Chapter 5

Monte Carlo Pricing

5.1 Simulating stock prices

5.1.1 The mathematics

We have already seen that we can price options by simulating in the risk-neutral measure, then computing expectations and discounting.

At the moment we can only simulate stock prices at a single time T, how can we simulate an entire history of stock prices? We will need to be able to do this to apply Monte Carlo pricing to path dependent options. Note that we should say what we mean by "simulating": we simply mean generating random variables which are identically distributed to the desired random variables.

We will suppose that we are working on the Black–Scholes model, so the stock price follows geometric Brownian motion.

$$\mathrm{d}S_t = S_t(\mu\,\mathrm{d}t + \sigma\,\mathrm{d}W_t)$$

As we saw in the last chapter, it is much easier to understand the process for the log of the stock price z_t.

$$\mathrm{d}z_t = \left(\mu - \frac{1}{2}\sigma^2\right)\mathrm{d}t + \sigma\,\mathrm{d}W_t$$

Integrating this we obtain

$$z_t = z_0 + \left(\mu - \frac{1}{2}\sigma^2\right)t + \sigma(W_t - W_0).$$

So to simulate z_T, we just need to generate variables which are normally distributed with mean $z_0 + (\mu - \frac{1}{2}\sigma^2)T$ and standard deviation $\sigma\sqrt{T}$. This is easy to do. We can then simulate S_T by exponentiating this. Let us summarize our findings.

Algorithm (Simulating Black–Scholes prices at time T)**.** *To simulate S_T at time T where*

$$\mathrm{d}S_t = S_t(\mu \, \mathrm{d}t + \sigma \, \mathrm{d}W_t)$$

(i) Generate a normally distributed random number ϵ with mean 0 *and standard deviation* 1*.*

(ii) Set

$$\tilde{s}_T = z_0 + \left(\mu - \frac{1}{2}\sigma^2\right) T + \sigma\sqrt{T}\epsilon$$

(iii) Set

$$\tilde{S}_T = \exp(\tilde{s}_T)$$

(iv) The probability density of

$$\tilde{S}_T$$

is the same as that of

$$S_T$$

(v) So $\tilde{S}_T$ is a simulated stock price.

One useful way to remember this method of simulating z_t is that it is a special case of the so-called Euler scheme for simulating stochastic differential equations (SDEs). The Euler scheme is a way of approximating stochastic differential equations with difference equations and we will discuss it in more detail in a later chapter. To obtain the Euler scheme from an SDE, one replaces all the d symbols with δ symbols that represent the change in a variable from one time to the next. One also replaces $\mathrm{d}W_t$ with a random variable $\sqrt{\delta t}\epsilon$, where ϵ is normally distributed with mean 0 and standard deviation 1.

For our example, we start with the process:

$$\mathrm{d}z_t = \left(\mu - \frac{1}{2}\sigma^2\right) \mathrm{d}t + \sigma \, \mathrm{d}W_t$$

We make the substitutions:

$$\mathrm{d}z_t \mapsto \delta z_T = z_T - z_0$$

$$\mathrm{d}t \mapsto \delta T = T - 0$$

$$\mathrm{d}W_t \mapsto \sqrt{\delta T}\,\epsilon.$$

To get:

$$z_T - z_0 = \left(\mu - \frac{1}{2}\sigma^2\right) T + \sigma\sqrt{T}\epsilon.$$

As we will see the Euler scheme allows you to *approximately* simulate many different stochastic differential equations. However, our algorithm allows us to *exactly* simulate geometric Brownian motion. Our simulated stock price will have *exactly* the same distribution as a stock price coming from geometric Brownian motion.

It is worth pausing to ask why we make the substitution

$$\mathrm{d}W_t \mapsto \sqrt{\delta T}\,\epsilon?$$

The reason is that the change in the log of the stock price over time Nh is composed of N independent small changes over time periods of length h. By the central limit theorem (assuming the small changes have finite variance), the change in the stock price over time Nh has variance proportional to $\sqrt{N}h$. Therefore, the cumulative effect of the noise grows at a rate proportional to $\sqrt{\delta T}$. Sigma is defined to be the constant of proportionality. This is why sigma has units of years$^{-1/2}$.

