Chapter 1

Vanilla Option Pricing

1.1 Formulation of the problem

We start from the Black-Scholes Partial Differential Equation :

$$\partial_t P + (r_t - q_t) S \partial_S P + \frac{1}{2}\sigma^2(t,S) S^2 \partial_{SS} P - r_t P_t = 0 \tag{1.1}$$

Note that we could also have used the below equation:

$$\begin{cases} \partial_t \tilde{P}(t,x) + \frac{1}{2}\sigma^2(t, F_t e^x)(\partial_{xx}\tilde{P} - \partial_x \tilde{P}) = 0, \\ \tilde{P}(T,x) = e^{-\int_0^T r_s ds}\Phi(F(0,T)e^x) \end{cases} \tag{1.2}$$

where $P(t,S) = \tilde{P}(t,x)e^{\int_0^t r_s ds} = \tilde{P}(t,x)e^{rt}$ if r_s is constant, after the change of variable $x_t = ln(\frac{S}{F(0,t}$).
The two solutions are equivalent, and we can retrieve the option price by using $P(0,S_0) = \tilde{P}(0,0)$.
In the C++ implementation, I have decided to use equation (1.1) for tractability and ease of debugging.

The price P is solution of the more general equation:

$$\partial_t u + a(t,x)u + b(t,x)\partial_x u + c(t,x)\partial_{xx} u + d(t,x) = 0 \tag{1.3}$$

where we can define the following functions in our case:

$$\begin{cases} a(t,x) = -r_t \\ b(t,x) = (r_t - q_t) \times S_t \\ c(t,x) = \frac{1}{2}\sigma(t,x)^2 \times S_t \\ d(t,x) = 0 \end{cases} \tag{1.4}$$

It should be noted that since these four functions are assumed to be functions of parameters (t,x) and that we are discretizing over time and space, we can construct matrices representing these functions.

We discretize equation (1.3) on the resolution domain $[0,T] \times [x_m, x_M]$, yielding $(t_i)_{0\leq i\leq n}$ and $(x_j)_{0\leq j\leq m}$ respectively the time and the space axes, on which u is solution of the following:

$$\begin{aligned} &\left[\left(-\frac{b_{i,j}}{2\delta x} + \frac{\theta c_{i,j}}{\delta x^2} \right) u_{i,j-1} - \left(\frac{1}{\delta t_i} + \frac{2\theta c_{i,j}}{\delta x^2} - a_{i,j} \right) u_{i,j} + \left(\frac{b_{i,j}}{2\delta x} + \frac{\theta c_{i,j}}{\delta x^2} \right) u_{i,j+1} \right] \\ &+ \left[\frac{(1-\theta)c_{i,j}}{\delta x^2} u_{i+1,j-1} + \left(\frac{1}{\delta t_i} - \frac{2(1-\theta)c_{i,j}}{\delta x^2} \right) u_{i+1,j} + \frac{(1-\theta)c_{i,j}}{\delta x^2} u_{i+1,j+1} \right] \\ &+ d_{i,j} = 0 \end{aligned} \tag{1.5}$$

Using a matrix form, we obtain the recursive equation:

$$U_i = -(P_i)^{-1}(Q_i U_{i+1} + V_i) \tag{1.6}$$

where:

$$P_i = \begin{pmatrix} a_{i,1} - \left(\frac{1}{\delta t_i} + \frac{2\theta c_{i,1}}{\delta x^2}\right) & \frac{b_{i,1}}{2\delta x} + \frac{\theta c_{i,1}}{\delta x^2} & 0 & \cdots & 0 \\ -\frac{b_{i,2}}{2\delta x} + \frac{\theta c_{i,2}}{\delta x^2} & a_{i,2} - \left(\frac{1}{\delta t_i} + \frac{2\theta c_{i,2}}{\delta x^2}\right) & \frac{b_{i,2}}{2\delta x} + \frac{\theta c_{i,2}}{\delta x^2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & -\frac{b_{i,m-1}}{2\delta x} + \frac{\theta c_{i,m-1}}{\delta x^2} & a_{i,m-1} - \left(\frac{1}{\delta t_i} + \frac{2\theta c_{i,m-1}}{\delta x^2}\right) \end{pmatrix}$$

