Asian options pricing with Monte-Carlo methods

Thomas AHN, Tristan AMADEI, Thibaut METZ, Thomas SALOMON

April 2023

Contents

Introduction

The aim of this project is to compare different pricing methods for an Asian option. Comparisons will be made in terms of MSE, CPU time and (empirical) variance of estimators. The pricing methods compared in this project are:

- Monte-Carlo standard (MC)
- Multi-Level Monte-Carlo (MLMC)
- Randomized Quasi Monte-Carlo (RQMC)
- Multi-Level Randomized Quasi Monte-Carlo (QMC MLMC)

Please note that the notebooks provided with this project may take some time to run, as the code has been run on a different environment in order to benefit from increased computing power.

1 Asian options pricing

1.1 Prerequisites

An Asian option is a type of option whose payoff depends on the average price of an underlying asset over a given period, rather than the price of the asset on a specific date. The average price is often calculated over a period of a week, a month or a quarter.

The payoff of an Asian option is generally expressed as the arithmetic mean of the underlying asset prices over the reference period, minus the option's strike price. If the average price of the underlying asset is higher than the strike price, the option holder receives a payment corresponding to the difference between the average price and the strike price. If the average price of the underlying is lower than the strike price, the option expires worthless. The price of an Asian option can be calculated in different ways, depending on the terms and conditions of the option defining the payoff.

In this project, we consider an Asian call option whose price form is adopted as follows:

$$C = \mathbb{E}\left[e^{-rT} \cdot max\left(\frac{1}{k}\sum_{i=1}^{k} S(t_i) - K, 0\right)\right]$$

with: - $t_0 = 0 < t_1 < ... < t_k = T$
- $r > 0$
- $K > 0$ fixed
- S: a process defined on $[0, T]$

1.2 Monte-Carlo simulations for pricing

The price of an Asian option can be evaluated using Monte Carlo simulations. This method involves simulating several price paths for the underlying asset. For each simulated trajectory, an estimate of the option price is made, and the final price is obtained by taking the average of the estimates. Several models can be used to generate trajectories for the underlying asset. In this project, the process adopted for the underlying's trajectory is the Cox-Ingersoll-Ross(CIR) process.

2 Simulation of the price of the underlying

2.1 Simulation of a Cox-Ingersoll-Ross (CIR) process

The CIR process is a stochastic process defined by the following stochastic differential equation: $dS_t = a(b - S_t)dt + sigma\sqrt{S_t}dW_t$ with W_t a Brownian motion.

Using the Euler-Maruyama method to obtain a numerical solution to a stochastic differential equation:

For $i \geq 0$, $S_{t_{i+1}} - S_{t_i} = a(b - S_{t_i})(t_{i+1} - t_i) + \sigma\sqrt{S_{t_i}}(W_{t_{i+1}} - W_{t_i})$ with $t_i = i\frac{T}{n}$

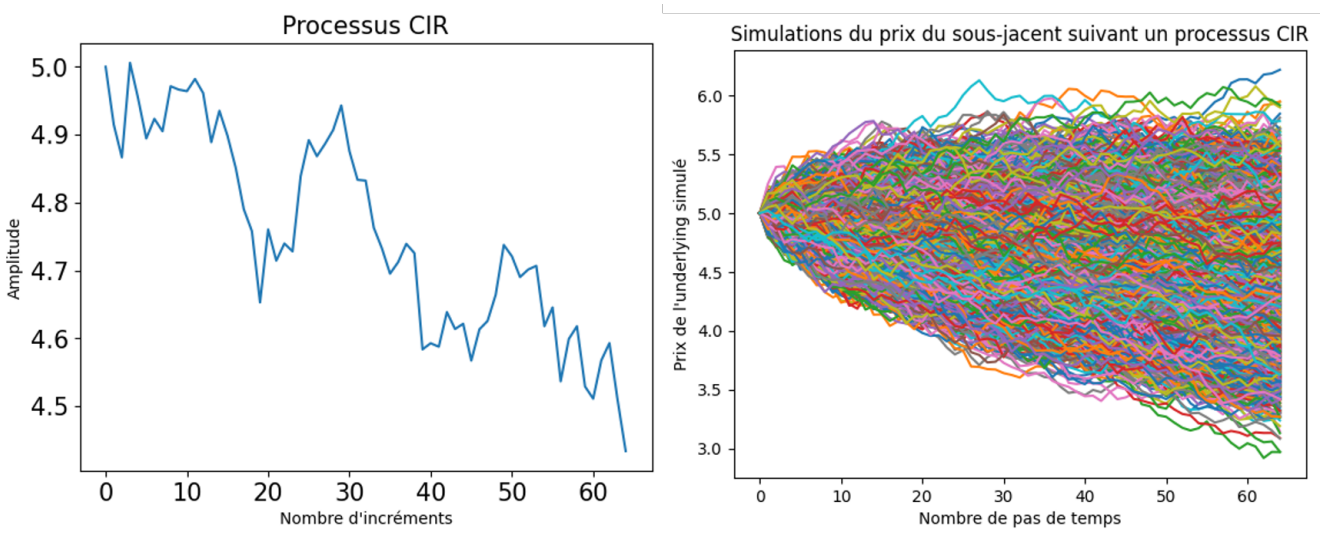

Figure 2.1: CIR process and simulations of the underlying price following a CIR

2.2 Simulation of Brownian increments

Simulating the increments of a Brownian motion involves generating a random movement for the underlying price and repeating this simulation many times. First, we define a discrete time step. Consider the time step $\frac{T}{n}$. At each time step, if the underlying price follows a Brownian motion, then it moves according to a generated random motion distance, which is calculated using a normal probability distribution centered on zero and of variance $\frac{T}{n}$.

So to simulate N increments of a Brownian motion with the standard Monte Carlo method, simply simulate N times a $\mathcal{N}(0, 1)$ and multiply each simulation by $\sqrt{\frac{T}{n}}$ because: $W_{t_{i+1}} - W_{t_i} \sim \mathcal{N}(0, \frac{T}{n})$

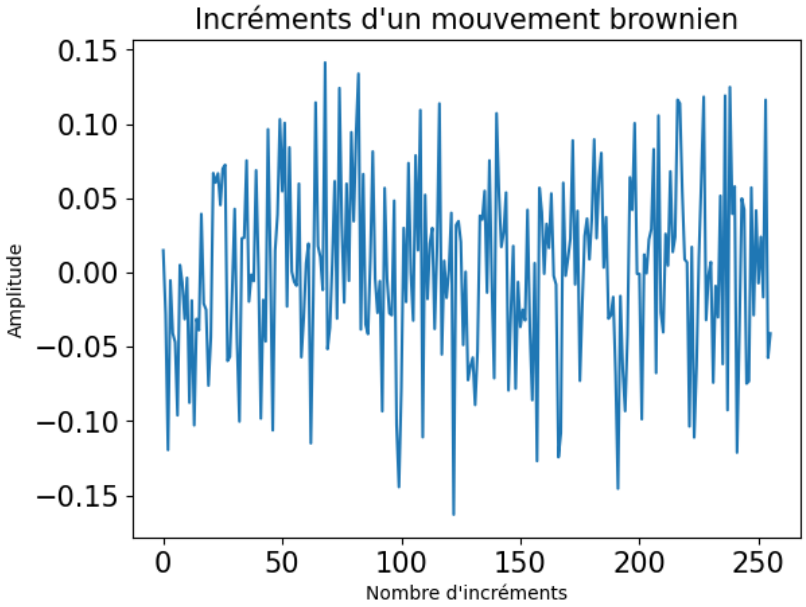

Figure 2.2: Incremental Brownian motion

In this project, we simulated Brownian increments in two different ways. The first uses standard Monte-Carlo simulations of a $\mathcal{N}(0,1)$, while the second uses Randomized Quasi Monte-Carlo (RQMC).

2.2.1 Standard Monte-Carlo

This method consists in drawing N samples from a $\mathcal{N}(0,1)$ with numpy.random.normal, for example, and multiplying each sample by $\sqrt{\frac{T}{n}}$. This is what we've implemented with the function named sim_dW.

