MSc 203 in Financial Markets
Paris Dauphine University
Numerical Finance

Pricing of Bermudan Basket Options

May 2023

Authors: **Ahmed El Aaradi, Agathe Hospital, Rhea Stephan**
Professor: Mr. AMOUH

Contents

Chapter 1

Introduction

The objective of this project is to price a European and Bermudan Basket Call Option using Monte Carlo simulation. More than the pricing itself, we aim at minimizing the variance of the Monte Carlo simulations through several methods presented later on.

The basis of this project is thus reliant on the simulation of random numbers. The file *RandomGenerator.py* only contains the mother class of all random generators no matter what distribution it generates. Several distributions and methods have been implemented as follows :

- Uniform numbers : In order to generate normally distributed numbers, we first need to simulate a uniform distribution. We chose to implement the Linear Congruential and the Ecuyer Combined methods. The latter is being used in the whole project as it converges faster to the mean for a given number of simulations and most importantly, it never gives a zero as output while the Linear Congruential method may us a zero (as log function are used in several steps, such output would be a clear issue). It also needs less input parameters than the Linear Congruential method.
  The file *UniformGenerator.py* shows that for 10 000 simulations, the Linear congruential methods has a mean of 0.395 while Ecuyer combined reaches 0.497, closer to the theoretical mean of 0.5 for $\mathcal{U}[0,1]$.

- Discrete Random variables : The basic discrete distribution are implemented such as the Bernoulli, Binomial and Poisson distributions. The file *DiscreteGenerator.py* also plots each distribution to ensure that the pseudo random numbers simulated follow the chosen parameters. These discrete generators are not used later on as we work with continuous generators only.

- Continuous Random Variables : The file *ContinuousGenerator.py* implements the exponential distribution according to 2 methods (Inverse and Rejection Sampling) as well as the Normal distribution. The class 'Pseudo_Random' implements 3 methods to generate pseudo random numbers following a normal distribution : Box Müller, Rejection Sampling and a method based on the Central Limit Theorem (CLT). We use the Rejection Sampling method as default because it shows the strongest convergence for a given number of simulations compared to the other 2. This behavior does not change when modifying the seed of the EcuyerCombined Uniform numbers, this choice seems thus quite robust. The class 'Normale' presented in this file is the mother of 'Pseudo_Random' and 'Quasi_Random' that will be explained later in a file dedicated to it.

Naturally, each method yields different results with a certain convergence and dispersion in the simulated sequences. They may thus have different impacts on the final price and Monte Carlo variance depending on the variance reduction used. We will analyze the results of the combinations of the different methods with the various variance reduction techniques and will also use combinations of variance reduction techniques to come up with a fitted combination that both yields an accurate price and a limited variance.

In the following report, we will first start by explaining the code interface to price European call options and Bermudan call options. Then, we will introduce and analyse the results of the pricing of a European Basket Call Option with Monte Carlo simulations including multiple variance reductions techniques. Third, we will explain and analyse the results of the pricing of Bermudan Basket options with Longstaff-Schwarz algorithm and Monte Carlo simulations including multiple variance reduction techniques. Last, we will conclude based on all the observations.

Chapter 2

Code Interface Explanation

The file *Main.py* is the user interface to access the whole code and choose the product to price, all its parameters and the variance reduction methods. It is organised as follows:

- Function MC_EU_Price_1D : the function used to price a European call option with a single asset, since it is out of the scope of this report we will not discuss it in details.
- Function MC_EU_Price: the function used to price a European Basket call option with Monte Carlo simulations including or not variance reduction techniques.
- Function LS_BMD_Price: the function used to price a Bermudan Basket call option using the Longstaff-Schwarz algorithm with Monte Carlo simulations including or not variance reduction techniques.

2.1 Pricing a European Basket Call Option with Monte Carlo

To price a European Basket call option we use the "MC_EU_Price" function. The function is defined as following:
MC_EU_Price(Spot, Weights, Strike, rate, vol_vector, corr_matrix, Maturity:float, nb_simul, method)
It takes as inputs the following parameters in this particular order:

- Spot: A parameter of type 'list' including the asset prices at t=0, thus with length = number of assets.
  example: [100, 120, 110]
- Weights: A parameter of type 'list' including the weight of each asset in the same order and with same length as the list "Spot".
  example: [0.5, 0.3, 0.2]
- Strike: A parameter of type 'float' as the Strike price.
  example: 110.0
- rate: A parameter of type 'float' as the risk-free rate.
  example: 0.05
- vol_vector: A parameter of type 'list' including the volatility of each asset in the same order and with same length as the list "Spot".
  example: [0.1, 0.2, 0.6]

- corr_matrix: A parameter of type 'list' representing the correlation matrix of size (number of assets × number of assets).
  example: $\begin{bmatrix} 1 & -0.1 & 0.09 \\ -0.1 & 1 & -0.6 \\ 0.09 & -0.6 & 1 \end{bmatrix}$

- Maturity: A parameter of type 'float' as the maturity.
  example: 1.0

- nb_simul: A parameter of type 'int' as the number of Monte Carlo simulations required.
  example: 1_000

- method: A parameter of type 'str' as the method used if you want a variance reduction method or a combination of methods.
  The possible inputs are: 'None', 'Quasi-Random','Antithetic','Control-Variate', 'Quasi-Random + Antithetic', 'Quasi-Random + Control-Variate', 'Antithetic + Control-Variate', 'all-3', 'compare all'.
  The input 'None' means there is no variance reduction, the inputs with a + means it's a combination of two variance reduction methods, the input 'all-3' applies Quasi-Random, Control-Variate and Antithetic variance reduction all together and the input 'compare all' is used to output all of the above options.
  example: 'None'

The function outputs the Monte Carlo option price, the simulation variance, the rate of convergence, the confidence interval and the computation time.
As an example if we run the function with the example inputs above we get the following results:

MC Call on Basket of 3 Assets pricer:	
The Monte Carlo results for the method 'None' are :	
Monte Carlo price :	6.42973
Simulation variance :	11.3723
Rate of convergence :	4.0932
Confidence interval :	(0, 16.973758125209468)
Time :	3.5324

2.2 Pricing a Bermudan Basket Call Option with Longstaff-Schwarz

To price a Bermudan Basket call option with the Longstaff-Schwarz algorithm we use the "LS_BMD_Price" function. The function is defined as following:
LS_BMD_Price(Spot, Weights, Strike, rate, vol_vector, corr_matrix, Maturity, nb_simul, method, nb_execution_dates=10)
It takes as inputs the following parameters in this particular order:

- Spot: A parameter of type 'list' including the asset prices at t=0, thus with length = number of assets.
  example: [100, 120, 110]

- Weights: A parameter of type 'list' including the weight of each asset in the same order and with same length as the list "Spot".
  example: [0.5, 0.3, 0.2]

- Strike: A parameter of type 'float' as the Strike price.
  example: 110.0

- rate: A parameter of type 'float' as the risk-free rate.
  example: 0.05

- vol_vector: A parameter of type 'list' including the volatility of each asset in the same order and with same length as the list "Spot".
  example: [0.1, 0.2, 0.6]

- corr_matrix: A parameter of type 'list' representing the correlation matrix of size (number of assets × number of assets).
  example: $\begin{bmatrix} 1 & -0.1 & 0.09 \\ -0.1 & 1 & -0.6 \\ 0.09 & -0.6 & 1 \end{bmatrix}$

- Maturity: A parameter of type 'float' as the maturity.
  example: 1.0

- nb_simul: A parameter of type 'int' as the number of Monte Carlo simulations required.
  example: 1_000

- method: A parameter of type 'str' as the method used if you want a variance reduction method or a combination of methods.
  The possible inputs are: 'None', 'Quasi-Random','Antithetic','Control-Variate', 'Quasi-Random + Antithetic', 'Quasi-Random + Control-Variate', 'Antithetic + Control-Variate', 'all-3', 'compare all'.
  The input 'None' means there is no variance reduction, the inputs with a + means it's a combination of two variance reduction methods, the input 'all-3' applies Quasi-Random, Control-Variate and Antithetic variance reduction all together and the input 'compare all' is used to output all of the above options.
  example: 'None'

- nb_execution_dates: A parameter of type 'int' as the number of execution dates. It takes the number 10 as a default value.
  example: 10

The function outputs the Longstaff-Schwarz option price, the simulation variance, the basic Monte Carlo option price and the computation time.
As an example if we run the function with the example inputs above we get the following results:

MC Call on Basket of 3 Assets pricer:	
The Longstaff-Schwarz results for the method 'None' are :	
LongStaff Schwartz price for the Bermudan option Call on basket is :	6.946321851678211
LS simulation volatility around price :	6.039659529435315
Monte Carlo price for the European Call option on basket is :	6.429733676801109
Time :	8.5749

Chapter 3

Pricing a European Call Basket Option

3.1 Monte Carlo Simulations with Pseudo-Random numbers and no variance reduction

3.1.1 Monte Carlo Simulation on a single asset

We start by simulating the spot paths of a single asset. In fact, in order to price a call option on a basket of risky assets, we need to simulate spot paths for each underlying asset. We define their value as follows :

$$X_t = x * e^{(r-\frac{\sigma^2}{2})t+\sigma W_t} \quad \text{where } X_0 = x > 0 \tag{3.1}$$

The number of Monte Carlo simulations will be denoted M for the rest of this report.

When considering only one asset, the simulation of its path is made with the BSEuler1D class, found in *New_ SDE.py*. Each simulation done using this class will be stored within the class attribute 'Paths'.
For each Normal distribution method, we use Monte Carlo simulations to estimate the price of a Call on one asset with the following parameters:

r=0.05, sigma=0.1, S_0=100, K=100, T=1, nb_Steps=100, nb_simulations=50 000

	Box Müller	Rejection Sampling	Central Limit Theorem
MC Price	6.74541	6.73376	6.83
Simulation variance	8.0823	8.098	8.1247
Rate of convergence	0.2927	0.2938	0.2958
Confidence interval (99%)	(5.9914, 7.4993)	(5.9769, 7.4906)	(6.0681, 7.5918)
Execution time (minutes)	01:22.2	02:19.0	06:15.4

Table 3.1: Normal distribution generators comparison

The actual Black Scholes price with the same input parameters is 6.805, the functions computing it can be found in the class 'BS_Price' of the file *MonteCarlo.py*. It will allow to compare our call price on a single underlying to the closed form formula given by Black Scholes. This verification is needed to ensure that the N dimension Monte Carlo gives an

accurate price. We however need to highlight the extremely high variance.
The closest price is generated with the CLT method but this method is computationally heavy and thus takes longer to run. The sweet spot when it comes to price accuracy and computation time will be when using the Box Müller Method.

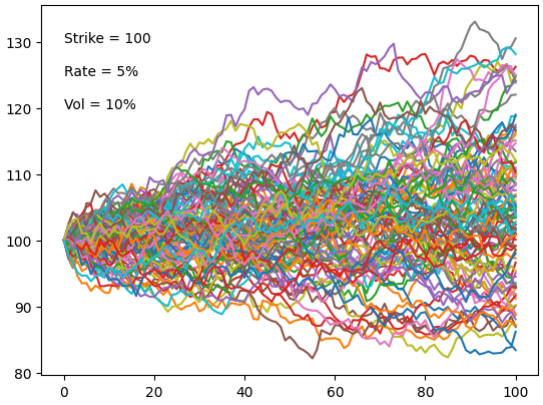

Figure 3.1: Monte Carlo Simulations of Spot prices

3.1.2 Monte Carlo Simulation on a basket of assets

To compute the pricing of a European call on a basket of risky assets, we use multiple assumptions:

- First, we assume that the user may use different spots as starting points for each asset and that the weighting of each asset will be adapted according to its spot. This is quite a strong assumption since the spots are in $\mathbb{R}+$ and we do not have any control over it.
  We have however implemented a check on the product of the weights and the initial spots to make sure that they're not already In The Money.
  We have also constructed another simulation function that takes the simulated paths with varying starting points and normalizes them all to start at 100. This function can be used in case the user does not want to tweak the weights himself and would rather the weights be applied on the performance of the asset rather than its face value. The functions behind these simulations can be found in the class 'BSEulerND' of the file *New_SDE.py*.