$$Q_i = \begin{pmatrix} \frac{1}{\delta t_i} - \frac{2(1-\theta)c_{i,1}}{\delta x^2} & \frac{(1-\theta)c_{i,1}}{\delta x^2} & 0 & \cdots & 0 \\ \frac{(1-\theta)c_{i,2}}{\delta x^2} & \frac{1}{\delta t_i} - \frac{2(1-\theta)c_{i,2}}{\delta x^2} & \frac{(1-\theta)c_{i,2}}{\delta x^2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & \frac{(1-\theta)c_{i,m-1}}{\delta x^2} & \frac{1}{\delta t_i} - \frac{2(1-\theta)c_{i,m-1}}{\delta x^2} \end{pmatrix}$$

$$V_i = \begin{pmatrix} d_{i,1} + \left(-\frac{b_{i,1}}{2\delta x} + \frac{\theta c_{i,1}}{\delta x^2}\right) u_{i,0} + \frac{(1-\theta)c_{i,1}}{\delta x^2} u_{i+1,0} \\ d_{i,2} \\ \vdots \\ d_{i,m-1} + \left(\frac{b_{i,m-1}}{2\delta x} + \frac{\theta c_{i,m-1}}{\delta x^2}\right) u_{i,m} + \frac{(1-\theta)c_{i,m-1}}{\delta x^2} u_{i+1,m} \end{pmatrix}$$

We solve this equation backward using the terminal condition in T:

$$u(T, x) = \psi(x), \forall x \in [x_m; x_M] \tag{1.7}$$

combined with boundary conditions:

$$\forall t \in [0, T], \begin{cases} u(t, x_m) = u_-(t) \\ u(t, x_M) = u_+(t) \end{cases} \tag{1.8}$$

1.2 C++ implementation

The code is decomposed in three main parts:

- Matrix class: representing vectors of vectors and enabling us to perform elementary matrix operations (addition, multiplication, ...) as well as computing the inverse of a matrix.
- PDEPricer class: performing the backward resolution of equation (1.6).
- utils file: computing the price given by the analytical Black-Scholes formula.

1.2.1 Matrix Class

The most important point to detail here is the computation of the inverse of a matrix. I initially used co-factor expansion, but for $n > 20$ this turned out to be intractable (the complexity is $O(n!)$).
I therefore used LU decomposition, for which the complexity is $O(n^3)$. I also checked if using a specialized library improved the performance (namely the Eigen library), but the gains were negligible even on large grids.

1.2.2 PDE Solver Class

We solve equation (1.6) backward by starting from $t = T - \delta t \leftrightarrow i = n$ and using the terminal condition at $t = T \leftrightarrow i = n + 1$.
At each timestep we recompute the matrices P_i, Q_i, V_i, though in the example we are considering here they are fixed. To compute these matrices, we use the Crank-Nicholson scheme by using the smoothing parameter $\theta = 0.5$.
Once we reach the timestep $t = 0$, i.e. the time at which we value our option, we do a linear interpolation to find the option price corresponding to the initial spot price S_0.

1.3 Example of a European call option

We use the following terminal conditions:

$$P(T, S) = (x - K)^+, \forall x \in [x_m; x_M] \tag{1.9}$$

and the associated boundary conditions are:

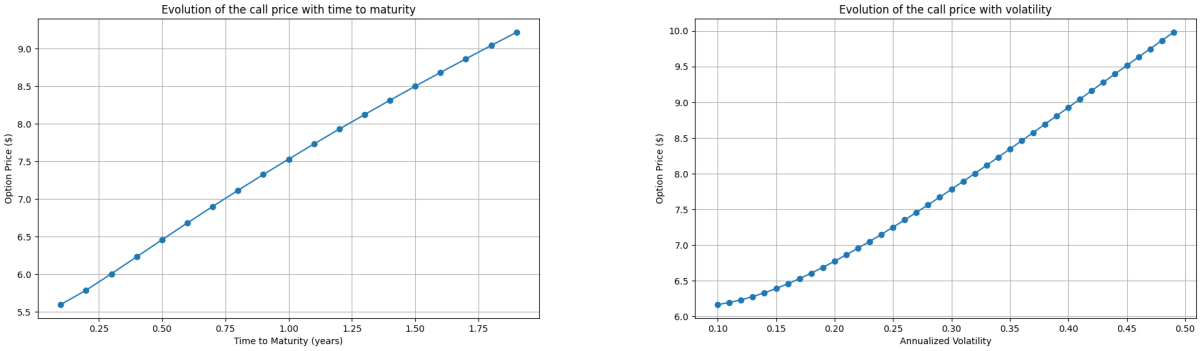
$$\forall t \in [0, T], \begin{cases} P(t, x_m) = (x_m - Ke^{-(r-q)t})^+ \\ P(t, x_M) = (x_M - Ke^{-(r-q)(T-t)})^+ \end{cases} \tag{1.10}$$

