APPLIED NUMERICAL FINANCE REPORT - 20247

A CROSS-LANGUAGE PROGRAM FOR PRICING REAL MARKET DATA

1 INTRODUCTION

The aim of this work is to understand how efficiently financial markets price financial derivatives. In particular, in this research, we will focus on the pricing of European options and then we will compare the real-time price of a specific option to the prices we will get by applying two different mathematical models, using both their closed form solutions and their Monte Carlo estimates: the standard Black-Scholes model and the jump-diffusion model.

The second aim of this work will be to compare two different programming languages (VBA and C++) and to understand which one is more powerful in terms of accuracy and in terms of time required for the computations.

1.1 GATHERING DATA FROM THE WEB

The first step in our research will be to gather data about options prices and stock prices. This can be done by choosing a specific company and writing its ticker in the designated cell in the excel file. Then, by clicking on the specific buttons (through the "*yahoo_options*" and the "*google_historic*" subroutines), the latest available prices will be downloaded from the *Yahoo Finance* site (for options data) and from *Google Finance* (for stock data). The last five years of daily stock prices will be downloaded and then the daily continuous returns are calculated automatically; consequently, also daily volatility and the correspondent annualized volatility will be computed. Moreover, another required input, the risk-free rate will be downloaded (through the "*risk_free_rate*" subroutine) from the *Market Watch* site; it is actually referred to the U.S. 10-Year Treasury Note.

For sake of simplicity, we will consider only options with a specific maturity date, in our case 21/09/2018, choosing the right option in the dropdown menu in the "*Options*" sheet.

In the case we are going to study, I considered the Microsoft stock and its call option with strike price equal to $70. At the moment I am writing (20/12/2017), the last available stock price is $85.83 and its call option is trading at $18.44, while the risk-free is 2.47%.

1.2 BLACK-SCHOLES MODEL

Now we have all our parameters to insert in the standard Black-Scholes model, which are shown below:

Parameters		
Option Type	c	
Strike price of the option (K)	70.00	
Stock price at time 0 (S_0)	85.83	
Volatility (sigma)	0.223794	
Time to maturity (in years, T)	0.750685	
Annual risk-free interest rate (r)	2.47%	

The analytical formulas of this model (coded in the "*BSOption*" excel function) for the price of a call option and a put option are:

$$c = S_0 N(d_1) - Ke^{-rT} N(d_2) \qquad d_1 = \frac{\ln(S_0/K) + (r + \sigma^2/2)T}{\sigma\sqrt{T}}$$

$$p = Ke^{-rT} N(-d_2) - S_0 N(-d_1) \qquad d_2 = \frac{\ln(S_0/K) + (r - \sigma^2/2)T}{\sigma\sqrt{T}} = d_1 - \sigma\sqrt{T}$$

1.3 MERTON JUMP-DIFFUSION MODEL

Observed asset prices do not move continuously in time. A reasonable model is therefore to let prices make discrete jumps from time to time. Since jumps can have important effects on the option value, we will study the Merton jump-diffusion model.

This model superimposes a jump component on a diffusion component. The diffusion component is the familiar geometric Brownian motion. The jump component is composed of lognormal jumps driven by a Poisson process.

The model assumes that the underlying asset price follows the jump-diffusion process:

$$\frac{dS_t}{S_t} = (r - \lambda m)\, dt + \sigma dW_t + dJ_t \qquad (1)$$

where *J(t*) is a compound Poisson process and *m* is the expected value of the jump sizes, which are lognormally distributed with mean *a (gamma)* and standard deviation *b (delta)*. The jump-diffusion model requires three additional parameters which we will fix for sake of simplicity; we will also choose small values in order to make the result comparable to the Black-Scholes model. Therefore, we will have the following parameters:

"Jump intensity" Lambda	**1.0**	
"Mean of jump size" Gamma (a)	**0.01**	
"Std. Dev. of jump size" Delta (b)	**0.01**	

The semi-closed form solution (coded in the "*JumpDiffusionMerton*" subroutine), derived in 1976 by Robert Merton, for the price of a call option is:

$$c_0 = \sum_{n\in\mathbb{N}} e^{-\lambda' T} \frac{(\lambda' T)^n}{n!} bsc(S_0, r_n, \sigma_n) \qquad r_n = r - m\lambda + \frac{n\log(1+m)}{T},\ \lambda' = \lambda(1+m) \text{ and } \sigma_n = \sqrt{\frac{\sigma^2 T + nb^2}{T}}$$