Simulating the stock price at times $0 = t_0 < t_1 < t_2 < \ldots < t_n$ means generating values $\tilde{S}_{t_1}, \tilde{S}_{t_2}, \ldots \tilde{S}_{t_n}$ so that the joint distribution function of $S_{t_1}, \ldots S_{t_n}$ is the same as that of $\tilde{S}_{t_1}, \ldots \tilde{S}_{t_n}$. By the Markov property of the stock price, we simply need to simulate the stock to time t_1, then use this as the starting point of a simulation up to time t_2 and so on up to time t_n. This allows us to write down an algorithm for simulating the stock price at multiple time points.

Algorithm. *To simulate a stock price following geometric Brownian motion at time points t_i*

(i) Define

$$\delta t_i = t_i - t_{i-1}$$

(ii) Define z_0 to be the log of the stock price at time 0.

(iii) Choose independent normally distributed ϵ_i with mean 0 *and standard deviation* 1.

(iv) Define

$$\tilde{z}_{t_i} = \tilde{z}_{t_{i-1}} + \left(\mu - \frac{1}{2}\sigma^2\right)\delta t_i + \sigma\sqrt{\delta t_i}\epsilon_i$$

(v) Define $\tilde{S}_{t_i} = \exp(z_{t_i})$.

$\tilde{S}_{t_i}$ simulates the stock price S_t at the desired times.

Of course, our first algorithm is a special case of this, but it is worth noticing that if you are only interested in stock prices at time T there is no need to simulate intermediate times.

5.1.2 MATLAB implementation

Let us start with a simple example problem.

Example 1: A stock follows the Black–Scholes price process with drift $\mu = 0.05$ and $\sigma = 0.1$. The initial stock price is $S_0 = 100$.

Simulate the stock price every day for 1 year and plot the result.

We begin by showing how to write a function that answers just this question directly and then try to generalize our code. The code below pretty much directly implements the algorithm from the previous section.

Note that we have used different notation when writing the code: we've written `dt` instead of δ_t and we've written `logS` instead of s. The reason for these changes is that I think they make the code easier to read. Also notice that the code stores every single simulated price in a vector `logS`, so this vector represents the entire history of the stock price.

```
% This is a very inefficient version of generateBSPaths
% which uses two for loops.
function  plotBSPath()
T = 1.0;
nSteps = 365;
S0 = 100;
mu = 0.05;
sigma = 0.1;

dt = T/nSteps;
logS0 = log( S0);
logS = zeros(nSteps,1);
for t=1:nSteps
    eps = randn();
    dlogS = (mu-0.5*sigma^2)*dt + sigma*sqrt(dt)*eps;
    if (t==1)
        lastLogS = logS0;
    else
        lastLogS = logS(t-1);
    end
    logS(t) = lastLogS + dlogS;
end

S = exp( logS );
times = (1:nSteps)*dt;
plot(times,S);

end
```

Don't worry about the comment about the inefficiency of the code, we will deal with this later on.

You should try running this code and experiment with different values of μ and σ.

Learning from our experience in previous chapters, we will try to improve this code by using MATLAB functions. We will write a function that takes the following parameters

- `S0`, `mu` and `sigma` describing the model
- Parameters `T` and `nSteps` describing the total time interval for the simulation and the number of equally sized pieces to divide the time interval into.
- A parameter `nPaths` indicating the number of price paths to be simulated.

We would like the function to return a matrix of simulated stock prices. Each row should correspond to a scenario and each column should correspond to a time step. For convenience, the function should also return a vector of time points indicating the time corresponding to each column.

