ENSAE Paris

C++ project report

Option pricing with Binomial trees and Monte-Carlo simulations

Maxime Coppa, Adam Wolljung, Marama Simoneau

December, 30th 2023

Contents

Introduction

Option pricing have been a topic of interest for many years in the financial industry. There are several option pricing methods, including Binomial trees and Monte-Carlo simulations. The first one has been introduced in 1979 by Cox, Ross and Rubinstein while the second one was applied first in 1977 by Phelim Boyle.

In this project, we have implemented both option pricing methods. The simulation allows users to calculate the price of an option knowing the current price of the underlying asset, the strike price and the time to maturity. The simulation was implemented using computer science principles and algorithms, including data structures and mathematical models.

The report is structured as follows: In the next section, we provide an overview on how to run our option pricing model. In the following section, we present the rules and implementation of options, including the algorithms and data structures used. Then we explain how we estimated volatility of APPL stocks. In the fourth part, we show how we implemented Binomial tree simulations and in the last part we breakdown our work on Monte-Carlo simulations.

1 How to run the program

Our project contains several files, including a main file. The idea was to separate the files to make them clearer.
So we have the following files :

- `main.cpp` file containing the main code, our test functions and the main function
- `Option.cpp` and `Option.hpp` files for creating our Option class
- `Binomial.cpp` and `Binomial.hpp` files for creating our Binomial class and applying the Binomial method
- `Monte_Carlo.cpp` and `Monte_Carlo.hpp` files to create our Monte-Carlo class and formalise the Monte-Carlo method in code
- `Volatility_estimator.cpp` and `Volatility_estimator.hpp` files to estimate the volatility of our underlying asset

In the `main.cpp`, we created two test functions `test1` and `test2`. They are void functions that return the price of the option with the Binomial Method and also with the Monte-Carlo Method. It allows us to compare the results that we find with the two methods.
Here is a more detailed explanation :

- `test1()` gives the price of a call option, with Apple being the underlying asset, with both the Binomial Method and the Monte-Carlo Method
- `test2()` gives the price of a put option, with Apple being the underlying asset, with both the Binomial Method and the Monte-Carlo Method

One could try one function at a time to see that our code works.

It is also possible to carry out tests quite easily. All you have to do is decide on which underlying asset the option is to be implemented. Then all we have to do is change the parameters we want (maturity, risk free rate, etc.) in relation to the option we want to price. All that's left to do is run our code and see the prices returned by the two methods.

We also implemented a visualization method `Monte_Carlo_Simulation_Visualization(Option)` for the underlying asset of the option simulates thanks to the Monte-Carlo Method. It requires you to have `GnuPlot` installed on your software and then run the code.

2 Rules and Implementation of the Options

2.1 Motivation

The first challenge that we had to face was to implement and to model an Option. We chose to price only European options and we then decided to describe an option by its own features and the ones of the underlying asset.
We created an accessible and universal model to represent an option based on the Financial Instruments course.

2.2 Class `Option`

In order to implement and to have the possibility to modify the Option, we have defined the class `Option`.

The attributes of that are:

- `Type`: Integer representing the type of the option 1 for a call option or -1 for a put option.
- `r`: Double. Represents the risk-free interest rate, a critical parameter in option pricing models like Black-Scholes.
- `K`: Double. Denotes the strike price of the option, the predetermined price at which the underlying asset can be bought or sold.
- `T`: Double. Represents the expiry date of the option, indicating when the option contract ceases to be valid.
- `sigma`: Double. Signifies the volatility of the underlying asset, a crucial input for option pricing models like Black-Scholes.
- `S0`: Double. Denotes the initial price of the underlying asset, essential for calculating the option's intrinsic value.

Once we defined the attributes we implement the Constructors, Destructors, Accessor methods and Modification methods to manipulate properly the class.

For the default constructor we used a European call option on the Apple Stock price with a maturity of 1 year, a $S_0 = \$200$, a strike price of $K = \$220$ based on a quick analysis of the trend of Apple, $r = 0.0473$ and a $\sigma = 0.126299$ based on an estimation that we will explain in the next paragraph.
At the end we implement `output()` to facilitates the display of option information and its key parameters.

