1

A simple Monte Carlo model

1.1 Introduction

In the first part of this book, we shall study the pricing of derivatives using Monte Carlo simulation. We do this not to study the intricacies of Monte Carlo but because it provides many convenient examples of concepts that can be abstracted. We proceed by example, that is we first give a simple program, discuss its good points, its shortcomings, various ways round them and then move on to a new example. We carry out this procedure repeatedly and eventually end up with a fancy program. We begin with a routine to price vanilla call options by Monte Carlo.

1.2 The theory

We commence by discussing the theory. The model for stock price evolution is

$$dS_t = \mu S_t dt + \sigma S_t dW_t, \tag{1.1}$$

and a riskless bond, B, grows at a continuously compounding rate r. The Black–Scholes pricing theory then tells us that the price of a vanilla option, with expiry T and pay-off f, is equal to

$$e^{-rT}\mathbb{E}(f(S_T)),$$

where the expectation is taken under the associated risk-neutral process,

$$dS_t = rS_t dt + \sigma S_t dW_t. \tag{1.2}$$

We solve equation (1.2) by passing to the log and using Ito's lemma; we compute

$$d\log S_t = \left(r - \frac{1}{2}\sigma^2\right) dt + \sigma dW_t. \tag{1.3}$$

As this process is constant-coefficient, it has the solution

$$\log S_t = \log S_0 + \left(r - \frac{1}{2}\sigma^2\right) t + \sigma W_t. \tag{1.4}$$

Since W_t is a Brownian motion, W_T is distributed as a Gaussian with mean zero and variance T, so we can write

$$W_T = \sqrt{T}N(0,1), \tag{1.5}$$

and hence

$$\log S_T = \log S_0 + \left(r - \frac{1}{2}\sigma^2\right)T + \sigma\sqrt{T}N(0,1), \tag{1.6}$$

or equivalently,

$$S_T = S_0 e^{(r-\frac{1}{2}\sigma^2)T+\sigma\sqrt{T}N(0,1)}. \tag{1.7}$$

The price of a vanilla option is therefore equal to

$$e^{-rT}\mathbb{E}\left(f\left(S_0 e^{(r-\frac{1}{2}\sigma^2)T+\sigma\sqrt{T}N(0,1)}\right)\right).$$

The objective of our Monte Carlo simulation is to approximate this expectation by using the law of large numbers, which tells us that if Y_j are a sequence of identically distributed independent random variables, then with probability 1 the sequence

$$\frac{1}{N}\sum_{j=1}^{N} Y_j$$

converges to $\mathbb{E}(Y_1)$.

So the algorithm to price a call option by Monte Carlo is clear. We draw a random variable, x, from an $N(0, 1)$ distribution and compute

$$f\left(S_0 e^{(r-\frac{1}{2}\sigma^2)T+\sigma\sqrt{T}x}\right),$$

where $f(S) = (S - K)_+$. We do this many times and take the average. We then multiply this average by e^{-rT} and we are done.

1.3 A simple implementation of a Monte Carlo call option pricer

A first implementation is given in the program `SimpleMCMain1.cpp`.

