Chapter 0

My First Monte Carlo Application One-Factor Problems

'Get it working, then get it right, then get it optimized'

0.1 Introduction and Objectives

The goal of this book is to develop algorithms for pricing a range of derivatives products and then mapping these algorithms to C++ code. The technique that we use is the Monte Carlo method. Part I of this book introduces the fundamental mathematical concepts, algorithms and a number of C++ programming techniques that are needed when using the Monte Carlo method.

The main goal in this chapter is to design and implement an initial C++ framework - consisting of a set of loosely coupled classes - that calculate the price of plain one-factor options. The framework has limited functionality and scope but it is structured in such a way that it can be - and will be - extended and generalized to n-factor and path-dependent problems as well as to problems with early exercise features. Some of the features discussed in this chapter are:

- Simple one-dimensional stochastic differential equations (SDE), their formulation and how to find exact and approximate solutions to them.

- Generation of uniform and normal random numbers using random number generators such as Polar Marsaglia and Box Muller.
- A short introduction to the Monte Carlo method and we apply it to finding the price of a one-factor European option.

In general, we partition the Monte Carlo application into three main subsystems. The core process in the application is to calculate the price of an option by simulation of the path of the underlying variable. We can also develop functionality that displays other relevant information, for example sensitivities or statistics.

This chapter is a self-contained introduction to the Monte Carlo method and its realisation in C++. It can be read by those who are new to the subject as well as by those who have programming experience. You can test your knowledge of C++ by examining and running the corresponding code on the CD. **If the syntax is difficult to understand, then this means that your C++ knowledge needs to be refreshed!**

0.2 Description of the Problem

We focus on a linear, constant-coefficient, scalar (one-factor) problem. In particular, we examine the case of a one-factor European call option using the assumptions for the original Black Scholes equation (see Clewlow 1998). We give an overview of the process. At the expiry date $t = T$ the option price is known as a function of the current stock price and the strike price. The essence of the Monte Carlo method is that we carry out a *simulation experiment* by finding the solution of a *stochastic differential equation* (SDE) from time $t = 0$ to time $t = T$. This process allows us to compute the stock price at $t = T$ and then the option price using the payoff function. We carry out M *simulations* or *draws* by finding the solution of the SDE and we calculate the option price at $t = T$. Finally, we calculate the discounted average of the simulated payoff and we are done.

Summarising, the process is:

1. Construct a simulated path of the underlying stock.
2. Calculate the stock price at $t = T$.
3. Calculate the call price at $t = T$ (use the payoff function).

   Execute steps 1-3 M times.

4. Calculate the averaged call price at $t = T$.
5. Discount the price found in step 4 to $t = 0$.

We elaborate this process in the rest of this chapter. We first need to provide some background information.

0.3 Ordinary Differential Equations (ODE)

We examine some simple ODEs. In general, the specification of an initial value problem (IVP) for an ODE is given by:

$$\begin{aligned} &\tfrac{du}{dt} + a(t)u = f(t), \quad 0 < t \leq T \\ &u(0) = A \end{aligned} \tag{0.1}$$

We see that the IVP consists of a linear ODE with a corresponding *initial condition* A. The term $f(t)$ is sometimes called the *inhomogeneous forcing term*. We note that all functions in system (1) are deterministic. We can find an exact solution to the system (1) by using the *integrating factor method*; in the case when the forcing term is identically zero the solution to (1) is given by:

$$u(t) = A \exp\left(- \int_0^t a(s) ds \right) \tag{0.2}$$

ODEs can be used to model simple problems in quantitative finance, for example bond modelling where the interest rate $r(t)$ is a

deterministic function of time. If V is the price of the security, then it satisfies the *terminal value problem* (TVP) (Wilmott 1995) page 267:

$$\begin{aligned} &\tfrac{dV}{dt} + K(t) = r(t)V, \quad K(t) = \text{coupon payment} \\ &V(T) = Z \end{aligned} \tag{0.3}$$

We see that the solution is given at $t = T$. This is in contrast to system (1) where the value is given at $t = 0$ (we can reduce (3) to an initial value problem of the type (1) by using a change of variables $\tau = T - t$). We see that the solution of system (3) is given by:

$$V(t) = \exp\left(-\int_t^T r(s)ds\right)\left\{Z + \int_t^T K(y)\exp\left(\int_y^T r(s)ds\right)dy\right\} \tag{0.4}$$

If we have a look at system (1) again we can see that it is possible to integrate it between times 0 and t:

$$u(t) - u(0) + \int_0^t a(s)u(s)ds = \int_0^t f(s)ds, \quad 0 < t \leq T \tag{0.5}$$

This is called a *Volterra integral equation of the second kind* and is given formally as:

$$\begin{aligned} &u(t) + \int_0^t a(s)u(s)ds = F(t) \\ &\text{where } F(t) = \int_0^t f(s)ds + u(0) = \int_0^t f(s)ds + A \end{aligned} \tag{0.6}$$

In this case the function $a = a(s)$ plays the role of the *kernel* and the limit of integration t is variable.

0.4 Stochastic Differential Equations (SDE) and their Solution

We now generalize the deterministic equations that we introduced in section 3. In this case we add some random or stochastic terms to the ODE. To this end, we introduce notation that is used in texts on stochastic equations. In general, we denote dependence on time t for a stochastic process X as follows:

$$X_t \equiv X(t) \tag{0.7}$$

Both forms are used in the literature. We must be aware of each form because they are used in many places, including this book. Our first example of an SDE is:

$$\begin{aligned} &dX_t = aX_t dt + bX_t W_t \quad a, b \text{ constant} \\ &W_t = W(t) \text{ (one-dimensional Brownian motion)} \end{aligned} \tag{0.8}$$

In this case the equation describes the changes in the stochastic process X_t. The constants a and b are called the *drift* and *diffusion* terms, respectively. Furthermore, we see the presence of the Wiener process W_t. We shall deal with these topics in more detail in later chapters.

