of *template function objects*: Generator, unary and binary functions.
- **C++11** <function> - C++11 brought new changes to how functors were handled. In addition, anonymous functions (lambdas) are now supported.

7.1 Function Pointers

Function pointers are a legacy feature from the C language. C++ is a superset of C and so includes function pointer syntax. In essence, function pointers point to executable code at a particular piece of memory, rather than a data value as with other pointers. Dereferencing the function pointer allows the code in the memory block to be executed. In addition, arguments can be passed to the function pointer, which are then passed to the executed code block. The main benefit of function pointers is that they provide a straightforward mechanism for choosing a function to execute at run-time.

Here is a C++ example which makes use of an add function and a multiply function to sum and multiply two double values:

```cpp
#include <iostream>

double add(double left, double right) {
    return left + right;
}

double multiply(double left, double right) {
    return left * right;
}

double binary_op(double left, double right, double (*f)(double, double)) {
    return (*f)(left, right);
}
```

```
int main( ) {
    double a = 5.0;
    double b = 10.0;

    std::cout << "Add: " << binary_op(a, b, add) << std::endl;
    std::cout << "Multiply: " << binary_op(a, b, multiply) << std::endl;

    return 0;
}
```

The output from this simple example is as follows:

```
Add: 15
Multiply: 50
```

In order for a function to receive a function pointer as a parameter it is necessary to specify its return type (in this case **double**), the parameter name of the function (in this case f) and the types of all parameters necessary for the function pointer (in this case two **double** values). We are then able to pass the function pointers into the binary_op function. binary_op then dereferences the function pointer in order to execute the correct function - add or multiply.

While function pointers are simple to use in your code, they do suffer from some significant drawbacks:

- **Efficiency** - Function pointers are inefficient when compared with *functors* (discussed below). The compiler will often pass them as raw pointers and as such the compiler will struggle to inline the code.

- **State** - Function pointers by themselves are not particularly flexible at storing *state*. Although it is possible, by using a local **static** variable within the function, there is only ever one global state for the function itself and as such this static variable must be shared. Furthermore this static variable will not be *thread-safe*, unless the appropriate thread synchronisation code is added. Thus it can lead to bottlenecks or even race conditions in multithreaded programs.

- **Templates** - Function pointers do not play too well with templates if there are multiple signatures of the function in your code. The solution is to use function pointer casting, which leads to difficult and ungainly syntax.

- **Adaptation -** Function pointers have fixed parameter types and quantities. Thus they are not particularly flexible when external functions with differing parameter types could be used. Although adapting the function pointers (by wrapping the external functions with hard-coded parameters) is possible, it leads to poor flexibility and bloated code.

The solution to these problems is to make use of the C++ *function object* (also known as a *functor*).

7.2 C++ Function Objects (Functors)

A function object allows an instance object of a class to be called or invoked as if it were an ordinary function. In C++ this is carried out by overloading **operator**(). The main benefit of using function objects is that *they are objects* and hence can contain state, either statically across all instances of the function objects or individually on a particular instance.

Here is a C++ example of a function object (in fact a function object hierarchy), which replaces the function pointer syntax from the version above, with functors:

```
#include <iostream>

// Abstract base class
class BinaryFunction {
public:
  BinaryFunction() {};
  virtual double operator() (double left, double right) = 0;
};

// Add two doubles
class Add : public BinaryFunction {
public:
  Add() {};
  virtual double operator() (double left, double right) { return left+right
      ; }
};

// Multiply two doubles
class Multiply : public BinaryFunction {
public:
  Multiply() {};
  virtual double operator() (double left, double right) { return left*right
      ; }
};

double binary_op(double left, double right, BinaryFunction* bin_func) {
  return (*bin_func)(left, right);
}

int main( ) {
  double a = 5.0;
  double b = 10.0;

  BinaryFunction* pAdd = new Add();
  BinaryFunction* pMultiply = new Multiply();

  std::cout << "Add: " << binary_op(a, b, pAdd) << std::endl;
  std::cout << "Multiply: " << binary_op(a, b, pMultiply) << std::endl;

  delete pAdd;
  delete pMultiply;

  return 0;
}
```

Firstly, note that there is a lot more happening in the code! We have created an *abstract base class*, called BinaryFunction and then inherited Add and Multiply classes. Since BinaryFunction is abstract, it cannot be instantiated. Hence we need to make use of pointers to pass in pAdd and pMultiply to the new binary_op function.

An obvious question is "What do we gain from all this extra code/syntax?". The main benefit is that we are now able to add *state* to the function objects. For Add and Multiply this is likely be to unnecessary. However, if our inheritance hierarchy were modelling connections to a database, then we might require information about how many connections currently exist (as we wouldn't want to open too many for performance reasons). We might also want to add extra database types. An inheritance hierarchy like this allows us to easily create more database connection classes without modifying any other code.

Chapter 8

Matrix Classes for Quantitative Finance

In order to do any serious work in quantitative finance it is necessary to be familiar with linear algebra. It is used extensively in statistical analysis and finite difference methods and thus plays a large role in quant finance. From your undergraduate mathematics days you will recall that (finite-dimensional) linear maps can be represented by matrices. Hence any computational version of such methods requires an implementation of a versatile matrix object. This chapter will discuss how to implement a *matrix class* that can be used for further quantitative finance techniques.

The first stage in the implementation of such a matrix class is to decide on a *specification.* We need to decide which mathematical operations we wish to include and how the interface to such operations should be implemented. Here is a list of factors we need to consider when creating a matrix class:

- The **type(s)** which will represent the underlying numerical values
- The **STL container(s)** that will be used to actually store the values
- The **mathematical operations** that will be available such as matrix addition, matrix multiplication, taking the transpose or elemental access
- How the matrix will interact with other objects, such as **vectors** and **scalars**

8.1 Custom Matrix Class Library

8.1.1 C++ STL Storage Mechanisms

C++ provides many container classes via the Standard Template Library (STL). The most appropriate choices here are std :: valarray and std :: vector. The std :: vector template class is more frequently used due to its generality. std :: vector can handle many different types (including pointers and smart pointer objects), whereas std :: valarray is designed solely for numerical values. In theory this would allow the compiler to make certain optimisations for such numerical work.

At first glance std :: valarray would seem like a great choice to provide storage for our matrix values. However, in reality it turns out that compiler support for the numerical optimisations that std :: valarray is supposed to provide does not really exist. Not only that but the std :: valarray has a poorer *application programming interface* (API) and isn't as flexible as a std :: vector. Thus it makes sense to use the std :: vector template class for our underlying storage mechanism.

Supposing that our matrix has M rows and N columns, we could either create a single std :: vector of length $N \times M$ or create a "vector of vector". The latter creates a single vector of length M, which takes a std :: vector<T> of types as its type. The inner vectors will each be of length N. We will utilise the "vector of vector" approach. The primary reason to use such a mechanism is that we gain a good API, helpful in accessing elements of such a vector "for free". We do not need to use a look-up formula to find the correct element as we would need to do in the single large vector approach. The declaration for this type of storage mechanism is given by:

```
std::vector<std::vector<T> >
```

Where T is our type placeholder. *Note: For nearly all of the quantitative work we carry out, we will use the* **double** *precision type for numerical storage.*

Note the extra space between the last two delimiters: > >. This is to stop the compiler into believing we are trying to access the bit shift >> operator.

The interface to the matrix class will be such that we could reconfigure the underlying storage mechanism to be more efficient for a particular use case, but the external client code would remain identical and would never need know of any such changes. Thus we could entirely replace our std :: vector storage with std :: valarray storage and our client calling code would be none the wiser. This is one of the benefits of the object oriented approach, and in particular *encapsulation*.

8.1.2 Matrix Mathematical Operations

A flexible matrix class should support a wide variety of mathematical operations in order to reduce the need for the client to write excess code. In particular, the basic binary operators should be supported for various matrix interactions. We would like to be able to add, subtract and multiply matrices, take their transpose, multiply a matrix and vector, as well as add, subtract, multiply or divide all elements by a scalar value. We will need to access elements individually by their row and column index.

We will also support an additional operation, which creates a vector of the diagonal elements of the matrix. This last method is useful within *numerical linear algebra*. We could also add the ability to calculate a determinant or an inverse (although there are good performance-based reasons *not* to do this directly). However, at this stage we won't support the latter two operations.

Here is a partial listing for the declaration of the matrix operations we will support:

```
// Matrix mathematical operations
QSMatrix<T> operator+(const QSMatrix<T>& rhs);
QSMatrix<T>& operator+=(const QSMatrix<T>& rhs);
QSMatrix<T> operator-(const QSMatrix<T>& rhs);
QSMatrix<T>& operator-=(const QSMatrix<T>& rhs);
QSMatrix<T> operator*(const QSMatrix<T>& rhs);
QSMatrix<T>& operator*=(const QSMatrix<T>& rhs);
QSMatrix<T> transpose();
```

We are making use of the object-oriented *operator overloading* technique. Essentially we are redefining how the individual mathematical operators (such as +, -, *) will behave when we apply them to other matrices as a *binary operator*. Let's consider the syntax for the addition operator:

```
QSMatrix<T> operator+(const QSMatrix<T>& rhs);
```

The function is returning a QSMatrix<T>. Note that we're not returning a reference to this object, we're directly returning the object itself. Thus a new matrix is being allocated when this method is called. The method requires a *const reference* to another matrix, which forms the "right hand side" (rhs) of the binary operation. It is a *const reference* because we don't want the matrix to be modified when the operation is carried out (the const part) and is passed by reference (as opposed to value) as we do not want to generate an expensive copy of this matrix when the function is called.

This operator will allow us to produce code such as the following:

```
// Create and initialise two square matrices with N = M = 10
// using element values 1.0 and 2.0, respectively
QSMatrix<double> mat1(10, 10, 1.0);
QSMatrix<double> mat2(10, 10, 2.0);

// Create a new matrix and set it equal to the sum
// of the first two matrices
QSMatrix<double> mat3 = mat1 + mat2;
```

The second type of mathematical operator we will study is the *operation assignment* variant. This is similar to the binary operator but will assign the result of the operation to the object calling it. Hence instead of creating a separate matrix C in the equation $C = A + B$, we will assign the result of $A + B$ into A. Here is the declaration syntax for such an operator overload:

```
QSMatrix<T>& operator+=(const QSMatrix<T>& rhs);
```

The major difference between this and the standard binary addition operator is that we are now returning a reference to a matrix object, not the object itself by value. This is because we need to return the original matrix that will hold the final result, not a copy of it. The method is implemented slightly differently, which will be outlined below in the source implementation.

The remaining matrix mathematical operators are similar to binary addition. We are going to support addition, subtraction and multiplication. We will leave out division as dividing one matrix by another is ambiguous and ill-defined. The transpose() method simply returns a new result matrix which holds the transpose of the original matrix, so we won't dwell too much on it here.

We also wish to support scalar addition, subtraction, multiplication and division. This means we will apply a scalar operation to each element of the matrix. The method declarations are given below:

```
// Matrix/scalar operations
QSMatrix<T> operator+(const T& rhs);
QSMatrix<T> operator-(const T& rhs);
QSMatrix<T> operator*(const T& rhs);
QSMatrix<T> operator/(const T& rhs);
```

The difference between these and those provided for the equivalent matrix operations is that the right hand side element is now a *type*, not a matrix of types. C++ allows us to reuse overloaded operators with the same method *name* but with a differing method *signature*. Thus we can use the same operator for multiple contexts, when such contexts are not ambiguous. We will see how these methods are implemented once we create the source file.

We also wish to support matrix/vector multiplication. The method signature declaration is given below:

```
std::vector<T> operator*(const std::vector<T>& rhs);
```

The method returns a vector of types and takes a const reference vector of types on the right hand side. This exactly mirrors the mathematical operation, which applies a matrix to a vector (right multiplication) and produces a vector as output.

The final aspect of the header file that requires discussion is the access to individual elements via the **operator**(). Usually this operator is used to turn the object into a *functor* (i.e. a function object). However, in this instance we are going to overload it to represent operator access. It is one of the only operators in C++ that can be used with multiple parameters. The parameters in question are the row and column indices (zero-based, i.e. running from 0 to $N-1$).

We have created two separate overloads of this method. The latter is similar to the first except we have added the **const** keyword in two places. The first **const** states that we are returning a constant type which should not be modified. The second **const** states that the method itself will not modify any values. This is necessary if we wish to have *read-only* access to the elements of the matrix. It prevents other **const** methods from throwing an error when obtaining individual element access.

The two methods are given below:

```
// Access the individual elements
T& operator()(const unsigned& row, const unsigned& col);
const T& operator()(const unsigned& row, const unsigned& col) const;
```

8.1.3 Full Declaration

For completeness, the full listing of the matrix header file is given below:

```
#ifndef __QS_MATRIX_H
#define __QS_MATRIX_H

#include <vector>

template <typename T> class QSMatrix {
 private:
  std::vector<std::vector<T> > mat;
  unsigned rows;
  unsigned cols;

 public:
  QSMatrix(unsigned _rows, unsigned _cols, const T& _initial);
  QSMatrix(const QSMatrix<T>& rhs);
  virtual ~QSMatrix();

  // Operator overloading, for "standard" mathematical matrix operations
  QSMatrix<T>& operator=(const QSMatrix<T>& rhs);

  // Matrix mathematical operations
  QSMatrix<T> operator+(const QSMatrix<T>& rhs);
  QSMatrix<T>& operator+=(const QSMatrix<T>& rhs);
  QSMatrix<T> operator-(const QSMatrix<T>& rhs);
  QSMatrix<T>& operator-=(const QSMatrix<T>& rhs);
  QSMatrix<T> operator*(const QSMatrix<T>& rhs);
  QSMatrix<T>& operator*=(const QSMatrix<T>& rhs);
  QSMatrix<T> transpose();

  // Matrix/scalar operations
  QSMatrix<T> operator+(const T& rhs);
  QSMatrix<T> operator-(const T& rhs);
  QSMatrix<T> operator*(const T& rhs);
  QSMatrix<T> operator/(const T& rhs);

  // Matrix/vector operations
  std::vector<T> operator*(const std::vector<T>& rhs);
  std::vector<T> diag_vec();

  // Access the individual elements
  T& operator()(const unsigned& row, const unsigned& col);
  const T& operator()(const unsigned& row, const unsigned& col) const;

  // Access the row and column sizes
  unsigned get_rows() const;
  unsigned get_cols() const;

};

#include "matrix.cpp"

#endif
```

You may have noticed that the matrix source file has been included before the final preprocessor directive:

```
#include "matrix.cpp"
```

This is actually a necessity when working with template classes. The compiler needs to see both the declaration and the implementation within the same file. This occurs here as the compiler replaces the include statement with the source file text directly in the preprocessor step, prior to any usage in an external client code. If this is not carried out the compiler will throw obscure-looking linker errors, which can be tricky to debug. *Make sure you always include your source file at the bottom of your declaration file if you make use of template classes.*

8.1.4 The Source File

Our task with the source file is to implement all of the methods outlined in the header file. In particular we need to implement methods for the following:

- Constructors (parameter and copy), destructor and assignment operator
- Matrix mathematical methods: Addition, subtraction, multiplication and the transpose
- Matrix/scalar element-wise mathematical methods: Addition, substraction, multiplication and division
- Matrix/vector multiplication methods
- Element-wise access (const and non-const)

We will begin with the construction, assignment and destruction of the class.

8.1.5 Allocation and Deallocation

The first method to implement is the constructor, with paramaters. The constructor takes three arguments - the number of rows, the number of columns and an initial type value to populate the matrix with. Since the "vector of vectors" constructor has already been called at this stage, we need to call its resize method in order to have enough elements to act as the row containers. Once the matrix mat has been resized, we need to resize each individual vector within the rows to the length representing the number of columns. The resize method can take an optional argument, which will initialise all elements to that particular value. Finally we adjust the private rows and cols unsigned integers to store the new row and column counts:

```
// Parameter Constructor
template<typename T>
QSMatrix<T>::QSMatrix(unsigned _rows, unsigned _cols, const T& _initial) {
  mat.resize(_rows);
  for (unsigned i=0; i<mat.size(); i++) {
    mat[i].resize(_cols, _initial);
  }
  rows = _rows;
  cols = _cols;
}
```

The copy constructor has a straightforward implementation. Since we have not used any *dynamic memory allocation*, we simply need to copy each private member from the corresponding copy matrix rhs:

```
// Copy Constructor
template<typename T>
QSMatrix<T>::QSMatrix(const QSMatrix<T>& rhs) {
  mat = rhs.mat;
  rows = rhs.get_rows();
  cols = rhs.get_cols();
}
```

The destructor is even simpler. Since there is no dynamic memory allocation, we don't need to do anything. We can let the compiler handle the destruction of the individual type members (mat, rows and cols):

```
// (Virtual) Destructor
template<typename T>
QSMatrix<T>::~QSMatrix() {}
```

The assignment operator is somewhat more complicated than the other construction/destruction methods. The first two lines of the method implementation check that the addresses of the two matrices aren't identical (i.e. we're not trying to assign a matrix to itself). If this is the case, then just return the dereferenced pointer to the current object (*this). This is purely for performance reasons. Why go through the process of copying exactly the same data into itself if it is already identical?

However, if the matrix in-memory addresses differ, then we resize the old matrix to the be the same size as the rhs matrix. Once that is complete we then populate the values element-wise and finally adjust the members holding the number of rows and columns. We then return the dereferenced pointer to this. This is a common pattern for assignment operators and is considered good practice:

```
// Assignment Operator
template<typename T>
QSMatrix<T>& QSMatrix<T>::operator=(const QSMatrix<T>& rhs) {
  if (&rhs == this)
    return *this;

  unsigned new_rows = rhs.get_rows();
  unsigned new_cols = rhs.get_cols();

  mat.resize(new_rows);
  for (unsigned i=0; i<mat.size(); i++) {
    mat[i].resize(new_cols);
  }

  for (unsigned i=0; i<new_rows; i++) {
    for (unsigned j=0; j<new_cols; j++) {
      mat[i][j] = rhs(i, j);
    }
  }
  rows = new_rows;
  cols = new_cols;

  return *this;
}
```

8.1.6 Mathematical Operators Implementation

The next part of the implementation concerns the methods overloading the binary operators that allow matrix algebra such as addition, subtraction and multiplication. There are two types of operators to be overloaded here. The first is operation *without assignment*. The second is operation *with assignment*. The first type of operator method creates a new matrix to store the result of an operation (such as addition), while the second type applies the result of the operation into the left-hand argument. For instance the first type will produce a new matrix C, from the equation $C = A + B$. The second type will overwrite A with the result of $A + B$.

The first operator to implement is for addition without assignment. A new matrix result is created with initial filled values equal to 0. Then each element is iterated through to be the pairwise sum of the **this** matrix and the new right hand side matrix rhs. Notice that we use the pointer dereferencing syntax with **this** when accessing the element values: **this**->mat[i][j]. This is identical to writing (***this**).mat[i][j]. We must dereference the pointer before we can access the underlying object. Finally, we return the result:

Note that this can be a particularly expensive operation. We are creating a new matrix for every call of this method. However, modern compilers are smart enough to make sure that this operation is not as performance heavy as it used to be, so for our current needs we are justified in creating the matrix here. Note again that if we were to return a matrix by reference and then create the matrix within the class via the **new** *operator, we would have an error as the matrix object would go out of scope as soon as the method returned.*

```
// Addition of two matrices
template<typename T>
QSMatrix<T> QSMatrix<T>::operator+(const QSMatrix<T>& rhs) {
  QSMatrix result(rows, cols, 0.0);

  for (unsigned i=0; i<rows; i++) {
    for (unsigned j=0; j<cols; j++) {
      result(i,j) = this->mat[i][j] + rhs(i,j);
    }
  }

  return result;
}
```

The operation with assignment method for addition is carried out slightly differently. It

DOES return a reference to an object, but this is fine since the object reference it returns is to **this**, which exists outside of the scope of the method. The method itself makes use of the **operator**+= that is bound to the *type* object. Thus when we carry out the line **this**->mat[i][j] += rhs(i,j); we are making use of the types own operator overload. Finally, we return a dereferenced pointer to **this** giving us back the modified matrix:

```
// Cumulative addition of this matrix and another
template<typename T>
QSMatrix<T>& QSMatrix<T>::operator+=(const QSMatrix<T>& rhs) {
  unsigned rows = rhs.get_rows();
  unsigned cols = rhs.get_cols();

  for (unsigned i=0; i<rows; i++) {
    for (unsigned j=0; j<cols; j++) {
      this->mat[i][j] += rhs(i,j);
    }
  }

  return *this;
}
```

The two matrix subtraction operators **operator**- and **operator**-= are almost identical to the addition variants, so I won't explain them here. If you wish to see their implementation, have a look at the full listing below.

I will discuss the matrix multiplication methods though as their syntax is sufficiently different to warrant explanation. The first operator is that without assignment, **operator***. We can use this to carry out an equation of the form $C = A \times B$. The first part of the method creates a new result matrix that has the same size as the right hand side matrix, rhs. Then we perform the triple loop associated with matrix multiplication. We iterate over each element in the result matrix and assign it the value of **this**->mat[i][k] * rhs(k,j), i.e. the value of $A_{ik} \times B_{kj}$, for $k \in \{0, ..., M-1\}$:

```
// Left multiplication of this matrix and another
template<typename T>
QSMatrix<T> QSMatrix<T>::operator*(const QSMatrix<T>& rhs) {
  unsigned rows = rhs.get_rows();
  unsigned cols = rhs.get_cols();
  QSMatrix result(rows, cols, 0.0);
  for (unsigned i=0; i<rows; i++) {
    for (unsigned j=0; j<cols; j++) {
      for (unsigned k=0; k<rows; k++) {
        result(i,j) += this->mat[i][k] * rhs(k,j);
      }
    }
  }

  return result;
}
```

The implementation of the operator*= is far simpler, but only because we are building on what already exists. The first line creates a new matrix called result which stores the result of multiplying the dereferenced pointer to this and the right hand side matrix, rhs. The second line then sets this to be equal to the result above. This is necessary as if it was carried out in one step, data would be overwritten before it could be used, creating an incorrect result. Finally the referenced pointer to this is returned. Most of the work is carried out by the operator* which is defined above. The listing is as follows:

```
// Cumulative left multiplication of this matrix and another
template<typename T>
QSMatrix<T>& QSMatrix<T>::operator*=(const QSMatrix<T>& rhs) {
  QSMatrix result = (*this) * rhs;
  (*this) = result;
  return *this;
}
```

We also wish to apply scalar element-wise operations to the matrix, in particular element-wise scalar addition, subtraction, multiplication and division. Since they are all very similar, I will only provide explanation for the addition operator. The first point of note is that the parameter is now a const T&, i.e. a reference to a const type. This is the scalar value that will be added to all matrix elements. We then create a new result matrix as before, of identical size to this. Then we iterate over the elements of the result matrix and set their values equal to the sum of the individual elements of this and our type value, rhs. Finally, we return the result matrix:

```
// Matrix/scalar addition
template<typename T>
```

```
QSMatrix<T> QSMatrix<T>::operator+(const T& rhs) {
  QSMatrix result(rows, cols, 0.0);

  for (unsigned i=0; i<rows; i++) {
    for (unsigned j=0; j<cols; j++) {
      result(i,j) = this->mat[i][j] + rhs;
    }
  }

  return result;
}
```

We also wish to allow (right) matrix vector multiplication. It is not too different from the implementation of matrix-matrix multiplication. In this instance we are returning a std::vector <T> and also providing a separate vector as a parameter. Upon invocation of the method we create a new result vector that has the same size as the right hand side, rhs. Then we perform a double loop over the elements of the **this** matrix and assign the result to an element of the result vector. Finally, we return the result vector:

```
// Multiply a matrix with a vector
template<typename T>
std::vector<T> QSMatrix<T>::operator*(const std::vector<T>& rhs) {
  std::vector<T> result(rhs.size(), 0.0);

  for (unsigned i=0; i<rows; i++) {
    for (unsigned j=0; j<cols; j++) {
      result[i] = this->mat[i][j] * rhs[j];
    }
  }

  return result;
}
```

I've added a final matrix method, which is useful for certain numerical linear algebra techniques. Essentially it returns a vector of the diagonal elements of the matrix. Firstly we create the result vector, then assign it the values of the diagonal elements and finally we return the result vector:

```
// Obtain a vector of the diagonal elements
```

```
template<typename T>
std::vector<T> QSMatrix<T>::diag_vec() {
  std::vector<T> result(rows, 0.0);

  for (unsigned i=0; i<rows; i++) {
    result[i] = this->mat[i][i];
  }

  return result;
}
```

The final set of methods to implement are for accessing the individual elements as well as getting the number of rows and columns from the matrix. They're all quite simple in their implementation. They dereference **this** and then obtain either an individual element or some private member data:

```
// Access the individual elements
template<typename T>
T& QSMatrix<T>::operator()(const unsigned& row, const unsigned& col) {
  return this->mat[row][col];
}

// Access the individual elements (const)
template<typename T>
const T& QSMatrix<T>::operator()(const unsigned& row, const unsigned& col)
    const {
  return this->mat[row][col];
}

// Get the number of rows of the matrix
template<typename T>
unsigned QSMatrix<T>::get_rows() const {
  return this->rows;
}

// Get the number of columns of the matrix
template<typename T>
unsigned QSMatrix<T>::get_cols() const {
  return this->cols;
}
```

8.1.7 Full Source Implementation

Now that we have described all the methods in full, here is the full source listing for the QSMatrix class:

```
#ifndef __QS_MATRIX_CPP
#define __QS_MATRIX_CPP

#include "matrix.h"

// Parameter Constructor
template<typename T>
QSMatrix<T>::QSMatrix(unsigned _rows, unsigned _cols, const T& _initial) {
  mat.resize(_rows);
  for (unsigned i=0; i<mat.size(); i++) {
    mat[i].resize(_cols, _initial);
  }
  rows = _rows;
  cols = _cols;
}

// Copy Constructor
template<typename T>
QSMatrix<T>::QSMatrix(const QSMatrix<T>& rhs) {
  mat = rhs.mat;
  rows = rhs.get_rows();
  cols = rhs.get_cols();
}

// (Virtual) Destructor
template<typename T>
QSMatrix<T>::~QSMatrix() {}

// Assignment Operator
template<typename T>
```

```
QSMatrix<T>& QSMatrix<T>::operator=(const QSMatrix<T>& rhs) {
  if (&rhs == this)
    return *this;

  unsigned new_rows = rhs.get_rows();
  unsigned new_cols = rhs.get_cols();

  mat.resize(new_rows);
  for (unsigned i=0; i<mat.size(); i++) {
    mat[i].resize(new_cols);
  }

  for (unsigned i=0; i<new_rows; i++) {
    for (unsigned j=0; j<new_cols; j++) {
      mat[i][j] = rhs(i, j);
    }
  }
  rows = new_rows;
  cols = new_cols;

  return *this;
}

// Addition of two matrices
template<typename T>
QSMatrix<T> QSMatrix<T>::operator+(const QSMatrix<T>& rhs) {
  QSMatrix result(rows, cols, 0.0);

  for (unsigned i=0; i<rows; i++) {
    for (unsigned j=0; j<cols; j++) {
      result(i,j) = this->mat[i][j] + rhs(i,j);
    }
  }

  return result;
}
```

```
// Cumulative addition of this matrix and another
template<typename T>
QSMatrix<T>& QSMatrix<T>::operator+=(const QSMatrix<T>& rhs) {
  unsigned rows = rhs.get_rows();
  unsigned cols = rhs.get_cols();

  for (unsigned i=0; i<rows; i++) {
    for (unsigned j=0; j<cols; j++) {
      this->mat[i][j] += rhs(i,j);
    }
  }

  return *this;
}

// Subtraction of this matrix and another
template<typename T>
QSMatrix<T> QSMatrix<T>::operator-(const QSMatrix<T>& rhs) {
  unsigned rows = rhs.get_rows();
  unsigned cols = rhs.get_cols();
  QSMatrix result(rows, cols, 0.0);

  for (unsigned i=0; i<rows; i++) {
    for (unsigned j=0; j<cols; j++) {
      result(i,j) = this->mat[i][j] - rhs(i,j);
    }
  }

  return result;
}

// Cumulative subtraction of this matrix and another
template<typename T>
QSMatrix<T>& QSMatrix<T>::operator-=(const QSMatrix<T>& rhs) {
  unsigned rows = rhs.get_rows();
  unsigned cols = rhs.get_cols();
  for (unsigned i=0; i<rows; i++) {
    for (unsigned j=0; j<cols; j++) {
      this->mat[i][j] -= rhs(i,j);
    }
  }

  return *this;
}

// Left multiplication of this matrix and another
template<typename T>
QSMatrix<T> QSMatrix<T>::operator*(const QSMatrix<T>& rhs) {
  unsigned rows = rhs.get_rows();
  unsigned cols = rhs.get_cols();
  QSMatrix result(rows, cols, 0.0);

  for (unsigned i=0; i<rows; i++) {
    for (unsigned j=0; j<cols; j++) {
      for (unsigned k=0; k<rows; k++) {
        result(i,j) += this->mat[i][k] * rhs(k,j);
      }
    }
  }

  return result;
}

// Cumulative left multiplication of this matrix and another
template<typename T>
QSMatrix<T>& QSMatrix<T>::operator*=(const QSMatrix<T>& rhs) {
  QSMatrix result = (*this) * rhs;
  (*this) = result;
  return *this;
}

// Calculate a transpose of this matrix
template<typename T>
```

```
QSMatrix<T> QSMatrix<T>::transpose() {
  QSMatrix result(rows, cols, 0.0);

  for (unsigned i=0; i<rows; i++) {
    for (unsigned j=0; j<cols; j++) {
      result(i,j) = this->mat[j][i];
    }
  }

  return result;
}

// Matrix/scalar addition
template<typename T>
QSMatrix<T> QSMatrix<T>::operator+(const T& rhs) {
  QSMatrix result(rows, cols, 0.0);

  for (unsigned i=0; i<rows; i++) {
    for (unsigned j=0; j<cols; j++) {
      result(i,j) = this->mat[i][j] + rhs;
    }
  }

  return result;
}

// Matrix/scalar subtraction
template<typename T>
QSMatrix<T> QSMatrix<T>::operator-(const T& rhs) {
  QSMatrix result(rows, cols, 0.0);

  for (unsigned i=0; i<rows; i++) {
    for (unsigned j=0; j<cols; j++) {
      result(i,j) = this->mat[i][j] - rhs;
    }
  }
  return result;
}

// Matrix/scalar multiplication
template<typename T>
QSMatrix<T> QSMatrix<T>::operator*(const T& rhs) {
  QSMatrix result(rows, cols, 0.0);

  for (unsigned i=0; i<rows; i++) {
    for (unsigned j=0; j<cols; j++) {
      result(i,j) = this->mat[i][j] * rhs;
    }
  }

  return result;
}

// Matrix/scalar division
template<typename T>
QSMatrix<T> QSMatrix<T>::operator/(const T& rhs) {
  QSMatrix result(rows, cols, 0.0);

  for (unsigned i=0; i<rows; i++) {
    for (unsigned j=0; j<cols; j++) {
      result(i,j) = this->mat[i][j] / rhs;
    }
  }

  return result;
}

// Multiply a matrix with a vector
template<typename T>
std::vector<T> QSMatrix<T>::operator*(const std::vector<T>& rhs) {
  std::vector<T> result(rhs.size(), 0.0);

  for (unsigned i=0; i<rows; i++) {
    for (unsigned j=0; j<cols; j++) {
      result[i] = this->mat[i][j] * rhs[j];
    }
  }

  return result;
}

// Obtain a vector of the diagonal elements
template<typename T>
std::vector<T> QSMatrix<T>::diag_vec() {
  std::vector<T> result(rows, 0.0);

  for (unsigned i=0; i<rows; i++) {
    result[i] = this->mat[i][i];
  }

  return result;
}

// Access the individual elements
template<typename T>
T& QSMatrix<T>::operator()(const unsigned& row, const unsigned& col) {
  return this->mat[row][col];
}

// Access the individual elements (const)
template<typename T>
const T& QSMatrix<T>::operator()(const unsigned& row, const unsigned& col)
    const {
  return this->mat[row][col];
}

// Get the number of rows of the matrix
template<typename T>
unsigned QSMatrix<T>::get_rows() const {
  return this->rows;
```

```
}

// Get the number of columns of the matrix
template<typename T>
unsigned QSMatrix<T>::get_cols() const {
  return this->cols;
}

#endif
```

8.1.8 Using the Matrix Class

We have the full listings for both the matrix header and source, so we can test the methods out with some examples. Here is the main listing showing the matrix addition operator:

```
#include "matrix.h"
#include <iostream>

int main(int argc, char **argv) {
  QSMatrix<double> mat1(10, 10, 1.0);
  QSMatrix<double> mat2(10, 10, 2.0);

  QSMatrix<double> mat3 = mat1 + mat2;

  for (int i=0; i<mat3.get_rows(); i++) {
    for (int j=0; j<mat3.get_cols(); j++) {
      std::cout << mat3(i,j) << ", ";
    }
    std::cout << std::endl;
  }

  return 0;
}
```

Here is the output of the code. We can see that the elements are all valued 3.0, which is simply the element-wise addition of mat1 and mat2:

```
3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
```

This concludes the section on creating a custom matrix library. Now we will consider how to use external libraries in a production environment.