1.4 MONTE CARLO METHOD

Monte Carlo simulation is a numerical method that is useful in many situations when no closed-form solution is available. The Monte Carlo method can be used to simulate a wide range of stochastic processes and is thus very general. To illustrate the use of Monte Carlo simulation, we will start with the processes where the natural logarithm of the underlying asset follows geometric Brownian motion. The process governing the asset price S is then given by:

$$S + dS = S\exp\left[\left(\mu - \frac{1}{2}\sigma^2\right)dt + \sigma dz\right]$$

where *dz* is a Wiener process with standard deviation 1 and mean 0. To simulate the process, we consider its values at given discrete time intervals, *Δt* apart:

$$S + \Delta S = S \exp\left[\left(\mu - \frac{1}{2}\sigma^2\right)\Delta t + \sigma\epsilon_t\sqrt{\Delta t}\right]$$

where *ΔS* is the change in *S* in the chosen time interval *Δt*, and ϵ_t is a random drawing from a standard normal distribution. The main drawback of Monte Carlo simulation is that it is computer-intensive. A minimum of 10,000 simulations are typically necessary to price an option with satisfactory accuracy. The standard error in the estimated value from the standard Monte Carlo simulation is normally related to the square root of the number of simulations. More precisely, if *s* is the standard deviation of the payoffs from *n* simulations, then the standard error is given by $\frac{s}{\sqrt{n}}$.

This means that to double the accuracy, we will need to quadruple the number of simulations. So if we want to double the accuracy from 10,000 simulations, we will need 40,000 simulations, and so on. For path-dependent options, we also need to divide each path into discrete time steps. In order to minimize discretization error and in order to reach an even more accurate estimate, we will choose a large number of time steps, fixing them to 300 for sake of simplicity (given also that our time to maturity is 0.750685 years).

In the case of the jump-diffusion model, the stock dynamics follow the process below, solution to equation (1):

$$S(T) = S(0)e^{\left(r - m\lambda - \frac{\sigma^2}{2}\right)T + \sigma W^Q(T)}\left(\prod_{j=1}^{N(T)} Y_j\right)$$

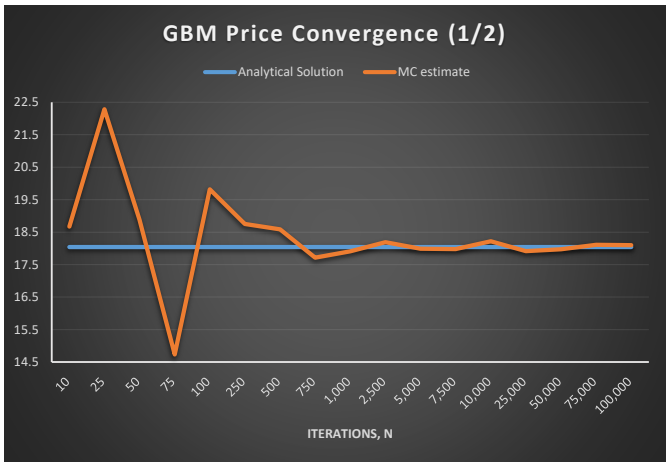
The steps in a standard Monte Carlo simulation is simply to simulate *n* number of paths of the asset price. The value is then given as the discounted average of the simulated paths. In the case of non-path-dependent European style options, we are only interested in the end value of the asset, and the value of a standard call option (a similar approach is used for the put) is simply given by:

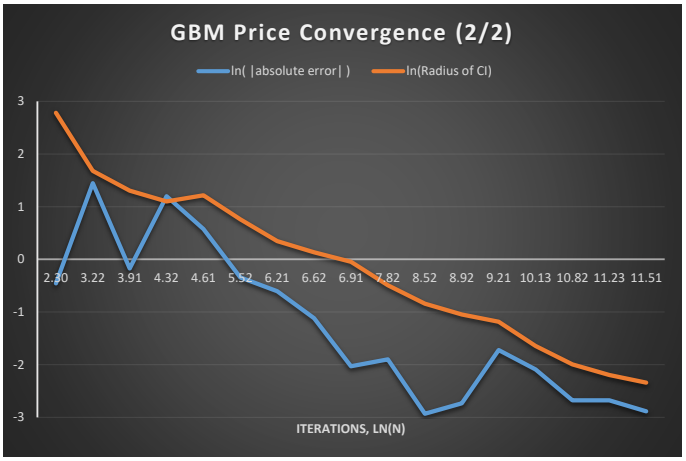
$$c_t = e^{-r(T-t)}E[\max(0, S_T - X)]$$