A more general SDE is:

$$\begin{aligned} &dS_t = \mu(t)S_t dt + \sigma(t)S_t dW_t \\ &\text{where} \\ &\mu(t) \text{ is the drift coefficient} \\ &\sigma(t) \text{ is the diffusion coefficient} \end{aligned} \tag{0.9}$$

This is a model for the evolution of a stock in a certain time interval. Again, we note the presence of a Wiener process in the

equation. We usually write the SDE in the 'differential' form (9); we can also write the SDE in the integral form:

$$S_t = S_0 + \int_0^t \mu(y) S_y dy + \int_0^t \sigma(y) S_y dW_y \tag{0.10}$$

This equation characterizes the behaviour of the continuous time stochastic process S_t as the sum of a Lebesque integral and an Ito integral. A heuristic explanation of the SDE (10) is that the stochastic process S_t changes by an amount that is normally distributed. We say that the stochastic process is a *diffusion process* and is an example of a *Markov process*. When the drift and diffusion terms in system (9) are constant we can express the solution in analytic form:

$$S_t = S_0 \exp((\mu - \frac{1}{2}\sigma^2)t + \sigma W_t) \tag{0.11}$$

Finally, the values of the stochastic process at two different points are related by:

$$\begin{aligned} &S_{t+\Delta t} = S_t \exp((\mu - \tfrac{1}{2}\sigma^2)\Delta t + \sigma(W_{t+\Delta t} - W_t)) \\ &\text{where } \Delta t > 0 \text{ is arbitrary} \end{aligned} \tag{0.12}$$

You can check that this result is correct.

0.5 Generating Uniform and Normal Random Numbers

We discuss the generation of random numbers. In particular, we generate random Gaussian numbers for the Wiener process appearing in equations (8) and (9).

0.5.1 Uniform Random Number Generation

Our starting point is the generation of numbers having a *uniform distribution*. To this end, let us suppose that X is a continuous random variable assuming all values in the closed interval $[a, b]$, where

a and b are finite. If the *probability density function* (pdf) of X is given by:

$$f(x) = \begin{cases} \frac{1}{b-a}, & a \leq x \leq b \\ \\ 0, \text{ otherwise} \end{cases} \tag{0.13}$$

then we say that X is *uniformly distributed* over the interval $[a, b]$. A shorthand notation is to say that X is $U(a, b)$. We generate uniformly distributed random numbers by using an algorithm that has been programmed in C or C++.

We now introduce two methods that were popular a number of years ago. We include a discussion of them for historical and pedagogical reasons. We also introduced them in the context of the numerical solution of SDEs in Duffy 2004.

0.5.2 Polar Marsaglia Method

This method uses the fact that if the random variable U is U(0, 1) then the random variable V defined by $V = 2U - 1$ is $U(-1, 1)$. We now choose two variables defined by:

$$V_j = 2U_j - 1, \quad U_j \sim U(0, 1), \quad j = 1, 2 \tag{0.14}$$

Then we define:

$$W = V_1^2 + V_2^2 \leq 1, \quad W \sim U(0, 1)$$

We keep trying with different values until the above inequality is satisfied. Continuing, we define the intermediate value:

$$Y = \sqrt{-2\log(W)/W}$$

Finally, the pair of values defined by:

$$N_j = V_j Y, \quad j = 1, 2$$

constitutes two standard normally (Gaussian) distributed random variables, and we are done.

0.5.3 Box-Muller Method

This method is based on the observation that if r and φ are two independent $U(0,1)$ random variables then the variables:

$$\begin{aligned} N_1 &= \sqrt{-2\log r}\cos(2\pi\varphi) \\ N_2 &= \sqrt{-2\log r}\sin(2\pi\varphi) \end{aligned} \tag{0.15}$$

are two independent standard Gaussian random variables. See addendum chapter 0.

0.5.4 C++ Code for Uniform and Normal Random Variate Generation

Central to the accuracy of the Monte Carlo method is the ability to generate normal random variates. In this section we create C++ classes that compute these numbers. In general, we first generate uniform random variates and based on these variates we then generate the corresponding normal variates. We discuss these issues more detail in chapter 22. At this stage however, we just need to use the corresponding generators. Please note that we use the authors' classes for vectors and matrices (as described in Duffy 2004, Duffy 2006).

The abstract base class for uniform generators is:

```
class UniformGenerator
{
private:

public:
     UniformGenerator();

     // Initialisation and setting the seed
     virtual void init(long Seed) = 0;

     // Getting random structures (Template Method pattern)
     virtual double getUniform() = 0;// Number in range (0,1), variant
     Vector<double, long> getUniformVector(long N); // Invariant part


};
```

Derived classes must implement the pure virtual functions for defining a seed and a single random number. For example, the generator for uniform random variate generation based on the `rand()` function is:

```
class TerribleRandGenerator : public UniformGenerator
{ // Based on the infamous rand(), that's why it's terrible

private:
     double factor;

public:
     TerribleRandGenerator();

     // Initialise the seed, among others
     void init(long Seed_);

     // Implement (variant) hook function
     double getUniform();

};
```

We have defined other generators and the C++ code can be found on the CD. Continuing, the abstact base class for normal random variate generation is given by:

```
class NormalGenerator
{
protected:

     UniformGenerator* ug;     // This is a strategy object

public:
     NormalGenerator(UniformGenerator& uniformGen);

     // Getting random structures (Template Method pattern)
     virtual double getNormal() = 0;   // Get a number in (0,1)

     Vector<double, long> getNormalVector(long N);
     NumericMatrix<double, long> getNormalMatrix(long N, long M);
};
```

The code that generates a vector of normal random variates is:

```
// Getting random structures (Template Method Pattern)
Vector<double, long> NormalGenerator::getNormalVector(long N)
{ // Invariant part

     Vector<double, long> vec(N);
```

```
     for(long i=vec.MinIndex(); i<=vec.MaxIndex();  ++i)
     {
          vec[i] = getNormal();
     }

     return vec;
}
```

Derived classes must implement the pure virtual function `getNormal()` for defining a random number. For example, here is the interface for generating standard normal variates based on the Box-Muller method:

```
class BoxMuller : public NormalGenerator
{
private:
     double U1, U2;     // Uniform numbers
     double N1, N2;     // 2 Normal numbers as product of BM

     double W;
     const double tpi;
public:
     BoxMuller(UniformGenerator& uniformGen);

     // Implement (variant) hook function
     double getNormal();
};
```

The code for this generator is based on equation (15) and is given by:

```
// Implement (variant) hook function
double BoxMuller::getNormal()
{

     U1 = ug->getUniform();
     U2 = ug->getUniform();
     W = sqrt( -2.0 * log(U1));

     N1 = W * cos(tpi * U2);
     N2 = W * sin(tpi * U2);

     return N1;
}
```

We give an example of use. First, we define a uniform random variate and we then use this object to generate a normal random variate:

```
     // Based on rand()
     TerribleRandGenerator myTerrible;
```

```
myTerrible.init(0);
NormalGenerator* myNormal = new BoxMuller(myTerrible);
Vector<double, long> arr2 = myNormal->getNormalVector(100);

NumericMatrix<double, long> mat =
     myNormal -> getNormalMatrix(5, 6);

delete myNormal;
```

Finally, here is an example using a Numerical Recipes (Press 2002) algorithm:

```
Ran1 myRan1;          // This is a derived class
myRan1.init(447);
Vector<double, long> arrRan1 = myRan1.getUniformVector(20);
```

0.5.5 Other Methods

The above methods are somewhat outdated. They have been superceded by other methods such as Mersenne-Twister and lagged Fibonacci generators and by methods for generating random numbers on multi-processor computers. We shall discuss these issues in more detail in later chapters.