- Second, we assume that the rates are constant and identical for all the assets. This is implied since we use the Black-Scholes and M-GAREB framework which will give us false results if we were to have a risky asset's trend different from the risk-free rate.

- Finally, we assume that the volatilities and correlations between assets are constant throughout the simulation period. Another strong assumption but in line with the Black-Scholes framework.

To construct the basket of risky assets, we start by inputting the volatilities of each asset as well as the correlation matrix of the assets. Depending on the covariance matrix resulting from the two inputs, and especially depending on whether it's invertible or not, we will proceed differently:

- If we end up with an invertible covariance matrix, we apply the Cholesky transformation on it to end up with the evolution of the correlated risky assets.
- If the covariance matrix doesn't turn out to be invertible, we check if it's positive semidefinite and start the diagonalization process. In our implementation, we do not find the correct values for the Diagonal or the Orthogonal matrices. And so, we had to resort to the numpy library's diagonalization process to be able to advance in the project.

With X the column vector of asset spot prices, M the number of assets, B the transformed covariance matrix using Cholesky or diagonalization and Y a column vector of random variables that follow a normal distribution $\mathcal{N}(0,1)$ we get the following formula for the spot price at t+1 using the spot prices at t:

$$\underbrace{dX}_{(M\times 1)} = \underbrace{\begin{bmatrix} \mu_1 \\ \cdot \\ \cdot \\ \mu_M \end{bmatrix}}_{(M\times 1)} \underbrace{[dt]}_{(1\times 1)} + \underbrace{B}_{(M\times M)} \times \underbrace{Y}_{(M\times 1)} \times \underbrace{[\sqrt{dt}]}_{(1\times 1)} \tag{3.2}$$

The below simulation is done using the BSEulerND class. Each simulation done using this class will be stored within the class attribute Paths.
The input parameters are as follows:

vol_vector = [0.1, 0.2, 0.6], Spots = [105, 100, 95], corr_matrix = $\begin{bmatrix} 1 & -0.1 & 0.09 \\ -0.1 & 1 & -0.6 \\ 0.09 & -0.6 & 1 \end{bmatrix}$

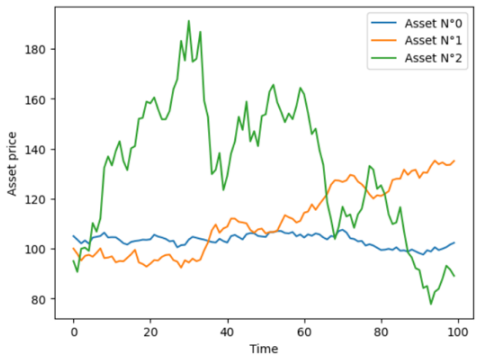

Figure 3.2: Simulation of 3 correlated assets, r=0.05, T=1, nb_steps=100

Now that we have the simulations of the various assets, we need a class that will use these dynamics and the generated paths to give us the actual payoff at maturity.

We will not explain what the class 'Call' does since we deem it to be intuitive (single underlying) and especially since it is not very important in the scope of our project.

As for the multi-dimensional case, we use the class 'CallBasket'. This class will use the generated correlated asset paths to compute the payoff at maturity of a Call option of that basket of risky assets using the following formula:

$$\left(\sum_{i=1}^{n} \alpha_i S_T^i - K\right)^+ \tag{3.3}$$

Now that we have our basic classes, we can move on to the Monte Carlo simulations. The Monte Carlo class works as follows:

- It will use the BSEuler classes to simulate paths (single or multi-dimensional),
- Then use the Call class to get the list of final payoffs,
- And finally use the rate and maturity to compute the final price(by actualizing the payoff to t).

It also computes the rate of convergence, the simulation variance, and the confidence interval using formulas from the course material.

We hereby price a call option on a basket of stocks. The weights are given as an input and the spots are simulated using a Monte Carlo and the classes explained earlier.

The figure 3.2 gives us the price of the basket call for one iteration. Using the same input parameters and weight vector w = [0.3, 0.3, 0.4], we repeat this process for a large number of times and apply the selected weights to each asset path to get the following results:

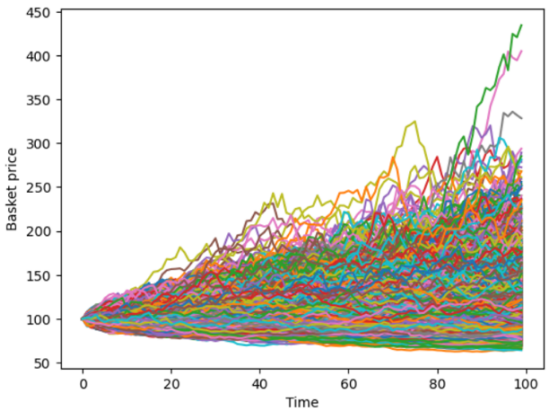

Figure 3.3: Simulation of 3 correlated assets, r=0.05, T=1, nb_steps=100, nb_simulations=10000

The final MC price will of course depend on the entered parameters, especially the volatility of the underlying and most importantly the correlation between the assets considered in the basket. Therefore, using the Box Müller method for the Normal distribution simulation and without using any variance reduction technique we obtain the following results for the parameters explicited above:

	Call Basket
MC Price	6.44103
Simulation variance	11.2058
Rate of convergence	0.5621
Confidence interval (99%)	(4.99316, 7.8888)
Computation time (minutes)	05:08.1

Table 3.2: Results MC simulation Call Basket

3.2 Variance Reduction techniques

In this section, we will try to improve the efficiency of Monte Carlo simulations through variance reduction. Thus, we will implement and analyse the results of three different techniques:Quasi-Random Numbers, Static Control Variate and Antithetic Random Variables.