2.2.2 Randomized Quasi Monte-Carlo

The Quasi Monte-Carlo method is similar to the standard Monte-Carlo method, but instead of generating random samples, it uses deterministic point sequences (a Sobol sequence in our case) to sample the search space. The aim of this technique is to better cover the interval [0, 1] during simulations, and thus reduce the variances of the estimators.

This sequence of points can then be randomized using the following formula:

$$\forall n \in \mathbb{N}, v_n = (u_n + w_n) \mod 1$$

with (u_n) a sequence of uniforms simulated by the Quasi Monte-Carlo method, a Sobol sequence for example, and $w_n \sim \mathcal{U}(0,1)$. This adds an element of randomness to the simulated uniforms.

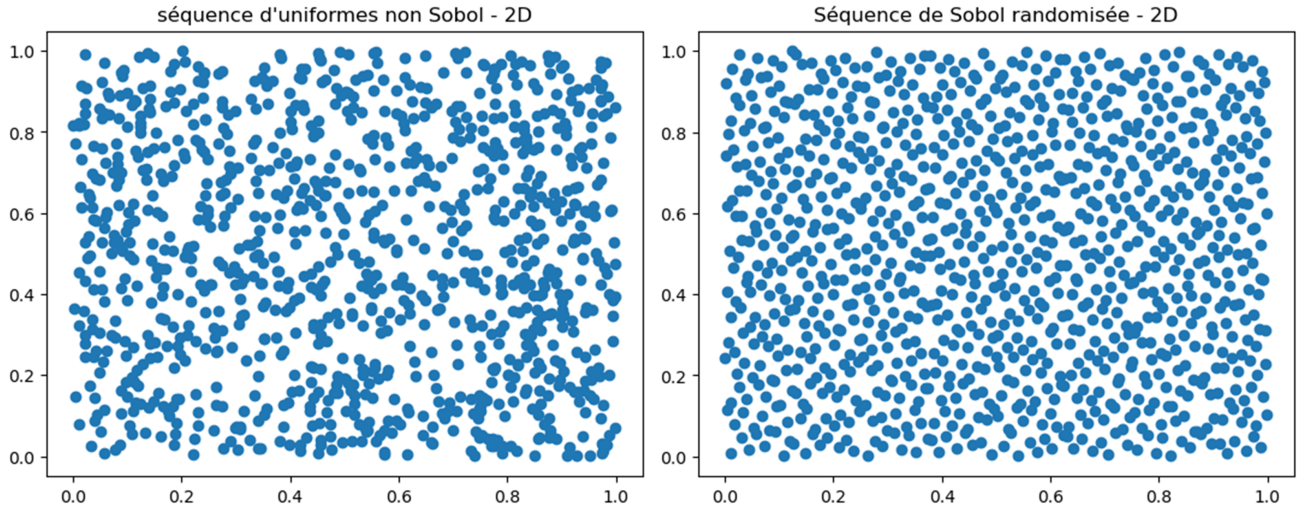

Figure 2.3: Comparison between a sequence of uniforms and a sequence of Sobol - 2D

There are several Randomized Quasi Monte-Carlo methods for simulating samples of a $\mathcal{N}(0,1)$. In our project, we used three of them:

0ccept-Reject method 0ox-Muller method 0nversion method

Accept-Reject method

The most intuitive (and mathematically safest) method in our case was the accept-reject one. The idea was to use the density function of Laplace's law of zero mean and scale parameter 1, whose formula is as follows:

$$\forall x \in \mathbb{R}, g(x) = \frac{1}{2}\exp(-|x|)$$

We can simulate realizations of this law by noting that $X = \log(1 - 2|U|)$ follows our Laplace law when $U \sim \mathcal{U}([-\frac{1}{2}, \frac{1}{2}])$. We can then simulate realizations of a $\mathcal{N}(0,1)$ law as follows:

1. Simulate x according to Laplace
2. Let u_i be a realization of $\mathcal{U}([0,1])$ simulated according to a randomized Sobol sequence.
3. If $u_i \leq \frac{f(x)}{Mg(x)}$, we accept x as a Gaussian simulation.
4. Otherwise we reject x, consider $i = i + 1$ and start again from 1

In this algorithm, f is the density of $\mathcal{N}(0,1)$ and $M = \sqrt{\frac{2e}{\pi}}$ is the optimal value for accepting simulations. The acceptance rate is $\frac{1}{M} \approx 0.76$. To simulate N normal realizations, we need to set up a randomized Sobol sequence of NM uniform samples. In practice, we set up a sequence of dimension $1.1 \times N \times M$ to make sure we had enough. This method works perfectly well for simulating the desired normals, but is very time-consuming. Applying it to our RQMC algorithm for calculating the option price multiplies the execution time by 10! So we looked for other ways to reduce computation time.

Box-Muller Method

The reason we initially discarded Box-Muller is that this method requires two *independent* uniforms to simulate two independent centered reduced Gaussians. But by choosing the uniforms in a Sobol sequence, we lose the mutual independence of these realizations.

However, another method is to simulate a randomized Sobol sequence of N/2 points in 2 dimensions. Thus, for any point $u_i = \begin{pmatrix} u_{i_1} \\ u_{i_2} \end{pmatrix}$, u_{i_1}et u_{i_2} are independant. We can then use the Box-Muller method to simulate our Gaussians:

$$\forall i \in \{1, ..., N\}, \begin{cases} X = \sqrt{-2\log(U_{i_1})} \cdot \cos(2\pi U_{i_2}) \\ Y = \sqrt{-2\log(U_{i_2})} \cdot \cos(2\pi U_{i_1}) \end{cases}$$

We use this version of Box-Muller, and not the improved version, because we need to know the size of the randomized Sobol sequence to be simulated beforehand, and so we wish to avoid the possible rejections of the second Box-Muller method.

Inversion method

If the quantile function of a probability law X is known, it is possible to simulate realizations of this law as follows:

$$X = F^{-1}(U),\ U \sim \mathcal{U}([0, 1])$$

In our case, the quantile function of a Gaussian distribution has no analytical expression. However, it is possible to approximate it, as was done in the Python module *scipy.stats*. This method is less mathematically sound than the previous two, due to the quantile function approximation, but it is very fast.

Comparison of the different methods

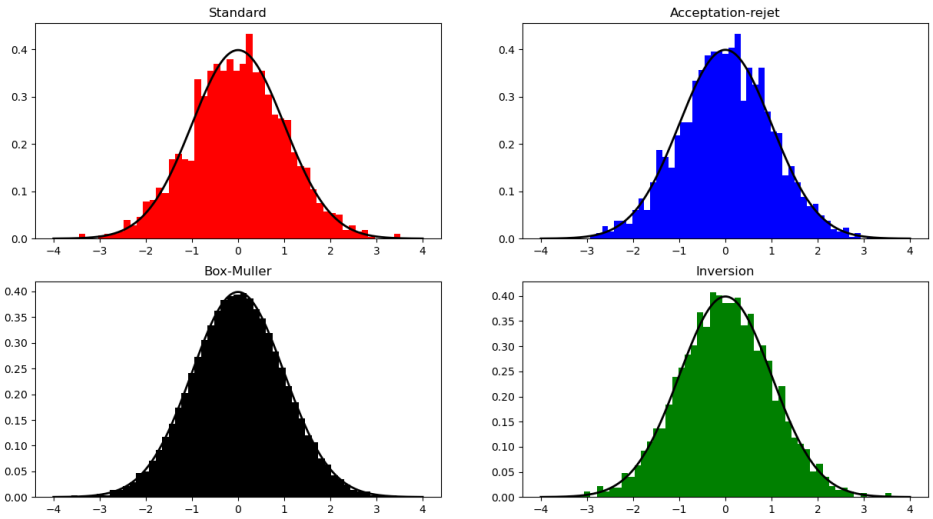

Figure 2.4: Comparison of Quasi Monte-Carlo methods for normal distribution simulations; with 1000 simulations

In the rest of our study, we use the Box-Muller method to implement Quasi Monte-Carlo (RQMC) and Multi-Level Quasi Monte-Carlo (QMC MLMC).