8.2 External Matrix Libraries

We have previously considered *operator overloading* and how to create our own matrix object in C++. As a learning exercise, creating a matrix class can be extremely beneficial as it often covers dynamic memory allocation (if not using std::vectors) and operator overloading across multiple object types (matrices, vectors and scalars). However, it is far from optimal to carry this out in a production environment. This section will explain why it is better to use an external dedicated matrix library project, such as Eigen.

Writing our own matrix libraries is likely to cause a few issues in a production environment:

- **Poor Performance** - Unless significant time is spent optimising a custom matrix library, it is often going to be far slower than the corresponding dedicated community projects such as uBLAS, Eigen, MTL or Blitz.
- **Buggy Code** - A large community surrounding an open-source project leads to a greater number of bug fixes as edge cases are discovered and corrected. With a custom matrix library, it is likely that it will be restricted solely for an individual use case and thus edge cases are unlikely to be checked.
- **Few Algorithms** - Once again, the benefit of a large community is that multiple efficient algorithms can be generated for all aspects of numerical linear algebra. These in turn feed optimisations back to the core library and thus increase overall efficiency.
- **Time Wasting** - Are you in the business of quantitative finance or in the business of building an all-singing all-dancing matrix library? By spending months optimising a custom

library, you are neglecting the original intent of its usage of the first place - solving quant problems!

While many libraries exist (see above), I have chosen to use the Eigen project. Here are some of the benefits of Eigen:

- **Up to date** - Eigen is actively developed and releases new versions frequently
- **API** - Eigen has a simple, straightforward and familiar API syntax
- **Dynamic matrices** - Supports matrices with sizes determined at runtime
- **Well tested** - Eigen has extensive "battle testing" and thus few bugs
- **Storage** - Can use either row-major or column-major storage
- **Optimised structures** - Dense and sparse matrices both available
- **Expression templates** - Lazy evaluation, which allows for complex matrix arithmetic, while maintaining performance

In this section we will install Eigen, look at examples of basic linear algebra usage and briefly study some of the advanced features, which will be the subject of later chapters.

8.2.1 Installation

Eigen is extremely easy to install as there is no library that needs linking to. Instead the header files are simply included in the code for your program. With GCC it is necessary to use the -I flag in order for the compiler to be able to find the Eigen header files:

```
g++ -I /your/path/to/eigen/ example_program.cpp -o example_program
```

8.2.2 Basic Usage

The following program requires the Eigen/Dense header. It initialises a 3x3 matrix (using the MatrixXd template) and then prints it to the console:

```
#include <iostream>
#include <Eigen/Dense>

int main() {
  Eigen::MatrixXd m(3,3);
  m << 1, 2, 3,
        4, 5, 6,
        7, 8, 9;
  std::cout << m << std::endl;
}
```

Here is the (rather simple!) output for the above program:

```
1 2 3
4 5 6
7 8 9
```

Notice that the overloaded << operator can accept comma-separated lists of values in order to initialise the matrix. This is an extremely useful part of the API syntax. In addition, we can also pass the MatrixXd to std::cout and have the numbers output in a human-readable fashion.

8.2.3 Basic Linear Algebra

Eigen is a large library and has many features. We will be exploring many of them over subsequent chapters. In this section I want to describe basic matrix and vector operations, including the matrix-vector and matrix-matrix multiplication facilities provided with the library.

8.2.4 Expression Templates

One of the most attractive features of the Eigen library is that it includes *expression objects* and *lazy evaluation*. This means that any arithmetic operation actually returns such an expression object, which is actually a description of the final computation to be performed rather than the actual computation itself. The benefit of such an approach is that most compilers are able to heavily optimise the expression such that additional loops are completely minimised. As an example, an expression such as:

```
VectorXd p(10), q(10), r(10), s(10);
...
p = 5*q + 11*r - 7*s;
```

will compile into a single loop:

```
for (unsigned i=0; i<10; ++i) {
  p[i] = 5*q[i] + 11*r[i] - 7*s[i];
}
```

This means that the underlying storage arrays are only looped over once. As such Eigen can optimise relatively complicated arithmetic expressions.

8.2.5 Matrix and Scalar Arithmetic

Eigen allows for straightforward addition and subtraction of vectors and matrices. However, it is necessary for the operations to be mathematically well-defined: The two operands must have the same number of rows and columns. In addition, they must also possess the same scalar type. Here is an example of usage:

```
#include <iostream>
#include <Eigen/Dense>

int main() {
  // Define two matrices, both 3x3
  Eigen::Matrix3d p;
  Eigen::Matrix3d q;

  // Define two three-dimensional vectors
  // The constructor provides initialisation
  Eigen::Vector3d r(1,2,3);
  Eigen::Vector3d s(4,5,6);

  // Use the << operator to fill the matrices
  p << 1, 2, 3,
       4, 5, 6,
       7, 8, 9;
  q << 10, 11, 12,
       13, 14, 15,
       16, 17, 18;

  // Output arithmetic operations for matrices
  std::cout << "p+q=\n" << p + q << std::endl;
  std::cout << "p-q=\n" << p - q << std::endl;

  // Output arithmetic operations for vectors
  std::cout << "r+s=\n" << r + s << std::endl;
  std::cout << "r-s=\n" << r - s << std::endl;
}
```

Here is the output:

```
p+q=
11  13  15
17  19  21
23  25  27
p−q=
−9  −9  −9
−9  −9  −9
−9  −9  −9
r+s=
5
7
9
r−s=
−3
−3
−3
```

Scalar multiplication and division are just as simple in Eigen:

```
#include <iostream>
#include <Eigen/Dense>

int main() {
  // Define a 3x3 matrix and initialise
  Eigen::Matrix3d p;
  p << 1, 2, 3,
       4, 5, 6,
       7, 8, 9;

  // Multiply and divide by a scalar
  std::cout << "p * 3.14159 =\n" << p * 3.14159 << std::endl;
  std::cout << "p / 2.71828 =\n" << p / 2.71828 << std::endl;
}
```

Here is the output:

```
p * 3.14159 =
3.14159 6.28318 9.42477
12.5664  15.708 18.8495
```

```
21.9911  25.1327  28.2743
p / 2.71828 =
 0.36788  0.735759   1.10364
 1.47152    1.8394   2.20728
 2.57516   2.94304   3.31092
```

8.2.6 Matrix Transposition

Eigen also has operations for the transpose, conjugate and the adjoint of a matrix. For quantitative finance work we are really only interested in the transpose, as we will not often be utilising complex numbers!

It is necessary to be careful when using the tranpose operation for in-place assignment (this applies to the conjugate and the adjoint too) as the transpose operation will write into the original matrix before finalising the calculation, which can lead to unexpected behaviour. Thus it is necessary to use Eigen's transposeInPlace method on matrix objects when carrying out in-place transposition.

```
#include <iostream>
#include <Eigen/Dense>

int main() {
  // Declare a 3x3 matrix with random entries
  Eigen::Matrix3d p = Eigen::Matrix3d::Random(3,3);

  // Output the transpose of p
  std::cout << "p^T =\n" << p.transpose() << std::endl;

  // In-place transposition
  p.transposeInPlace();

  // Output the in-place transpose of p
  std::cout << "p^T =\n" << p << std::endl;
}
```

Here is the output:

```
p^T =
-0.999984  -0.736924   0.511211
```

```
-0.0826997 0.0655345  -0.562082
-0.905911   0.357729   0.358593
p^T =
-0.999984  -0.736924   0.511211
-0.0826997 0.0655345  -0.562082
-0.905911   0.357729   0.358593
```

8.2.7 Matrix/Matrix and Matrix/Vector Multiplication

Eigen handles matrix/matrix and matrix/vector multiplication with a simple API. Vectors are matrices of a particular type (and defined that way in Eigen) so all operations simply overload the **operator**∗. Here is an example of usage for matrices, vectors and transpose operations:

```
#include <iostream>
#include <Eigen/Dense>

int main() {
  // Define a 3x3 matrix and two 3-dimensional vectors
  Eigen::Matrix3d p;
  p << 1, 2, 3,
       4, 5, 6,
       7, 8, 9;
  Eigen::Vector3d r(10, 11, 12);
  Eigen::Vector3d s(13, 14, 15);

  // Matrix/matrix multiplication
  std::cout << "p*p:\n" << p*p << std::endl;

  // Matrix/vector multiplication
  std::cout << "p*r:\n" << p*r << std::endl;
  std::cout << "r^T*p:\n" << r.transpose()*p << std::endl;

  // Vector/vector multiplication (inner product)
  std::cout << "r^T*s:\n" << r.transpose()*s << std::endl;
}
```

Here is the output:

```
p*p:
 30   36   42
 66   81   96
102  126  150
p*r:
68
167
266
r^T*p:
138  171  204
r^T*s:
464
```

8.2.8 Vector Operations

Eigen also supports common vector operations, such as the inner product ("dot" product) and the vector product ("cross" product). Note that the sizes of the operand vectors are restricted by the mathematical definitions of each operator. The dot product must be applied to two vectors of equal dimension, while the cross product is only defined for three-dimensional vectors:

```cpp
#include <iostream>
#include <Eigen/Dense>

int main() {
  // Declare and initialise two 3D vectors
  Eigen::Vector3d r(10,20,30);
  Eigen::Vector3d s(40,50,60);

  // Apply the 'dot' and 'cross' products
  std::cout << "r . s =\n" << r.dot(s) << std::endl;
  std::cout << "r x s =\n" << r.cross(s) << std::endl;
}
```

Here is the output:

```
r . s =
3200
r x s =
-300
600
```

```
−300
```

8.2.9 Reduction

"Reduction" in this context means to take a matrix or vector as an argument and obtain a scalar as a result. There are six reduction operations which interest us:

- sum - Calculates the sum of all elements in a vector or matrix
- prod - Calculates the product of all elements in a vector or matrix
- mean - Calculates the mean average of all elements in a vector or matrix
- minCoeff - Calculates the minimum element in a vector or matrix
- maxCoeff - Calculates the maximum element in a vector or matrix
- trace - Calculates the *trace* of a matrix, i.e. the sum of the diagonal elements

Here is an example of usage:

```cpp
#include <iostream>
#include <Eigen/Dense>

int main() {
  // Declare and initialise a 3D matrix
  Eigen::Matrix3d p;
  p << 1, 2, 3,
       4, 5, 6,
       7, 8, 9;

  // Output the reduction operations
  std::cout << "p.sum(): " << p.sum() << std::endl;
  std::cout << "p.prod(): " << p.prod() << std::endl;
  std::cout << "p.mean(): " << p.mean() << std::endl;
  std::cout << "p.minCoeff(): " << p.minCoeff() << std::endl;
  std::cout << "p.maxCoeff(): " << p.maxCoeff() << std::endl;
  std::cout << "p.trace(): " << p.trace() << std::endl;
}
```

Here is the output:

```
p.sum(): 45
p.prod(): 362880
p.mean(): 5
p.minCoeff(): 1
p.maxCoeff(): 9
p.trace(): 15
```

8.2.10 Useful Features

Eigen contains many more features than I have listed here. In particular, it supports multiple data structures for efficient matrix storage, depending on structural sparsity of values via the Sparse namespace. Further, Eigen has support for LR, Cholesky, SVD and QR decomposition. Eigenvalues can also be calculated in an optimised manner. The Geometry module allows calculation of geometric quantities, including transforms, translations, scaling, rotations and quaternions.

8.2.11 Next Steps

Now that we have explored the basic usage of the Eigen library, as well as glimpsed at the higher functionality, we are ready to carry out Numerical Linear Algebra (NLA) with Eigen as the basis (excuse the pun) for our code. This will help us solve more advanced problems in quantitative trading and derivatives pricing.

Chapter 9

Numerical Linear Algebra

Numerical Linear Algebra (NLA) is a branch of numerical analysis that concerns itself with the study of algorithms for performing linear algebra calculations. It is an extremely important area of computational finance as many quant methods rely on NLA techniques. In particular, the matrix computations carried out in NLA form the basis of the Finite Difference Method, which is often used to price certain path-dependent options via numerical solution of the Black-Scholes PDE.

9.1 Overview

While the Finite Difference Method motivates the use of NLA techniques, such as LU Decomposition and the Thomas Algorithm, the subject is applicable to a wide range of tasks within quantitative finance. The Cholesky Decomposition finds application among correlation matrices. QR Decomposition is an essential component of the *linear least square* problem, used in *regression analysis*. Singular Value Decomposition (SVD) and Eigenvalue Decomposition are two other techniques commonly used in finance for creating predictive models or carrying out expiatory data analysis.

In this section we will study the LU Decomposition, the Thomas Algorithm, the Cholesky Decomposition and the QR Decomposition. We will apply these algorithms to quantitative problems later in the book.

9.2 LU Decomposition

In this section we will discuss the **LU Decomposition** method, which is used in certain quantitative finance algorithms.

One of the key methods for solving the Black-Scholes Partial Differential Equation (PDE) model of options pricing is using Finite Difference Methods (FDM) to discretise the PDE and evaluate the solution numerically. Certain implicit Finite Difference Methods eventually lead to a system of linear equations.

This system of linear equations can be formulated as a matrix equation, involving the matrix A and the vectors x and b, of which x is the solution to be determined. Often these matrices are banded (their non-zero elements are confined to a subset of diagonals) and specialist algorithms (such as the Thomas Algorithm, see below) are used to solve them. LU Decomposition will aid us in solving the following matrix equation, without the direct need to invert the matrix A.

$$Ax = b$$

Although suboptimal from a performance point of view, we are going to implement our own version of the LU Decomposition in order to see how the algorithm is ”translated” from the mathematical realm into C++.

We will make use of the *Doolittle's LUP decomposition with partial pivoting* to decompose our matrix A into $PA = LU$, where L is a lower triangular matrix, U is an upper triangular matrix and P is a permutation matrix. P is needed to resolve certain singularity issues.

9.2.1 Algorithm

To calculate the upper triangular section we use the following formula for elements of U:

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} u_{kj} l_{ik}$$

The formula for elements of the lower triangular matrix L is similar, except that we need to divide each term by the corresponding diagonal element of U. To ensure that the algorithm is numerically stable when $u_{jj} \ll 0$, a *pivoting* matrix P is used to re-order A so that the largest element of each column of A gets shifted to the diagonal of A. The formula for elements of L follows:

$$l_{ij} = \frac{1}{u_{jj}}(a_{ij} - \sum_{k=1}^{j-1} u_{kj} l_{ik})$$

9.2.2 Eigen C++ Implementation

Creating a custom numerical linear algebra library, while a worthwhile learning exercise, is suboptimal from an implementation point of view. We have discussed this issue in the previous chapter and came to the conclusion that an external, high-performance matrix library was more appropriate. We will continue to make use of Eigen (although it isn't technically a library!) for our NLA work.

Here is a listing that shows how to make use of Eigen in order to create a LU Decomposition matrix. The code outputs the lower and upper triangular matrices separately.

Firstly, a typedef is created to allow us to use a 4×4 matrix in all subsequent lines. Then we create and populate a 4×4 matrix, outputting it to the terminal. Subsequently, we use the PartialPivLU template to create a LU Decomposition object, which is then used to provide lower and upper triangular views for the matrix. The key method is triangularView, which allows us to pass StrictlyLower or Upper into the template parameter in order to allow us to output the correct matrix.

Notice how straightforward the API syntax is:

```
#include <iostream>
#include <Eigen/Dense>
#include <Eigen/LU>

int main() {
  typedef Eigen::Matrix<double, 4, 4> Matrix4x4;

  // Declare a 4x4 matrix with defined entries
  Matrix4x4 p;
  p << 7, 3, -1, 2,
       3, 8, 1, -4,
       -1, 1, 4, -1,
       2, -4, -1, 6;
  std::cout << "Matrix P:\n" << p << std::endl << std::endl;

  // Create LU Decomposition template object for p
  Eigen::PartialPivLU<Matrix4x4> lu(p);
  std::cout<< "LU Matrix:\n" << lu.matrixLU() << std::endl << std::endl;

  // Output L, the lower triangular matrix
  Matrix4x4 l = Eigen::MatrixXd::Identity(4,4);
  l.block<4,4>(0,0).triangularView<Eigen::StrictlyLower>() = lu.matrixLU();
  std::cout << "L Matrix:\n" << l << std::endl << std::endl;

  // Output U, the upper triangular matrix
  Matrix4x4 u = lu.matrixLU().triangularView<Eigen::Upper>();
  std::cout << "R Matrix:\n" << u << std::endl;

  return 0;
}
```

The output is as follows:

```
Matrix P:
 7  3 -1  2
 3  8  1 -4
-1  1  4 -1
 2 -4 -1  6

LU Matrix:
        7          3         -1          2
 0.428571    6.71429    1.42857   -4.85714
-0.142857   0.212766    3.55319   0.319149
 0.285714  -0.723404  0.0898204    1.88623

L Matrix:
        1          0          0          0
 0.428571          1          0          0
-0.142857   0.212766          1          0
 0.285714  -0.723404  0.0898204          1

R Matrix:
       7         3        -1         2
       0   6.71429   1.42857  -4.85714
```

```
        0            0    3.55319  0.319149
        0            0          0   1.88623
```

9.3 Thomas Tridiagonal Matrix Algorithm

The Tridiagonal Matrix Algorithm, also known as the Thomas Algorithm (due to Llewellyn Thomas), is an application of gaussian elimination to a tri-banded matrix. The algorithm itself requires five parameters, each vectors. The first three parameters **a**, **b** and **c** represent the elements in the tridiagonal bands. Since **b** represents the diagonal elements it is one element longer than **a** and **c**, which represent the off-diagonal bands. The latter two parameters represent the solution vector **f** and **d**, the right-hand column vector.

The original system is written as:

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & ... & 0 \\ a_2 & b_2 & c_2 & 0 & ... & 0 \\ 0 & a_3 & b_3 & c_3 & 0 & 0 \\ . & . & & & & . \\ . & . & & & & . \\ . & . & & & & c_{k-1} \\ 0 & 0 & 0 & 0 & a_{k-1} & b_k \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ . \\ . \\ . \\ f_k \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ . \\ . \\ . \\ d_k \end{bmatrix}$$

The method begins by forming coefficients c_i^* and d_i^* in place of a_i, b_i and c_i as follows:

$$c_i^* = \begin{cases} \frac{c_1}{b_1} & ; i = 1 \\ \frac{c_i}{b_i - c_{i-1}^* a_i} & ; i = 2, 3, ..., k-1 \end{cases}$$

$$d_i^* = \begin{cases} \frac{d_1}{b_1} & ; i = 1 \\ \frac{d_i - d_{i-1}^* a_i}{b_i - c_{i-1}^* a_i} & ; i = 2, 3, ..., k-1 \end{cases}$$

With these new coefficients, the matrix equation can be rewritten as:

$$\begin{bmatrix} 1 & c_1^* & 0 & 0 & ... & 0 \\ 0 & 1 & c_2^* & 0 & ... & 0 \\ 0 & 0 & 1 & c_3^* & 0 & 0 \\ . & . & & & & . \\ . & . & & & & . \\ . & . & & & & c_{k-1}^* \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ . \\ . \\ . \\ f_k \end{bmatrix} = \begin{bmatrix} d_1^* \\ d_2^* \\ d_3^* \\ . \\ . \\ . \\ d_k^* \end{bmatrix}$$

The algorithm for the solution of these equations is now straightforward and works 'in reverse':

$$f_k = d_k^*, \qquad f_i = d_k^* - c_i^* x_{i+1}, \qquad i = k-1, k-2, ..., 2, 1$$

9.3.1 C++ Implementation

You can see the function prototype here:

```
void thomas_algorithm(const std::vector<double>& a,
                      const std::vector<double>& b,
                      const std::vector<double>& c,
                      const std::vector<double>& d,
                      std::vector<double>& f) {
```

Notice that **f** is a non-const vector, which means it is the only vector to be modified within the function.

It is possible to write the Thomas Algorithm function in a more efficient manner to override the **c** and **d** vectors. Finite difference methods require the vectors to be re-used for each time step, so the following implementation utilises two additional temporary vectors, $\mathbf{c}^*$ and $\mathbf{d}^*$. The memory for the vectors has been allocated within the function. Every time the function is called these vectors are allocated and deallocated, which is suboptimal from an efficiency point of view.

A proper "production" implementation would pass references to these vectors from an external function that only requires a single allocation and deallocation. However, in order to keep the function straightforward to understand I've not included this aspect:

```
std::vector<double> c_star(N, 0.0);
std::vector<double> d_star(N, 0.0);
```

The first step in the algorithm is to initialise the beginning elements of the $\mathbf{c}^*$ and $\mathbf{d}^*$ vectors:

```
c_star[0] = c[0] / b[0];
```

```
d_star[0] = d[0] / b[0];
```

The next step, known as the "forward sweep" is to fill the $\mathbf{c}^*$ and $\mathbf{d}^*$ vectors such that we form the second matrix equation $\mathbf{A}^*f = d^*$:

```
for (int i=1; i<N; i++) {
  double m = 1.0 / (b[i] - a[i] * c_star[i-1]);
  c_star[i] = c[i] * m;
  d_star[i] = (d[i] - a[i] * d_star[i-1]) * m;
}
```

Once the forward sweep is carried out the final step is to carry out the "reverse sweep". Notice that the vector **f** is actually being assigned here. The function itself is **void**, so we don't return any values.

```
for (int i=N-1; i-- > 0; ) {
  f[i] = d_star[i] - c_star[i] * d[i+1];
}
```

I've included a main function, which sets up the Thomas Algorithm to solve one time-step of the Crank-Nicolson finite difference method discretised diffusion equation. The details of the algorithm are not so important here, as I will be elucidating on the method in further articles on QuantStart.com when we come to solve the Black-Scholes equation. This is only provided in order to show you how the function works in a "real world" situation:

```
#include <cmath>
#include <iostream>
#include <vector>

// Vectors a, b, c and d are const. They will not be modified
// by the function. Vector f (the solution vector) is non-const
// and thus will be calculated and updated by the function.
void thomas_algorithm(const std::vector<double>& a,
                      const std::vector<double>& b,
                      const std::vector<double>& c,
                      const std::vector<double>& d,
                      std::vector<double>& f) {
  size_t N = d.size();

  // Create the temporary vectors
  // Note that this is inefficient as it is possible to call
  // this function many times. A better implementation would
  // pass these temporary matrices by non-const reference to
  // save excess allocation and deallocation
  std::vector<double> c_star(N, 0.0);
  std::vector<double> d_star(N, 0.0);

  // This updates the coefficients in the first row
  // Note that we should be checking for division by zero here
  c_star[0] = c[0] / b[0];
  d_star[0] = d[0] / b[0];

  // Create the c_star and d_star coefficients in the forward sweep
  for (int i=1; i<N; i++) {
    double m = 1.0 / (b[i] - a[i] * c_star[i-1]);
    c_star[i] = c[i] * m;
    d_star[i] = (d[i] - a[i] * d_star[i-1]) * m;
  }

  // This is the reverse sweep, used to update the solution vector f
  for (int i=N-1; i-- > 0; ) {
    f[i] = d_star[i] - c_star[i] * d[i+1];
  }
}

// Although thomas_algorithm provides everything necessary to solve
// a tridiagonal system, it is helpful to wrap it up in a "real world"
// example. The main function below uses a tridiagonal system from
// a Boundary Value Problem (BVP). This is the discretisation of the
// 1D heat equation.
int main(int argc, char **argv) {

  // Create a Finite Difference Method (FDM) mesh with 13 points
  // using the Crank-Nicolson method to solve the discretised
  // heat equation.
  size_t N = 13;
  double delta_x = 1.0/static_cast<double>(N);
double delta_t = 0.001;
double r = delta_t/(delta_x*delta_x);

// First we create the vectors to store the coefficients
std::vector<double> a(N-1, -r/2.0);
std::vector<double> b(N, 1.0+r);
std::vector<double> c(N-1, -r/2.0);
std::vector<double> d(N, 0.0);
std::vector<double> f(N, 0.0);

// Fill in the current time step initial value
// vector using three peaks with various amplitudes
f[5] = 1; f[6] = 2; f[7] = 1;

// We output the solution vector f, prior to a
// new time-step
std::cout << "f = (";
for (int i=0; i<N; i++) {
  std::cout << f[i];
  if (i < N-1) {
    std:: cout << ", ";
  }
}
std::cout << ")" << std::endl << std::endl;

// Fill in the current time step vector d
for (int i=1; i<N-1; i++) {
  d[i] = r*0.5*f[i+1] + (1.0-r)*f[i] + r*0.5*f[i-1];
}

// Now we solve the tridiagonal system
thomas_algorithm(a, b, c, d, f);

// Finally we output the solution vector f
std::cout << "f = (";
for (int i=0; i<N; i++) {
  std::cout << f[i];
```

```
    if (i < N-1) {
      std:: cout << ",␣";
    }
  }
  std::cout << ")" << std::endl;

  return 0;
}
```

The output of the program is follows:

```
f = (0, 0, 0, 0, 0, 1, 2, 1, 0, 0, 0, 0, 0)

f = (0, 0, 0, 0.00614025, 0.145331, 0.99828, 1.7101, 0.985075, 0.143801,
0.0104494, 0.000759311, 0, 0)
```

You can see the the initial vector and then the solution vector after one time-step. Since we are modelling the diffusion/heat equation, you can see how the initial "heat" has spread out. Later on we will see how the diffusion equation represents the Black-Scholes equation with modified boundary conditions. One can think of solving the Black-Scholes equation as solving the heat equation "in reverse". In that process the "diffusion" happens against the flow of time.

9.4 Cholesky Decomposition

The Cholesky decomposition method makes an appearance in Monte Carlo Methods where it is used in simulating systems with correlated variables (we make use of it in the later chapter on Stochastic Volatility models). Cholesky decomposition is applied to the correlation matrix, providing a lower triangular matrix L, which when applied to a vector of uncorrelated samples, u, produces the covariance vector of the system. Thus it is highly relevant for quantitative trading.

Cholesky decomposition assumes that the matrix being decomposed is *Hermitian* and *positive-definite*. Since we are only interested in real-valued matrices, we can replace the property of Hermitian with that of *symmetric* (i.e. the matrix equals its own transpose). Cholesky decomposition is approximately 2x faster than LU Decomposition, where it applies.

For a Hermitian, positive-definite matrix P, the algorithm creates a lower-triangular matrix L, which when multiplied by its (conjugate) transpose, $L*^T$ (which is of course upper triangular), provides the original Hermitian, positive-definite matrix:

$$P = LL*^T \tag{9.1}$$

9.4.1 Algorithm

In order to solve for the lower triangular matrix, we will make use of the **Cholesky-Banachiewicz Algorithm**. First, we calculate the values for L on the main diagonal. Subsequently, we calculate the off-diagonals for the elements below the diagonal:

$$\begin{aligned} l_{kk} &= \sqrt{a_{kk} - \sum_{j=1}^{k-1} l_{kj}^2} \\ l_{ik} &= \frac{1}{l_{kk}} \left(a_{ik} - \sum_{j=1}^{k-1} l_{ij} l_{kj} \right), i > k \end{aligned}$$

9.4.2 Eigen Implementation

As with the LU Decomposition, we will make use of the Eigen headers to calculate the Cholesky Decomposition. Firstly, we declare a typedef for a 4×4 matrix, as with the LU Decomposition. We create and populate such a matrix and use the Eigen::LLT template to create the actual Cholesky Decomposition. "LLT" here refers to the matrix expression $LL*^T$. Once we have the Eigen::LLT object, we call the matrixL method to obtain the lower triangular matrix L. Finally, we obtain its transpose and then check that the multiplication of both successfully reproduces the matrix P.

```
#include <iostream>
#include <Eigen/Dense>

int main() {
  typedef Eigen::Matrix<double, 4, 4> Matrix4x4;

  // Declare a 4x4 matrix with defined entries
  Matrix4x4 p;
  p << 6, 3, 4, 8,
       3, 6, 5, 1,
       4, 5, 10, 7,
       8, 1, 7, 25;
  std::cout << "Matrix P:\n" << p << std::endl << std::endl;

  // Create the L and L^T matrices (LLT)
  Eigen::LLT<Matrix4x4> llt(p);

  // Output L, the lower triangular matrix
  Matrix4x4 l = llt.matrixL();
  std::cout << "L Matrix:\n" << l << std::endl << std::endl;

  // Output L^T, the upper triangular conjugate transpose of L
  Matrix4x4 u = l.transpose();
  std::cout << "L^T Matrix:\n" << u << std::endl << std::endl;

  // Check that LL^T = P
  std::cout << "LL^T Matrix:\n" << l*u << std::endl;

  return 0;
}
```

The output is given as follows:

```
Matrix P:
 6  3  4  8
 3  6  5  1
 4  5 10  7
 8  1  7 25

L Matrix:
 2.44949        0        0        0
 1.22474  2.12132        0        0
 1.63299  1.41421   2.3094        0
 3.26599 -1.41421  1.58771  3.13249

L^T Matrix:
 2.44949  1.22474  1.63299  3.26599
       0  2.12132  1.41421 -1.41421
       0        0   2.3094  1.58771
       0        0        0  3.13249
```

```
LL^T Matrix:
 6   3   4   8
 3   6   5   1
 4   5  10   7
 8   1   7  25
```

9.5 QR Decomposition

QR Decomposition is widely used in quantitative finance as the basis for the solution of the *linear least squares* problem, which itself is used for statistical *regression analysis*.

