Variance reduction's methods for options' pricing using Monte Carlo

by

Anthony JACOB • Antoine BLEYS

EDHEC Business School and Université Côte d'Azur

Supervisor: **Professor Laurent DEVILLE**

SUMMARY

LIST OF FIGURES AND TABLES

1. INTRODUCTION

The valuation of financial options is a crucial component of modern financial markets. These derivative instruments enable investors to hedge risks, speculate, or structure portfolios optimally. The inherent complexity of options and their valuation necessitates a deep understanding of mathematical models and advanced simulation methods. Among these tools, variance reduction methods play an essential role in enhancing the efficiency and accuracy of Monte Carlo simulations, a widely used technique for option pricing.

In this thesis, we focus particularly on the efficiency of variance reduction methods in the context of different types of options: European, American, Asian, and barrier options. Each type of option has specific characteristics that influence their behavior and valuation. For example, European options, the simplest form, can only be exercised at expiration, while American options can be exercised at any time before expiration. Asian options depend on the average price of the underlying asset over a given period, and barrier options are activated or deactivated when the price of the underlying asset reaches a certain level.

The Monte Carlo method, widely used for simulating option prices, relies on random simulations to estimate option values. Although powerful, this method can suffer from slow convergence and high variances, requiring a large number of simulations to obtain precise estimates. Variance reduction methods, such as antithetic variables or control variates, are introduced to mitigate these drawbacks. These techniques aim to reduce the inherent uncertainty in random simulations, thereby increasing the efficiency of the computations.

This thesis is structured around three main axes. First, we present a general theory of options, including the different categories of options and their payoff characteristics. Then, we delve into option valuation models, particularly the Black-Scholes-Merton model and the Monte Carlo method. Special attention is given to variance reduction methods and their application in the context of option price simulations. Finally, we conduct a numerical analysis to evaluate the efficiency of these methods for different categories of options.

The central question of our research is: **Are variance reduction methods similarly efficient for all types of options?** To answer this question, we will analyze and compare the efficiency of different variance reduction techniques applied to European, American, Asian, and Barrier options. We will implement Monte Carlo simulations for each type of option and assess the efficiency gains obtained through the application of variance reduction techniques.

2. GENERAL THEORY OF OPTION PRICING

2.1. Options

2.1.1. Definition

An option is a type of financial asset. It is a contract conferring to the buyer (also called the holder) the right to do something. An option is linked to what is called the underlying asset. This thesis is concerned exclusively with stock options (options whose underlying asset is a stock). There exist two generic types of options, Call options and Put options. Call options confer to the buyer the right to buy the underlying stock at a predetermined price (the strike price), at a predetermined date or period. Put options confer to the buyer the right to sell the underlying stock at a predetermined price and at a predetermined date or period. The buyer of an option is buying the right to buy (resp. sell) the underlying stock, the seller will therefore have the obligation to sell (resp. buy) the underlying stock.

2.1.2. Payoff of options

As financial assets, one of the essential property of options is called the payoff function and refers to its profit and loss (PnL). This payoff function is a mathematical expression of the holder's (resp. seller's) PnL without considering the premium paid (resp. received) or transaction costs. The payoff is generally a function of some parameters such as the current stock price and the strike. We will see that there exist different types of options leading to different ways to compute the payoff function.

2.1.3. Payoff of European Options

European options are probably the simplest type of options. European options confer to the holder the right to buy or sell the underlying stock at the strike price, which value is defined in the option's contract, at a unique date in the future called the expiration date.

If we consider the case of a European call option, with strike price K, and expiration date T, there are two possible generic outcomes:

- Spot price of the stock at expiration date $S_T < K$

  The buyer of the Call has the right to buy the stock at the strike price K which is lower than the spot price S_T. Then, the investor will use his right to buy the stock at the strike price K and sell it at the spot price S_T. The payoff of the transaction is $S_T - K > 0$.

- Spot price of the stock at expiration date $S_T < K$

  In this case, the buyer has the right to buy the stock at the strike price K which is higher than the spot price S_T. Then, the investor will not use his right to buy the stock at the strike price K and sell it at the spot price S_T. The payoff of the transaction is 0.

We can summarize the situation in what is called a payoff table, hereunder in *Table 1*.

Outcome	$\boldsymbol{S_T > K}$	$\boldsymbol{S_T < K}$
Payoff	$S_T - K$	0

Table 1 - Payoff Table for a European Call Option

More generally, the payoff function of a long position for a European call option is:

$$\max(S_T - K, 0) \tag{1}$$

With a similar reasoning, we can compute the payoff of the short position for a call option, and the equivalent positions on European put options. *Table 2* provides the payoff functions for the different positions on European options. *Figure 1* provides the graphical representation of those functions, by representing the payoff as a function of the stock price at expiration of the option. The change in slope occurs when the stock price at expiration is exactly equal to the strike price K.

	Long Position	**Short Position**
European Call Option	$\max(S_T - K, 0)$	$-\max(S_T - K, 0)$
European Put Option	$\max(K - S_T, 0)$	$-\max(K - S_T, 0)$

Table 2 - Payoff functions for the different positions on European Options

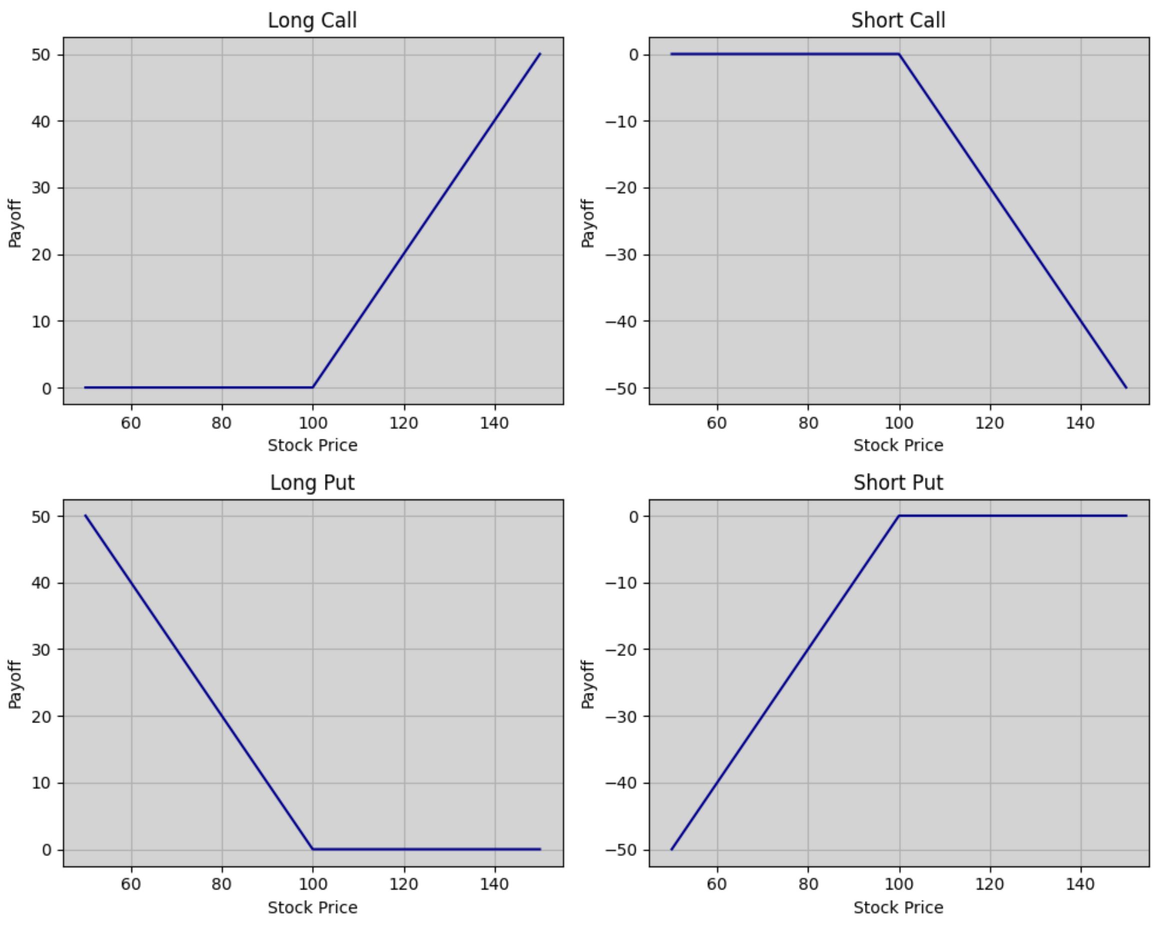

Figure 1 - Graphical representation of the payoff functions for the different possible positions on European Options

2.1.4. Payoff of American Options

American options are similar to European options. What distinguish them from European options is that the holder of an American option can exercises his right on several exercise dates $t \in [0,T] \coloneqq \{0, \delta t, 2\delta t, 3\delta t, \dots, T\}$. For each of those dates wan be computed the option's payoff in the same way it has been done for European options. For a long position on an American call option, the payoff is given by:

$$\forall k \in \left[0, \frac{T}{\delta t}\right], \varphi_{k.\delta t} = \max\left(S_{k.\delta t} - K, continuation_{Payoff}(C')\right) \tag{2}$$

Where φ is the payoff function of the American call option. There are as many payoffs to be computed as there are possible exercise dates. We will consider that the possible exercise dates are all the dates between the emission of the option and its expiration. We cannot replicate a graphical representation of an American option's payoff as we have done it for the European option's one. In *Table 3* are provided the different payoff functions for the different positions on American options.

$\forall \boldsymbol{k} \in \left[\boldsymbol{0}, \frac{\boldsymbol{T}}{\boldsymbol{\delta t}}\right]$	**Long Position**	**Short Position**
American Call Option	$\varphi_{k.\delta t} = \max\left(S_{k.\delta t} - K, C'\right)$	$\varphi_{k.\delta t} = -\max\left(S_{k.\delta t} - K, C'\right)$
American Put Option	$\varphi_{k.\delta t} = \max\left(K - S_{k.\delta t}, C'\right)$	$\varphi_{k.\delta t} = -\max\left(K - S_{k.\delta t}, C'\right)$

Table 3 - Intermediate payoff functions for the different positions on American Options

We will see that the determination of the price of American options is a much more difficult task than it is for European options.

2.1.5. Payoff of Asian Options

Asian options differ from European options through the way the payoff is calculated. Indeed, if an Asian option has a unique expiration date as European options have, the payoff function is now depending on the overall evolution of the underlying's price over the lifetime of the option. There is a notion of average intrinsically linked to Asian Options.

There exist two main types of defining the payoff of an Asian option. We will refer to Asian-In and Asian-out. However, there is no real consensus on what Asian-in and Asian-out refer to.

For some people, the attributes "in" and "out" refer to what extent is computed the average of the underlying price. The attribute "in" would refer to options whose settlement price is computed as the average price of the underlying stock over the lifetime of the option. The attribute "out" would refer to options whose strike price is computed as the average price of the underlying stock over the lifetime of the option.

For others, attributes "in" and "out" refers to the way the average is computed. In this case, attribute "in" refers to computing the average of the underlying stock price at different dates that are uniformly distributed. The attribute "out" in this case related to the computation of the average of the underlying stock price at different dates that are defined within a period generally near the expiration date.

In this paper, we will only work with Asian options with a settlement price computed as the average of the underlying stock's price computed at different dates that are uniformly distributed. Also, because there are two ways to compute the average (arithmetic and geometric), we will focus on arithmetic Asian options since they are commonly used. Moreover, there is no need to use Monte Carlo simulation for geometric Asian Option since there exist analytical methods to price them precisely.

Finally, *Table 4* provides the payoff functions for the different positions on Asian options.

	Long Position	**Short Position**
Asian Call Option	$\max(\overline{S} - K, 0)$	$-\max(\overline{S} - K, 0)$
Asian Put Option	$\max(K - \overline{S}, 0)$	$-\max(K - \overline{S}, 0)$

Table 4 - Payoff functions for the different positions on Arithmetic Asian Options

$where\ \bar{S} = \frac{1}{T}\sum_{i=1}^{T} S_i\ with\ S_i\ the\ underlying\ stock's\ price\ on\ date\ i.$

2.1.6. Payoff of Bermudan Options

Bermudan options have been called from the Bermudan Islands which are located is the Atlantic Ocean, between America and Europe. Indeed, Bermudan options are in between American options and European options since the expiration are multiple alike for American options, but the exercise dates' set do not correspond to all the date beween the emission of the option and its expiration. Therefore, the payoff functions are multiple for each different positions on Bermudan options and are given in *Table 5*.

$\forall \boldsymbol{i} \in \boldsymbol{E}$	**Long Position**	**Short Position**
Bermudan Call Option	$\varphi_i = \max\,(S_i - K, 0)$	$\varphi_i = -\max\,(S_i - K, 0)$
Bermudan Put Option	$\varphi_i = \max\,(K - S_i, 0)$	$\varphi_i = -\max\,(K - S_i, 0)$

Table 5 - Intermediate payoff functions for the different positions on Bermudan Options

Where E correspond to the set of the possible exercise dates.

2.1.7. Payoff of Barrier Options

Barrier options are popular among retail investor as they provide either more protection or more leverage. The main characteristic of Barrier option is that the payoff depends upon whether the stock achieves specific predetermined barrier level at or before the option's expiration.

The Barrier is a level of price for the underlying asset. If the barrier is crossed, there are two possibilities. Either the option becomes worthless, in this case, we talk about Knock-Out Barrier Options (KO), either the option is activated, and in this case, we talk about Knock-In Barrier Options (KI). Finally, the barrier can be set to a higher or a lower level than the initial underlying stock's price. If the barrier is set higher, it will be called an Up Barrier Option, and if the barrier is set to a lower level, it will be called a Down Barrier Option.

To sum up, there are 4 possible combinations possible.

- Up-and-Out Barrier Options (UO)
- Up-and-In Barrier Options (UI)
- Down-and-Out Barrier Options (DO)
- Down-and-In Barrier Options (DI)

And as for each of the 4 Barrier Options there are 4 possible positions, Long Call, Short Call, Long Put and Short Put, we will finally deal with 16 payoff functions. Those payoff functions are given in *Table 6*.

	Long Position	**Short Position**
UO Call Option	$max(S_T - K, 0).1_{\{max_{t\in[0,T]}S(t)<B\}}$	$-max(S_T - K, 0).1_{\{max_{t\in[0,T]}S(t)<B\}}$
UO Put Option	$max(K - S_T, 0).1_{\{max_{t\in[0,T]}S(t)<B\}}$	$-max(K - S_T, 0).1_{\{max_{t\in[0,T]}S(t)<B\}}$
UI Call Option	$max(S_T - K, 0).1_{\{max_{t\in[0,T]}S(t)\geq B\}}$	$-max(S_T - K, 0).1_{\{max_{t\in[0,T]}S(t)\geq B\}}$
UI Put Option	$max(K - S_T, 0).1_{\{max_{t\in[0,T]}S(t)\geq B\}}$	$-max(K - S_T, 0).1_{\{max_{t\in[0,T]}S(t)\geq B\}}$
DO Call Option	$max(S_T - K, 0).1_{\{min_{t\in[0,T]}S(t)>B\}}$	$-max(S_T - K, 0).1_{\{min_{t\in[0,T]}S(t)>B\}}$
DO Put Option	$max(K - S_T, 0).1_{\{min_{t\in[0,T]}S(t)>B\}}$	$-max(K - S_T, 0).1_{\{min_{t\in[0,T]}S(t)>B\}}$
DI Call Option	$max(S_T - K, 0).1_{\{min_{t\in[0,T]}S(t)\leq B\}}$	$-max(S_T - K, 0).1_{\{min_{t\in[0,T]}S(t)\leq B\}}$
DI Put Option	$max(K - S_T, 0).1_{\{min_{t\in[0,T]}S(t)\leq B\}}$	$-max(K - S_T, 0).1_{\{min_{t\in[0,T]}S(t)\leq B\}}$

Table 6 - Payoff functions for the different positions on Barrier Options

2.2. Pricing Options with the Black-Scholes-Merton model

The Black-Scholes-Merton (BSM) model is one of the most widely used models for pricing European options. It was developed by Fischer Black, Myron Scholes and Robert Merton in the early 1970s. We will use it as a reference to compare the results we obtain with Monte Carlo methods.

2.2.1. Assumptions

The model is built upon the following assumptions:

- The stock price follows a geometric Brownian motion.
- The short selling of securities with full use of proceeds is permitted.
- There are no transactions costs or taxes. All securities are perfectly divisible.
- There are no dividends during the life of the derivative.
- There are no riskless arbitrage opportunities.

- Security trading is continuous.
- The risk-free rate of interest is constant and the same for all maturities.

2.2.2. Geometric Brownian motion

The model of stock price behavior is a continuous-time stochastic process known as geometric Brownian motion. Its discrete-time version is the following:

$$\Delta S = \mu S \Delta t + \sigma S \varepsilon \sqrt{\Delta t} \tag{3}$$

Where:

- ΔS is the change in the stock price S in a small-time interval Δt .
- ε follows a standard normal distribution (i.e. a normal distribution with a mean of zero and standard deviation of one).
- μ is the expected rate of return per unit of time from the stock.
- σ is the volatility of the stock price.

The left-hand side of equation is the return provided by the stock in a short period of time (Δt). The term $\mu \Delta t$ is the expected value of this return, and the term $\sigma \varepsilon \sqrt{\Delta t}$ is the stochastic component of the return.

2.2.3. Derivation of the BSM differential equation

The concept underlying this model is to create a riskless portfolio consisting of a given quantity of the stock and the option. Because under no-arbitrage opportunities, it will enable to establish an equation where the only unknown variable is the option price.

So, consider the portfolio Π consisting in one short position in the option and n long positions in the stock. As the stock price and the option price are both affected by the same underlying source of uncertainty: the stock price movements, let $f = f(S, t)$ be the price of the option (which is a function of the stock price and the time). Thus, the value of the portfolio is

$$\Pi = -f + nS \tag{4}$$

and its change in value in the time interval Δt is

$$\Delta \Pi = -\Delta f + n \Delta S \tag{5}$$

We already know the process of the stock price, so let's use Itô's lemma to derive the process followed by f .