Since we want to run multiple simulations, our code is essentially the same as that on previous slides but with an extra for loop to run through multiple scenarios. Here is our code.

```
% This is a very inefficient version of generateBSPaths
% which uses two for loops.
function [ S, times ] = generateBSPaths2Loops( ...
    T, S0, mu, sigma, nPaths, nSteps  )

dt = T/nSteps;
logS0 = log( S0);
logS = zeros(nPaths,nSteps);
for p=1:nPaths
    for t=1:nSteps
        eps = randn();
        dlogS = (mu-0.5*sigma^2)*dt + sigma*sqrt(dt)*eps;
        if (t==1)
            lastLogS = logS0;
        else
            lastLogS = logS(p,t-1);
        end
        logS(p,t) = lastLogS + dlogS;
    end
end
S = exp(logS);
times = dt:dt:T;
end
```

Note that the name of our function comes from the fact that our implementation contains two for loops. We will show now how to vectorize the code to remove the loops. This is the only reason we've given the function such a peculiar name.

As we have discussed in earlier chapters, in MATLAB you can often "vectorize" your code to eliminate loops. This makes your code easier to read and often makes it considerably faster. Whenever you repeat the same operation across independent scenarios, you can vectorize your code.

Here is a vectorized version of path generation code that eliminates the loop over scenarios.

```
% This is a reasonably efficient version
% of generateBSPaths that uses vectorization across
% scenarios
function [ S, times ] = generateBSPaths1Loop( ...
    T, S0, mu, sigma, nPaths, nSteps )

dt = T/nSteps;
logS0 = log( S0);
eps = randn(nPaths, nSteps);
dlogS = (mu-0.5*sigma^2)*dt + sigma*sqrt(dt)*eps;
logS = zeros(nPaths,nSteps);
for t=1:nSteps
    if (t==1)
        lastLogS = logS0;
    else
        lastLogS = logS(:,t-1);
    end
    logS(:,t) = lastLogS + dlogS(:,t);
end
S = exp(logS);
times = dt:dt:T;
end
```

Note that we are using some new MATLAB syntax in this code. We've already seen that `A(1:end,j)` means the j-th column of x. We are using the fact that you can abbreviate this to just $A(:,j)$.

In fact it is possible to further vectorize the code and eliminate the loop over time steps. This is only possible because MATLAB has a built in function `cumsum` which allows you to compute the cumulative effect of adding a number of increments. `cumsum` stands for cumulative sum.

If we suppose that we have a vector consisting of a number of row vectors x_1, x_2, x_3 and so on, the `cumsum` allows us to compute the cumulative sum as we add together rows.

$$\texttt{cumsum}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \dots \\ x_n \end{pmatrix} = \begin{pmatrix} x_1 \\ x_1 + x_2 \\ x_1 + x_2 + x_3 \\ \dots \\ x_1 + x_2 + x_3 + \dots + x_n \end{pmatrix}$$

Although `cumsum` by default computes the cumulative sum of the rows, it has an optional second parameter which can be used to ask it to sum the columns instead. Here is a summary of how `cumsum` will respond to different kinds of parameter:

- `cumsum(v)` computes the cumulative sums of a vector `v`.
- `cumsum(A)` computes the cumulative sums of the rows of a matrix `A`.
- `cumsum(A,1)` computes the cumulative sums of the rows of a matrix `A`. Rows are the first dimension.
- `cumsum(A,2)` computes the cumulative sums of the rows of a matrix `A`. Columns are the second dimension.

This final option is just what we need to eliminate all the loops from our code. Here is the final version of our stock price simulator.

```
% Generate random price paths according to the black scholes model
% from time 0 to time T. There should be nSteps in the path and
% nPaths different paths
function [ S, times ] = generateBSPaths( ...
    T, S0, mu, sigma,nPaths, nSteps  )

dt = T/nSteps;
logS0 = log( S0);
eps = randn( nPaths, nSteps );
dlogS = (mu-0.5*sigma^2)*dt + sigma*sqrt(dt)*eps;
logS = logS0 + cumsum( dlogS, 2);
S = exp(logS);
times = dt:dt:T;

end
```

You can think of `cumsum` as meaning "integrate" in the code above. You will notice that this code is quite a bit shorter than the first version. It also runs quite a bit faster. Against that you may find it a little harder to understand.