One of the key benefits of using QR Decomposition over other methods for solving linear least squares is that it is more numerically stable, albeit at the expense of being slower to execute. Hence if you are performing a large quantity of regressions as part of a trading backtest, for instance, you will need to consider very extensively whether QR Decomposition is the best fit (excuse the pun).

For a square matrix A the QR Decomposition converts A into the product of an orthogonal matrix Q (i.e. $Q^T Q = I$) and an upper triangular matrix R. Hence:

$$A = QR$$

There are a few different algorithms for calculating the matrices Q and R. We will outline the method of Householder Reflections, which is known to be more numerically stable the the alternative Gramm-Schmidt method. I've outlined the Householder Reflections method below.

Note, the following explanation is an expansion of the extremely detailed article on QR Decomposition using Householder Reflections over at Wikipedia.

9.5.1 Algorithm

A Householder Reflection is a linear transformation that enables a vector to be reflected through a plane or hyperplane. Essentially, we use this method because we want to create an upper triangular matrix, R. The householder reflection is able to carry out this vector reflection such that all but one of the coordinates disappears. The matrix Q will be built up as a sequence of matrix multiplications that eliminate each coordinate in turn, up to the rank of the matrix A.

The first step is to create the vector $\mathbf{x}$, which is the k-th column of the matrix A, for step k. We define $\alpha = -sgn(\mathbf{x}_k)(||\mathbf{x}||)$. The norm $||\cdot||$ used here is the *Euclidean norm.* Given the first column vector of the identity matrix, I of equal size to A, $_1 = (1, 0, ..., 0)^T$, we create the vector $\mathbf{u}$:

$$\mathbf{u} = \mathbf{x} + \alpha\mathbf{e}_1$$

Once we have the vector $\mathbf{u}$, we need to convert it to a unit vector, which we denote as $\mathbf{v}$:

$$\mathbf{v} = \mathbf{u}/||\mathbf{u}||$$

Now we form the matrix Q out of the identity matrix I and the vector multiplication of $\mathbf{v}$:

$$Q = I - 2\mathbf{v}\mathbf{v}^T$$

Q is now an $m \times m$ Householder matrix, with $Q\mathbf{x} = (\alpha, 0, ..., 0)^T$. We will use Q to transform A to upper triangular form, giving us the matrix R. We denote Q as Q_k and, since $k = 1$ in this first step, we have Q_1 as our first Householder matrix. We multiply this with A to give us:

$$Q_1 A = \begin{bmatrix} \alpha_1 & \star & \dots & \star \\ 0 & & & \\ \vdots & & A' & \\ 0 & & & \end{bmatrix}$$

The whole process is now repeated for the minor matrix A', which will give a second Householder matrix Q'_2. Now we have to "pad out" this minor matrix with elements from the identity matrix such that we can consistently multiply the Householder matrices together. Hence, we define Q_k as the block matrix:

$$Q_k = \begin{pmatrix} I_{k-1} & 0 \\ 0 & Q'_k \end{pmatrix}$$

Once we have carried out t iterations of this process we have R as an upper triangular matrix:

$$R = Q_t ... Q_2 Q_1 A$$

Q is then fully defined as the multiplication of the transposes of each Q_k:

$$Q = Q_1^T Q_2^T ... Q_t^T$$

This gives $A = QR$, the QR Decomposition of A.

9.5.2 Eigen Implementation

As with the prior examples making use of the Eigen headers, calculation of a QR decomposition is extremely straightforward. Firstly, we create a 3×3 matrix and populate it with values. We then use the HouseholderQR template class to create a QR object. It supports a method householderQ that returns the matrix Q described above.

```
#include <iostream>
#include <Eigen/Dense>

int main() {
  // Declare a 3x3 matrix with defined entries
  Eigen::MatrixXf p(3,3);
  p << 12, -51, 4,
       6, 167, -68,
       -4, 24, -41;
  std::cout << "Matrix P:\n" << p << std::endl << std::endl;

  // Create the Householder QR Matrix object
  Eigen::HouseholderQR<Eigen::MatrixXf> qr(p);
  Eigen::MatrixXf q = qr.householderQ();

  // Output Q, the Householder matrix
  std::cout << "Q Matrix:\n" << q << std::endl << std::endl;

  return 0;
}
```

We have now described four separate numerical linear algebra algorithms used extensively in quantitative finance. In subsequent chapters we will apply these algorithms to specific quant problems.

Chapter 10

European Options with Monte Carlo

In this chapter we will price a European vanilla option via the correct analytic solution of the Black-Scholes equation, as well as via the Monte Carlo method. We won't be concentrating on an extremely efficient or optimised implementation at this stage. Right now I just want to show you how the mathematical formulae correspond to the C++ code.

10.1 Black-Scholes Analytic Pricing Formula

The first stage in implementation is to briefly discuss the Black-Scholes analytic solution for the price of a vanilla call or put option. Consider the price of a European Vanilla Call, $C(S,t)$. S is the underlying asset price, K is the strike price, r is the interest rate (or the "risk-free rate"), T is the time to maturity and σ is the (constant) volatility of the underlying asset S. N is a function which will be described in detail below. The analytical formula for $C(S,t)$ is given by:

$$C(S,t) = SN(d_1) - Ke^{-rT}N(d_2)$$

With d_1 and d_2 defined as follows:

$$\begin{aligned} d_1 &= \frac{log(S/K) + (r + \frac{\sigma^2}{2})T}{\sigma\sqrt{T}} \\ d_2 &= d_1 - \sigma\sqrt{T} \end{aligned}$$

Making use of *put-call parity*, we can also price a European vanilla put, $P(S,t)$, with the following formula:

$$P(S,t) = Ke^{-rT} - S + C(S,t) = Ke^{-rT} - S + (SN(d_1) - Ke^{-rT}N(d_2))$$

All that remains is to describe the function N, which is the cumulative distribution function of the standard normal distribution. The formula for N is given by:

$$N(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2} dt$$

It would also help to have closed form solutions for the "Greeks". These are the sensitivities of the option price to the various underlying parameters. They will be calculated in the next chapter. However, in order to calculate these sensitivities we need the formula for the probability density function of the standard normal distribution which is given below:

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$$

10.2 C++ Implementation

This code will not consist of any object oriented or generic programming techniques at this stage. Right now the goal is to help you understand a basic C++ implementation of a pricing engine, without all of the additional object machinery to encapsulate away the mathematical functions. I've written out the program in full and then below I'll explain how each component works subsequently:

```
#define _USE_MATH_DEFINES

#include <iostream>
#include <cmath>

// Standard normal probability density function
double norm_pdf(const double& x) {
    return (1.0/(pow(2*M_PI,0.5)))*exp(-0.5*x*x);
```

```
}

// An approximation to the cumulative distribution function
// for the standard normal distribution
// Note: This is a recursive function
double norm_cdf(const double& x) {
    double k = 1.0/(1.0 + 0.2316419*x);
    double k_sum = k*(0.319381530 + k*(-0.356563782 + k*(1.781477937 +
                   k*(-1.821255978 + 1.330274429*k))));

    if (x >= 0.0) {
        return (1.0 - (1.0/(pow(2*M_PI,0.5)))*exp(-0.5*x*x) * k_sum);
    } else {
        return 1.0 - norm_cdf(-x);
    }
}

// This calculates d_j, for j in {1,2}. This term appears in the closed
// form solution for the European call or put price
double d_j(const int& j, const double& S, const double& K, const double& r,
           const double& v, const double& T) {
    return (log(S/K) + (r + (pow(-1,j-1))*0.5*v*v)*T)/(v*(pow(T,0.5)));
}

// Calculate the European vanilla call price based on
// underlying S, strike K, risk-free rate r, volatility of
// underlying sigma and time to maturity T
double call_price(const double& S, const double& K, const double& r,
           const double& v, const double& T) {
    return S * norm_cdf(d_j(1, S, K, r, v, T))-K*exp(-r*T) *
               norm_cdf(d_j(2, S, K, r, v, T));
}

// Calculate the European vanilla put price based on
// underlying S, strike K, risk-free rate r, volatility of
// underlying sigma and time to maturity T
double put_price(const double& S, const double& K, const double& r,
           const double& v, const double& T) {
    return -S*norm_cdf(-d_j(1, S, K, r, v, T))+K*exp(-r*T) *
              norm_cdf(-d_j(2, S, K, r, v, T));
}

int main(int argc, char **argv) {
    // First we create the parameter list
    double S = 100.0;  // Option price
    double K = 100.0;  // Strike price
    double r = 0.05;   // Risk-free rate (5%)
    double v = 0.2;    // Volatility of the underlying (20%)
    double T = 1.0;    // One year until expiry

    // Then we calculate the call/put values
    double call = call_price(S, K, r, v, T);
    double put = put_price(S, K, r, v, T);

    // Finally we output the parameters and prices
    std::cout << "Underlying:␣␣␣␣␣␣" << S << std::endl;
    std::cout << "Strike:␣␣␣␣␣␣␣␣␣␣" << K << std::endl;
    std::cout << "Risk-Free␣Rate:␣␣" << r << std::endl;
    std::cout << "Volatility:␣␣␣␣␣␣" << v << std::endl;
    std::cout << "Maturity:␣␣␣␣␣␣␣␣" << T << std::endl;

    std::cout << "Call␣Price:␣␣␣␣␣␣" << call << std::endl;
    std::cout << "Put␣Price:␣␣␣␣␣␣␣" << put << std::endl;

    return 0;
}
```

Let's now take a look at the program step-by-step.

The first line is a pre-processor macro which tells the C++ compiler to make use of the C-standard mathematical constants. Note that some compilers may not fully support these constants. Make sure you test them out first!

```
#define _USE_MATH_DEFINES
```

The next section imports the iostream and cmath libraries. This allows us to use the std::cout command to output code. In addition we now have access to the C mathematical functions such as exp, pow, log and sqrt:

```
#include <iostream>
#include <cmath>
```

Once we have the correct libraries imported we need to create the core statistics functions which make up most of the computation for the prices. Here is the standard normal probability density function. M_PI is the C-standard mathematical constant for π:

```
// Standard normal probability density function
double norm_pdf(const double& x) {
    return (1.0/(pow(2*M_PI,0.5)))*exp(-0.5*x*x);
}
```

The next statistics function is the approximation to the cumulative distribution function for the standard normal distribution. This approximation is found in Joshi. Note that this is a recursive function (i.e. it calls itself!):

```
// An approximation to the cumulative distribution function
// for the standard normal distribution
// Note: This is a recursive function
double norm_cdf(const double& x) {
    double k = 1.0/(1.0 + 0.2316419*x);
    double k_sum = k*(0.319381530 + k*(-0.356563782 + k*(1.781477937 +
                   k*(-1.821255978 + 1.330274429*k))));

    if (x >= 0.0) {
        return (1.0 - (1.0/(pow(2*M_PI,0.5)))*exp(-0.5*x*x) * k_sum);
    } else {
        return 1.0 - norm_cdf(-x);
    }
}
```

The last remaining set of functions are directly related to the pricing of European vanilla calls and puts. We need to calculate the d_j values first, otherwise we would be repeating ourselves by adding the function code directly to each of call_price and put_price:

```
// This calculates d_j, for j in {1,2}. This term appears in the closed
// form solution for the European call or put price
```

```
double d_j(const int& j, const double& S, const double& K, const double& r,
           const double& v, const double& T) {
    return (log(S/K) + (r + (pow(-1,j-1))*0.5*v*v)*T)/(v*(pow(T,0.5)));
}
```

Now that we have the cumulative distribution function for the standard normal distribution (norm_cdf) coded up, as well as the d_j function, we can calculate the closed-form solution for the European vanilla call price. You can see that it is a fairly simple formula, assuming that the aforementioned functions have already been implemented:

```
// Calculate the European vanilla call price based on
// underlying S, strike K, risk-free rate r, volatility of
// underlying sigma and time to maturity T
double call_price(const double& S, const double& K, const double& r,
           const double& v, const double& T) {
    return S * norm_cdf(d_j(1, S, K, r, v, T))-K*exp(-r*T) *
           norm_cdf(d_j(2, S, K, r, v, T));
}
```

And similarly for the vanilla put (this formula can be derived easily via put-call parity):

```
// Calculate the European vanilla put price based on
// underlying S, strike K, risk-free rate r, volatility of
// underlying sigma and time to maturity T
double put_price(const double& S, const double& K, const double& r,
                 const double& v, const double& T) {
    return -S*norm_cdf(-d_j(1, S, K, r, v, T))+K*exp(-r*T) *
           norm_cdf(-d_j(2, S, K, r, v, T));
}
```

Every C++ program must have a main program. This is where the functions described above are actually called and the values derived are output to the console. We make use of the iostream library to provide us with the std::cout and std::endl output handlers. Finally, we exit the program:

```
int main(int argc, char **argv) {
    // First we create the parameter list
    double S = 100.0;  // Option price
    double K = 100.0;  // Strike price
    double r = 0.05;   // Risk-free rate (5%)
    double v = 0.2;     // Volatility of the underlying (20%)
    double T = 1.0;     // One year until expiry

    // Then we calculate the call/put values
    double call = call_price(S, K, r, v, T);
    double put = put_price(S, K, r, v, T);

    // Finally we output the parameters and prices
    std::cout << "Underlying:      " << S << std::endl;
    std::cout << "Strike:          " << K << std::endl;
    std::cout << "Risk-Free Rate:  " << r << std::endl;
    std::cout << "Volatility:      " << v << std::endl;
    std::cout << "Maturity:        " << T << std::endl;

    std::cout << "Call Price:      " << call << std::endl;
    std::cout << "Put Price:       " << put << std::endl;

    return 0;
}
```

The output of the code on my Mac OSX system is as follows:

```
Underlying:      100
Strike:          100
Risk-Free Rate:  0.05
Volatility:      0.2
Maturity:        1
Call Price:      10.4506
Put Price:       5.57352
```

This code should give you a good idea of how closed form solutions to the Black-Scholes equations can be coded up in a *procedural* manner with C++. The next steps are to calculate the "Greeks" in the same vein, as closed form solutions exist, solutions for digital and power options, as well as a basic Monte Carlo pricer with which to validate against.

We have now shown how to price a European option with analytic solutions. We were able to take the closed-form solution of the Black-Scholes equation for a European vanilla call or put and provide a price.

This is possible because the *boundary conditions* generated by the pay-off function of the European vanilla option allow us to easily calculate a closed-form solution. Many option pay-off functions lead to boundary conditions which are much harder to solve analytically and some are impossible to solve this way. Thus in these situations we need to rely on *numerical approximation*.

We will now price the same European vanilla option with a very basic Monte Carlo solver in C++ and then compare our numerical values to the analytical case. We won't be concentrating on an extremely efficient or optimised implementation at this stage. Right now I just want to show you how to get up and running to give you an understanding of how risk neutral pricing works numerically. We will also see how our values differ from those generated analytically.

10.3 Risk Neutral Pricing of a European Vanilla Option

The first stage in implementation is to briefly discuss how risk neutral pricing works for a vanilla call or put option. We haven't yet discussed on risk neutral pricing in depth, so in the meantime, it might help to have a look at the article on Geometric Brownian Motion on QuantStart.com, which is the underlying model of stock price evolution that we will be using. It is given by the following stochastic differential equation:

$$dS(t) = \mu S(t)dt + \sigma S(t)dB(t)$$

where S is the asset price, μ is the *drift* of the stock, σ is the volatility of the stock and B is a Brownian motion (or Wiener process).

You can think of dB as being a normally distributed random variable with zero mean and variance dt. We are going to use the fact that the price of a European vanilla option is given as the discounted expectation of the option pay-off of the final spot price (at maturity T):

$$e^{-rT}\mathbb{E}(f(S(T)))$$

This expectation is taken under the appropriate risk-neutral measure, which sets the drift μ equal to the risk-free rate r:

$$dS(t) = rS(t)dt + \sigma S(t)dB(t)$$

We can make use of Ito's Lemma to give us:

$$d \log S(t) = \left(r - \frac{1}{2}\sigma^2\right) dt + \sigma dB(t)$$

This is a constant coefficient SDE and therefore the solution is given by:

$$\log S(t) = \log S(0) + \left(r - \frac{1}{2}\sigma^2\right) t + \sigma\sqrt{t}N(0,1)$$

Here I've used the fact that since $B(t)$ is a Brownian motion, it has the distribution of a normal distribution with variance t and mean zero so it can be written as $B(t) = \sqrt{t}N(0,1)$.

Rewriting the above equation in terms of $S(t)$ by taking the exponential gives:

$$S(t) = S(0)e^{(r-\frac{1}{2}\sigma^2)t+\sigma\sqrt{t}N(0,1)}$$

Using the risk-neutral pricing method above leads to an expression for the option price as follows:

$$e^{-rT}\mathbb{E}(f(S(0)e^{(r-\frac{1}{2}\sigma^2)T+\sigma\sqrt{T}N(0,1)}))$$

The key to the Monte Carlo method is to make use of the law of large numbers in order to approximate the expectation. Thus the essence of the method is to compute many draws from the normal distribution $N(0,1)$, as the variable x, and then compute the pay-off via the following formula:

$$f(S(0)e^{(r-\frac{1}{2}\sigma^2)T+\sigma\sqrt{T}x})$$

In this case the value of f is the pay-off for a call or put. By averaging the sum of these pay-offs and then taking the risk-free discount we obtain the *approximate* price for the option.

10.4 C++ Implementation

For this implementation we won't be utilising any object oriented or generic programming techniques right now. At this stage the goal is to generate understanding of a basic C++ Monte Carlo implementation. I've written out the program in full and then below I'll explain how each component works subsequently:

```
#include <algorithm>    // Needed for the "max" function
#include <cmath>
#include <iostream>

// A simple implementation of the Box-Muller algorithm, used to generate
// gaussian random numbers - necessary for the Monte Carlo method below
// Note that C++11 actually provides std::normal_distribution<> in
// the <random> library, which can be used instead of this function
double gaussian_box_muller() {
  double x = 0.0;
  double y = 0.0;
  double euclid_sq = 0.0;

  // Continue generating two uniform random variables
  // until the square of their "euclidean distance"
  // is less than unity
  do {
    x = 2.0 * rand() / static_cast<double>(RAND_MAX)-1;
    y = 2.0 * rand() / static_cast<double>(RAND_MAX)-1;
    euclid_sq = x*x + y*y;
  } while (euclid_sq >= 1.0);

  return x*sqrt(-2*log(euclid_sq)/euclid_sq);
}

// Pricing a European vanilla call option with a Monte Carlo method
double monte_carlo_call_price(const int& num_sims, const double& S,
                              const double& K, const double& r,
                              const double& v, const double& T) {
  double S_adjust = S * exp(T*(r-0.5*v*v));
  double S_cur = 0.0;
  double payoff_sum = 0.0;

  for (int i=0; i<num_sims; i++) {
    double gauss_bm = gaussian_box_muller();
    S_cur = S_adjust * exp(sqrt(v*v*T)*gauss_bm);
    payoff_sum += std::max(S_cur - K, 0.0);
  }

  return (payoff_sum / static_cast<double>(num_sims)) * exp(-r*T);
}

// Pricing a European vanilla put option with a Monte Carlo method
double monte_carlo_put_price(const int& num_sims, const double& S,
                             const double& K, const double& r,
                             const double& v, const double& T) {
  double S_adjust = S * exp(T*(r-0.5*v*v));
  double S_cur = 0.0;
  double payoff_sum = 0.0;

  for (int i=0; i<num_sims; i++) {
    double gauss_bm = gaussian_box_muller();
    S_cur = S_adjust * exp(sqrt(v*v*T)*gauss_bm);
    payoff_sum += std::max(K - S_cur, 0.0);
  }

  return (payoff_sum / static_cast<double>(num_sims)) * exp(-r*T);
}

int main(int argc, char **argv) {
  // First we create the parameter list
  int num_sims = 10000000;   // Number of simulated asset paths
  double S = 100.0;  // Option price
  double K = 100.0;  // Strike price
  double r = 0.05;   // Risk-free rate (5%)
  double v = 0.2;    // Volatility of the underlying (20%)
  double T = 1.0;    // One year until expiry
  // Then we calculate the call/put values via Monte Carlo
  double call = monte_carlo_call_price(num_sims, S, K, r, v, T);
  double put = monte_carlo_put_price(num_sims, S, K, r, v, T);

  // Finally we output the parameters and prices
  std::cout << "Number of Paths: " << num_sims << std::endl;
  std::cout << "Underlying:      " << S << std::endl;
  std::cout << "Strike:          " << K << std::endl;
  std::cout << "Risk-Free Rate:  " << r << std::endl;
  std::cout << "Volatility:      " << v << std::endl;
  std::cout << "Maturity:        " << T << std::endl;

  std::cout << "Call Price:      " << call << std::endl;
  std::cout << "Put Price:       " << put << std::endl;

  return 0;
}
```

Let's now look at the program in a step-by-step fashion.

The first section imports the algorithm, iostream and cmath libraries. The algorithm library provides access to the max comparison function. This allows us to use the std::cout command to output code. In addition we now have access to the C mathematical functions such as exp, pow, log and sqrt:

```
#include <algorithm>
#include <cmath>
#include <iostream>
```

The first function to implement is the Box-Muller algorithm. As stated above, it is designed to convert two uniform random variables into a standard Gaussian random variable. Box-Muller is a good choice for a random number generator if your compiler doesn't support the C++11 standard. However, if you do have a compiler that supports it, you can make use of the std::normal_distribution<> template class found in the <random> library.

```
// A simple implementation of the Box-Muller algorithm, used to generate
// gaussian random numbers - necessary for the Monte Carlo method below
// Note that C++11 actually provides std::normal_distribution<> in
// the <random> library, which can be used instead of this function
double gaussian_box_muller() {
  double x = 0.0;
  double y = 0.0;
  double euclid_sq = 0.0;

  // Continue generating two uniform random variables
  // until the square of their "euclidean distance"
  // is less than unity
  do {
    x = 2.0 * rand() / static_cast<double>(RAND_MAX)-1;
    y = 2.0 * rand() / static_cast<double>(RAND_MAX)-1;
    euclid_sq = x*x + y*y;
  } while (euclid_sq >= 1.0);

  return x*sqrt(-2*log(euclid_sq)/euclid_sq);
}
```

The next two functions are used to price a European vanilla call or put via the Monte Carlo method. We've outlined above how the theory works, so let's study the implementation itself. The comments will give you the best overview. I've neglected to include the function for the put, as it is almost identical to the call and only differs in the nature of the pay-off. In fact, this is a hint telling us that our code is possibly repeating itself (a violation of the Do-Not-Repeat-Yourself, DRY, principle):

```
// Pricing a European vanilla call option with a Monte Carlo method
double monte_carlo_call_price(const int& num_sims, const double& S,
                              const double& K, const double& r,
                              const double& v, const double& T) {
  double S_adjust = S * exp(T*(r-0.5*v*v));    // The adjustment to the spot
      price
  double S_cur = 0.0;    // Our current asset price ("spot")
  double payoff_sum = 0.0;   // Holds the sum of all of the final option pay
     -offs

  for (int i=0; i<num_sims; i++) {
    // Generate a Gaussian random number via Box-Muller
    double gauss_bm = gaussian_box_muller();

    // Adjust the spot price via the Brownian motion final distribution
    S_cur = S_adjust * exp(sqrt(v*v*T)*gauss_bm);

    // Take the option pay-off, then add it to the rest of the pay-off
        values
    payoff_sum += std::max(S_cur - K, 0.0);
  }

  // Average the pay-off sum via the number of paths and then
  // discount the risk-free rate from the price
  return (payoff_sum / static_cast<double>(num_sims)) * exp(-r*T);
}
```

The main function is extremely similar to that found in European vanilla option pricing with C++ and analytic formulae. The exception is the addition of the num_sims variable which stores the number of calculated asset paths. The larger this value, the more accurate the option price will be. Hence an inevitable tradeoff between execution time and price accuracy exists. Unfortunately, this is not a problem limited to this book! Another modification is that the call and put values are now derived from the Monte Carlo function calls:

```
int main(int argc, char **argv) {
  // First we create the parameter list
  int num_sims = 10000000;   // Number of simulated asset paths
  double S = 100.0;  // Option price
  double K = 100.0;  // Strike price
  double r = 0.05;   // Risk-free rate (5%)
  double v = 0.2;    // Volatility of the underlying (20%)
  double T = 1.0;    // One year until expiry

  // Then we calculate the call/put values via Monte Carlo
  double call = monte_carlo_call_price(num_sims, S, K, r, v, T);
  double put = monte_carlo_put_price(num_sims, S, K, r, v, T);

  // Finally we output the parameters and prices
  std::cout << "Number of Paths: " << num_sims << std::endl;
  std::cout << "Underlying:      " << S << std::endl;
  std::cout << "Strike:          " << K << std::endl;
  std::cout << "Risk-Free Rate:  " << r << std::endl;
  std::cout << "Volatility:      " << v << std::endl;
  std::cout << "Maturity:         " << T << std::endl;

  std::cout << "Call Price:       " << call << std::endl;
  std::cout << "Put Price:        " << put << std::endl;

  return 0;
}
```

The output of the program is given as follows:

```
Number of Paths: 10000000
Underlying:      100
Strike:          100
Risk-Free Rate:  0.05
Volatility:      0.2
Maturity:        1
Call Price:      10.4553
Put Price:       5.57388
```

We can compare this with the output from the analytical formulae generated in European vanilla option pricing with C++ and analytic formulae, which are given below for convenience. This serves two purposes. Firstly, we can see that the values are almost identical, which provides an extra level of validation that the code in both instances is pricing correctly. Secondly, we can compare accuracy:

```
Call Price:      10.4506
Put Price:       5.57352
```

As can be seen the prices are relatively accurate for 10^7 simulation paths. On my MacBook Air this program took a few seconds to calculate. Adding a couple of orders of magnitude on to the number of paths would quickly make the program prohibitive to execute. Thus we have run into the first problem - that of optimising execution speed. The second is that of making the above code maintainable. Notice that there is a significant duplication of code in both the monte_carlo_call_price and monte_carlo_put_price functions.

Chapter 11

Calculating the "Greeks"

One of the core financial applications of derivatives pricing theory is to be able to *manage risk* via a liquid options market. Such a market provides the capability for firms and individuals to tailor their risk exposure depending upon their hedging or speculation requirements. In order to effectively assess such risks, it is necessary to calculate the *sensitivity* of an options price to the factors that affect it, such as the underlying asset price, volatility and time to option expiry.

If we assume the existence of an analytical formula for an option price (such as in the case of European vanilla call/put options on a single asset) then it is possible to differentiate the call price with respect to its parameters in order to generate these sensitivities. The common sensitivities of interest include:

- **Delta** - Derivative of an option with respect to (w.r.t.) the spot price, $\frac{\partial C}{\partial S}$
- **Gamma** - Second derivative of an option w.r.t. the spot price, $\frac{\partial^2 C}{\partial S^2}$
- **Vega** - Derivative of an option w.r.t. the underlying volatility, $\frac{\partial C}{\partial \sigma}$
- **Theta** - (Negative) derivative of an option w.r.t. the time to expiry, $\frac{\partial C}{\partial t}$
- **Rho** - Derivative of an option w.r.t. the interest rate, $\frac{\partial C}{\partial \rho}$

Since all of the sensitivities are commonly denoted by letters of the Greek alphabet (except Vega!) they have come to be known colloquially as "the Greeks".

In this chapter we will calculate the Greeks using three separate methods. In the first instance we will utilise formula derived directly from the analytic formulae for European vanilla call and put options on a single asset. This will provide us with a baseline to determine the accuracy of subsequent numerical methods.

The first numerical approach utilised will be based on a Finite Difference Method (FDM) and the original analytical formulae. The second numerical method will use a combination of the FDM technique and Monte Carlo for pricing. The latter approach is readily applicable to a wider range of contingent claims as it is not dependent upon the existence of an analytic solution.

11.1 Analytic Formulae

The formulae of the Greeks for a European vanilla call and put option on a single asset are tabulated below:

	Calls	Puts
Delta, $\frac{\partial C}{\partial S}$	$N(d_1)$	$N(d_1) - 1$
Gamma, $\frac{\partial^2 C}{\partial S^2}$	$\frac{N'(d_1)}{S\sigma\sqrt{T-t}}$	
Vega, $\frac{\partial C}{\partial \sigma}$	$SN'(d_1)\sqrt{T-t}$	
Theta, $\frac{\partial C}{\partial t}$	$-\frac{SN'(d_1)\sigma}{2\sqrt{T-t}} - rKe^{-r(T-t)}N(d_2)$	$-\frac{SN'(d_1)\sigma}{2\sqrt{T-t}} + rKe^{-r(T-t)}N(-d_2)$
Rho, $\frac{\partial C}{\partial r}$	$K(T-t)e^{-r(T-t)}N(d_2)$	$-K(T-t)e^{-r(T-t)}N(-d_2)$

Where N is the cumulative distribution function of the standard normal distribution, N' is the probability density function of the standard normal distribution, $d_1 = (log(S/K) + (r + \frac{\sigma^2}{2})T)/(\sigma\sqrt{T})$ and $d_2 = d_1 - \sigma\sqrt{T}$.