Itô's Lemma:

Suppose that x follows the stochastic process

$$\Delta x = a(x,t)\Delta t + b(x,t)\Delta t \tag{6}$$

where a *and* b *are functions of* x *and* t. *The variable* x *has a drift rate of* a *and a variance rate of* b^2. *Itô's lemma shows that a function* G *of* x *and* t *follows the process*

$$\Delta G = \left(\frac{\partial G}{\partial x}a + \frac{\partial G}{\partial t} + \frac{1}{2}\frac{\partial^2 G}{\partial x^2}b^2\right)\Delta t + \frac{\partial G}{\partial x}b\varepsilon\sqrt{\Delta t} \tag{7}$$

Thus, applying Itô's lemma to the stock price process (GBM), (i.e. with $x = S$, $a(x,t) = \mu S$ and $b(x,t) = \sigma S$), we find that the stochastic process for f is

$$\Delta f = \left(\frac{\partial f}{\partial S}\mu S + \frac{\partial f}{\partial t} + \frac{1}{2}\frac{\partial^2 f}{\partial S^2}\sigma^2 S^2\right)\Delta t + \frac{\partial f}{\partial S}\sigma S\varepsilon\sqrt{\Delta t} \tag{8}$$

Therefore, by replacing the stochastic process for f and S we obtain

$$\Delta\Pi = -\left[\left(\frac{\partial f}{\partial S}\mu S + \frac{\partial f}{\partial t} + \frac{1}{2}\frac{\partial^2 f}{\partial S^2}\sigma^2 S^2\right)\Delta t + \frac{\partial f}{\partial S}\sigma S\varepsilon\sqrt{\Delta t}\right] + n\left(\mu S\Delta t + \sigma S\varepsilon\sqrt{\Delta t}\right) \tag{9}$$

So that

$$\Delta\Pi = -\left[\left(\frac{\partial f}{\partial S} - n\right)\mu S + \frac{\partial f}{\partial t} + \frac{1}{2}\frac{\partial^2 f}{\partial S^2}\sigma^2 S^2\right]\Delta t - \left(\frac{\partial f}{\partial S} - n\right)\sigma S\varepsilon\sqrt{\Delta t} \tag{10}$$

As the right-hand member is the only indeterministic member, we can eliminate the uncertainty generated by the random variable ε by setting

$$n = \frac{\partial f}{\partial S} \tag{11}$$

So we get

$$\Delta\Pi = -\left(\frac{\partial f}{\partial t} + \frac{1}{2}\frac{\partial^2 f}{\partial S^2}\sigma^2 S^2\right)\Delta \mathrm{t} \tag{12}$$

Since the portfolio is riskless, in the absence of arbitrage opportunities, the return from the portfolio must be the risk-free interest rate r over any short period of time Δt. It follows that

$$\Delta\Pi = r\Pi\Delta t \tag{13}$$

By substituting from equations (4) and (12) into (13), we obtain

$$-\left(\frac{\partial f}{\partial t}+\frac{1}{2}\frac{\partial^2 f}{\partial S^2}\sigma^2 S^2\right)\Delta t = r\left(-f+\frac{\partial f}{\partial S}S\right)\Delta t \tag{14}$$

So that

$$\frac{\partial f}{\partial t}+rS\frac{\partial f}{\partial S}+\frac{1}{2}\frac{\partial^2 f}{\partial S^2}\sigma^2 S^2 = rf \tag{15}$$

This is the Black–Scholes–Merton differential equation.

2.2.4. Black-Scholes-Merton pricing formulas

The Black-Scholes-Merton differential equation has many solutions, corresponding to all the different derivatives that can be defined with the stock price as the underlying variable. The derivative being valued is determined by the boundary conditions of the differential equation. In the case of European options, the boundaries conditions are $f = \max(S-K,\ 0)\ when\, t = T,$ for a call option $f = \max(K-S,\ 0)\ when\, t = T,$ for a put option.

Finally, the solutions of this differential equation give us the Black-Scholes-Merton formulas for the prices of European call and put options:

$$c = SN(d_1) - Ke^{-rT}N(d_2) \tag{16}$$

and

$$p = Ke^{-rT}N(-d_2) - SN(-d_1) \tag{17}$$

Where:

- c and p are the European call and put price.
- $d_1 = \frac{\ln\left(\frac{S}{K}\right)+\left(r+\frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}},\ d_2 = \frac{\ln\left(\frac{S}{K}\right)+\left(r-\frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}}$
- S stands for the current value of the underlying.
- K stands for the strike price.
- T is the time at maturity.
- $N(x)$ is the standard normal cumulative probability distribution. It is the probability that a variable with a standard normal distribution, will be less than x. As illustrated in *Figure 2*.

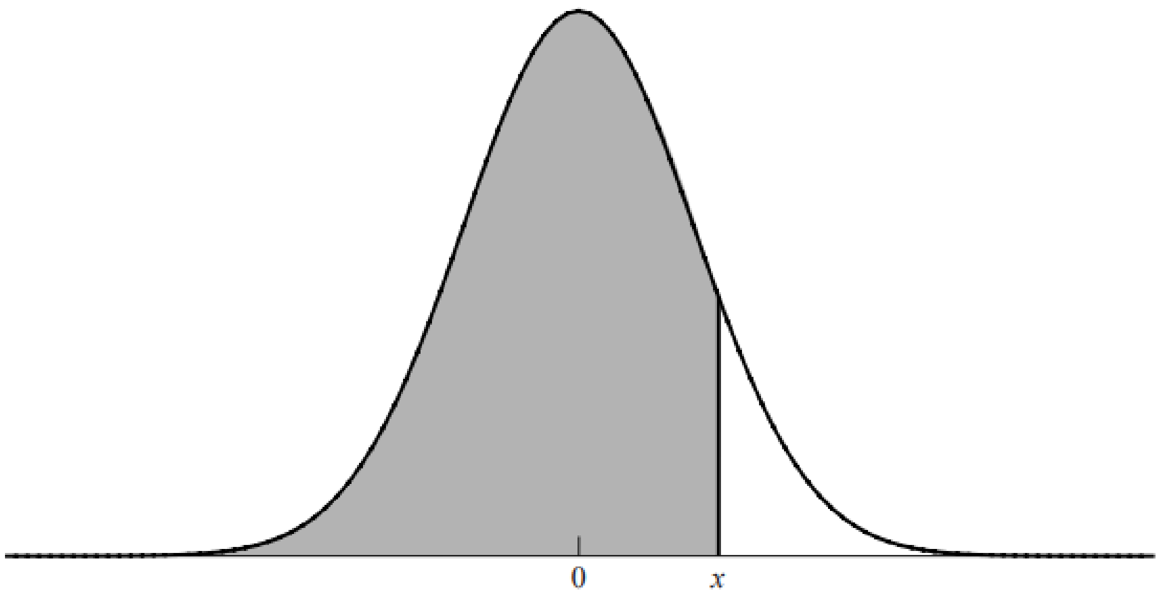

Figure 2 - Graphical representation of the density function of a normal law. The shaded area represents N(x).

2.3. Pricing options using Monte Carlo

2.3.1. Theory around Monte Carlo

The Monte Carlo method is a stochastic method, based on the simulation of random variables, for calculating an approximate value of a quantity E . The first step is to put this quantity in the form of an expected value $E = E[X]$ with $X \in L^1$. If we know how to simulate variables X_1, X_2, . . . i.i.d. (independent and identically distributed) with the same distribution as X . Then the Law of Large Numbers tells us that we can approximate E by the expression (18) where N is "large".

$$E = \frac{X_1 + X_2 + \cdots + X_N}{N} \tag{18}$$

We can also express the quantity as follow: $E = E[g(X)]$, because if X is a random variable, then $g(X)$ is a random variable, provided that g is Borelian.

Strong Law of Large Numbers:

Let $X_1, X_2, \ldots$ be a sequence of i.i.d. random variables. Assume that $E[|X_1|] < +\infty$ and set $\mu = E[X_1]$. Then

$$P\left(\lim_{N\to+\infty}\frac{X_1+X_2+\cdots+X_N}{N}=\mu\right) = 1 \tag{19}$$

Thus, this theorem shows us that the Monte Carlo approximation is valid for i.i.d. random variables under the simple existence assumption of $E = E[X]$.

2.3.2. Pricing European, Asian and Barrier Options with Monte Carlo

To apply the Monte Carlo method for options pricing, we first need to express the option price as an expected value. For that, we will use the probabilistic interpretation of the Cox-Ross-Rubinstein binomial model.

2.3.2.1 Cox-Ross-Rubinstein model

The Cox-Ross-Rubinstein model, also known as the binomial options pricing model, is a widely used method for valuing options. It was developed by John Cox, Stephen Ross, and Mark Rubinstein in 1979. This model provides a discrete-time framework for option valuation, which contrasts with the continuous-time framework of the Black-Scholes model.

Assumptions:

- One risky and one risk-free asset trade
- Trading is opened at discrete times
- The duration between two dates is of length δt
- Over one period, two outcomes are possible for the stock price: up or down move
- There are no transaction costs
- Volatility remains constant over time

We consider a stock whose price is S_0 and an option on the stock whose current price is f_0. We suppose that the option lasts for time T and that during the life of the option the stock price can either move up from S_0 to a new level, S_0u, where $u > 1$, or down from S_0 to a new level, S_0d, where $d < 1$. If the stock price moves up, we note the payoff from the option f_u, and f_d if the stock price moves down.

Now, we set up a replication portfolio composed of Δ shares of the underlying asset and a cash amount B remunerated at the risk-free rate r which value will be equal to the payoffs of the option.

So, we must find Δ and B such that the portfolio value is equal to the payoffs of the option at time T in all states of the world:

$$\Delta S_0 u + Be^{rT} = f_u \tag{20}$$

and

$$\Delta S_0 d + Be^{rt} = f_d \tag{21}$$

The solution is:

$$\Delta = \frac{f_u - f_d}{S(u-d)} \tag{22}$$

and

$$B = -e^{-rT}\frac{df_u - uf_d}{u-d} \tag{23}$$

Thus, under no arbitrage opportunities the portfolio and the option must have the same price:

$$f_0 = \Delta S + B = \frac{(1 - e^{-rT}d)f_u + (ue^{-rT} - 1)f_d}{u-d} \tag{24}$$

Probabilistic interpretation:

We can rewrite the equation (24) as

$$f_0 = e^{-rT}[pf_u + (1-p)f_d] \tag{25}$$

with $p = \frac{e^{-rT} - d}{u-d}$.

Finally, interpreting p as a measure of probability that the price moves up, the value of the option today is its expected payoff discounted at the risk-free rate:

$$f_0 = e^{-rT}E[f_T] = E[e^{-rT}f_T] \tag{26}$$

2.3.2.2 Application of the Monte Carlo method

We can now apply the Monte Carlo method because we are in the situation where the quantity E (which is the option price f_0) that we want to estimate is of the form $E = E[g(X)]$ with:

- $g(x) = e^{-rT}f_T(x)$ where $f_T(x)$ is the payoff function of the option.
- $X = S_T$ (Spot price of the underlying asset at time T).

And we know how to simulate $S_{T,1}$, $S_{T,2}$, ... i.i.d. with the same distribution as S_T (following the GBM process).

Thus, we obtain the following formula for the price of European, Asian and barrier options:

$$f_0 = \frac{1}{N}\sum_{i=1}^{N}\left[e^{-rT} f_{T,i}\left(S_{T,i}\right)\right] \tag{27}$$

Where:

- f_0 is the option price.
- N is the number of simulations of the stock price trajectory.
- $S_{T,i}$ stands for the stock price at time T of the i -th trajectory simulated, $i \in \{1, \dots, N\}$.
- $f_{T,i}$ stands for the payoff (at time T) of the i -th trajectory simulated.
- T is the time at maturity.
- r is the risk-free rate.

So, by replacing the payoff function according to the type of option we obtain the following results in Table 7 and Table 8.

Option type	**Price of a call**
European	$\frac{1}{N}\sum_{i=1}^{N}\left[e^{-rT}\max\left(S_{T,i} - K, 0\right)\right]$
Asian	$\frac{1}{N}\sum_{i=1}^{N}\left[e^{-rT}\max\left(\overline{S_i} - K, 0\right)\right]$
UO Barrier	$\frac{1}{N}\sum_{i=1}^{N}\left[e^{-rT}\max\left(S_{T,i} - K, 0\right).1_{\left\{\left(\max_{t\in[0,T]} S_{t,i}\right)<B\right\}}\right]$
UI Barrier	$\frac{1}{N}\sum_{i=1}^{N}\left[e^{-rT}\max\left(S_{T,i} - K, 0\right).1_{\left\{\left(\max_{t\in[0,T]} S_{t,i}\right)\geq B\right\}}\right]$
DO Barrier	$\frac{1}{N}\sum_{i=1}^{N}\left[e^{-rT}\max\left(S_{T,i} - K, 0\right).1_{\left\{\left(\min_{t\in[0,T]} S_{t,i}\right)>B\right\}}\right]$
DI Barrier	$\frac{1}{N}\sum_{i=1}^{N}\left[e^{-rT}\max\left(S_{T,i} - K, 0\right).1_{\left\{\left(\min_{t\in[0,T]} S_{t,i}\right)\leq B\right\}}\right]$

Table 7 - Pricing functions using Monte Carlo for European, Asian et Barrier Calls

Option type	Price of a put
European	$\frac{1}{N}\sum_{i=1}^{N}\left[e^{-rT}\max\left(K-S_{T,i},0\right)\right]$
Asian	$\frac{1}{N}\sum_{i=1}^{N}\left[e^{-rT}\max\left(\overline{S_i}-K,0\right)\right]$
UO Barrier	$\frac{1}{N}\sum_{i=1}^{N}\left[e^{-rT}\max\left(K-S_{T,i},0\right).1_{\left\{\left(\max\limits_{t\in[0,T]} S_{t,i}\right)<B\right\}}\right]$
UI Barrier	$\frac{1}{N}\sum_{i=1}^{N}\left[e^{-rT}\max\left(K-S_{T,i},0\right).1_{\left\{\left(\max\limits_{t\in[0,T]} S_{t,i}\right)\geq B\right\}}\right]$
DO Barrier	$\frac{1}{N}\sum_{i=1}^{N}\left[e^{-rT}\max\left(K-S_{T,i},0\right).1_{\left\{\left(\min\limits_{t\in[0,T]} S_{t,i}\right)>B\right\}}\right]$
DI Barrier	$\frac{1}{N}\sum_{i=1}^{N}\left[e^{-rT}\max\left(K-S_{T,i},0\right).1_{\left\{\left(\min\limits_{t\in[0,T]} S_{t,i}\right)\leq B\right\}}\right]$

Table 8 - Pricing functions using Monte Carlo for European, Asian et Barrier Puts

2.3.2. Pricing American Options with Monte Carlo

To price American option, the process is quite similar but requires an additional computation that is a regression step. To price American option, we decided to use the Longstaff Schwartz Algorithm.

This algorithm is a Monte Carlo simulation-based approach using least squares regression to estimate the continuation value of holding the option. The first step is to simulate paths of evolution of the stock price in the same ways it has been described previously. The second step, often named the backwardation induction, consists of deciding whether it is optimal to exercise the option or to continue holding it at each step. And at each step, the algorithm estimates the continuation value, which is the expected value of holding the option and potentially exercising it later. This is done using least squares regression.

By doing this, the algorithm constructs an optimal exercise strategy based on the simulated paths. This strategy tells us when it is optimal to exercise the option at each point in time.

Finally, as it is a Monte Carlo, we will average discounted payoff and get the estimated price for the option.

2.4. Variance reduction

2.4.1. Definition

We have previously seen that if we approximate a quantity $E[g(X)]$ using the Monte Carlo method, i.e., through an empirical average $\frac{g(X_1)+g(X_2)+\cdots+g(X_N)}{N}$, where $X_1, X_2, \dots, X_N$ are i.i.d. variables with the same distribution as X, then the error $\varepsilon = \left|E[g(X)] - \frac{g(X_1)+g(X_2)+\cdots+g(X_N)}{N}\right|$ is of order $\frac{\sigma}{\sqrt{N}}$, where $\sigma^2 = V[g(X)]$. Here, we will present methods that allow us to express $E[g(X)] = E[Y]$ with $V[Y] \leq V[g(X)]$. These methods are referred to as "variance reduction techniques" and will improve the efficiency of the algorithm. Indeed, if one can lower the variance σ^2, he can have the same error using a smaller number of simulations of X.

2.4.2. Antithetic Variables Monte Carlo

Let suppose that we want to estimate $I = E[X]$. Here, maybe the variance of X is important and will require a high number of simulations to get a proper error ε. So, we want to express X as a function of an integrable function h such that:

$$I = E[h(U)] \tag{28}$$

where U is a uniformly distributed random variable.

On top of that, $1 - U$ is identically distributed as U. Introduction the notation $U = (U_1, U_2, \dots, U_N)$.We get the following result:

$$I = E[h(U)] = E\left[\frac{h\big((1,1,\dots,1) - U\big) + h(U)}{2}\right] \tag{29}$$

Writing $X = h\big((1,1,\dots,1) - U\big)$ and $Y = h(U)$, Cauchy-Schwarz ensures that:

$$V\left[\frac{X+Y}{2}\right] = \frac{1}{4}V[X+Y] = \frac{1}{4}\big(V[X] + V[Y] + 2Cov(X,Y)\big) \leq \frac{1}{4}\left(V[X] + V[Y] + 2\sqrt{V[X]V[Y]}\right) \tag{30}$$

$$V\left[\frac{X+Y}{2}\right] \leq \frac{1}{4}\left(V[X] + V[X] + 2\sqrt{V[X]V[X]}\right) = V[X] \tag{31}$$

Using h we successfully reduced the variance and will be able to lower the number of simulations. Practically, we will generate N uniformly distributed random variables X_i and we will approximate I with:

$$I \approx \frac{1}{N}\sum_{i=1}^{N} \frac{h\big((1,1,\dots,1) - X_i\big) + h(X_i)}{2} \tag{32}$$

2.4.3. Controlling Variables Monte Carlo

Here we remain in the previous framework where the goal is to approximate $E[g(X)]$. This second approach consists of finding a function h which is close to g such as $V[g(X)] - V[h(X)]$ is smaller than $V[g(X)]$ and such as $E[h(X)]$ can be efficiently computed.

Then we make the following approximation:

$$E[g(X)] \;\approx\; \frac{\big(g(X_1) - h(X_1)\big) + \cdots + \big(g(X_N) - h(X_N)\big)}{N} + E[(h(X)] \tag{33}$$

In this case, the error can be approximate:

$$\varepsilon \approx\; E[g(X)] - \left(\frac{\big(g(X_1) - h(X_1)\big) + \cdots + \big(g(X_N) - h(X_N)\big)}{N} + E[(h(X)]\right) \tag{34}$$

$$\varepsilon \approx\; E[g(X) - h(X)] - \left(\frac{\big(g(X_1) - h(X_1)\big) + \cdots + \big(g(X_N) - h(X_N)\big)}{N}\right) \tag{35}$$

And this approximation implies that $\varepsilon \approx \sqrt{\frac{V[g(X)-h(X)]}{N}} \leq \frac{\sigma}{\sqrt{N}}$

In order to implement this method for option pricing, we must find a function h which is close from $g(x) = e^{-rT} f_T(x)$. To do so, we might use a replication of the portfolio in the same way it has been done in the Black-Scholes-Merton model. This is called a delta control variate, relying on delta hedging strategies.

If we consider a perfect hedge, we get the following expression:

$$f_0 e^{rT} \;=\; f_T + \sum_{i=1}^{N} \left(\frac{\delta f_i}{\delta S} - \frac{\delta f_{i-1}}{\delta S}\right) S_T e^{r(T-i)} \tag{36}$$

$$f_0 e^{rT} \;=\; f_T - \sum_{i=0}^{N-1} \frac{\delta f_i}{\delta S}\left(S_{i+1} - S_i e^{r\delta t}\right) e^{r(T-(i+1))} \tag{37}$$

We then get a new expression for the value of the option at date $t = 0$.

$$f_0 = \left[f_T - \sum_{i=0}^{N-1} \frac{\delta f_i}{\delta S}\left(S_{i+1} - S_i e^{r\delta t}\right) e^{r(T-(i+1))} \right] e^{-rT} \tag{38}$$

This first solution will work for option whose delta can be analytically determined which is not the case for Asian and American options. For Asian option, we will use Vorst Control Variable. This method consists in using the analytical price formula of the geometric Asian option (which is perfectly known in the Black-Scholes-Merton model) to reduce the variance of the Monte Carlo method.

The price of a geometric Asian option is given by the following expression:

$$P_{geometric} = \exp(-rT)\left[S_0 \exp\left(\left(b - \frac{1}{2}\sigma^2\right)\frac{T}{2}\right)\varphi(d_1) - K\varphi(d_2)\right] \tag{39}$$

In the equality (39), φ represent the cumulative density function of the normal law. Then we perform a Monte Carlo on an arithmetic Asian option and a geometric Asian option, and we control it with the analytical price of the geometric Asian option.

For controlling on American option, we used the European Option price and perform a similar method as the previous one.

