From Naive Python to Optimized C++: Monte Carlo Option Pricing

Mathis Lebret

July 2025

Abstract

In algorithmic finance, managing latency is essential to reacting quickly to market information. This project explores algorithmic optimization in European call pricing using Monte Carlo simulation. After presenting the Black–Scholes model and the Euler–Maruyama discretization scheme, we implement random path simulations in both Python and C++. Performance tests show that C++ greatly outperforms Python in terms of runtime. The most optimized C++ version achieves a speedup of about 3,000 compared to naive Python and 40 compared to a NumPy-based approach thanks to parallelization with OpenMP and vectorization with EigenRand, which illustrates the significant impact of implementation choices on execution speed in latency-sensitive financial applications. An analysis of the convergence of the pricing estimator confirmed the excessiveness of Monte Carlo simulation for such a simple payoff; however the study remains highly relevant for path-dependent derivatives. While CPU optimization brought significant improvements, further performance gains could be expected from GPU acceleration.

Contents

1 Introduction

The objective of this project is to explore various strategies used for reducing latency in Monte Carlo simulations. We begin with a naïve Python implementation, then introduce vectorization with numpy. We then transition to C++, starting with a basic implementation and progressively applying optimizations such as parallelization with OpenMP and the use of libraries (Eigen, EigenRand) to accelerate random number generation.

While the project uses the Black-Scholes model to price a European call option, the focus is made on latency optimization rather than option pricing itself. Monte Carlo simulations are excessive for such a simple payoff, but the optimization techniques presented here become valuable for more complex derivatives, such as Asian options, where the full path of the underlying asset price must be simulated.

The report presents the theoretical background for the BS model and the numerical methods used in the project (Monte Carlo simulation with Euler-Maruyama approximation), followed by their implementation in Python and C++.

2 Mathematical Background [3]

2.1 Black-Scholes Model

The **standard Black-Scholes model** is particularly adapted for European Options pricing without dividend distributions, where the only cash flow is the option payoff $(S_T - K)^+$ at the maturity date T.

With the BS model, the price evolution of the underlying asset S_t (with an initial price S_0) follows a Geometric Brownian Motion (GBM) with a constant volatility σ and a drift μ:

$$dS_t = \mu S_t \, dt + \sigma S_t \, dW_t$$

where W_t is a standard Brownian motion (or Wiener Process).

Itô's Lemma gives a solution to this Stochastic Differential Equation (SDE) for the log-normal process S_t:

$$S_t = S_0 \cdot \exp\left(\left(\mu - \frac{\sigma^2}{2}\right) t + \sigma W_t\right)$$

2.2 Black-Scholes Formula

The construction of an auto-financing portfolio replicating the underlying asset under no-arbitrage condition gives us an explicit solution known as the **Black-Scholes Formula**.

The European call Price at $t = 0$ is given by:

$$C = S_0 \Phi(d_1) - K e^{-rT} \Phi(d_2)$$

$$\text{where} \quad \begin{cases} d_1 = \dfrac{\ln\left(\frac{S_0}{K}\right) + \left(r + \frac{\sigma^2}{2}\right) T}{\sigma\sqrt{T}} \\ d_2 = d_1 - \sigma\sqrt{T} \\ \\ \Phi \text{ is the cumulative distribution function of } \mathcal{N}(0,1) \end{cases}$$

2.3 Numerical Methods

The discounted price of the underlying asset $\tilde{S}_t = S_t \cdot e^{-rt}$ is a martingale when S_t follows a GBM. We can then obtain the price of a European call, with a strike-price K, at $t = 0$:

$$C_0 = \mathbb{E}_Q \left[(S_T - K)^+ \right] e^{-rT}$$

With a **Monte-Carlo Simulation**, we can simulate a large number N of paths $\{S_0, S_1, ..., S_T\}$ with n steps and calculate the average discounted payoff at maturity date[1].