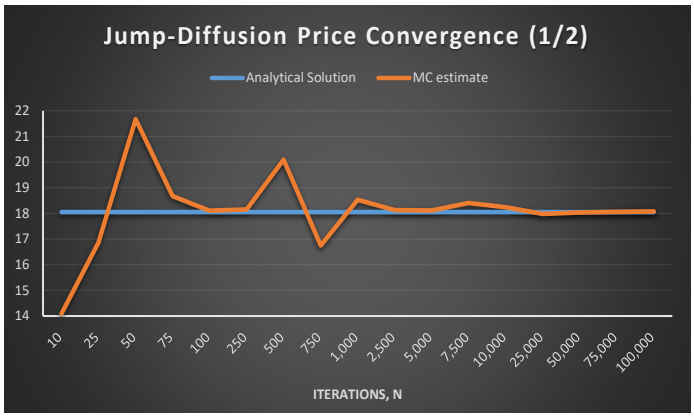
$$price_{mean} = \frac{1}{N}\sum_{i=1}^{N} c_i(S,T)$$

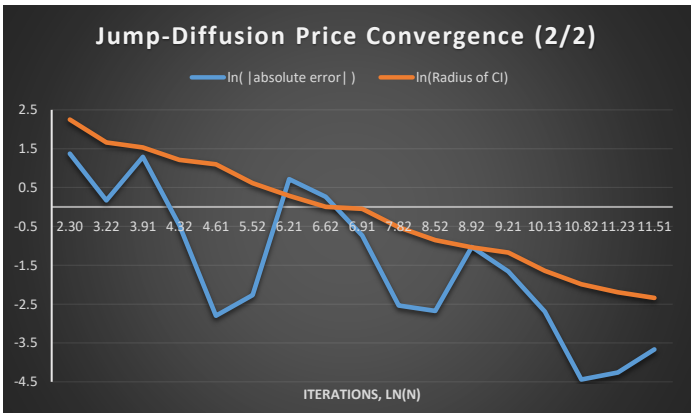
Since Monte Carlo results are estimates, it is important to focus on the relative accuracy of these estimates by computing the 95% confidence intervals and their radius. The higher the number of paths the more accurate the estimate will be.

The convergence of the estimate towards the true price (given by the analytical formula) for both the standard geometric Brownian motion model and the jump-diffusion framework can be observed through the following samples of Monte Carlo simulations, carried out through the C++ applications ("*StdMonteCarlo.exe"* and "*JDMonteCarlo.exe"*) by increasing the number of paths up to 100.000 iterations:









1.5 ANALYSIS OF THE RESULTS

In the previous graphs, we can observe that the accuracy of the Monte Carlo estimate increases as the number of iterations increases; the estimate gets closer and closer to the true price (especially after 5.000 iterations). Moreover, due to the Central Limit Theorem, we also observe that, considering a logarithmic scale, the absolute error and the radius of the confidence intervals shows a rate of convergence very close to ½ (not considering discretization error, the orange line representing the radius of the confidence intervals should be a straight line of slope -½). All the closed form solutions and the Monte Carlo estimates (for a large *n*) are indicating that the real world market price is slightly overvaluating the call option. This is obviously due to the fact that not all the assumptions of the underlying models are respected.

2 EFFICIENCY OF PROGRAMMING LANGUAGES ON MONTE CARLO SIMULATIONS

In this section, we will briefly study in more details the consequences of using different programming languages for implementing Monte Carlo simulations. Besides the standard Black-Scholes and the Merton jump-diffusion models, we will also consider a new type of options, the floating-strike lookback call option.

The Monte Carlo simulation (coded in the "*LookBackMonteCarlo*" excel subroutine and in the "mainLookBackMC.cpp" file) is performed as usual, with the asset price following the standard GBM, but substituting the strike price with the minimum (for a call) or maximum (for a put) reached by the stock price until the maturity date. The closed form analytical equations (coded in the "*LookBackClosed*" subroutine) used to price

lookback options with floating strikes were derived by Goldman, Sosin & Satto (1979) and are shown below (only for the call):

$$c = Se^{(b-r)T}N(a_1) - S_{min}e^{-rT}N(a_2) + Se^{-rT}\frac{\sigma^2}{2b}\left[\left(\frac{S}{S_{min}}\right)^{-\frac{2b}{\sigma^2}}N\left(-a_1 + \frac{2b}{\sigma}\sqrt{T}\right) - e^{bT}N(-a_1)\right]$$