We are now equipped to compute the price of our call, using both the finite difference method and B.S. formula. We will use the following characteristics (arbitrarily):

Parameter	Value	Meaning
S_0	50	Initial spot price
K	45	Strike
σ_0	45%	ATM volatility
r	3%	Risk-free rate
$q = repo + divYield$	0%	Repo rate and dividends
n	50	Number of points in the time grid
m	100	Number of points in the space grid
λ	4	Parameter controlling the width of the space grid

Table 1.1: Parameters of our call option

Before checking if the the prices we obtain using both methods match, we want to check if the evolutions of the prices obtained using the finite difference method are intuitive. We plot the prices against time to maturity and volatility:

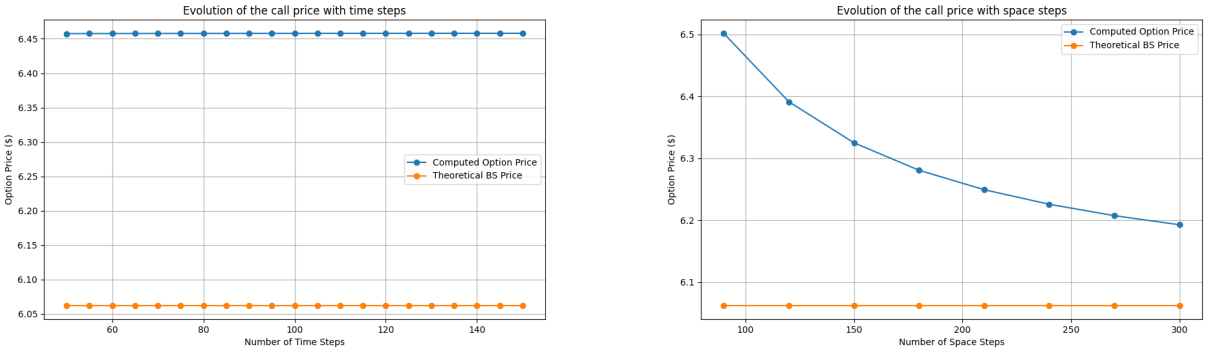


(a) $P = f(T)$ (b) $P = f(\sigma_0)$

Figure 1.1: Sanity checks of our pricer

The shapes are the ones we are expecting and are consistent with the intuition: higher time to expiry leads to greater potential fluctuations in the stock price, whereas a higher volatility increases the likelihood of the option ending I.T.M., both increasing the option price.

We now look at the convergence (if any) towards the Black-Scholes price, by playing with the parameters n and m:



(a) $P = f(n)$ (b) $P = f(m)$

Figure 1.2: Evolution of the prices depending on the discretization parameters

As we can see, as long as the number of time steps is reasonably high, the marginal gains we get by increasing it are quite low. We observe much better results by increasing the number of steps in the space grid, since a linear interpolation between two spot prices is a quite strong assumption (but simple to implement). However, the prices obtained using both methods remain fairly similar, and the stability of the discretization is encouraging.
We now turn ourselves to a different approach, free of these two parameters, namely Monte Carlo and Quasi-Monte Carlo methods. These two methods are mostly used when dealing with more complex payoffs or higher dimensional problems (e.g. when an option depends on multiple underlying assets) and the P.D.E. method becomes impractical due to the "curse of dimensionality."

Chapter 2

Extensions to Monte Carlo & Quasi-Monte Carlo methods

2.1 Formulation of the problem

While there is almost no advantage in using a Monte Carlo (M.C. later on) simulation to price a European call option (since a closed-form formula exists), the methodology is still interesting and can easily be extended to other derivatives (e.g. the pricing of American & Asian options).
This is also the occasion to introduce Quasi-Monte Carlo (Q.M.C.) methods, which we will summarize in the following.