3 Estimation of the volatility of the underlying asset

3.1 How can we estimate the volatility of the underlying asset ?

We studied volatility in the Financial Instruments course and we learnt that its estimator is quite easy to compute. It is given by the following formula :
$\hat{\sigma} = \sqrt{\frac{1}{N-1} \sum_{i=1}^{N} \left(\ln\left(\frac{P_i}{P_{i-1}} \right) - \bar{r} \right)^2}$

Where $\bar{r}$ is the average log-return ie the average of $\ln(\frac{P_i}{P_{i-1}})$. As you may noticed, it only requires historical prices of the asset, we picked APPL stock monthly prices from 1980 to 2023. We picked a long period of time because both pricing models assume that volatility is constant. Note that this assumption is strong but it is outside of the scope of this work.

3.2 VolatilityEstimator class

The C++ `VolatilityEstimator` class handles the estimation of volatility based on historical prices of a financial asset. This class uses `fstream` which is a header file used for file output and input operations. We especially used its `ifstream` class which enables reading data from files. We also use `stringstream` from `sstream` header, it provides input and output functionalities for strings.

3.2.1 Private member

This class contains one private member which is `prices`, it is a vector of doubles containing historical prices.

3.2.2 Public members

The public members are :

- constructor : the constructor initializes the `VolatilityEstimator` object. It takes a filename as input to load historical price data from a .txt file.
- `readTextFile` : this method reads the content of a .txt file given its filename and returns the content as a string. This method will be used in the following method.
- `readStockFile` : this method reads stock prices from a .txt and store them in a vector of doubles.
- `calculateVolatility` : this method calculates the volatility estimator based on the historical prices stored in the class. It's marked as const since it doesn't modify the internal state of the object.

You might have noticed that this class is defined by the filename of historical prices, actually we could define it directly with the vector of prices but it would not be relevant since financial data are stored in files so we adapted our class to make it work with the filename.

3.3 How to use this class ?

In order to compute the volatility estimator, you must first download historical prices of an asset under .txt format. This can be done by downloading a .csv file on Yahoo finance and convert it to .txt format. We chose to work with .txt files because it seemed simpler with C++ but it is possible to do so with .csv documents. Once you have downloaded the .txt document, you just have to replace the filename by the path to the text file in the main C++ file.

4 Binomial tree Simulation

4.1 Explanation of the Binomial method

The Binomial model for option pricing is a method that divides time into discrete intervals, imagining two possible movements (up or down) for the underlying asset's price at each step. The model constructs a binomial tree, where each node represents a possible future price. Option values are calculated at each node based on the expected payoffs, considering factors like strike price and current stock price. The model iteratively calculates these values backward through the tree, ultimately determining the option's present value. While conceptually simple, the Binomial model assumes constant volatility and discrete price changes, which may limit its accuracy in capturing real-world market dynamics. Despite its limitations, it serves as a foundation for understanding more complex option pricing models.

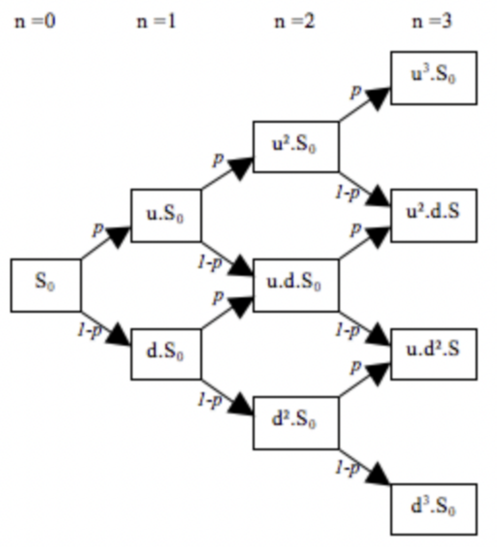

Figure 1: Illustration of the Binomial Method

In order to implement this method in our program, we decided to create a class called `Binomial`.

4.2 Binomial class

The idea of this class is to create an object with attributes that are needed in the Binomial method. In order to have this, we need to settle attributes and also setter methods. Then we are going to make a method that takes in argument an object from the class `Option`, that will compute the price of the option thanks to the Binomial method.