3 Option price approximation and Mean Square Error calculation

3.1 Standard Monte-Carlo

This method involves the simulation of several CIR processes to represent different trajectories of the underlying price. The increments of Brownian motion required for this simulation are obtained by standard Monte Carlo methods, as explained above. The number of simulations of the underlying's trajectories and the step size adopted will be explained below.

3.2 Multi-Level Monte-Carlo

3.2.1 Method presentation

The Multi-Level Monte-Carlo (MLMC) method is based on the idea that a more accurate estimate of a quantity of interest can be obtained by using simulations at several levels of granularity. In the case of the Asian option, the quantity of interest is the option price, which depends on the average price of the underlying asset over a certain period.

To estimate the Asian option price, the MLMC method uses two levels of granularity, determined by the values of parameters M and l in the code. The "coarse" granularity level corresponds to a longer time period, where the underlying asset is simulated with a coarser discretization (i.e. with a larger time step). The "fine" granularity level corresponds to a shorter time period, where the underlying asset is simulated with a finer discretization (i.e. with a smaller time step).

For each level of granularity, the MLMC method performs a large number of simulations to estimate the average price of the underlying asset over the corresponding period. It then uses these estimates to calculate an estimate of the Asian option price. More precisely, the option price estimate is calculated as the difference between the average price estimated from the fine-level simulations and the average price estimated from the coarse-level simulations.

Let C be the option price and P_l its approximation at level l, i.e. its approximation with a discretization of step $h_l = \frac{1}{M^l}$. By telescopic sum:

$$\mathbb{E}[P_L] = \mathbb{E}[P_0] + \sum_{l=1}^{L} \mathbb{E}[P_l - P_{l-1}]$$

The idea behind MLMC is to independently estimate each of the right-hand side expectations in a way that minimizes the overall variance for a given computational cost. Let $\hat{Y}_0$ be an estimator of $\mathbb{E}[P_0]$ using N_0 samples and $\hat{Y}_l$ for $l > 0$ an estimator of $\mathbb{E}[P_l - P_{l-1}]$ using N_l samples.

$$\hat{Y}_l = \frac{1}{N_l} \sum_{i=1}^{N_l} (P_l^i - P_{l-1}^i)$$

Let $\hat{Y}$ be the MLMC estimator of the option price.

$\hat{Y} = \sum_{l=0}^{L} \hat{Y}_l = \sum_{l=0}^{L} \frac{1}{N_l} \sum_{i=1}^{N_l} (P_l^i - P_{l-1}^i)$

$\mathbb{V}[\hat{Y}] = \sum_{l=0}^{L} \mathbb{V}[\hat{Y}_l]$ by independence, with: $\mathbb{V}[\hat{Y}_l] = \frac{1}{N_l} V_l$ où $V_l = \mathbb{V}[P_l - P_{l-1}]$

If the N_l, $l \in \{0, .., L\}$, are optimally calculated with formula (2) (discussed in the next section), then it can be shown that for a fixed ϵ error term:

$$\mathbb{V}[\hat{Y}] = \sum_{l=0}^{L} \mathbb{V}[\hat{Y}_l] \leq \frac{1}{2}\epsilon^{-2}$$

The MLMC method therefore makes it possible to reduce the variance and set it arbitrarily low, by making ϵ tend towards 0. However, this presents us with a trade-off, as the MLMC computation time increases quadratically as ϵ decreases.

Despite this, we can easily prove that the MLMC estimator remains unbiased:

$$\begin{aligned}
\mathbb{E}[\hat{Y}] &= \sum_{l=1}^{L} \mathbb{E}[\hat{Y}_l] \\
&= \sum_{l=1}^{L} \mathbb{E}[P_l - P_{l-1}] \\
&= \mathbb{E}[P_L] - \mathbb{E}[P_0] \\
&= \mathbb{E}[P_L]
\end{aligned}$$

Thus, the MLMC estimator is ***unbiased***.

3.2.2 MLMC parameters tuning

M parameter

In this project, we have chosen to set M equal to 4 insofar as Michael B. Giles chooses this value for his numerical applications in his article. Indeed, this value of M gives most of the advantages of a higher value, but at the same time M is small enough to give a reasonable number of levels from which to estimate the bias.

L parameter

The parameter L was chosen so as to take the smallest possible L (to reduce CPU time) such that the estimator $\hat{Y}_L$ is convergent. The criterion used (sufficiently small bias) is as follows:

$$\left| \hat{Y}_L - \frac{\hat{Y}_{L-1}}{M} \right| < \frac{1}{\sqrt{2}}(M^2 - 1)\epsilon \qquad (1)$$

with ϵ the chosen error term.

N_l parameter

The selection of this parameter is crucial, as an optimal choice of N_l reduces the variance.

As explained in the article and in the course, there is a simple method which consists in choosing, for each value of l, N_l proportional to $\sqrt{V_l h_l} = \sqrt{\frac{V_l}{M^l}}$. This choice is demonstrated in the appendix.

Dans son article, Michael B. Giles propose une expression précise de N_l qui est également proportionnelle à $\sqrt{V_l h_l}$, mais qui fournit en plus l'expression du coefficient de proportionnalité. Nous avons donc choisit d'utiliser la formule suivante proposée par Michael B. Giles :

$$N_l = 2\epsilon^{-2}\sqrt{V_l \cdot h_l}\left(\sum_{l=0}^{L}\sqrt{\frac{V_l}{h_l}}\right) \qquad (2)$$

Once these parameters have been optimally chosen, we can implement the following algorithm:

1. Set L = 0
2. Estimate V_L with $N_L = 10^4$.
3. For $l \in [0, L]$ calculate the optimal N_l with Equation (2)
4. For each N_l:
   - If $N_l^{new} \geq N_l^{former}$: simulate a number of new trajectories of the underlying equal to à $N_l^{new} - N_l^{former}$
   - If $N_l^{new} < N_l^{former}$: delete a number of trajectories of the underlying equal to à $N_l^{former} - N_l^{new}$
5. - If $L \geq 2$: Test convergence with Equation (1)
   - If $L < 2$ or if there is no convergence: set L = L + 1 and go back to step 2

3.3 Randomized Quasi Monte-Carlo

In the same way as the standard Monte Carlo method, this method involves the simulation of several CIR processes to represent different underlying price trajectories. On the other hand, the increments of Brownian motion required for this simulation are obtained by Randomized Quasi Monte Carlo using acceptance-rejection, Box-Muller or inversion. As explained above, in our study we have chosen to implement the Box-Muller method combined with a 2D Sobol sequence.

3.4 Mean Squared Error Computation

The Mean Squared Error (MSE) is defined as : $MSE = \sum_{i=1}^{n}(C - \hat{Y}_i)^2$ avec :

- C: the "true" value of the option
- Y_i: approximation of the price of option n°i
- n: the number of option price approximations performed

For an Asian option whose expiry date has not yet been reached, it is impossible to know its "true value", since the option price depends on the trajectory of the underlying's price. However, this underlying price trajectory cannot be known with certainty in the future. So, to calculate the MSE, we need to know this "true value". To do this, we ran a standard Monte Carlo simulation with a very large number of simulations. Based on the law of large numbers, we can consider the approximation thus obtained to be the true target value C. We then use this approximation, calculated on a very large sample, as the true value to estimate the MSE of the various methods.

Thus, we looked for the value of C, considered as the true value in the rest of the project, by carrying out 10^6 standard Monte-Carlo simulations.
We got C= 0.6797012714083033.

4 Comparison of price approximations in terms of MSE and CPU time

As presented in the article, the discretization step adopted for the standard Monte Carlo method corresponds to the finest step of the Multi-Level Monte Carlo method (l=L), i.e. $h_L = \frac{1}{M^L}$. To make our comparisons, the number of simulations N_{sim} of the underlying adopted for the standard Monte Carlo method is equal to the sum of simulations N_l for $l \in [0, L]$ Multi-Level Monte-Carlo method:

$$N_{sim} = \sum_{i=1}^{L} N_l$$

The results are obtained with the following values: $K = 4, S_0 = 5, r = 0.05, \sigma = 0.2, a = 0.15, b = 1, M = 4$.