0.6 The Monte Carlo Method

In this section we describe the steps to price a one-factor option using the Monte Carlo method. We have already assembled the building blocks in the previous sections. We describe the algorithm and we create working C++ code that computes pricing information for a simple one-factor model. There are several advantages associated with this approach. First, it is a concrete problem that we solve in detail while the simple software framework will be generalised to one based on design and system patterns in later chapters. We recall the SDE that describes the behaviour of a dividend-paying stock. The SDE is given by:

$$dS_t = (r - D)S_t dt + \sigma S_t dW_t \tag{0.16}$$

where

r = (constant) interest rate

D = constant dividend

σ = constant volatility

dW_t = increments of the Wiener process

For the current problem it is possible to transform the SDE to a simpler one. To this end, define the variable:

$$x_t = \log(S_t) \tag{0.17}$$

Then the modified SDE is described as:

$$dx_t = \nu dt + \sigma dW_t, \quad \nu \equiv r - D - \frac{1}{2}\sigma^2 \tag{0.18}$$

We now discretise this equation by replacing the continuously-defined quantities dx_t, dt and dW_t by their discrete analogues. This gives the following discrete stochastic equation:

$$\Delta x_t = \nu \Delta t + \sigma \Delta W_t$$

or (0.19)

$$x_{t+\Delta t} = x_t + \nu \Delta t + \sigma(W_{t+\Delta t} - W_t)$$

Since $x_t = log(S_t)$ we can see that equation (19) is equivalent to a discrete equation in the stock price S_t:

$$S_{t+\Delta t} = S_t \exp(\nu \Delta t + \sigma(W_{t+\Delta t} - W_t)) \tag{0.20}$$

In this case the Wiener increments have distribution:

$$N(0, \Delta t) \tag{0.21}$$

Formulae (19) and (20) constitutes the basic *path-generation algorithm*: we calculate the value in equation (20) at a set of discrete *mesh points*:

$$
\begin{aligned}
&0 = t_0 < t_1 < \ldots < t_{N-1} < t_N = T \\
&t_n = n\Delta t, \quad n = 0, .., N \\
&\Delta t = N/T
\end{aligned}
$$

Then equation (20) takes the more computationally attractive form:

$$
\begin{aligned}
&x_{t_n} = x_{t_{n-1}} + \nu\Delta t + \sigma\sqrt{\Delta t}\epsilon_n, \quad n = 1, .., N \text{ where} \\
&\epsilon_n \text{ is a sample from } N(0,1) \text{ and } S_{t_n} = \exp(x_{t_n})
\end{aligned}
\tag{0.22}
$$

The next step is to run the algorithm in equation (22) M times (M is called the number of *simulations* or *draws*); for each price of the stock at $t = T$ we calculate the payoff function for a call option, namely:

$$
\text{payoff} = \max(0, S_T - K) \tag{0.23}
$$

This formula gives the value of the payoff, and we are done, almost. Finally, we calculate the average call price over all call prices (evaluated at $t = T$) and we take the discounted average of these *simulated paths*. Summarising, the basic algorithm for a call option is given by:

$$
\begin{aligned}
&\text{For each } j = 1, .., M \text{ calculate} \\
&C_{T,j} = \max(0, S_{T,j} - K)
\end{aligned}
\tag{0.24}
$$

and

$$
\begin{aligned}
&\hat{C} = \exp(-rT)\tfrac{1}{M}\textstyle\sum_{j=1}^{M} \max(0, S_{T,j} - K) \\
&\text{Then } \hat{C} \text{ is the desired call price.}
\end{aligned}
\tag{0.25}
$$

0.7 Calculating Sensitivities

The Monte Carlo method calculates the price of an option for a specific value of the underlying stock. In some applications we wish

to calculate the derivative's *hedge sensitivities* (also known as the *Greeks*). These quantities are the partial derivatives of the price with respect to one of the option's parameters. Two of the most important ones are the *delta* and the *gamma*; in order to calculate these quantities we perturb the underlying price by a small amount and we use centred finite difference schemes to approximate the derivatives. For example, the formulae for delta and gamma are:

$$\begin{aligned} \delta = \frac{\partial C}{\partial S} \equiv \text{delta} &\sim \frac{C(S+\Delta S)-C(S-\Delta S)}{2\Delta S} \\ \Gamma = \frac{\partial^2 C}{\partial S^2} = \frac{\partial \delta}{\partial S} = \text{Gamma} &\sim \frac{\delta(S+\Delta S)-\delta(S-\Delta S)}{2\Delta S} \end{aligned} \tag{0.26}$$

In the above examples we have used centred-difference schemes to approximate the derivative. The approximations are second-order accurate if the security is sufficiently smooth. Failing that, we sometimes take first-order one-sided finite difference approximations (see Duffy 2006 for a discussion).

We need values for sensitivities in order to manage risk. This is a major challenge for a Monte Carlo engine, both from a theoretical and from a computational point of view. If we use finite difference methods, for example we tend to get biased estimates (Glasserman 2004).

We discuss some of the issues when computing sensitivities and we examine how to compute the delta as a representative example. To this end, we deploy the Monte Carlo engine with initial estimates of $S + \Delta S$ and $S - \Delta S$, where ΔS is a small fraction of S. We thus have to run the engine twice and we use the computed values in equation (26). On a single-processor CPU we must run the engine twice while on multi-processor CPUs we can run the calculations in parallel. Performance is an important issue when developing Monte Carlo code and we shall discuss this topic in more detail in chapters 24 and 25.

The problems associated with estimating sensitivities in the Monte Carlo method and the associated techniques for approximating them

are discussed in Part III.