The attributes are :

- `PV` : the present price of the option (what we are trying to estimate). It takes positive real values

- `p` : the probability that the price of the underlying asset goes up (1-p would be the probability that the price goes down). It takes values between 0 and 1
- `fu` : the value of the option if the price of the underlying asset goes up. It takes positive real values
- `fd` : the value of the option if the price of the underlying asset goes down. It takes positive real values
- `u` : the magnitude of an up-jump of the underlying asset. It takes positive real values
- `d` : the magnitude of a down-jump of the underlying asset. It takes positive real values

All variables have been initialized to 0, except p, which takes the value 0.5. The value of p, u and d can be calculated pretty easily thanks to some attributes of an objet from the class `Option`. We have :

- $u = exp(\sigma * \sqrt{\Delta * T})$
- $d = exp(-\sigma * \sqrt{\Delta * T}) = 1/u$
- $p = (e^{r0*T} - d)/(u - d)$

We will apply those formulas for our setter methods in our class `Binomial`.

4.3 Methods in the class

4.3.1 Constructor, Setter and get methods

We created simple functions that are part of a class for pricing European options using the binomial model:

Constructors:
Default Constructor: Initializes with default values. Parameterized Constructor: Sets attributes based on provided values.

Getters:
Returns values such as up-jump magnitude, down-jump magnitude, probability of up-jump, option values at up and down nodes, and current option value.

Setters:
Sets up-jump and down-jump magnitudes, probability of up-jump, and call/put option values.

These functions enable the setup and manipulation of option parameters and values for pricing using the binomial model.

4.3.2 The Binomial method with a one step tree

First, we wanted to make a function that will apply the Binomial method with a one step tree. This function, `Binomial_Method_Price_option_one_step`, implements a one-step binomial method for pricing a European option. Let's explain its steps:

Input Parameters:
The function takes an `Option` object o as input, representing the details of the option (e.g., type, risk-free rate, strike price, expiration date, volatility, and initial asset price).

Calculation Using Binomial Model:
It initializes a local `European_Option_Binomial` object B. Depending on the option type (call or put), it sets the magnitude of an up-jump, calculates the probability of an up-jump, and sets the call or put option values at the expiration date.
To set the magnitude of an up-jump, the magnitude of a down-jump and the probability of an up-jump, we call the functions defined above.
In order to compute `fu` and `fd`, we apply those formula :

- $fu = max(0, \alpha(K0 - (u * S0)))$
- $fd = max(0, \alpha(K0 - (d * S0)))$
- with $\alpha = 1$ for a put option and $\alpha = -1$ for a call option

Calculation of the option's price:
Now that we have `fu` and `fd`, we compute the option's price with this formula :

- $PV = ((p * fu) + (1 - p) * fd)/(1 + r)$

Return Value:
The function returns the current value of the option, as calculated by the binomial model for the one-step scenario.

Note:
The local `European_Option_Binomial` object B is used to perform the binomial calculations and obtain option values. The function then prints information and returns the calculated option value. In summary, this function provides a simple demonstration of the one-step binomial method for pricing European options.

4.3.3 The Binomial method with a n step tree

Now we want to apply the Binomial method, but this time with several steps in the tree. We do this to be more precise. This function, `Binomial_Method_Price_option_steps`, implements the Binomial method for option pricing with a specified number of steps (n) in the tree. Here's a breakdown:

Input Parameters:
The function takes an `Option` object o representing the details of the option and an integer n specifying the number of steps in the binomial tree.

Validity Check:
It checks if the number of steps n is a valid positive value. If n is not positive, it prints an error message and returns 0.

Binomial Model Parameters:
It extracts parameters such as the risk-free rate (r), strike price (K), expiration date (T), volatility (sigma), initial asset price (S0), and option type (type0) from the Option object.

Calculation of Up and Down Movements:
It calculates the factors u0 and d0 based on the specified number of steps and the time increment δT.

Risk-Neutral Probabilities:
It calculates the risk-neutral probabilities p and q based on the calculated up and down movements.

Option Value Calculation:
It initializes an array `valueoff` to store option values at each node. Depending on the option type, it calculates option values at each node using the binomial formula.

Price Computation:
It computes the final option price at time 0 by discounting the value at the initial node using the risk-free rate.

Return Value:
It returns the calculated option price.