A nice application of our simulation code is simply to draw a plot of a simulated stock price (Figure 5.1.2. Let us go slightly further and create a so-called *fan diagram* which shows the 5th, 50th and 95th percentiles of the stock price at each moment in time together with a single example stock price.

To generate this plot we use the `plot` function together with the `prctile` function. We leave it as an exercise for you to generate this plot yourself.

5.2 Monte Carlo Pricing

Algorithm (Monte Carlo Pricing)**.** *To compute the Black–Scholes price of an option whose payoff is given in terms of the prices at times* t_1, t_2, ..., t_n

(i) *Simulate stock price paths* in the risk neutral measure. *i.e. use the algorithm above with* $\mu = r$.

(ii) *Compute the payoff for each price path*

(iii) *Compute the discounted mean value*

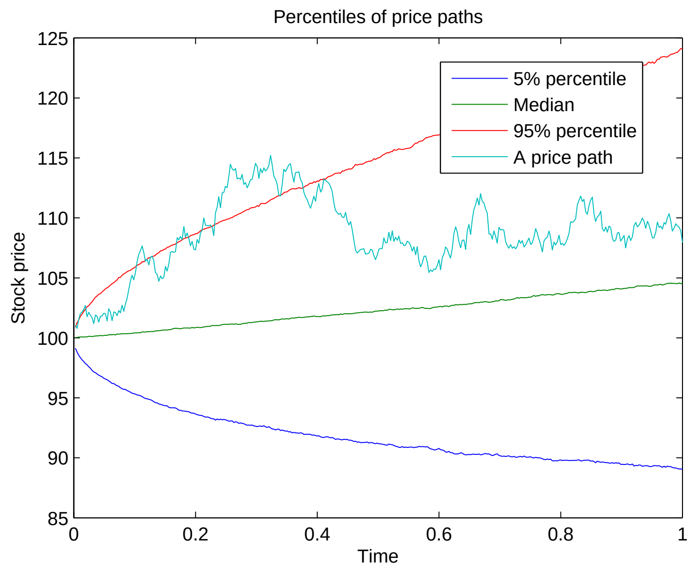

Figure 5.1: A fan diagram of a stock price over time for a stock obeying geometric Brownian motion

(iv) *This gives an unbiased estimate of the true risk neutral price*

We can use this algorithm to price path independent options. Here are some examples of path independent options.

Example 1: A discrete up and out call option with strike K and barrier B and maturity T is an option that pays 0 if the stock price is ever above B at the end of the business day. If it reaches time T without hitting the barrier, its payoff is given by $\max(S_T - K, 0)$ The payoff is determined entirely by the prices at the end of each business day.

Example 2: An Asian call option with maturity T has its payoff determined by the average price $\overline{S}$ at the close of the last n days of trading up to and including maturity. The payoff is given by $\max(\overline{S} - K, 0)$

You should familiarize yourself with the names of various kinds of exotic derivative which you can price by Monte Carlo methods.

- An option that becomes worthless if the price goes below a barrier is called a down and out option.
- Up-and-out and down-and-out options are termed “knockout options”.
- A knock in option is one where the option is worthless unless the stock price crosses a barrier

- One has "up-and-in" and "down-and-in" options.
- You can create knockout puts, digital Asians etc..
- One can also mathematically model continuous barrier and Asian options. We can approximate these by using a large number of time points.

We note that European put and call options can be priced by this Monte Carlo method too, and then the method is equivalent to one of the Monte Carlo integration techniques covered last week (exercise: convince yourself). This means that there are more efficient ways of computing the price in this case than using Monte Carlo.

Monte Carlo pricing is always a numerical integration technique (we are computing expectations and expectations are defined as integrals). If we use n points on the price path then we are computing an n-dimensional integral. This means that Monte Carlo integration will be better than variants on the rectangle rule etc. if n is greater than roughly 3 or 4. In practice this means that we will use Monte Carlo methods for path-dependent options.

