Monte Carlo Simulations in Financial Derivatives Pricing

Yuchen Wang

March 2025

1 European Financial Contract

In this study, we aim to price a European financial contract with a specified payoff structure. The payoff function at maturity T is given by:

$$C(S_T, T) = g(S_T) = \begin{cases} X_1 - X_2, & \text{if } S_T < X_1, \\ X_1, & \text{if } X_1 \leq S_T < X_2, \\ X_2 - X_1, & \text{if } S_T \geq X_2. \end{cases} \tag{1}$$

Given the current asset price S_0, market prices indicate that the risk-neutral distribution of the asset price at time T follows:

$$S_T \sim \mathcal{N}\left(f(S_0, T), v^2(S_0, T)T\right),$$

where the functions governing the distribution of the stock price are defined as:

$$f(S_0, T) = S_0\left(e^{\alpha T} - e^{\beta T}\right) + \theta\left(1 + \tanh(\alpha T) - \beta T\right), \tag{2}$$

$$v(S_0, T) = \sigma\sqrt{1 + \alpha T S_0^{\gamma}}. \tag{3}$$

These equations describe the expected evolution of the underlying asset price under the risk-neutral measure. The goal of this study is to apply Monte Carlo simulations to approximate the fair value of this financial contract, analyse its convergence, and explore alternative sampling methods for improved efficiency.

1.1 Monte Carlo Simulation for Contract Valuation

In this section, we implement a Monte Carlo simulation to approximate the value of the financial contract using the parameters given in Figure 7. Additionally, we compute the option price using a numerical quadrature method to obtain an analytic benchmark. The Python code for the implementation is provided in Figure 8.

Methodology The theoretical price of the financial contract can be determined using the following integral representation:

$$C(S_0, t=0) = \frac{e^{-rT}}{v\sqrt{2\pi T}} \int_{-\infty}^{\infty} g(z) \cdot \exp\left[-\frac{(z-f)^2}{2v^2T}\right] dz. \tag{4}$$

Here, $g(S)$ represents the given payoff function, and $f(S_0, T)$, $v(S_0, T)$ are the expected asset value and volatility functions under the risk-neutral measure.

The Monte Carlo method relies on sampling from the underlying stock price distribution at time T. We use the risk-neutral dynamics:

$$\phi \sim \mathcal{N}(0, 1), \tag{5}$$

$$S_T^i = f(S_0, T) + v(S_0, T)\sqrt{T}\phi_i. \tag{6}$$

We then compute the discounted expected payoff as:

$$C(S_0, t=0) \approx e^{-rT} \frac{1}{n} \sum_{i=1}^{n} g(S_T^i).$$

In our implementation, we use $N = 100000$ Monte Carlo paths to ensure convergence of the estimated option price.

Results The computed option values using both methods are summarized below:

Method	**Option Price**
Analytic Solution	30651.76
Monte Carlo (N = 100000)	30654.35

Table 1: Computed European Option Prices using Different Methods

The results indicate that the Monte Carlo estimate closely approximates the analytic solution, demonstrating the validity and accuracy of the simulation method.

1.2 Convergence Analysis

This section examines the convergence behaviour of the Monte Carlo approximation for the financial contract value $C(S_0, t=0)$ as the number of simulations N increases. The estimated values are compared against the analytical solution to evaluate the accuracy of the method.

Theoretical Convergence Rate The Monte Carlo method exhibits a well-known convergence rate of $O(N^{-1/2})$ as established by Asmussen and Glynn (2007). This result follows directly from the Central Limit Theorem , which states that the normalized error of the Monte Carlo estimator asymptotically follows a normal distribution:

$$R(\hat{z} - z) \xrightarrow{d} N(0, \sigma^2).$$

Rearranging this expression, we obtain:

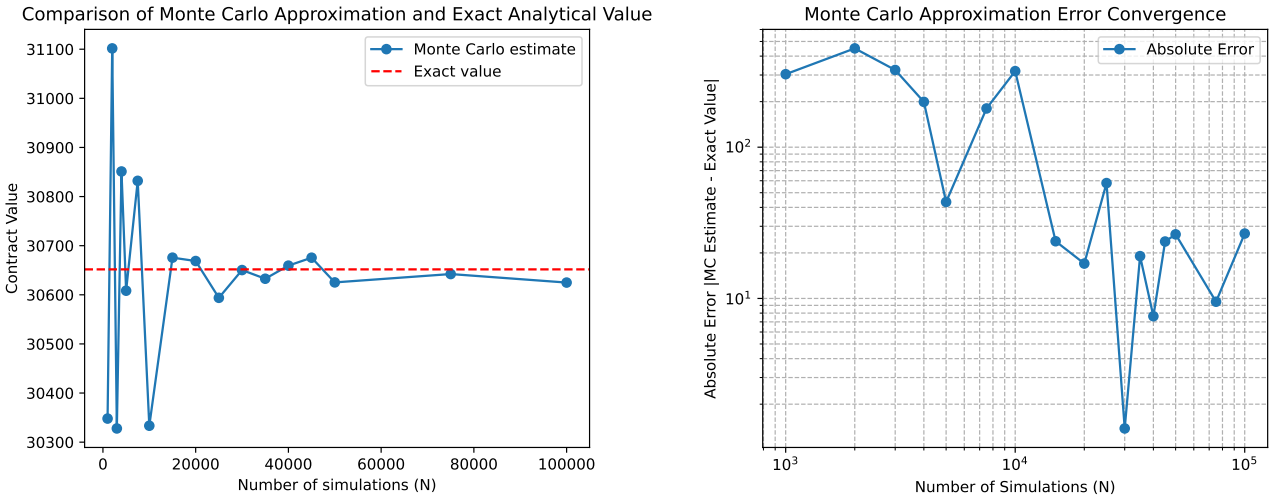
$$\hat{z} \approx z + \sigma RV, \quad V \sim N(0,1).$$

This implies that the standard deviation of the estimator decreases at a rate proportional to $1/\sqrt{N}$, leading to the characteristic $O(N^{-1/2})$ convergence rate of the Monte Carlo method. However, despite this convergence, stochastic fluctuations remain due to the inherent variance of the method.

To validate the theoretical convergence rate, we conduct a numerical experiment by computing the Monte Carlo estimate for varying values of N, specifically $N = 1000, 2000, \ldots, 50000$, and comparing these estimates to the exact value obtained via numerical quadrature. Additionally, the error behaviour is examined by considering the log-log transformation of the error:

$$\log(\text{error}) \approx \log C - \frac{1}{2} \log N.$$

According to theoretical predictions, the error should exhibit a linear trend with a slope of approximately -0.5 when plotted in a log-log scale.



(a) Monte Carlo Estimate

(b) Monte Carlo Approximation Error

Figure 1: Comparison of Monte Carlo Approximation and its Error Convergence

Observations from the Monte Carlo Convergence Plot The empirical results confirm the expected convergence behaviour of the Monte Carlo method. The log-log plot of the error against the number of simulations N reveals a downward trend with a slope close to -0.5, which is consistent with the theoretical $O(N^{-1/2})$ rate. However, the error does not decrease monotonically but instead exhibits fluctuations, reflecting the impact of random sampling variance.

Despite the overall convergence, the presence of stochastic noise suggests that further improvements can be achieved through variance reduction techniques. Methods such as antithetic variates and control variates can be employed to accelerate convergence and achieve a more stable error decay. The integration of such techniques into the Monte Carlo framework is explored in the following section.

1.3 Advanced Techniques for Accuracy and Efficiency

Random Number Generators As discussed in Dagpunar (2007), the performance of Monte Carlo simulations depends critically on the choice of the pseudorandom number generator (PRNG). The key factors influencing the quality of a PRNG include its period length, statistical randomness, and computational efficiency. An ideal PRNG should have a sufficiently long period to prevent cyclical patterns, ensuring that simulated paths remain independent over the course of the computation. Furthermore, the generator should exhibit strong statistical properties, such as uniformity and low correlation, to avoid biases in price estimations. Finally, computational efficiency is crucial, as Monte Carlo simulations often require a large number of random samples to achieve accurate results.