Listing 1.1 (`SimpleMCMain1.cpp`)

```
//  requires Random1.cpp

#include <Random1.h>
#include <iostream>
#include <cmath>
using namespace std;
```

```
double SimpleMonteCarlo1(double Expiry,
                         double Strike,
                         double Spot,
                         double Vol,
                         double r,
                         unsigned long NumberOfPaths)
{
    double variance = Vol*Vol*Expiry;
    double rootVariance = sqrt(variance);
    double itoCorrection = -0.5*variance;

    double movedSpot = Spot*exp(r*Expiry +itoCorrection);
    double thisSpot;
    double runningSum=0;

    for (unsigned long i=0; i < NumberOfPaths; i++)
    {
    double thisGaussian = GetOneGaussianByBoxMuller();
    thisSpot = movedSpot*exp( rootVariance*thisGaussian);
    double thisPayoff = thisSpot - Strike;
    thisPayoff = thisPayoff >0 ? thisPayoff : 0;
    runningSum += thisPayoff;
    }

    double mean = runningSum / NumberOfPaths;
    mean *= exp(-r*Expiry);
    return mean;
}

int main()
{
    double Expiry;
    double Strike;
    double Spot;
    double Vol;
    double r;
    unsigned long NumberOfPaths;
    cout << "\nEnter expiry\n";
    cin >> Expiry;
```

```
    cout << "\nEnter strike\n";
    cin >> Strike;

    cout << "\nEnter spot\n";
    cin >> Spot;

    cout << "\nEnter vol\n";
    cin >> Vol;

    cout << "\nr\n";
    cin >> r;

    cout << "\nNumber of paths\n";
    cin >> NumberOfPaths;

    double result = SimpleMonteCarlo1(Expiry,
                                      Strike,
                                      Spot,
                                      Vol,
                                      r,
                                      NumberOfPaths);

    cout <<"the price is " << result << "\n";

    double tmp;
    cin >> tmp;

 return 0;
}
```

Our program uses the auxiliary files `Random1.h` and `Random1.cpp`.

Listing 1.2 (`Random1.h`)

```
#ifndef RANDOM1_H
#define RANDOM1_H

double GetOneGaussianBySummation();
double GetOneGaussianByBoxMuller();
#endif
```

Listing 1.3 (Random1.cpp)

```
#include <Random1.h>
#include <cstdlib>
#include <cmath>

// the basic math functions should be in namespace
// std but aren't in VCPP6
#if !defined(_MSC_VER)
using namespace std;
#endif

double GetOneGaussianBySummation()
{
double result=0;

for (unsigned long j=0; j < 12; j++)
result += rand()/static_cast<double>(RAND_MAX);

result -= 6.0;

return result;
}

double GetOneGaussianByBoxMuller()
{
double result;

double x;
double y;

double sizeSquared;
do
{
x = 2.0*rand()/static_cast<double>(RAND_MAX)-1;
y = 2.0*rand()/static_cast<double>(RAND_MAX)-1;
sizeSquared = x*x + y*y;
}
while
( sizeSquared >= 1.0);
```

```
result = x*sqrt(-2*log(sizeSquared)/sizeSquared);

return result;
}
```

We first include the header file Random1.h. Note that the program has `<Random1.h>` rather than `"Random1.h"`. This means that we have set our compiler settings to look for header files in the directory where `Random1.h` is. In this case, this is in the directory `C/include`. (In Visual C++, the directories for include files can be changed via the menus tools, options, directories.)

Random1.h tells the main file that the functions

```
double GetOneGaussianBySummation()
```

and

```
double GetOneGaussianByBoxMuller()
```

exist. We include the system file `iostream` as we want to use `cin` and `cout` for the user interface. The system file `cmath` is included as it contains the basic mathematical functions `exp` and `sqrt`.

We have the command `using namespace std` because all the standard library commands are contained in the namespace `std`. If we did not give the `using` directive, then we would have to prefix all their uses by `std::`, so then it would be `std::cout` rather than `cout`.

The function `SimpleMonteCarlo1` does all the work. It takes in all the standard inputs for the Black–Scholes model, the expiry and strike of the option, and in addition the number of paths to be used in the Monte Carlo.

Before starting the Monte Carlo we precompute as much as possible. Thus we compute the variance of the log of the stock over the option's life, the adjustment term $-\frac{1}{2}\sigma^2 T$ for the drift of the log, and the square root of the variance. Whilst we cannot precompute the final value of spot, we precompute what we can and put it in the variable `movedSpot`.

We initialize the variable, `runningSum`, to zero as it will store the sum so far of the option pay-offs at all times.

We now loop over all the paths. For each path, we first draw the random number from the $N(0, 1)$ distribution using the Box–Muller algorithm and put it in the variable `thisGaussian`.