2.4.4. Importance Sampling Monte Carlo

Here again, we want to approximate $E[g(X)]$, where X are i.i.d. with the density function f. The approach of the importance sampling method is to find an alternative density function called the importance function and denoted $\tilde{f}$ to be able to write:

$$E[g(X)] = \int g(x)f(x)dx = \int \frac{g(x)f(x)\tilde{f}(x)}{\tilde{f}(x)}dx = \int \frac{g(x)f(x)}{\tilde{f}(x)}\tilde{f}(x)dx \tag{40}$$

Now we can rewrite $Y = \frac{g(x)f(x)}{\tilde{f}(x)}$ such as Y has a density $\tilde{f}$:

$$E[Y] = \int Y(x)\tilde{f}(x)dx = \int \frac{g(x)f(x)}{\tilde{f}(x)}\tilde{f}(x)dx = E[g(X)] \tag{41}$$

Then, we can perform the following approximation:

$$E[g(X)] \approx \frac{g(X_1) + g(X_2) + \cdots + g(X_N)}{N} \approx \frac{1}{N}\left(\frac{g(Y_1)f(Y_1)}{\tilde{f}(Y_1)} + \cdots + \frac{g(Y_N)f(Y_N)}{\tilde{f}(Y_N)}\right) \tag{42}$$

And if we can verify $V\left[\frac{g(Y)f(Y)}{\tilde{f}(Y)}\right] \leq V[g(X)]$, then the method is interesting and improve the efficiency of the Monte Carlo algorithm.

3. NUMERICAL ANALYSIS

3.1. Parameters of the simulation

Throughout the different simulation, we used fixed parameters that we chose arbitrarily. The first choice we had to make was to determine the asset we would use for the simulations. We decided to go for Total Energy stock ("TTE.PA"). We fixed the risk-free rate "R" at 3,9%, the initial stock price "S_0" as the last quoted price of the stock, the strike price "K" equal to S_0 (at-the-money options), the option-type (call or put) and the time to expiration "T" of the option. Finally, the volatility was computed as the standard deviation of the log-returns over the entire life of the stock (from 04/01/1991 to the date the simulations were performed: 10/06/2024). All options were call options, and for the barrier option, we chose a Down-Out option.

3.2. Simulation of the stock price evolution

3.2.1 Prices simulation with Itô's lemma

For simulating the different stock price paths, we have applied Itô's lemma to the Geometric Brownian Motion to derive the process for $\ln(S)$:

We have

$$\frac{\partial \ln(S)}{\partial S} = \frac{1}{S};\ \frac{\partial^2 \ln(S)}{\partial S^2} = -\frac{1}{S^2};\ \frac{\partial \ln(S)}{\partial t} = 0 \tag{43}$$

And we get

$$\partial \ln(S) = \left(\frac{1}{S}\mu S - 0.5\frac{1}{S^2}(\sigma S)^2\right)dt + \frac{1}{S}\sigma S\varepsilon\sqrt{dt} = (\mu - 0.5\sigma^2)dt + \sigma\varepsilon\sqrt{dt} \tag{44}$$

And we know that

$$\partial \ln(S) = \ln(S_{t+\delta t}) - \ln(S_t) = \ln\left(\frac{S_{t+\delta t}}{S_t}\right) \tag{45}$$

So we finally get the following expression (46):

$$\frac{S_{t+\delta t}}{S_t} = e^{(\mu-0.5\sigma^2)dt+\sigma\varepsilon\sqrt{dt}} \Rightarrow S_{t+\delta t} = S_t e^{(\mu-0.5\sigma^2)dt+\sigma\varepsilon\sqrt{dt}} \tag{46}$$

Over a period of length δt , we can draw a random stock price using

$$S_{t+\delta t} = S_t e^{(\mu - 0.5\sigma^2)\delta t + \sigma\varepsilon\sqrt{\delta t}} \tag{47}$$

Where ε is a standard normal draw.

Figure 3 represents 100 simulated paths with 100 steps with the fixed parameters defined in the previous section.

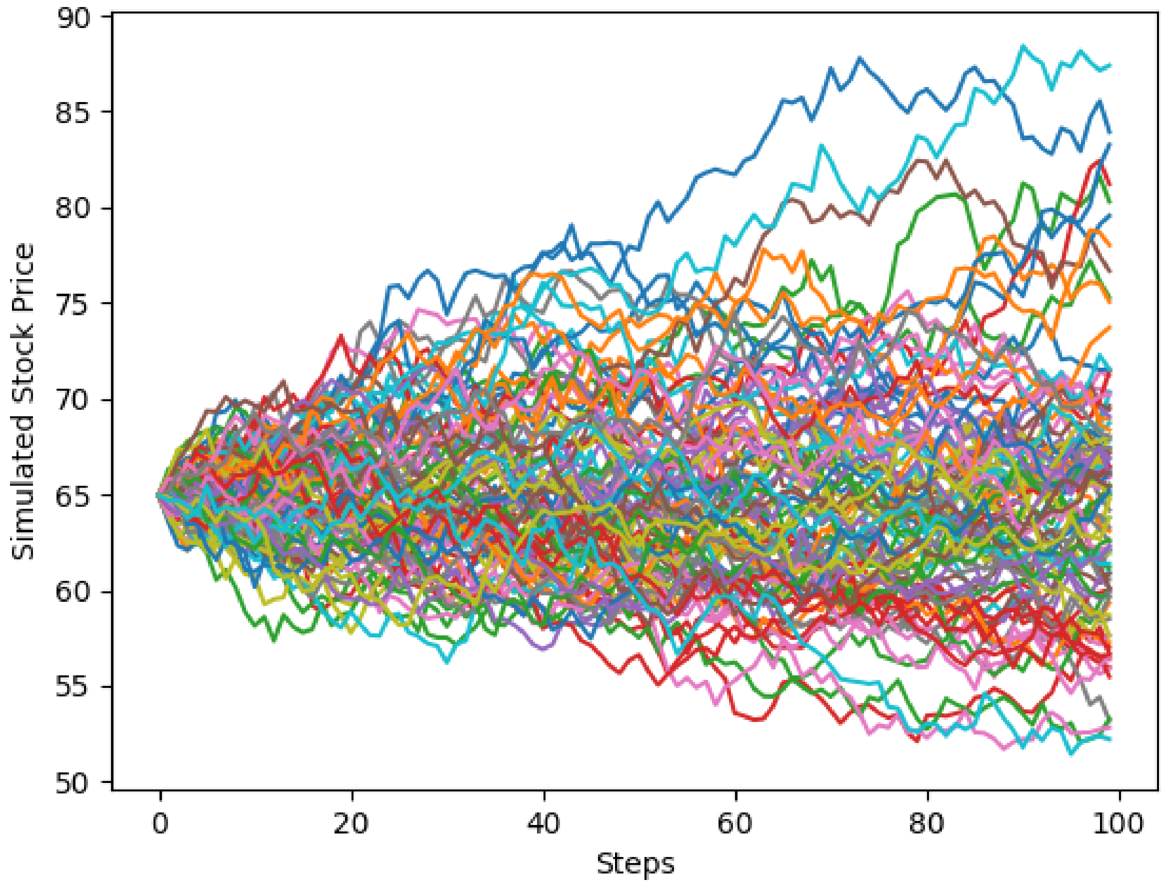

Figure 3 - Simulation of 100 paths using Geometric Brownian motion with 100 steps for each path

3.2.2 Box-Müller algorithm

The random part of the stock price evolution is generated by a random variable which follows a standard normal distribution. To simulate this variable, we used the Box-Müller algorithm, because it is 20% more efficient than the function already available in the NumPy library for simulating a standard normal distribution. This method consists of simulating a standard normal distribution using only uniform distributions on $[0,1]$.

Lemma:

If U, V *follow a uniform distribution on* $[0;1]$ *and are independent, then*

$$(X,Y) = \left(\sqrt{-2\,ln(U)}\,cos(2\pi V),\sqrt{-2\,ln(U)}\,sin(2\pi V)\right) \tag{48}$$

is a pair of independent random variables with the same distribution $N(0,1)$.

Therefore, to simulate a variable $X \sim N(0,1)$, simply simulate independently $U,V \sim U([0,1])$ and set $X = \sqrt{-2\ln(U)}\cos(2\pi V)$.

3.3. Variance Reduction Efficiency

3.3.1. European Options

Since the number of simulations is large, and the computation time long for this reason, we decided first to find the optimal number of steps to perform the simulations. We found it by performing the standard Monte Carlo and estimating the option price.

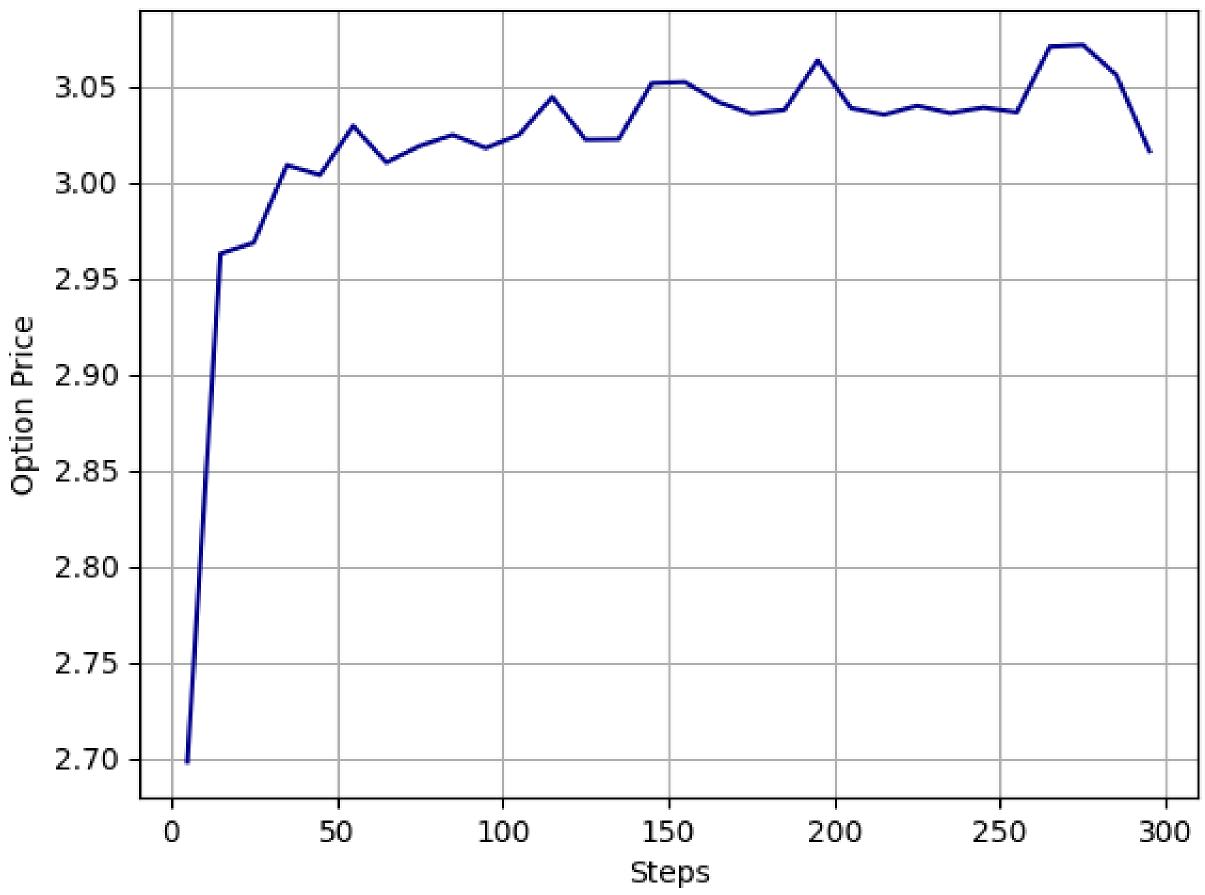

Figure 4 - Evolution of the simulated option's price when the number of steps used to simulate Monte Carlo stock prices is increasing.

Figure 4 clearly shows that there is no significant improvement in the estimation of the option price beyond 50 steps. Therefore, we selected 50 steps for all subsequent computations as it represents the optimal number of steps.

We implemented various algorithms based on the Monte Carlo method, incorporating different variance reduction techniques. Each algorithm provides the option price using a specified number of simulated paths and a given step size (that is now fixed to 50).

The speed of convergence for the different algorithms can be observed. The graphics in *Figure 5* depict the option price obtained for increasing the number of simulated paths, allowing us to compare the convergence speed of each algorithm.

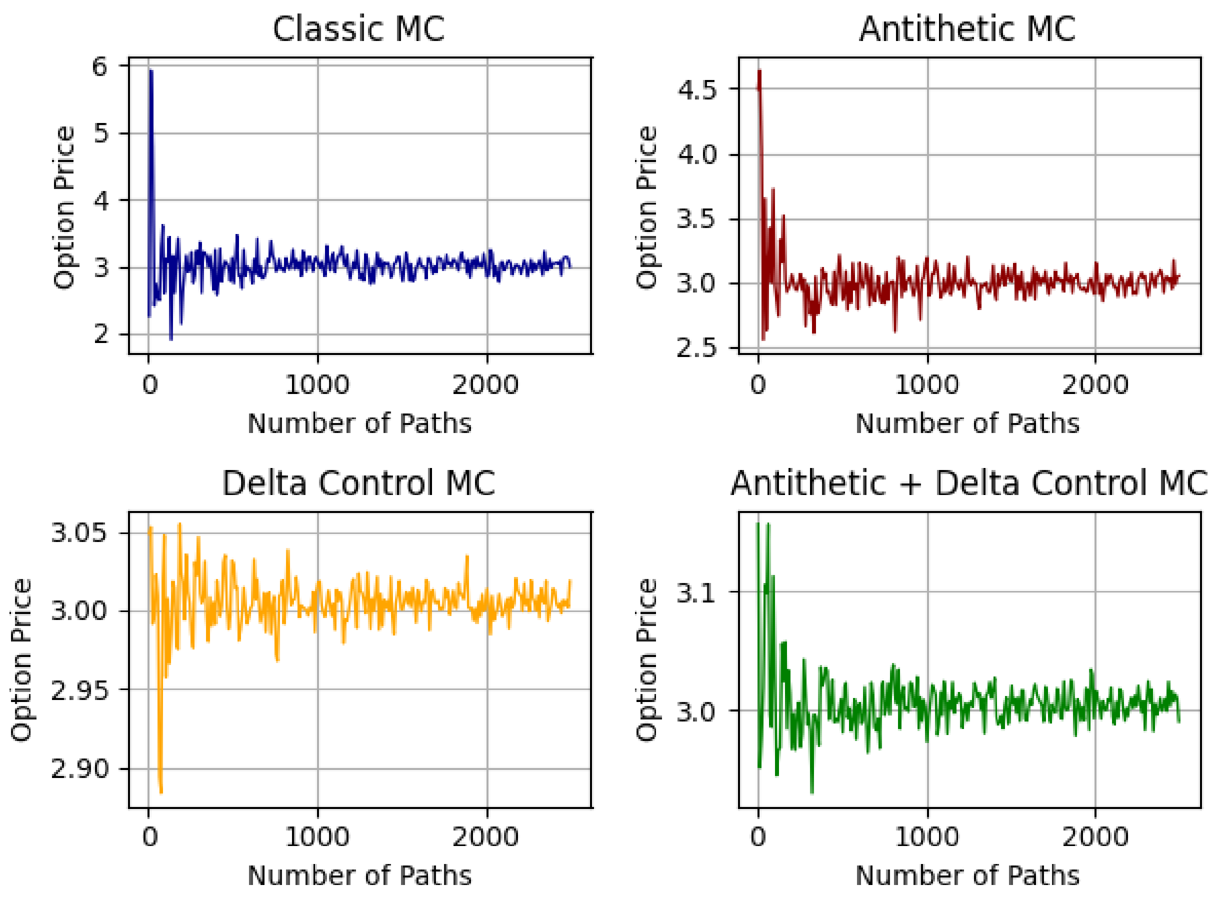

Figure 5 - Graphical representation of the European option's price convergence for different Monte Carlo algorithms when the number of simulated paths increase

On the graphics of *Figure 5*, we can see from the y-axis that the Delta Control Monte Carlo and "Antithetic + Delta Control" Monte Carlo methods appear to converge much quicker than the

Classic Monte Carlo and Antithetic Monte Carlo methods. By setting all the graphics on the same scale this difference in performance will be easier to observe.

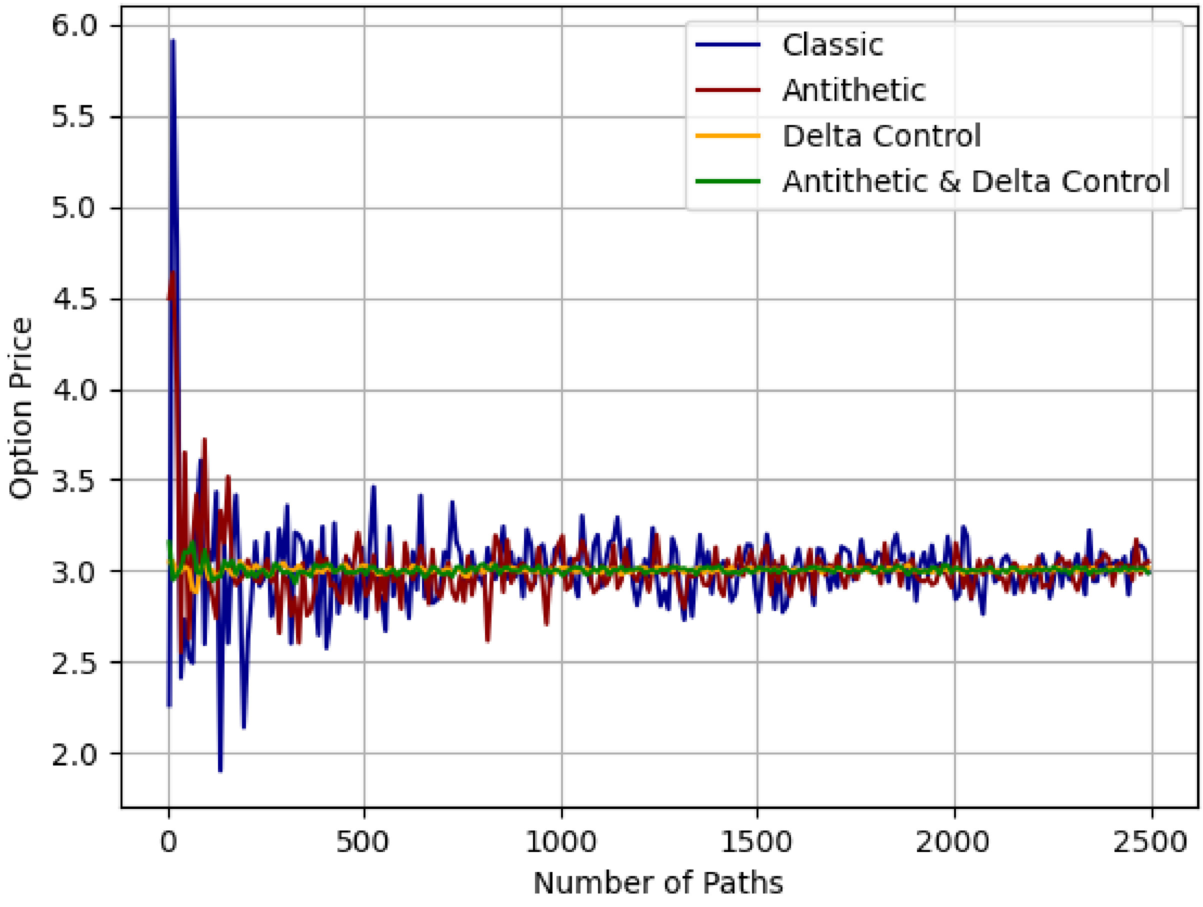

Figure 6 – Convergence speed comparison for different Monte Carlo algorithms on European options

Figure 6 illustrates the difference in efficiency in terms of convergence between the 4 methods discussed previously. However, to determine which of Delta Control Monte Carlo or "Antithetic + Delta Control" Monte Carlo is more effective, we had to quantify this efficiency.