4.1 Multiprocessing

Calculating different estimators is costly, especially when it comes to reducing the ϵ error tolerance or increasing the number of simulations required. So, to enable us to test several scenarios, we have introduced multiprocessing - parallelization in other words - into our code. For example, if we want to calculate the variance of Monte Carlo estimators for N_{sim} simulations, this multiprocessing allows us to calculate $N_{estimators}$ in parallel with the same number of simulations; this then allows us to calculate the empirical variance of these estimators, the MSE, etc.

4.2 Comparison of MC Standard and MLMC

In this section, we make comparisons between the standard Monte Carlo (MC) method and Multi-Level Monte Carlo (MLMC), whose Brownian motion increments are simulated by the standard Monte Carlo method as explained in 2.2.1.

4.2.1 MLMC with non optimal parameters

To compare these two methods, we varied the number N_{sim} of simulations performed to construct our Monte Carlo estimator. For the MLMC estimator, we took this number N_{sim} and divided it uniformly to build each level, with a final level L = 3.

Looking at the figures below, we can see that the trajectories follow the same trend. As the number of simulations increases, variance and MSE decrease, but this leads to an increase in computation time, which is not surprising. However, it is interesting to focus on the values presented in these graphs. Despite the challenges associated with the implementation of the MLMC method, we find a higher MSE and variance for the MLMC method than for the standard MC method. So, to get the best performance from the MLMC method, it's crucial to use the optimal N_l. Furthermore, with the same number of simulations N_{sim} as the standard MC method, the MLMC method is significantly faster than the standard MC method. This difference in execution time is explained by the fact that many simulations in MLMC are

performed with a larger step size, whereas standard Monte Carlo uses the same fine step size for all its simulations.

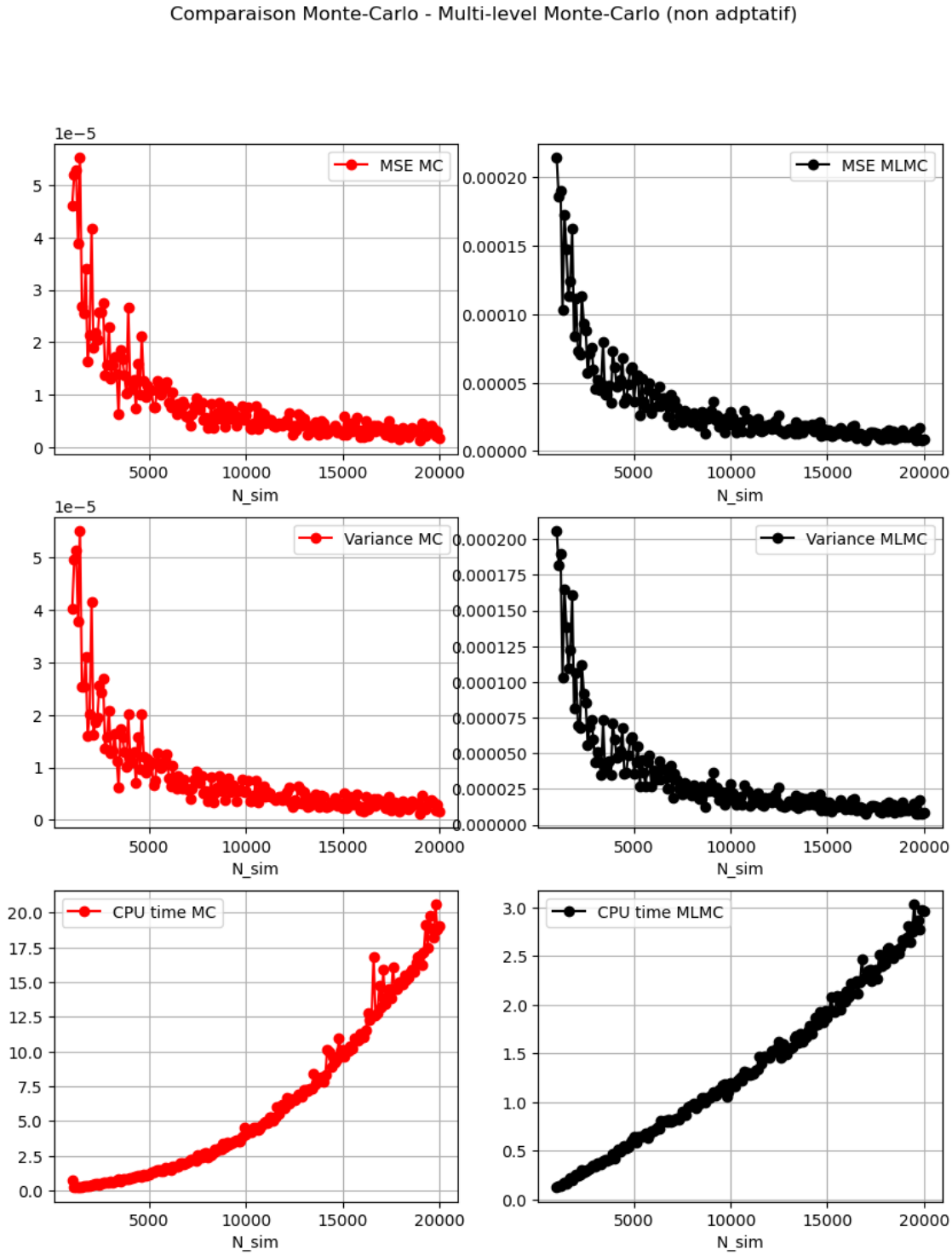

Figure 4.1: Performance comparison between MC qnd MLMC (non optimal)

4.2.2 MLMC with optimal parameters

If we compare the same indicators with MLMC's optimal character, we come to similar conclusions.
However, if we look at the low values of the ϵ tolerance granted, we see that the MLMC algorithm significantly reduces the variance, demonstrating the need for optimal values of N_l for it to work properly.
To carry out the following comparisons, different ϵ values are set in order to calculate the variance and MSE of the MLMC method for each value. Next, the standard Monte Carlo estimator is calculated with a number of simulations equal to the sum of N_l required for the MLMC estimator to reach a given ϵ value.

In the table below, the values on the left in each cell are the values obtained for standard Monte Carlo and those on the right are those for adptative MLMC.

Monte-Carlo standard // MLMC Adaptatif			
ϵ & MSE	Variance	CPU Time (s)	
3.10^{-4}	$1{,}2.10^{-4}//7, 3.10^{-7}$	$1{,}1.10^{-4}//5, 1.10^{-8}$	8,1 // 19110,2
4.10^{-3}	$1{,}1.10^{-4}//1, 2.10^{-5}$	$1{,}2.10^{-4}//1.2.10^{-6}$	4,9 // 16,7
7.10^{-3}	$2{,}0.10^{-4}//1, 7.10^{-5}$	$1{,}7.10^{-4}//1, 7.10^{-5}$	3,9 // 6,9

It is noticeable that execution time increases considerably for the last ϵ level (3.10^{-4}), but this allows a clear reduction in variance and MSE. Nevertheless, in order to achieve execution performance similar to that of Monte-Carlo, while having a lower variance, it seems wise to opt for a ϵ value close to 10^{-3}.

4.3 Comparison of RQMC and QMC MLMC

In this section, we make comparisons between the Randomized Quasi Monte-Carlo (RQMC) method and the Multi-Level Quasi Monte-Carlo (QMC MLMC) method, i.e. Multi Level whose Brownian motion increments are simulated by Randomized Quasi Monte-Carlo as explained in 2.2.2.