0.8 The Initial C++ Monte Carlo Framework: Hierarchy and Paths

Since this a book on the Monte Carlo method, its mathematical foundations and its implementation in C++ we now discuss how to integrate these threads into a working C++ program. In order to reduce the scope we create the code that computes the price of a one-factor plain option. We exclude non-essential details from the discussion and we focus on those issues that will help the reader understand the 'big picture' as it were. To this end, we start with the design of the problem, which we have created using design patterns (we discuss these in more detail in Part II of this book) and documented using an UML (Unified Modeling Language) class diagram, as shown in Figure 1. Each box represents a class (this is usually an abstract base class) and it has a clearly defined responsibility; each class is assigned a code for convenience and it corresponds to one of the following activities:

- A1: Defining and initializing the SDE (class B).
- A2: Random number generator (RNG) (class D).
- A3: Approximating the solution of the SDE using finite differences (class C).
- A4: Displaying the results and statistics (in class A).

We have defined a sophisticated factory class E (called a *builder* that is responsible for the creation of the other objects, inter-object links and data in the application. Each class in Figure 1 has services (in the form of member functions) that it offers to other classes (called *clients*) while each class uses the services of other classes (called *servers*).

We describe the process for calculating option price as follows:

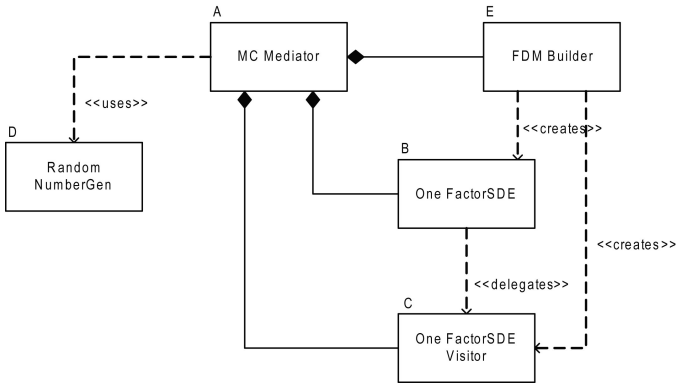

Figure 0.1: Top Level Class Diagram for simple MC Framework (including code names)

"We model a nonlinear one-factor SDE that describes the behaviour of an underlying asset. We approximate its solution by discretising the SDE using a finite-difference scheme (FDM), for example the Euler method. The FDM class uses the services of the RNG class that is an implementation of a random number generator (this is needed for the Wiener increments). The control of the program is the responsibility of the Mediator. Finally, the output is the responsibility of the Datasim Excel visualisation driver."

We now discuss classes B and C. These are the classes that model the various kinds of one-factor SDEs and the finite difference methods (FDM) that approximate them, respectively. The class hierarchy for SDEs is shown in Figure 2; the derived classes are defined in terms of whether the drift and diffusion terms are linear or nonlinear. For example, for type D equations both terms are nonlinear. We mention that there are various ways to create class hierarchies. The interface for the base class SDE is given by:

```
class OneFactorSDE
{
private:
     double ic;              // Initial condition
     Range<double> ran;      // Interval where SDE 'lives'
public:
     OneFactorSDE();
     OneFactorSDE(double initialCondition,
                  const Range<double>& interval)
     const double& InitialCondition() const;
     const Range<double>& Interval() const;
     double getExpiry() const;
```

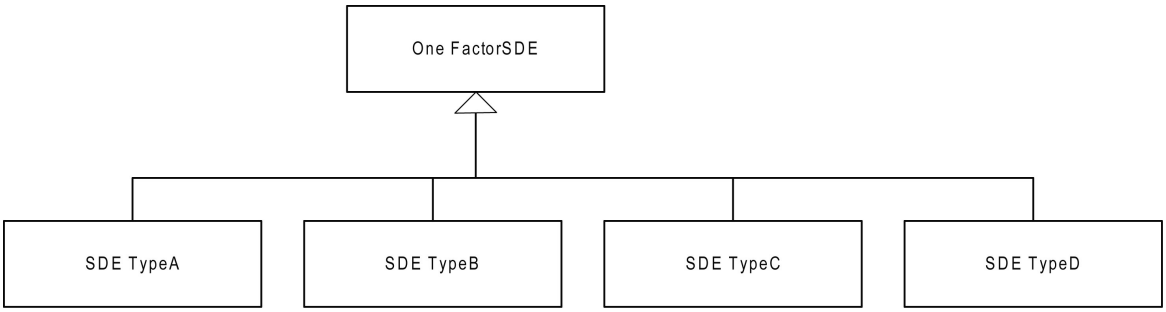

Figure 0.2: SDE Hierarchy

```
     // Functional extensions (Visitor pattern, see Part II)
     virtual void Accept(OneFactorSDEVisitor& visitor) = 0;
};
```

You should understand the syntax of this code because this book assumes that you are already a C++ developer.

We examine the type D class in this section. Its interface is given by:

```
class SDETypeD : public OneFactorSDE
{ // Nonlinear/nonlinear case

private:

     // General function pointers
     double (*drift) (double t, double X);
     double (*diffusion) (double t, double X);

public:
     SDETypeD() : OneFactorSDE();
     SDETypeD(double initialCondition, const Range<double>& interval,
              double (*driftFunction) (double t, double X),
              double (*diffusionFunction) (double t, double X));

     // Selector functions
     double calculateDrift (double t, double X) const;
     double calculateDiffusion (double t, double X) const

     virtual void Accept(OneFactorSDEVisitor& visitor);

};
```

We instantiate this class by defining the initial condition, the interval in which the SDE is defined as well as its drift and diffusion functions. In this chapter we define this information in a namespace defined by:

```
namespace ExactSDE
{ // Known solution, dS = aSdt + bSdW

     double a; // Drift
     double b; // Diffusion

     double drift(double t, double X)
     {

          return a*X;

     }

     double diffusion(double t, double X)
     {

          return b*X;
     }

} // End ExactSDE
```

We shall see how this information is used to instantiate a type D class.