The results of the option price, standard error and the computation time are given in Table 9. We also introduced a Standard Error Efficiency Gain (SEG) as the gain of efficiency related to the decrease in standard error when using a variance reduction method. Moreover, in order to consider the computation time, we defined a performance gain over computation time ratio (P/T) as the SEG over the computation time in second. The higher this ratio, the more efficient the algorithm is. For those results we used many simulations with 100,000 simulated paths. The Black-Scholes price is estimated at 3.064€ as a comparison price.

Method	Option Price (€)	Standard Error	SEG	Computation Time (s)	P/T
Standard MC	3.02122	0.01437	1	25.51241	0.03920
Antithetic Variables	3.01134	0.01066	1.35	17.32638	0.07780
Control Variables	3.0325	0.0011	13.06	25.05866	0.5213
Antithetic + Control	3.02987	0.00155	9.27	17.67816	0.5244

Table 9 - Option price, standard error, standard error gain, computation time and performance gain over computation time ratio obtained for the different Monte Carlo algorithms for European options price simulation (100,000 simulated paths for each algorithm).
Source: Author's calculations.

From *Table 9*, we can see that the Antithetic method is more efficient than the classic Monte Carlo, mainly because of its computation time. However, it is the Control Variable method that is the most efficient because it greatly reduces the standard error. As far as the combination of the two variance reduction methods is concerned, it is ultimately very close to the Control Variable method in terms of performance. In fact, even though it has a larger standard error than the latter, it is the gain in computation speed of the Antithetic method that enables it to match the P/T ratio of the Control Variable method.

3.3.2. Asian Options

For Asian option, we implemented 4 algorithms. The Standard Monte Carlo method, the Antithetic variables Monte Carlo, the Vorst control Monte Carlo (relying on the geometric Asian option price as a control variate) and finally, the combination of Vorst control variable and antithetic variable. Vorst control variable was particularly efficient. This property will be observed both graphically on *Figure 8* and *Figure 9*, and numerically in *Table 10*.

Figure 7 represents the price of an arithmetic Asian call option obtained for different number of simulated paths. We can then observe the speed of convergence of each algorithm. Indeed, we can see that as the number of simulated paths increase, the option price converges to a value around 1,8€. However, since the four graphics in *Figure 7* don't have the same scale, we cannot compare the convergence's speed on this figure.

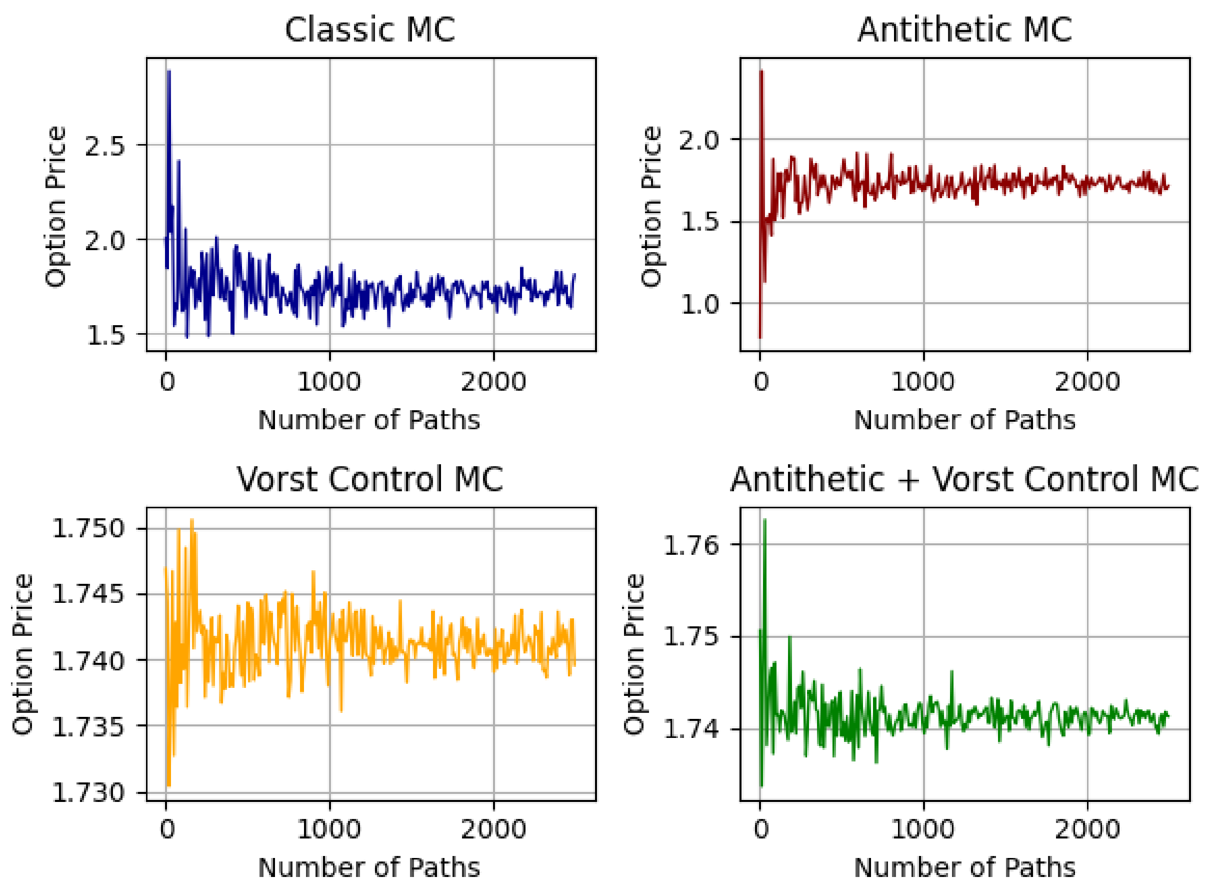

Figure 7 - Graphical representation of the Asian option's price convergence for different Monte Carlo algorithms when the number of simulated paths increase

We can see graphically that all the variance reduction methods clearly speed up the convergence of the Monte Carlo algorithm for Asian options. Moreover, it is evident that some methods allowed faster convergence rates than others. This difference in performance gain is illustrated in *Figure 8*, where the convergence performances of all four methods are depicted on the same graph using a unified scale. This comparative visualization highlights the varying efficiencies of each method, demonstrating that while all variance reduction techniques enhance convergence, some are more effective than others.

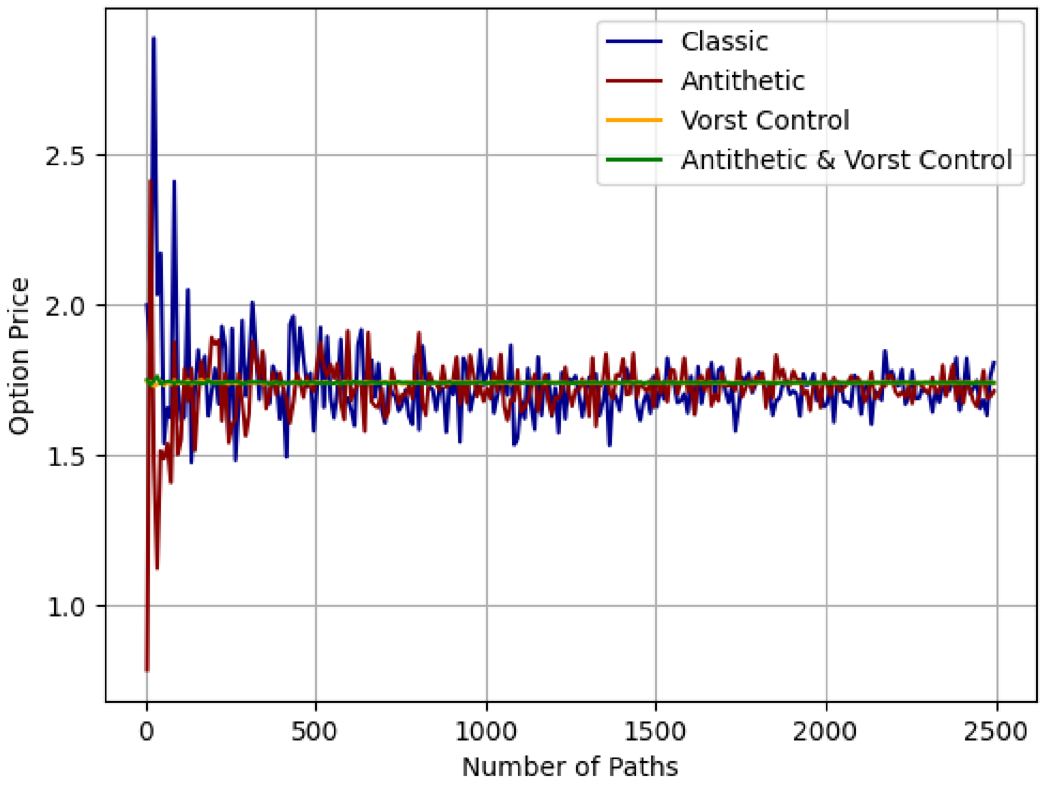

Figure 8 - Convergence speed comparison for different Monte Carlo algorithms on Asian option

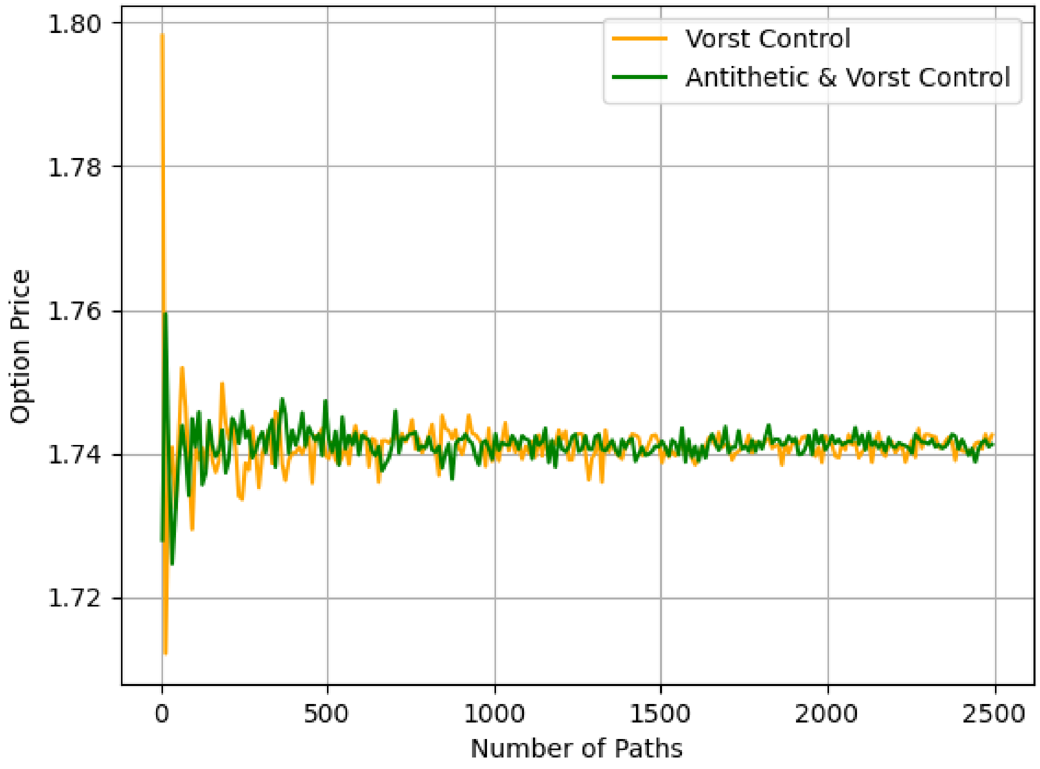

Figure 9 - Convergence speed comparison for Vorst control variate Monte Carlo and Vorst control variate combined with antithetic algorithm on Asian options

Figure 9 highlights that the methods that stand out most significantly are Vorst Control and "Antithetic + Vorst Control" algorithms. This is something we could have predicted since it combines both the Antithetic Monte Carlo method and the Control Variable Monte Carlo with Vorst variable (price of the geometric Asian option). Table 10 allows us to compare these two methods to see which is the most efficient relying on quantitative data.

Method	Option Price (€)	Standard Error	SEG	Computation Time (s)	P/T
Standard MC	1.72344	0.00807	1	25.05297	0.0299
Antithetic Variables	1.73638	0.006	1.35	17.19872	0.0782
Control Variables	1.74116	0.00018	44.83	24.95259	1.7967
Antithetic + Control	1.74099	0.00014	57.64	16.93714	3.4033

Table 10 - Option price, standard error, standard error gain, computation time and performance gain over computation time ratio obtained for the different Monte Carlo algorithms for Asian options price simulation (100,000 simulated paths for each algorithm). Source: Author's calculations.

First, from *Table 10* we observe that, as for the European options, the Antithetic method is more efficient mainly because of its gain in computation time, whereas the Vorst method has a big gain in terms of SEG. However, we see that for Asian options the combination of these two methods is more efficient than the Control Variable method in every respect. Indeed, it allows to obtain both a better standard error and a better computation speed. This is reflected in our P/T ratio, which indicates that the "Antithetic + Vorst control" method is twice as efficient as the Vorst control method alone.

3.3.3. Barrier Options

For this type of option, we defined the barrier as $B = S_0 exp(-rT)$ and we chose a down-out call option. Unfortunately, we were unable to identify an appropriate control variable for this option type. Therefore, we only studied the antithetic Monte Carlo algorithm which performances were compared to the standard Monte Carlo method without any variance reduction.

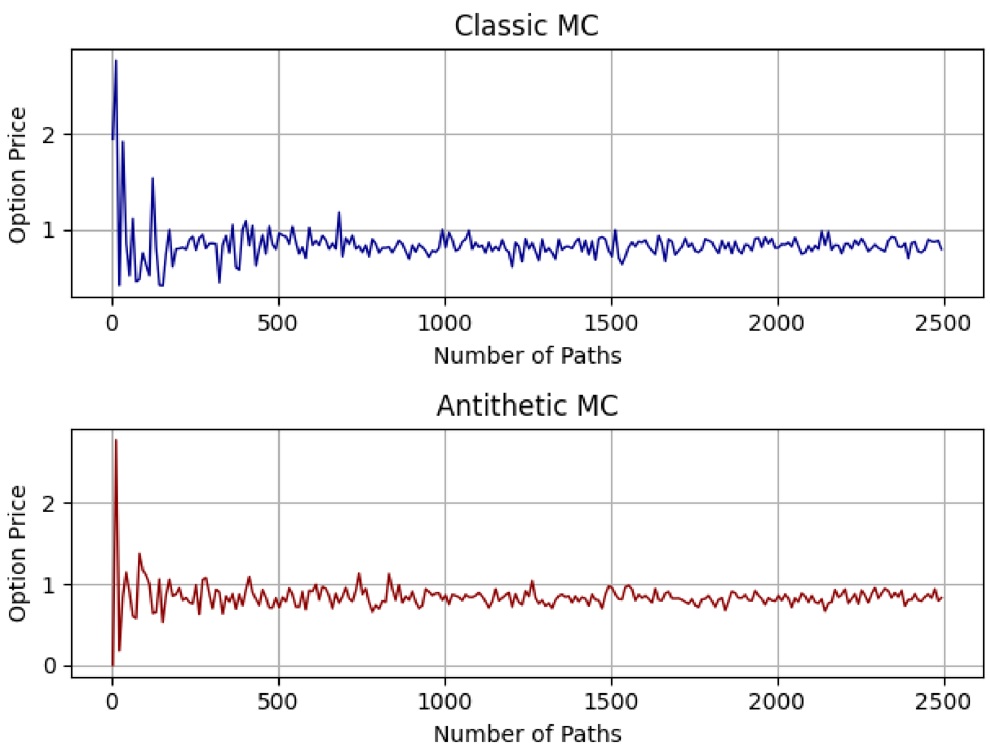

Figure 10 - Graphical representation of the Barrier option's price convergence for different Monte Carlo algorithms when the number of simulated paths increase

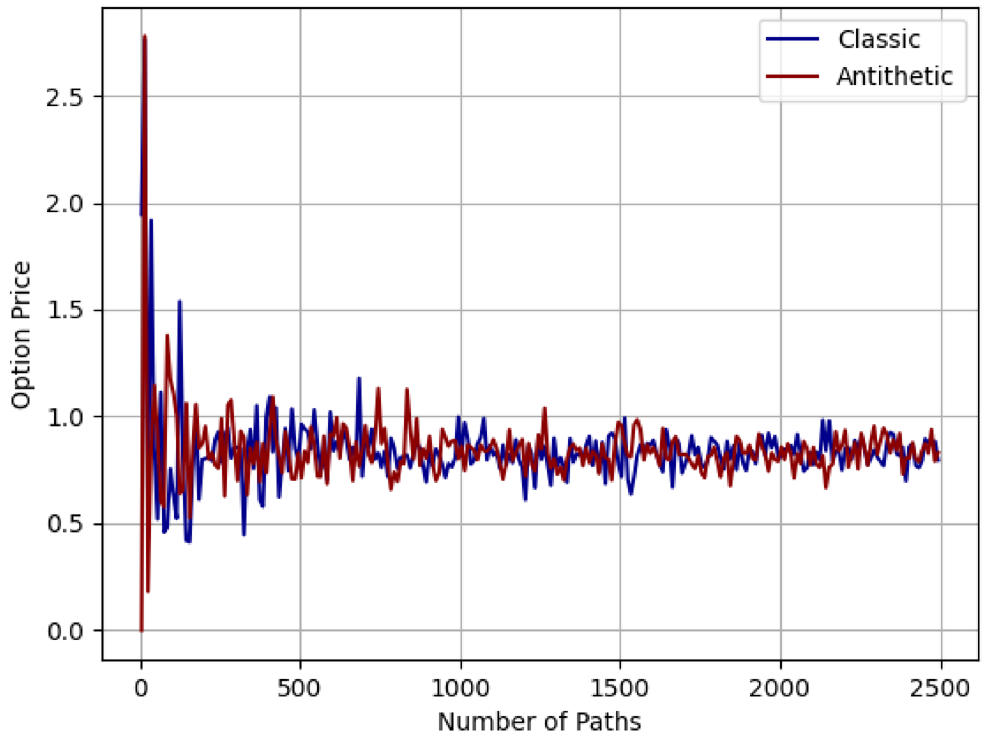

Figure 11 - Convergence speed comparison for different Monte Carlo algorithms on Barrier options

Figures 10 and *Figure 11* show the convergence of the option price using two different algorithms. However, the speed of convergence is quite similar. *Table 11* contains quantitative data illustrating the performance of each algorithm.

Method	**Option Price (€)**	**Standard Error**	**SEG**	**Computation Time (s)**	**P/T**
Standard MC	0.99107	0.01035	1	25.39720	0.0394
Antithetic Variables	1.00667	0.00992	1.0433	19.40992	0.0538

Table 11 - Option price, standard error, standard error gain, computation time and performance gain over computation time ratio obtained for the different Monte Carlo algorithms for Barrier options price simulation (100,000 simulated paths for each algorithm). Source: Author's calculations.

We observe a significant reduction in standard error when utilizing the Antithetic Monte Carlo method compared to the standard Monte Carlo simulation. This reduction is attributable to the method's design, which balances the variance by using antithetic variates. Consequently, the precision of our estimates improves.

Moreover, the Antithetic Monte Carlo method enhances the performance-time ratio significantly. It achieves this improvement by decreasing the overall computation time by approximately 25%. This reduction in computation time, coupled with the decreased standard error, underscores the efficiency and effectiveness of the Antithetic Monte Carlo method for simulations that require high precision and computational efficiency.