The following listing implements these methods in C++ making use of the formulae for the probability functions and also includes a basic Monte Carlo pricer, both taken from the chapter on European Options Pricing. Here is the listing for black_scholes.h:

```
#define _USE_MATH_DEFINES

#include <iostream>
#include <cmath>
#include <algorithm>    // Needed for the "max" function

// ================
// ANALYTIC FORMULAE
// ================

// Standard normal probability density function
```

```
double norm_pdf(const double x) {
  return (1.0/(pow(2*M_PI,0.5)))*exp(-0.5*x*x);
}

// An approximation to the cumulative distribution function
// for the standard normal distribution
// Note: This is a recursive function
double norm_cdf(const double x) {
  double k = 1.0/(1.0 + 0.2316419*x);
  double k_sum = k*(0.319381530 + k*(-0.356563782 + k*(1.781477937 + k
      *(-1.821255978 + 1.330274429*k))));

  if (x >= 0.0) {
    return (1.0 - (1.0/(pow(2*M_PI,0.5)))*exp(-0.5*x*x) * k_sum);
  } else {
    return 1.0 - norm_cdf(-x);
  }
}

// This calculates d_j, for j in {1,2}. This term appears in the closed
// form solution for the European call or put price
double d_j(const int j, const double S, const double K, const double r,
    const double v, const double T) {
  return (log(S/K) + (r + (pow(-1,j-1))*0.5*v*v)*T)/(v*(pow(T,0.5)));
}

// Calculate the European vanilla call price based on
// underlying S, strike K, risk-free rate r, volatility of
// underlying sigma and time to maturity T
double call_price(const double S, const double K, const double r, const
    double v, const double T) {
  return S * norm_cdf(d_j(1, S, K, r, v, T))-K*exp(-r*T) * norm_cdf(d_j(2,
      S, K, r, v, T));
}

// Calculate the European vanilla call Delta
double call_delta(const double S, const double K, const double r, const
    double v, const double T) {
  return norm_cdf(d_j(1, S, K, r, v, T));
}

// Calculate the European vanilla call Gamma
double call_gamma(const double S, const double K, const double r, const
    double v, const double T) {
  return norm_pdf(d_j(1, S, K, r, v, T))/(S*v*sqrt(T));
}

// Calculate the European vanilla call Vega
double call_vega(const double S, const double K, const double r, const
    double v, const double T) {
  return S*norm_pdf(d_j(1, S, K, r, v, T))*sqrt(T);
}

// Calculate the European vanilla call Theta
double call_theta(const double S, const double K, const double r, const
    double v, const double T) {
  return -(S*norm_pdf(d_j(1, S, K, r, v, T))*v)/(2*sqrt(T))
    - r*K*exp(-r*T)*norm_cdf(d_j(2, S, K, r, v, T));
}

// Calculate the European vanilla call Rho
double call_rho(const double S, const double K, const double r, const
    double v, const double T) {
  return K*T*exp(-r*T)*norm_cdf(d_j(2, S, K, r, v, T));
}

// Calculate the European vanilla put price based on
// underlying S, strike K, risk-free rate r, volatility of
// underlying sigma and time to maturity T
double put_price(const double S, const double K, const double r, const
    double v, const double T) {
  return -S*norm_cdf(-d_j(1, S, K, r, v, T))+K*exp(-r*T) * norm_cdf(-d_j(2,
       S, K, r, v, T));
}
```

```
// Calculate the European vanilla put Delta
double put_delta(const double S, const double K, const double r, const
    double v, const double T) {
  return norm_cdf(d_j(1, S, K, r, v, T)) - 1;
}

// Calculate the European vanilla put Gamma
double put_gamma(const double S, const double K, const double r, const
    double v, const double T) {
  return call_gamma(S, K, r, v, T); // Identical to call by put-call parity
}

// Calculate the European vanilla put Vega
double put_vega(const double S, const double K, const double r, const
    double v, const double T) {
  return call_vega(S, K, r, v, T); // Identical to call by put-call parity
}

// Calculate the European vanilla put Theta
double put_theta(const double S, const double K, const double r, const
    double v, const double T) {
  return -(S*norm_pdf(d_j(1, S, K, r, v, T))*v)/(2*sqrt(T))
    + r*K*exp(-r*T)*norm_cdf(-d_j(2, S, K, r, v, T));
}

// Calculate the European vanilla put Rho
double put_rho(const double S, const double K, const double r, const double
     v, const double T) {
  return -T*K*exp(-r*T)*norm_cdf(-d_j(2, S, K, r, v, T));
}

// ===========
// MONTE CARLO
// ===========

// A simple implementation of the Box-Muller algorithm, used to generate
// gaussian random numbers - necessary for the Monte Carlo method below
// Note that C++11 actually provides std::normal_distribution<> in
// the <random> library, which can be used instead of this function
double gaussian_box_muller() {
  double x = 0.0;
  double y = 0.0;
  double euclid_sq = 0.0;

  // Continue generating two uniform random variables
  // until the square of their "euclidean distance"
  // is less than unity
  do {
    x = 2.0 * rand() / static_cast<double>(RAND_MAX)-1;
    y = 2.0 * rand() / static_cast<double>(RAND_MAX)-1;
    euclid_sq = x*x + y*y;
  } while (euclid_sq >= 1.0);

  return x*sqrt(-2*log(euclid_sq)/euclid_sq);
}
```

Here is the listing for the main.cpp file making use of the above header:

```
#include "black_scholes.h"

int main(int argc, char **argv) {
  // First we create the parameter list
  double S = 100.0;  // Option price
  double K = 100.0;  // Strike price
  double r = 0.05;   // Risk-free rate (5%)
  double v = 0.2;    // Volatility of the underlying (20%)
  double T = 1.0;    // One year until expiry

  // Then we calculate the call/put values and the Greeks
  double call = call_price(S, K, r, v, T);
  double call_delta_v = call_delta(S, K, r, v, T);
  double call_gamma_v = call_gamma(S, K, r, v, T);
  double call_vega_v = call_vega(S, K, r, v, T);
  double call_theta_v = call_theta(S, K, r, v, T);
  double call_rho_v = call_rho(S, K, r, v, T);

  double put = put_price(S, K, r, v, T);
  double put_delta_v = put_delta(S, K, r, v, T);
  double put_gamma_v = put_gamma(S, K, r, v, T);
  double put_vega_v = put_vega(S, K, r, v, T);
  double put_theta_v = put_theta(S, K, r, v, T);
  double put_rho_v = put_rho(S, K, r, v, T);

  // Finally we output the parameters and prices
  std::cout << "Underlying:␣␣␣␣␣␣" << S << std::endl;
  std::cout << "Strike:␣␣␣␣␣␣␣␣␣␣" << K << std::endl;
  std::cout << "Risk-Free␣Rate:␣␣" << r << std::endl;
  std::cout << "Volatility:␣␣␣␣␣␣" << v << std::endl;
  std::cout << "Maturity:␣␣␣␣␣␣␣␣" << T << std::endl << std::endl;

  std::cout << "Call␣Price:␣␣␣␣␣␣" << call << std::endl;
  std::cout << "Call␣Delta:␣␣␣␣␣␣" << call_delta_v << std::endl;
  std::cout << "Call␣Gamma:␣␣␣␣␣␣" << call_gamma_v << std::endl;
  std::cout << "Call␣Vega:␣␣␣␣␣␣␣" << call_vega_v << std::endl;
  std::cout << "Call␣Theta:␣␣␣␣␣␣" << call_theta_v << std::endl;
  std::cout << "Call␣Rho:␣␣␣␣␣␣␣␣" << call_rho_v << std::endl << std::endl;

  std::cout << "Put␣Price:␣␣␣␣␣␣␣" << put << std::endl;
  std::cout << "Put␣Delta:␣␣␣␣␣␣␣" << put_delta_v << std::endl;
  std::cout << "Put␣Gamma:␣␣␣␣␣␣␣" << put_gamma_v << std::endl;
  std::cout << "Put␣Vega:␣␣␣␣␣␣␣␣" << put_vega_v << std::endl;
  std::cout << "Put␣Theta:␣␣␣␣␣␣␣" << put_theta_v << std::endl;
  std::cout << "Put␣Rho:␣␣␣␣␣␣␣␣␣" << put_rho_v << std::endl;

  return 0;
}
```

The output of this program is as follows:

```
Underlying:      100
Strike:          100
Risk-Free Rate:  0.05
```

```
Volatility:        0.2
Maturity:          1

Call Price:        10.4506
Call Delta:        0.636831
Call Gamma:        0.018762
Call Vega:         37.524
Call Theta:        −6.41403
Call Rho:          53.2325

Put Price:         5.57352
Put Delta:         −0.363169
Put Gamma:         0.018762
Put Vega:          37.524
Put Theta:         −1.65788
Put Rho:           −41.8905
```

We are now going to compare the analytical prices with those derived from a Finite Difference Method.

11.2 Finite Difference Method

FDM is widely used in derivatives pricing (as well as engineering/physics in general) to solve partial differential equations (PDE). In fact, we will see in a later chapter how to use FDM to solve the Black-Scholes equation in a numerical manner. In this section, however, we are going to apply the same technique, namely the discretisation of the partial derivatives, to create a simple approximation to the Greek sensitivities with which we can compare to the analytical solution.

The essence of the method is that we will approximate the partial derivative representing the particular sensitivity of interest. To do this we make use of the analytical formulae for the Black-Scholes prices of the call and puts. These formulae are covered in the previous chapter on European Options pricing.

As an example, let's assume we want to calculate the Delta of a call option. The Delta is given by $\frac{\partial C}{\partial S}(S, T, \sigma, r, K)$. If we calculate two call prices, one at spot S and the other at spot $S + \Delta S$, subtract the prices and divide by ΔS, we have a *forward difference approximation* to the derivative:

$$\frac{\partial C}{\partial S} \approx \frac{C(S+\Delta S, T, \sigma, r, K) - C(S, T, \sigma, r, K)}{\Delta S} \tag{11.1}$$

Each of the additional first order sensitivities (Vega, Rho and Theta) can be calculated in this manner by simply incrementing the correct parameter dimension. Gamma on the other hand is a second order derivative and so must be approximated in a different way. The usual approach in FDM is to use a *central difference approximation* to produce the following formula:

$$\frac{\partial^2 C}{\partial S^2} \approx \frac{C(S+\Delta S, T, \sigma, r, K) - 2C(S, T, \sigma, r, K) + C(S-\Delta S, T, \sigma, r, K)}{(\Delta S)^2} \tag{11.2}$$

At this stage we will keep the code procedural as we wish to emphasise the mathematical formulae. We are now able to implement the FDM numerical approximations in C++. For the sake of brevity we will restrict ourselves to the calculation of the call Delta and Gamma, as the remaining sensitivities are similar:

```
#include "black_scholes.h"

// This uses the forward difference approximation to calculate the Delta of
     a call option
double call_delta_fdm(const double S, const double K, const double r, const
     double v, const double T, const double delta_S) {
  return (call_price(S + delta_S, K, r, v, T) - call_price(S, K, r, v, T))/
       delta_S;
}

// This uses the centred difference approximation to calculate the Gamma of
     a call option
double call_gamma_fdm(const double S, const double K, const double r, const
     double v, const double T, const double delta_S) {
  return (call_price(S + delta_S, K, r, v, T) - 2*call_price(S, K, r, v, T)
        + call_price(S - delta_S, K, r, v, T))/(delta_S*delta_S);
}

int main(int argc, char **argv) {
  // First we create the parameter list
  double S = 100.0;          // Option price
  double delta_S = 0.001;    // Option price increment
  double K = 100.0;          // Strike price
  double r = 0.05;           // Risk-free rate (5%)
  double v = 0.2;            // Volatility of the underlying (20%)
  double T = 1.0;            // One year until expiry

  // Then we calculate the Delta and the Gamma for the call
  double call_delta_f = call_delta_fdm(S, K, r, v, T, delta_S);
  double call_gamma_f = call_gamma_fdm(S, K, r, v, T, delta_S);

  // Finally we output the parameters and greeks
  std::cout << "Underlying:        " << S << std::endl;
  std::cout << "Delta underlying:  " << delta_S << std::endl;
  std::cout << "Strike:            " << K << std::endl;
  std::cout << "Risk-Free Rate:    " << r << std::endl;
  std::cout << "Volatility:        " << v << std::endl;
  std::cout << "Maturity:          " << T << std::endl << std::endl;

  std::cout << "Call Delta:        " << call_delta_f << std::endl;
  std::cout << "Call Gamma:        " << call_gamma_f << std::endl;
}
```

The output of this program is as follows:

```
Underlying:        100
Delta underlying:  0.001
Strike:            100
Risk-Free Rate:    0.05
Volatility:        0.2
Maturity:          1

Call Delta:        0.636845
Call Gamma:        0.0187633
```

11.3 Monte Carlo Method

The final method of calculating the Greeks is to use a combination of the FDM and Monte Carlo. The overall method is the same as above, with the exception that we will replace the analytical prices of the call/puts in the Finite Difference approximation and use a Monte Carlo engine instead to calculate the prices. This method is significantly more versatile as it can be extended to many differing types of contingent claim prices.

It is extremely important to re-use the random draws from the initial Monte Carlo simulation in calculating the incremented/decremented parameter paths. Thus, when calculating $C(S, K, r, v, T)$ and $C(S + \Delta S, K, r, v, T)$ it is necessary to use the same random draws. Further, it is significantly more optimal as there is no need to calculate additional asset paths for each incremented parameter instance.

The following code calculates the Monte Carlo price for the Delta and the Gamma, making use of separate Monte Carlo prices for each instance. The essence of the Monte Carlo method is to calculate three separate stock paths, all based on the same Gaussian draws. Each of these draws will represent an increment (or not) to the asset path parameter, S. Once those paths have been generated a price for each can be calculated. This price is then substituted into the FDM derivative approximations, in exactly the same way as with the analytical formulae.

The final prices are passed to the monte_carlo_call_price function by reference and the function itself is **void**. This provides a simple mechanism for returning multiple **double**s from the function. An alternative would be to pass a vector of values by reference, to be set.

It is straightforward to modify the code to calculate the Vega, Rho or Theta (based on the Delta). A more 'production ready' implementation would utilise an object-oriented framework. However, for the purposes of this chapter an OOP-based implementation would likely obscure the algorithm:

```
#include "black_scholes.h"

// Pricing a European vanilla call option with a Monte Carlo method
// Create three separate paths, each with either an increment, non-
// increment or decrement based on delta_S, the stock path parameter
void monte_carlo_call_price(const int num_sims,
                            const double S, const double K, const double
                               r,
                            const double v, const double T, const double
                               delta_S,
                                double& price_Sp , double& price_S , double&
                                    price_Sm) {

// Since we wish to use the same Gaussian random draws for each path, it
    is
// necessary to create three separated adjusted stock paths for each
// increment/decrement of the asset
double Sp_adjust = (S+delta_S) * exp(T*(r-0.5*v*v));
double S_adjust = S * exp(T*(r-0.5*v*v));
double Sm_adjust = (S-delta_S) * exp(T*(r-0.5*v*v));

// These will store all three 'current' prices as the Monte Carlo
// algorithm is carried out
double Sp_cur = 0.0;
double S_cur = 0.0;
double Sm_cur = 0.0;

// There are three separate pay-off sums for the final price
double payoff_sum_p = 0.0;
double payoff_sum = 0.0;
double payoff_sum_m = 0.0;

// Loop over the number of simulations
for (int i=0; i<num_sims; i++) {
  double gauss_bm = gaussian_box_muller(); // Random gaussian draw

  // Adjust three stock paths
  double expgauss = exp(sqrt(v*v*T)*gauss_bm);  // Precalculate
  Sp_cur = Sp_adjust * expgauss;
  S_cur = S_adjust * expgauss;
  Sm_cur = Sm_adjust * expgauss;

  // Calculate the continual pay-off sum for each increment/decrement
  payoff_sum_p += std::max(Sp_cur - K, 0.0);
  payoff_sum += std::max(S_cur - K, 0.0);
  payoff_sum_m += std::max(Sm_cur - K, 0.0);
}
```

```
  // There are three separate prices
  price_Sp = (payoff_sum_p / static_cast<double>(num_sims)) * exp(-r*T);
  price_S = (payoff_sum / static_cast<double>(num_sims)) * exp(-r*T);
  price_Sm = (payoff_sum_m / static_cast<double>(num_sims)) * exp(-r*T);
}

double call_delta_mc(const int num_sims, const double S, const double K,
    const double r, const double v, const double T, const double delta_S) {
  // These values will be populated via the monte_carlo_call_price function
      .
  // They represent the incremented Sp (S+delta_S), non-incremented S (S)
      and
  // decremented Sm (S-delta_S) prices.
  double price_Sp = 0.0;
  double price_S = 0.0;
  double price_Sm = 0.0;

  // Call the Monte Carlo pricer for each of the three stock paths
  // (We only need two for the Delta)
  monte_carlo_call_price(num_sims, S, K, r, v, T, delta_S, price_Sp,
      price_S, price_Sm);
  return (price_Sp - price_S)/delta_S;
}

double call_gamma_mc(const int num_sims, const double S, const double K,
    const double r, const double v, const double T, const double delta_S) {
  // These values will be populated via the monte_carlo_call_price function
      .
  // They represent the incremented Sp (S+delta_S), non-incremented S (S)
      and
  // decremented Sm (S-delta_S) prices.
  double price_Sp = 0.0;
  double price_S = 0.0;
  double price_Sm = 0.0;

  // Call the Monte Carlo pricer for each of the three stock paths
  // (We need all three for the Gamma)
  monte_carlo_call_price(num_sims, S, K, r, v, T, delta_S, price_Sp,
      price_S, price_Sm);
  return (price_Sp - 2*price_S + price_Sm)/(delta_S*delta_S);
}

int main(int argc, char **argv) {
  // First we create the parameter list
  double S = 100.0;            // Option price
  double delta_S = 0.001;      // Option price increment
  double K = 100.0;            // Strike price
  double r = 0.05;             // Risk-free rate (5%)
  double v = 0.2;              // Volatility of the underlying (20%)
  double T = 1.0;              // One year until expiry
  int num_sims = 100000000;    // Number of simulations to carry out for
      Monte Carlo

  // Then we calculate the Delta and the Gamma for the call
  double call_delta_m = call_delta_mc(num_sims, S, K, r, v, T, delta_S);
  double call_gamma_m = call_gamma_mc(num_sims, S, K, r, v, T, delta_S);

  // Finally we output the parameters and greeks
  std::cout << "Number of sims:    " << num_sims << std::endl;
  std::cout << "Underlying:        " << S << std::endl;
  std::cout << "Delta underlying:  " << delta_S << std::endl;
  std::cout << "Strike:            " << K << std::endl;
  std::cout << "Risk-Free Rate:    " << r << std::endl;
  std::cout << "Volatility:        " << v << std::endl;
  std::cout << "Maturity:          " << T << std::endl << std::endl;

  std::cout << "Call Delta:        " << call_delta_m << std::endl;
  std::cout << "Call Gamma:        " << call_gamma_m << std::endl;
}
```

Here is the output of the program:

```
Number of sims:    100000000
Underlying:        100
```

	Delta	Gamma
Analytic	0.636831	0.018762
FDM/Analytic	0.636845	0.0187633
FDM/MC	0.636894	0.0188733

```
Delta underlying:  0.001
Strike:            100
Risk-Free Rate:    0.05
Volatility:        0.2
Maturity:          1

Call Delta:        0.636894
Call Gamma:        0.0188733
```

Here is a summary table with the calculation of the Delta and Gamma for a European vanilla call option across all of the methods listed above:

The values are extremely similar, even for the Monte Carlo based approach, albeit at the computational expense of generating the random draws. The FDM/MC approach can be applied to other contingent claims where an analytical solution is not forthcoming and this is often the method used in practice.

Chapter 12

Asian/Path-Dependent Options with Monte Carlo

In this chapter I'm going to discuss how to price a certain type of *Exotic* option known as a *Path-Dependent Asian* in C++ using Monte Carlo Methods. It is considered "exotic" in the sense that the pay-off is a function of the underlying asset at multiple points throughout its lifetime, rather than just the value at expiry. It is an example of a *multi look option.* An Asian option utilises the mean of the underlying asset price sampled at appropriate intervals along its lifetime as the basis for its pay-off, which is where the "path-dependency" of the asset comes from. The name actually arises because they were first devised in 1987 in Tokyo as options on crude oil futures.

There are two types of Asian option that we will be pricing. They are the *arithmetic Asian* and the *geometric Asian.* They differ only in how the mean of the underlying values is calculated. We will be studying discrete Asian options in this chapter, as opposed to the theoretical continuous Asian options. The first step will be to break down the components of the program and construct a set of classes that represent the various aspects of the pricer. Before that however, I will brief explain how Asian options work (and are priced by Monte Carlo).

12.1 Overview of Asian Options

An Asian option is a type of *exotic* option. Unlike a vanilla European option where the price of the option is dependent upon the price of the underlying asset at expiry, an Asian option pay-off is a function of multiple points up to and including the price at expiry. Thus it is *path-dependent* as the price relies on knowing how the underlying behaved at certain points *before* expiry. Asian options in particular base their price off the *mean average* price of these sampled points. To simplify this chapter we will consider N equally distributed sample points beginning at time $t = 0$ and ending at maturity, T.

Unlike in the vanilla European option Monte Carlo case, where we only needed to generate multiple spot values at expiry, we now need to generate multiple asset *paths*, each sampled at multiple equally-spaced points in time. Thus instead of providing a **double** value representing spot to our option, we now need to provide a std::vector<**double**> (i.e. a vector of double values), each element of which represents a sample of the asset price on a particular path. We will still be modelling our asset price path via a Geometric Brownian Motion (GBM), and we will create each path by adding the correct drift and variance at each step in order to maintain the properties of GBM.

In order to calculate the arithmetic mean A of the asset prices we use the following formula:

$$A(0,T) = \frac{1}{N}\sum_{i=1}^{N} S(t_i)$$

where $S(t_i)$ is the underlying asset price at the i-th time sample. Similarly for the geometric mean:

$$A(0,T) = \exp\left(\frac{1}{N}\sum_{i=1}^{N}\log(S(t_i))\right)$$

These two formulae will then form the basis of our pay_off_price method, which is attached to our AsianOption class.

12.2 Asian Options - An Object Oriented Approach

In order to increase maintainability we will separate the components of the Asian options pricer. As mentioned in the chapter on option pay-off hierarchies we are able to create an *abstract base class* called PayOff, which defines an interface that all subsequent inherited pay-off classes will implement. The major benefit of this approach is that we can encapsulate multiple various types of pay-off functionality without the need to modify the remaining classes, such as our AsianOption class (to be discussed below). We make use of the **operator**() to turn our pay-off classes into a *functor* (i.e. a function object). This means that we can "call" the object just as we would a function. "Calling" a PayOff object has the effect of calculating the pay-off and returning it.

Here is the declaration for the PayOff base class:

```
class PayOff {
 public:
  PayOff(); // Default (no parameter) constructor
  virtual ~PayOff() {}; // Virtual destructor

  // Overloaded () operator, turns the PayOff into an abstract function
      object
  virtual double operator() (const double& S) const = 0;
};
```

The second class will represent many aspects of the exotic path-dependent Asian option. This class will be called AsianOption. It will once again be an abstract base class, which means that it contains at least one *pure virtual function*. Since there are many types of Asian option - arithmetic, geometric, continuous, discrete - we will once again make use of an inheritance hierarchy. In particular, we will override the pay_off_price function, which determines how the mean pay-off is to be calculated, once the appropriate PayOff object has been provided.

Here is the declaration for the AsianOption base class:

```
class AsianOption {
 protected:
  PayOff* pay_off;  // Pointer to pay-off class (in this instance call or
      put)

 public:
  AsianOption(PayOff* _pay_off);
  virtual ~AsianOption() {};

  // Pure virtual pay-off operator (this will determine arithmetic or
      geometric)
  virtual double pay_off_price(const std::vector<double>& spot_prices)
      const = 0;
};
```

In C++ there is usually a trade-off between simplicity and extensibility. More often than not a program must become more complex if it is to be extendable elsewhere. Thus we have the choice of creating a separate object to handle the path generation used by the Asian option or write it procedurally. In this instance I have elected to use a procedural approach because I don't feel that delving into the complexities of random number generation classes has a beneficial effect on learning how Asian options are priced *at this stage*.

12.3 Pay-Off Classes

The first class we will consider is the PayOff. As stated above this is an abstract base class and so can never be instantiated directly. Instead it provides an *interface* through which all inherited classes will be bound to. The destructor is **virtual** to avoid memory leaks when the inherited and base classes are destroyed. We'll take a look at the full listing for the payoff.h header file and then step through the important sections:

```
#ifndef __PAY_OFF_H
#define __PAY_OFF_H

#include <algorithm> // This is needed for the std::max comparison function

class PayOff {
 public:
  PayOff(); // Default (no parameter) constructor
  virtual ~PayOff() {}; // Virtual destructor

  // Overloaded () operator, turns the PayOff into an abstract function
      object
  virtual double operator() (const double& S) const = 0;
};

class PayOffCall : public PayOff {
 private:
  double K; // Strike price

 public:
  PayOffCall(const double& K_);
  virtual ~PayOffCall() {};

  // Virtual function is now over-ridden (not pure-virtual anymore)
  virtual double operator() (const double& S) const;
};

class PayOffPut : public PayOff {
 private:
  double K; // Strike

 public:
  PayOffPut(const double& K_);
  virtual ~PayOffPut() {};
  virtual double operator() (const double& S) const;
};

#endif
```

The declaration for the PayOff class is straightforward except for the overload of **operator**(). The syntax says that this method cannot be implemented within this class (= 0) and that it should be overridden by an inherited class (**virtual**). It is also a **const** method as it does not modify anything. It simply takes a spot price S and returns the option pay-off, for a given strike, K:

```
// Overloaded () operator, turns the PayOff into an abstract function
    object
virtual double operator() (const double& S) const = 0;
```

Beyond the PayOff class we implement the PayOffCall and PayOffPut classes which implement call and put functionality. We could also introduce digital and double digital pay-off classes here, but for the purposes of demonstrating Asian options, I will leave that as an exercise! Notice now that in each of the declarations of the **operator**() the "= 0" syntax of the pure virtual declaration has been removed. This states that the methods should be implemented by these classes:

```
// Virtual function is now over-ridden (not pure-virtual anymore)
virtual double operator() (const double& S) const;
```

That sums up the header file for the PayOff class hierarchy. We will now look at the source file (below) and then step through the important sections:

```
#ifndef __PAY_OFF_CPP
#define __PAY_OFF_CPP

```

```
#include "payoff.h"

PayOff::PayOff() {}

// ==========
// PayOffCall
// ==========

// Constructor with single strike parameter
PayOffCall::PayOffCall(const double& _K) { K = _K; }

// Over-ridden operator() method, which turns PayOffCall into a function
    object
double PayOffCall::operator() (const double& S) const {
  return std::max(S-K, 0.0); // Standard European call pay-off
}

// =========
// PayOffPut
// =========

// Constructor with single strike parameter
PayOffPut::PayOffPut(const double& _K) {
  K = _K;
}

// Over-ridden operator() method, which turns PayOffPut into a function
    object
double PayOffPut::operator() (const double& S) const {
  return std::max(K-S, 0.0); // Standard European put pay-off
}

#endif
```

The only major point of note within the source file is the implementation of the call and put **operator**() methods. They make use of the std::max function found within the <algorithm> standard library:

```
// Over-ridden operator() method, which turns PayOffCall into a function
    object
double PayOffCall::operator() (const double& S) const {
  return std::max(S-K, 0.0); // Standard European call pay-off
}

..

// Over-ridden operator() method, which turns PayOffPut into a function
    object
double PayOffPut::operator() (const double& S) const {
  return std::max(K-S, 0.0); // Standard European put pay-off
}
```

The concludes the PayOff class hierarchy. It is quite straightforward and we're not using any real advanced features beyond abstract base classes and pure virtual functions. However, we have written quite a lot of code to encapsulate something seemingly as simple as a simple pay-off function. The benefits of such an approach will become clear later on, when or if we decide we wish to create more complex pay-offs as we will not need to modify our AsianOption class to do so.

12.4 Path Generation Header

To generate the paths necessary for an Asian option we will use a procedural approach. Instead of encapsulating the random number generator and path generator in a set of objects, we will make use of two functions, one of which generates the random Gaussian numbers (via the Box-Muller method) and the other which takes these numbers and generates sampled Geometric Brownian Motion asset paths for use in the AsianOption object. The chapter on European option pricing via Monte Carlo explains the concept of risk-neutral pricing, Monte Carlo techniques and the Box-Muller method. I have included the full function in the listing below for completeness.

The second function, calc_path_spot_prices is more relevant. It takes a reference to a vector of doubles (std::vector<double>&) and updates it to reflect a random asset path based on a set of random Gaussian increments of the asset price. I will show the full listing for the header file which contains these functions, then I will step through calc_path_spot_prices in detail:

```
#ifndef __PATH_GENERATION_H
#define __PATH_GENERATION_H
```

```cpp
#include <vector>
#include <cmath>

// For random Gaussian generation
// Note that there are many ways of doing this, but we will
// be using the Box-Muller method for demonstration purposes
double gaussian_box_muller() {
  double x = 0.0;
  double y = 0.0;
  double euclid_sq = 0.0;

  do {
    x = 2.0 * rand() / static_cast<double>(RAND_MAX)-1;
    y = 2.0 * rand() / static_cast<double>(RAND_MAX)-1;
    euclid_sq = x*x + y*y;
  } while (euclid_sq >= 1.0);

  return x*sqrt(-2*log(euclid_sq)/euclid_sq);
}

// This provides a vector containing sampled points of a
// Geometric Brownian Motion stock price path
void calc_path_spot_prices(std::vector<double>& spot_prices,  // Asset path
                           const double& r,   // Risk free interest rate
                           const double& v,   // Volatility of underlying
                           const double& T) { // Expiry
  // Since the drift and volatility of the asset are constant
  // we will precalculate as much as possible for maximum efficiency
  double dt = T/static_cast<double>(spot_prices.size());
  double drift = exp(dt*(r-0.5*v*v));
  double vol = sqrt(v*v*dt);

  for (int i=1; i<spot_prices.size(); i++) {
    double gauss_bm = gaussian_box_muller();
    spot_prices[i] = spot_prices[i-1] * drift * exp(vol*gauss_bm);
  }
}

#endif
```

Turning our attention to calc_path_spot_prices we can see that the function requires a vector of doubles, the risk-free rate r, the volatility of the underlying σ and the time at expiry, T:

```
void calc_path_spot_prices(std::vector<double>& spot_prices,  // Asset path
                           const double& r,   // Risk free interest rate (
                               constant)
                           const double& v,   // Volatility of underlying (
                               constant)
                           const double& T) { // Expiry
```

Since we are dealing with constant increments of time for our path sampling frequency we need to calculate this increment. These increments are always identical so in actual fact it can be pre-calculated outside of the loop for the spot price path generation. Similarly, via the properties of Geometric Brownian Motion, we know that we can increment the drift and variance of the asset in a manner which can be pre-computed. The only difference between this increment and the European case is that we are replacing T with dt for each subsequent increment of the path. See the European option pricing chapter for a comparison:

```
double dt = T/static_cast<double>(spot_prices.size());
double drift = exp(dt*(r-0.5*v*v));
double vol = sqrt(v*v*dt);
```

The final part of the function calculates the new spot prices by iterating over the spot_price vector and adding the drift and variance to each piece. We are using the arithmetic of logarithms here, and thus can *multiply* by our drift and variance terms, since it is the *log of the asset price* that is subject to normally distributed increments in Geometric Brownian Motion. Notice that the loop runs from $i = 1$, not $i = 0$. This is because the spot_price vector is already pre-filled with S, the initial asset price $S(0)$, elsewhere in the program:

```
for (int i=1; i<spot_prices.size(); i++) {
  double gauss_bm = gaussian_box_muller();
  spot_prices[i] = spot_prices[i-1] * drift * exp(vol*gauss_bm);
}
```

That concludes the path-generation header file. Now we'll take a look at the AsianOption classes themselves.

12.5 Asian Option Classes

The final component of our program (besides the main file of course!) is the Asian option inheritance hierarchy. We wish to price multiple types of Asian option, including geometric Asian options and arithmetic Asian options. One way to achieve this is to have separate methods on an AsianOption class. However this comes at the price of having to continually add more methods if we wish to make more granular changes to our AsianOption class. In a production environment this would become unwieldy. Instead we can use the abstract base class approach and generate an abstract AsianOption class with a pure virtual method for pay_off_price. This method is implemented in subclasses and determines how the averaging procedure for the asset prices over the asset path lifetime will occur. Two publicly-inherited subclasses AsianOptionArithmetic and AsianOptionGeometric implement this method.