4.3.1 QMC MLMC avec paramètres non optimisés

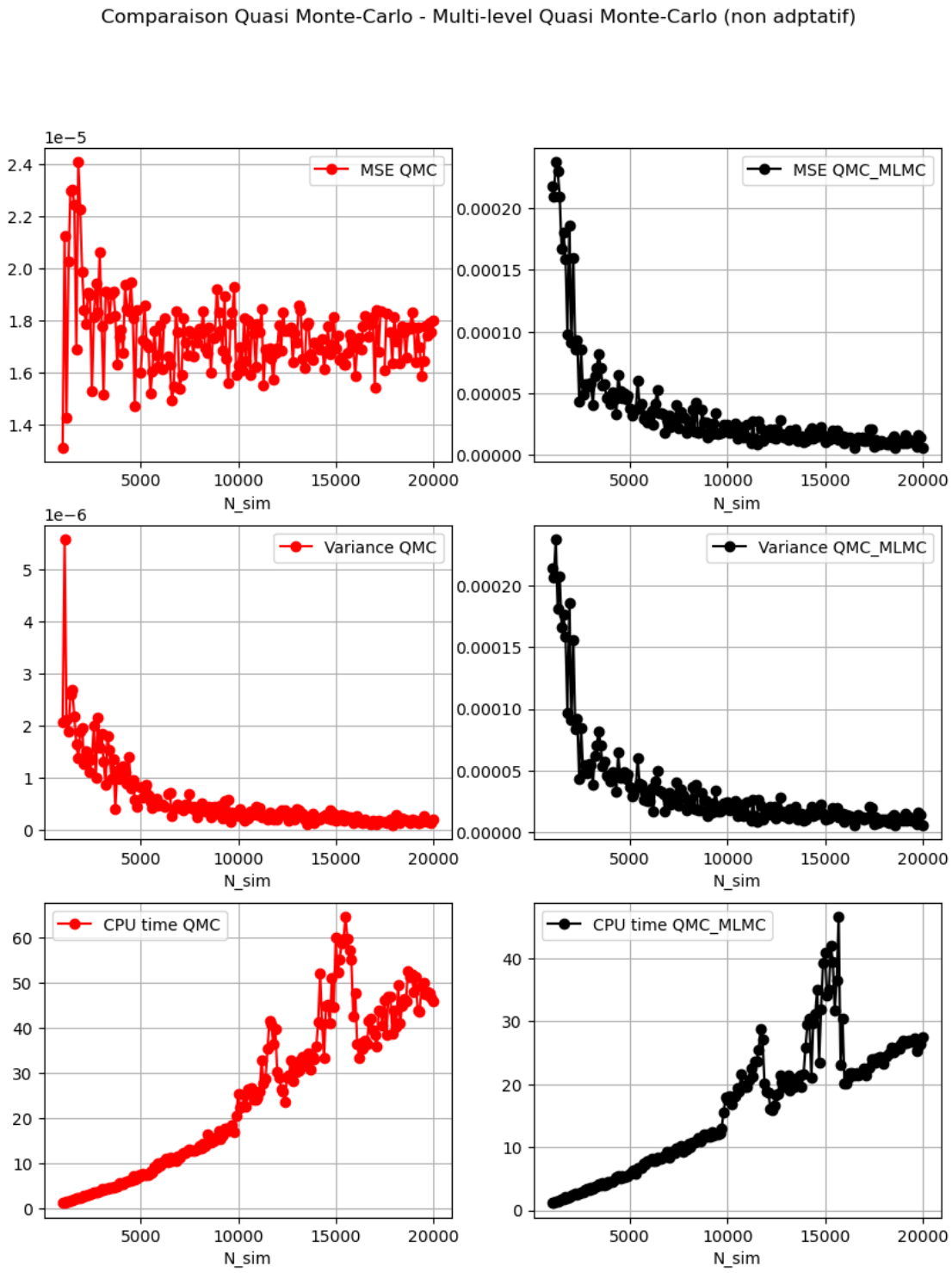

Figure 4.2: Performance comparison between QMC qnd QMC_MLMC (non optimal)

This scenario is similar to the previous one, in which we compared MC and MLMC. However, it should be pointed out that the variance of the QMC estimator is considerably lower with the same number of simulations as the standard Monte Carlo estimator. However, as with the trade-off between a low ϵ and a reasonable computation time for MLMC, the use of QMC also leads to an increase in the computation time needed to build an estimator.

4.3.2 QMC MLMC with optimql parameters

We can carry out a study comparable to the one carried out in section 4.1.2.

Quasi Monte-Carlo // QMC MLMC Adaptatif			
ϵ	MSE	Variance	CPU Time (s)
3.10^{-4}	$4{,}3.10^{-6}//7, 3.10^{-7}$	$6{,}1.10^{-7}//1, 4.10^{-8}$	71,4 // 38678,6
4.10^{-3}	$1{,}7.10^{-5}//6, 5.10^{-6}$	$5{,}1.10^{-6}//2, 6.10^{-7}$	40,5 // 59,7
7.10^{-3}	$1{,}8.10^{-5}//1, 5.10^{-5}$	$2{,}1.10^{-5}//1, 0.10^{-6}$	7,7 // 43,1

Clearly, using the Quasi Monte Carlo method can significantly reduce the variance in both cases. So, the best technique for reducing variance is to combine the MLMC algorithm with the QMC method, i.e. to use the Multi-Level Quasi Monte Carlo (QMC MLMC) method.

4.4 MSE and CPU time evolution

4.4.1 Variation of MSE with total number of simulations

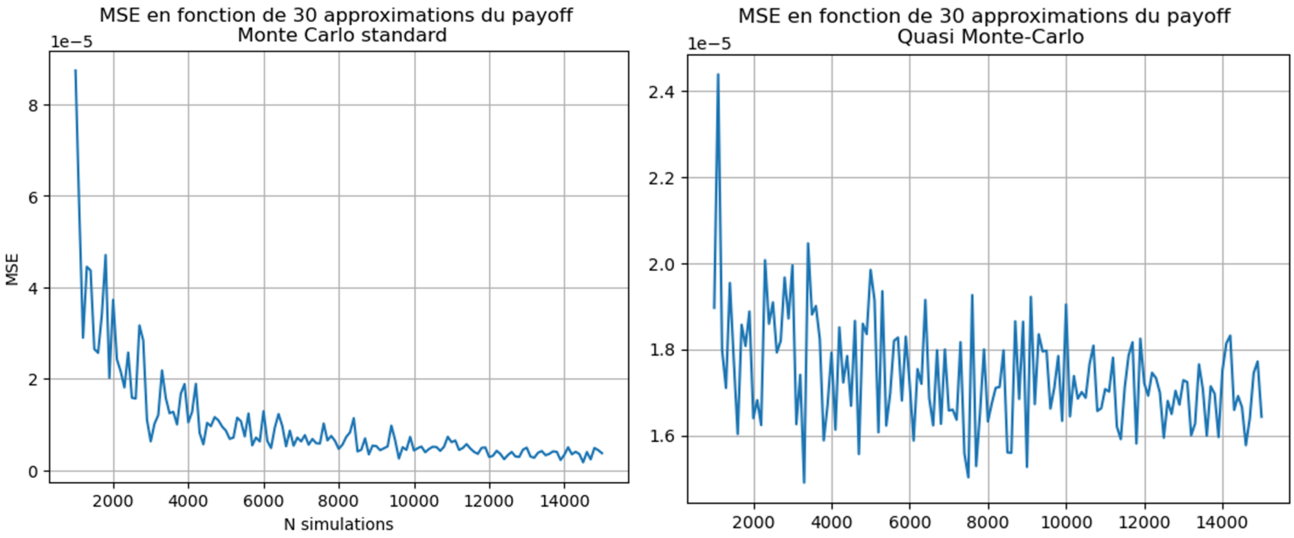

Figure 4.3: Evolution of MSE as a function of the number of simulations

The MSE decreases with the number of simulations, which is consistent with our intuition.

