How to build a Black Scholes VBA Option Pricer using Monte Carlo Simulation

Introduction

If a company wishes to hedge a foreign currency cashflow at a future date, it can use FX forward contracts, futures, swaps or FX options. FX options provide flexibility as they give the right, but not the obligation for the company to exercise the contract.

This means that the company can benefit from favourable currency movements. For example, if a British company is expecting to receive US dollars for an acquisition in 30 days' time, it can buy a put option on US dollars to hedge the risk from an increase in the value of the US dollar vs sterling (an increase in the USD/GBP exchange rate). A European put option would give the right to sell GBP at a lower spot exchange rate K, if the USD/GBP spot exchange rate is below K at the time of expiry of the option. K would be the strike price of the option.

The payoff from the transaction would be max (S – K, 0) * number of put options bought, where S is the USD/GBP spot exchange rate.

The geometric Brownian motion model of asset prices can be expanded to cater for spot exchange rates, where the spot exchange rate S is defined as the units of foreign currency for each unit of domestic currency.

The stochastic partial differential equation can be solved using Monte Carlo simulation, as well as analytically, or via lattice methods, finite differences, or via Fourier Transforms. In this document, we briefly look at a Monte Carlo approach to solving the option price.

Monte Carlo Simulation

Monte Carlo simulation is a numerical method used in finance to solve partial differential equations such as the Black Scholes equation for pricing stock options.

The basic tenet of Monte Carlo simulation is that there is a stochastic variable in an equation that can be sampled many times over. When the result of the many simulations of a function containing the random variable is averaged, it approximates to the real mean. This is a result from the Central Limit Theorem of probability.

The variable being sampled in Monte Carlo simulation usually comes from the uniform distribution as simulated by a random number generator. In finance the most common transforms are to the normal distribution for use in lognormal price paths such as those for Black Scholes. However, the random variable can be transformed to Beta, Gamma, Poisson distributions depending on the pattern exhibited by the data that is being modelled.

In the classic Black-Scholes context, by using Monte Carlo simulation we’re attempting to model the price path of an asset such as a stock, then calculating the payoff and price of an equity derivative.
Black Scholes FX Option Pricer Model

According to the Black Scholes model, the FX spot rate is defined by the following stochastic partial differential equation:

\[ dS = (r - rf - 1/2\sigma^2)dt + \sigma dz \]

where \( dz \) is a standard Brownian motion, defined by \( dz = \epsilon \times \sqrt{dt} \)

where \( \epsilon \) is a standard normal random variable; \( dS \) is the change in spot rate, \( r \) is the domestic risk-free interest rate, \( rf \) is the risk-free interest rate of the foreign security, \( \sigma \) the volatility of the stock.

As such the price at time \( 0 < t <= T \) is given by

\[ St = S_0 \times \exp((r - rf - \frac{1}{2} \sigma^2) dt + \sigma \times \epsilon \times \sqrt{dt}) \]

The price of a European call option is given by \( \max(St - K, 0) \) where \( St \) is the final stock price at expiry.

The Monte Carlo simulation of the stock option pricing equation therefore simulates the expected price of the call option, \( E[\max(St - K, 0)] \), or \( \max(S_0 \times \exp((r - \frac{1}{2} \sigma^2) dt + \sigma \times \epsilon \times \sqrt{dt}), 0] \)

The next step is to apply the time value of money by discounting the expected option price by \( \exp(-rt) \). By put-call parity, the price of the European put option is given by \( p = c + K \times \exp(-rt) - S_0 \), where \( p \) is the price of the put option, \( c \) is the price of the call option.

The ‘proper’ way to define the function would be

Coding the MC simulation in Excel VBA

The first coding issue is, do we use a subroutine or a function? Typically a function is used for numerical calculations. This is especially useful in a Monte Carlo simulation context as using subroutines induces complications with the persistence of variables after the simulation has been run, especially if the variables are declared as static or public. This would need to be done for compartmentalisation of code if using a subroutine as one would need to code the setting of input ranges, as well as output ranges. If using a subroutine, it quickly becomes complicated and necessitates the use of classes. Much easier to use functions.

The function therefore takes in 6 parameters, \( S, K, r, rf, t, \sigma, N \) (number of simulations) and should return the call price and the put price. To return the two values, we need to create a function that returns an array.

The next issue with Monte Carlo simulations is performance. Excel VBA has late and early binding depending on the data type declarations. Late binding slows down VBA code execution noticeably over many different simulation cycles. As such, it is to be avoided. This is done by not defining the variables of the function as variant, and by explicitly defining each and every variable’s data type, including the array returned by the function.