Turning to the finite difference schemes we show the class hierarchy in Figure 3. We have implemented it as a *Visitor* design pattern (GOF 1995) because this allows us to add new functionality to an SDE in a non-intrusive way. We discuss this pattern in more detail in Part II. The top-level abstract base class in Figure 3 has the interface:

```
class OneFactorSDEVisitor
{ // Inline code

private:
public:
     // Constructors and Destructor
     OneFactorSDEVisitor() {}                // Default constructor
     OneFactorSDEVisitor(const OneFactorSDEVisitor& source) {}
     virtual ~OneFactorSDEVisitor() {}

     // The visit functions
     virtual void Visit(SDETypeA& sde)=0;    // Linear/Linear
     //virtual void Visit(SDETypeB& sde)=0;  // L/NL (for reader)
     //virtual void Visit(SDETypeC& sde)=0;  // NL/L (for reader)
     virtual void Visit(SDETypeD& sde)=0;    // NL/NL


     // Operators
     OneFactorSDEVisitor& operator =
            (const OneFactorSDEVisitor& source) {}
};
```

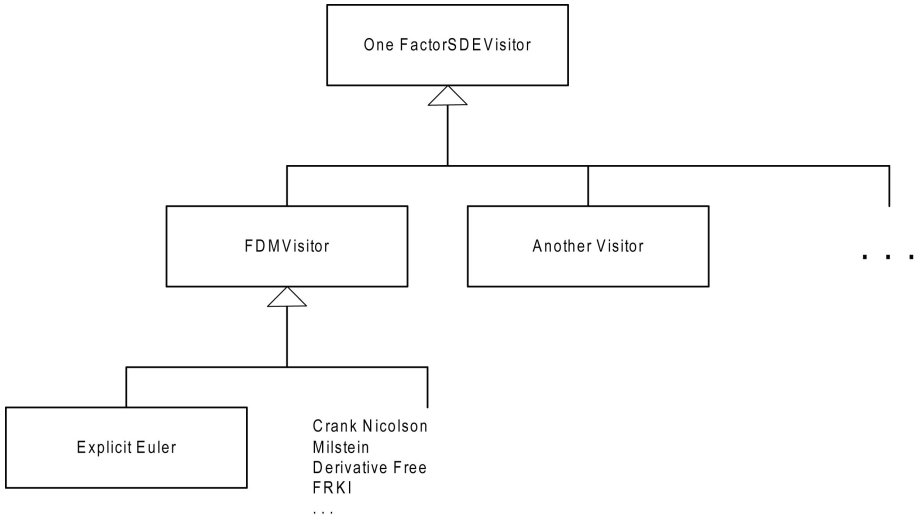

Figure 0.3: FDM Hierarchy

The base class for all finite difference schemes has the interface:

```
class FDMVisitor : public OneFactorSDEVisitor
{ // Base class for all schemes

//protected:
public: // For convenience, ONLY

     // Initial conditions
     double initVal, VOld;
     // Solution at time level n+1
     double VNew;

     // Mesh data (Vector<T,I> is authors' class)
     Vector<double, long> x;
     double k;                  // Time step
     double sqrk; // Square root of k (for dW stuff)

     // Result path
     Vector<double, long> res;

     // Random numbers
     Vector<double, long> dW;

     // Number of steps
     long N;

public:
     FDMVisitor(long NSteps, const Range<double>& interval,
                double initialValue);
     FDMVisitor(long NSteps, const OneFactorSDE& sde);
```

```
     void SetRandomArray(const Vector<double,long>& randomArrray);

     virtual Vector<double, long> path() const;

     long getNumberOfSteps() const;
};
```

We can now define (many) specific finite difference schemes to approximate the various SDE types. We examine the explicit Euler scheme for convenience; its interface is defined by:

```
class ExplicitEuler : public FDMVisitor
{
public:
     ExplicitEuler(long NSteps, const Range<double> & interval,
                   double initialValue);
     ExplicitEuler(long NSteps, const OneFactorSDE& sde);

     void Visit(SDETypeA& sde);
     void Visit(SDETypeD& sde);
};
```

The corresponding code file is:

```
// Euler Method
ExplicitEuler::ExplicitEuler(long NSteps, const Range<double>& interval,double initialValue )
                       : FDMVisitor(NSteps, interval, initialValue)
{


}

ExplicitEuler::ExplicitEuler(long NSteps, const OneFactorSDE& sde)
                       : FDMVisitor(NSteps, sde)
{


}

void ExplicitEuler::Visit(SDETypeA& sde)
{
     VOld = initVal;
     res[x.MinIndex()] = VOld;
     for (long index = x.MinIndex()+1; index <= x.MaxIndex(); ++index)
     {
          VNew = VOld *(1.0 + k * sde.calculateDrift(x[index-1])+
            sqrk * sde.calculateDiffusion(x[index-1]) * dW[index-1]);

          res[index] = VNew;
          VOld = VNew;
     }
```

```
}

void ExplicitEuler::Visit(SDETypeD& sde)
{
     VOld = initVal;
     res[x.MinIndex()] = VOld;
     for (long index = x.MinIndex()+1; index <= x.MaxIndex(); ++index)
     {
          VNew = VOld  + k * sde.calculateDrift(x[index-1], VOld)+
            sqrk*sde.calculateDiffusion(x[index-1], VOld)*dW[index-1];

          res[index] = VNew;
          VOld = VNew;
     }
}
```

We now discuss some schemes to calculate the path of the following nonlinear autonomous SDE:

$$\begin{aligned} &dX(t) = \mu(X(t))dt + \sigma(X(t))dW(t) \quad 0 < t \leq T \\ &X(0) = A \end{aligned} \tag{0.27}$$

We discretise the interval [0, T] into N subintervals and we adopt the notation:

$$\mu_n \equiv \mu(X_n), \quad \sigma_n = \sigma(X_n) \quad n = 0, .., N, \ \Delta W_n = \sqrt{\Delta t} Z, \ Z \sim N(0, 1)$$

We have created classes for some other finite difference schemes (from Saito 1996) and you can find the code on the CD.

- Explicit Euler:

$$X_{n+1} = X_n + \mu_n \Delta t + \sigma_n \Delta W_n \tag{0.28}$$

- Semi-implicit Euler:

$$\begin{aligned} &X_{n+1} = X_n + [\alpha \mu_{n+1} + (1 - \alpha)\mu_n]\Delta t + \sigma_n \Delta W_n \\ &\text{with special cases} \\ &\alpha = \tfrac{1}{2} \text{ (Trapezoidal)} \\ &\alpha = 1 \text{ (Backward Euler)} \end{aligned} \tag{0.29}$$

- Heun:

$$X_{n+1} = X_n + \tfrac{1}{2}[F_1 + F_2]\Delta t + \tfrac{1}{2}[G_1 + G_2]\Delta W_n$$

where

$$\begin{aligned} &F(x) \equiv \mu(x) - \tfrac{1}{2}\sigma'(x)\sigma(x) \\ &\sigma'(x) \equiv \tfrac{d\sigma}{dx} \\ &F_1 = F(X_n), \quad G_1 = \sigma(X_n) \\ &F_2 = F(X_n + F_1\Delta t + G_1\Delta W_n) \\ &G_2 = \sigma(X_n + F_1\Delta t + G_1\Delta W_n) \end{aligned} \tag{0.30}$$

- Milstein:

$$X_{n+1} = X_n + \mu_n\Delta t + \sigma_n\Delta W_n + \tfrac{1}{2}[\sigma'\sigma]_n((\Delta W_n)^2 - \Delta t) \tag{0.31}$$