Using the classic geometric Brownian motion to model the stock price:

$$dS_t = (r - q)S_t dt + \sigma S_t dW_t \tag{2.1}$$

one can easily retrieve the expression of S_t:

$$S_t = S_0 e^{(r-q-\frac{1}{2}\sigma^2)t+\sigma\sqrt{t}Z}, \text{ where } Z \sim \mathcal{N}(0,1) \tag{2.2}$$

By using a random number generator (commonly available in Python an C++ for example), one can generate thousands of random paths for the stock price and price the derivative using the expectation of the discounted payoff. In the case of a European call option:

$$C = e^{-rT}\mathbb{E}[\max(S_T - K, 0)] \tag{2.3}$$

The main difference between M.C. and Q.M.C. lies in the fact that the latter uses deterministic sequences of number instead of random ones to achieve a faster convergence ($O(N^{-1})$ against $O(N^{-\frac{1}{2}})$). In practice, instead of having a sequence (Z_i) where $\forall i, Z_i \sim \mathcal{N}(0,1)$, we construct (Z_i) as a Sobol sequence to get a more evenly distributed sample (we chose a Sobol sequence to generate quasi-random numbers but other choices are possible too, e.g. Faure sequence):

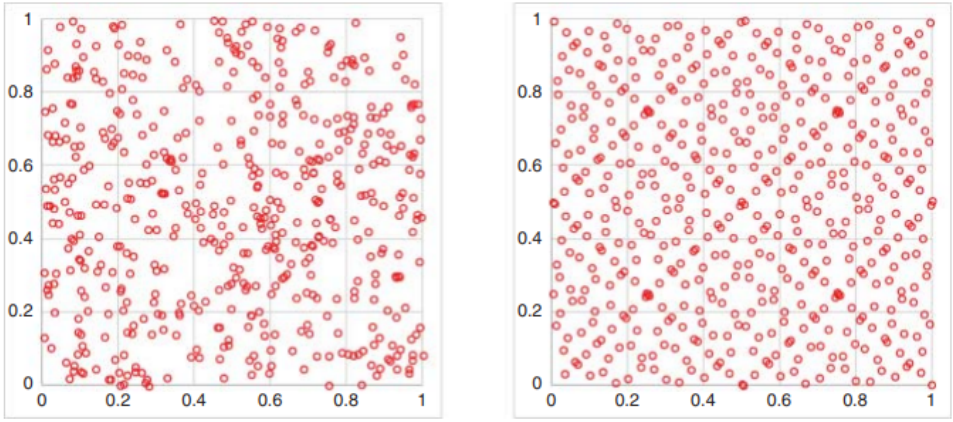


Figure 2.1: A random and a Sobol sequence. Source: Modern Computational Finance, 2018

It can indeed be shown using the Koksma-Hhlawka inequality that random sampling is not optimal for numerical integration, as it allows for clusters and does not specifically look to minimize discrepancy.
To construct a Sobol sequence, we use the updated algorithm proposed by Antonov and Saleev in 1979 and detailed in *Modern Computational Finance* (Savine 2018), which consists in:

- Generate sequences of integers (y_i^d) between 0 and $2^{32}-1$ so the i-th number on the axis d is:

$$x_i^d = \frac{y_i^d}{2^{32}} \tag{2.4}$$

  The integers y_i^d are produced recursively using the bit-wise XOR operation:

$$\begin{cases} y_0^d = 0 \\ y_{i+1}^d = y_i^d \oplus DN_{J_i}^d \end{cases}$$

  where J_i is the rightmost 0 bit in the binary expansion of i (0 for even numbers), and $(DN_j^d), j \in [0, 31]$ are the 32 direction numbers for the sequence number d.