3.2.1 Quasi-Random Numbers

This method consists in using quasi-random numbers instead of pseudo-random. The difference lies in the dependence between generated numbers, and thus their dispersion on the uniform spectrum. $\Longrightarrow$ Quasi-random samples are not independent from the previous one, it 'remembers' the previous samples and attempts to position itself away from other samples. This behaviour is ideal for obtaining a faster convergence in a Monte Carlo simulation in order to reduce the number of simulations M needed.

- Implementation : We decide to use the Halton sequence to generate our uniform numbers. Depending on the dimension we need, the Van der Corput method is also implemented as it represents the specific case of Halton for 1-dimension sequences. In order to use a Halton sequence to generate normally distributed numbers via the Box Müller method, we need to determine which dimension we need : *Is it better to use 2 numbers from the same sequence (dimension = 1) or the first number of 2 Halton sequences (dimension = 2) ?* We provide elements to answer this question in the 'Halton_Sequence' file by testing each method in terms of normality and final price characteristics (price, volatility and confidence interval).

- Files and classes : The Halton class and the p-adic decomposition can be found in the '*Halton_Sequence.py*' file. All inputs are detailed in the file. The generation of normally distributed quasi-random numbers is made in the same file, in the 'Quasi_Random' class which presents two generators depending on the chosen direction.

- Direction : Looking at the figure 3.4 and the statistical description of both simulations presented in table 3.3, we notice that the 2 dimensional method is better in terms of skew and kurtosis but overall, we observe that the 1-dimension method is unstable and does not converge smoothly when the number of simulations increases. We note that with only 500 simulations, the 2-dimension converges while the 1-dimension is already skewed. We will thus use the 2-dimension method for the rest of our computations.

- Results : This method fails to reduce the variance of our Monte-Carlo simulations and needs a much more important execution time. The lack of variance reduction could be caused by the fact that low discrepancy sequences could behave in an undesired manner and be uneven when assets are correlated. In addition, in practice, it should lead to faster convergence but the execution time is a limit to the efficiency.

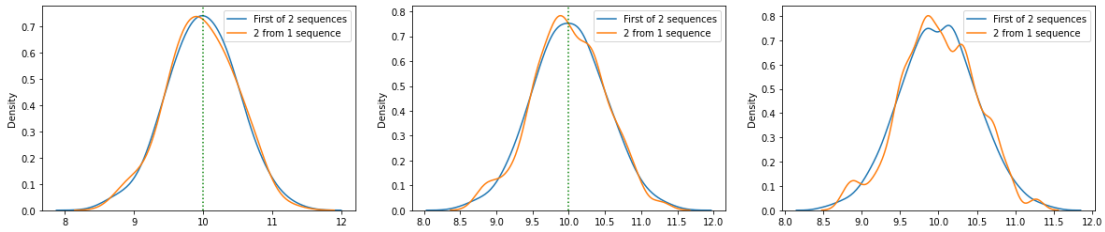

Figure 3.4: $\mathcal{N}(10, 0.5)$ generated from Halton Sequence (1 000, 10 000 and 50 000 simulations)

	1-Dimension	2-Dimension
Mean	10.001	9.9970
Std	0.2453	0.2535
Skew	-0.062	0.0047
Kurtosis	-0.1931	-0.0747

Table 3.3: Halton Sequence directions comparison (50 000 simulations)

3.2.2 Static Control Variate

The static or pseudo control variate method uses a known function to reduce the variance of our MC simulations following the idea that the pricing algorithm miss-prices by a similar amount an option that has a known or unknown price.
This can be expressed with the following formula:

$$X' = X + \beta(Y - \mathbb{E}(Y)) \tag{3.4}$$

Where X' is a better estimation of the option price, X is the estimated option price obtained in the pseudo-random MC simulation with no variance reduction, Y is the control variate, the price of a similar option estimated with MC simulations and $\mathbb{E}(Y)$ is its known expectation.
β is a coefficient that leads to an optimal variance reduction of the control variate and is equal to $\frac{Cov(X,Y)}{Var(Y}$ which corresponds to the least squares estimate of the slope of simple linear regression. However its calculation requires simulations prior to the pricing simulations themselves. The added complexity and computation time would make the Control Variate technique inefficient. Based on empirical studies (Zdravko Botev, Ad Ridder (2017), *Variance Reduction*, School of Mathematics and Statistics), this coefficient is close to -1. Thus, we will consider the following equation:

$$X' = X - Y + \mathbb{E}(Y) \tag{3.5}$$

Naturally, we can see that $\mathbb{E}(X') = \mathbb{E}(X)$ but that Variance(X') = Variance(X-Y) < Variance(X). Thus this technique should reduce the variance without impacting the price and improve our MC simulation efficiency.