In summary, this function implements the Binomial method for option pricing with a specified number of steps in the tree. It calculates option values at each node, considering the up and down movements, and computes the final option price at time 0. Debugging statements are included for better understanding and validation of intermediate calculations.
The function has a time complexity of $O(n^2)$, where 'n' is the number of steps in the binomial tree. This means that as the number of steps increases, the time it takes for the function to run grows quadratically. The reason for this complexity is the presence of nested loops in the code. The more steps you have, the more computations are performed, and the overall time needed increases significantly. Using this method, we find as a result c `= 6.16409`.

5 Monte Carlo Simulation

5.1 Motivation

The decision to employ a Monte-Carlo Simulation in our project stems from its ability to address the fundamental principles of financial markets, namely uncertainty and randomness. In the context of option pricing, where future outcomes are inherently uncertain, the Monte-Carlo Simulation excels by generating a multitude of potential future scenarios. By simulating the evolution of the underlying asset's price over time, this method captures the stochastic nature of financial markets, adhering to the key principle that market dynamics are unpredictable.

To simulate the underlying asset we used Black-Scholes-Merton model and we discretize the time to express S_t in terms of S_{t-1} over a small time interval Δt. By differentiating the Black-Scholes-Merton equation we obtain $S_t = S_{t-1}e^{(r-\frac{\sigma^2}{2})\Delta t+\sigma\sqrt{\Delta t}Z_t}$ where Z_t is a random sample from a standard normal distribution.

In order to provide an accurate and clear used of the Monte-Carlo Simulation we decided to implement a class `Monte_Carlo` and we focus our work on the simulations. Afterwards we implement a method to visualize the different scenarios.

5.2 Global structure

The different attribute of the class `Monte_Carlo` are the following:

- `N`: Integer representing the number of simulations. This parameter defines the quantity of random paths generated in the Monte-Carlo simulation, influencing the accuracy and granularity of the option pricing estimation.
- `n`: Integer indicating the number of divisions for the time of the option. In the context of the Monte-Carlo simulation, this parameter influences the discretization of time intervals, affecting the precision of the simulation results.
- `N_visualization`: Integer specifying the number of simulations for visualization purposes. This parameter allows a subset of simulations to be chosen specifically for creating visual representations, streamlining the process of generating graphs and charts to interprate results.

Once we defined the attributes we implement the Constructors, Destructors, Accessor methods, Modification methods and an `output()` function to manipulate properly the class.
We then implement the Monte-Carlo simulation method and the visualisation.

5.3 Detailed methods overview

5.3.1 `random_number`

A key feature from Monte-Carlo simulations is randomness. Thus we created a function: `random_number()` to simulate a random sample from a standard normal distribution.

To achieve this goal, we used the `<random>` library. This library provides facilities for generating random numbers. It is a part of the C++ Standard Library and includes several classes and functions for various random number generation tasks.

At first, we initialize a random device to obtain a nondeterministic seed for random number generation. This seed is then used to initialize a Mersenne Twister pseudorandom number generator. It was very important to initialize the Mersenne Twister pseudorandom number generator to ensure that the sequence of random numbers generated is not predictable and varies between different runs of the program.
Following the engine setup, a standard normal distribution is created. Finally, we return a random number sampled from this normal distribution.

This process ensures that the Monte-Carlo simulation simulates proper random sample.

5.3.2 Monte-Carlo simulation

At this time we had all the features to implement `Monte_Carlo_Simulation_Price`, to conduct a Monte-Carlo simulation for option pricing using the Black-Scholes model. The method takes an instance of the `Option` class as we defined above.

The Monte Carlo simulation is executed N times, where N is the number of simulations specified in the `Monte_Carlo` class. Each simulation involves generating a random trajectory for the underlying asset's price over time, with the BlackScholes model used to update the asset price at each time step. We used n steps to discretize the time. The trajectory is influenced by factors such as the risk-free rate, volatility, and the random component Z generated thanks to the `random_number()` method. The payoff of the option is then calculated based on the simulated asset prices.

The discounted payoffs are accumulated over all simulations, and the final result `ret` represents the average discounted payoff over the specified number of simulations N. This provides an estimation of the option's expected value based on the Monte Carlo simulation.