Under the risk-neutral probability Q, the drift μ is equal to the risk-free rate r, and the underlying asset price S_t is now given by the SDE:

$$dS_t = rS_t\, dt + \sigma S_t\, dW_t$$

The **Euler-Maruyama Approximation**, gives us a solution for S_{t+1}:

$$S_{t+1} = S_t \exp\left(\left(r - \frac{\sigma^2}{2}\right) dt + \sigma\sqrt{dt}\, Z\right)$$

$$\text{where } dt = \frac{T}{n} \text{ and } \frac{dW_t}{\sqrt{dt}} = Z \sim \mathcal{N}(0,1)$$

To optimize the program with vectorization, we use the increment of log-prices due to their additive characteristic. They are given by:

$$\Delta \ln S = \left(r - \frac{\sigma^2}{2}\right) dt + \sigma\sqrt{dt}\, Z$$

2.4 Monte Carlo Simulation Convergence

The Monte Carlo method is based on the Law of Large Numbers, which guarantees that the empirical estimator $\hat{C}_N$ converges almost surely toward the theoretical expectation C as $N \to +\infty$. The convergence rate is of order $\mathcal{O}\left(\frac{1}{\sqrt{N}}\right)$, meaning that the standard error of the estimator decreases proportionally to

[1] For European Options, path simulation is unnecessary since a closed-form solution exists for S_T. Here, we perform it in the sole purpose of having a simple model in order to focus on latency optimization in Monte Carlo simulations.

$1/\sqrt{N}$.

This result comes from the Central Limit Theorem, which states that the distribution of the Monte Carlo estimator satisfies

$$\sqrt{N}\left(\hat{C}_N - C\right) \xrightarrow{d} \mathcal{N}(0, \sigma_p^2)$$

where C is the true option price and σ_p^2 is the variance of the discounted payoff under the risk-neutral measure.

This asymptotic normality enables the construction of approximate confidence intervals at confidence level $1 - \alpha$:

$$\left[\hat{C}_N - z_{\alpha/2}\frac{\hat{\sigma}_p}{\sqrt{N}}, \quad \hat{C}_N + z_{\alpha/2}\frac{\hat{\sigma}_p}{\sqrt{N}}\right]$$

where $z_{\alpha/2}$ is the $\alpha/2$ quantile of the standard normal distribution and $\hat{\sigma}_p$ is the sample standard deviation of the discounted payoffs used in the Monte Carlo simulation.

Pricing a European call under the Black-Scholes model provides a closed-form solution for C. This allows us to benchmark the Monte Carlo estimator $\hat{C}_N$, verify the convergence, and assess the accuracy and efficiency of numerical approximations and optimizations.

To empirically assess the convergence and variability of the Monte Carlo estimator $\hat{C}_N$, we perform k independent simulations of $\hat{C}_N$, each based on N paths. This allows us to approximate the sampling distribution of $\hat{C}_N$. We base our estimation of the accuracy and variability of the Monte Carlo estimator on the empirical mean

$$\bar{C} = \frac{1}{k}\sum_{i=1}^{k} \hat{C}_N^{(i)},$$

We consider a 95% confidence interval for the empirical mean $\bar{C}$ defined by:

$$\left[\bar{C} - z_{0.025}\frac{\hat{\sigma}_{\hat{C}_N}}{\sqrt{k}}, \quad \bar{C} + z_{0.025}\frac{\hat{\sigma}_{\hat{C}_N}}{\sqrt{k}}\right]$$

with

$$\bar{C} = \frac{1}{k}\sum_{i=1}^{k} \hat{C}_N^{(i)} \quad \text{and} \quad \hat{\sigma}^2_{\hat{C}_N} = \frac{1}{k-1}\sum_{i=1}^{k}\left(\hat{C}_N^{(i)} - \bar{C}\right)^2,$$

where $z_{0.025} \approx 1.96$ is the quantile of the standard normal distribution.