Note that you *cannot* use our algorithm to price American options. This is because the payoff of an American option is not specified in the contract as a function of the price at fixed times. There is a more sophisticated technique called American Monte Carlo, but we will not discuss that in this course.

We should be aware that if we use Monte Carlo pricing, we are only estimating the price. How accurate is our answer? By the central limit theorem, we expect that the sample mean of the discounted payoff is approximately normally distributed with standard deviation

$$\frac{\tilde{\sigma}_P}{\sqrt{N}}$$

where $\tilde{\sigma}_P$ is the population standard deviation of the discounted payoff. If we use the sample standard deviation $\tilde{\sigma}_S$ as an estimate for the population standard deviation, we can estimate that the standard error of our price is

$$\frac{\tilde{\sigma}_S}{\sqrt{N}}.$$

We can then use this to compute approximate confidence intervals. (The sample standard deviation gives a slightly biased estimator, but this bias can be ignored for large N).

5.2.1 MATLAB implementation

As a concrete example, we wish to price a discrete up-and-out call option with barrier B, strike K and maturity T where one tests to see if the option has hit the barrier at *nSteps* evenly spaced times over the lifetime of the option.

The stock price follows the Black Scholes model with parameters S_0, μ, r, σ as usual. We wish to write a function to price the option using Monte Carlo simulations with *nPaths* paths. The function should also return an estimate of the error.

We already have a function that computes a matrix of prices with rows corresponding to scenarios and columns corresponding to times. We would like to write a function that computes the payoff of our option as a vector given the matrix of prices. We will then be able to price the option by taking the mean of this vector and discounting.

Here is a first attempt at this code, but notice that it contains two loops.

```
% A very inefficient version of computeKnockoutPayoff
% which uses 2 loops
function [ payoff ] = computeKnockoutPayoff2Loops(...
    strike, barrier, priceHistory )

nPaths = size( priceHistory, 1 );
nSteps = size( priceHistory, 2 );
payoff = zeros( nPaths, 1 );
for p=1:nPaths;
    knockedOut = 0;
    for t=1:nSteps
        if priceHistory(p,t)>barrier
            knockedOut = 1;
        end
    end
    if (~knockedOut)
        finalPrice = priceHistory( p, nSteps );
        if (finalPrice>strike)
            payoff(p)=finalPrice-strike;
        end
    end
end
end
```

You might notice that we don't need to pass the number of paths and the number of steps to this function, we've used the `size` function to deduce that from the matrix `priceHistory`.

Since this code is completely repetitive across scenarios we know that we can vectorize it to improve efficiency. We'll do this shortly, but it is more important to have code that works than code that is fast. So let us begin by testing our code.

```
function testComputeKnockoutPayoff2Loops()

stockPrices = [100,101,102; 100,120,107; 100,103,108 ];
payoffs = computeKnockoutPayoff2Loops(105,110,stockPrices);
assertApproxEqual( payoffs(1), 0, 0.001);
assertApproxEqual( payoffs(2), 0, 0.001);
assertApproxEqual( payoffs(3), 3, 0.001);

end
```

We can now combine our code with our stock price simulator to price a knockout option by Monte Carlo.

```
function [price, errorEstimate]=priceKnockoutByMonteCarlo(...
    strike, barrier, T,...
    S0, r, sigma, ...
    nPaths, nSteps )

% Generate paths in risk neutral measure (mu=r)
priceHistory = generateBSPaths(T,S0,r,sigma,nPaths,nSteps);
payoffs = computeKnockoutPayoff(strike,barrier,priceHistory);
discountedPayoff = exp(-r*T)*payoffs;
price = mean( discountedPayoff );
errorEstimate = std( discountedPayoff )/sqrt(nPaths);

end
```

The main pricing function is remarkably simple. It does little more than explain in English what the Monte Carlo pricing algorithm actually is. The real work is done in `generateBSPaths` and `computeKnockoutPayoff`. This is an example of well-written code. We have divided our code into small pieces each of which is reasonably easy to understand and easy to test.