- Implementation : In the particular case of a European Basket Call option, the Black-Scholes formula can be used as the known function and thus the control variate. Under certain assumptions, it gives an exact value for the price of the option with

similar characteristics to the estimated option price. Using Jensen's inequality and the idea behind the control variate technique we compute MC simulations with the following elements instead of the basket payoff previously used:

$$Y = \left(e^{\sum_{i=1}^{n} \alpha_i log(S_T^i) - K}\right)^+ \tag{3.6}$$

With n the number of assets, T the maturity, α the vector of the weights of each asset and K the strike. It's expectation is calculated with the BS call price formula with different spot, rate and volatility:

$$\mathbb{E}(Y) = e^{rT} Call_{BS}\left(\prod_{i=1}^{n} {S_0^i}^{\alpha_i}, K, (r - \frac{1}{2}\sum_{i=1}^{n} \alpha_i \sigma_{i.}^2 + \frac{1}{2}\alpha' \sigma\sigma' \alpha, \sqrt{\alpha' \sigma\sigma' \alpha}, T\right) \tag{3.7}$$

With σ the lower triangular covariance matrix and r the risk free rate.

- Files and classes : This variance reduction method impacts the payoff that is used in the MC simulation class(cf. *'Monte_ Carlo.py'* file)).We thus create a new class in the file *Payoff.py* called 'CallBasket_Control' that computes the control variate and the BS price with the new inputs adapted to this method and outputs a modified payout that will then be used in the class 'MonteCarlo' to compute the option price, the variance and other parameters.

- Limitations : The Control Variate we chose to implement requires to set the hypothesis of positive weights with their sum being equal to one. Thus it cannot be used for a portfolio with short positions.
  In addition, as a computational limitation, let's note that the price resulting from the Control Variate method in our case is too far from the price that we got without any variance reduction technique. Thus, there must me an error happening somewhere in the code and for the rest of the report, all conclusions taken from the results of the Control-Variate method may not be accurate.

- Results : This method leads to a sharp decrease in variance and a lower execution time which improves the effectiveness of the Monte Carlo simulations.

3.2.3 Antithetic Random Variables

This method is based on the simulation of 2 paths for each simulation of one underlying asset. While a standard Monte Carlo uses M paths, this methods needs 2*M paths. We first simulate the path of a given asset based on the standard MC procedure. At the same time, we also simulate the path of another asset with the same expectation and variance but with a negative correlation to the first asset. The random spot process χ is then computed as the average of these 2 series of random variables negatively correlated to each other. To ensure their negative correlation, we consider that the second asset decreases when the first one increases. This method can reduce the overall variance only if the second asset if negatively correlated to the first one on average for all simulations.

- Implementation : This variance reduction method impacts the path simulations that are otherwise made in the Euler class. Therefore, in order to use this method, we need to change the random process input of the Monte Carlo class (cf. *'Monte_ Carlo.py'* file)).
  We thus created a new class 'Antithetic_Path' in the 'Variance_Reduction.py' file

that allows to generate 2*M simulations. These N*2 paths are used in the Monte Carlo file which takes this random process as an input where N is the number of assets in the basket.

- Results : This method gives few to no gain in variance reduction and more generally even increases the variance if the number of simulations is below 10 000.

3.3 Gain in variance and required number of simulations

Now that we have explained the various applied methods, we will discuss the obtained results: we'll start with the three variance reduction methods separately to then move on to combinations of 2 or all the methods.

For the incoming simulations, we use the following parameters:
Spots = [100, 120, 110], Weights = [0.5, 0.3, 0.2], Strike = 110.0, Maturity = 1.0, Rate = 0.05, vol_vector = [0.1, 0.2, 0.6], corr_matrix = $\begin{bmatrix} 1 & -0.1 & 0.09 \\ -0.1 & 1 & -0.6 \\ 0.09 & -0.6 & 1 \end{bmatrix}$
Using the main Simulate() function in the Monte Carlo class we obtain the following results nb_simulations=50000:

Var. Red. methods	None	Quasi-Random	Antithetic	Control Variate
Price	6.42584	6.47996	6.4363	4.77863
Simulation std	11.0727	11.4644	11.2152	5.3998
Conf. intervals (99%)	(5.012, 7.840)	(4.965, 7.995)	(4.986, 7.887)	(4.442, 5.115)
Exec. time(seconds)	64.5	1303.1	76.4	64.2

Table 3.4: Price, Simulation Standard Deviation and Confidence Intervals for the basic MC vs MC using the various variance reduction methods.

We notice that the gain in variance for all the variance reduction methods is negative. This means that using the variance reduction methods caused a gain in simulation variance ! This could occur if the variance reduction method is not well-suited for the particular problem being simulated. Additionally, the use of variance reduction techniques may introduce bias into the simulation, which can lead to larger variances in the long run.

In order to test for the gain in the number of iterations, we use the price of the MC as a target. We then compute the standard deviation of the simulation as well as the price as we go through the simulations. As the price stabilizes and gets closer to our target price (within a chosen interval), we stop the simulations. We assume that when the price evolution throughout the simulations' standard deviation gets below a 0.5, we get a price with a good accuracy.
All these simulations have somewhat the same behavior: very agitated at first and stabilizes around 4000 – 5000 simulations.

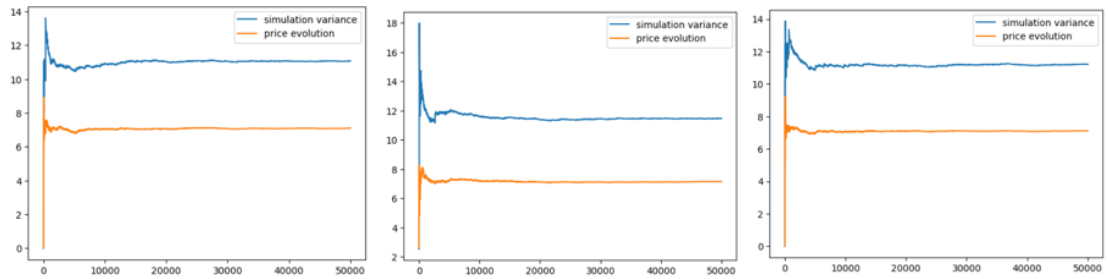

Figure 3.5: Var Reduction methods on basket call - price & simulation variance vs number of simulations (from left to right: No method – Quasi-Random – Antithetic)