4.4.2 Variation of MSE with ϵ

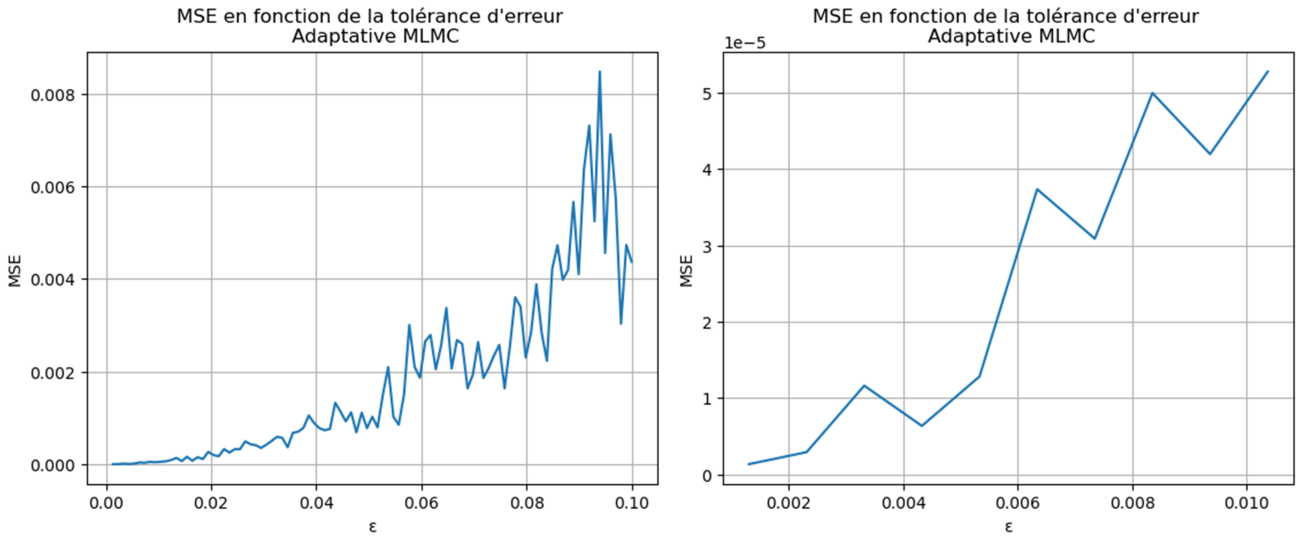

Figure 4.4: Evolution of MSE as a function of tolerance ϵ – MLMC

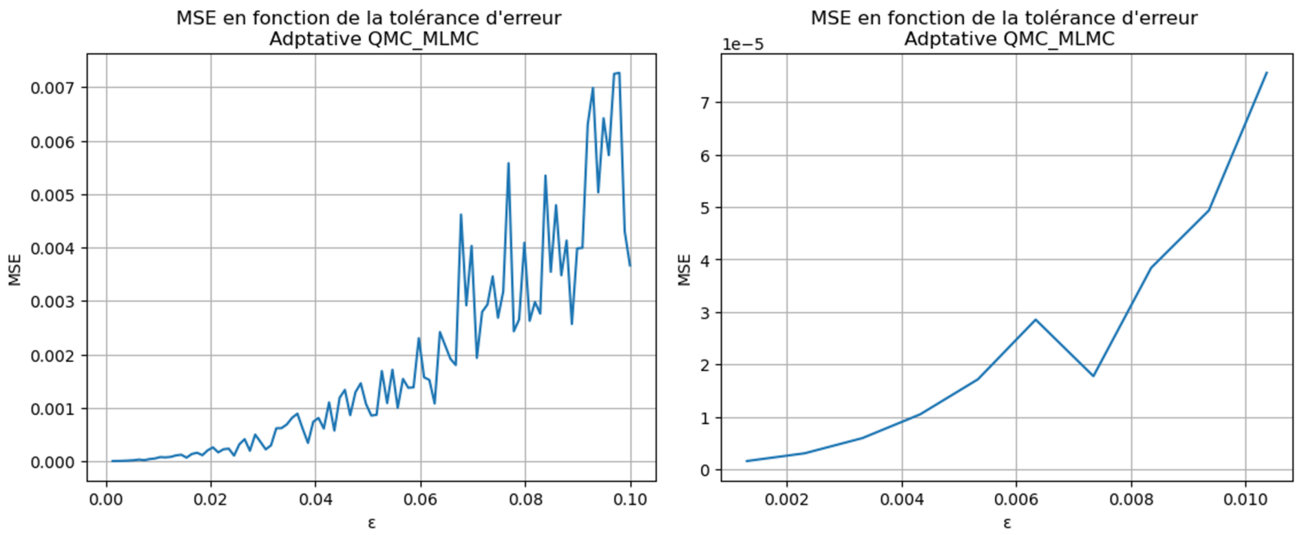

Figure 4.5: Evolution of MSE as a function of tolerance ϵ – QMC_MLMC

Although the variance is more stable than the MSE, the latter shows an increasing trend with increasing *epsilon*. The QMC_MLMC method, i.e. the MLMC method with increments of Brownian motion simulated by Randomized Quasi Monte Carlo, accentuates this growth.

5 Brownian Bridge

The goal is to simulate a Brownian motion $(W_{t_1}, W_{t_2}, ..., W_{t_k})$ according to a Brownian bridge i.e. $W_{t_i}|W_{t_{i-1}}, W_{t_{i+1}} \sim \mathcal{N}(\frac{(t_{i+1}-t_i)W_{t_{i-1}}+(t_i-t_{i-1})W_{t_{i+1}}}{t_{i+1}-t_{i-1}}, \frac{(t_{i+1}-t_i)(t_i-t_{i-1})}{t_{i+1}-t_{i-1}})$ following a base 2 Van der Corput sequence.

5.1 Van der Corput sequence

A Van der Corput sequence is a sequence whose main advantage is that it provides a more uniform distribution of points in a space than a random sequence. To implement the algorithm for obtaining a Van der Corput sequence in base 2, the method is as follows:

- For each index i in the sequence, we calculate its base-2 representation
- Invert the digits of the base-2 representation of index i to obtain the base-2 representation of the Van der Corput sequence for index i.
- The inverted base-2 representation is converted to a real number between 0 and 1 by dividing each digit by 2^z, where z is the position of the digit in the inverted base-2 representation.
- These terms are then added together to obtain the value of the Van der Corput sequence for index i

5.2 Brownian bridge method

Now it's time to generate the $W_{t_i}|W_{t_{i-1}}, W_{t_{i+1}} \sim \mathcal{N}(\frac{(t_{i+1}-t_i)W_{t_{i-1}}+(t_i-t_{i-1})W_{t_{i+1}}}{t_{i+1}-t_{i-1}}, \frac{(t_{i+1}-t_i)(t_i-t_{i-1})}{t_{i+1}-t_{i-1}})$ in the index order given by the Van der Corput sequence, i.e. W_{t_k} puis $W_{t_{\frac{k}{2}}}$, $W_{t_{\frac{k}{4}}}$, etc.

We combined this method with QMC methods for simulating normals, notably the 2D Box-Muller method and a Sobol sequence (explained in 2.2.2). The results obtained for standard Monte Carlo estimation with this Brownian bridge were very interesting, as the variance of the estimators was reduced by up to 10^{-11}!

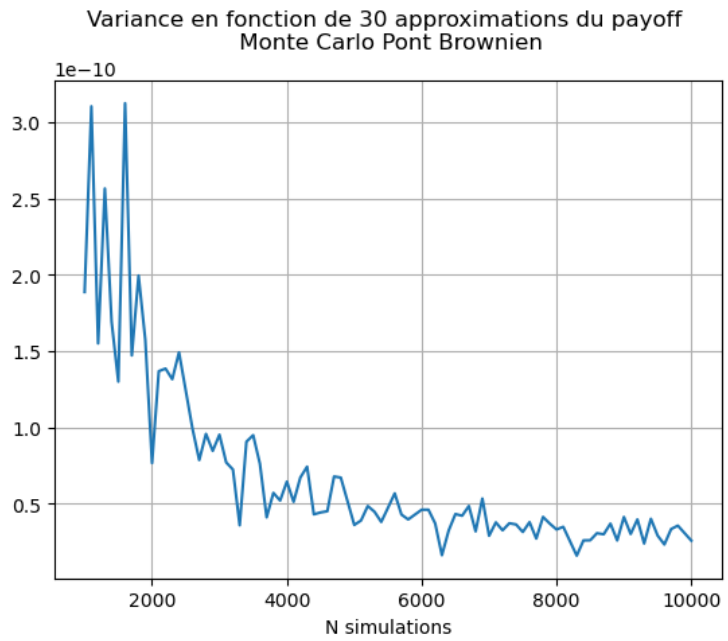

Figure 5.1: Evolution of the variance of QMC estimators with Brownian bridge as a function of the number of simulations

However, the results are somewhat disappointing for the Brownian bridge associated with the MLMC algorithm, since the variance and MSE remain similar to those obtained in the QMC MLMC case.
If we refer to the conclusion of Michael B. Giles' paper, in section 7 "Final Remarks", we can understand that the implementation of the Brownian bridge should take place ***during*** the execution of the MLMC algorithm rather than before, as we have done. In other words, at each step up to convergence, the algorithm uses the Brownian bridge increments already calculated while creating new ones.