Since we don't have an exact formula for the price of discrete time knock out option, it is hard to test our pricing code directly. However the code that computes the *payoff* of a knock-out option is very easy to test.

If we assume that the barrier is infinite, a knock-out option becomes equivalent to a vanilla option. This means it is easy to test our pricing code in this extreme case. On the other hand we have tested our payoff code with a variety of barriers. So when we put everything together we can be pretty confident of our option pricing code even when the barrier is not infinite. This again shows the value of writing lots of small functions each with individual tests.

5.2.2 Vectorizing our code

Our code works so we could stop here. However, it is useful practice in writing MATLAB to see how we can eliminate unnecessary for loops. Our code treats all scenarios identically and independently. This means that we know the code can be vectorized over the scenarios relatively easily. We also know how to use vectorization to get rid of unwanted `if` statements. Here is the result of a first attempt at vectorizing our code.

```
% A slightly inefficient version of computeKnockoutPayoff
% which uses a loop
```

```
function [ payoff ] = computeKnockoutPayoff1Loop( ...
    strike, barrier, priceHistory )

nPaths = size( priceHistory, 1 );
nSteps = size( priceHistory, 2 );

knockedOut = zeros( nPaths, 1);
for t=1:nSteps
    knockedOutThisTime = (priceHistory(:,t) > barrier);
    knockedOut = knockedOut | knockedOutThisTime;
end
finalPrice = priceHistory( :, nSteps );
inMoney = finalPrice>strike;
payoff=(~knockedOut).*inMoney.*(finalPrice-strike);
end
```

The variable `knockedOutThisTime` tells us whether the option has knocked out at a particular time step. On the next line we use `|`, which means element-by-element 'OR', to compute whether the option has knocked out at any time up to the current time step.

Also in the last line of the code we have used the vectorization trick of using the fact that `knockedOut` and `inMoney` take the values `1` and `0` since `1` represents `true` and `0` represents false.

This means that the quantity

```
(~knockedOut).*inMoney.*(finalPrice-strike)
```

will only be non zero when we haven't knocked out and aren't in the money.

We can actually get rid of the remaining for loop using the `max` function. We are currently looping to find out if the price was ever above the barrier. Suppose we compute `priceHistory > barrier`. This will consist of 1's when the price is above the barrier and 0's when the price is below the barrier. This means that the maximum of `priceHistory>barrier` in each row will be 1 if the barrier knocked out and 0 otherwise.

We now use MATLABs documentation to learn that you can use `max(-A,[],2)` to compute a vector of the maximum across the columns. `max(A,[],1)` computes the maximum across the rows.

This means that the following code will compute the payoff of a knockout option and it manages to do so without any loops.

```
%COMPUTEKNOCKOUTPAYOFF
% Computes the payoff of a knockout
% option given the priceHistory. priceHistory
% should have rows corresponding to scenarios
% and columns corresponding to times
function [ payoff ] = computeKnockoutPayoff( ...
    strike, barrier, priceHistory )

knockedOut = max( priceHistory>barrier, [], 2);
notKnockedOut = 1-knockedOut;
finalPrice = priceHistory(:,end);
inMoney = finalPrice>strike;
payoff = inMoney .* notKnockedOut .* (finalPrice-strike);

end
```

We have now seen how to price the code using three short functions `computeKnockoutPayoff`, `priceKnockoutCallOption` and `generateBSPaths`. We have three equivalent versions of `generateBSPaths` and `computeKnockoutPayoff` each using different amounts of vectorization. I recommend that you focus on understanding the final versions of the code. The other versions were introduced to help you understand the final version.

When you write your own code, should you vectorize it? The answer to this is that it is up to you.

The good things about vectorization are: Vectorization makes the code more readable *once you know the standard tricks*; Vectorization may make the code faster.

But it isn't all good. Vectorization makes the code less readable if you don't know the standard tricks. Vectorization may make the code take longer for you to write (until you've mastered it)

I recommend writing what you find comes most naturally and optimizing only if there is a problem (or the question tells you to)

5.2.3 Testing Monte Carlo code

We discussed that our pricing code is now fairly easy to test, but we haven't explicitly described how to write the tests.