However, the function Simulatewstdobjective() makes sure to avoid having to go through additional simulations with little to no added value. As explained above, the moment the standard deviation of the expected price computed using these simulations goes below a fixed threshold, we stop the simulations and assume that the price is stable enough to be used. We obtain the following results:

variance reduction methods	None	Quasi-Random	Antithetic	Control variates
Price	7.00482	7.19763	7.25328	7.00482
Simulation std	10.7622	11.8918	12.1302	10.7622
number of simulations	2160	4077	1181	2160

Table 3.5: Price, Simulation Standard Deviation and Confidence Intervals for the basic MC vs MC using the various variance reduction methods - using the Simulatewstdobjective() function

It is important to note that the prices shown by this new function (that stops simulations as the price stabilizes) and those returned by the basic MC simulation differ!(ref table 3.4-4.3) One can observe how the results from the control variates method are quite different from the rest of the methods, this can be due to a mismatch between the approach and the problem at hand: for example, using a control variates method that is poorly correlated with the function being estimated can actually increase the variance rather than reduce it.

The difference in prices shown in figures 3.4 and 3.5 can be explained first by the fact that the number of simulations is different, so the naturally the result in the end should be different, especially given the difference in the number of simulations: an average around 2700 simulations for the stdobjective function vs 50000 for the MC simulations done earlier.
Now let's compare the various performances of the simple MC vs the combinations of multiple variance reduction methods.

Variance Reduction Combos	Q-R + Antithetic	Q-R + CV	Antithetic + CV
Price	6.53509	4.82193	4.76791
Simulation std	11.5667	5.9143	5.4372
Confidence intervals (99%)	(3.0858, 9.9843)	(3.9192, 5.7245)	(4.0048, 5.5309)
Execution time(seconds)	1337.0	1314.8	74.0

Table 3.6: Price, Simulation Standard Deviation and Confidence Intervals for various variance reduction methods - applying 2 at a time - nb_simulations=10000

Right off the bat, we see that when we use the Control Variates method, the prices tend to differ from the rest of the prices we've obtained till now. Still, we can see that using Quasi-Random numbers with Control Variates yields more variance than if we were to use the Antithetic method along with Control Variates. Therefore the confidence interval is much smaller which implies that the price should be more accurate. Another important point to pay attention to is the staggering difference in computation time between the different pairs of methods: ≈ 1325 seconds for pairs containing the Quasi-Random method vs 74 seconds for the pair without it (≈ 20 times ratio).

Finally, we try to apply the three methods at the same time and we obtain the following:

Variance Reduction Combos	All three methods
Price	4.807
Simulation std	5.8444
Confidence intervals (99%)	(3.9255, 5.6884)
Execution time(seconds)	543.4

Table 3.7: Price, Simulation Standard Deviation and Confidence Intervals for various variance reduction methods - applying all - nbsimulations=10000

We notice that the price is closer to the prices obtained earlier when using the control variates method. We will avoid drawing any conclusions related to the price since we'll have to find out why we do not obtain the same price for CV compared to the other methods. However, we do notice an increase in the simulation variance now that we're using all three methods compared to the Antithetic + CV method. The introduction of the Quasi numbers has once again impacted the execution time: 543.4 seconds vs 74 seconds without the Quasi-Random numbers method.

Chapter 4

Bermudan Basket Option

4.1 Pricing using Longstaff-Schwarz algorithm with Pseudo-Random numbers and no variance reduction

Now onto the pricing of a Bermudan option. Given the nature of this option, at each exercise date t, we will have to execute multiple Monte Carlo simulations to be able to price the option at the set exercise date t.

$$\begin{cases} V_T = f(S_T) \\ V_{T_k} = max[f(S_{t_k}); e^{-r(t_{k+1}-t_k)}\mathbb{E}\left(\frac{V_{t_{k+1}}}{F_{t_k}}\right)] \end{cases} \tag{4.1}$$

However, since we want its current price, we'll have to price it at t=0 considering the various exercise dates, implying that at each exercise date, we will have to perform another Monte Carlo simulation: This will result in an exponential complexity given the fact that one simulated path will contain multiple Monte Carlo simulations. And thus, this approach can very quickly become very heavy and time consuming.

This is why we will be using the Longstaff-Schwarz algorithm as seen in class. The expected payoff at each exercise date will be approximated by a regression.

Brief explanation of the implementation: First, we start by simulating our basket of correlated assets. Using the multiple resulting simulations and the inputs, we construct the value of the basket as well as the option value at each time t. Now, starting from maturity T, we separate the paths where we end up with a positive payoff at t and iterate backwards on t till 0. For the paths where the payoff at t is positive, we compute the continuation value using regressions: And so, if the exercise value is greater than the continuation value, we apply the exercise value as the value of the option for the time t, otherwise, it takes the value of the option at t+1. As for paths where the payoff isn't positive, we discount the option value at t+1. We end up with a (NxM) matrix - with N the number of steps and M the number of simulations – containing the payoff of the option for all simulations throughout the N steps.
As explained above, the LS is here applied due to the context of the Bermudan option. We expect this context to result in higher simulation variance compared to the European option, but also in higher prices for the bermudan given the nature and the options this contract gives to its holder.