3.3.4. American Options

For the valuation of American options, we implemented the Longstaff-Schwartz algorithm. This algorithm is effective for estimating the optimal stopping rule (exercising the option at the optimal date). Additionally, we incorporated the antithetic Monte Carlo algorithm, utilizing the European option price as a control variable. This combination leverages the strengths of both methods, further refining our pricing accuracy.

The graphics presented in *Figure 12* illustrate the convergence behavior of the different algorithms as the number of simulated paths for Monte Carlo stock prices increases.

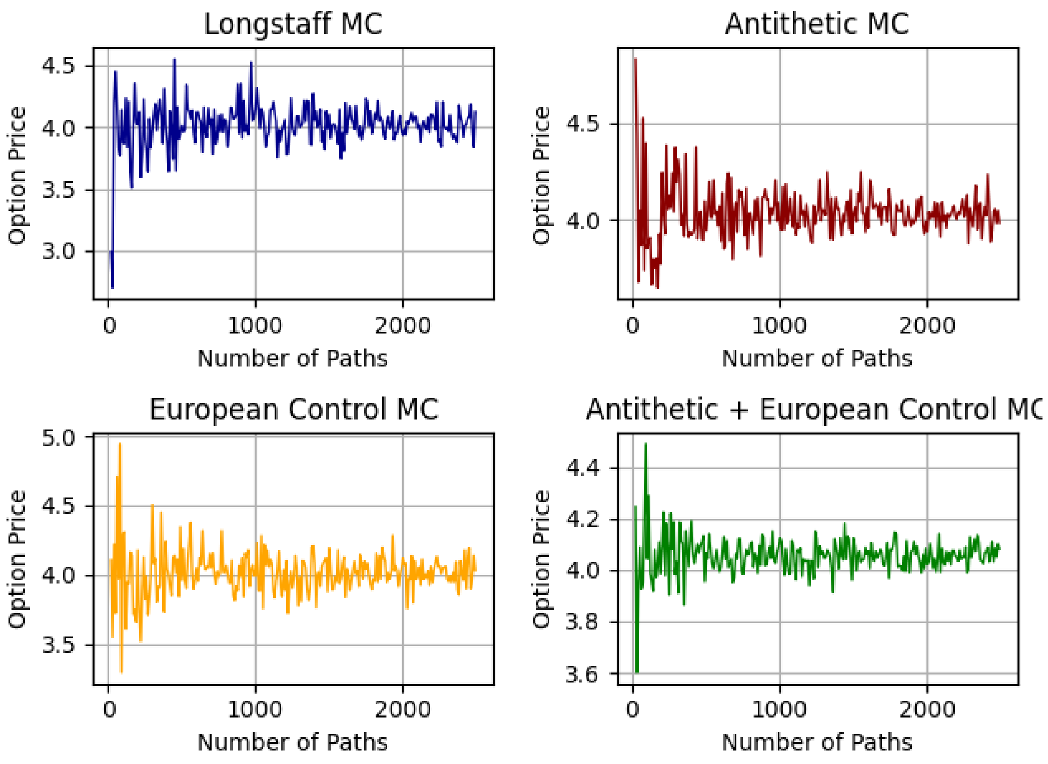

Figure 12 - Graphical Representation of the American option's price convergence for different Monte Carlo algorithms when the number of simulated paths increase

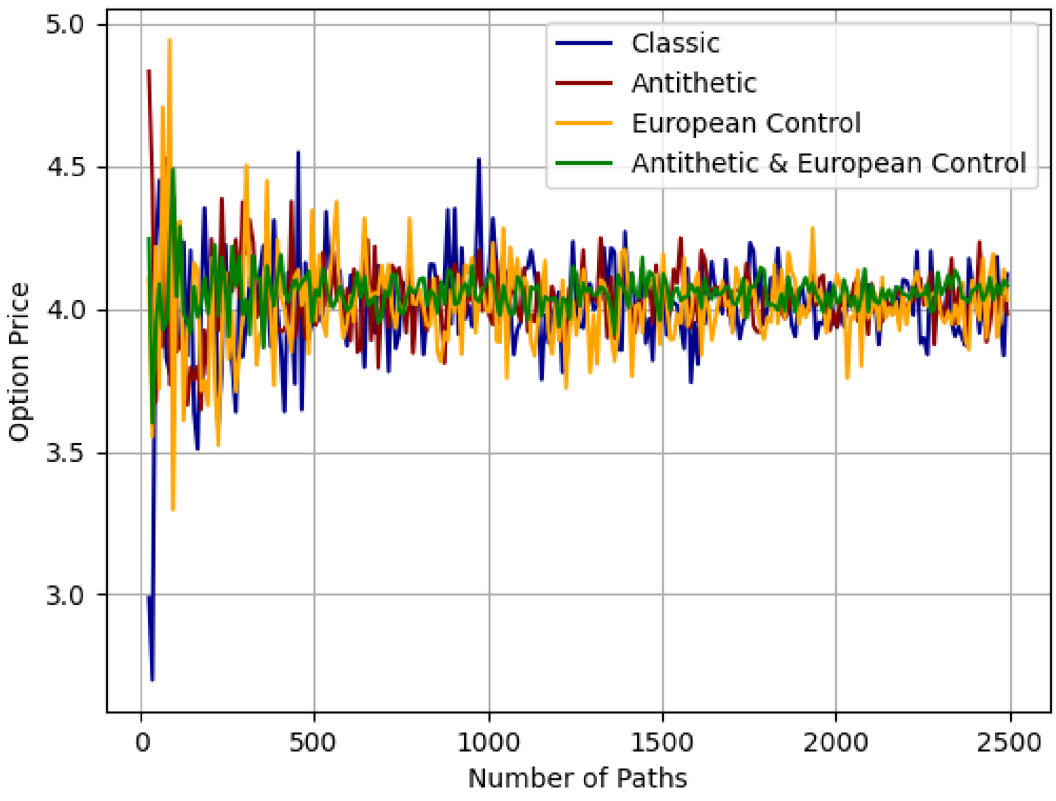

Figure 13 - Convergence speed comparison for different Monte Carlo algorithms on American options

By merging all the four graphics, *Figure 13* allows to visualize the difference in convergence's speed of the different algorithms.

Finally, *Table 12* provides the quantitative data allowing to compare numerically the performance of each algorithm. It is also important to note that due to the higher amount of calculations resources required for the Longstaff Schwartz algorithm, only 10,000 paths were simulated (10 times less than for other types of options). This must be taken into account when comparing computation time and P/T ratio of American algorithms with the one of other option's valuation algorithms.

Method	**Option Price (€)**	**Standard Error**	**SEG**	**Computation Time (s)**	**P/T**
Longstaff Schwartz	4.04609	0.04253	1	2.89974	0.3449
Antithetic Variables	4.01915	0.031	1.3719	2.01672	0.6803
Control Variables	3.99938	0.06043	0.7038	3.79394	0.1855
Antithetic + Control	4.00511	0.04549	0.9349	2.57622	0.3629

Table 12 - Option price, standard error, standard error gain, computation time and performance gain over computation time ratio obtained for the different Monte Carlo algorithms for American options price simulation (10,000 simulated paths for each algorithm).
Source: Author's calculations.

We observe that there is a performance gain with the antithetic variable algorithm. However, the control variable method is not efficient. Indeed, the Standard Error obtained with the control variable algorithm is higher than the Standard Error obtained with the classic Monte Carlo Method using the Longstaff Schwartz regression algorithm. This is due to the fact that the European price for the option is not sufficiently near the American price to be used as a control variable.

4. CONCLUSIONS

To address the problem: “**Are variance reduction methods similarly efficient across different types of options?”**, we evaluated the performance of several Monte Carlo (MC) methods (standard MC, Antithetic Variables, Control Variables, and a combination of Antithetic + Control) on European, Asian, Barrier, and American options. Figure 14 is a summary encapsulating the performance of each method for each option type.

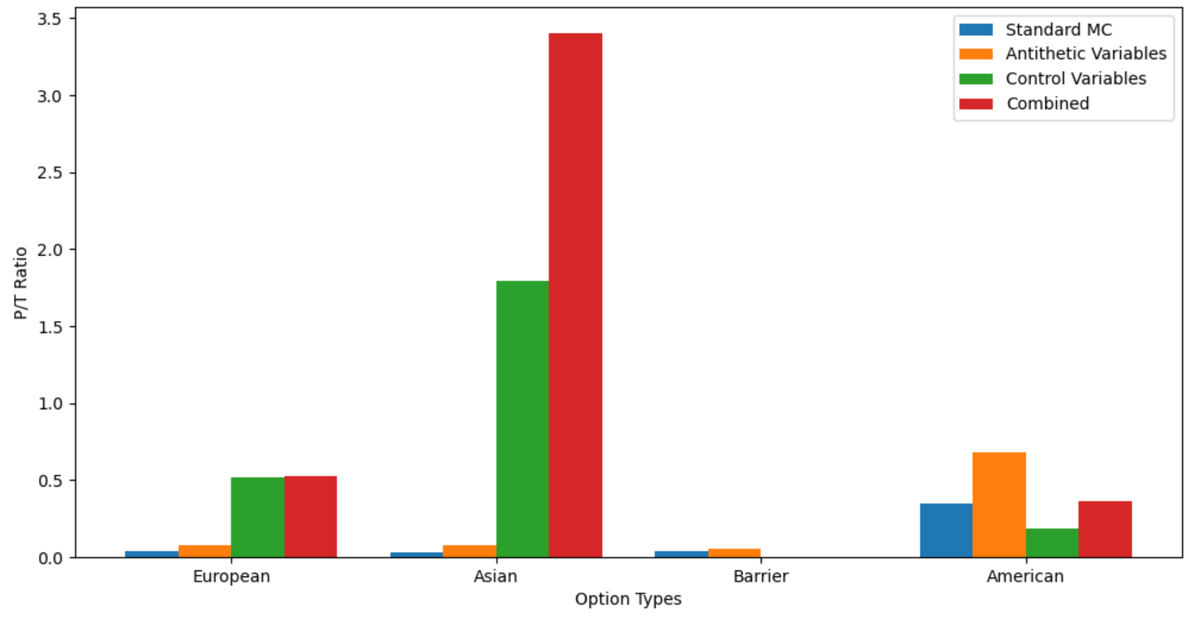

Figure 14 - Efficiency of Variance Reduction Methods Across Different Option Types

The performance of variance reduction methods varies significantly across different types of options. For European options, the Control Variables method shows a substantial improvement over the Standard MC, with a P/T ratio of 0.5213 compared to 0.0392. The combination of Antithetic Variables and Control Variables further enhances efficiency slightly to 0.5244.

Asian options see the most dramatic improvement with variance reduction methods. The Control Variables method alone achieves a P/T ratio of 1.7967 and combining it with Antithetic Variables boosts this ratio to 3.4033. This indicates that variance reduction methods are particularly effective for path-dependent options like Asian options.

Barrier options also benefit from variance reduction, though to a lesser extent. The Antithetic Variables method improves the P/T ratio from 0.0394 (Standard MC) to 0.0538, showing a moderate gain in efficiency.

For American options, where the Longstaff-Schwartz method is used as the Standard MC, the P/T ratio is 0.3449. Applying Antithetic Variables increases the efficiency to 0.6803. However, the Control Variables method results in a lower P/T ratio of 0.1855, suggesting it might not be as effective for American options in isolation. The combined method provides a balanced improvement with a P/T ratio of 0.3629.

In summary, the effectiveness of variance reduction methods is not uniform across different types of options:

- **European Options**: Control Variables and Combined methods provide significant efficiency improvements.
- **Asian Options**: The most substantial efficiency gains are observed, particularly with combined methods.
- **Barrier Options**: Moderate improvements are seen with Antithetic Variables.
- **American Options**: Antithetic Variables enhance efficiency, but the combination of methods shows a balanced improvement.

These findings suggest that while variance reduction methods are broadly effective, their impact varies depending on the option type. This underscores the importance of selecting the appropriate variance reduction technique tailored to the specific characteristics of the option being priced.

However, our analysis using the P/T ratio as an efficiency metric revealed potential limitations in this approach. This led us to propose a new problematic: **What are the most relevant and robust metrics to evaluate the efficiency of variance reduction methods for different types of options?** Evaluating alternative metrics such as the Variance Reduction Factor and the trade-off between computational cost and accuracy will provide a more comprehensive understanding of the true efficiency of these methods across various option types.

BIBLIOGRAPHY

1. Wiklund, Erik. *Asian Option Pricing and Volatility*. Master’s thesis, KTH, 2012.

2. Dodiya, Abhay. *"Barrier Option and Its Pricing in Python." Medium*. Accessed [12/06/2024].

3. Deville, Laurent. *"Introduction to Derivatives."* Course conducted at EDHEC Business School, Lille, 2024.

4. Barola, Alberto. *Monte Carlo Methods for American Option Pricing*. Master’s thesis, Copenhagen Business School, 2013.

5. Schweizer, Martin. *On Bermudan Options*. Institute of Mathematics, University of Munich, Germany.

6. Hull, John C. *Options, Futures, and Other Derivatives*. 8th ed. Boston: Prentice Hall, 2012.

7. Nielsen, Lars B. *Pricing Asian Options*. Master's thesis.

8. QuantPy: *Quant Finance with Python.* Accessed [13/06/2024].

9. Sarkar, P. K., and M. A. Prasad. "Variance Reduction in Monte-Carlo Radiation Transport Using Antithetic Variates." *Health Physics Unit, Variable Energy Cyclotron Centre*, I/AF Bidhan Nagar, Calcutta 700064, India; Division of Radiological Protection, Bhabha Atomic Research Centre, Bombay 400085, India, 1991.

APPENDIX

Appendix I – General Functions

```
#%% Definition of useful functions for the whole project

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from scipy.stats import norm
import math
from scipy.misc import derivative
from random import *

np.random.seed(42)

def box_muller():
    u1 = np.random.rand()
    u2 = np.random.rand()
    x = np.sqrt(-2*np.log(u1))*np.cos(2*np.pi*u2)
    return(x)

def simu_normal(n):
    simu = np.zeros(n)
    for k in range(n):
        simu[k] = box_muller()
    return(simu)

def densite_normal(x):
    d = (1/(np.sqrt(2*np.pi))) * np.exp(-0.5 * x**2)
    return(d)
```

```
#Theorical Price using Black-Scholes Model
def Black_Scholes(option_type, s, k, t, r, sigma):
    d1 = (np.log(k/s) + (r + sigma**2/2)*t) / (sigma*np.sqrt(t))
    d2 = d1 - sigma * np.sqrt(t)
    if option_type == "call":
        return (s*norm.cdf(d1) - (k * np.exp(-r*t)* norm.cdf(d2)))
    else:
        return ((k * np.exp(-r*t)* norm.cdf(-d2)) - s*norm.cdf(-d1))


def mult_brownian_paths(t, steps, S_0, number_paths, mu, sigma):  #number_years = 1, 2 or
3... and trading_days = 252 generally
    PATHS = np.zeros((number_paths, steps)) # List of the different prices paths
    delta_t = t/steps
    for i in range(number_paths):
        PATHS[i, 0] = S_0
        for j in range(1, steps):
            eps = box_muller()
            PATHS[i,j] = PATHS[i,j-1] * np.exp(((mu - 0.5 * sigma**2) * delta_t) + sigma * eps *
np.sqrt(delta_t))
    return PATHS


def mult_brownian_paths_antithetic(t, steps, S_0, number_paths, mu, sigma):  #number_years
= 1, 2 or 3... and trading_days = 252 generally
    PATHS_1 = np.zeros((number_paths, steps)) # List of the different prices paths
    PATHS_2 = np.zeros((number_paths, steps))
    delta_t = t/steps
    for i in range(number_paths):
        PATHS_1[i, 0] = S_0
        PATHS_2[i, 0] = S_0
        for j in range(1, steps):
            eps = box_muller()
            PATHS_1[i,j] = PATHS_1[i,j-1] * np.exp(((mu - 0.5 * sigma**2) * delta_t) + sigma *
eps * np.sqrt(delta_t))
```

```
            PATHS_2[i,j] = PATHS_2[i,j-1] * np.exp(((mu - 0.5 * sigma**2) * delta_t) - sigma *
eps * np.sqrt(delta_t))
    return PATHS_1, PATHS_2



def plot_paths(paths):
    plt.figure()
    plt.xlabel("Steps")
    plt.ylabel("Simulated Stock Price")
    for k in range(paths.shape[0]):
        plt.plot(paths[k])
    plt.savefig("Simulated Stock Price")
    plt.show()

def delta_european(s, k, t, r, sigma, option_type="call"):
    d1 = (np.log(s/k) + (r + 0.5*sigma**2)*t)/(sigma*np.sqrt(t))
    if option_type == "call":
        delta = norm.cdf(d1)
    elif option_type == "put":
        delta = norm.cdf(d1) - 1
    return delta



def geometric_asian_option_price(S_0, K, r, sigma, T, option_type, steps):
    sigma_adj = sigma * np.sqrt((2 * steps + 1) / (6 * (steps + 1)))
    mu_adj = 0.5 * (r - 0.5 * sigma ** 2) * (steps + 1) / (steps + 2) + 0.5 * sigma_adj ** 2
    d1 = (np.log(S_0 / K) + (mu_adj + 0.5 * sigma_adj ** 2) * T) / (sigma_adj * np.sqrt(T))
    d2 = d1 - sigma_adj * np.sqrt(T)

    if option_type == "call":
        price = np.exp(-r * T) * (S_0 * np.exp(mu_adj * T) * norm.cdf(d1) - K * norm.cdf(d2))
    else:
        price = np.exp(-r * T) * (K * norm.cdf(-d2) - S_0 * np.exp(mu_adj * T) * norm.cdf(-d1))

    return price

def conv_speed(list, precision):
    limit = list[-1]
    sup = limit * (1+precision)
    inf = limit*(1-precision)
    for i in range(len(list)-1, -1, -1):
        if list[i]>sup or list[i]<inf:
            return 1/i

    return -1
```