3 Programming Implementation

3.1 Simulation Parameters

The Monte Carlo simulations rely on a set of initial parameters that define the option and the underlying asset dynamics:

- $S_0 = 100.0$: Initial underlying asset price
- $K = 105.0$: Strike price of the European call option
- $r = 0.05$: Risk-free interest rate
- $\sigma = 0.2$: Volatility of the underlying asset
- $T = 1.0$: Time to maturity (in years)
- $steps = 100$: Number of time steps used for path discretization
- $N = ...$: Number of Monte Carlo paths simulated per run

These parameters correspond to a typical setting in the Black-Scholes framework and are used throughout the simulations presented in this report. The only parameter affecting the simulation time latency is the number of steps for each path, we will keep it constant throughout the project.

With this data and the analytical formula for the Black-Scholes European call price, we obtain:

$$\mathbf{C = 8.02135\ \$}$$

3.2 Optimal Number of Paths

We want to estimate the number of paths N that are required for a high quality simulations. We will target this number of paths in the next sections.

The following plot shows us the distribution of the estimator $\hat{C}_N$ with $N = 10,000$ paths for different values of k, with $\bar{C}$ and $\hat{\sigma}_{\hat{C}_N}$. This allows us to consider that we obtain a realistic estimation of the variability of $\hat{C}_N$ when $k = 1,000$, this guarantees that the computing time is not excessive when N is large. However, for smaller values of N, we will use $k = 10{,}000$ for precision.

We recall that N controls the precision of each individual estimator $\hat{C}_N$, while k is used to characterize the empirical distribution of the estimator.

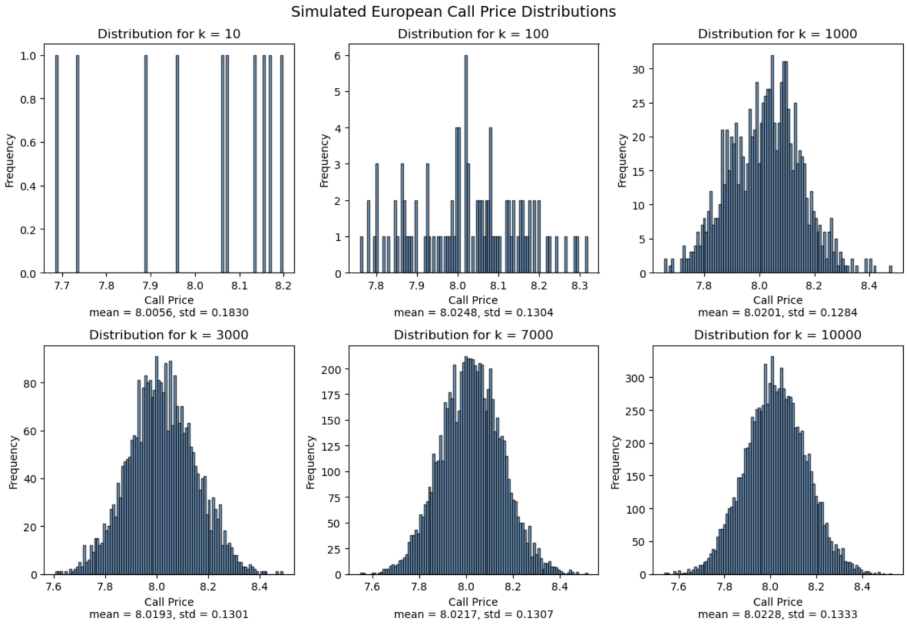

Figure 1: Estimator $\hat{C}_N$ distribution based on the number of simulations k

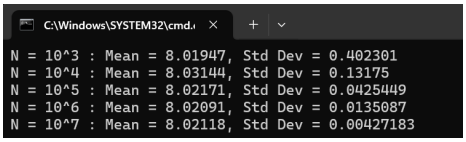

Figure 2: Mean and Standard Deviation of $\hat{C}_N$ for 1000 simulations

Using C++, we ran 10,000 simulations for $N = 10^3$,10^4,10^5 and 1,000 simulations for $N = 10^6$,10^7. The results obtained from these simulations can be summarized as follows:

N	Mean	Std Dev	Margin ($\alpha = 5\%$)	Bias	Lower	Upper	Correct at cent
10^3	8.01687	0.42005	0.02604	-0.00448	7.99083	8.04291	False
10^4	8.02427	0.13296	0.00824	0.00292	8.01603	8.03251	False
10^5	8.02254	0.13207	0.00819	0.00119	8.01435	8.03073	False
10^6	8.02091	0.01351	0.00084	-0.00044	8.02007	8.02175	True
10^7	8.02118	0.00427	0.00026	-0.00017	8.02092	8.02144	True

Table 1: Results of the Monte Carlo estimator for various N

For a European call option, we need to know the price at cent. From the simulation results shown above, we can conclude that above 10^6 paths, we have a good estimation of the call price

Moreover, the Bias defined as $\bar{C} - C$ for $N = 10^4$ shows that for a low value of N, the estimator can give a false estimation on average by half a cent.

3.3 Python

3.3.1 Naïve Python

The implementation begins with a **straightforward naïve Python** approach. The goal here is to estimate a time benchmark for our Monte Carlo Simulation.

This first version of the code is made of 2 nested loops. An optimization was made by calculating the drift and diffusion outside of these loops, dividing computation time by 2.

3.3.2 NumPy Vectorization

The **NumPy library** based on C programming language allows us to avoid loops by vectorizing. We compute a (N x *steps*)-size matrix of draws from the $N(0, 1)$ distribution. From this matrix, and with a matrix product, we obtain the diffusion part of the Euler-Maruyama approximation. We then add to the matrix the constant drift, which gives us the log-returns. By using a cumulative sum (this justifies the use of log-returns) we get N paths (made of returns between time 0 and time t). From the last price (value of the path at time T multiplied by S0), we obtain the payoff for every path, and we can calculate the call price.

3.4 C++

3.4.1 Naive C++

The first implementation of the simulation in C++ is a copy of the Naïve Python version. The only improvement done is the use of references in the for loops to store the payoffs.

3.4.2 OpenMP

Significant speed-up in computation can be achieved by distributing tasks across multiple threads on multi-core CPUs. For Monte Carlo simulations, tasks are independent and parallelization plays a significant role in reducing runtime.

The **Open Multi-Processing** (OpenMP) API allows us to parallelize code on shared-memory systems. It uses compiler directives to specify which parts of the code will run concurrently across threads. This allows developers to benefit from parallel execution without manually handling thread management.

In our code, we still used nested loops but paths were parallelized. However, we couldn't parallelize steps as they are not independents.

3.4.3 Eigen Vectorization [1]

The **Eigen** library allows us to efficiently manipulate vectors and matrices, enabling vectorized operations similar to Python NumPy. However, Eigen does not provide a built-in generator for normally distributed random variables, so we use the standard C++ generator (std::normal_distribution) to fill vectors element by element.

We can couple Eigen with OpenMP to further enhance vectorization and parallelize vector operations.

3.4.4 EigenRand [2]

EigenRand includes a built-in generator for normally distributed variables, which permits us to generate matrices of random variables returned directly in Eigen format for vectorized operations.

However, when combined with OpenMP, generating entire matrices is not optimal, so we generate vectors for each path using nested loops.

4 Performance Analysis

In this section, we summarize the performance of the different techniques presented in this report.

4.1 Testing Environment

All tests were conducted on the same hardware and software configurations. However no special measures were taken to optimize the benchmarking process (no CPU limiting or process isolation), which may cause some variability in the recorded times.

To guarantee a certain quality of measurement, the simulations were repeated 10 times and the average computing time was recorded. Although this approach may lack high precision, it is sufficient for capturing the general trend of improvement between the models that we study here.

Also, no random seed was set for this project, since the computing time does not depend on the random values generated during the simulations.