- Derivative-free:

$$X_{n+1} = X_n + F_1\Delta t + G_1\Delta W_n + [G_2 - G_1]\Delta t^{-1/2}\tfrac{(\Delta W_n)^2 - \Delta t}{2}$$

where

$$F_1 = \mu(X_n), \quad G_1 = \sigma(X_n)$$

$$G_2 = \sigma(X_n + G_1\Delta t^{1/2}) \tag{0.32}$$

- First-order Runge Kutta with Ito coefficient (FRKI):

$$X_{n+1} = X_n + F_1\Delta t + G_2\Delta W_n + [G_2 - G_1]\Delta t^{1/2}$$

where

$$F_1 = \mu(X_n), \quad G_1 = \sigma(X_n) \tag{0.33}$$

$$G_2 = \sigma(X_n + \tfrac{G_1(\Delta W_n - \Delta t^{1/2})}{2})$$

We now discuss the code for calculating a path of the constant-coefficient SDE:

$$\begin{aligned} &dX(t) = \mu X dt + \sigma X dW(t), \quad 0 \leq t \leq T \\ &X(0) = 1 \end{aligned} \tag{0.34}$$

whose exact solution is:

$$X(t) = \exp\left\{(\mu - \frac{1}{2}\sigma^2)t + \sigma W(t)\right\} \tag{0.35}$$

In this case we compare the path taken by the exact solution and the solution delivered by the scheme (0.32); we generate normal random variates using classes that we discussed in section 5. The code is:

```
int main()
{

     // Define the array of random numbers and use this one
     long N = 3000;
     cout << "Give number of paths: "; cin >> N;

     // Create a U(0,1) and N(0,1) array
     MSSecureRandGenerator myUniform;           // .NET generator
     myUniform.init(0);
     NormalGenerator* myNormal = new BoxMuller(myUniform);
     Vector<double, long> dW = myNormal->getNormalVector(N+1);
     delete myNormal;


     // Create the basic SDE (Context class)
     double T = 1.0;
     cout << "Give the value of T:"; cin >> T;

     Range<double> ran (0.0, T);
     Vector<double, long> x = ran.mesh(N);
     double ic = 1.0;

     // Choose the drift and diffusion functions
     // using namespace GenericNLDiffusion;
     using namespace ExactSDE;
     cout << "Give drift: "; cin >> a;
     cout << "Give diffusion: "; cin >> b;

     // Create the nonlinear SDE object
     SDETypeD sde(ic, ran, drift, diffusion);
```

```
    // Choose the FDM scheme (Visitor pattern)
    DerivativeFree visitor (N, ran, ic);
    // or Euler method . . .

    // Choose the array of N(0,1) numbers
    visitor.SetRandomArray(dW);

    // Calculate the FDM 'path'
    sde.Accept(visitor);
    Vector<double, long> result = visitor.path();

    try
    {
    printOneExcel(x,                  // Abscissa array
        result,
         string("Derivative Free"),   // The title of the chart
         string("time"),              // The title of x axis
         string("Underlying"),        // The title of y axis
         string("Value"));            // Legend describing the line graph
    }
    catch(DatasimException& e)
    { // Catch logical errors

         e.print();
    }

    Vector<double, long> exactSolution =
                    GBMRandomWalk(ic, a, b, T, N, dW);

    try
    {
    printOneExcel(x,                      // Abscissa array
         exactSolution,
         string("Exact path solution"),   // The title of the chart
         string("time"),                  // The title of x axis
         string("Underlying"),            // The title of y axis
         string("Value"));                // Legend describing the line graph
    }
    catch(DatasimException& e)
    { // Catch logical errors

         e.print();
    }
    return 0;
}
```

You can run this program to see the results. The output is presented in Excel. You can easily adapt the code to test other schemes, for example the Euler method.

0.9 The Initial C++ Monte Carlo Framework: Calculating Option Price

We now extend the results of section 8. In particular, we focus on:

- Modelling option data and payoff functions.
- The structure of Monte Carlo algorithm.
- Initialising and creating the object network in Figure 1.

First, since we are interested in calculating option prices it is useful to encapsulate all relevant information in a structure. To this end, we define the structure:

```
struct OptionData
{ // Option data + behaviour

     double K;
     double T;
     double r;
     double sig;

     double myPayOffFunction(double S)
     { // Call option


          return max (S - K, 0);
    }
};
```

All data and functions are public (by default). Second, the `MCMediator` class integrates all control and data flow in the application. Its internal structure and public interface reflect the UML class diagram in Figure 1:

```
class MCMediator
{
private:
     FDMTypeDBuilder* bld;    // Creates SDE and FDM objects
     long NSim;               // Number of simulations,for discounting
     OptionData* opt;

     double (*payoff) (double S);

     OneFactorSDE* sde;
     FDMVisitor* fdm;
public:
     MCMediator(FDMTypeDBuilder& builder, long NSimulations,
```

```
                OptionData& optionData);
    double price() const;
};
```

Here we see that this class has a member function `price()` for calculating the option price. We break the body of this function into logical blocks and we explain each block in turn:

A. Loop over each iteration.

B. Find value at $t = T$.

C. Return the vectors and calculate payoff vectors and average.

D. Discount the value.

We first need to define arrays, variables and other 'work' data structures:

```
// The critical variables in the MC process
double price;     // Option price

// Create the random numbers
long N = fdm->getNumberOfSteps();

// The array of values of the payoff function at t = T
Vector<double, long > TerminalValue(NSim, 1);

// The vector of UNDERLYING values at t = T;
Vector<double, long> result(N+1, 1);

// Array of normal random numbers
Vector<double, long> dW(N+1, 1);

// Create a U(0,1) and N(0,1) array
MSSecureRandGenerator myUniform;        // .NET generator
//TerribleRandGenerator myUniform;      // rand()
// OR YOUR FAVOURITE!
myUniform.init(0);
NormalGenerator* myNormal = new BoxMuller(myUniform);

// Work variables
double sqrT = sqrt(opt->T);
```

Step A involves calculating the paths based on the SDE and finite difference method that we have already discussed in section 8. We produce a vector `TerminalValue` of size `NSim` containing the values of the underlying stock at $t = T$;

```
// A.
```

```
for (long i = 1; i <= NSim; ++i)
{ // Calculate a path at each iteration

     if ((i/10000) * 10000 == i)
     {
         cout << i << endl;

     }

     dW = myNormal->getNormalVector(N+1);       // Performance bottleneck!
     fdm ->SetRandomArray(dW);

     // Calculate the path by delegation to a visitor
     sde -> Accept(*fdm);
     result = fdm->path(); // Size is N+1

     // Now us this array in the payoff function
     TerminalValue[i] = result[result.MaxIndex()];


}
```