So where a ‘lazy’ function declaration may look like

Function Black_Scholes (S, K, r, rf, t, sigma, N)
How to build a Black Scholes VBA Option Pricer using Monte Carlo Simulation

Function Black_Scholes (S as double, K as double, r as double, rf as double, t as double, sigma as double, N as long) as double

Depending on how we wish to use the function in future, we may also want to define the parameters as being passed in by value. This would protect encapsulation of the function if calling it from a subroutine – an important step when moving towards an object-orientated approach.

Function Black_Scholes (ByVal S as double, ByVal K as double, ByVal r as double, ByVal rf as double, ByVal t as double, ByVal sigma as double) as double

We use a simple for....next structure in the code to loop through the simulations. So the code structure is as follows:

[Function declaration]
[Internal variables declaration e.g. loop counter, call and option prices]
[For...<counter> =1 to N]
[Calculate stock price path]
[Calculate call option payoff]
[Sum the call option payoffs – essential for averaging these later]
[Next <counter>]
[Calculate the average call option payoff]
[Discount the call option payoff to give the call option price]
[Use the put-call parity equation to calculate put price]
[Write the call and put option prices into an array]
[Return the array]

Generating Random Numbers in Excel VBA

The random number generator is given by the Excel VBA Rnd function, which generates random numbers between 0 and 1. These random numbers are supposed to be uniformly distributed across the interval of 0 and 1. The degree of uniformity impacts the efficiency of the Monte Carlo simulation as it impacts the rate of convergence of the simulation-derived price to the actual value. The problem with Excel’s random generator is that it produces clustered numbers, which greatly diminishes the rate of convergence to the actual mean of the sampled population (in this case the option price). There are a number of ways to improve the rate of convergence, and these will be covered in a separate document. Suffice it to say for now that the Rnd function has flaws, and for an entry level and simplistic analysis, will be adequate provided there is a large number of runs.

Black Scholes VBA FX Monte Carlo Simulation

Function MC_Sim_Black_Scholes(ByVal S As Double, ByVal K As Double, ByVal r As Double, ByVal rf As Double, ByVal sigma As Double, ByVal t As Double, ByVal N As Double) As Variant

Dim i As Long
Dim call_price As Double
Dim put_price As Double
Dim put_price_sum As Double
Dim S_t As Double
How to build a Black Scholes VBA Option Pricer using Monte Carlo Simulation

```vba
For i = 1 To N
    S_t = S * Exp((r - rf - 0.5 * sigma ^ 2) * t + sigma * WorksheetFunction.NormSInv(Rnd) * Sqr(t))
    call_price = WorksheetFunction.Max(S_t - K, 0)
    put_price = WorksheetFunction.Max(K - S_t, 0)
    call_price_sum = call_price_sum + call_price
    put_price_sum = put_price_sum + put_price
Next i

    call_price = call_price_sum / N
    'discounting
    call_price = call_price * Exp(r * t)
    put_price = put_price_sum / N
    'discounting
    put_price = put_price * Exp(r * t)

MC_Sim_Black_Scholes = Array(call_price, put_price)

End Function
```

Next Steps

There are various improvements and additional features that can be made to the model above:

- Improved convergence; this can be achieved through antithetic variates, pseudo-random number generators, low discrepancy number generators, control variates, stratified sampling, importance sampling, moment matching, Latin Hypercube sampling
- Greeks – calculation of delta, gamma, rho, vega, theta
- Movement away from the classical Black Scholes assumptions re, for example, volatility and modelling this variable as a Levy or Jump diffusion process, also dynamic interest rate models for both foreign and domestic interest rates

How VBA Developer.net Can Save You Time and money

You can get complete Excel apps from VBA Developer.net containing the code in this document, customisation, VBA development of any Excel, Access and Outlook apps, as well as C# and C++ add-ins and technical documentation.

Visit VBA Developer.net
How to build a Black Scholes VBA Option Pricer using Monte Carlo Simulation

How to build an Equity Option Pricer using the Binomial Tree in Excel VBA

How to Use Cursors in SQL for a Portfolio Reporting VBA Tool

How to Move Data from Access to Excel with SQL for a Portfolio Reporting VBA App

Portfolio Reporting VBA Tool: Inserting Data into SQL/Access Databases from Excel

Portfolio Reporting VBA Tool: Connecting Excel with SQL & Access Databases

How To Design Classes For an Option Pricer in VBA: UML Concepts

How To Design Classes for Object Orientated VBA Programming