Let's take a look at the full listing for the header declaration file and then we'll step through the interesting sections:

```
#ifndef __ASIAN_H
#define __ASIAN_H

#include <vector>
#include "payoff.h"

class AsianOption {
 protected:
  PayOff* pay_off;  // Pay-off class (in this instance call or put)

 public:
  AsianOption(PayOff* _pay_off);
  virtual ~AsianOption() {};

  // Pure virtual pay-off operator (this will determine arithmetic or
      geometric)
  virtual double pay_off_price(const std::vector<double>& spot_prices)
      const = 0;
};

class AsianOptionArithmetic : public AsianOption {
 public:
  AsianOptionArithmetic(PayOff* _pay_off);
  virtual ~AsianOptionArithmetic() {};

  // Override the pure virtual function to produce arithmetic Asian Options
  virtual double pay_off_price(const std::vector<double>& spot_prices)
      const;
};

class AsianOptionGeometric : public AsianOption {
 public:
  AsianOptionGeometric(PayOff* _pay_off);
  virtual ~AsianOptionGeometric() {};

  // Overide the pure virtual function to produce geometric Asian Options
  virtual double pay_off_price(const std::vector<double>& spot_prices)
      const;
};

#endif
```

The first thing to notice about the abstract AsianOption class is that it has a pointer to a PayOff class as a protected member:

```
protected:
 PayOff* pay_off;  // Pay-off class (in this instance call or put)
```

This is an example of *polymorphism*. The object will not know what type of PayOff class will be passed in. It could be a PayOffCall or a PayOffPut. Thus we can use a pointer to the PayOff abstract class to represent "storage" of this as-yet-unknown PayOff class. Note also that the constructor takes this as its only parameter:

```
public:
 AsianOption(PayOff* _pay_off);
```

Finally we have the declaration for the pure virtual pay_off_price method. This takes a vector of spot prices, but notice that it is passed as a reference to **const**, so this vector will not be modified within the method:

```
// Pure virtual pay-off operator (this will determine arithmetic or
    geometric)
```

```
virtual double pay_off_price(const std::vector<double>& spot_prices) const
    = 0;
```

The listings for the AsianOptionArithmetic and AsianOptionGeometric classes are analogous to those in the PayOff hierarchy, with the exception that their constructors take a pointer to a PayOff object. That concludes the header file.

The source file essentially implements the two pay_off_price methods for the inherited subclasses of AsianOption:

```
#ifndef __ASIAN_CPP
#define __ASIAN_CPP

#include <numeric>  // Necessary for std::accumulate
#include <cmath>  // For log/exp functions
#include "asian.h"

// =====================
// AsianOptionArithmetic
// =====================

AsianOption::AsianOption(PayOff* _pay_off) : pay_off(_pay_off) {}

// =====================
// AsianOptionArithmetic
// =====================

AsianOptionArithmetic::AsianOptionArithmetic(PayOff* _pay_off)
    : AsianOption(_pay_off) {}

// Arithmetic mean pay-off price
double AsianOptionArithmetic::pay_off_price(
    const std::vector<double>& spot_prices) const {
  unsigned num_times = spot_prices.size();
  double sum = std::accumulate(spot_prices.begin(), spot_prices.end(), 0);
  double arith_mean = sum / static_cast<double>(num_times);
  return (*pay_off)(arith_mean);
}
```

```
// ====================
// AsianOptionGeometric
// ====================

AsianOptionGeometric::AsianOptionGeometric(PayOff* _pay_off)
    : AsianOption(_pay_off) {}

// Geometric mean pay-off price
double AsianOptionGeometric::pay_off_price(
    const std::vector<double>& spot_prices) const {
  unsigned num_times = spot_prices.size();
  double log_sum = 0.0;
  for (int i=0; i<spot_prices.size(); i++) {
    log_sum += log(spot_prices[i]);
  }
  double geom_mean = exp(log_sum / static_cast<double>(num_times) );
  return (*pay_off)(geom_mean);
}

#endif
```

Let's take a look at the arithmetic option version. First of all we determine the number of sample points via the size of the spot_price vector. Then we use the std::accumulate algorithm and iterator syntax to sum the spot values in the vector. Finally we take the arithmetic mean of those values and use pointer dereferencing to call the **operator**() for the PayOff object. For this program it will provide a call or a put pay-off function for the average of the spot prices:

```
double AsianOptionArithmetic::pay_off_price(
    const std::vector<double>& spot_prices) const {
  unsigned num_times = spot_prices.size();
  double sum = std::accumulate(spot_prices.begin(), spot_prices.end(), 0);
  double arith_mean = sum / static_cast<double>(num_times);
  return (*pay_off)(arith_mean);
}
```

The geometric Asian is similar. We once again determine the number of spot prices. Then we loop over the spot prices, summing the logarithm of each of them and adding it to the grand total. Then we take the geometric mean of these values and finally use pointer dereferencing once again to determine the correct call/put pay-off value:

```
double AsianOptionGeometric::pay_off_price(
    const std::vector<double>& spot_prices) const {
  unsigned num_times = spot_prices.size();
  double log_sum = 0.0;
  for (int i=0; i<spot_prices.size(); i++) {
    log_sum += log(spot_prices[i]);
  }
  double geom_mean = exp(log_sum / static_cast<double>(num_times) );
  return (*pay_off)(geom_mean);
}
```

Note here what the AsianOption classes *do not* require. Firstly, they don't require information about the underlying (i.e. vol). They also don't require time to expiry or the interest rate. Thus we are really trying to encapsulate the *term sheet* of the option in this object, i.e. all of the parameters that would appear on the contract when the option is made. However, for simplicity we have neglected to include the actual *sample times*, which would also be written on the contract. This instead is moved to the path-generator. However, later code will amend this, particularly as we can re-use the AsianOption objects in more sophisticated programs, where interest rates and volatility are subject to stochastic models.

12.6 The Main Program

The final part of the program is the main.cpp file. It brings all of the previous components together to produce an output for the option price based on some default parameters. The full listing is below:

```
#include <iostream>

#include "payoff.h"
#include "asian.h"
#include "path_generate.h"

int main(int argc, char **argv) {
  // First we create the parameter list
  // Note that you could easily modify this code to input the parameters
  // either from the command line or via a file
```

```
unsigned num_sims = 100000;   // Number of simulated asset paths
unsigned num_intervals = 250;  // Number of asset sample intervals
double S = 30.0;  // Option price
double K = 29.0;  // Strike price
double r = 0.08;   // Risk-free rate (8%)
double v = 0.3;    // Volatility of the underlying (30%)
double T = 1.00;    // One year until expiry
std::vector<double> spot_prices(num_intervals, S);  // Vector of spot
    prices

// Create the PayOff objects
PayOff* pay_off_call = new PayOffCall(K);

// Create the AsianOption objects
AsianOptionArithmetic asian(pay_off_call);

// Update the spot price vector with correct
// spot price paths at constant intervals
double payoff_sum = 0.0;
for (int i=0; i<num_sims; i++) {
  calc_path_spot_prices(spot_prices, r, v, T);
  payoff_sum += asian.pay_off_price(spot_prices);
}
double discount_payoff_avg = (payoff_sum / static_cast<double>(num_sims))
    * exp(-r*T);

delete pay_off_call;

// Finally we output the parameters and prices
std::cout << "Number␣of␣Paths:␣" << num_sims << std::endl;
std::cout << "Number␣of␣Ints:␣␣" << num_intervals << std::endl;
std::cout << "Underlying:␣␣␣␣␣␣" << S << std::endl;
std::cout << "Strike:␣␣␣␣␣␣␣␣␣␣" << K << std::endl;
std::cout << "Risk-Free␣Rate:␣␣" << r << std::endl;
std::cout << "Volatility:␣␣␣␣␣␣" << v << std::endl;
std::cout << "Maturity:␣␣␣␣␣␣␣␣" << T << std::endl;
```

```
  std::cout << "Asian Price:     " << discount_payoff_avg << std::endl;

  return 0;
}
```

The first interesting aspect of the main.cpp program is that we have now added a **unsigned** num_intervals = 250; line which determines how frequently the spot price will be sampled in the Asian option. As stated above, this would usually be incorporated into the option *term sheet*, but I have included it here instead to help make the pricer easier to understand without too much object communication overhead. We have also created the vector of spot prices, which are pre-filled with the default spot price, S:

```
unsigned num_intervals = 250;  // Number of asset sample intervals
..
std::vector<double> spot_prices(num_intervals, S);  // The vector of spot
    prices
```

Then we create a PayOffCall object and assign it to a PayOff pointer. This allows us to leverage *polymorphism* and pass that object through to the AsianOption, without the option needing to know the actual type of PayOff. *Note that whenever we use the* ***new*** *operator, we must make use of the corresponding* ***delete*** *operator.*

```
PayOff* pay_off_call = new PayOffCall(K);
..
delete pay_off_call;
```

The next step is to create the AsianOptionArithmetic object. We could have as easily chosen the AsianOptionGeometric and the program would be trivial to modify to do so. It takes in the pointer to the PayOff as its lone constructor argument:

```
AsianOptionArithmetic asian(pay_off_call);
```

Then we create a loop for the total number of path simulations. In the loop we recalculate a new spot price path and then add that pay-off to a running sum of all pay-offs. The final step is to discount the average of this pay-off via the risk-free rate across the lifetime of the option ($T - 0 = T$). This discounted price is then the final price of the option, with the above parameters:

```
double payoff_sum = 0.0;
for (int i=0; i<num_sims; i++) {
  calc_path_spot_prices(spot_prices, r, v, T);
  payoff_sum += asian.pay_off_price(spot_prices);
}
double discount_payoff_avg = (payoff_sum / static_cast<double>(num_sims))
    * exp(-r*T);
```

The final stage of the main program is to output the parameters and the options price:

```
Number of Paths: 100000
Number of Ints:  250
Underlying:      30
Strike:          29
Risk-Free Rate:  0.08
Volatility:      0.3
Maturity:        1
Asian Price:     2.85425
```

The benefits of such an object-oriented approach are now clear. We can easily add more PayOff or AsianOption classes without needing to extensively modify any of the remaining code. These objects are said to have a *separation of concerns*, which is exactly what is needed for large-scale software projects.

Chapter 13

Implied Volatility

In this chapter we will discuss a practical application of the Black-Scholes model, design patterns and function objects in C++. In particular, we are going to consider the concept of **Implied Volatility**. In derivatives pricing, the implied volatility of an option is the value of the underlyings volatility (usually denoted by σ), which when input into an derivatives pricing model (such as with the Black-Scholes equation) provides a pricing value for the option which is equal to the currently observed market price of that option.

13.1 Motivation

The use of implied volatility is motivated by the fact that it is often more useful as a measure of *relative value* between options than the actual price of those options. This is because the option price is heavily dependent upon the price of the underlying asset. Options are often held in a *delta neutral portfolio*, which means that the portfolio is hedged for small moves in the price of the underlying asset. Given the need to continuously rebalance these portfolios, an option with a higher actual price can be *cheaper in terms of its volatility* if the underlying itself rises, as the underlying asset can be sold at a higher price.

Implied volatilities can in fact be considered a form of "prices", as their values have been determined by actual transactions in the marketplace, not on the basis of statistical estimates. Professional traders actually tend to quote values of options in terms of their "vol", rather than their actual market prices.

Our task for this chapter is to attempt to calculate implied volatilities using the Black-Scholes equation. If we let the market price of an option be given by C_M and the Black-Scholes function by $B(S, K, r, T, \sigma)$, where S is the underlying price, K is the option strike, r is the risk-free interest rate, T is the time to maturity, then we are trying to find the value of σ such that:

$$B(S, K, r, T, \sigma) = C_M \tag{13.1}$$

13.2 Root-Finding Algorithms

The first place to start is to consider whether an analytical inverse exists. Unfortunately, the Black-Scholes model does not lend itself to an analytical inverse and thus we are forced to carry out the computation using numerical methods. The common methods for achieving these are known as *root finding algorithms*. In this chapter we will look at two techniques: Interval Bisection and Newton-Raphson.

13.3 Interval Bisection Method

The first numerical algorithm considered is Interval Bisection. It makes use of a real analysis concept known as the **intermediate value theorem**. Since the Black-Scholes formula is continuous (and "well behaved", which for us means sufficiently smooth), we can make use of the theorem to help us find a volatility.

The theorem states, mathematically, that if $\exists m, n \in \mathbb{R}$, and $g(x) : \mathbb{R} \rightarrow \mathbb{R}$, continuous, such that $g(m) < y$, $g(n) > y$, then $\exists p \in (m, n) \subset \mathbb{R}$ such that $g(p) = y$. For us, g is the Black-Scholes function, y is the current market price and p is the volatility.

Heuristically, the theorem states that if a function is continuous and we know two (differing) values in the domain that generate an image interval, then there must exist a value in the domain between these two values that provides a mapped value lying within the image domain. This is exactly what we need in order to determine the implied volatility from the market prices.

Interval bisection is quite straightforward to understand. It is a "trial and error" algorithm. We pick the mid-point value, c, of an interval, and then either $g(c) = y$, $g(c) < y$ or $g(c) > y$. In the first instance the algorithm terminates. In the latter two cases, we subdivide the interval (c, n) (respectively (m, c)) and find the corresponding midpoints. At each level of the recursion, the interval size is halved. In practice the algorithm is terminated using a tolerance value, ϵ, once $|g(c) - y| < \epsilon$.

Interval bisection is not an efficient method for root-finding, but it is straightforward to implement, especially when we make use of *functors* and *function templates*.

13.3.1 Interval Bisection with Function Templates

So far we've made extensive use of object-orientation and, in particular, inheritance hierarchies as one of the main design patterns for solving quantitative finance problems. We're now going to use a very different set of tools, that of *functors* and *function templates*. We've discussed how functors work before and have used them to create PayOff classes. Now we are going to supplement their use by including function templates, which are a means of applying the template generic programming paradigm that we have previously applied to classes, to functions themselves.

The motivation for using functors and function templates is that we wish to create our interval bisection code in a *reusable fashion*. Ideally, the interval bisection code should be able to cope with a generic function, including those with their own set of (fixed) parameters. This is the situation we are in, because the Black-Scholes formula requires not only the volatility, but the strike, spot price, maturity time and interest rate.

Creating the Black-Scholes call option as a functor allows us to pass it to the interval bisection function with *attached state*, which in this case is the collection of extra arguments representing the option parameters. In fact, the interval bisection method doesn't even need to know about these parameters, as the only interface exposed to it is an option pricing **operator**() method, which accepts only a volatility.

Another reason to use function templatisation over inheritance is that we would be at the mercy of virtual functions. Virtual functions are relatively slow, as each time a function is called the *vtable* has to be queried, which is extra overhead. Further, virtual functions cannot be inlined, which is a major disadvantage. This leads to slower code. These problems also apply to function pointers as well, which is why we aren't making use of them either.

The approach we will follow is to make the interval_bisection function a template function, such that it can accept a generic function object, which in our case will be the Black-Scholes call pricing method. We'll now discuss the C++ implementation.

13.3.2 C++ Implementation

We've previously considered analytical pricing of European call options in some depth. However, I've reproduced and modified the code here for completeness (bs_prices.h):

```
#ifndef __BS_PRICES_H
#define __BS_PRICES_H

```

```
#define _USE_MATH_DEFINES

#include <iostream>
#include <cmath>

// Standard normal probability density function
double norm_pdf(const double x) {
  return (1.0/(pow(2*M_PI,0.5)))*exp(-0.5*x*x);
}

// An approximation to the cumulative distribution function
// for the standard normal distribution
// Note: This is a recursive function
double norm_cdf(const double x) {
  double k = 1.0/(1.0 + 0.2316419*x);
  double k_sum = k*(0.319381530 + k*(-0.356563782 + k*(1.781477937 +
    k*(-1.821255978 + 1.330274429*k))));

  if (x >= 0.0) {
    return (1.0 - (1.0/(pow(2*M_PI,0.5)))*exp(-0.5*x*x) * k_sum);
  } else {
    return 1.0 - norm_cdf(-x);
  }
}

// This calculates d_j, for j in {1,2}. This term appears in the closed
// form solution for the European call or put price
double d_j(const int j, const double S, const double K, const double r,
  const double sigma, const double T) {
  return (log(S/K) + (r + (pow(-1,j-1))*0.5*sigma*sigma)*T)/(sigma*(pow(T
      ,0.5)));
}

// Calculate the European vanilla call price based on
// underlying S, strike K, risk-free rate r, volatility of
// underlying sigma and time to maturity T
double call_price(const double S, const double K, const double r,
  const double sigma, const double T) {
  return S * norm_cdf(d_j(1, S, K, r, sigma, T))-K*exp(-r*T) *
    norm_cdf(d_j(2, S, K, r, sigma, T));
}

#endif
```

The following is a listing for black_scholes.h, which contains a basic function object (functor) for handling calculation of options prices when provided with a volatility, σ. Notice that all of the Black-Scholes model option paramaters are stored as private variables, with the exception of the volatility σ. This is because the function call **operator**() method takes it as a parameter. This method will eventually be (repeatedly) called by the interval_bisection function:

```
#ifndef __BLACK_SCHOLES_H
#define __BLACK_SCHOLES_H

class BlackScholesCall {
private:
  double S;   // Underlying asset price
  double K;   // Strike price
  double r;   // Risk-free rate
  double T;   // Time to maturity

public:
  BlackScholesCall(double _S, double _K,
                   double _r, double _T);

  double operator()(double sigma) const;
};
#endif
```

The following is a listing for black_scholes.cpp, which contains the source implementation for the Black-Scholes options class. This class simply provides a wrapper around the analytical price for the call option, but crucially specifies the needed parameters, in such a way that the interval_bisection function avoids having to concern itself with them. Notice that S, K, r and T are referencing private member data, while σ is being passed from the method call as a parameter:

```
#ifndef __BLACK_SCHOLES_CPP
#define __BLACK_SCHOLES_CPP
```

```
#include "black_scholes.h"
#include "bs_prices.h"

BlackScholesCall::BlackScholesCall(double _S, double _K,
                                   double _r, double _T) :
  S(_S), K(_K), r(_r), T(_T) {}

double BlackScholesCall::operator()(double sigma) const {
  return call_price(S, K, r, sigma, T);
}

#endif
```

The following is a listing for interval_bisection.h, which contains the *function template* for carrying out interval bisection. As with classes, we need to add the **template**<**typename** T> syntax to the signature of the function in order to tell the compiler it should be expecting a generic type as one of its parameters. The function body implicitly calls **operator**() of the function object g, so any object passed to it must define **operator**() for a sole parameter.

The remainder of the code carries out the Interval Bisection algorithm for finding a root of the generic function g:

```
#ifndef __INTERVAL_BISECTION_H
#define __INTERVAL_BISECTION_H

#include <cmath>

// Creating a function template
// Trying to find an x such that |g(x) - y| < epsilon,
// starting with the interval (m, n).
template<typename T>
double interval_bisection(double y_target,  // Target y value
                          double m,         // Left interval value
                          double n,         // Right interval value
                          double epsilon,   // Tolerance
                          T g) {            // Function object of type T,
    named g
```

```
  // Create the initial x mid-point value
  // Find the mapped y value of g(x)
  double x = 0.5 * (m + n);
  double y = g(x);

  // While the difference between y and the y_target
  // value is greater than epsilon, keep subdividing
  // the interval into successively smaller halves
  // and re-evaluate the new y.
  do {
    if (y < y_target) {
      m = x;
    }

    if (y > y_target) {
      n = x;
    }

    x = 0.5 * (m + n);
    y = g(x);
  } while (fabs(y-y_target) > epsilon);

  return x;
}

#endif
```

The final listing is for the main implementation (main.cpp). I've hardcoded some option parameters (but you could easily modify this to input them from the command line), so that the implied volatility can be calculated. Firstly we create a BlackScholesCall instance and pass it the required parameters (without the volatility, at this stage). Then we define the parameters for the interval bisection itself, and finally pass the interval_bisection the bsc function object as a generic parameter. This then calculates the implied volatility of the option for us:

```
#ifndef __MAIN_CPP
#define __MAIN_CPP
```

```
#include "black_scholes.h"
#include "interval_bisection.h"
#include <iostream>

int main(int argc, char **argv) {
  // First we create the parameter list
  double S = 100.0;  // Underlying spot price
  double K = 100.0;  // Strike price
  double r = 0.05;   // Risk-free rate (5%)
  double T = 1.0;    // One year until expiry
  double C_M = 10.5; // Option market price

  // Create the Black-Scholes Call functor
  BlackScholesCall bsc(S, K, r, T);

  // Interval Bisection parameters
  double low_vol = 0.05;
  double high_vol = 0.35;
  double episilon = 0.001;

  // Calculate the implied volatility
  double sigma = interval_bisection(C_M, low_vol, high_vol, episilon, bsc);

  // Output the values
  std::cout << "Implied Vol: " << sigma << std::endl;

  return 0;
}
#endif
```

The output of the code is as follows:

```
Implied Vol: 0.201318
```

Thus we obtain an implied volatility of 20.1% for this particular call option. The object-oriented and generic aspects of the above code lend themselves naturally to extension and re-use.

13.4 Implied Volatility via Newton-Raphson

In the above section we made use of interval bisection to numerically solve for the implied volatility. In this section we are going to modify our code to make use of the Newton-Raphson process, which is more optimal for this problem domain than interval bisection.

For the previous calculation we made use of a *function template* to carry out the interval bisection. The template function accepted a function object (functor) of type `T`, g, which itself accepted a volatility parameter (σ) to provide an option price. The main design issue we have to contend with is that to use Newton-Raphson, we require a second function to represent g', the derivative of g (the *vega* of the option). There are multiple approaches to deal with this issue:

- **Two functors** - We can create a separate function object to represent the derivative of the function stored in the first functor. However, this method is not scalable if we wish to create second (or higher order) derivatives, or partial derivatives for any other variables. A derivative is also fundamentally a component of the original function, rather than a separate entity. Thus we won't be using this approach.

- **Derivative method** - It is possible to create a derivative method for the function object g. However, we still suffer from the same scaling issue in that we would need to create derivative methods for all orders or variables. We could create a generalised derivative function, but this would be a substantial amount of work for very little initial gain.

- **Pointer to member function** - To avoid the ugliness of the previous method, we can make use of a *pointer to a member function.* This method allows us to specify which function is to be called at the point of compilation, using templates. It is the approach taken in Joshi and we will follow it here.

Before considering the C++ implementation, we will briefly discuss how the Newton-Raphson root-finding algorithm works.

13.4.1 Newton-Raphson Method

Newton-Raphson is a more efficient algorithm for finding roots provided that some assumptions are met. In particular, g must possess an easily calculated derivative. If the derivative is analytically calculable, then this aids efficiency even further. g must also be 'well behaved', that is it cannot vary too wildly, else the derivative approximation made use of in the method becomes more innacurate.

In the case of the Black-Scholes formula, the derivative we seek is the derivative of the option price with respect to the volatility, $\frac{\partial B}{\partial \sigma}$. This is known as the **vega** of the option model. For a call option an analytical function exists for the vega, so we are in luck.

The idea of Newton-Raphson is to use the analytic derivative to make a linear estimate of where the solution should occur, which is much more accurate than the mid-point approach taken by Interval Bisection. Thus the starting approximation to g, g_0, is given by (where x_0 is our initial guess):

$$g_0(x) = g(x_0) + (x - x_0)g'(x_0) \tag{13.2}$$

This can be rearranged for x:

$$x = \frac{y - g(x_0)}{g'(x_0)} + x_0 \tag{13.3}$$

This then leads to a recurrence relation for more accurate values of x:

$$x_{n+1} = \frac{y - g(x_n)}{g'(x_n)} + x_n \tag{13.4}$$

As with interval bisection, the algorithm is terminated when $|g(x_n) - y| < \epsilon$, for some pre-specified tolerance ϵ.

Rewritten with our financial notation (where I have suppressed the other parameters for the Black-Scholes formula, B), the relation becomes:

$$\sigma_{n+1} = \frac{C_M - B(\sigma_n)}{\frac{\partial B}{\partial \sigma}(\sigma_n)} + \sigma_n \tag{13.5}$$

We can use this relation in order to find the implied volatility via a terminating criterion.

13.4.2 Pointer to a Member Function

The idea of a pointer to a member function is to restrict the usage of a function pointer to methods of a particular class. The scope resolution operator :: combined with the pointer dereference operator $*$ is used to carry this out. In order to define such a function pointer we use the following syntax (where g_prime represents g', the derivative of g):

```
double (T::*g_prime)(double) const
```

To invoke the method we use the following syntax:

```
(root_func.*g_prime)(x)
```

Where root_func is an object of type T, containing two methods g and g_prime, representing the *value* of the function and the *derivative value* of the function respectively that will be tested for roots.

One of the major benefits of this approach is that the calls to these function pointers can be *inlined* as they are treated as template parameters and so are evaluated at compile-time.

13.4.3 Implementing Newton-Raphson

In order to make use of the new approach, we need to add the explicit formula for a call option vega to our bs_prices.h header file and modify the original BlackScholesCall object we created in the previous tutorial to make use of it. Here is the added function to calculate the option vega:

```
..
..

// Calculate the European vanilla call vega 'Greek' based on
// underlying S, strike K, risk-free rate r, volatility of
// underlying sigma and time to maturity T
double call_vega(const double S, const double K, const double r,
    const double sigma, const double T) {
  return S * sqrt(T) * norm_pdf(d_j(1, S, K, r, sigma, T));
}

..
..
```

Here is the new header listing for black_scholes.h:

```
#ifndef __BLACK_SCHOLES_H
#define __BLACK_SCHOLES_H

class BlackScholesCall {
private:
  double S;  // Underlying asset price
  double K;  // Strike price
  double r;  // Risk-free rate
  double T;  // Time to maturity

public:
  BlackScholesCall(double _S, double _K,
                   double _r, double _T);

  // This is the modified section. operator()
  // has been replaced with option_price and
  // we have added option_vega (both const)
  double option_price(double sigma) const;
  double option_vega(double sigma) const;
};
#endif
```

The source listing has also been changed to include the implementation of option_vega, given in black_scholes.cpp:

```
#ifndef __BLACK_SCHOLES_CPP
#define __BLACK_SCHOLES_CPP

#include "black_scholes.h"
#include "bs_prices.h"

BlackScholesCall::BlackScholesCall(double _S, double _K,
                                   double _r, double _T) :
  S(_S), K(_K), r(_r), T(_T) {}

// Renamed from operator() to option_price()
double BlackScholesCall::option_price(double sigma) const {
  return call_price(S, K, r, sigma, T);
}

// New method added, which calls call_vega
// to obtain the actual price
double BlackScholesCall::option_vega(double sigma) const {
  return call_vega(S, K, r, sigma, T);
}

#endif
```

The next stage is to create the Newton-Raphson solver itself. The function template will accept an object of type T (the functor) and two pointers to member functions (methods) of T, g and g_prime. Here is the listing for newton_raphson.h:

```
#ifndef __NEWTON_RAPHSON_H
#define __NEWTON_RAPHSON_H

#include <cmath>

template<typename T,
         double (T::*g)(double) const,
         double (T::*g_prime)(double) const>
double newton_raphson(double y_target,        // Target y value
                      double init,            // Initial x value
                      double epsilon,         // Tolerance
                      const T& root_func) {   // Function objec
  // Set the initial option prices and volatility
  double y = (root_func.*g)(init);  // Initial option prices
  double x = init;                  // Initial volatility

  // While y and y_target are not similar enough
  // Take the vega of the option and recalculate
  // a new call price based on the best linear
  // approximation at that particular vol value
  while (fabs(y-y_target) > epsilon) {
    double d_x = (root_func.*g_prime)(x);
    x += (y_target-y)/d_x;
    y = (root_func.*g)(x);
  }
  return x;
}

#endif
```

Now we can create the main() function to wrap all of our code together:

```
#ifndef __MAIN_CPP
#define __MAIN_CPP

#include "black_scholes.h"
#include "newton_raphson.h"
#include <iostream>

int main(int argc, char **argv) {
  // First we create the parameter list
  double S = 100.0;  // Underlying spot price
  double K = 100.0;  // Strike price
  double r = 0.05;   // Risk-free rate (5%)
  double T = 1.0;    // One year until expiry
  double C_M = 10.5; // Option market price

  // Create the Black-Scholes Call functor
  BlackScholesCall bsc(S, K, r, T);

  // Newton Raphson parameters
  double init = 0.3;  // Our guess impl. vol of 30%
  double epsilon = 0.001;

  // Calculate the implied volatility
  double sigma = newton_raphson<BlackScholesCall,
                                &BlackScholesCall::option_price,
                                &BlackScholesCall::option_vega>
    (C_M, init, epsilon, bsc);

  // Output the values
  std::cout << "Implied Vol: " << sigma << std::endl;

  return 0;
}

#endif
```

The output of the code is given by:

```
Implied Vol: 0.201317
```

This matches the implied volatility given in the previous section, although the calculation was far more optimal.

Chapter 14

Random Number Generation and Statistical Distributions

In this chapter we are going to construct classes to help us encapsulate the generation of random numbers. Random number generators (RNG) are an essential tool in quantitative finance as they are necessary for Monte Carlo simulations that power numerical option pricing techniques. Other chapters have so far used RNGs in a procedural manner. In particular, we have utilised the Box-Muller technique to generate one or more random variables distributed as a standard Gaussian.

14.1 Overview

We will now show how to construct a random number generator class hierarchy. This allows us to separate the generation of random numbers from the Monte Carlo solvers that make use of them. It helps us reduce the amount of code we will need to write in the future, increases extensibility by allowing easy creation of additional random number generators and makes the code more maintainable.

There are further reasons to write our own random number generators:

- It allows us to make use of **pseudo-random numbers**. These are sequences of numbers that possess the correct statistical properties to "emulate" random numbers in order to improve the convergence rates of Monte Carlo simulations. The interface for random numbers and pseudo-random numbers is identical and we can hide away the details in the specific classes. In particular we can implement *low-discrepancy numbers* and *anti-thetic sampling* in this manner.

- Relying on the rand function provided with the C++ standard is **unreliable**. Not only is rand implementation specific, because it varies across multiple vendor compilers, but we are unaware of the efficiency of each implementation. This leads to difficulties in cross-platform testing as we cannot guarantee reproducibility.

- We are able to provide **multiple separate streams of random numbers** for different parts of our running program. The seed for the rand function, srand, is a global variable and hence will affect all components of our program, which is usually unwanted behaviour. By implementing our own RNG we avoid this issue.

14.2 Random Number Generator Class Hierarchy

Our random number generators will be formed from an *inheritance hierarchy*. We have already used this method when constructing PayOff classes for option pay-off functions. To form the hierarchy we will create an *abstract base class* that specifies the interface to the random number generator. All subsequent generators will inherit the interface from this class.

The primary considerations of this interface are as follows:

- Quantity or **dimension** of the generator: Many of the options pricers we have already created require more than a single random number in order to be accurately priced. This is the case for *path-dependent* options such as *Asians*, *Barriers* and *Lookbacks*. Thus our first consideration is to make sure that the generator provides a vector of random numbers, with the dimension specified at the creation of the instance.

- The supported **statistical distributions** from which to draw random variables: For options pricing, the two main statistical distributions of interest will be the *uniform distribution* and the *standard normal distribution* (i.e. the "Gaussian" distribution). Gaussian random draws are calculated from uniform random draws. We can use the statistical classes in order to obtain random draws from any particular distribution we wish, without modifying the RNG.

- We will need methods to support obtaining and setting the *random seed*, so that we can control which random numbers are generated and to ensure reproducibility across separate runs and platforms.