Algorithms that use random numbers will sometimes give poor answers by chance. If you're not careful this means that your tests will sometimes fail by chance. This is a serious problem if you have millions of tests to run to check all of your bank's code. It is therefore essential that all tests reliably pass or fail, even if they use random numbers.

To make your tests reliable you should *seed the random number generator*. To explain the terminology, you need to undertand that the pseudo random number generator used by the computer has a state which determines what they will do next. Each time you generate a number the generator changes state. This ensures they will generate a different number the next time. Manually fixing the state is called *seeding* the random number generator. This is because the current state is the "seed" out of which all subsequent random numbers "grow".

In MATLAB `rng('default')` sets the random number generator back to its default state. Therefore you should start unit tests of functions that use the random generator with a call of `rng('default')`.

5.3 Monte Carlo Greeks

A trader will not thank you if you can compute the price but not the Greeks. This is because the "price" is in fact the risk-neutral price and so is heavily associated with the delta hedging trading strategy. You can only guarantee to break even if you delta hedge, and that requires computing the delta. In general, a price is useless without a trading strategy to achieve the price.

5.3.1 Numerical differentiation

Before discussing how to compute the delta, we need to discuss how to numerically compute partial derivatives of functions in general.

Let f be a smooth function. We can approximate the derivative at x as

$$f'(x) = \frac{f(x+h) - f(x)}{h}$$

for small h. This is called the *forward estimate*. It has an error of $O(h)$. There is also the *backward estimate*

$$f'(x) = \frac{f(x) - f(x-h)}{h}$$

which also has an error of $O(h)$. You can use Taylor's theorem to prove these error estimates. But a better approximation is

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h}$$

In the latter case, the first order error terms in the Taylor expansion for the error cancel. This means that we can use Taylor's theorem to prove that the error is bounded by

$$\sup_{c \in [x-h, x+h]} \frac{|f^{(3)}(c)|}{6} h^2.$$

The proof is left as an exercise.

What value should one choose for h? If you want the best computation possible, we want to choose h that is small enough to give a reasonably accurate value value but on the other hand we want to choose h so it isn't so small that rounding errors dominate the calculation. Choosing $h > \sqrt{\epsilon} x$ where ϵ denotes the accuracy of the computer will ensure that h isn't too small. For our purposes $\epsilon = 2.2 \times 10^{-16}$. To be more precise, one should really worries about whether $h + x$ can be represented accurately on the computer and should modify the value accordingly. However, since we will apply this to a Monte Carlo method, we needn't worry about choosing the optimal h. What is important however, is not to choose too small an h because we will then hit problems.

One can compute higher order derivatives numerically too.

$$\begin{aligned} f'(x-h) &\approx \frac{f(x)-f(x-h)}{h} \\ f'(x) &\approx \frac{f(x+h)-f(x)}{h} \\ f''(x) &\approx f''(x-h) \approx \frac{f'(x)-f'(x-h)}{h} \\ &\approx \frac{f(x+h)-2f(x)+f(x-h)}{h^2} \end{aligned}$$

This gives a formula that can be used to numerically approximate second derivatives. Somewhat more rigorously, one can remark that this formula for f'' is accurate for quadratics and so, using Taylor's theorem one can give bounds on the error of the approximation.

As with approximating first derivatives it is important not to use too small an h when applying this formula.

5.3.2 Monte Carlo Greeks

So how should we compute the Delta by Monte Carlo? Your first thought might be the following (and if so it is incorrect!)

- Compute the Monte Carlo price with an initial stock price of S_0
- Compute the Monte Carlo price with an initial stock price of $S_0 + h$
- Take the difference and divide by h

This is because the Monte Carlo prices are randomly generated. The random error will overwhelm the systematic difference we are trying to measure.

Here is an algorithm that does not suffer from the problem.

Algorithm (Monte Carlo Delta).