RNG	N	Estimated Price	Error (%)	Time (s)	Period
PCG64	100000	30674.918341	0.075560	0.034902	2^{128}
MT19937	100000	30726.095428	0.242523	0.032676	$2^{19937} - 1$
PCG64	100000	30687.084969	0.115253	0.031881	2^{128}
Philox	100000	30662.317191	0.034449	0.035765	2^{256}
SFC64	100000	30629.583167	0.072344	0.030406	2^{64}

Table 2: Comparison of Different RNGs in Monte Carlo Simulatio

The results in Table 2 demonstrate how different PRNGs impact the accuracy and efficiency of Monte Carlo-based option pricing. While Mersenne Twister (MT19937) has an exceptionally long period of $2^{19937} - 1$, it exhibits the highest error percentage, suggesting that its statistical properties may not be ideal for financial applications. Philox, on the other hand, achieves the lowest error but requires slightly more computational time. SFC64 is the fastest among the tested generators, making it suitable for large-scale simulations where efficiency is a priority. Selecting an appropriate PRNG involves a trade-off between accuracy and computational speed, with different applications favoring different characteristics.

Antithetic Variables The antithetic variates method is a variance reduction technique that aims to improve the stability and convergence of Monte Carlo simulations. When sampling from the standard normal distribution, as described in Equation (5), we not only use the random variable $\phi \sim \mathcal{N}(0,1)$ but also its negation, $-\phi$, to generate an additional path. Using the risk-neutral dynamics given in Equation (6), the two corresponding asset price paths at time T are:

$$S_T = f(S_0, T) + v(S_0, T)\sqrt{T}\phi, \tag{7}$$

$$S'_T = f(S_0, T) + v(S_0, T)\sqrt{T}(-\phi). \tag{8}$$

The option price estimate is computed as the average of the two discounted payoffs:

$$V = \frac{1}{2}\left(e^{-rT}\max(S_T - K, 0) + e^{-rT}\max(S'_T - K, 0)\right). \tag{9}$$

The primary advantage of this method is that since ϕ and $-\phi$ are perfectly negatively correlated, the constructed paths should theoretically reduce variance, thereby improving the stability of the estimate. By mitigating the stochastic impact of volatility on option pricing, the Monte Carlo method is expected to converge more rapidly.