6 Conclusion

The MLMC method is based on the simple mathematical idea of using a telescopic sum of increasingly complex estimators to approximate an expectation without bias, while considerably reducing the variance of the final estimator. However, for this method to work effectively, it is imperative to optimally calculate the values of N_l, i.e. the number of simulations to be performed at each level l.

What's more, as this MLMC method works as a set of Monte-Carlo methods in sequence, it can be combined with classical Monte-Carlo variance reduction methods to achieve even better results - albeit at the cost of longer computation time.

In theory, the MLMC method can be used to arbitrarily reduce the variance of calculated estimators, but in practice, as shown in this study, the computation time and complexity quickly become very high, requiring ever greater computing power.

7 Appendix

7.1 Proof $N_l \propto \sqrt{V_l h_l}$

From section 3.2.1 :

$\mathbb{V}[\hat{Y}] = \sum_{l=1}^{L} \mathbb{V}[\hat{Y}_l] = \sum_{l=1}^{L} \frac{V_l}{N_l}$

We wish to minimize the variance. The constrained optimization problem is as follows:

$min(\mathbb{V}[\hat{Y}])$ constrained by $\sum_{l=1}^{L} C_l N_l = A$ with C_l the CPU cost per sample and A a constant representing the total cost

$\mathcal{L} = \sum_{l=1}^{L} \frac{V_l}{N_l} + \lambda(\sum_{l=1}^{L} C_l N_l - A)$

$\frac{\partial \mathcal{L}}{\partial N_l} = -\frac{V_l}{N_l^2} + \lambda C_l$

Thus : $N_l = \sqrt{\frac{V_l}{\lambda C_l}}$

C_l cqn be approximated by M^l hence $N_l = \sqrt{\frac{V_l}{\lambda M^l}} \propto \sqrt{V_l h_l}$

7.2 Evolution of N_l

We can also look at the N_l values given by M. B Giles' algorithm. So it's interesting to compare these values by varying *epsilon*.