The spot at the end of the path is then computed and placed in `thisSpot`. Note that although our derivation of the SDE involved working with the log of the spot, we have carefully avoided using log in this routine. The reason is that log and exp

are slow to compute in comparison to addition and multiplication, we therefore want to make as few calls to them as possible.

We then compute the call option's pay-off by subtracting the strike and taking the maximum with zero. The pay-off is then added to `runningSum` and the loop continues.

Once the loop is complete, we divide by the number of paths to get the expectation. Finally, we discount to get our estimate of the price which we return.

The `main` program takes in the inputs from the user, calls the Monte Carlo function, and displays the results. It asks for a final input to stop the routine from returning before the user has had a chance to read the results.

1.4 Critiquing the simple Monte Carlo routine

The routine we have written runs quickly and does what it was intended to do. It is a simple straightforward procedural program that performs as required. However, if we worked with this program we would swiftly run into annoyances. The essence of good coding is reusability. What does this mean? One simple definition is that code is reusable if someone has reused it. Thus reusability is as much a social concept as a technical one. What will make it easy for someone to reuse your code? Ultimately, the important attributes are clarity and elegance of design. If another coder decides that it would take as much effort to recode your routines as to understand them, then he will recode, and his inclination will be to recode in any case, as it is more fun than poring over someone else's implementation.

The second issue of elegance is equally important. If the code is clear but difficult to adapt then another coder will simply abandon it, rather than put lots of effort into forcing it to work in a way that is unnatural for how it was built.

The demands of reusability therefore mean we should strive for clarity and elegance. In addition, we should keep in mind when considering our original design the possibility that in future our code might need to be extended.

We return to our simple Monte Carlo program. Suppose we have a boss and each day he comes by and asks for our program to do something more. If we have designed it well then we will simply have to add features; if we have designed poorly then we will have to rewrite existing code.

So what might the evil boss demand?

"Do puts as well as calls!"

"I can't see how accurate the price is, put in the standard error."

"The convergence is too slow, put in anti-thetic sampling."

"I want the most accurate price possible by 9am tomorrow so set it running for 14 hours."

"It's crucial that the standard error is less than 0.0001, so run it until that's achieved. We're in a hurry though so don't run it any longer than strictly necessary."

"I read about low-discrepancy numbers at the weekend. Just plug them in and see how good they are."

"Apparently, standard error is a poor measure of error for low-discrepancy simulations. Put in a convergence table instead."

"Hmm, I miss the standard error can we see that too."

"We need a digital call pricer now!"

"What about geometric average Asian calls?"

"How about arithmetic average Asian puts?"

"Take care of variable parameters for the volatility and interest rates."

"Use the geometric Asian option as a control variate for the arithmetic one."

"These low-discrepancy numbers apparently only work well if you Brownian bridge. Put that in as well."

"Put in a double digital geometric Asian option."

"What about jump risk? Put in a jump-diffusion model."

To adapt the routine as written would require a rewrite to do any of these. We have written the simplest routine we could think of, without considering design issues. This means that each change is not particularly natural and requires extra work.

For example, with this style of programming how would we would do the put option?

Option one: copy the function, change the name by adding put at the end, and rewrite the two lines where the pay-off is computed.

Option two: pass in an extra parameter, possibly as an `enum` and compute the pay-off via a `switch` statement in each loop of the Monte Carlo. The problem with the first option is that when we come to the next task, we have to adapt both the functions in the same way and do the same thing twice. If we then need more pay-offs this will rapidly become a maintenance nightmare.

The issues with the other option are more subtle. One problem is that a switch statement is an additional overhead so that the routine will now run a little slower. A deeper problem is that when we come to do a different routine which also uses a pay-off, we will have to copy the code from inside the first routine or rewrite it as necessary. This once again becomes a maintenance problem; every time we want to add a new sort of pay-off we would have to go through every place where pay-offs are used and add it on.