Appendix II – European Options Monte Carlo

```
#%% Settings of the simulation
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import yfinance as yf
from sklearn.linear_model import LinearRegression
from scipy.stats import norm
import math
import time
from MC_functions import *


STOCK = "TTE.PA"                                    #TotalEnergie
data = yf.download(STOCK)

stock = data.drop(labels = ['Open', "High", "Low", "Close", "Volume"], axis = 'columns')
stock["Returns"] = stock["Adj Close"] / stock["Adj Close"].shift(1) - 1
stock['Log Return'] = np.log(stock['Adj Close'] / stock['Adj Close'].shift(1))
sigma = stock["Log Return"].std()*np.sqrt(252)                 #Annualize volatility
of Log-Return
print("")

#Parameters of the simulation

R = 0.0390                                      #Risk free rate
SIGMA = sigma                                     #Volatility
S_0 = stock["Adj Close"].iloc[-1]                      #Initial Price
NUMBER_PATHS = 10**5                                  #Number of Simulations
K = S_0                                         #Strike Price (ATM Option)
```

```
T = 60/365                                          #Number of year until expiration of
the contract

STEPS = 50                                          #Number of trading day until
expiration

OPTION_TYPE = "call"


print("The parameters of the simulation are:")

print("")

print(f"Stock {OPTION_TYPE} option on {STOCK}")

print("")

print(f"Initial Price = {round(S_0, 5)}")

print(f"Annualized Volatility = {round(SIGMA, 5)}")

print(f"Risk free rate = {R}")

print(f"Number of steps = {STEPS}")

print(f"Strike Price = {K}")

print("")


print("With those settings, the theorical price using")

print(f"BS Model is : Option_Price = {round(Black_Scholes(OPTION_TYPE, S_0, K, T, R,
SIGMA), 3)}€")


#%% PATHS Simulation


#PATHS = mult_brownian_paths(T, STEPS, S_0, 100, R, SIGMA)

#plot_paths(PATHS)


#%% Pricing European Option with Classic MC


start = time.time()


PATHS = mult_brownian_paths(T, STEPS, S_0, NUMBER_PATHS, R, SIGMA)


def Classic_MC_European(paths, option_type, k, r, t, steps):
    dt = t/steps

    n = paths.shape[0]

    if option_type == "call":                                    #Computing the Payoff at
each nodes of the simulation

        Payoffs = np.maximum(paths[:,-1]-k,0)

    else:

        Payoffs = np.maximum(k-paths[:,-1],0)


    Actualized_Payoffs = np.exp(-r*t)*Payoffs                             #Actualizing the
Payoffs

    PRICE = np.mean(Actualized_Payoffs)                              #Averaging the payoffs
on the last date provide the option price

    SE = Actualized_Payoffs.std()/np.sqrt(n)                      #Computing the standard
error



    print("Using the standard Monte Carlo Simulation")

    print(f"Price of the option is = {round(PRICE, 5)}€")

    print(f"Standard Error for the price = {round(SE, 5)}")

    print("")

    print("The 95% confidence interval for the true value of the price :")

    print(f"[{round(PRICE - 1.96*SE,3)}, {round(PRICE + 1.96*SE, 3)}]")

    return(PRICE, SE)


Classic_MC_European(PATHS, OPTION_TYPE, K, R, T, STEPS);


print(time.time() - start)

#%% Pricing European Options using Antithetic Variable


start = time.time()

PATHS_1, PATHS_2 = mult_brownian_paths_antithetic(T, STEPS, S_0,
NUMBER_PATHS//2, R, SIGMA)


def Antithetic_MC_European(paths_1, paths_2, option_type, k, r, t, steps):
    dt = t/steps
    n = paths_1.shape[0]

    if option_type == "call":                                          #Computing the Payoff at
each nodes of the simulation
        Payoffs = (np.maximum(paths_1[:,-1]-k,0) + np.maximum(paths_2[:,-1]-k,0))/2
    else:
        Payoffs = (np.maximum(k-paths_1[:,-1],0) + np.maximum(k-paths_2[:,-1],0))/2

    Actualized_Payoffs = np.exp(-r*t)*Payoffs                                  #Actualizing the
Payoffs
    PRICE = np.mean(Actualized_Payoffs)                                   #Averaging the payoffs
on the last date provide the option price
    SE = Actualized_Payoffs.std()/np.sqrt(n)                           #Computing the standard
error



    print("Using the Antithetic Monte Carlo Simulation")
    print(f"Price of the option is = {round(PRICE, 5)}€")
    print(f"Standard Error for the price = {round(SE, 5)}")
    print("")
    print("The 95% confidence interval for the true value of the price :")
    print(f"[{round(PRICE - 1.96*SE,3)}, {round(PRICE + 1.96*SE, 3)}]")
    return(PRICE, SE)

Antithetic_MC_European(PATHS_1, PATHS_2, OPTION_TYPE, K, R, T, STEPS);
print(time.time() - start)

#%% Pricing European Options using Delta Hedge Control Variable
start = time.time()
PATHS = mult_brownian_paths(T, STEPS, S_0, NUMBER_PATHS, R, SIGMA)

def Delta_Control_MC_European(paths, option_type, k, r, t, steps):
    dt = t/steps
    n = paths.shape[0]

    if option_type == "call":                                     #Computing the Payoff at
each nodes of the simulation
        Payoffs = np.maximum(paths[:,-1]-k,0)
    else:
        Payoffs = np.maximum(k-paths[:,-1],0)

    delta_matrix = delta_european(paths[:, :-1], k, np.linspace(t, 0, steps)[:-1], r, SIGMA,
option_type)
    control_matrix = delta_matrix*(paths[:,1:] - paths[:,:-
1]*np.exp(r*dt))*np.exp(r*np.linspace(t, 0, steps-1))
    control_variate_total = np.sum(control_matrix, axis=1)
    Controled_Payoffs = Payoffs - control_variate_total

    #control_variate_total = np.cumsum(delta_matrix*(paths[:,1:] - paths[:,:-
1]*np.exp(r*dt))*np.exp(r*np.linspace(t, 0, steps-1)), axis=1)
    #Controled_Payoffs = Payoffs - control_variate_total[:,-1]

    Actualized_Controled_Payoffs = np.exp(-r*t)*Controled_Payoffs
    PRICE = np.mean(Actualized_Controled_Payoffs)
    SE = Actualized_Controled_Payoffs.std()/np.sqrt(n)

    print("Using the Delta hedging control variable Monte Carlo Simulation")
    print(f"Price of the option is = {round(PRICE, 5)}€")
    print(f"Standard Error for the price = {round(SE, 5)}")
    print("")
    print("The 95% confidence interval for the true value of the price :")
    print(f"[{round(PRICE - 1.96*SE,3)}, {round(PRICE + 1.96*SE, 3)}]")
    return(PRICE, SE)

Delta_Control_MC_European(PATHS, OPTION_TYPE, K, R, T, STEPS)
```

```
print(time.time() - start)


#%% Pricing European Options using Antithetic and Control Variables
start = time.time()


PATHS_3, PATHS_4 = mult_brownian_paths_antithetic(T, STEPS, S_0,
NUMBER_PATHS//2, R, SIGMA)


def Antithetic_Control_MC_European(paths_1, paths_2, option_type, k, r, t, steps):
    dt = t/steps
    n = paths_1.shape[0]


    if option_type == "call":                                        #Computing the Payoff at
each nodes of the simulation
        Payoffs = (np.maximum(paths_1[:,-1]-k,0) + np.maximum(paths_2[:,-1]-k,0))/2
    else:
        Payoffs = (np.maximum(k-paths_1[:,-1],0) + np.maximum(k-paths_2[:,-1],0))/2



    delta_matrix_1 = delta_european(paths_1[:, :-1], k, np.linspace(t, 0, steps)[:-1], r, SIGMA,
option_type)
    control_matrix_1 = delta_matrix_1*(paths_1[:,1:] - paths_1[:,:-
1]*np.exp(r*dt))*np.exp(r*np.linspace(t, 0, steps-1))
    control_variate_total_1 = np.sum(control_matrix_1, axis=1)


    delta_matrix_2 = delta_european(paths_2[:, :-1], k, np.linspace(t, 0, steps)[:-1], r, SIGMA,
option_type)
    control_matrix_2 = delta_matrix_2*(paths_2[:,1:] - paths_2[:,:-
1]*np.exp(r*dt))*np.exp(r*np.linspace(t, 0, steps-1))
    control_variate_total_2 = np.sum(control_matrix_2, axis=1)


    control_variate_total = (control_variate_total_1 + control_variate_total_2)/2


    Controled_Payoffs = (Payoffs - control_variate_total)
    Actualized_Controled_Payoffs = np.exp(-r*t)*Controled_Payoffs
    PRICE = np.mean(Actualized_Controled_Payoffs)
    SE = Actualized_Controled_Payoffs.std()/np.sqrt(n)

    print("Using Antithetic and Delta hedging control variable Monte Carlo Simulation")
    print(f"Price of the option is = {round(PRICE, 5)}€")
    print(f"Standard Error for the price = {round(SE, 5)}")
    print("")
    print("The 95% confidence interval for the true value of the price :")
    print(f"[{round(PRICE - 1.96*SE,3)}, {round(PRICE + 1.96*SE, 3)}]")
    return(PRICE, SE)

Antithetic_Control_MC_European(PATHS_3, PATHS_4, OPTION_TYPE, K, R, T, STEPS)
print(time.time() - start)

#%% Convergence of different algos

interval = np.arange(4, 2500+4,10)
Price_Classic_European = []
Price_Antithetic_European = []
Price_Delta_European = []
Price_Anti_Delta_European = []

for n in interval:
    PATHS = mult_brownian_paths(T, STEPS, S_0, n, R, SIGMA)
    Price_Classic_European.append(Classic_MC_European(PATHS, OPTION_TYPE, K, R, T,
STEPS)[0])

for n in interval:
    PATHS_1, PATHS_2 = mult_brownian_paths_antithetic(T, STEPS, S_0, n//2, R, SIGMA)
    Price_Antithetic_European.append(Antithetic_MC_European(PATHS_1, PATHS_2,
OPTION_TYPE, K, R, T, STEPS)[0])
```

```
for n in interval:
    PATHS = mult_brownian_paths(T, STEPS, S_0, n, R, SIGMA)
    Price_Delta_European.append(Delta_Control_MC_European(PATHS, OPTION_TYPE, K,
R, T, STEPS)[0])

for n in interval:
    PATHS_3, PATHS_4 = mult_brownian_paths_antithetic(T, STEPS, S_0, n//2, R, SIGMA)
    Price_Anti_Delta_European.append(Antithetic_Control_MC_European(PATHS_3,
PATHS_4, OPTION_TYPE, K, R, T, STEPS)[0])

#%% Plots
fig, axs = plt.subplots(2, 2)

axs[0, 0].plot(interval, Price_Classic_European, color = 'darkblue', linewidth = 1)
axs[0, 0].set_title('Classic MC')
axs[0, 0].grid(True)
axs[0, 0].set_xlabel('Number of Paths')
axs[0, 0].set_ylabel('Option Price')

axs[0, 1].plot(interval, Price_Antithetic_European, color = 'darkred', linewidth = 1)
axs[0, 1].set_title('Antithetic MC')
axs[0, 1].grid(True)
axs[0, 1].set_xlabel('Number of Paths')
axs[0, 1].set_ylabel('Option Price')

axs[1, 0].plot(interval, Price_Delta_European, color = 'orange', linewidth = 1)
axs[1, 0].set_title('Delta Control MC')
axs[1, 0].grid(True)
axs[1, 0].set_xlabel('Number of Paths')
axs[1, 0].set_ylabel('Option Price')

axs[1, 1].plot(interval, Price_Anti_Delta_European, color = 'green', linewidth = 1)
axs[1, 1].set_title('Antithetic + Delta Control MC')
```

```
axs[1, 1].grid(True)
axs[1, 1].set_xlabel('Number of Paths')
axs[1, 1].set_ylabel('Option Price')

fig.tight_layout()
plt.savefig("European MC Convergence Speed for each algos")



plt.figure()
plt.plot(interval, Price_Classic_European, color = "darkblue", label = 'Classic')
plt.plot(interval, Price_Antithetic_European, color = "darkred", label = 'Antithetic')
plt.plot(interval, Price_Delta_European, color = "orange", label = 'Delta Control')
plt.plot(interval, Price_Anti_Delta_European, color = "green", label = 'Antithetic & Delta
Control')
plt.legend()
plt.xlabel("Number of Paths")
plt.ylabel("Option Price")
plt.grid()

plt.savefig('European Speed Convergence Comparison')



# %%
def conv_speed(list, precision):
    limit = list[-1]
    sup = limit * (1+precision)
    inf = limit*(1-precision)
    for i in range(len(list)-1, 0, -1):
        if list[i]>sup or list[i]<inf:
            return 1/i

    return -1           #If all value in interval
```

```
print(conv_speed(Price_Classic_European, 0.025))
print(conv_speed(Price_Antithetic_European, 0.025))
print(conv_speed(Price_Delta_European, 0.025))
print(conv_speed(Price_Anti_Delta_European, 0.025))
# %%
```

Appendix III – Asian Options Monte Carlo

```
#%% Settings of the simulation
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import yfinance as yf
from scipy.stats import norm
from MC_functions import *
import time



np.random.seed(2024)

STOCK = "TTE.PA"                                        #TotalEnergie
data = yf.download(STOCK)

stock = data.drop(labels = ['Open', "High", "Low", "Close", "Volume"], axis = 'columns')
stock["Returns"] = stock["Adj Close"] / stock["Adj Close"].shift(1) - 1
stock['Log Return'] = np.log(stock['Adj Close'] / stock['Adj Close'].shift(1))

sigma = stock["Log Return"].std()*np.sqrt(252)                    #Annualize volatility
of Log-Return
print("")



#Parameters of the simulation

R = 0.0390                                          #Risk free rate
SIGMA = sigma                                       #Volatility
S_0 = stock["Adj Close"].iloc[-1]                   #Initial Price
NUMBER_PATHS = 10**3                                #Number of Simulations
#Strike Price (ATM Option)
```

```
T = 60/365                                              #Number of year until expiration of
the contract

STEPS = 50

K = S_0                             #Number of trading day until expiration

OPTION_TYPE = "call"


print("The parameters of the simulation are:")

print("")

print(f"Stock {OPTION_TYPE} option on {STOCK}")

print("")

print(f"Initial Price = {round(S_0, 5)}")

print(f"Annualized Volatility = {round(SIGMA, 5)}")

print(f"Risk free rate = {R}")

print(f"Number of steps = {STEPS}")

print("")


#%% Pricing Asian Option with Classic MC


start = time.time()


PATHS = mult_brownian_paths(T, STEPS, S_0, NUMBER_PATHS, R, SIGMA)
#Generating the differents paths


def Classic_MC_Asian(paths, k, option_type, r, t):                          #Vector of
strikes for each path simulation

   if option_type == "call":

      Payoffs = np.maximum(np.mean(paths, axis = 1) - k,0)

   else:

      Payoffs = np.maximum(k - np.mean(paths, axis = 1),0)

   Actualized_Payoffs = np.exp(-r*t)*Payoffs                     #Actualizing the Payoffs

   PRICE = np.mean(Actualized_Payoffs)                           #Averaging the payoffs
(MC method)

   SE = Actualized_Payoffs.std()/np.sqrt(NUMBER_PATHS)                #Computing
the standard error
```

```
    print("Using the standard Monte Carlo Simulation")
    print(f"Price of the option is = {round(PRICE, 5)}€")
    print(f"Standard Error for the price = {round(SE, 5)}")
    print("")
    print("The 95% confidence interval for the true value of the price :")
    print(f"[{round(PRICE - 1.96*SE,3)}, {round(PRICE + 1.96*SE, 3)}]")
    return(PRICE, SE)


Classic_MC_Asian(PATHS, K, OPTION_TYPE, R, T)


print(time.time() - start)
#%% Pricing Asian Options using Antithetic Variable
start = time.time()
PATHS_1, PATHS_2 = mult_brownian_paths_antithetic(T, STEPS, S_0,
NUMBER_PATHS//2, R, SIGMA)


def Antithetic_MC_Asian(paths_1, paths_2, k, option_type, r, t):
    n = paths_1.shape[0]                                    #Vector of strikes for each path_2
simulation
    if option_type == "call":                                      #Computing the Payoff at
each nodes of the simulation
        Payoffs = (np.maximum(np.mean(paths_1, axis = 1) - k,0) +
np.maximum(np.mean(paths_2, axis = 1)-k,0))/2
    else:
        Payoffs = (np.maximum(k - np.mean(paths_1, axis = 1) ,0) + np.maximum(k-
np.mean(paths_2, axis = 1),0))/2


    Actualized_Payoffs = np.exp(-r*t)*Payoffs                              #Actualizing the
Payoffs
    PRICE = np.mean(Actualized_Payoffs)                              #Averaging the payoffs
on the last date provide the option price
    SE = Actualized_Payoffs.std()/np.sqrt(n)                         #Computing the standard
error
```

```
    print("Using the Antithetic Monte Carlo Simulation")
    print(f"Price of the option is = {round(PRICE, 5)}€")
    print(f"Standard Error for the price = {round(SE, 5)}")
    print("")
    print("The 95% confidence interval for the true value of the price :")
    print(f"[{round(PRICE - 1.96*SE,3)}, {round(PRICE + 1.96*SE, 3)}]")
    return(PRICE, SE)

Antithetic_MC_Asian(PATHS_1, PATHS_2, K,OPTION_TYPE, R, T);
print(time.time() - start)
#%% Pricing Asian Options using Vorst Control Variable

start = time.time()

PATHS = mult_brownian_paths(T, STEPS, S_0, NUMBER_PATHS, R, SIGMA)
#Generating the differents paths
# Geometric Asian option price
geom_price_exact = geometric_asian_option_price(S_0, K, R, SIGMA, T, OPTION_TYPE,
STEPS)
print(f"Geometric Asian option price (exact) = {round(geom_price_exact, 5)}€")

# Function for Monte Carlo pricing with control variate
def Vorst_Control_MC_Asian(paths, k, option_type, r, t):
    if option_type == "call":
        arith_payoffs = np.maximum(np.mean(paths[:, 1:], axis=1) - k, 0)
        geom_means = np.exp(np.mean(np.log(paths[:, 1:]), axis=1))
        geom_payoffs = np.maximum(geom_means - k, 0)
    else:
        arith_payoffs = np.maximum(k - np.mean(paths[:, 1:], axis=1), 0)
        geom_means = np.exp(np.mean(np.log(paths[:, 1:]), axis=1))
        geom_payoffs = np.maximum(k - geom_means, 0)

    actualized_arith = np.exp(-r * t) * arith_payoffs
    actualized_geom = np.exp(-r * t) * geom_payoffs
    Actualized_Payoffs = actualized_arith - actualized_geom + geom_price_exact

    PRICE = np.mean(Actualized_Payoffs)                          #Averaging the payoffs on
the last date provide the option price
    SE = Actualized_Payoffs.std()/np.sqrt(NUMBER_PATHS)



    print("Using the Monte Carlo Simulation with Control Variate")
    print(f"Price of the option is = {round(PRICE, 5)}€")
    print(f"Standard Error for the price = {round(SE, 5)}")
    print("The 95% confidence interval for the true value of the price :")
    print(f"[{round(PRICE - 1.96 * SE, 5)}, {round(PRICE + 1.96 * SE, 5)}]")
    return PRICE, SE

Vorst_Control_MC_Asian(PATHS, K, OPTION_TYPE, R, T)

print(time.time() - start)



#%% Pricing Asians Options using Antithetic and Control Variables

start = time.time()

PATHS_3, PATHS_4 = mult_brownian_paths_antithetic(T, STEPS, S_0,
NUMBER_PATHS//2, R, SIGMA)

def Antithetic_Control_MC_Asian(paths_1, paths_2, option_type, k, r, t):
    n = paths_1.shape[0]

    geom_price_exact = geometric_asian_option_price(S_0, K, R, SIGMA, T,
OPTION_TYPE, STEPS)
```

```
    if option_type == "call":                                    #Computing the Payoff at
each nodes of the simulation
        arith_payoffs1 = np.maximum(np.mean(paths_1[:, 1:], axis=1) - k, 0)
        geom_means1 = np.exp(np.mean(np.log(paths_1[:, 1:]), axis=1))
        geom_payoffs1 = np.maximum(geom_means1 - k, 0)

        arith_payoffs2 = np.maximum(np.mean(paths_2[:, 1:], axis=1) - k, 0)
        geom_means2 = np.exp(np.mean(np.log(paths_2[:, 1:]), axis=1))
        geom_payoffs2 = np.maximum(geom_means2 - k, 0)
    else:
        arith_payoffs1 = np.maximum(k - np.mean(paths_1[:, 1:], axis=1), 0)
        geom_means1 = np.exp(np.mean(np.log(paths_1[:, 1:]), axis=1))
        geom_payoffs1 = np.maximum(k - geom_means1, 0)

        arith_payoffs2 = np.maximum(k - np.mean(paths_2[:, 1:], axis=1), 0)
        geom_means2 = np.exp(np.mean(np.log(paths_2[:, 1:]), axis=1))
        geom_payoffs2 = np.maximum(k - geom_means2, 0)

    actualized_arith1 = np.exp(-r * t) * arith_payoffs1
    actualized_geom1 = np.exp(-r * t) * geom_payoffs1
    Actualized_Payoffs1 = actualized_arith1 - actualized_geom1 + geom_price_exact

    actualized_arith2 = np.exp(-r * t) * arith_payoffs2
    actualized_geom2 = np.exp(-r * t) * geom_payoffs2
    Actualized_Payoffs2 = actualized_arith2 - actualized_geom2 + geom_price_exact

    Actualized_Controled_Payoffs = (Actualized_Payoffs1 + Actualized_Payoffs2)/2

    PRICE = np.mean(Actualized_Controled_Payoffs)
    SE = Actualized_Controled_Payoffs.std()/np.sqrt(n)

    print("Using Antithetic and Vorst control variable Monte Carlo Simulation")
    print(f"Price of the option is = {round(PRICE, 5)}€")
    print(f"Standard Error for the price = {round(SE, 5)}")
    print("")
    print("The 95% confidence interval for the true value of the price :")
    print(f"[{round(PRICE - 1.96*SE,3)}, {round(PRICE + 1.96*SE, 3)}]")
    return(PRICE, SE)

Antithetic_Control_MC_Asian(PATHS_3, PATHS_4, OPTION_TYPE, K, R, T)
print(time.time() - start)




#%% Defining delta function for asian options
def Classic_MC_Asian2(paths, k, option_type, r, t):                         #Vector of
strikes for each path simulation
    if option_type == "call":
        Payoffs = np.maximum(np.mean(paths, axis = 1) - k,0)
    else:
        Payoffs = np.maximum(k - np.mean(paths, axis = 1),0)
    Actualized_Payoffs = np.exp(-r*t)*Payoffs
    PRICE = np.mean(Actualized_Payoffs)
    return(PRICE)

def only_price_asian(x):
    PATHS = mult_brownian_paths(T, STEPS, x, NUMBER_PATHS, R, SIGMA)
    p = Classic_MC_Asian2(PATHS, K, OPTION_TYPE, R, T);
    return p;

def delta_asian(x):
    return derivative(only_price_asian,x)

print(only_price_asian(S_0))
print(delta_asian(S_0))


x = np.linspace(50, 80 , 100)
y = np.zeros(100)
for k in range(100):
    y[k] = delta_asian(x[k])
plt.plot(x,y)


#%% Convergence Classic


interval = np.arange(4, 2500+4,10)
Price_Classic_Asian = []
Price_Antithetic_Asian = []
Price_Vorst_Asian = []
Price_Antithetic_Vorst_Asian = []


for n in interval:
    PATHS = mult_brownian_paths(T, STEPS, S_0, n, R, SIGMA)
    Price_Classic_Asian.append(Classic_MC_Asian(PATHS, K, OPTION_TYPE, R, T)[0])


for n in interval:
    PATHS_1, PATHS_2 = mult_brownian_paths_antithetic(T, STEPS, S_0, n//2, R, SIGMA)
    Price_Antithetic_Asian.append(Antithetic_MC_Asian(PATHS_1, PATHS_2, K, 
OPTION_TYPE, R, T)[0])


for n in interval:
    PATHS = mult_brownian_paths(T, STEPS, S_0, n, R, SIGMA)
    Price_Vorst_Asian.append(Vorst_Control_MC_Asian(PATHS, K, OPTION_TYPE, R, 
T)[0])


for n in interval:
    PATHS_3, PATHS_4 = mult_brownian_paths_antithetic(T, STEPS, S_0, n//2, R, SIGMA)
    Price_Antithetic_Vorst_Asian.append(Antithetic_Control_MC_Asian(PATHS_3,
PATHS_4, OPTION_TYPE, K, R, T)[0])

#%% MPlots



fig, axs = plt.subplots(2, 2)

axs[0, 0].plot(interval, Price_Classic_Asian, color = 'darkblue', linewidth = 1)
axs[0, 0].set_title('Classic MC')
axs[0, 0].grid(True)
axs[0, 0].set_xlabel('Number of Paths')
axs[0, 0].set_ylabel('Option Price')

axs[0, 1].plot(interval, Price_Antithetic_Asian, color = 'darkred', linewidth = 1)
axs[0, 1].set_title('Antithetic MC')
axs[0, 1].grid(True)
axs[0, 1].set_xlabel('Number of Paths')
axs[0, 1].set_ylabel('Option Price')

axs[1, 0].plot(interval, Price_Vorst_Asian, color = 'orange', linewidth = 1)
axs[1, 0].set_title('Vorst Control MC')
axs[1, 0].grid(True)
axs[1, 0].set_xlabel('Number of Paths')
axs[1, 0].set_ylabel('Option Price')

axs[1, 1].plot(interval, Price_Antithetic_Vorst_Asian, color = 'green', linewidth = 1)
axs[1, 1].set_title('Antithetic + Vorst Control MC')
axs[1, 1].grid(True)
axs[1, 1].set_xlabel('Number of Paths')
axs[1, 1].set_ylabel('Option Price')
```

```
fig.tight_layout()
plt.savefig("Asian MC Convergence Speed for each algos")



plt.figure()
plt.plot(interval, Price_Classic_Asian, color = "darkblue", label = 'Classic')
plt.plot(interval, Price_Antithetic_Asian, color = "darkred", label = 'Antithetic')
plt.plot(interval, Price_Vorst_Asian, color = "orange", label = 'Vorst Control')
plt.plot(interval, Price_Antithetic_Vorst_Asian, color = "green", label = 'Antithetic & Vorst
Control')
plt.legend()
plt.xlabel("Number of Paths")
plt.ylabel("Option Price")
plt.grid()

plt.savefig('Asian Speed Convergence Comparison')

# %%

plt.figure()
plt.plot(interval, Price_Vorst_Asian, color = "orange", label = 'Vorst Control')
plt.plot(interval, Price_Antithetic_Vorst_Asian, color = "green", label = 'Antithetic & Vorst
Control')
plt.legend()
plt.xlabel("Number of Paths")
plt.ylabel("Option Price")
plt.grid()

plt.savefig('Asian Speed Convergence Comparison Two Bests algorithms')
# %%
```