Step B involves the evaluation of the payoff function at $t = T$:

```
//  B. Calculate the payoff function for each asset price
for (long index = TerminalValue.MinIndex();
     index <= TerminalValue.MaxIndex(); ++index )
{
     TerminalValue[index] =
        opt->myPayOffFunction(TerminalValue[index]);

}
```

Step C involves taking the average option price:

```
//  C. Take the average
price = TerminalValue[TerminalValue.MinIndex()];

for (long ii = TerminalValue.MinIndex()+1;
     ii <= TerminalValue.MaxIndex(); ++ii)
{
        price += TerminalValue[ii];

}
```

Finally, step D discounts the option price:

```
price = price / double(NSim);

// D. Finally, discounting the average price
price *= exp(-opt->r * opt->T);
```

```
    // Print out critical values
    cout << "price, after discounting: " << price << endl;;

    // Cleanup; V2 use scoped pointer
    delete myNormal;

    return price;
```

We are now finished because we have described all the steps in the algorithm. Finally, the main program that ties in all the classes in Figure 0.1 is:

```
int main()
{
    // Initialise the option data
    OptionData callOption;
    callOption.K = 65.0;
    callOption.T = 0.25;
    callOption.r = 0.08;
    callOption.sig = 0.30; // IC = 60

    // Create the basic SDE (Context class)
    Range<double> range (0.0, callOption.T);
    double initialCondition = 60.0;

    // Discrete stuff

    long N = 100;
    cout << "Number of subintervals: ";
    cin >> N;


    // Tell the Builder what kinds of SDE and FDM Types you want
    // You can use these to test SDE and FDM
    // enum SDEType {A, B, C, D};
    // enum FDMType {Euler, PC, CN, /*MIL*/, SIE,
    // IE, DerivFree, FRKIto, Fit};

    // The builder creates the UML class diagram,chapter 0 of MC book
    FDMTypeDBuilder fdmBuilder(FDMTypeDBuilder::D,
          FDMTypeDBuilder::DerivFree, N, range, initialCondition, drift, diffusion);


    // V2 mediator stuff
    long NSimulations = 50000;
    cout << "Number of simulations: ";
    cin >> NSimulations;


    try
    {
         MCMediator mediator(fdmBuilder, NSimulations, callOption);
```

```
        cout << "Final Price: " << mediator.price() << endl;
    }
    catch(string& exception)
    { // CN, IE or IE cannot be used with NL/NL SDEs
        cout << exception << endl;
        cout << "Press any key to stop\n";
        int yy; cin >> yy;
        exit(1);
    }

    return 0;
}
```

You can run this program and test it with your own values. See the CD for more details. The code can be optimized to make it run faster, but this is not a concern in this chapter. We have difined the code so that it can be easily understood and extended.

0.10 The Predictor-Corrector Method: a Scheme for all Seasons?

In this chapter we have introduced a number of schemes for one-factor SDEs and you can experiment with the C++ code to see how well they approximate the solution of linear and nonlinear equations. By experimenting with a range of input parameters you will get a feeling for which schemes are suitable for your situation.

In this chapter we discuss a robust scheme that is be used for one-factor and multi-factor SDEs in finance. It is called the predictor-corrector method and is a generalization of a similar scheme that is used to solve ordinary differential equations (Lambert 1991). In order to apply it to SDEs we examine the nonlinear problem given by equation (0.27). If we discretise this SDE using the trapezoidal rule, for example we will get a nonlinear system of equations at each time level that we need to solve using Newton's or Steffensen's method:

$$X_{n+1} = X_n + \{\alpha\ \mu\ (X_{n+1}) + (1-\alpha)\ \mu\ (X_n)\}\ \Delta t$$
$$+ \{\beta\sigma(X_{n+1}) + (1-\beta)\sigma(X_n)\}\ \Delta W_n, n \geq 0 \tag{0.36}$$

where

$$0 \leq \alpha \leq 1, \quad 0 \leq \beta \leq 1$$

Instead of solving this equation we attempt to linearise it by replacing the 'offending' unknown solution at time level n + 1 by another known quantity (called the *predictor*) that is hopefully close to it. To this end, we compute the predictor as:

$$X_{n+1} = X_n + \mu(X_n)\Delta t + \sigma(X_n)\Delta W_n, \quad n \geq 0 \tag{0.37}$$

that we subsequently use in the corrector equation:

$$X_{n+1} = X_n + \{\alpha\mu(X_{n+1}) + (1-\alpha)\mu(X_n)\}\ \Delta t$$
$$+ \{\beta\sigma(X_{n+1}) + (1-\beta)\sigma(X_n)\}\ \Delta W_n, \quad n \geq 0 \tag{0.38}$$

However, we modify equation (0.8) by using the so-called *corrector drift function*:

$$\overline{\mu}_\beta(x) = \mu(x) - \beta\sigma(x)\frac{\delta\sigma}{\delta x}(X) \tag{0.39}$$

The new corrector equation is given by:

$$X_{n+1} = X_n + \left\{\alpha\overline{\mu}_\beta(X_{n+1}) + (1-\alpha)\overline{\mu}_\beta(X_n)\right\}\ \Delta t$$
$$+ \{\beta\sigma(X_{n+1}) + (1-\beta)\sigma(X_n)\}\ \Delta W_n, n \geq 0 \tag{0.40}$$

This solution of this scheme can be solved without the need to use a nonlinear solver. Furthermore, you can customize the scheme to

support different levels of implicitness and explicitness in the drift and diffusion terms:

$$\begin{array}{l} \text{A. Fully explicit}(\alpha = \beta = 0) \\ \text{B. Fully implicit}(\alpha = \beta = 1) \\ \text{C. Implicit in drift explicit in diffusion } (\alpha = 1, \beta = 0) \\ \text{D. Symmetric}(\alpha = \beta = 1/2) \end{array} \tag{0.41}$$

We need to determine which combination of parameters results in stable schemes. We have seen with the schemes in this chapter that reducing the time steps (which is the same as increasing the number of subdivisions of the interval [0,T]) does not increase accuracy and may actually lead to serious inaccuracies. This phenomenon is not common when discretising deterministic equations and can come as a shock if you are not prepared. We have experimented with a number of test cases and the general conclusion is that the most stable schemes are those that have the same level of implicitness in drift and diffusion, for example the options A, B and D in the equation (0.41). In particular, the symmetric case D is a good all-rounder in the sense that it gives good results for a range of SDEs. Finally, the predictor-corrector method is popular in interest-rate applications using the Libor Market Model (LMM) (see Gatarek 2006).