- *Compute the Monte Carlo price with an initial stock price of* $S_0 - h$
- *Compute the Monte Carlo price with an initial stock price of* $S_0 + h$ using exactly the same random numbers in the simulation
- *Take the difference and divide by* $2h$

To see mathematically why this algorithm will work, recall that Monte Carlo pricing is really just numerical integration. Under reasonable conditions, the partial derivative of an integral is the integral of the partial derivative. Our Monte Carlo Delta algorithm is equivalent to computing the derivative of the pricing kernel numerically and then integrating by Monte Carlo integration.

Here is the MATLAB code to compute the delta by Monte Carlo.

```
function [ delta ] = computeDeltaByMonteCarlo( ...
    strike, barrier, T,...
    S0, r, sigma, ...
    nPaths, nSteps )

h = 10^(-6)*S0; % Won't cause rounding problems
                % but a minute change financially

rng('default');
p1 = priceKnockoutByMonteCarlo(strike,barrier,T,...
    S0-h, r, sigma, ...
    nPaths, nSteps );
rng('default');
p2 = priceKnockoutByMonteCarlo(strike,barrier,T,...
    S0+h, r, sigma, ...
    nPaths, nSteps );
delta = (p2-p1)/(2*h);

end
```

To keep our code simple, we have seeded the random number generator to ensure that the same random numbers are used at $S_0 - h$ and at $S_0 + h$.

It would be better in practice to avoid generating the same random numbers twice as this inevitably be faster and would get rid of the bias of always using the same random numbers in calculations. Implementing this is left as an exercise.

Finally we note that our Monte Carlo delta algorithm can easily be generalized to other Greeks.

5.4 Antithetic Sampling

This chapter is simply an introduction to Monte Carlo pricing. There are entire books written on the subject. We will also return to Monte Carlo pricing later in the course and see some ways in which it can be improved.

As a taster, we will describe the technique of antithetic sampling which can often be used to improve Monte Carlo estimates with very little effort.

The idea of antithetic sampling is to simulate N stock prices using the algorithm above and using a matrix of normally distributed random numbers ϵ. One then computes another simulation of stock prices but this time using the random numbers $-\epsilon$. This gives us two far from independent stock price simulations. The surprising idea is that taking the average price obtained in this way will often a better estimate than generating $2N$ independent samples.

To see why, let P_1 and P_2 be random variables with $\mathrm{E}(P_1) = \mathrm{E}(P_2)$. We wish to calculate this expectation.

Define
$$Z = \frac{P_1 + P_2}{2}.$$
Then
$$\begin{aligned} \mathrm{E}(Z) &= \mathrm{E}(P_1) = \mathrm{E}(P_2) \\ \mathrm{Var}(Z) &= \frac{\mathrm{Var}(P_1) + \mathrm{Var}(P_2) + 2\,\mathrm{Cov}(P_1, P_2)}{4} \end{aligned}$$
So, all other things being equal, if P_1 and P_2 are negatively correlated, the estimate $\mathrm{E}(Z)$ will be improved over the case when P_1 and P_2 are independent.

Suppose that in our Monte Carlo pricing algorithm, we generate a random normally distributed vector ϵ and use this to compute a payoff $P_1 = P(S(\epsilon))$. We can use the vector $-\epsilon$ to compute another payoff $P_2 = P(S(-\epsilon))$. Here S is the stock price given ϵ, P then computes the payoff.

$\mathrm{Cov}(P_1, P_2)$ will be negative for many payoff functions (if ϵ leads to a good payoff, $-\epsilon$ will often be bad). So the estimate $\mathrm{E}(Z)$ may be a better estimate than we would have obtained by taking independent samples P_1 as claimed.

Implementing antithetic sampling is left as an exercise.

5.5 Further Reading

For a review of the basic Mathematical Finance used in this chapter see the Appendix of [1].

A good book that includes a lot on Monte Carlo methods is [2].

Bibliography

[1] John Armstrong. *C++ for Financial Mathematics.* CRC Press, 2017.

[2] M. S. Joshi. *The concepts and practice of mathematical finance*, volume 1. Cambridge University Press, 2003.