With those considerations in mind, let's create a simple abstract base class for our random number generator, in the file random.h:

```
#ifndef __RANDOM_H
#define __RANDOM_H

#include <vector>

class RandomNumberGenerator {
 protected:
  unsigned long init_seed;  // Initial random seed value
  unsigned long cur_seed;   // Current random seed value
  unsigned long num_draws;  // Dimensionality of the RNG

 public:
  RandomNumberGenerator(unsigned long _num_draws, unsigned long _init_seed)
    : num_draws(_num_draws), init_seed(_init_seed), cur_seed(_init_seed)
       {};
  virtual ~RandomNumberGenerator() {};

  virtual unsigned long get_random_seed() const { return cur_seed; }
  virtual void set_random_seed(unsigned long _seed) { cur_seed = _seed; }
  virtual void reset_random_seed() { cur_seed = init_seed; }
  virtual void set_num_draws(unsigned long _num_draws) { num_draws =
      _num_draws; }

  // Obtain a random integer (needed for creating random uniforms)
  virtual unsigned long get_random_integer() = 0;

  // Fills a vector with uniform random variables on the open interval
      (0,1)
  virtual void get_uniform_draws(std::vector<double>& draws) = 0;
};

#endif
```

Let's run through the code. Firstly, note that we have three **protected** member variables (which are all large **unsigned long** integers). cur_seed is the RNG current seed value. init_seed is the initial seed value, which does not change once the RNG has been instantiated. The current seed can only be reset to the initial seed. num_draws represents the *dimensionality* of the random number generator (i.e. how many random draws to create):

```
protected:
 unsigned long init_seed;  // Initial random seed value
 unsigned long cur_seed;   // Current random seed value
 unsigned long num_draws;  // Dimensionality of the RNG
```

Since we're creating an *abstract* base class is it a good idea to use **protected** data?

This is actually a contentious issue. Sometimes protected variables are frowned upon. Instead, it is argued that all data should be private and that accessor methods should be used. However, inherited classes -are- clients of the base class, just as "public" clients of the classes are. The alternative argument is that it is extremely convenient to use protected member data because it reduces the amount of cluttered accessor and modifier methods. For brevity I have used protected member data here.

Although the class will never be instantiated directly, it still has a constructor which must be called to populate the protected members. We use a member initialisation list to carry this out. We also create an empty method implementation for the constructor ({}), avoiding the need to create a random.cpp source file. Notice that we're setting the current seed to the initial seed as well.

```
RandomNumberGenerator(unsigned long _num_draws, unsigned long _init_seed)
    : num_draws(_num_draws), init_seed(_init_seed), cur_seed(_init_seed)
        {};
```

We then have four separate access and reset methods (all virtual), which get, set and reset the *random seed* and another which resets the number of random draws. They are all directly implemented in the header file, once again stopping us from needing to create a random.cpp source file:

```
virtual unsigned long get_random_seed() const { return cur_seed; }
virtual void set_random_seed(unsigned long _seed) { cur_seed = _seed; }
virtual void reset_random_seed() { cur_seed = init_seed; }
virtual void set_num_draws(unsigned long _num_draws) { num_draws =
    _num_draws; }
```

We now need a method to create a random integer. This is because subsequent random number generators will rely on transforming random **unsigned long**s into uniform variables on the open interval $(0, 1)$. The method is declared pure virtual as different RNGs will implement this differently. We don't want to "force" an approach on future clients of our code:

```
// Obtain a random integer (needed for creating random uniforms)
virtual unsigned long get_random_integer() = 0;
```

Finally we fill a supplied vector with uniform random draws. This vector will then be passed to a statistical distribution class in order to generate random draws from any chosen distribution that we implement. In this way we are completely separating the generation of the uniform random variables (on the open interval $(0, 1)$) and the draws from various statistical distributions. This maximises code re-use and aids testing:

```
// Fills a vector with uniform random variables on the open interval (0,1)
virtual void get_uniform_draws(std::vector<double>& draws) = 0;
```

Our next task is to implement a **linear congruential generator** algorithm as a means for creating our uniform random draws.

14.2.1 Linear Congruential Generators

Linear congruential generators (LCG) are a form of random number generator based on the following general recurrence relation:

$$x_{k+1} = g \cdot x_k mod n$$

Where n is a prime number (or power of a prime number), g has *high multiplicative order modulo* n and x_0 (the initial seed) is co-prime to n. Essentially, if g is chosen correctly, all integers from 1 to $n - 1$ will eventually appear in a periodic fashion. This is why LCGs are termed *pseudo-random*. Although they possess "enough" randomness for our needs (as n can be large), they are far from truly random. We won't dwell on the details of the mathematics behind LCGs, as we will not be making strong use of them going forward in our studies. However, most system-supplied RNGs make use of LCGs, so it is worth being aware of the algorithm. The listing below (lin_con_gen.cpp) contains the implementation of the algorithm. If you want to learn more about how LCGs work, take a look at Numerical Recipes[20].

14.2.2 Implementing a Linear Congruential Generator

With the mathematical algorithm described, it is straightforward to create the header file listing (lin_con_gen.h) for the Linear Congruential Generator. The LCG simply inherits from the RNG abstract base class, adds a **private** member variable called max_multiplier (used for pre-computing a specific ratio required in the uniform draw implementation) and implements the two pure virtual methods that were part of the RNG abstract base class:

```
#ifndef __LINEAR_CONGRUENTIAL_GENERATOR_H
#define __LINEAR_CONGRUENTIAL_GENERATOR_H

#include "random.h"

class LinearCongruentialGenerator : public RandomNumberGenerator {
 private:
  double max_multiplier;

 public:
  LinearCongruentialGenerator(unsigned long _num_draws,
                              unsigned long _init_seed = 1);
  virtual ~LinearCongruentialGenerator() {};

  virtual unsigned long get_random_integer();
  virtual void get_uniform_draws(std::vector<double> draws);
};

#endif
```

The source file (lin_con_gen.cpp) contains the implementation of the linear congruential generator algorithm. We make heavy use of *Numerical Recipes in C*[20], the famed numerical algorithms cookbook. The book itself is freely available online. I would strongly suggest reading the chapter on random number generator (Chapter 7) as it describes many of the pitfalls with using a basic linear congruential generator, which I do not have time to elucidate on in this chapter. Here is the listing in full:

```
#ifndef __LINEAR_CONGRUENTIAL_GENERATOR_CPP
#define __LINEAR_CONGRUENTIAL_GENERATOR_CPP

#include "lin_con_gen.h"
```

```
// This uses the Park & Miller algorithm found in "Numerical Recipes in C"
// Define the constants for the Park & Miller algorithm

const unsigned long a = 16807;        // 7^5
const unsigned long m = 2147483647;   // 2^32 - 1 (and thus prime)

// Schrage's algorithm constants

const unsigned long q = 127773;
const unsigned long r = 2836;

// Parameter constructor
LinearCongruentialGenerator::LinearCongruentialGenerator(
    unsigned long _num_draws,
    unsigned long _init_seed
) : RandomNumberGenerator(_num_draws, _init_seed) {

  if (_init_seed == 0) {
    init_seed = 1;
    cur_seed = 1;
  }

  max_multiplier = 1.0 / (1.0 + (m-1));
}

// Obtains a random unsigned long integer
unsigned long LinearCongruentialGenerator::get_random_integer() {
  unsigned long k = 0;
  k = cur_seed / q;
  cur_seed = a * (cur_seed - k * q) - r * k;

  if (cur_seed < 0) {
    cur_seed += m;
  }

  return cur_seed;
}

// Create a vector of uniform draws between (0,1)
void LinearCongruentialGenerator::get_uniform_draws(std::vector<double>&
    draws) {
  for (unsigned long i=0; i<num_draws; i++) {
    draws[i] = get_random_integer() * max_multiplier;
  }
}

#endif
```

Firstly, we set all of the necessary constants (see [20] for the explanation of the chosen values). *Note that if we created another LCG we could inherit from the RNG base class and use different constants*:

```
// Define the constants for the Park & Miller algorithm

const unsigned long a = 16807;        // 7^5
const unsigned long m = 2147483647;   // 2^32 - 1

// Schrage's algorithm constants

const unsigned long q = 127773;
const unsigned long r = 2836;
```

Secondly we implement the constructor for the LCG. If the seed is set to zero by the client, we set it to unity, as the LCG algorithm does not work with a seed of zero. The max_mutliplier is a pre-computed scaling factor necessary for converting a random **unsigned long** into a uniform value on on the open interval $(0, 1) \subset \mathbb{R}$:

```
// Parameter constructor
LinearCongruentialGenerator::LinearCongruentialGenerator(
    unsigned long _num_draws,
    unsigned long _init_seed
) : RandomNumberGenerator(_num_draws, _init_seed) {

  if (_init_seed == 0) {
    init_seed = 1;
    cur_seed = 1;
  }

  max_multiplier = 1.0 / (1.0 + (m-1));
}
```

We now concretely implement the two pure virtual functions of the RNG base class, namely get_random_integer and get_uniform_draws. get_random_integer applies the LCG modulus algorithm and modifies the current seed (as described in the algorithm above):

```
// Obtains a random unsigned long integer
unsigned long LinearCongruentialGenerator::get_random_integer() {
  unsigned long k = 0;
  k = cur_seed / q;
  cur_seed = a * (cur_seed - k * q) - r * k;

  if (cur_seed < 0) {
    cur_seed += m;
  }

  return cur_seed;
}
```

get_uniform_draws takes in a vector of the correct length (num_draws) and loops over it converting random integers generated by the LCG into uniform random variables on the interval $(0, 1)$:

```
// Create a vector of uniform draws between (0,1)
void LinearCongruentialGenerator::get_uniform_draws(std::vector<double>&
    draws) {
  for (unsigned long i=0; i<num_draws; i++) {
    draws[i] = get_random_integer() * max_multiplier;
  }
}
```

The concludes the implementation of the linear congruential generator. The final component is to tie it all together with a main.cpp program.

14.2.3 Implementation of the Main Program

Because we have already carried out most of the hard work in random.h, lin_con_gen.h, lin_con_gen.cpp, the main implementation (main.cpp) is straightforward:

```
#include <iostream>
#include "lin_con_gen.h"

int main(int argc, char **argv) {
  // Set the initial seed and the dimensionality of the RNG
  unsigned long init_seed = 1;
  unsigned long num_draws = 20;
  std::vector<double> random_draws(num_draws, 0.0);

  // Create the LCG object and create the random uniform draws
  // on the open interval (0,1)
  LinearCongruentialGenerator lcg(num_draws, init_seed);
  lcg.get_uniform_draws(random_draws);

  // Output the random draws to the console/stdout
  for (unsigned long i=0; i<num_draws; i++) {
    std::cout << random_draws[i] << std::endl;
  }

  return 0;
}
```

Firstly, we set up the initial seed and the dimensionality of the random number generator. Then we pre-initialise the vector, which will ultimately contain the uniform draws. Then we instantiate the linear congruential generator and pass the random_draws vector into the get_uniform_draws method. Finally we output the uniform variables. The output of the code is as follows:

```
7.82637e-06
0.131538
0.755605
0.45865
0.532767
0.218959
```

```
0.0470446
0.678865
0.679296
0.934693
0.383502
0.519416
0.830965
0.0345721
0.0534616
0.5297
0.671149
0.00769819
0.383416
0.0668422
```

As can be seen, all of the values lie between $(0, 1)$. We are now in a position to utilise statistical distributions with the uniform random number generator to obtain random draws.

14.3 Statistical Distributions

One of the commonest concepts in quantitative finance is that of a *statistical distribution*. Random variables play a huge part in quantitative financial modelling. Derivatives pricing, cash-flow forecasting and quantitative trading all make use of statistical methods in some fashion. Hence, modelling statistical distributions is extremely important in C++.

Many of the chapters within this book have made use of random number generators in order to carry out pricing tasks. So far this has been carried out in a *procedural* manner. Functions have been called to provide random numbers without any data encapsulation of those random number generators. The goal of this chapter is to show you that it is beneficial to create a *class hierarchy* both for statistical distributions and random number generators, separating them out in order to gain the most leverage from code reuse.

In a nutshell, we are splitting the generation of (uniform integer) random numbers from draws of specific statistical distributions, such that we can use the statistics classes elsewhere without bringing along the ”heavy” random number generation functions. Equally useful is the fact that we will be able ”swap out” different random number generators for our statistics classes for reasons of reliability, extensibility and efficiency.

14.3.1 Statistical Distribution Inheritance Hierarchy

The inheritance hierarchy for modelling of statistical distributions is relatively simple. Each distribution of interest will share the same interface, so we will create an *abstract base class*, as was carried out for the PayOff hierarchy. We are primarily interested in modelling continuous probability distributions for the time being.

Each (continuous) statistical distribution contains the following properties to be modelled:

- **Domain Interval** - The interval subset of $\mathbb{R}$ with which the distribution is defined for
- **Probability Density Function** - Describes the frequency for any particular value in our domain
- **Cumulative Density Function** - The function describing the probability that a value is less than or equal to a particular value
- **Expectation** - The expected value (the mean) of the distribution
- **Variance** - Characterisation of the spread of values around the expected value
- **Standard Deviation** - The square root of the variance, used because it possesses the same units as the expected value, unlike the variance

We also wish to produce a sequence of random draws from this distribution, assuming a sequence of random numbers is available to provide the "randomness". We can achieve this in two ways. We can either use the *inverse cumulative distribution function* (also known as the *quantile function*), which is a property of the distribution itself, or we can use a custom method (such as Box-Muller). Some of the distributions do not possess an analytical inverse to the CDF and hence they will need to be approximated numerically, via an appropriate algorithm. This calculation will be encapsulated into the class of the relevant inherited distribution.

Here is the partial header file for the StatisticalDistribution abstract base class (we will add extra distributions later):

```
#ifndef __STATISTICS_H
#define __STATISTICS_H

#include <cmath>
#include <vector>

class StatisticalDistribution {
```

```
public:
  StatisticalDistribution();
  virtual ~StatisticalDistribution();

  // Distribution functions
  virtual double pdf(const double& x) const = 0;
  virtual double cdf(const double& x) const = 0;

  // Inverse cumulative distribution functions (aka the quantile function)
  virtual double inv_cdf(const double& quantile) const = 0;

  // Descriptive stats
  virtual double mean() const = 0;
  virtual double var() const = 0;
  virtual double stdev() const = 0;

  // Obtain a sequence of random draws from this distribution
  virtual void random_draws(const std::vector<double>& uniform_draws,
                            std::vector<double>& dist_draws) = 0;
};

#endif
```

We've specified pure virtual methods for the probability density function (pdf), cumulative density function (cdf), inverse cdf (inv_cdf), as well as descriptive statistics functions such as mean, var (variance) and stdev (standard deviation). Finally we have a method that takes in a vector of uniform random variables on the open interval $(0, 1)$, then fills a vector of identical length with draws from the distribution.

Since all of the methods are pure virtual, we only need a very simple implementation of a source file for this class, since we are simply specifying an interface. However, we would like to see a concrete implementation of a particular class. We will consider, arguably, the most important distribution in quantitative finance, namely the standard normal distribution.

14.3.2 Standard Normal Distribution Implementation

Firstly we'll briefly review the formulae for the various methods we need to implement for the standard normal distribution. The probability density function of the standard normal distribution is given by:

$$\phi(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}$$

The cumulative density function is given by:

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{t^2}{2}} dt$$

The inverse cumulative density function of the standard normal distribution (also known as the *probit function*) is somewhat more involved. No analytical formula exists for this particular function and so it must be approximated by numerical methods. We will utilise the Beasley-Springer-Moro algorithm, found in Korn[15].

Given that we are dealing with the standard normal distribution, the mean is simply $\mu = 0$, variance $\sigma^2 = 1$ and standard deviation, $\sigma = 1$. The implementation for the header file (which is a continuation of statistics.h above) is as follows:

```
class StandardNormalDistribution : public StatisticalDistribution {
 public:
  StandardNormalDistribution();
  virtual ~StandardNormalDistribution();

  // Distribution functions
  virtual double pdf(const double& x) const;
  virtual double cdf(const double& x) const;

  // Inverse cumulative distribution function (aka the probit function)
  virtual double inv_cdf(const double& quantile) const;

  // Descriptive stats
  virtual double mean() const;   // equal to 0
  virtual double var() const;    // equal to 1
  virtual double stdev() const;  // equal to 1

  // Obtain a sequence of random draws from the standard normal
      distribution
  virtual void random_draws(const std::vector<double>& uniform_draws,
                            std::vector<double>& dist_draws);
};
```

The source file is given below:

```
#ifndef __STATISTICS_CPP
#define __STATISTICS_CPP

#define _USE_MATH_DEFINES

#include "statistics.h"
#include <iostream>

StatisticalDistribution::StatisticalDistribution() {}
StatisticalDistribution::~StatisticalDistribution() {}

// Constructor/destructor
StandardNormalDistribution::StandardNormalDistribution() {}
StandardNormalDistribution::~StandardNormalDistribution() {}

// Probability density function
double StandardNormalDistribution::pdf(const double& x) const {
  return (1.0/sqrt(2.0 * M_PI)) * exp(-0.5*x*x);
}

// Cumulative density function
double StandardNormalDistribution::cdf(const double& x) const {
  double k = 1.0/(1.0 + 0.2316419*x);
  double k_sum = k*(0.319381530 + k*(-0.356563782 + k*(1.781477937 +
      k*(-1.821255978 + 1.330274429*k))));

  if (x >= 0.0) {
    return (1.0 - (1.0/(pow(2*M_PI,0.5)))*exp(-0.5*x*x) * k_sum);
  } else {
    return 1.0 - cdf(-x);
  }
}
```

```
// Inverse cumulative distribution function (aka the probit function)
double StandardNormalDistribution::inv_cdf(const double& quantile) const {
  // This is the Beasley-Springer-Moro algorithm which can
  // be found in Glasserman [2004]. We won't go into the
  // details here, so have a look at the reference for more info
  static double a[4] = {   2.50662823884,
                         -18.61500062529,
                          41.39119773534,
                         -25.44106049637};

  static double b[4] = {  -8.47351093090,
                          23.08336743743,
                         -21.06224101826,
                           3.13082909833};

  static double c[9] = {0.3374754822726147,
                        0.9761690190917186,
                        0.1607979714918209,
                        0.0276438810333863,
                        0.0038405729373609,
                        0.0003951896511919,
                        0.0000321767881768,
                        0.0000002888167364,
                        0.0000003960315187};

  if (quantile >= 0.5 && quantile <= 0.92) {
    double num = 0.0;
    double denom = 1.0;

    for (int i=0; i<4; i++) {
      num += a[i] * pow((quantile - 0.5), 2*i + 1);
      denom += b[i] * pow((quantile - 0.5), 2*i);
    }
    return num/denom;

  } else if (quantile > 0.92 && quantile < 1) {
    double num = 0.0;

    for (int i=0; i<9; i++) {
      num += c[i] * pow((log(-log(1-quantile))), i);
    }
    return num;

  } else {
    return -1.0*inv_cdf(1-quantile);
  }
}

// Expectation/mean
double StandardNormalDistribution::mean() const { return 0.0; }

// Variance
double StandardNormalDistribution::var() const { return 1.0; }

// Standard Deviation
double StandardNormalDistribution::stdev() const { return 1.0; }

// Obtain a sequence of random draws from this distribution
void StandardNormalDistribution::random_draws(
    const std::vector<double>& uniform_draws,
    std::vector<double>& dist_draws
) {
  // The simplest method to calculate this is with the Box-Muller method,
  // which has been used procedurally in many other chapters

  // Check that the uniform draws and dist_draws are the same size and
  // have an even number of elements (necessary for B-M)
  if (uniform_draws.size() != dist_draws.size()) {
    std::cout << "Draw vectors are of unequal size." << std::endl;
    return;
  }

  // Check that uniform draws have an even number of elements (necessary
      for B-M)
  if (uniform_draws.size() % 2 != 0) {
    std::cout << "Uniform draw vector size not an even number." << std::
        endl;
    return;
  }

  // Slow, but easy to implement
  for (int i=0; i<uniform_draws.size() / 2; i++) {
    dist_draws[2*i] = sqrt(-2.0*log(uniform_draws[2*i])) *
        sin(2*M_PI*uniform_draws[2*i+1]);
    dist_draws[2*i+1] = sqrt(-2.0*log(uniform_draws[2*i])) *
        cos(2*M_PI*uniform_draws[2*i+1]);
  }

  return;
}
#endif
```

I'll discuss briefly some of the implementations here. The cumulative distribution function (cdf) is referenced from Joshi[12]. It is an approximation, rather than closed-form solution. The inverse CDF (inv_cdf) makes use of the Beasley-Springer-Moro algorithm, which was implemented via the algorithm given in Korn[15]. A similar method can be found in Joshi[11]. Once again the algorithm is an approximation to the real function, rather than a closed form solution. The final method is random_draws. In this instance we are using the Box-Muller algorithm. However, we could instead utilise the more efficient Ziggurat algorithm, although we won't do so here.

14.3.3 The Main Listing

We will now utilise the new statistical distribution classes with a simple random number generator in order to output statistical values:

```
#include "statistics.h"
#include <iostream>
#include <vector>

int main(int argc, char **argv) {
  // Create the Standard Normal Distribution and random draw vectors
  StandardNormalDistribution snd;
  std::vector<double> uniform_draws(20, 0.0);
  std::vector<double> normal_draws(20, 0.0);

  // Simple random number generation method based on RAND
  for (int i=0; i<uniform_draws.size(); i++) {
    uniform_draws[i] = rand() / static_cast<double>(RAND_MAX);
  }

  // Create standard normal random draws
  // Notice that the uniform draws are unaffected. We have separated
  // out the uniform creation from the normal draw creation, which
  // will allow us to create sophisticated random number generators
  // without interfering with the statistical classes
  snd.random_draws(uniform_draws, normal_draws);

  // Output the values of the standard normal random draws
  for (int i=0; i<normal_draws.size(); i++) {
    std::cout << normal_draws[i] << std::endl;
  }

  return 0;
}
```

The output from the program is as follows (a sequence of normally distributed random variables):

```
3.56692
3.28529
0.192324
-0.723522
1.10093
0.217484
-2.22963
-1.06868
-0.35082
0.806425
```

```
-0.168485
-1.3742
0.131154
0.59425
-0.449029
-2.37823
0.0431789
0.891999
0.564585
1.26432
```

Now that we have set up the inheritance hierarchy, we could construct additional (continuous) statistical distributions, such as the *log-normal distribution*, the *gamma distribution* and the *chi-square distribution*.

Chapter 15

Jump-Diffusion Models

This chapter will deal with a particular assumption of the Black-Scholes model and how to refine it to produce more accurate pricing models. In the Black-Scholes model the stock price evolves as a geometric Brownian motion. Crucially, this allows continuous Delta hedging and thus a fixed no-arbitrage price for any option on the stock.

If we relax the assumption of GBM and introduce the concept of discontinuous jumps in the stock price then it becomes impossible to perfectly hedge and leads to *market incompleteness*. This means that options prices are only *bounded* rather than fixed. In this article we will consider the effect on options prices when such jumps occur and implement a semi-closed form pricer in C++ based on the analytical formula derived by Merton[16].

15.1 Modelling Jump-Diffusion Processes

This section closely follows the chapter on Jump Diffusions in Joshi[12], where more theoretical details are provided.

In order to model such stock "jumps" we require certain properties. Firstly, the jumps should occur in an instantaneous fashion, neglecting the possibility of a Delta hedge. Secondly, we require that the probability of any jump occuring in a particular interval of time should be approximately proportional to the length of that time interval. The statistical method that models such a situation is given by the Poisson process[12].

A Poisson process states the probability of an event occuring in a given time interval Δt is given by $\lambda \Delta t + \epsilon$, where λ is the *intensity* of the process and ϵ is an error term. The integer-valued number of events that have occured at time t is given by $N(t)$. A necessary property is that the probability of a jump occuring is independent of the number of jumps already occured, i.e. all future jumps should have no "memory" of past jumps. The probability of j jumps occuring by time t is given by:

$$\mathbb{P}(N(t) = j) = \frac{(\lambda t)^j}{j!} e^{-\lambda t} \tag{15.1}$$

Thus, we simply need to modify our underlying GBM model for the stock via the addition of jumps. This is achieved by allowing the stock price to be multiplied by a random factor J:

$$dS_t = \mu S_t dt + \sigma S_t dW_t + (J - 1) S_t dN(t) \tag{15.2}$$

Where dN(t) is Poisson distributed with factor λdt.

15.2 Prices of European Options under Jump-Diffusions

After an appropriate application of risk neutrality[12] we have that $\log S_T$, the log of the final price of the stock at option expiry, is given by:

$$\log(S_T) = \log(S_0) + \left(\mu + \frac{1}{2}\sigma^2\right) T + \sigma\sqrt{T} N(0,1) + \sum_{j=1}^{N(T)} log J_j \tag{15.3}$$

This is all we need to price European options under a Monte Carlo framework. To carry this out we simply generate multiple final spot prices by drawing from a normal distribution and a Poisson distribution, and then selecting the J_j values to form the jumps. However, this is somewhat unsatisfactory as we are specifically choosing the J_j values for the jumps. Shouldn't they themselves also be random variables distributed in some manner?

In 1976, Robert Merton[16] was able to derive a semi-closed form solution for the price of European options where the jump values are themselves normally distributed. If the price of an option priced under Black-Scholes is given by $BS(S_0, \sigma, r, T, K)$ with S_0 initial spot, σ constant volatility, r constant risk-free rate, T time to maturity and K strike price, then in the jump-diffusion framework the price is given by[12]:

$$\sum_{n=0}^{\infty} \frac{e^{-\lambda' T} (\lambda' T)^n}{n!} BS(S_0, \sigma_n, r_n, T, K) \tag{15.4}$$

Where

$$\sigma_n = \sqrt{\sigma^2 + n\nu^2/T} \tag{15.5}$$

$$r_n = r - \lambda(m-1) + n\log m/T \tag{15.6}$$

$$\lambda' = \lambda m \tag{15.7}$$

The extra parameters ν and m represent the standard deviation of the lognormal jump process and the scale factor for jump intensity, respectively.

15.3 C++ Implementation

We will avoid a fully-fledged object-oriented approach for this model as it extends simply the analytical pricing of European call options quite straightforwardly. Below is the code in its entirety. The major change includes the calculation of the factorial within the summation loop and the weighted sum of the Black-Scholes prices:

```cpp
#define _USE_MATH_DEFINES

#include <iostream>
#include <cmath>

// Standard normal probability density function
double norm_pdf(const double x) {
    return (1.0/(pow(2*M_PI,0.5)))*exp(-0.5*x*x);
}

// An approximation to the cumulative distribution function
// for the standard normal distribution
// Note: This is a recursive function
double norm_cdf(const double x) {
    double k = 1.0/(1.0 + 0.2316419*x);
    double k_sum = k*(0.319381530 + k*(-0.356563782 + k*(1.781477937 + k
        *(-1.821255978 + 1.330274429*k))));

    if (x >= 0.0) {
        return (1.0 - (1.0/(pow(2*M_PI,0.5)))*exp(-0.5*x*x) * k_sum);
    } else {
        return 1.0 - norm_cdf(-x);
    }
}

// This calculates d_j, for j in {1,2}. This term appears in the closed
// form solution for the European call or put price
double d_j(const int j, const double S, const double K, const double r,
    const double v, const double T) {
    return (log(S/K) + (r + (pow(-1,j-1))*0.5*v*v)*T)/(v*(pow(T,0.5)));
}

// Calculate the European vanilla call price based on
// underlying S, strike K, risk-free rate r, volatility of
// underlying sigma and time to maturity T
double bs_call_price(const double S, const double K, const double r,
    const double sigma, const double T) {
    return S * norm_cdf(d_j(1, S, K, r, sigma, T))-K*exp(-r*T) *
        norm_cdf(d_j(2, S, K, r, sigma, T));
}

// Calculate the Merton jump-diffusion price based on
// a finite sum approximation to the infinite series
// solution, making use of the BS call price.
double bs_jd_call_price(const double S, const double K, const double r,
    const double sigma, const double T, const int N, const double m,
    const double lambda, const double nu) {
  double price = 0.0;  // Stores the final call price
  double factorial = 1.0;

  // Pre-calculate as much as possible
  double lambda_p = lambda * m;
  double lambda_p_T = lambda_p * T;

  // Calculate the finite sum over N terms
  for (int n=0; n<N; n++) {
    double sigma_n = sqrt(sigma*sigma + n*nu*nu/T);
    double r_n = r - lambda*(m - 1) + n*log(m)/T;

    // Calculate n!
    if (n == 0) {
      factorial *= 1;
    } else {
      factorial *= n;
    }

    // Refine the jump price over the loop
    price += ((exp(-lambda_p_T) * pow(lambda_p_T,n))/factorial) *
      bs_call_price(S, K, r_n, sigma_n, T);
  }

  return price;
}

int main(int argc, char **argv) {
    // First we create the parameter list
    double S = 100.0;      // Option price
    double K = 100.0;      // Strike price
    double r = 0.05;       // Risk-free rate (5%)
    double v = 0.2;        // Volatility of the underlying (20%)
    double T = 1.0;        // One year until expiry
    int N = 50;            // Terms in the finite sum approximation
    double m = 1.083287;   // Scale factor for J
    double lambda = 1.0;   // Intensity of jumps
    double nu = 0.4;       // Stdev of lognormal jump process

    // Then we calculate the call jump-diffusion value
    double call_jd = bs_jd_call_price(S, K, r, v, T, N, m, lambda, nu);
    std::cout << "Call␣Price␣under␣JD:␣␣␣␣␣␣" << call_jd << std::endl;

    return 0;
}
```

The output from the code is:

```
Call Price under JD:        18.7336
```

We can clearly see that in comparison to the Black-Scholes price of 10.4506 given in the prior chapter the value of the call under the jump diffusion process is much higher. This is to be expected since the jumps introduce extra volatility into the model.

Chapter 16

Stochastic Volatility

We have spent a good deal of time looking at vanilla and path-dependent options on QuantStart so far. We have created separate classes for random number generation and sampling from a standard normal distribution. We're now going to build on this by generating correlated time series paths.

Correlated asset paths crop up in many areas of quantitative finance and options pricing. In particular, the Heston Stochastic Volatility Model requires two correlated GBM asset paths as a basis for modelling volatility.

16.1 Motivation

Let's motivate the generation of correlated asset paths via the Heston Model. The original Black-Scholes model assumes that volatility, σ, is constant over the lifetime of the option, leading to the stochastic differential equation (SDE) for GBM:

$$dS_t = \mu S_t dt + \sigma S_t dW_t \tag{16.1}$$

The basic Heston model now replaces the constant σ coefficient with the square root of the instantaneous variance, $\sqrt{\nu_t}$. Thus the full model is given by:

$$\begin{aligned} dS_t &= \mu S_t dt + \sqrt{\nu_t} S_t dW_t^S && (16.2) \\ d\nu_t &= \kappa(\theta - \nu_t)dt + \xi\sqrt{\nu_t} dW_t^{\nu} && (16.3) \end{aligned}$$

Where dW_t^S and dW_t^ν are Brownian motions with correlation ρ. Hence we have two correlated stochastic processes. In order to price path-dependent options in the Heston framework by Monte Carlo, it is necessary to generate these two asset paths.

16.2 Process for Correlated Path Generation

Rather than considering the case of two correlated assets, we will look at N seperate assets and then reduce the general procedure to $N = 2$ for the case of our Heston motivating example.

At each time step in the path generation we require N correlated random numbers, where ρ_{ij} denotes the correlation coefficient between the ith and jth asset, x_i will be the uncorrelated random number (which we will sample from the standard normal distribution), ϵ_i will be a correlated random number, used in the asset path generation and α_{ij} will be a matrix coefficient necessary for obtaining ϵ_i.