7.2.1 Multi-Level Monte-Carlo

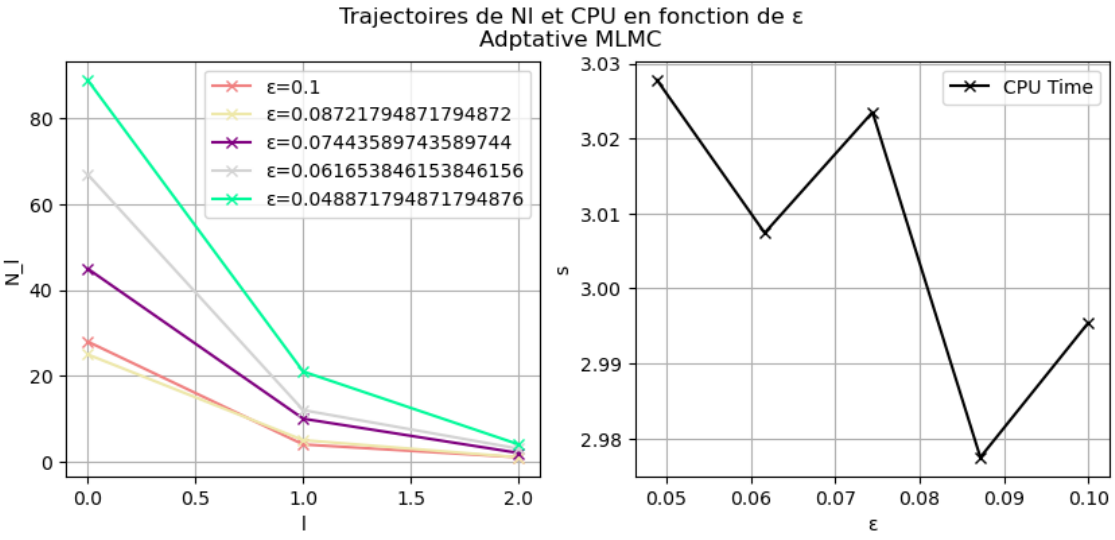

Figure 7.1: Evolution of N_l and CPU time for high ϵ - MLMC

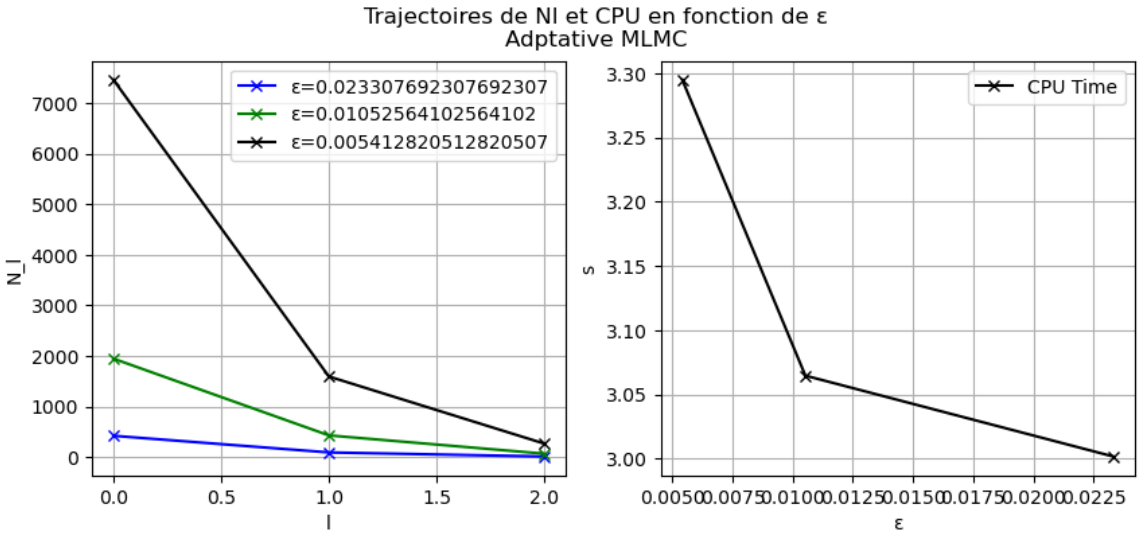

Figure 7.2: Evolution of N_l and CPU time for lower ϵ - MLMC

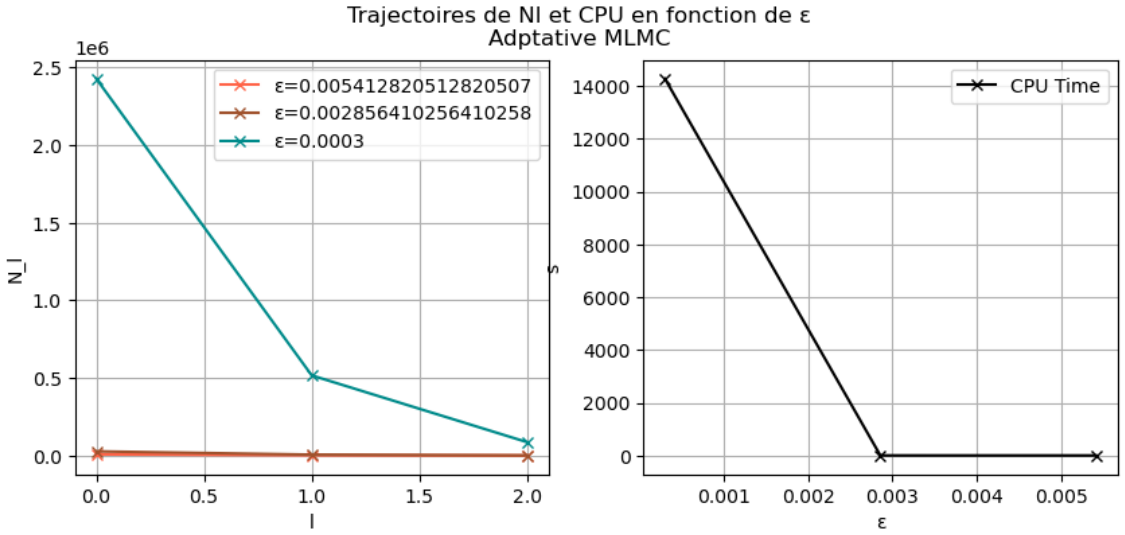

Figure 7.3: Evolution of N_l and CPU time for very low ϵ - MLMC

It's worth noting that the value of N_l explodes very quickly when the value of ϵ falls below 10^{-3}. The number of N_l to be calculated then rises into the millions, resulting in a considerably longer calculation time than for larger ϵ. The trade-off between error tolerance and computation time is clear to see here. Although it is theoretically possible to reduce the variance considerably, in practice this would require considerable computing power and prolonged calculation time.

7.2.2 Multi-Level Quasi Monte-Carlo

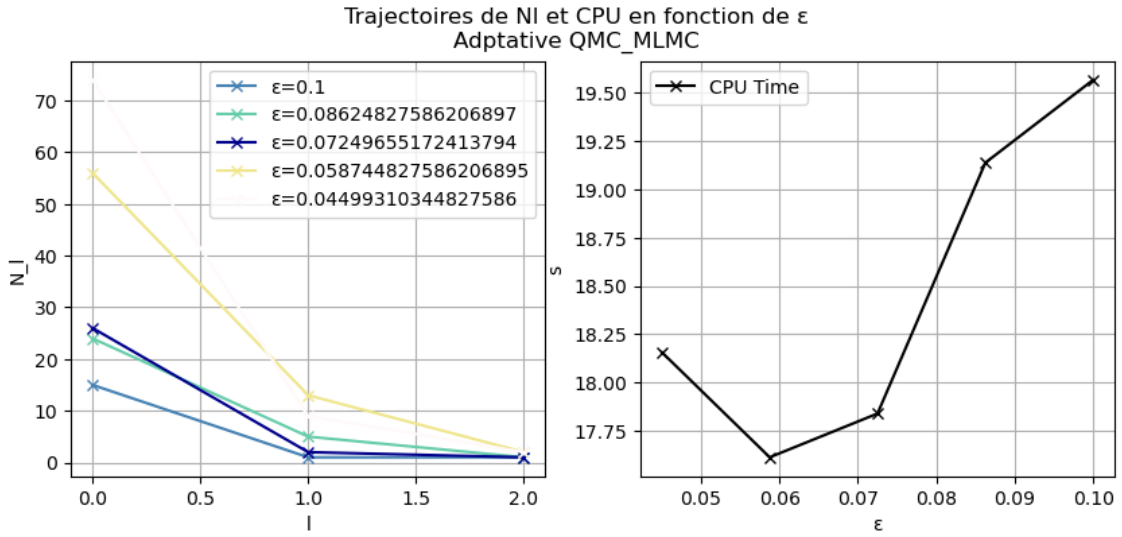

Figure 7.4: Evolution of N_l and CPU time for high ϵ - RQMC_MLMC

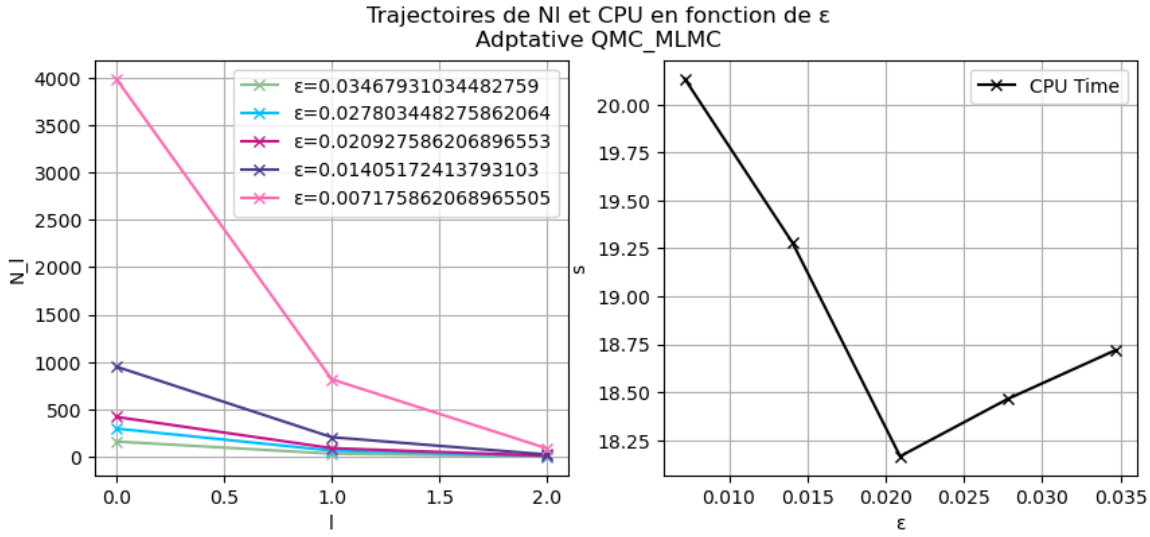

Figure 7.5: Evolution of N_l and CPU time for lower ϵ - RQMC_MLMC

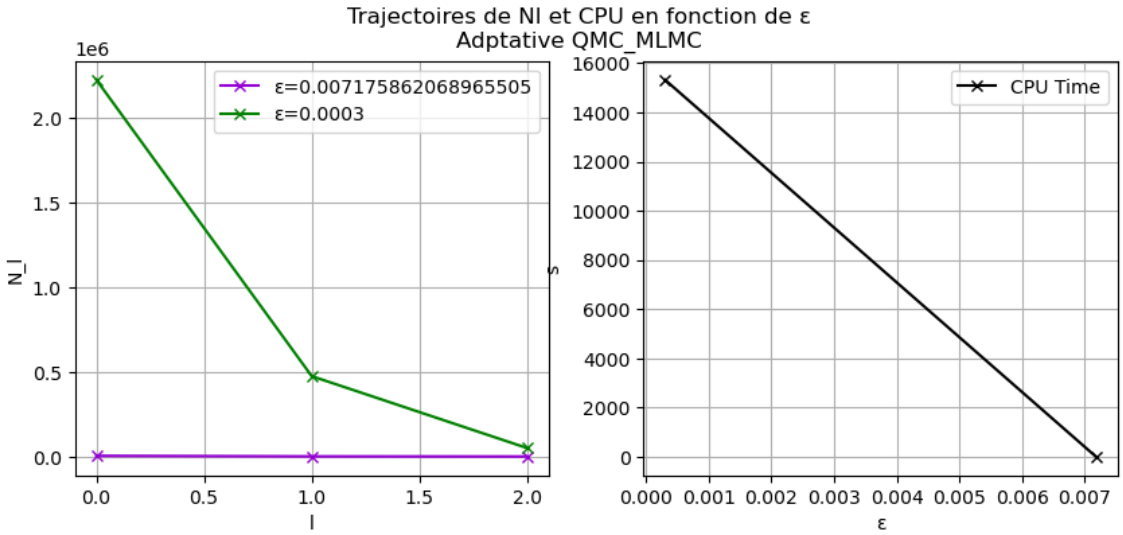

Figure 7.6: Evolution of N_l and CPU time for very low ϵ - RQMC_MLMC

We can see that the same phenomenon can be observed in the case of Multi-Level Quasi Monte-Carlo.