Appendix IV – Barrier Options Monte Carlo

```
#%% Settings of the simulation
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import yfinance as yf
from scipy.stats import norm
from MC_functions import *
import time

np.random.seed(2024)

STOCK = "TTE.PA"                                    #TotalEnergie
data = yf.download(STOCK)

stock = data.drop(labels = ['Open', "High", "Low", "Close", "Volume"], axis = 'columns')
stock["Returns"] = stock["Adj Close"] / stock["Adj Close"].shift(1) - 1
stock['Log Return'] = np.log(stock['Adj Close'] / stock['Adj Close'].shift(1))

sigma = stock["Log Return"].std()*np.sqrt(252)                 #Annualize volatility
of Log-Return
print("")



#Parameters of the simulation

R = 0.0390                                    #Risk free rate
SIGMA = sigma                                   #Volatility
S_0 = stock["Adj Close"].iloc[-1]                      #Initial Price
NUMBER_PATHS = 10**5                                #Number of Simulations
K = S_0                                       #Strike Price (ATM Option)
#Strike Price (ATM Option)
```

```
T = 60/365                                           #Number of year until expiration of
the contract

STEPS = 50                                            #Number of trading day until
expiration

OPTION_TYPE = "call"

BARRIER_TYPE = "DO"

B = S_0*np.exp(-R*T)                                     #Barrier of the option


print("The parameters of the simulation are:")

print("")

print(f"Stock {BARRIER_TYPE} {OPTION_TYPE} option on {STOCK}")

print("")

print(f"Initial Price = {round(S_0, 5)}")

print(f"Annualized Volatility = {round(SIGMA, 5)}")

print(f"Risk free rate = {R}")

print(f"Number of steps = {STEPS}")

print(f"Strike Price = {K}")

print(f"Barrier = {B}")

print("")



#%% Pricing Barrier Option with Classic MC

start = time.time()

PATHS = mult_brownian_paths(T, STEPS, S_0, NUMBER_PATHS, R, SIGMA)


def Classic_MC_Barrier(paths, option_type, barrier_type, b, k, r, t):


    if option_type == "call":

        Payoffs = np.maximum(paths[:,-1]-k,0)

    else:

        Payoffs = np.maximum(k-paths[:,-1],0)

    if barrier_type == "UO":

        Payoffs = Payoffs*((np.all(paths<b,axis=1)).astype(int))
    elif barrier_type == "UI":
        Payoffs = Payoffs*((np.any(paths>=b,axis=1)).astype(int))
    elif barrier_type == "DO":
        Payoffs = Payoffs*((np.all(paths>b,axis=1)).astype(int))
    elif barrier_type == "DI":
        Payoffs = Payoffs*((np.any(paths<=b,axis=1)).astype(int))
    Actualized_Payoffs = np.exp(-r*t)*Payoffs                          #Actualizing the Payoffs
    PRICE = np.mean(Actualized_Payoffs)                                #Averaging the payoffs
(MC method)
    SE = Actualized_Payoffs.std()/np.sqrt(NUMBER_PATHS)                        #Computing
the standard error

    print("Using the standard Monte Carlo Simulation")
    print(f"Price of the option is = {round(PRICE, 5)}€")
    print(f"Standard Error for the price = {round(SE, 5)}")
    print("")
    print("The 95% confidence interval for the true value of the price :")
    print(f"[{round(PRICE - 1.96*SE,3)}, {round(PRICE + 1.96*SE, 3)}]")
    return(PRICE, SE)

Classic_MC_Barrier(PATHS, OPTION_TYPE, BARRIER_TYPE, B, K, R, T)
print(time.time() - start)
#%% Pricing Barrier Options using Antithetic Variable
start = time.time()
PATHS_1, PATHS_2 = mult_brownian_paths_antithetic(T, STEPS, S_0,
NUMBER_PATHS//2, R, SIGMA)

def Antithetic_MC_Barrier(paths_1, paths_2, option_type, barrier_type, b, k, r, t):
    n = paths_1.shape[0]

    if option_type == "call":
        Payoffs_1 = np.maximum(paths_1[:,-1]-k,0)
        Payoffs_2 = np.maximum(paths_2[:,-1]-k,0)
    else:
        Payoffs_1 = np.maximum(k-paths_1[:,-1],0)
        Payoffs_2 = np.maximum(k-paths_2[:,-1],0)

    if barrier_type == "UO":
        Payoffs_1 = Payoffs_1*((np.all(paths_1<b,axis=1)).astype(int))
        Payoffs_2 = Payoffs_2*((np.all(paths_2<b,axis=1)).astype(int))
    elif barrier_type == "UI":
        Payoffs_1 = Payoffs_1*((np.any(paths_1>=b,axis=1)).astype(int))
        Payoffs_2 = Payoffs_2*((np.any(paths_2>=b,axis=1)).astype(int))
    elif barrier_type == "DO":
        Payoffs_1 = Payoffs_1*((np.all(paths_1>b,axis=1)).astype(int))
        Payoffs_2 = Payoffs_2*((np.all(paths_2>b,axis=1)).astype(int))
    else :
        Payoffs_1 = Payoffs_1*((np.any(paths_1<=b,axis=1)).astype(int))
        Payoffs_2 = Payoffs_2*((np.any(paths_2<=b,axis=1)).astype(int))

    Payoffs = (Payoffs_1 + Payoffs_2)/2
    Actualized_Payoffs = np.exp(-r*t)*Payoffs                    #Actualizing the Payoffs
    PRICE = np.mean(Actualized_Payoffs)                           #Averaging the payoffs
(MC method)
    SE = Actualized_Payoffs.std()/np.sqrt(n)

    print("Using the Antithetic Monte Carlo Simulation")
    print(f"Price of the option is = {round(PRICE, 5)}€")
    print(f"Standard Error for the price = {round(SE, 5)}")
    print("")
    print("The 95% confidence interval for the true value of the price :")
    print(f"[{round(PRICE - 1.96*SE,3)}, {round(PRICE + 1.96*SE, 3)}]")
    return(PRICE, SE)

Antithetic_MC_Barrier(PATHS_1, PATHS_2, OPTION_TYPE, BARRIER_TYPE, B, K, R,
T);
```

```
print(time.time() - start)
#%%
interval = np.arange(4, 2500+4,10)
Price_Classic_Barrier = []
Price_Antithetic_Barrier = []

for n in interval:
    PATHS = mult_brownian_paths(T, STEPS, S_0, n, R, SIGMA)
    Price_Classic_Barrier.append(Classic_MC_Barrier(PATHS, OPTION_TYPE,
BARRIER_TYPE, B, K, R, R)[0])

for n in interval:
    PATHS_1, PATHS_2 = mult_brownian_paths_antithetic(T, STEPS, S_0, n//2, R, SIGMA)
    Price_Antithetic_Barrier.append(Antithetic_MC_Barrier(PATHS_1, PATHS_2,
OPTION_TYPE, BARRIER_TYPE,B, K, R, T)[0])

#%% Plots
fig, axs = plt.subplots(2, 1)

axs[0].plot(interval, Price_Classic_Barrier, color = 'darkblue', linewidth = 1)
axs[0].set_title('Classic MC')
axs[0].grid(True)
axs[0].set_xlabel('Number of Paths')
axs[0].set_ylabel('Option Price')

axs[1].plot(interval, Price_Antithetic_Barrier, color = 'darkred', linewidth = 1)
axs[1].set_title('Antithetic MC')
axs[1].grid(True)
axs[1].set_xlabel('Number of Paths')
axs[1].set_ylabel('Option Price')



fig.tight_layout()
```

```
plt.savefig("Barrier MC Convergence Speed for each algos")



plt.figure()
plt.plot(interval, Price_Classic_Barrier, color = "darkblue", label = 'Classic')
plt.plot(interval, Price_Antithetic_Barrier, color = "darkred", label = 'Antithetic')
plt.legend()
plt.xlabel("Number of Paths")
plt.ylabel("Option Price")
plt.grid()

plt.savefig('Barrier Speed Convergence Comparison')

# %%
```