0.11 The Monte Carlo Approach: Caveats and Nasty Surprises

In this section we give a short description of the problems that we encountered when designing and implementing the algorithms related to the Monte Carlo method. In general, some of the obstacles we have encountered are sparsely documented in the literature:

- Many existence and uniqueness results are based on measure theory. Some authors use functional analysis to prove the same results for stochastic differential equations (SDEs). The advan-

tage in the latter case is that these methods allow us to actually construct the solution to such equations. In general, we feel that there is too much emphasis on measure theory in this field.

- The popular numerical schemes (finite difference schemes) for SDEs tend to be extensions of similar numerical schemes for ordinary differential equations (ODEs). This analogy does not always work in practice and we have seen that the most popular schemes fail to realize the levels of robustness and accuracy that they produce in the ODE case.

- Standard numerical schemes break down for stiff SDEs and SDEs

- A number of widely-used random number generators (the C function rand(), for example) do not work properly. We shall discuss their shortcomings in this book and we provide improved random number generators. In particular, we use the Mersenne Twister random number generator in this book. We advise against using the Box-Muller and Polar Marsaglia methods because their use leads to the so-called Neave effect (Neave 1973, Chay 1975, Jckel 2002). This phenomenon describes the poor agreement between observed and expected frequencies in the tails of the normal distribution when the random normal deviates are generated by a multiplicative congruential scheme. The problem can be resolved by reversing the order of the members of the pairs of successive pseudo-random as in equation (0.15). The discrepancy between observed and expected values in the tails is due to the manner in which the two random numbers in equation (0.15) are generated and the resulting dependence between these two numbers. In short, we interchange the members of the pairs of random numbers generated by multiplicative congruential schemes.

Summarising, we feel that there is more fundamental and applied research to be done in this area.

0.12 Summary and Conclusions

We have written this chapter for the benefit of those readers for whom stochastic analysis and the Monte Carlo method are new. Our intention was to describe a simple - but complete - Monte Carlo engine for a one-factor European option. In particular, we developed enough background theory that allowed us to code a simple application in C++.

We generalize the results in this chapter to more interesting and complex products as we progress in this book. You should be able to understand the C++ code in this chapter. **If you do not, it will be difficult to continue with the more advanced chapters in this book.**

0.13 Exercises and Projects

1. (*)Prove (by using a change of variables) that the system (3) can be posed as an IVP of the form (1).
2. (*)Use the integrating factor method to find a closed form solution to system (1) (hint: use formula (4)).
3. (*)Prove that if the random variable x is $U(0,1)$ then the random variable y defined by $y = 2x - 1$ is $U(-1,1)$ (remark: this fact is used in the Polar Marsaglia method)
4. (*)Verify equation (18) when using the change of variables (17).
5. (***)We have used the same notation (namely the variable x_t) for the solutions of equations (18) and (19). Mathematically speaking, this is incorrect because these are two different equations; one solution (the *continuous solution*) solves equation (18) while the other solution (the *discrete solution*) solves equation (19). How can we assure that the discrete solution is a *good* approximation to the continuous solution (We discuss this topic in chapter 3)?

6. (***) Determine how to incorporate the following sensitivities into the Monte Carlo engine:

$$\text{vega} = \frac{\partial C}{\partial \sigma} \cong \frac{C(\sigma+\Delta\sigma)-C(\sigma-\Delta\sigma)}{2\Delta\sigma}$$

$$\text{theta} = \frac{\partial C}{\partial t} \cong \frac{C(t+\Delta t)-C(t)}{\Delta t}$$

$$\text{rho} = \frac{\partial C}{\partial r} \cong \frac{C(r+\Delta r)-C(r-\Delta r)}{2\Delta r}$$

7. (**) We consider a formula (it is first-order accurate) for approximating the delta of an option, namely:

$$\delta = \frac{C(S+\Delta S)-C(S)}{\Delta S}$$

Compare this representation with equation (26). What are the performance differences?

8. (***) We wish to *stress test* the code in section 9 by varying the input parameters. For example, test long-dated options:

```
OptionData callOption;
callOption.K = 100.0;
callOption.T = 30.0;
callOption.r = 0.08;
callOption.sig = 0.3;

double initialCondition = 100.0;
```

Answer the following questions:

a. Calculate the call option price using the code on the CD. Compare the answer with the exact solution using the Black Scholes formula. What do you notice?

b. Now compute the put price by modifying the payoff function in `OptionData`. To find the corresponding call price, use the *put-call parity relationship* between the price of a European call option and a European put option having the same strike price K and maturity date T:

$$C + Ke^{-rT} = P + S$$

where C = call price, P = put price and S = stock price at t = 0. What do you notice now? Do you get a more accurate answer?

9. (**) An important issue in Monte Carlo simulation is *efficiency*; this refers to the time it takes to compute the option price. Examine the code and determine where performance can be improved (we discuss performance issues in chapters 21 and 25). In particular, examine how the array of normal random numbers is generated in the code.

10. (***) We wish to modify the code in section 9 to support some *path-dependent* options. In this exercise we examine a *down-and-out* call option. This is an option whose value is zero if the underlying stock reaches a *barrier level* H from above. Determine how to modify the code for this case.

    Modify the code so that the contribution to the final call price for a given simulation is zero if the simulated stock price is less than or equal to H. Test your code with problems whose solutions you know (Clewlow 1998, page 116, Haug 2007).

11. (**) This is an exercise for C++ novices and for those who need to refresh their knowledge before continuing with the more advanced topics in later chapters. To this end, consider the four categories of one-factor SDE:

Type A
$dX = \mu(t)Xdt + \sigma(t)XdW$

Type B
$dX = \mu(t)Xdt + \sigma(t, X)dW$

Type C
$dX = \mu(t, X)dt + \sigma(t)XdW$

Type D
$dX = \mu(t, X)dt + \sigma(t, X)dW$

We have already written the C++ code for types A and D. The objective of this exercise is to write the code for types B and C as well for the following kinds of SDEs that are used to model the *short rate*:

$dX = A(B - X)dt + \sigma\sqrt{X}dW$ where A, B and σ are constants

and

$dX = [\theta(t) - aX]dt + \sigma dW$
where
$\theta(t)$ defines the average direction that X moves at time t.

Test your software.

12. (***) Optimize the code in section 9 so that it runs faster.