A C style approach to this problem would be to use a function pointer, we pass a pointer to a function as an argument to the Monte Carlo. The function pointed to is then called via the pointer in each loop to specify the price. Note that the call to the function would have to specify the strike as well as spot since the function

could not know its value. Note also that if we wanted to do a double-digital option we would have problems as the double digital pays if and only if spot is between two levels, and we only have one argument, the strike, to play with.

The C++ approach to this problem is to use a class. The class would encapsulate the behaviour of the pay-off of a vanilla option. A pay-off object would then be passed into the function as an argument and in each loop a method expressing its value would be called to output the price for that pay-off. We look at the implementation of such a class in the next chapter.

1.5 Identifying the classes

In the previous section, we saw that the problem of wanting to add different sorts of vanilla options led naturally to the use of a class to encapsulate the notion of a pay-off. In this section, we look at identifying other natural classes which arise from the boss's demands.

Some of the demands were linked to differing forms that the boss wanted the information in. We could therefore abstract this notion by creating a statistics gatherer class.

We also had differing ways of terminating the Monte Carlo. We could terminate on time, on standard error or simply after a fixed number of paths. We could abstract this by writing a terminator class.

There were many different issues with the method of random number generation. The routine as it stands relies on the inbuilt generator which we do not know much about. We therefore want to be able to use other random number generators. We also want the flexibility of using low-discrepancy numbers which means another form of generation. (In addition, Box–Muller does not work well with low-discrepancy numbers so we will need flexibility in the inputs.) Another natural abstraction is therefore a random number generator class.

As long as our option is vanilla then specifying its parameters via pay-off and strike is fairly natural and easy; however, it would be neater to have one class that contains both pieces of information. More generally, when we pass to path-dependent exotic options, it becomes natural to have a class that expresses the option's properties. What would we expect such a class to do? Ultimately, an easy way to decide what the class should and should not know is to think of whether a piece of information would be contained in the term-sheet. Thus the class would know the pay-off of the option. It would know the expiry time. If it was an Asian it would know the averaging dates. It would also know whether the averaging was geometric or arithmetic. It would not know anything about interest rates, nor the value of spot nor the volatility of the spot rate as none these facts are contained in the term-sheet. The point here is that by choosing a real-world concept to

encapsulate, it is easy to decide what to include or not to include. It is also easy for another user to understand how you have decided what to include or not to include.

What other concepts can we identify? The concept of a variable parameter could be made into a class. The process from which spot is drawn is another concept. The variable interest rates could be encapsulated via a class that expresses the notion of a discount curve.

1.6 What will the classes buy us?

Suppose that having identified all these classes, we implement them. What do we gain?

The first gain is that because these classes encapsulate natural financial concepts, we will need them when doing other pieces of coding. For example, if we have a class that does yield curves then we will use it time and time again, as to price any derivative using any reasonable method involves knowledge of the discount curve. Not only will we save time on the writing of code but we will also save time on the debugging. A class that has been tested thoroughly once has been tested forever and in addition, any little quirks that evade the testing regime will be found through repeated reuse. The more times and ways something has been reused the fewer the bugs that will be left. So using reusable classes leads to more reliable code as well as saving us coding time. Debugging often takes at least as much time as coding in any case, so saving time on debugging is a big benefit.

A second gain is that our code becomes clearer. We have written the code in terms of natural concepts, so another coder can identify the natural concepts and pick up our code much more easily.

A third gain is that the classes will allow us to separate interface from implementation. All the user needs to know about a pay-off class or discount curve class are what inputs yield what outputs? How the class works internally does not matter. This has multiple advantages. The first is that the class can be reused without the coder having to study its internal workings. The second advantage is that because the defining characteristic of the class is what it does but not how it does it, we can change how it does it at will. And crucially, we can do this without rewriting the rest of our program. One aspect of this is that we can first quickly write a suboptimal implementation and improve it later at no cost. This allows us to provide enough functionality to test the rest of the code before devoting a lot of time to the class. A third advantage of separating interface from implementation is that we can write multiple classes that implement the same interface and use them without rewriting all the interface routines. This is one of the biggest advantages of object-oriented design.