In summary, the complexity of this program is $O(nN)$ which is quite good. We simulate an example with $n = 1000$ and $N = 100000$ for a basic `Option` as explained above. We find as a result c `= 6.01526`. To test the program we could used known benchmarks or analytical solutions to enhance the program's reliability.

5.3.3 Monte-Carlo visualization

The last part of our work on the Monte-Carlo simulation was to visualize the different trajectories of the underlying asset simulating thanks to the Monte Carlo simulation and BSM model. This visualization gives us the opportunity to communicate the result of the simulation and also to understand the dynamics of the model used.

To achieve this goal we used Gnuplot. Gnuplot is a plotting utility that facilitates the visualization of data through various types of plots and charts. Gnuplot is well suited with C++. Gnuplot is easy to use and we can quickly integrate Gnuplot into our C++ code without significant overhead, making it accessible for a wide range of applications. Moreover Gnuplot is available on multiple platforms, including Unix, Linux, Windows, and macOS. This cross-platform compatibility was a versatile choice for us because we all worked on MacOS and there are not a lot of tools to draw plots in C++ for MacOS.

As a consequence we used Gnuplot to draw the different trajectories of the Monte Carlo simulation. After initializing the option parameters, we established a communication channel with Gnuplot using a pipe. Subsequently, we crafted a Gnuplot script within the C++ code,

dynamically generating plot specifications such as title, labels, and initial setup. The heart of the interaction lies in the loop where asset values at different time points are sent to Gnuplot through the established pipe, enabling real-time construction of the plot. The script is punctuated with the 'e' notation to signify the end of data points. Upon the completion of the simulation, the pipe to Gnuplot is closed.
We had a long discussion between plotting each point at each step and creating a `Vector` to stock them and then plot them at the end. The second step has the advantage to create a more universal class `Plot` and which could be used for others purpose. Nevertheless we prefer the second option because the memory complexity was lower and we don't want to focus on Gnuplot but only used it as an help and a communication tool.

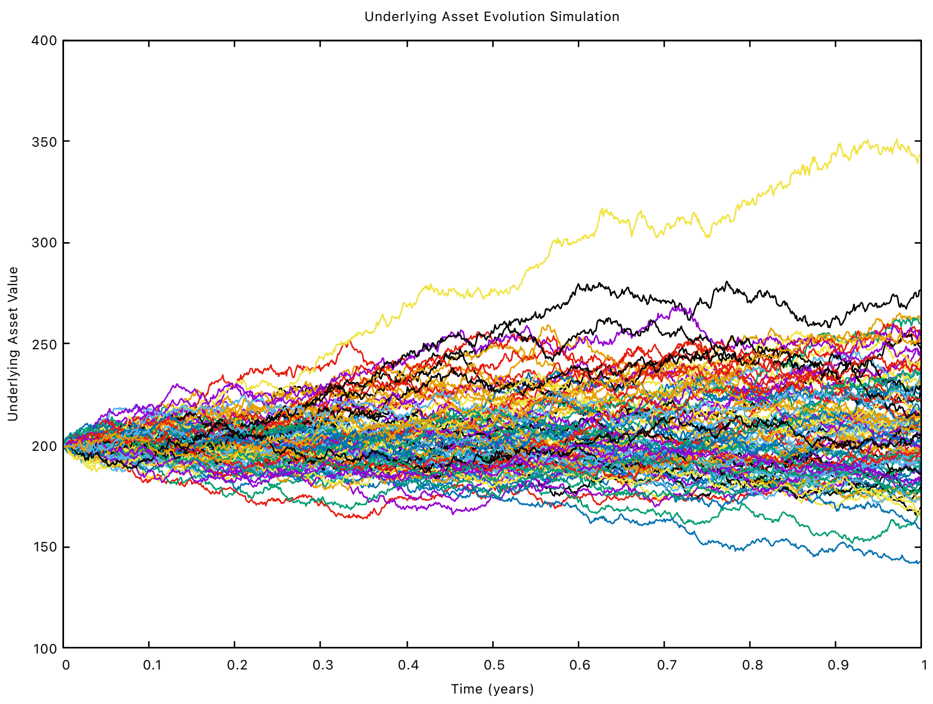

Figure 2: Visualisation des simulations de Monte Carlo pour `N_visualization = 100`