Using the following inputs:
Spots = [100, 120, 110], Weights = [0.5, 0.3, 0.2], Strike = 110.0, Maturity = 1.0, Rate = 0.05, vol_vector = [0.1, 0.2, 0.6], corr_matrix = $\begin{bmatrix} 1 & -0.1 & 0.09 \\ -0.1 & 1 & -0.6 \\ 0.09 & -0.6 & 1 \end{bmatrix}$
We obtain the following results:

	Monte Carlo	Longstaff-Schwarz
Price	6.44103	6.94300
Simulation variance	11.2058	7.9011
Computation time (minutes)	02:03.6	04:50.1

Table 4.1: Results LS Bermudan Call basket (nb_execution_dates=2) vs MC simulation European Call Basket - nbSteps=100, nb_simulations=50000

The logical thing to do next is to check the variations of the price of the Bermudan option as the number of execution dates increases. We obtain the following:

Longstaff-Schwarz				
Nb_execution_dates	10	40	70	90
Price	6.95300	7.44546	7.53922	7.53922
Simulation variance	5.77900	10.9620	11.2187	11.2186

Table 4.2: Longstaff-Schwarz vs nb_execution_dates – nbSteps=100, nb_simulations=50000

The results seem to be correct since the price of the Bermudan increases as it should when the number of execution dates increases. And as the number of execution dates becomes equal to the nb of steps, we have a sort of an American option which is considered to be the most expensive option compared to the European and the Bermudan options.

After a more in depth analysis, we find that the resulting variations of the Bermudan option prices are not as smooth as we thought (ref. figure 4.1). We can notice the upward trend on the right side of the graph (dashed line) indicating that as the number of exercise dates increase. However, the start of the graph in unclear and the price of the option keeps jumping to the maximum value throughout the graph.

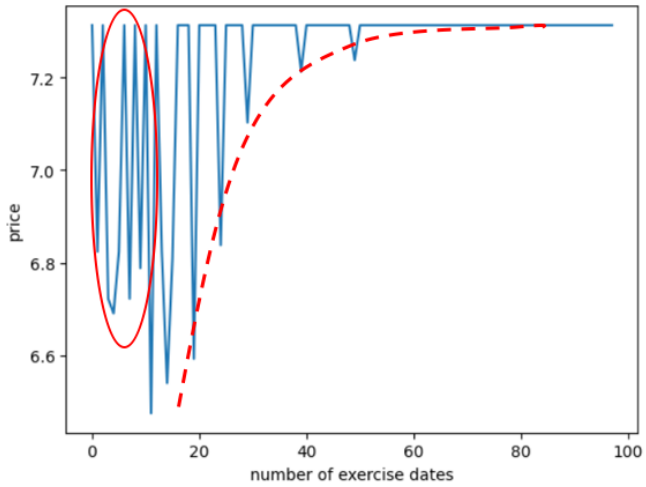

Figure 4.1: Price of a Bermudan option vs number of exercise dates, nb_Steps=100, nb_simulations=1000

These results are very surprising. First because we do not expect the prices of a Bermudan option with less than 10 exercise dates to be superior to Bermudan options with higher number of exercise dates. The second and most surprising of all is the fact that the graph keeps going back to that maximum price value. This creates irregularities and demands that we check back the various classes involved in the computation of these prices.

4.2 Variance Reduction techniques

We now apply the variance reduction methods to the Longstaff-Schwarz approach and take a look at the results. We use the following parameters:
Spots = [100, 120, 110], Weights = [0.5, 0.3, 0.2], Strike = 110.0, Maturity = 1.0, Rate = 0.05, vol_vector = [0.1, 0.2, 0.6], corr_matrix = $\begin{bmatrix} 1 & -0.1 & 0.09 \\ -0.1 & 1 & -0.6 \\ 0.09 & -0.6 & 1 \end{bmatrix}$

We obtain the following results:
First, for the three methods separately vs no method:

Variance Reduction Methods	None	Quasi-Random	Antithetic	Control variates
Price	6.95328	6.94844	6.96589	6.95328
Simulation std	5.76372	5.82810	5.80568	5.76372

Table 4.3: Price & Simulation Standard Deviation for the basic LS vs LS using the various variance reduction methods

The first and biggest surprise is the fact that we observe much smaller variance than expected. This can be explained by the irregularities observed earlier: if the price contains any unexpected behavior, we can expect the variance of the simulation to change greatly depending on the chosen number of exercise dates.
We can also notice the different results from the different methods. Once again, the vari-

ance reduction methods do not bring any positive value to our simulations, except for the Control Variates method whose results seem to be equal to the basic LS simulation. A possible improvement could be to add Control Variates to the least squares regression used in the LS approach.

Then, we combine multiple variance reduction methods and get the following results:

Variance Reduction Combos	Q-R + Antithetic	Q-R + CV	Antithetic + CV
Price	6.94877	6.94844	6.96589
Simulation std	5.8171	5.82810	5.80568

Table 4.4: Price, Simulation Standard Deviation and Confidence Intervals for various variance reduction methods - applying 2 at a time - nb_simulations=10000

When combining quasi-random and antithetic variance reduction techniques, we get a lower variance compared to quasi-random but higher compared to antithetic. Thus suggesting that the combination of both techniques does not improve the efficiency of the simulations. The combinations including the control variate technique lead to results that are the same as without the control variate. This could be explained by the added complexity with the LS algorithm leading to an error in the computation with control variates that seem to be left out of the process. In fact, combining LS algorithm with MC simulations and variance reduction techniques requires to carefully check that each technique works properly and may need to adapt the choice of control variate in accordance.

The below table shows the results when applying the 3 methods of variance reduction together. We can see that the variance at 5.82 is higher than the variance obtained with the LS algorithm without any variance reduction technique that is 5.76. Thus this would mean that either the techniques are less effective when combined with the LS algorithm or that the implementation becomes more complex and needs further checks to make sure that each technique works correctly and that there is no extra bias or error in the estimation process.

LongStaff Schwartz price for the Bermudan option Call on basket is :	6.94877
LS simulation volatility around price :	5.81718
Time (seconds) :	7930.97

Table 4.5: Price, Simulation Standard Deviation and computation time - applying all - nb_simulations=10000

Chapter 5

Results

The aim of this project has been to price Bermudan Call options on a basket of correlated assets, but also to minimize the variance of the Monte Carlo simulations through several methods in order to accomplish faster or more accurate results.