To calculate ϵ_i we use the following process *for each of the path generation time-steps*:

$$\epsilon_i = \sum_{k=1}^{i} \alpha_{ik} x_k, 1 \leq i \leq N \tag{16.4}$$

$$\sum_{k=1}^{i} \alpha_{ik}^2 = 1, 1 \leq i \leq N \tag{16.5}$$

$$\sum_{k=1}^{i} \alpha_{ik}\alpha_{jk} = \rho_{ij}, \forall j < i \tag{16.6}$$

Thankfully, for our Heston model, we have $N = 2$ and this reduces the above equation set to the far simpler relations:

$$\epsilon_1 = x_1 \tag{16.7}$$

$$\epsilon_2 = \rho x_1 + x_2\sqrt{1-\rho^2} \tag{16.8}$$

This motivates a potential C++ implementation. We already have the capability to generate paths of standard normal distributions. If we inherit a new class CorrelatedSND (SND for 'Standard Normal Distribution'), we can provide it with a correlation coefficient and an original time series of random standard normal variables draws to generate a new correlated asset path.

16.3 Cholesky Decomposition

It can be seen that the process used to generate N correlated ϵ_i values is in fact a matrix equation. It turns out that it is actually a Cholesky Decomposition, which we have discussed in the chapter on Numerical Linear Algebra. Thus a far more efficient implementation than I am constructing here would make use of an optimised matrix class and a pre-computed Cholesky decomposition matrix.

The reason for this link is that the correlation matrix, Σ, is symmetric positive definite. Thus it can be decomposed into $\Sigma = RR^*$, although R^*, the conjugate-transpose matrix simply reduces to the transpose in the case of real-valued entries, with R a lower-triangular matrix.

Hence it is possible to calculate the correlated random variable vector ϵ via:

$$\epsilon = \mathbf{Rx} \tag{16.9}$$

Where $\mathbf{x}$ is the vector of uncorrelated variables.

We will explore the Cholesky Decomposition as applied to multiple correlated asset paths later in this chapter.

16.4 C++ Implementation

As we pointed out above the procedure for obtaining the second path will involve calculating an uncorrelated set of standard normal draws, which are then recalculated via an inherited subclass to generate a new, correlated set of random variables. For this we will make use of statistics.h and statistics.cpp, which can be found in the chapter on Statistical Distributions.

Our next task is to write the header and source files for CorrelatedSND. The listing for correlated_snd.h follows:

```
#ifndef __CORRELATED_SND_H
#define __CORRELATED_SND_H

#include "statistics.h"

class CorrelatedSND : public StandardNormalDistribution {
 protected:
  double rho;
  const std::vector<double>* uncorr_draws;

  // Modify an uncorrelated set of distribution draws to be correlated
  virtual void correlation_calc(std::vector<double>& dist_draws);

 public:
  CorrelatedSND(const double _rho,
                const std::vector<double>* _uncorr_draws);
  virtual ~CorrelatedSND();

  // Obtain a sequence of correlated random draws from another set of SND
      draws
  virtual void random_draws(const std::vector<double>& uniform_draws,
                            std::vector<double>& dist_draws);
};

#endif
```

The class inherits from StandardNormalDistribution, provided in statistcs.h. We are adding two **protected** members, rho (the correlation coefficient) and uncorr_draws, a pointer to a const vector of doubles. We also create an additional virtual method, correlation_calc, that actually performs the correlation calculation. The only additional modification is to add the parameters, which will ultimately become stored as protected member data, to the constructor.

Next up is the source file, correlated_snd.cpp:

```
#ifndef __CORRELATED_SND_CPP
#define __CORRELATED_SND_CPP

#include "correlated_snd.h"
#include <iostream>
#include <cmath>

CorrelatedSND::CorrelatedSND(const double _rho,
                             const std::vector<double>* _uncorr_draws)
  : rho(_rho), uncorr_draws(_uncorr_draws) {}

CorrelatedSND::~CorrelatedSND() {}
```

```
// This carries out the actual correlation modification. It is easy to see
    that if
// rho = 0.0, then dist_draws is unmodified, whereas if rho = 1.0, then
    dist_draws
// is simply set equal to uncorr_draws. Thus with 0 < rho < 1 we have a
// weighted average of each set.
void CorrelatedSND::correlation_calc(std::vector<double>& dist_draws) {
    for (int i=0; i<dist_draws.size(); i++) {
      dist_draws[i] = rho * (*uncorr_draws)[i] + dist_draws[i] * sqrt(1-rho
          *rho);
    }
}

void CorrelatedSND::random_draws(const std::vector<double>& uniform_draws,
                                 std::vector<double>& dist_draws) {
  // The following functionality is lifted directly from
  // statistics.h, which is fully commented!
  if (uniform_draws.size() != dist_draws.size()) {
    std::cout << "Draws␣vectors␣are␣of␣unequal␣size␣in␣standard␣normal␣dist
        ."
      << std::endl;
    return;
  }

  if (uniform_draws.size() % 2 != 0) {
    std::cout << "Uniform␣draw␣vector␣size␣not␣an␣even␣number." << std::
        endl;
    return;
  }

  for (int i=0; i<uniform_draws.size() / 2; i++) {
    dist_draws[2*i] = sqrt(-2.0*log(uniform_draws[2*i])) *
      sin(2*M_PI*uniform_draws[2*i+1]);
    dist_draws[2*i+1] = sqrt(-2.0*log(uniform_draws[2*i])) *
      cos(2*M_PI*uniform_draws[2*i+1]);
  }
  // Modify the random draws via the correlation calculation
  correlation_calc(dist_draws);

  return;
}

#endif
```

The work is carried out in correlation_calc. It is easy to see that if $\rho = 0$, then dist_draws is unmodified, whereas if $\rho = 1$, then dist_draws is simply equated to uncorr_draws. Thus with $0 < \rho < 1$ we have a weighted average of each set of random draws. Note that I have reproduced the Box-Muller functionality here so that you don't have to look it up in statistics.cpp. In a production code this would be centralised elsewhere (such as with a random number generator class).

Now we can tie it all together. Here is the listing of main.cpp:

```
#include "statistics.h"
#include "correlated_snd.h"
#include <iostream>
#include <vector>

int main(int argc, char **argv) {

  // Number of values
  int vals = 30;

  /* UNCORRELATED SND */
  /* ================ */

  // Create the Standard Normal Distribution and random draw vectors
  StandardNormalDistribution snd;
  std::vector<double> snd_uniform_draws(vals, 0.0);
  std::vector<double> snd_normal_draws(vals, 0.0);

  // Simple random number generation method based on RAND
  // We could be more sophisticated an use a LCG or Mersenne Twister
  // but we're trying to demonstrate correlation, not efficient
  // random number generation!
  for (int i=0; i<snd_uniform_draws.size(); i++) {
    snd_uniform_draws[i] = rand() / static_cast<double>(RAND_MAX);
  }

  // Create standard normal random draws
  snd.random_draws(snd_uniform_draws, snd_normal_draws);

  /* CORRELATION SND */
  /* =============== */

  // Correlation coefficient
  double rho = 0.5;

  // Create the correlated standard normal distribution
  CorrelatedSND csnd(rho, &snd_normal_draws);
  std::vector<double> csnd_uniform_draws(vals, 0.0);
  std::vector<double> csnd_normal_draws(vals, 0.0);

  // Uniform generation for the correlated SND
  for (int i=0; i<csnd_uniform_draws.size(); i++) {
    csnd_uniform_draws[i] = rand() / static_cast<double>(RAND_MAX);
  }

  // Now create the -correlated- standard normal draw series
  csnd.random_draws(csnd_uniform_draws, csnd_normal_draws);

  // Output the values of the standard normal random draws
  for (int i=0; i<snd_normal_draws.size(); i++) {
    std::cout << snd_normal_draws[i] << ", " << csnd_normal_draws[i] << std
        ::endl;
  }

  return 0;
}
```

The above code is somewhat verbose, but that is simply a consequence of not encapsulating the random number generation capability. Once we have created an initial set of standard normal draws, we simply have to pass that to an instance of CorrelatedSND (in this line: CorrelatedSND csnd(rho, &snd_normal_draws);) and then call random_draws(..) to create the correlated stream. Finally, we output both sets of values:

```
3.56692, 1.40915
3.28529, 1.67139
0.192324, 0.512374
−0.723522, 0.992231
1.10093, 1.14815
0.217484, −0.211253
−2.22963, −1.94287
−1.06868, −0.500967
−0.35082, −0.0884041
0.806425, 0.326177
−0.168485, −0.242706
−1.3742, 0.752414
0.131154, −0.632282
0.59425, 0.311842
−0.449029, 0.129012
−2.37823, −0.469604
0.0431789, −0.52855
0.891999, 1.0677
0.564585, 0.825356
1.26432, −0.653957
−1.21881, −0.521325
−0.511385, −0.881099
−0.43555, 1.23216
0.93222, 0.237333
−0.0973298, 1.02387
−0.569741, 0.33579
−1.7985, −1.52262
−1.2402, 0.211848
−1.26264, −0.490981
−0.39984, 0.150902
```

There are plenty of extensions we could make to this code. The obvious two are encapsulating the random number generation and converting it to use an efficient Cholesky Decomposition implementation. Now that we have correlated streams, we can also implement the Heston Model in Monte Carlo.

Up until this point we have priced all of our options under the assumption that the volatility, σ, of the underlying asset has been constant over the lifetime of the option. In reality financial markets do not behave this way. Assets exist under *market regimes* where their volatility can vary signficantly during different time periods. The 2007-2008 financial crisis and the May Flash Crash of 2010 are good examples of periods of intense market volatility.

Thus a natural extension of the Black Scholes model is to consider a non-constant volatility. Steven Heston formulated a model that not only considered a time-dependent volatility, but also introduced a stochastic (i.e. non-deterministic) component as well. This is the famous **Heston model for stochastic volatility**.

In this chapter we will outline the mathematical model and use a discretisation technique known as Full Truncation Euler Discretisation, coupled with Monte Carlo simulation, in order to price a European vanilla call option with C++. As with the majority of the models implemented on QuantStart, the code is object-oriented, allowing us to "plug-in" other option types (such as Path-Dependent Asians) with minimal changes.

16.5 Mathematical Model

The Black Scholes model uses a stochastic differential equation with a geometric Brownian motion to model the dynamics of the asset path. It is given by:

$$dS_t = \mu S_t dt + \sigma S_t dW_t^S \tag{16.10}$$

S_t is the price of the underlying asset at time t, μ is the (constant) drift of the asset, σ is the (constant) volatility of the underlying and dW_t^S is a Weiner process (i.e. a random walk).

The Heston model extends this by introducing a second stochastic differential equation to represent the "path" of the volatility of the underlying over the lifetime of the option. The SDE for the variance is given by a Cox-Ingersoll-Ross process:

$$\begin{aligned} dS_t &= \mu S_t dt + \sqrt{\nu_t} S_t dW_t^S & (16.11) \\ d\nu_t &= \kappa(\theta - \nu_t)dt + \xi\sqrt{\nu_t} dW_t^\nu & (16.12) \end{aligned}$$

Where:

- μ is the drift of the asset
- θ is the *expected value* of ν_t, i.e. the long run average price variance
- κ is the rate of mean reversion of ν_t to the long run average θ
- ξ is the "vol of vol", i.e. the variance of ν_t

Note that none of the parameters have any time-dependence. Extensions of the Heston model generally allow the values to become piecewise constant.

In order for $\nu_t > 0$, the Feller condition must be satisfied:

$$2\kappa\theta > \xi^2 \tag{16.13}$$

In addition, the model enforces that the two separate Weiner processes making up the randomness are in fact correlated, with instantaneous constant correlation ρ:

$$dW_t^S dW_t^\nu = \rho dt \tag{16.14}$$

16.6 Euler Discretisation

Given that the SDE for the asset path is now dependent (in a temporal manner) upon the solution of the second volatility SDE, it is necessary to simulate the volatility process first and then utilise this "volatility path" in order to simulate the asset path. In the case of the original Black Scholes SDE it is possible to use Ito's Lemma to directly solve for S_t. However, we are unable to utilise that procedure here and must use a *numerical approximation* in order to obtain both paths. The method utilised is known as **Euler Discretisation**.

The volatility path will be discretised into constant-increment time steps of Δt, with the updated volatility, ν_{i+1} given as an *explicit* function of ν_i:

$$\nu_{i+1} = \nu_i + \kappa(\theta - \nu_i)\Delta t + \xi\sqrt{\nu_i}\Delta W_{i+1}^\nu \tag{16.15}$$

However, since this is a finite discretisation of a continuous process, it is possible to introduce *discretisation errors* where ν_{i+1} can become negative. This is not a "physical" situation and so is a direct consequence of the numerical approximation. In order to handle negative values, we need to modify the above formula to include methods of eliminating negative values for subsequent iterations of the volatility path. Thus we introduce three new functions f_1, f_2, f_3, which lead to three separate "schemes" for how to handle the negative volatility values:

$$\nu_{i+1} \quad = \quad f_1(\nu_i) + \kappa(\theta - f_2(\nu_i))\Delta t + \xi\sqrt{f_3(\nu_i)}\Delta W^{\nu}_{i+1} \tag{16.16}$$

The three separate schemes are listed below:

Scheme	f_1	f_2	f_3
Reflection	$\|x\|$	$\|x\|$	$\|x\|$
Partial Truncation	x	x	x
Full Truncation	x	x^+	x^+

Where $x^+ = \max(x, 0)$.

The literature tends to suggest that the Full Truncation method is the "best" and so this is what we will utilise here. The Full Truncation scheme discretisation equation for the volatility path will thus be given by:

$$\nu_{i+1} \quad = \quad \nu_i + \kappa(\theta - \nu_i^+)\Delta t + \xi\sqrt{\nu_i^+}\Delta W^{\nu}_{i+1} \tag{16.17}$$

In order to simulate ΔW^{ν}_{i+1}, we can make use of the fact that since it is a Brownian motion, $W^{\nu}_{i+1} - W^{\nu}_i$ is normally distributed with variance Δt and that the distribution of $W^{\nu}_{i+1} - W^{\nu}_i$ is independent of i. This means it can be replaced with $\sqrt{\Delta t}N(0,1)$, where $N(0,1)$ is a random draw from the standard normal distribution.

We will return to the question of how to calculate the W^{ν}_i terms in the next section. Assuming we have the ability to do so, we are able to simulate the price of the asset path with the following discretisation:

$$S_{i+1} = S_i \exp\left(\mu - \frac{1}{2}v_i^+\right)\Delta t + \sqrt{v_i^+}\sqrt{\Delta t}\Delta W^S_{i+1} \tag{16.18}$$

As with the Full Truncation mechanism outlined above, the volatility term appearing in the asset SDE discretisation has also been truncated and so ν_i is replaced by ν_i^+.

16.6.1 Correlated Asset Paths

The next major issue that we need to look at is how to generate the W_i^ν and W_i^S terms for the volatility path and the asset path respectively, such that they remain correlated with correlation ρ, as prescribed via the mathematical model. This is exactly what is necessary here.

Once we have two uniform random draw vectors it is possible to use the StandardNormalDistribution class outlined in the chapter on statistical distributions to create two new vectors containing standard normal random draws - exactly what we need for the volatility and asset path simulation!

16.6.2 Monte Carlo Algorithm

In order to price a European vanilla call option under the Heston stochastic volatility model, we will need to generate many asset paths and then calculate the risk-free discounted average pay-off. This will be our option price.

The algorithm that we will follow to calculate the full options price is as follows:

1. Choose number of asset simulations for Monte Carlo and number of intervals to discretise asset/volatility paths over
2. For each Monte Carlo simulation, generate two uniform random number vectors, with the second correlated to the first
3. Use the statistics distribution class to convert these vectors into two new vectors containing standard normal draws
4. For each time-step in the discretisation of the vol path, calculate the next volatility value from the normal draw vector
5. For each time-step in the discretisation of the asset path, calculate the next asset value from the vol path vector and normal draw vector
6. For each Monte Carlo simulation, store the pay-off of the European call option
7. Take the mean of these pay-offs and then discount via the risk-free rate to produce an option price, under risk-neutral pricing.

We will now present a C++ implementation of this algorithm using a mixture of new code and prior classes written in prior chapters.

16.7 C++ Implementation

We are going to take an object-oriented approach and break the calculation domain into various re-usable classes. In particular we will split the calculation into the following objects:

- PayOff - This class represents an option pay-off object. We have discussed it at length on QuantStart.
- Option - This class holds the parameters associated with the *term sheet* of the European option, as well as the risk-free rate. It requires a PayOff instance.
- StandardNormalDistribution - This class allows us to create standard normal random draw values from a uniform distribution or random draws.
- CorrelatedSND - This class takes two standard normal random draws and correlates the second with the first by a correlation factor ρ.
- HestonEuler - This class accepts Heston model parameters and then performs a Full Truncation of the Heston model, generating both a volatility path and a subequent asset path.

We will now discuss the classes individually.

16.7.1 PayOff Class

The PayOff class won't be discussed in any great detail within this chapter as it is described fully in previous chapters. The PayOff class is a functor and as such is *callable*.

16.7.2 Option Class

The Option class is straightforward. It simply contains a set of public members for the option *term sheet* parameters (strike K, time to maturity T) as well as the (constant) risk-free rate r. The class also takes a pointer to a PayOff object, making it straightforward to "swap out" another pay-off (such as that for a Put option).

The listing for option.h follows:

```
#ifndef __OPTION_H
#define __OPTION_H
```

```
#include "payoff.h"

class Option {
 public:
  PayOff* pay_off;
  double K;
  double r;
  double T;

  Option(double _K, double _r,
         double _T, PayOff* _pay_off);

  virtual ~Option();
};

#endif
```

The listing for option.cpp follows:

```
#ifndef __OPTION_CPP
#define __OPTION_CPP

#include "option.h"

Option::Option(double _K, double _r,
               double _T, PayOff* _pay_off) :
  K(_K), r(_r), T(_T), pay_off(_pay_off) {}

Option::~Option() {}

#endif
```

As can be seen from the above listings, the class doesn't do much beyond storing some data members and exposing them.

16.7.3 Statistics and CorrelatedSND Classes

The StandardNormalDistribution and CorrelatedSND classes are described in detail within the chapter on statistical distributions and in the sections above so we will not go into detail here.

16.7.4 HestonEuler Class

The HestonEuler class is designed to accept the parameters of the Heston Model - in this case κ, θ, ξ and ρ - and then calculate both the volatility and asset price paths. As such there are private data members for these parameters, as well as a pointer member representing the option itself. There are two calculation methods designed to accept the normal draw vectors and produce the respective volatility or asset spot paths.

The listing for heston_mc.h follows:

```
#ifndef __HESTON_MC_H
#define __HESTON_MC_H

#include <cmath>
#include <vector>
#include "option.h"

// The HestonEuler class stores the necessary information
// for creating the volatility and spot paths based on the
// Heston Stochastic Volatility model.
class HestonEuler {
 private:
  Option* pOption;
  double kappa;
  double theta;
  double xi;
  double rho;

 public:
  HestonEuler(Option* _pOption,
              double _kappa, double _theta,
              double _xi, double _rho);
  virtual ~HestonEuler();
  // Calculate the volatility path
  void calc_vol_path(const std::vector<double>& vol_draws,
                     std::vector<double>& vol_path);

  // Calculate the asset price path
  void calc_spot_path(const std::vector<double>& spot_draws,
                      const std::vector<double>& vol_path,
                      std::vector<double>& spot_path);
};

#endif
```

The listing for heston_mc.cpp follows:

```
#ifndef __HESTON_MC_CPP
#define __HESTON_MC_CPP

#include "heston_mc.h"

// HestonEuler
// ===========

HestonEuler::HestonEuler(Option* _pOption,
                         double _kappa, double _theta,
                         double _xi, double _rho) :
  pOption(_pOption), kappa(_kappa), theta(_theta), xi(_xi), rho(_rho) {}

HestonEuler::~HestonEuler() {}

void HestonEuler::calc_vol_path(const std::vector<double>& vol_draws,
                                std::vector<double>& vol_path) {
  size_t vec_size = vol_draws.size();
  double dt = pOption->T/static_cast<double>(vec_size);

  // Iterate through the correlated random draws vector and
  // use the 'Full Truncation' scheme to create the volatility path
  for (int i=1; i<vec_size; i++) {
    double v_max = std::max(vol_path[i-1], 0.0);
    vol_path[i] = vol_path[i-1] + kappa * dt * (theta - v_max) +
      xi * sqrt(v_max * dt) * vol_draws[i-1];
  }
}

void HestonEuler::calc_spot_path(const std::vector<double>& spot_draws,
                                 const std::vector<double>& vol_path,
                                 std::vector<double>& spot_path) {
  size_t vec_size = spot_draws.size();
  double dt = pOption->T/static_cast<double>(vec_size);

  // Create the spot price path making use of the volatility
  // path. Uses a similar Euler Truncation method to the vol path.
  for (int i=1; i<vec_size; i++) {
    double v_max = std::max(vol_path[i-1], 0.0);
    spot_path[i] = spot_path[i-1] * exp( (pOption->r - 0.5*v_max)*dt +
        sqrt(v_max*dt)*spot_draws[i-1]);
  }
}

#endif
```

The calc_vol_path method takes references to a const vector of normal draws and a vector to store the volatility path. It calculates the Δt value (as dt), based on the option maturity time. Then, the stochastic simulation of the volatility path is carried out by means of the Full Truncation Euler Discretisation, outlined in the mathematical treatment above. Notice that ν_i^+ is precalculated, for efficiency reasons.

The calc_spot_path method is similar to the calc_vol_path method, with the exception that it accepts another vector, vol_path that contains the volatility path values at each time increment. The risk-free rate r is obtained from the option pointer and, once again, ν_i^+ is precalculated. Note that all vectors are passed by reference in order to reduce unnecessary copying.

16.7.5 Main Program

This is where it all comes together. There are two components to this listing: The generate_normal_correlation_paths function and the main function. The former is designed to handle the "boilerplate" code of generating the necessary uniform random draw vectors and then utilising the CorrelatedSND object to produce correlated standard normal distribution random draw vectors.

I wanted to keep this entire example of the Heston model tractable, so I have simply used the C++ built-in rand function to produce the uniform standard draws. However, in a production environment a Mersenne Twister uniform number generator (or something even more sophisticated) would be used to produce high-quality pseudo-random numbers. The output of the function is to calculate the values for the spot_normals and cor_normals vectors, which are used by the asset spot path and the volatility path respectively.

The main function begins by defining the parameters of the simulation, including the Monte Carlo values and those necessary for the option and Heston model. The actual parameter values are those give in the paper by Broadie and Kaya[3]. The next task is to create the pointers to the PayOff and Option classes, as well as the HestonEuler instance itself.

After declaration of the various vectors used to hold the path values, a basic Monte Carlo loop is created. For each asset simulation, the new correlated values are generated, leading to the calculation of the vol path and the asset spot path. The option pay-off is calculated for each path and added to the total, which is then subsequently averaged and discounted via the risk-free rate. The option price is output to the terminal and finally the pointers are deleted.

Here is the listing for main.cpp:

```
#include <iostream>

#include "payoff.h"
#include "option.h"
#include "correlated_snd.h"
#include "heston_mc.h"

void generate_normal_correlation_paths(double rho,
    std::vector<double>& spot_normals, std::vector<double>& cor_normals) {
  unsigned vals = spot_normals.size();

  // Create the Standard Normal Distribution and random draw vectors
  StandardNormalDistribution snd;
  std::vector<double> snd_uniform_draws(vals, 0.0);

  // Simple random number generation method based on RAND
  for (int i=0; i<snd_uniform_draws.size(); i++) {
    snd_uniform_draws[i] = rand() / static_cast<double>(RAND_MAX);
  }

  // Create standard normal random draws
  snd.random_draws(snd_uniform_draws, spot_normals);

  // Create the correlated standard normal distribution
  CorrelatedSND csnd(rho, &spot_normals);
  std::vector<double> csnd_uniform_draws(vals, 0.0);

  // Uniform generation for the correlated SND
  for (int i=0; i<csnd_uniform_draws.size(); i++) {
    csnd_uniform_draws[i] = rand() / static_cast<double>(RAND_MAX);
  }

  // Now create the -correlated- standard normal draw series
  csnd.random_draws(csnd_uniform_draws, cor_normals);
}

int main(int argc, char **argv) {
  // First we create the parameter list
  // Note that you could easily modify this code to input the parameters
  // either from the command line or via a file
  unsigned num_sims = 100000;    // Number of simulated asset paths
  unsigned num_intervals = 1000;  // Number of intervals for the asset path
      to be sampled

  double S_0 = 100.0;     // Initial spot price
  double K = 100.0;       // Strike price
  double r = 0.0319;      // Risk-free rate
  double v_0 = 0.010201; // Initial volatility
  double T = 1.00;        // One year until expiry

  double rho = -0.7;      // Correlation of asset and volatility
  double kappa = 6.21;    // Mean-reversion rate
  double theta = 0.019;   // Long run average volatility
  double xi = 0.61;       // "Vol of vol"
  // Create the PayOff, Option and Heston objects
  PayOff* pPayOffCall = new PayOffCall(K);
  Option* pOption = new Option(K, r, T, pPayOffCall);
  HestonEuler hest_euler(pOption, kappa, theta, xi, rho);

  // Create the spot and vol initial normal and price paths
  std::vector<double> spot_draws(num_intervals, 0.0);  // Vector of initial
       spot normal draws
  std::vector<double> vol_draws(num_intervals, 0.0);   // Vector of initial
       correlated vol normal draws
  std::vector<double> spot_prices(num_intervals, S_0);  // Vector of
      initial spot prices
  std::vector<double> vol_prices(num_intervals, v_0);   // Vector of
      initial vol prices

  // Monte Carlo options pricing
  double payoff_sum = 0.0;
  for (unsigned i=0; i<num_sims; i++) {
    std::cout << "Calculating path " << i+1 << " of " << num_sims << std::
        endl;
    generate_normal_correlation_paths(rho, spot_draws, vol_draws);
    hest_euler.calc_vol_path(vol_draws, vol_prices);
    hest_euler.calc_spot_path(spot_draws, vol_prices, spot_prices);
    payoff_sum += pOption->pay_off->operator()(spot_prices[num_intervals
        -1]);
  }
  double option_price = (payoff_sum / static_cast<double>(num_sims)) * exp
      (-r*T);
  std::cout << "Option Price: " << option_price << std::endl;

  // Free memory
  delete pOption;
  delete pPayOffCall;

  return 0;
}
```

For completeness, I have included the *makefile* utilised on my MacBook Air, running Mac OSX 10.7.4:

```
heston: main.cpp heston_mc.o correlated_snd.o statistics.o option.o payoff.
    o
    clang++ -o heston main.cpp heston_mc.o correlated_snd.o statistics.o
        option.o payoff.o -arch x86_64

heston_mc.o: heston_mc.cpp option.o
    clang++ -c heston_mc.cpp option.o -arch x86_64

correlated_snd.o: correlated_snd.cpp statistics.o
    clang++ -c correlated_snd.cpp statistics.o -arch x86_64

statistics.o: statistics.cpp
    clang++ -c statistics.cpp -arch x86_64

option.o: option.cpp payoff.o
    clang++ -c option.cpp payoff.o -arch x86_64

payoff.o: payoff.cpp
    clang++ -c payoff.cpp -arch x86_64
```

Here is the output of the program:

```
..
..
Calculating path 99997 of 100000
Calculating path 99998 of 100000
Calculating path 99999 of 100000
Calculating path 100000 of 100000
Option Price: 6.81982
```

The exact option price is 6.8061, as reported by Broadie and Kaya[3]. It can be made somewhat more accurate by increasing the number of asset paths and discretisation intervals.

There are a few extensions that could be made at this stage. One is to allow the various "schemes" to be implemented, rather than hard-coded as above. Another is to introduce time-dependence into the parameters. The next step after creating a model of this type is to actually calibrate to a set of market data such that the parameters may be determined.

Chapter 17

Single Factor Black-Scholes with Finite Difference Methods

We've spent a lot of time in this book looking at Monte Carlo Methods for pricing of derivatives. However, we've so far neglected a very deep theory of pricing that takes a different approach. In this chapter we are going to begin making use of Finite Difference Methods (FDM) in order to price European options, via the **Explicit Euler Method**.

Finite Difference Methods are extremely common in fields such as fluid dynamics where they are used to provide numerical solutions to partial differential equations (PDE), which often possess no analytical equivalent. Finite Difference Methods are relevant to us since the Black Scholes equation, which represents the price of an option as a function of underlying asset spot price, is a partial differential equation. In particular, it is actually a **convection-diffusion equation**, a type of second-order PDE.

Finite Difference Methods make appropriate approximations to the derivative terms in a PDE, such that the problem is reduced from a continuous differential equation to a finite set of discrete algebraic equations. The solution of these equations, under certain conditions, approximates the continuous solution. By refining the number of discretisation points it is possible to more closely approximate the continuous solution to any accuracy desired, assuming certain *stability conditions*.

Finite Difference Methods use a *time-marching* approach to take a known approximate solution at time N, C^n, and calculate a new approximate solution at a stepped time $N+1$, C^{n+1}. However, the Black-Scholes equation is slightly different in that the known solution is actually the pay-off of the option at expiry time, T. Hence time is marched *backwards* from the expiry point to the initial time, $t = 0$. This is analogous to the diffusion of heat in the *heat equation. In fact, the Black-Scholes can be transformed into the heat equation by a suitable coordinate change and solved analytically, although this is beyond the scope of this chapter!*

In order to obtain a solution to the Black-Scholes PDE for a European vanilla call option, we will carry out the following steps:

- **Describe the PDE** - Our first task is to outline the mathematical formalism by describing the Black-Scholes equation itself along with any initial and boundary conditions that apply as well as the domain over which the solution will be calculated.
- **Discretisation** - We will then discretise the Black-Scholes PDE using suitable approximations to the derivative terms.
- **Object Orientation** - Once we have the discretisation in place we will decide how to define the objects representing our finite difference method in C++ code.
- **Execution and Output** - After we have created all of the C++ code for the implementation, and executed it, we will plot the resulting option pricing surface using Python and matplotlib.

The formula for the Black-Scholes PDE is as follows:

$$-\frac{\partial C}{\partial t} + rS\frac{\partial C}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 C}{\partial S^2} - rC = 0$$

Our goal is to find a *stable* discretisation for this formula that we can implement. It will produce an option pricing surface, $C(S, t)$ as a function of spot S and time t that we can plot.

17.1 Black-Scholes PDE for a European Call Option

In this chapter we are going to start simply by approximating the solution to a European vanilla call option. In fact an analytic solution exists for such an option. However, our task here is to outline the Finite Difference Method, not to solve the most exotic option we can find right away!

In order to carry out the procedure we must specify the Black-Scholes PDE, the domain on which the solution will exist and the constraints - namely the initial and boundary conditions - that apply. Here is the full mathematical formalism of the problem:

$$\begin{aligned}
\frac{\partial C}{\partial t} &= rS\frac{\partial C}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 C}{\partial S^2} - rC \\
C(0,t) &= 0,\ C(S_{\max},t) = S_{\max} - Ke^{-r(T-t)}, \quad 0 \le t \le T \\
C(S,T) &= \max(S-K,0), \quad 0 \le S \le S_{\max}
\end{aligned}$$

Let's step through the problem. The first line is simply the Black-Scholes equation.