The C++ code was compiled using the following command with various optimization and parallelization flags:

```
g++ -O3 -march=native -funroll-loops -ffast-math -flto -std=c++17
-fopenmp %f -o %e
```

- **-O3** : Enable high-level optimizations for performance
- **-march=native** : Optimize code for the architecture of the compiling machine
- **-funroll-loops** : Unroll loops to improve runtime speed

- **-ffast-math** : Allow faster but less strict floating-point math optimizations
- **-flto** : Enable Link Time Optimization across compilation units
- **-std=c++17** : Use the C++17 standard
- **-fopenmp** : Enable OpenMP for parallel programming support

4.2 Results

Naïve Python As expected, the Naïve Python approach is poorly effective. On average simulating $N = 10^5$ paths takes 34 seconds... A linear estimation of the time required for $N = 10^6$ indicates that we need 6 minutes to have a correct to the cent price for a single European call.

Numpy The NumPy simulation with vectorization brings a big improvement, dividing the computing time by 80 from the naive code. However, for $N = 10^7$, the simulation takes over a minute, which is not optimal.

Naive C++ Surprisingly, the Naïve C++ approach outperforms the NumPy simulation, dividing runtime by 2.

OpenMP The use of parallelization over standard C++ brings a small improvement (2 times speedup from Naïve C++). However, when combined with Eigen or Eigen Rand, it brings up to x8 improvement.

Eigen & EigenRand Compared to Python, in C++ there is not a huge performance gain from vectorization. However, the parallelization of vectorized simulations gave a significant boost. We notice that EigenRand approach is 2 times faster than Eigen.

Highest Performance Ultimately, the EigenRand + OpenMP simulation takes only 0.0125 second when $N = 10^5$, representing a speed reduction of nearly 3,000 times over Naïve Python.

Furthermore, when Eigen or EigenRand libraries are coupled with OpenMP parallelization, simulating $N = 10^8$ paths becomes achievable, with runtimes of 21 seconds and 8 seconds respectively.

4.3 Summary

The table below summarizes the average runtime (in seconds) over 10 simulations for various models and different values of N. For some large values of N, simulations were not performed due to time constraints and are marked with an "X" in the table.

	Python		C++					
N	Naive	Numpy (Vect)	Naive	openMP	Eigen	Eigen & openMP	EigenRand	EigenRand & openMP
10^5	34.874	0.441	0.202	0.106	0.172	0.0245	0.0873	0.0125
10^6	X	4.766	2.065	0.976	1.872	0.203	0.850	0.092
10^7	X	69.919	21.566	9.726	17.356	1.996	10.777	0.763
10^8	X	X	X	X	X	21.468	X	8.026

Table 2: Average runtime (in seconds) for 10 simulations for different models and different values of N

5 Conclusion

This project illustrated the significant impact of algorithmic choices on the efficiency of Monte Carlo simulations for option pricing. Starting from a naïve Python implementation, we progressively introduced vectorization, C++ translation, and advances optimization techniques such as OpenMP parallelization and EigenRand vectorized number generation.

The results clearly demonstrate that C++ can achieve significant speedups compared to Python, highlighting the relevance of low-level optimizations in latency-sensitive financial applications. While Monte Carlo simulations are excessive for plain European options, the methods and performance gains studied here are crucial for more complex, path-dependent derivatives, where computational efficiency directly affects practical feasibility.

Finally, the analysis of convergence confirmed the theoretical properties of Monte Carlo estimators and showed that sufficiently large numbers of paths are required to achieve precision at the cent level. In this report we only addressed CPU optimization; additional improvements could be expected from *GPU optimization* techniques using programs like CUDA (by Nvidia), which allow C++ parallelization to be executed on a GPU.

References

References

[1] Gael Guennebaud and Benoit Jacob. *Eigen, a C++ template library for linear algebra*, 2020. Version 3.4.0.

[2] Baek Min. Eigenrand: The fastest c++11-compatible random distribution generator for eigen, 2023.

[3] Portait R Poncet P. *Finance de Marché*. Lefebvre Dalloz, 5th edition, 2023.