To do so, we started by implementing multiple distributions and methods (Uniform numbers, Discrete Random variables & Continuous Random variables). In the scope of this project, we will only be using the implementations of the Normal distribution (itself using other distributions). And so, we simulated single asset evolutions using the different Normal random variables simulations methods. Comparing the results to the Black and Scholes prices we found that the Box Müller method was the best suited for us: both the speed and accuracy of the method made it a much better choice compared to Rejection Sampling and Central Limit Theorem methods.

Then, we moved to the simulation of correlated assets where we used the Cholesky transformation or the Diagonalization process to obtain a transformation of the covariance matrix that we will later on use to create these correlated assets. Once the basket of correlated asset is created, the construction of the European Call option on such basket isn't complicated, and the price of such an option can easily be found using the Monte Carlo simulations even though they're not the fastest.
This is where we introduce the variance reduction methods. After implementing and analyzing the results, we find that even though the theory seems promising, we were not able to find any positive impact coming from the variance reduction methods in the context of the European Call basket option. All the variance reduction methods do not seem to be adapted to the European Call basket option, the resulting standard deviation of each of the methods is higher than the basic Monte Carlo simulation. Worth noting that there is a lot of confusion around the results of the Control Variate method and so, we do not draw any conclusions from this method. After finding that all of these methods do not really reduce variance, we decide to take the other approach: comparing the number of simulations for the price to stabilize. We end up fixing a threshold on the standard deviation of the vector containing all the prices simulated by the simulations, and as soon as the rolling standard deviation on this vector goes below the fixed threshold, we stop the simulations and assume that the price is stable and accurate. Using this approach, we get different prices (≈ 0.5 difference) but also different standard deviations and most importantly a number of simulations much smaller than 50000 used for the MC simulations. We find in short that the Antithetic method is quite efficient but nowhere near the basic Monte Carlo simulation.

When using different combinations of these methods, we observe that the CV method impacts the price greatly which makes it impossible for us to interpret the price. However, we do find that the normal CV presents less standard deviation vs when using Quasi random number + CV or when using Antithetic method + CV. When we're not using CV, the Quasi-Random numbers method impacts greatly the computation time, and when used with the Antithetic method, doesn't show any gain in standard deviation!

Now that we have applied our different methods on a European Call basket option, we move to a Bermudan Call basket option. We were first stroke with the difference in the standard deviation resulting from the two used methods: MC vs LS. After analyzing the results from the LS algorithm, we find that the results are not very consistent with the number of execution dates. When applying variance reduction on LS and the Bermudan Call option basket, we notice that there is once again no impact of these methods on the simulation's standard deviation. Regarding the gain in the number of simulations, the LS algorithm is based on regressions(matrices containing all of the simulations) and so, given the nature of this algorithm, we couldn't find a way to implement a tool that will compute the gain in the number of simulations.

Through this project, we were able to price a Bermudan Call basket option. This option, as expected, was pricier than the European option, and its price increase with the number of exercise dates. When the number of exercise dates was set to the maximum, we obtained the highest price for all the Bermudan options using the same parameters. This price represents or can be considered close to the price of an American option. The results are as follows:

	European	Bermudan	American
Price	6.44103	6.95300	7.53922

Table 5.1: Prices of the different Call basket options: European, Bermudan(10 exercise dates) & American(nb exercise dates = nb steps) - nb_simulations=50000

Chapter 6

Annex

6.1 Table listing files, classes and functions in code

Code elements		
File name	Classes	Function
ContinuousGenerator.py	ContinuousGenerator	__init__
	Exponential	__init__, Generate
	Normale	__init__, Generate
	Pseudo_Random	__init__, Generate
DiscreteGenerator.py	DiscreteGenerator	__init__
	HeadTail	Generate
	Bernoulli	__init__, Generate
	Binomial	__init__, Generate
	Poisson	__init__, Generate
Halton_Sequence.py	PAdicDecomposition	__init__, __getitem__, Get_Size, Get_Decomp
	Prime	__init__, is_prime, __call__
	Halton	__init__, __call__
	Quasi_Random	__init__, Generate, Generate_bis
Main.py		MC_EU_Price_1D, MC_EU_Price, LS_BMD_Price
Matrix_decomp.py	Matrix	__init__, to_list, Transpose, isSymmetric, is_invertible, matrix_product, identity, Determinant, cofactor, concat, inverse_matrix
	Mx_decomp	__init__, Decompose, eigenvalues, diagonalize, IsPositiveDefinite

File name	Classes	Function
MonteCarlo.py	MonteCarlo	__init__, Simulate, Simulate_w_variance_objective, isArbitrage
	LS	__init__, Simulate
	BS_Price	__init__, d1, d2, cdf, EU_Call, EU_Put
New_SDE.py	SinglePath	__init__, GetTimeStep, AddValue, GetValue, Get_Values, Get_Times
	RandomProcess	__init__, GetPath, Simulate
	BlackScholes1D	__init__
	BSEuler1D	__init__, Simulate
	BSEulerND	__init__, Simulate, Simulate_perf
	Brownian1D	__init__, Simulate
	BrownianND	__init__
Payoffs.py	payoff	__init__, simulate_payoff, add_end_values
	Call	__init__, simulate_payoff, add_end_values
	CallBasket	__init__, simulate_payoff, add_end_values, add_value_to_last_path
	Call_Control	__init__, simulate_payoff, add_end_values
	CallBasket_Control	__init__, simulate_payoff, add_end_values
	Put	__init__, simulate_payoff, add_end_values
RandomGenerator.py	RandomGenerator	__init__,Generate, Mean, Variance, Testmean, TestVar
UniformGenerator.py	UniformGenerator	__init__
	PseudoGenerator	__init__, seed
	LinearCongruential	__init__, Generate
	EcuyerCombined	__init__, generator1, generator2, Generate
Variance_Reduction.py	Antithetic_Path_1D	__init__, Simulate
	Antithetic_Path_ND	__init__, Simulate