The second line describes the **boundary conditions** for the case of a European vanilla call option. The left-hand boundary equation states that on the left-boundary, for all times up to expiry, the value of the call will equal 0. The right-hand equation for the right-boundary states that, for all times up to expiry, the value of the call will equal the pay-off function, albeit slightly adjusted to take into account discounting due to the risk-free rate, r. This is a consequence of Put-Call Parity. Both of these boundary conditions are of the Dirichlet type.

The final line is the **initial condition**, which describes how the solution should behave at the start of the time-marching procedure. Since we are marching backwards, this is actually the final pay-off of the option at expiry, which is the familiar expression for a vanilla call option pay-off.

17.2 Finite Difference Discretisation

Now that we have specified the continuous problem mathematically, we have to take this and create a finite set of discretised algebraic equations, which are able to be solved by the computer. We will make use of *forward differencing* for the time partial derivative, *centred differencing* for the first order spatial derivative and a *centred second difference* for the diffusion term:

$$-\frac{C^{n+1} - C^n}{\Delta t} + \frac{1}{2}\sigma^2 S_j^2 \left(\frac{C^n_{j+1} - 2C^n_j + C^n_{j-1}}{\Delta x^2} \right) + rS_j \left(\frac{C^n_{j+1} - C^n_{j-1}}{2\Delta x} \right) - rC^n_j = 0$$

This can be rearranged so that the solution at time level $N+1$ is given in terms of the solution at time level N. This is what gives the method its *explicit* name. The solution is *explicitly* dependent upon previous time levels. In later FDM methods we will see that this does not have to be the case. Here is the rearranged equation:

$$C_j^{n+1} = \alpha_j C_{j-1}^n + \beta_j C_j^n + \gamma_j C_{j+1}^n$$

In order to describe these coefficients succinctly we can use the fact that the spot values, S_j, increase linearly with Δx. Therefore $S_j = j\Delta x$. After some algebraic rearrangement, the coefficients are given by:

$$\begin{aligned}
\alpha_j &= \frac{\sigma^2 j^2 \Delta t}{2} - \frac{rj\Delta t}{2} \\
\beta_j &= 1 - \sigma^2 j^2 \Delta t - r\Delta t \\
\gamma_j &= \frac{\sigma^2 j^2 \Delta t}{2} + \frac{rj\Delta t}{2}
\end{aligned}$$

This provides us with everything we need to begin our Finite Difference implementation.

Note that I have not discussed some extremely important topics with regards to finite difference methods - particularly **consistency***,* **stability** *and* **convergence***. These are all deep areas of numerical analysis in their own right. However, I will be going into more detail about these topics later!*

17.3 Implementation

We're now ready to begin turning our mathematical algorithm into a C++ implementation. At this stage it isn't immediately obvious how we will go about doing this. One could create a monolithic procedural code to calculate the entire solution. However, frequent QuantStarters will know that this is a suboptimal approach for many reasons. Instead we will make use of the object-oriented paradigm, as well as previous code that we have already written, in order to save development time.

Let's discuss the classes which will form the basis of our FDM solver:

- **PayOff** - This class represents the pay-off functionality of an option. It is also a functor. We have already made extensive use of it in our Monte Carlo options pricing chapters. I have added the code listing below for completeness.

- **VanillaOption** - This is a simple class that encapsulates the option parameters. We are using it as well as PayOff as we will want to extend the FDM solver to allow more exotic options in the future.

- **ConvectionDiffusionPDE** - This is an *abstract base class* designed to provide an interface to all subsequent derived classes. It consists of pure virtual functions representing the various coefficients of the PDE as well as boundary and initial conditions.
- **BlackScholesPDE** - This inherits from ConvectionDiffusionPDE and provides concrete implementations of the coefficients and boundary/initial conditions specific to the Black-Scholes equation.
- **FDMBase** - This is another abstract base class that provides discretisation parameters and result storage for the Finite Difference scheme. It possesses a pointer to a ConvectionDiffusionPDE.
- **FDMEulerExplicit** - This inherits from FDMBase and provides concrete methods for the Finite Difference scheme methods for the particular case of the Explicit Euler Method, which we described above.

Let's now describe each class in detail.

17.3.1 PayOff Class

I don't want elaborate too much on the PayOff class as it has been discussed before in earlier chapters, such as when pricing Asian options by Monte Carlo. However, I have included the listings for the header and source files for completeness.

Here is the listing for payoff.h:

```
#ifndef __PAY_OFF_HPP
#define __PAY_OFF_HPP

// This is needed for the std::max comparison
// function, used in the pay-off calculations
#include <algorithm>

class PayOff {
 public:
  PayOff(); // Default (no parameter) constructor
  virtual ~PayOff() {}; // Virtual destructor
  // Overloaded () operator, turns the PayOff into an abstract function
      object
  virtual double operator() (const double& S) const = 0;
};

class PayOffCall : public PayOff {
 private:
  double K; // Strike price

 public:
  PayOffCall(const double& K_);
  virtual ~PayOffCall() {};

  // Virtual function is now over-ridden (not pure-virtual anymore)
  virtual double operator() (const double& S) const;
};

class PayOffPut : public PayOff {
 private:
  double K; // Strike

 public:
  PayOffPut(const double& K_);
  virtual ~PayOffPut() {};
  virtual double operator() (const double& S) const;
};

#endif
```

Here is the listing for payoff.cpp:

```
#ifndef __PAY_OFF_CPP
#define __PAY_OFF_CPP

#include "payoff.h"

PayOff::PayOff() {}
```

```
// ==========
// PayOffCall
// ==========

// Constructor with single strike parameter
PayOffCall::PayOffCall(const double& _K) { K = _K; }

// Over-ridden operator() method, which turns PayOffCall into a function
     object
double PayOffCall::operator() (const double& S) const {
  return std::max(S-K, 0.0); // Standard European call pay-off
}

// =========
// PayOffPut
// =========

// Constructor with single strike parameter
PayOffPut::PayOffPut(const double& _K) {
  K = _K;
}

// Over-ridden operator() method, which turns PayOffPut into a function
     object
double PayOffPut::operator() (const double& S) const {
  return std::max(K-S, 0.0); // Standard European put pay-off
}

#endif
```

17.3.2 VanillaOption Class

In the future we may wish to price many differing types of exotic options via Finite Difference Methods. Thus it is sensible to create a VanillaOption class to encapsulate this functionality. In particular, we are going to encapsulate the storage of the parameters of a European vanilla option. Despite the fact that the interest rate, r, and the volatility, σ, are not part of an option *term sheet*, we will include them as parameters for simplicity.

The notable component of the option is the pointer to a PayOff class. This allows us to use a call, put or some other form of pay-off without needing to expose this to the "outside world", which in this instance refers to the FDM solver.

Here is the listing for option.h:

```
#ifndef __VANILLA_OPTION_H
#define __VANILLA_OPTION_H

#include "payoff.h"

class VanillaOption {
 public:
  PayOff* pay_off;

  double K;
  double r;
  double T;
  double sigma;

  VanillaOption();
  VanillaOption(double _K, double _r, double _T,
                double _sigma, PayOff* _pay_off);
};

#endif
```

The source file only really provides implementations for the constructors, both of which are blank as the member initialisation list takes care of member initialisation.

Here is the listing for option.cpp:

```
#ifndef __VANILLA_OPTION_CPP
#define __VANILLA_OPTION_CPP

#include "option.h"

VanillaOption::VanillaOption() {}

VanillaOption::VanillaOption(double _K, double _r, double _T,
                             double _sigma, PayOff* _pay_off) :
```

```
  K(_K), r(_r), T(_T), sigma(_sigma), pay_off(_pay_off) {}

#endif
```

17.3.3 PDE Classes

Separating the mathematical formalism of the PDE with the finite difference method that solves it leads to the creation of the ConvectionDiffusionPDE and BlackScholesPDE classes. ConvectionDiffusionPDE is simply an abstract base class, providing an interface for all subsequent inherited classes.

The pure virtual methods consist of all of the coefficients found in a second-order convection-diffusion PDE. In addition, pure virtual methods are provided for "left" and "right" boundary conditions. Thus we are defining a one-dimensional (in 'space') PDE here, as is evident by the parameters for each method - they only require a single spatial value x, along with the temporal parameter, t. The final method is init_cond, which allows an initial condition to be applied to the PDE.

The big difference between BlackScholesPDE and ConvectionDiffusionPDE, apart from the abstractness of the latter, is that BlackScholesPDE contains a public pointer member to a VanillaOption class, which is where it obtains the parameters necessary for the calculation of the coefficients.

Here is the listing for pde.h:

```
#ifndef __PDE_H
#define __PDE_H

#include "option.h"

// Convection Diffusion Equation - Second-order PDE
class ConvectionDiffusionPDE {
 public:
  // PDE Coefficients
  virtual double diff_coeff(double t, double x) const = 0;
  virtual double conv_coeff(double t, double x) const = 0;
  virtual double zero_coeff(double t, double x) const = 0;
  virtual double source_coeff(double t, double x) const = 0;

  // Boundary and initial conditions
  virtual double boundary_left(double t, double x) const = 0;
  virtual double boundary_right(double t, double x) const = 0;
  virtual double init_cond(double x) const = 0;
};

// Black-Scholes PDE
class BlackScholesPDE : public ConvectionDiffusionPDE {
 public:
  VanillaOption* option;
  BlackScholesPDE(VanillaOption* _option);

  double diff_coeff(double t, double x) const;
  double conv_coeff(double t, double x) const;
  double zero_coeff(double t, double x) const;
  double source_coeff(double t, double x) const;

  double boundary_left(double t, double x) const;
  double boundary_right(double t, double x) const;
  double init_cond(double x) const;
};

#endif
```

The source file pde.cpp implements the virtual methods for the BlackScholesPDE class. In particular, the diffusion, convection, zero-term and source coefficients are provided, based on the Black-Scholes equation. In this instance, x is the spot price.

diff_coeff , conv_coeff, zero_coeff and source_coeff are self-explanatory from the Black-Scholes PDE. However, note that the way to access the parameters from the option is through the dereferenced pointer member access syntax. option−>r is equivalent to (*option).r.

The right boundary condition makes use of Put-Call Parity and is actually a Dirichlet specification. The left boundary is also Dirichlet and set to 0.

The syntax for init_cond may require some explanation. Essentially the option and subsequent pay-off are being dereferenced by the pointer member access syntax at which point the function call operator **operator**() is called on the pay-off. This is possible because it is a functor.

Here is a listing for pde.cpp:

```
#ifndef __PDE_CPP
#define __PDE_CPP

```

```
#include "pde.h"
#include <math.h>

BlackScholesPDE::BlackScholesPDE(VanillaOption* _option) : option(_option)
    {}

// Diffusion coefficient
double BlackScholesPDE::diff_coeff(double t, double x) const {
  double vol = option->sigma;
  return 0.5*vol*vol*x*x;  // \frac{1}{2} \sigma^2 S^2
}

// Convection coefficient
double BlackScholesPDE::conv_coeff(double t, double x) const {
  return (option->)*x;  // rS
}

// Zero-term coefficient
double BlackScholesPDE::zero_coeff(double t, double x) const {
  return -(option->r);  // -r
}

// Source coefficient
double BlackScholesPDE::source_coeff(double t, double x) const {
  return 0.0;
}

// Left boundary-condition (vanilla call option)
double BlackScholesPDE::boundary_left(double t, double x) const {
  return 0.0;  // Specifically for a CALL option
}

// Right boundary-condition (vanilla call option)
double BlackScholesPDE::boundary_right(double t, double x) const {
  // This is via Put-Call Parity and works for a call option
  return (x-(option->K)*exp(-(option->r)*((option->T)-t)));
}
```

```
// Initial condition (vanilla call option)
double BlackScholesPDE::init_cond(double x) const {
  return option->pay_off->operator()(x);
}

#endif
```

17.3.4 FDM Class

The separation of the PDE from the Finite Difference Method to solve it means that we need a separate inheritance hierarchy for FDM discretisation. FDMBase constitutes the abstract base class for a FDM solver specific to a convection-diffusion PDE. It contains members for parameters related to spatial discretisation, temporal discretisation, time-marching, the coefficients for the actual derivative approximation as well as storage for the current and previous solutions. The comments in the listing should be explanatory for each of the values.

There are five pure virtual methods to implement in the derived classes. calculate_step_sizes is called on construction and populates all of the spatial and temporal step sizes. set_initial_conditions makes use of the PDE itself, and subsequently the option pay-off, to create the solution pay-off profile at expiry, i.e. the "initial" condition. calculate_boundary_conditions is called on every time-step to set the boundary conditions. In the case of FDMEulerExplicit, with the Black-Scholes European vanilla call, these are Dirichlet conditions.

calculate_inner_domain updates all solution discretisation points which do not lie on the boundary. The bulk of the work is carried out in this method. The client interacts with an FDM solver via the public method step_march, which performs the actual time looping across the temporal domain.

Here is the listing for fdm.h:

```
#ifndef __FDM_H
#define __FDM_H

#include "pde.h"
#include <vector>

// Finite Difference Method - Abstract Base Class
class FDMBase {
```

```
protected:
  ConvectionDiffusionPDE* pde;

  // Space discretisation
  double x_dom;         // Spatial extent [0.0, x_dom]
  unsigned long J;      // Number of spatial differencing points
  double dx;            // Spatial step size (calculated from above)
  std::vector<double> x_values;  // Stores the coordinates of the x
      dimension

  // Time discretisation
  double t_dom;         // Temporal extent [0.0, t_dom]
  unsigned long N;      // Number of temporal differencing points
  double dt;            // Temporal step size (calculated from above)

  // Time-marching
  double prev_t, cur_t;    // Current and previous times

  // Differencing coefficients
  double alpha, beta, gamma;

  // Storage
  std::vector<double> new_result;    // New solution (becomes N+1)
  std::vector<double> old_result;    // Old solution (becomes N)

  // Constructor
  FDMBase(double _x_dom, unsigned long _J,
          double _t_dom, unsigned long _N,
          ConvectionDiffusionPDE* _pde);

  // Override these virtual methods in derived classes for
  // specific FDM techniques, such as explicit Euler, Crank-Nicolson, etc.
  virtual void calculate_step_sizes() = 0;
  virtual void set_initial_conditions() = 0;
  virtual void calculate_boundary_conditions() = 0;
  virtual void calculate_inner_domain() = 0;
```

```
 public:
  // Carry out the actual time-stepping
  virtual void step_march() = 0;
};

class FDMEulerExplicit : public FDMBase {
 protected:
  void calculate_step_sizes();
  void set_initial_conditions();
  void calculate_boundary_conditions();
  void calculate_inner_domain();

 public:
  FDMEulerExplicit(double _x_dom, unsigned long _J,
                   double _t_dom, unsigned long _N,
                   ConvectionDiffusionPDE* _pde);

  void step_march();
};

#endif
```

The source listing for the FDM hierarchy, fdm.cpp contains some tricky code. We'll run through each part separately below. Here's the full listing before we get started:

```
#ifndef __FDM_CPP
#define __FDM_CPP

#include <fstream>
#include "fdm.h"

FDMBase::FDMBase(double _x_dom, unsigned long _J,
                 double _t_dom, unsigned long _N,
                 ConvectionDiffusionPDE* _pde)
  : x_dom(_x_dom), J(_J), t_dom(_t_dom), N(_N), pde(_pde) {}

FDMEulerExplicit::FDMEulerExplicit(double _x_dom, unsigned long _J,
                                   double _t_dom, unsigned long _N,
                                                ConvectionDiffusionPDE* _pde)
  : FDMBase(_x_dom, _J, _t_dom, _N, _pde) {
  calculate_step_sizes();
  set_initial_conditions();
}

void FDMEulerExplicit::calculate_step_sizes() {
  dx = x_dom/static_cast<double>(J-1);
  dt = t_dom/static_cast<double>(N-1);
}

void FDMEulerExplicit::set_initial_conditions() {
  // Spatial settings
  double cur_spot = 0.0;

  old_result.resize(J, 0.0);
  new_result.resize(J, 0.0);
  x_values.resize(J, 0.0);

  for (unsigned long j=0; j<J; j++) {
    cur_spot = static_cast<double>(j)*dx;
    old_result[j] = pde->init_cond(cur_spot);
    x_values[j] = cur_spot;
  }

  // Temporal settings
  prev_t = 0.0;
  cur_t = 0.0;
}

void FDMEulerExplicit::calculate_boundary_conditions() {
  new_result[0] = pde->boundary_left(prev_t, x_values[0]);
  new_result[J-1] = pde->boundary_right(prev_t, x_values[J-1]);
}

void FDMEulerExplicit::calculate_inner_domain() {
  // Only use inner result indices (1 to J-2)
  for (unsigned long j=1; j<J-1; j++) {
    // Temporary variables used throughout
    double dt_sig = dt * (pde->diff_coeff(prev_t, x_values[j]));
    double dt_sig_2 = dt * dx * 0.5 * (pde->conv_coeff(prev_t, x_values[j])
        );

    // Differencing coefficients (see \alpha, \beta and \gamma in text)
    alpha = dt_sig - dt_sig_2;
    beta = dx * dx - (2.0 * dt_sig) + (dt * dx * dx *
        (pde->zero_coeff(prev_t, x_values[j])));
    gamma = dt_sig + dt_sig_2;

    // Update inner values of spatial discretisation grid (Explicit Euler)
    new_result[j] = ( (alpha * old_result[j-1]) +
                      (beta * old_result[j]) +
                      (gamma * old_result[j+1]) )/(dx*dx) -
      (dt*(pde->source_coeff(prev_t, x_values[j])));
  }
}

void FDMEulerExplicit::step_march() {
  std::ofstream fdm_out("fdm.csv");

  while(cur_t < t_dom) {
    cur_t = prev_t + dt;
    calculate_boundary_conditions();
    calculate_inner_domain();
    for (int j=0; j<J; j++) {
      fdm_out << x_values[j] << "␣" << prev_t << "␣"
          << new_result[j] << std::endl;
    }

    old_result = new_result;
    prev_t = cur_t;
  }

  fdm_out.close();
```

```
}

#endif
```

Let's go through each part in turn. The first thing to note is that we're including the fstream library. This is necessary to output the solution surface to disk:

```
#include <fstream>
```

You'll notice that we actually need to implement a constructor for the abstract base class FDMBase. This is because it is actually storing member data and so we need to call this constructor from a derived class' member initialisation list:

```
FDMBase::FDMBase(double _x_dom, unsigned long _J,
                 double _t_dom, unsigned long _N,
                 ConvectionDiffusionPDE* _pde)
  : x_dom(_x_dom), J(_J), t_dom(_t_dom), N(_N), pde(_pde) {}
```

The constructor for FDMEulerExplicit initialises the parent class members, as well as calls the methods to fill in the step sizes and initial conditions:

```
FDMEulerExplicit::FDMEulerExplicit(double _x_dom, unsigned long _J,
                                   double _t_dom, unsigned long _N,
                                   ConvectionDiffusionPDE* _pde)
  : FDMBase(_x_dom, _J, _t_dom, _N, _pde) {
  calculate_step_sizes();
  set_initial_conditions();
}
```

For N temporal discretisation points we have $N - 1$ intervals. Similarly for the spatial discretisation. This method calculates these steps values for later use:

Note that in more sophisticated finite difference solvers the step size need not be constant. One can create "stretched grids" where areas of interest (such as boundaries) are given greater resolution at the expense of poorer resolution where it is less important.

```
void FDMEulerExplicit::calculate_step_sizes() {
  dx = x_dom/static_cast<double>(J-1);
  dt = t_dom/static_cast<double>(N-1);
}
```

Now that the step sizes are set it is time to pre-fill the initial condition. All of the spatial arrays are set to have J points and are zeroed. Then the method loops these arrays and uses the pointer to the PDE to obtain the initial condition as a function of spot. We also have a useful "helper" array, x_values which stores the spot value at each discretisation point to save us calculating it every time step. Finally, we set the current and previous times to zero:

```
void FDMEulerExplicit::set_initial_conditions() {
  // Spatial settings
  double cur_spot = 0.0;

  old_result.resize(J, 0.0);
  new_result.resize(J, 0.0);
  x_values.resize(J, 0.0);

  for (unsigned long j=0; j<J; j++) {
    cur_spot = static_cast<double>(j)*dx;
    old_result[j] = pde->init_cond(cur_spot);
    x_values[j] = cur_spot;
  }

  // Temporal settings
  prev_t = 0.0;
  cur_t = 0.0;
}
```

Now that the initial conditions are set we can begin the time-marching. However, in each time step we must first set the boundary conditions. In this instance the edge points (at index 0 and index $J-1$) are set using the PDE boundary_left and boundary_right method. This is an example of a Dirichlet condition. More sophisticated boundary conditions approximate the derivative at these points (Neumann conditions), although we won't be utilising these conditions here:

```
void FDMEulerExplicit::calculate_boundary_conditions() {
  new_result[0] = pde->boundary_left(prev_t, x_values[0]);
  new_result[J-1] = pde->boundary_right(prev_t, x_values[J-1]);
}
```

The "meat" of the FDM solver occurs in the calculate_inner_domain method. Let's run through how it works. A loop is carried out over the spatial cells - but only those away from the boundary, hence the index 1 to $J-2$. In order to save excessive repetitive calculation (and enhance readability), we create two helper variables called dt_sig and dt_sig_2. If you refer back to the mathematical formalism at the start of the chapter, it will be apparent where these variables arise from.

The next step is to calculate the α, β and γ coefficients which represent the algebraic re-arrangement of the derivative discretisation. Again, these terms will be clear from the above mathematical formalism. In fact, I've tried to make the C++ implementation as close as possible to the mathematical algorithm in order to make it easier to follow! Notice that we need to obtain certain coefficients via pointer dereferencing of the underlying PDE.

Once α, β and γ are defined we can use the finite differencing to update the solution into the new_result vector. This formula will also be clear from the mathematical algorithm above:

```
void FDMEulerExplicit::calculate_inner_domain() {
  // Only use inner result indices (1 to J-2)
  for (unsigned long j=1; j<J-1; j++) {
    // Temporary variables used throughout
    double dt_sig = dt * (pde->diff_coeff(prev_t, x_values[j]));
    double dt_sig_2 = dt * dx * 0.5 * (pde->conv_coeff(prev_t, x_values[j])
        );

    // Differencing coefficients (see \alpha, \beta and \gamma in text)
    alpha = dt_sig - dt_sig_2;
    beta = dx * dx - (2.0 * dt_sig) + (dt * dx * dx *
        (pde->zero_coeff(prev_t, x_values[j])));
    gamma = dt_sig + dt_sig_2;

    // Update inner values of spatial discretisation grid (Explicit Euler)
    new_result[j] = ( (alpha * old_result[j-1]) +
                      (beta * old_result[j]) +
                      (gamma * old_result[j+1]) )/(dx*dx) -
      (dt*(pde->source_coeff(prev_t, x_values[j])));
  }
}
```

Now that all of the prior pure virtual methods have been given an implementation in the derived FDMEulerExplicit class it is possible to write the step_march method to ”wrap everything together”.

The method opens a file stream to disk and then carries out a while loop in time, only exiting when the current time exceeds the maximum domain time. Within the loop time is advanced by the time-step dt. In each step the boundary conditions are reapplied, the inner domain new solution values are calculated and then this new result is output to disk sequentially. The old solution vector is then set to the new solution vector and the looping continues. Finally, we close the file stream. This concludes the implementation of FDMEulerExplicit:

```
void FDMEulerExplicit::step_march() {
  std::ofstream fdm_out("fdm.csv");

  while(cur_t < t_dom) {
    cur_t = prev_t + dt;
    calculate_boundary_conditions();
    calculate_inner_domain();
    for (int j=0; j<J; j++) {
      fdm_out << x_values[j] << " " << prev_t << " "
          << new_result[j] << std::endl;
    }

    old_result = new_result;
    prev_t = cur_t;
  }

  fdm_out.close();
}
```

17.3.5 Main Implementation

It is now time to tie all of the previous components together. The main implementation is actually quite straightforward, which is a consequence of the design choices we made above. Firstly, we include the relevant header files. Then we define the parameters of our European vanilla call option. The next stage is to define the discretisation parameters for the finite difference solver. In this instance we're using a mesh 20x20 representing a domain $[0.0, 1.0] \times [0.0, 1.0]$ in size.

Once the option and FDM parameters have been declared, the next step is to create the pay-off, call option, Black-Scholes PDE and the FDM solver objects. Since we're dynamically allocating some of our objects we need to make sure we call delete prior to the pointers going out of scope.

After the declaration of all the relevant solver machinery, we call the public step_march method of the FDMEulerExplicit instance, which then carries out the solution, outputting it to fdm.csv.

```
#include "payoff.h"
#include "option.h"
#include "pde.h"
#include "fdm.h"

int main(int argc, char **argv) {
  // Create the option parameters
  double K = 0.5;  // Strike price
  double r = 0.05;   // Risk-free rate (5%)
  double v = 0.2;    // Volatility of the underlying (20%)
  double T = 1.00;    // One year until expiry

  // FDM discretisation parameters
  double x_dom = 1.0;       // Spot goes from [0.0, 1.0]
  unsigned long J = 20;
  double t_dom = T;         // Time period as for the option
  unsigned long N = 20;

  // Create the PayOff and Option objects
  PayOff* pay_off_call = new PayOffCall(K);
  VanillaOption* call_option = new VanillaOption(K, r, T, v, pay_off_call);

  // Create the PDE and FDM objects
  BlackScholesPDE* bs_pde = new BlackScholesPDE(call_option);
  FDMEulerExplicit fdm_euler(x_dom, J, t_dom, N, bs_pde);

  // Run the FDM solver
  fdm_euler.step_march();

  // Delete the PDE, PayOff and Option objects
  delete bs_pde;
  delete call_option;
  delete pay_off_call;

  return 0;
}
```

17.4 Execution and Output

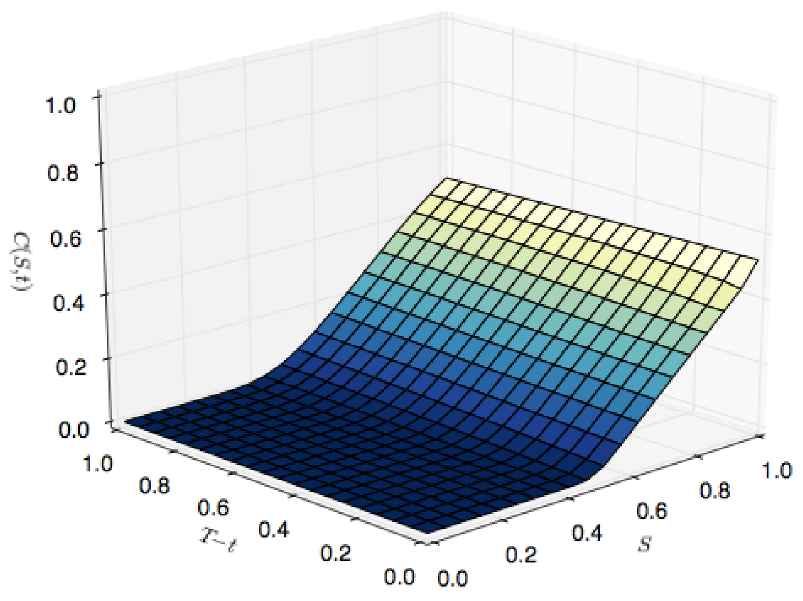
Upon execution of the code a file fdm.csv is generated in the same directory as main.cpp. I've written a simple Python script (using the matplotlib library) to plot the option price surface $C(S, t)$ as a function of time to expiry, t, and spot price, S.

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib
import numpy as np
from matplotlib import cm
from matplotlib import pyplot as plt

x, y, z = np.loadtxt('fdm.csv', unpack=True)

X = np.reshape(x, (20,20))
Y = np.reshape(y, (20,20))
Z = np.reshape(z, (20,20))

print X.shape, Y.shape, Z.shape

step = 0.04
maxval = 1.0
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=cm.YlGnBu_r)
ax.set_zlim3d(0, 1.0)
ax.set_xlabel(r'$S$')
ax.set_ylabel(r'$T-t$')
ax.set_zlabel(r'$C(S,t)$')
plt.show()
```

The output of the code is provided in Figure 17.1.

Note that at $T - t = 0.0$, when the option expires, the profile follows that for a vanilla call option pay-off. This is a non-smooth piecewise linear pay-off function, with a turning point at the point at which spot equals strike, i.e. $S = K$. As the option is "marched back in time" via the FDM solver, the profile "smooths out" (see the smoothed region around $S = K$ at $T - t = 1.0$). This is exactly the same behaviour as in a forward heat equation, where heat diffuses from an

Figure 17.1: European vanilla call price surface $C(S,t)$



initial profile to a smoother profile.

Bibliography

[1] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied.* Addison-Wesley, 2001.

[2] M. Baxter and A. Rennie. *Financial Calculus.* Cambridge University Press, 1999.

[3] M. Broadie and O. Kaya. Exact simulation of stochastic volatility and other affine jump diffusion processes. *Journal of Operations Research*, 54:217–231, 2006.

[4] D.J. Duffy. *Financial instrument pricing using C++.* Wiley, 2004.

[5] D.J. Duffy. *Introduction to C++ for financial engineers.* Wiley, 2006.

[6] D.J. Duffy and J. Kienitz. *Monte Carlo frameworks.* Wiley, 2009.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[8] P. Glasserman. *Monte Carlo Methods in Financial Engineering.* Springer, 2003.

[9] G. H. Golub and C. F. van Loan. *Matrix Computations (4th Ed).* Johns Hopkins University Press, 2012.

[10] J. Hull. *Options, Futures and Other Derivatives, 8th Ed.* Pearson, 2011.

[11] M. S. Joshi. *C++ Design Patterns and Derivatives Pricing, 2nd Ed.* Cambridge University Press, 2008.

[12] M. S. Joshi. *The Concepts and Practice of Mathematical Finance, 2nd Ed.* Cambridge University Press, 2008.

[13] N. M. Josuttis. *The C++ Standard Library: A Tutorial and Reference (2nd Ed).* Addison-Wesley, 2012.

[14] A. Koenig and B. E. Moo. *Accelerated C++.* Addison-Wesley, 2000.

[15] R. Korn. *Monte Carlo Methods and Models in Finance and Insurance.* CRC Press, 2010.

[16] R. Merton. Option pricing when the underlying stock returns are discontinuous. *Journal of Financial Economics*, 3:125–144, 1976.

[17] S. Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs.* Addison Wesley, 1995.

[18] S. Meyers. *Effective STL: 50 Specific Ways to Improve the Use of the Standard Template Library.* Addison Wesley, 2001.

[19] S. Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs, 3rd Ed.* Addison Wesley, 2005.

[20] W. H. Press. *Numerical Recipes in C, 2nd Ed.* Cambridge University Press, 1992.

[21] S. E. Shreve. *Stochastic Calculus for Finance I - The Binomial Asset Pricing Model.* Springer, 2004.

[22] S. E. Shreve. *Stochastic Calculus for Finance II - Continuous Time Models.* Springer, 2004.

[23] B. Stroustrup. *The C++ Programming Language, 4th Ed.* Addison-Wesley, 2013.

[24] H. Sutter. *Exceptional C++.* Addison-Wesley, 2000.

[25] L. N. Trefethen and D. Bau III. *Numerical Linear Algebra.* Society for Industrial and Applied Mathematics, 1997.

[26] P. Wilmott. *Paul Wilmott Introduces Quantitative Finance, 2nd Ed.* John Wiley & Sons, 2007.