Appendix V – American Options Monte Carlo

```
#%% Settings of the simulation
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import yfinance as yf
from sklearn.linear_model import LinearRegression
from scipy.stats import norm
from MC_functions import *
import time

np.random.seed(2024)

STOCK = "TTE.PA"                                        #TotalEnergie
data = yf.download(STOCK)

stock = data.drop(labels = ['Open', "High", "Low", "Close", "Volume"], axis = 'columns')
stock["Returns"] = stock["Adj Close"] / stock["Adj Close"].shift(1) - 1
stock['Log Return'] = np.log(stock['Adj Close'] / stock['Adj Close'].shift(1))

sigma = stock["Log Return"].std()*np.sqrt(252)                  #Annualize volatility
of Log-Return
print("")


#Parameters of the simulation

R = 0.0390                                          #Risk free rate
SIGMA = sigma                                          #Volatility
S_0 = stock["Adj Close"].iloc[-1]                          #Initial Price
NUMBER_PATHS = 10**4                                   #Number of Simulations
```

```
K = S_0                                             #Strike Price (ATM Option)

T = 60/365

STEPS = 50                                           #Number of year until expiration of
the contract                     #Number of trading day until expiration

OPTION_TYPE = "call"


print("The parameters of the simulation are:")

print("")

print(f"Stock {OPTION_TYPE} option on {STOCK}")

print("")

print(f"Initial Price = {round(S_0, 5)}")

print(f"Annualized Volatility = {round(SIGMA, 5)}")

print(f"Risk free rate = {R}")

print(f"Number of steps = {STEPS}")

print(f"Strike Price = {K}")

print("")


#%% Pricing American Option with Classic MC

# Taken from : https://medium.com/@ptlabadie/pricing-american-options-in-python-
8e357221d2a9

start = time.time()


PATHS = mult_brownian_paths(T, STEPS, S_0, NUMBER_PATHS, R, SIGMA)


def Longstaff_Schwartz_American(paths, option_type, k, r, t, sigma):

   n = paths.shape[0]

   Payoffs = np.zeros_like(paths)                              #Defining a Payoff Matrix


   european_price_exact = Black_Scholes(option_type, paths[0,0], k, t, r, sigma)

   if option_type == "call":                                  #Computing the Payoff at
each nodes of the simulation

      Payoffs = np.maximum(paths-k,0)

   else:
      Payoffs = np.maximum(k-paths,0)

   Actualized_Payoffs = np.zeros_like(Payoffs)

   D = Payoffs.shape[1] - 1

   for t in range(1,D):

      in_the_money =paths[:,t] < k

      X = (paths[in_the_money,t])
      X2 = X*X
      Xs = np.column_stack([X,X2])
      Y = Payoffs[in_the_money, t-1]  * np.exp(-r*t/D)
      model_sklearn = LinearRegression()
      model = model_sklearn.fit(Xs, Y)
      conditional_exp = model.predict(Xs)
      continuations = np.zeros_like(paths[:,t])
      continuations[in_the_money] = conditional_exp

      Payoffs[:,t] = np.where(continuations> Payoffs[:,t], 0, Payoffs[:,t])

      exercised_early = continuations < Payoffs[:,t]
      Payoffs[:,0:t][exercised_early, :] = 0
      Actualized_Payoffs[:,t-1] = Payoffs[:,t-1]* np.exp(-r * T)

   Actualized_Payoffs[:,D-1] = Payoffs[:,D-1]* np.exp(-r * t/D)

   final_cfs = np.zeros((Actualized_Payoffs.shape[0], 1), dtype=float)
   for i,row in enumerate(final_cfs):
      final_cfs[i] = sum(Actualized_Payoffs[i,:])
   PRICE = np.mean(final_cfs)
```

```
    SE = final_cfs.std()/np.sqrt(n)
    print("Using the Longstaff-Schwartz Monte Carlo Simulation")
    print(f"Price of the option is = {round(PRICE, 5)}€")
    print(f"Standard Error for the price = {round(SE, 5)}")
    print("")
    print("The 95% confidence interval for the true value of the price :")
    print(f"[{round(PRICE - 1.96*SE,3)}, {round(PRICE + 1.96*SE, 3)}]")
    return(PRICE, SE)



Longstaff_Schwartz_American(PATHS, OPTION_TYPE, K, R, T, SIGMA)
print(time.time() - start)
# %% Pricing American Options using antithetic MonteCarlo
start = time.time()
PATHS_1, PATHS_2 = mult_brownian_paths_antithetic(T, STEPS, S_0,
NUMBER_PATHS//2, R, SIGMA)


def Antithetic_MC_American(paths_1, paths_2, option_type, k, r, t):
    n = paths_1.shape[0]                                   #Defining a Payoff Matrix


    if option_type == "call":                                    #Computing the Payoff at
each nodes of the simulation
        Payoffs1 = np.maximum(paths_1-k,0)
        Payoffs2 = np.maximum(paths_2-k,0)
    else:
        Payoffs1 = np.maximum(k-paths_1,0)
        Payoffs2 = np.maximum(k-paths_2,0)


    Actualized_Payoffs1 = np.zeros_like(Payoffs1)
    Actualized_Payoffs2 = np.zeros_like(Payoffs2)


    D = Payoffs1.shape[1] - 1
```

```
for t in range(1,D):

    in_the_money_1 = paths_1[:,t] < k
    in_the_money_2 = paths_2[:,t] < k

    X_1 = (paths_1[in_the_money_1,t])
    X_2 = (paths_2[in_the_money_2,t])
    X2_1 = X_1*X_1
    X2_2 = X_2*X_2
    Xs_1 = np.column_stack([X_1,X2_1])
    Xs_2 = np.column_stack([X_2,X2_2])
    Y_1 = Payoffs1[in_the_money_1, t-1]  * np.exp(-r*t/D)
    Y_2 = Payoffs2[in_the_money_2, t-1]  * np.exp(-r*t/D)
    model_sklearn = LinearRegression()
    model_1 = model_sklearn.fit(Xs_1, Y_1)
    model_2 = model_sklearn.fit(Xs_2, Y_2)
    conditional_exp_1 = model_1.predict(Xs_1)
    conditional_exp_2 = model_2.predict(Xs_2)
    continuations_1 = np.zeros_like(paths_1[:,t])
    continuations_2 = np.zeros_like(paths_2[:,t])
    continuations_1[in_the_money_1] = conditional_exp_1
    continuations_2[in_the_money_2] = conditional_exp_2

    Payoffs1[:,t] = np.where(continuations_1 > Payoffs1[:,t], 0, Payoffs1[:,t])
    Payoffs2[:,t] = np.where(continuations_2 > Payoffs2[:,t], 0, Payoffs2[:,t])

    exercised_early1 = continuations_1 < Payoffs1[:,t]
    exercised_early2 = continuations_2 < Payoffs2[:,t]

    Payoffs1[:,0:t][exercised_early1, :] = 0
    Payoffs2[:,0:t][exercised_early2, :] = 0
```

```
        Actualized_Payoffs1[:,t-1] = Payoffs1[:,t-1]* np.exp(-r * T)
        Actualized_Payoffs2[:,t-1] = Payoffs2[:,t-1]* np.exp(-r * T)

    Actualized_Payoffs1[:,D-1] = Payoffs1[:,D-1]* np.exp(-r * t/D)
    Actualized_Payoffs2[:,D-1] = Payoffs2[:,D-1]* np.exp(-r * t/D)

    final_cfs1 = np.zeros((Actualized_Payoffs1.shape[0], 1), dtype=float)
    final_cfs2 = np.zeros((Actualized_Payoffs2.shape[0], 1), dtype=float)
    for i,row in enumerate(final_cfs1):
        final_cfs1[i] = sum(Actualized_Payoffs1[i,:])
    for j,row in enumerate(final_cfs2):
        final_cfs2[j] = sum(Actualized_Payoffs2[j,:])

    final_cfs_antithetic = (final_cfs1 + final_cfs2)/2
    PRICE = np.mean(final_cfs_antithetic)
    SE = final_cfs_antithetic.std()/np.sqrt(n)
    print("Using the Antithetic Monte Carlo Simulation")
    print(f"Price of the option is = {round(PRICE, 5)}€")
    print(f"Standard Error for the price = {round(SE, 5)}")
    print("")
    print("The 95% confidence interval for the true value of the price :")
    print(f"[{round(PRICE - 1.96*SE,3)}, {round(PRICE + 1.96*SE, 3)}]")
    return(PRICE, SE)

Antithetic_MC_American(PATHS_1, PATHS_2, OPTION_TYPE, K, R, T)
print(time.time() - start)
#%% Pricing American Options using European Price Control Variable

start = time.time()
import scipy.stats as stats

PATHS = mult_brownian_paths(T, STEPS, S_0, NUMBER_PATHS, R, SIGMA)
```

```
def Longstaff_Schwartz_American_Control_Variate(paths, option_type, k, r, T, sigma):
    n = paths.shape[0]
    Payoffs = np.zeros_like(paths)                                  #Defining a Payoff Matrix



    if option_type == "call":                                       #Computing the Payoff at
each nodes of the simulation
        Payoffs = np.maximum(paths-k,0)
    else:
        Payoffs = np.maximum(k-paths,0)

    Actualized_Payoffs = np.zeros_like(Payoffs)

    D = Payoffs.shape[1] - 1

    for t in range(1,D):

        in_the_money =paths[:,t] < k

        X = (paths[in_the_money,t])
        X2 = X*X
        Xs = np.column_stack([X,X2])
        Y = Payoffs[in_the_money, t-1]  * np.exp(-r*t/D)
        model_sklearn = LinearRegression()
        model = model_sklearn.fit(Xs, Y)
        conditional_exp = model.predict(Xs)
        continuations = np.zeros_like(paths[:,t])
        continuations[in_the_money] = conditional_exp

        Payoffs[:,t] = np.where(continuations> Payoffs[:,t], 0, Payoffs[:,t])
        exercised_early = continuations < Payoffs[:,t]
        Payoffs[:,0:t][exercised_early, :] = 0
        Actualized_Payoffs[:,t-1] = Payoffs[:,t-1]* np.exp(-r * T)

    Actualized_Payoffs[:,D-1] = Payoffs[:,D-1]* np.exp(-r * t/D)

    final_cfs = np.zeros((Actualized_Payoffs.shape[0], 1), dtype=float)
    for i,row in enumerate(final_cfs):
        final_cfs[i] = sum(Actualized_Payoffs[i,:])

    # Control variate adjustment
    european_option_prices = Black_Scholes(option_type, paths[:,0], k, T, r, sigma)
    Z = final_cfs - european_option_prices
    PRICE = np.mean(Z) + np.mean(european_option_prices)
    SE = np.std(Z) / np.sqrt(n)

    print("Using the Longstaff-Schwartz Monte Carlo Simulation with Control Variate")
    print(f"Price of the option is = {round(PRICE, 5)}€")
    print(f"Standard Error for the price = {round(SE, 5)}")
    print("")
    print("The 95% confidence interval for the true value of the price :")
    print(f"[{round(PRICE - 1.96 * SE, 3)}, {round(PRICE + 1.96 * SE, 3)}]")
    return PRICE, SE

Longstaff_Schwartz_American_Control_Variate(PATHS, OPTION_TYPE, K, R, T, SIGMA)
print(time.time() - start)
# %% Pricing American Option using Control Variate V2
start = time.time()
import scipy.stats as stats

PATHS = mult_brownian_paths(T, STEPS, S_0, NUMBER_PATHS, R, SIGMA)
```

```
def Longstaff_Schwartz_American_Control_Variate(paths, option_type, k, r, T, sigma):
    n = paths.shape[0]
    Payoffs = np.zeros_like(paths)                                   #Defining a Payoff Matrix

    euro_price_exact = Black_Scholes(option_type, paths[0,0], k, T, r, sigma)

    if option_type == "call":                                        #Computing the Payoff at
each nodes of the simulation
        Payoffs = np.maximum(paths-k,0)
        Payoffs_Euro = np.maximum(paths[:,-1]-k,0)
    else:
        Payoffs = np.maximum(k-paths,0)
        Payoffs_Euro = np.maximum(k-paths[:,-1],0)



    Actualized_Payoffs_Euro = np.exp(-r*T)*Payoffs_Euro
    Actualized_Payoffs = np.zeros_like(Payoffs)

    D = Payoffs.shape[1] - 1

    for t in range(1,D):

        in_the_money =paths[:,t] < k

        X = (paths[in_the_money,t])
        X2 = X*X
        Xs = np.column_stack([X,X2])
        Y = Payoffs[in_the_money, t-1]  * np.exp(-r*t/D)
        model_sklearn = LinearRegression()
        model = model_sklearn.fit(Xs, Y)
        conditional_exp = model.predict(Xs)
        continuations = np.zeros_like(paths[:,t])
        continuations[in_the_money] = conditional_exp

        Payoffs[:,t] = np.where(continuations> Payoffs[:,t], 0, Payoffs[:,t])

        exercised_early = continuations < Payoffs[:,t]
        Payoffs[:,0:t][exercised_early, :] = 0
        Actualized_Payoffs[:,t-1] = Payoffs[:,t-1]* np.exp(-r * T)

    Actualized_Payoffs[:,D-1] = Payoffs[:,D-1]* np.exp(-r * t/D)

    final_cfs = np.zeros((Actualized_Payoffs.shape[0], 1), dtype=float)
    for i,row in enumerate(final_cfs):
        final_cfs[i] = sum(Actualized_Payoffs[i,:])

    # Control variate adjustment
    Z = final_cfs - Actualized_Payoffs_Euro + euro_price_exact
    PRICE = np.mean(Z)
    SE = np.std(Z) / np.sqrt(n)

    print("Using the Longstaff-Schwartz Monte Carlo Simulation with Control Variate and
Antithetic")
    print(f"Price of the option is = {round(PRICE, 5)}€")
    print(f"Standard Error for the price = {round(SE, 5)}")
    print("")
    print("The 95% confidence interval for the true value of the price :")
    print(f"[{round(PRICE - 1.96 * SE, 3)}, {round(PRICE + 1.96 * SE, 3)}]")
    return PRICE, SE

Longstaff_Schwartz_American_Control_Variate(PATHS, OPTION_TYPE, K, R, T, SIGMA)
print(time.time() - start)
#%%LS Algo with control and antithetics
```

```
start = time.time()

PATHS_3, PATHS_4 = mult_brownian_paths_antithetic(T, STEPS, S_0,
NUMBER_PATHS//2, R, SIGMA)


def Longstaff_Schwartz_American_Anti_Control_Variate(paths_1, paths_2, option_type, k, r,
T, sigma):
    n = paths_1.shape[0]


    #First Path
    Payoffs_1 = np.zeros_like(paths_1)


    euro_price_exact_1 = Black_Scholes(option_type, paths_1[0,0], k, T, r, sigma)


    if option_type == "call":                                              #Computing the Payoff at
each nodes of the simulation
        Payoffs_1 = np.maximum(paths_1-k,0)
        Payoffs_Euro_1 = np.maximum(paths_1[:,-1]-k,0)
    else:
        Payoffs_1 = np.maximum(k-paths_1,0)
        Payoffs_Euro_1 = np.maximum(k-paths_1[:,-1],0)


    Actualized_Payoffs_Euro_1 = np.exp(-r*T)*Payoffs_Euro_1
    Actualized_Payoffs_1 = np.zeros_like(Payoffs_1)


    D = Payoffs_1.shape[1] - 1


    for t in range(1,D):


        in_the_money_1 =paths_1[:,t] < k


        X = (paths_1[in_the_money_1,t])
        X2 = X*X
        Xs = np.column_stack([X,X2])
        Y = Payoffs_1[in_the_money_1, t-1]  * np.exp(-r*t/D)
        model_sklearn_1 = LinearRegression()
        model_1 = model_sklearn_1.fit(Xs, Y)
        conditional_exp_1 = model_1.predict(Xs)
        continuations_1 = np.zeros_like(paths_1[:,t])
        continuations_1[in_the_money_1] = conditional_exp_1

        Payoffs_1[:,t] = np.where(continuations_1> Payoffs_1[:,t], 0, Payoffs_1[:,t])


        exercised_early_1 = continuations_1 < Payoffs_1[:,t]
        Payoffs_1[:,0:t][exercised_early_1, :] = 0
        Actualized_Payoffs_1[:,t-1] = Payoffs_1[:,t-1]* np.exp(-r * T)

    Actualized_Payoffs_1[:,D-1] = Payoffs_1[:,D-1]* np.exp(-r * t/D)


    final_cfs_1 = np.zeros((Actualized_Payoffs_1.shape[0], 1), dtype=float)
    for i,row in enumerate(final_cfs_1):
        final_cfs_1[i] = sum(Actualized_Payoffs_1[i,:])

    Z1 = final_cfs_1 - Actualized_Payoffs_Euro_1 + euro_price_exact_1



    #Second Path



    Payoffs_2 = np.zeros_like(paths_2)



    euro_price_exact_2 = Black_Scholes(option_type, paths_2[0,0], k, T, r, sigma)

    if option_type == "call":                                          #Computing the Payoff at
each nodes of the simulation
```

```
    Payoffs_2 = np.maximum(paths_2-k,0)
    Payoffs_Euro_2 = np.maximum(paths_2[:,-1]-k,0)
else:
    Payoffs_2 = np.maximum(k-paths_2,0)
    Payoffs_Euro_2 = np.maximum(k-paths_2[:,-1],0)



Actualized_Payoffs_Euro_2 = np.exp(-r*T)*Payoffs_Euro_2
Actualized_Payoffs_2 = np.zeros_like(Payoffs_2)

D = Payoffs_2.shape[1] - 1

for t in range(1,D):

    in_the_money_2 =paths_2[:,t] < k

    X = (paths_2[in_the_money_2,t])
    X2 = X*X
    Xs = np.column_stack([X,X2])
    Y = Payoffs_2[in_the_money_2, t-1]  * np.exp(-r*t/D)
    model_sklearn_2 = LinearRegression()
    model_2 = model_sklearn_2.fit(Xs, Y)
    conditional_exp_2 = model_2.predict(Xs)
    continuations_2 = np.zeros_like(paths_2[:,t])
    continuations_2[in_the_money_2] = conditional_exp_2

    Payoffs_2[:,t] = np.where(continuations_2> Payoffs_2[:,t], 0, Payoffs_2[:,t])

    exercised_early_2 = continuations_2 < Payoffs_2[:,t]
    Payoffs_2[:,0:t][exercised_early_2, :] = 0
    Actualized_Payoffs_2[:,t-1] = Payoffs_2[:,t-1]* np.exp(-r * T)
    Actualized_Payoffs_2[:,D-1] = Payoffs_2[:,D-1]* np.exp(-r * t/D)

    final_cfs_2 = np.zeros((Actualized_Payoffs_2.shape[0], 1), dtype=float)
    for i,row in enumerate(final_cfs_2):
        final_cfs_2[i] = sum(Actualized_Payoffs_2[i,:])

    # Control variate adjustment
    Z2 = final_cfs_2 - Actualized_Payoffs_Euro_2 + euro_price_exact_2



    Z = (Z1 + Z2)*0.5
    PRICE = np.mean(Z)
    SE = np.std(Z) / np.sqrt(n)

    print("Using the Longstaff-Schwartz Monte Carlo Simulation with Control Variate")
    print(f"Price of the option is = {round(PRICE, 5)}€")
    print(f"Standard Error for the price = {round(SE, 5)}")
    print("")
    print("The 95% confidence interval for the true value of the price :")
    print(f"[{round(PRICE - 1.96 * SE, 3)}, {round(PRICE + 1.96 * SE, 3)}]")
    return PRICE, SE

Longstaff_Schwartz_American_Anti_Control_Variate(PATHS_3, PATHS_4,
OPTION_TYPE, K, R, T, SIGMA)
print(time.time() - start)
# %% Convergence of different algos

interval = np.arange(24, 2500+4,10)
Price_LS_American = []
Price_Antithetic_American = []
Price_Control_American = []
Price_Anti_Control_American = []
```

```
for n in interval:
    PATHS = mult_brownian_paths(T, STEPS, S_0, n, R, SIGMA)
    Price_LS_American.append(Longstaff_Schwartz_American(PATHS, OPTION_TYPE, K,
R, T, SIGMA)[0])

for n in interval:
    PATHS_1, PATHS_2 = mult_brownian_paths_antithetic(T, STEPS, S_0, n//2, R, SIGMA)
    Price_Antithetic_American.append(Antithetic_MC_American(PATHS_1, PATHS_2,
OPTION_TYPE, K, R, T)[0])

for n in interval:
    PATHS = mult_brownian_paths(T, STEPS, S_0, n, R, SIGMA)
    Price_Control_American.append(Longstaff_Schwartz_American_Control_Variate(PATHS,
OPTION_TYPE, K, R, T, SIGMA)[0])

for n in interval:
    PATHS_3, PATHS_4 = mult_brownian_paths_antithetic(T, STEPS, S_0, n//2, R, SIGMA)

Price_Anti_Control_American.append(Longstaff_Schwartz_American_Anti_Control_Variate
(PATHS_3, PATHS_4, OPTION_TYPE, K, R, T, SIGMA)[0])

#%% Plots
fig, axs = plt.subplots(2, 2)

axs[0, 0].plot(interval, Price_LS_American, color = 'darkblue', linewidth = 1)
axs[0, 0].set_title('Longstaff MC')
axs[0, 0].grid(True)
axs[0, 0].set_xlabel('Number of Paths')
axs[0, 0].set_ylabel('Option Price')

axs[0, 1].plot(interval, Price_Antithetic_American, color = 'darkred', linewidth = 1)
axs[0, 1].set_title('Antithetic MC')
axs[0, 1].grid(True)
```

```
axs[0, 1].set_xlabel('Number of Paths')
axs[0, 1].set_ylabel('Option Price')

axs[1, 0].plot(interval, Price_Control_American, color = 'orange', linewidth = 1)
axs[1, 0].set_title('European Control MC')
axs[1, 0].grid(True)
axs[1, 0].set_xlabel('Number of Paths')
axs[1, 0].set_ylabel('Option Price')

axs[1, 1].plot(interval, Price_Anti_Control_American, color = 'green', linewidth = 1)
axs[1, 1].set_title('Antithetic + European Control MC')
axs[1, 1].grid(True)
axs[1, 1].set_xlabel('Number of Paths')
axs[1, 1].set_ylabel('Option Price')

fig.tight_layout()
plt.savefig("American MC Convergence Speed for each algos")



plt.figure()
plt.plot(interval, Price_LS_American, color = "darkblue", label = 'Classic')
plt.plot(interval, Price_Antithetic_American, color = "darkred", label = 'Antithetic')
plt.plot(interval, Price_Control_American, color = "orange", label = 'European Control')
plt.plot(interval, Price_Anti_Control_American, color = "green", label = 'Antithetic &
European Control')
plt.legend()
plt.xlabel("Number of Paths")
plt.ylabel("Option Price")
plt.grid()

plt.savefig('American Speed Convergence Comparison')
# %%
```

Appendix VI – Conclusion plots and other code

```
import matplotlib.pyplot as plt
import numpy as np

# Data
option_types = ['European', 'Asian', 'Barrier', 'American']
methods = ['Standard MC', 'Antithetic Variables', 'Control Variables', 'Combined']

pt_ratios = {
    'European': [0.0392, 0.0778, 0.5213, 0.5244],
    'Asian': [0.0299, 0.0782, 1.7967, 3.4033],
    'Barrier': [0.0394, 0.0538, 0, 0],
    'American': [0.3449, 0.6803, 0.1855, 0.3629]
}

# Creating the bar chart
x = np.arange(len(option_types))
width = 0.2

fig, ax = plt.subplots(figsize=(12, 6))
for i, method in enumerate(methods):
    ax.bar(x + i*width, [pt_ratios[opt][i] for opt in option_types], width, label=method)

# Adding labels and title
ax.set_xlabel('Option Types')
ax.set_ylabel('P/T Ratio')
ax.set_title('Efficiency of Variance Reduction Methods Across Different Option Types')
ax.set_xticks(x + width * 1.5)
ax.set_xticklabels(option_types)
ax.legend()
```

```
plt.show()
#%% European Options Payoffs

import numpy as np
import matplotlib.pyplot as plt

K = 100
x = np.linspace(50, 150, 10**3)
ZEROS = np.zeros(10**3)

# Calculate the payoff for a long call option
y_long_call = np.maximum(x-K, ZEROS)
y_short_call = -np.maximum(x-K, ZEROS)
y_long_put = np.maximum(K-x, ZEROS)
y_short_put = -np.maximum(K-x, ZEROS)

#Plotting
fig, axs = plt.subplots(2, 2, figsize=(10, 8))

axs[0, 0].plot(x, y_long_call, color = "darkblue")
axs[0, 0].set_title('Long Call')
axs[0, 0].grid(True)
axs[0, 0].set_facecolor('lightgrey')

axs[0, 1].plot(x, y_short_call, color = "darkblue")
axs[0, 1].set_title('Short Call')
axs[0, 1].grid(True)
axs[0, 1].set_facecolor('lightgrey')

axs[1, 0].plot(x, y_long_put, color = "darkblue")
axs[1, 0].set_title('Long Put')
axs[1, 0].grid(True)
```

```
axs[1, 0].set_facecolor('lightgrey')

axs[1, 1].plot(x, y_short_put, color = "darkblue")
axs[1, 1].set_title('Short Put')
axs[1, 1].grid(True)
axs[1, 1].set_facecolor('lightgrey')

for ax in axs.flat:
    ax.set_xlabel('Stock Price')
    ax.set_ylabel('Payoff')

plt.tight_layout()
plt.savefig("Payoff_European")
plt.show()

# %%
```