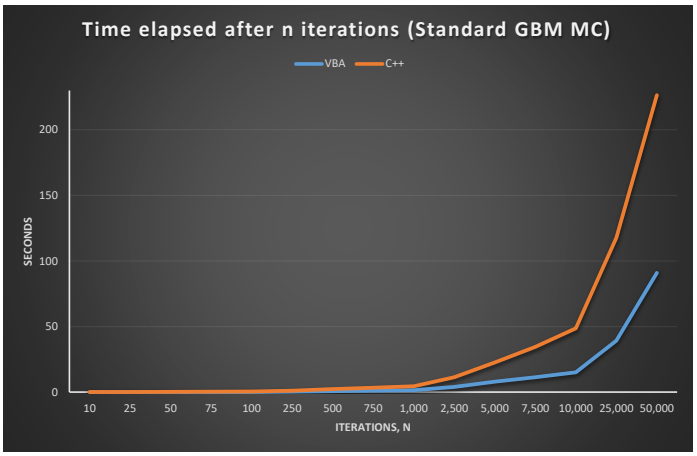
$$a_1 = \frac{\ln(S/S_{min}) + (b + \sigma^2/2)T}{\sigma\sqrt{T}}$$

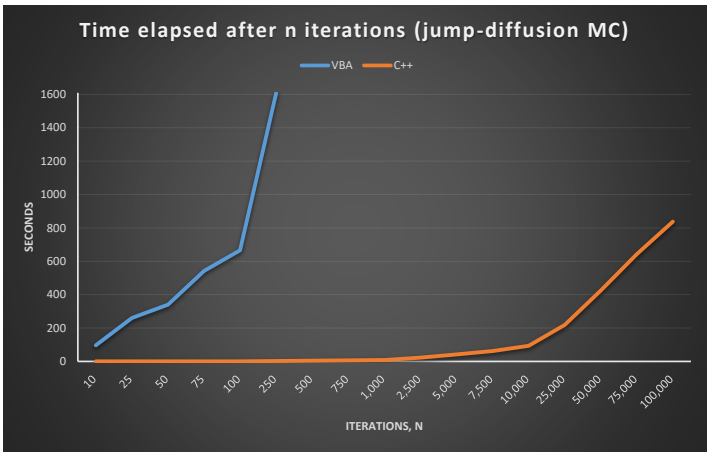
$$a_2 = a_1 - \sigma\sqrt{T}$$

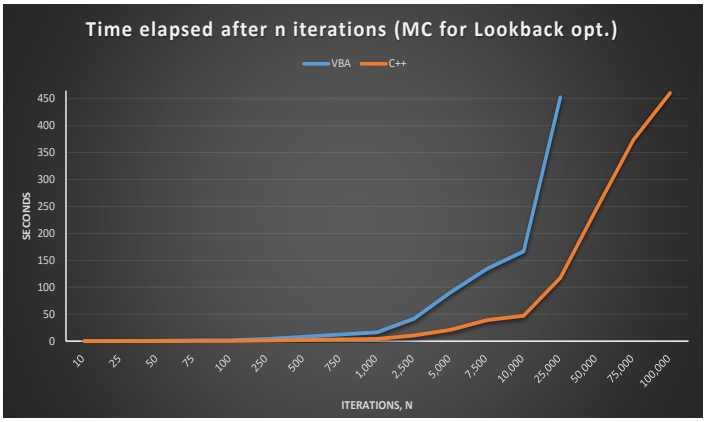
where $b = r$ since we assume our dividend yield is zero for sake of simplicity.

The first advantage of C++ over VBA is that C++ can theoretically implement a very large number of iterations without technical issues while VBA frequently runs into overflow problems, thus reducing the possible number of iterations. This allows consequently to reach a higher degree of accuracy with C++, which can compute up to 100.000 iterations without problems.

The second difference between the two programming languages is also the time required to perform a Monte Carlo simulation. If we measure time elapsed after n iterations, we obtain the following graphs:







We can therefore conclude that while VBA performs better in the case of the standard GBM Monte Carlo, C++ is extremely more efficient when we refer to the Monte Carlo simulations of the jump-diffusion process and the floating-strike lookback option; this is particularly true for the jump-diffusion Monte Carlo simulations, where VBA really struggles with computing the estimates, because of the compound Poisson process which is very computer-intensive, while C++ enjoys a built-in function much more efficient (coded in the "poisson" excel function and in the *"mainJDMonteCarlo.cpp"* file).