- Construct the sequence of the direction numbers (how to do so is out of the scope of this project -and even out of the scope of the book mentioned above- but they are available in most C++ and Python libraries up to high dimensions).
  In the case of our European call, we are only interested in one dimension since we only want to know the stock price at $t = T$, and for the first axis $d = 0$, the direction numbers are simply:

$$DN_k^0 = 2^{31-k}$$

Once our Sobol sequence is constructed, we have a uniformly $(0, 1)$ distributed set of numbers, which we want to convert to a standard normal distribution. To do so, we need to use the inverse normal CDF Φ^{-1}, solving:

$$Z_i = \Phi^{-1}(u_i), \text{ where } Z_i \sim \mathcal{N}(0,1) \text{ and } u_i \sim \mathcal{U}(0,1)$$

However, solving this equation is computationally intensive, given that:

$$\Phi^{-1}(p) = \arg\max_x \left\{ \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-t^2/2} dt = p \right\}$$

To palliate this, we use the Box-Muller method to obtain standard normal variables from uniform variables.

Box-Muller Theorem: Suppose U_1 and U_2 are independent samples chosen from the uniform distribution on the unit interval $(0, 1)$. Let:

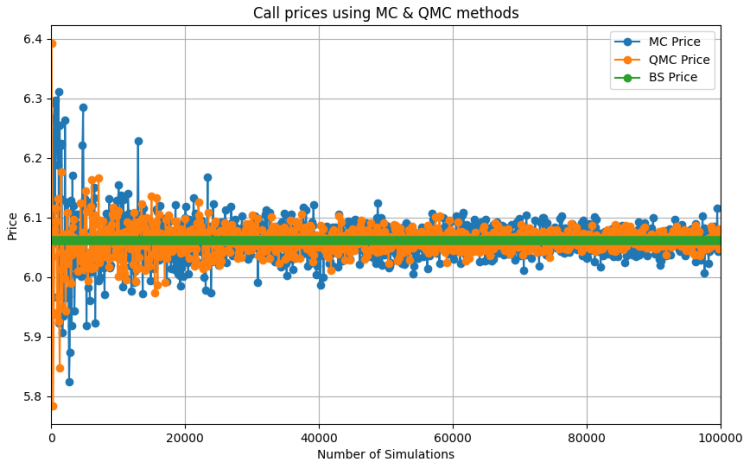
$$\begin{cases} Z_0 = \sqrt{-2ln(U_1)} \cos 2\pi U_2 \\ Z_1 = \sqrt{-2ln(U_1)} \sin 2\pi U_2 \end{cases}$$

Then Z_0 and Z_1 are independent random variables with a standard normal distribution.

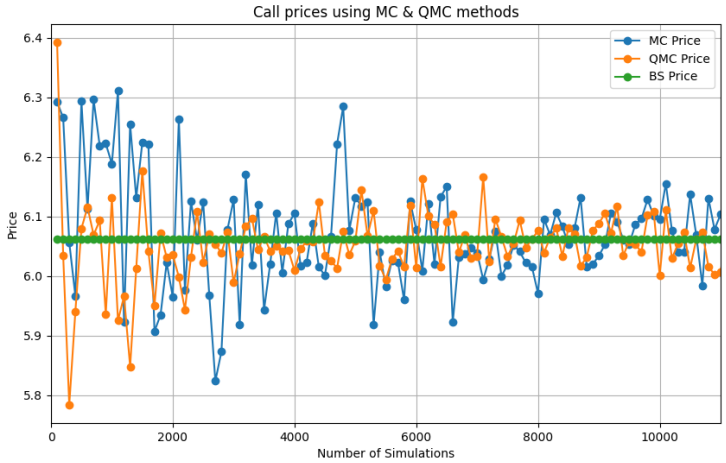
Equipped with this theory, we can build our pricer using both M.C. and Q.M.C. methods, which we will compare in the next part.

2.2 Going back to our European call option

We want to compare here the speed of convergence of the two methods. To do so, we run various simulations for different numbers of iterations (i.e. different numbers of stock paths) and compute the option price with the two methods. We do so from 100 to 100000 iterations and obtain:



(a) Prices full sample



(b) Prices reduced sample

Figure 2.2: Evolution of the prices depending on the number of simulations

While we can observe on (a) that the Q.M.C. prices have a lower standard deviation compared to the M.C. prices, this is even more obvious on (b), where the convergence appears as soon as the ≈ 2000 iterations for the Q.M.C method while we have to wait around the ≈ 7000 iterations for the M.C. method. Both methods yield prices close to the Black-Scholes price, which is expected by formulation of the problem. The true strength of the Quasi-Monte Carlo method is revealed in more complex settings (such as the pricing of basket of options, American options, etc.) where the number of dimensions $d > 1$.