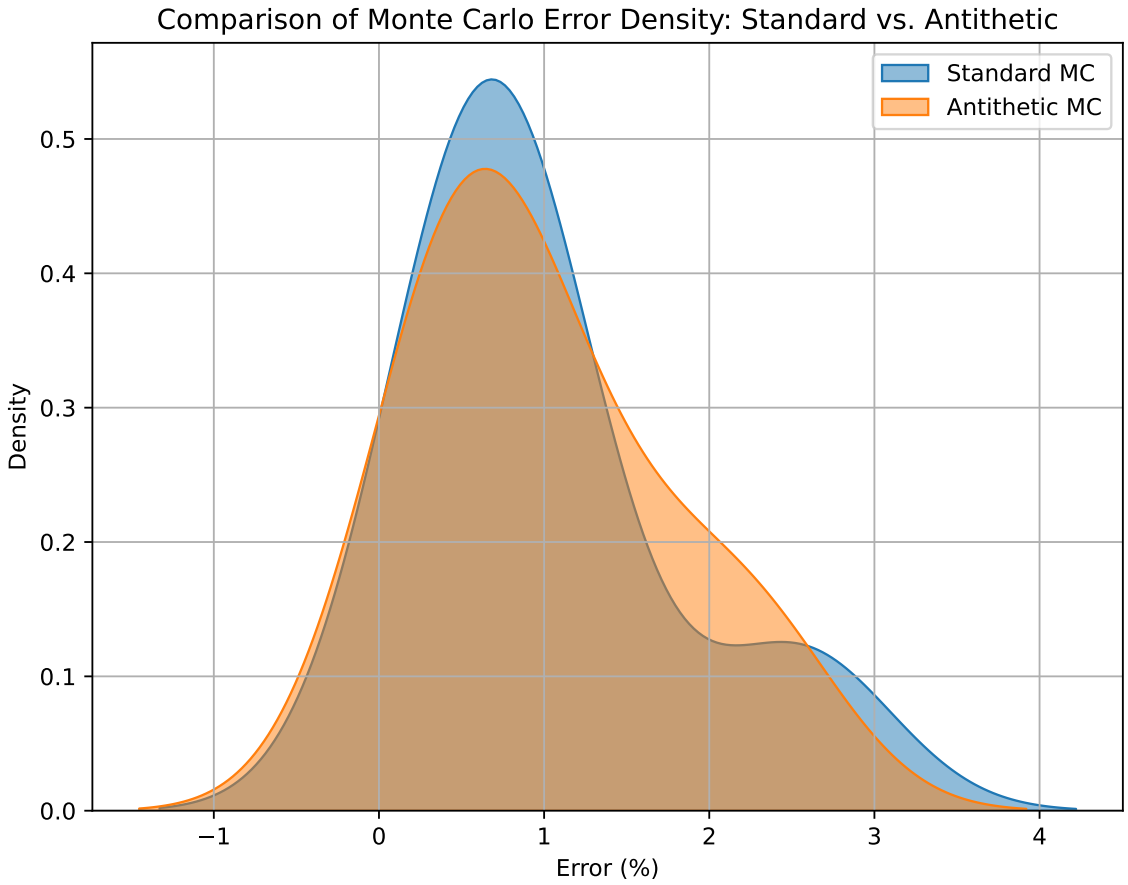


Figure 2: Comparison of Monte Carlo Error Distributions: Standard and Antithetic

However, as shown in Figure 2, which compares the error distributions of standard Monte Carlo and the antithetic variates method, the application of the latter does not significantly reduce the error. This suggests that the method is ineffective in this case, and several factors may contribute to this observation:

- **Dependence on Path Symmetry:** Higham (2004) in *An Introduction to Financial Option Valuation* discusses that while the antithetic variates method generally reduces variance, its effectiveness depends on the symmetry of the sampled paths. Referring to Equations (2) and (3), where the functions $f(S_0, T)$ and $v(S_0, T)$ are defined, if the distribution of S_T and S'_T exhibits strong skewness, they may not be evenly distributed across different payoff intervals, weakening their negative correlation.

- **Nonlinearity in the Volatility Function:** The volatility function $v(S_0, T)$ is nonlinear and dependent on S_0^γ. This nonlinearity may introduce asymmetry, meaning that S_T and S'_T are not symmetrically distributed relative to the payoff structure, thereby diminishing the variance reduction effect of the antithetic variates method.

- **Impact of the Payoff Function:** Another critical issue arises from the payoff function itself, as described in Equation (1). The payoff is a piecewise constant function, meaning that regardless of how S_T and S'_T vary, their corresponding payoffs remain unchanged

within each interval. Consequently, if S_T and S'_T fall into the same segment of the payoff function, they yield identical payoffs, eliminating any potential variance reduction. For the antithetic variates method to be effective, the payoff function must have a continuous dependence on S_T. Without such dependence, the fundamental mechanism that drives variance reduction—negative correlation in sampled values—fails to operate, rendering the method ineffective in this specific pricing model.

Moment Matching The primary objective of Moment Matching is to reduce the stochastic error introduced by finite sample sizes. This method is particularly effective in Monte Carlo simulations that assume normality, making it well-suited for our option pricing framework. The approach involves adjusting the generated sample ϕ to ensure that its mean is precisely zero and its variance is exactly one. Specifically, the transformed sample ϕ' is computed as:

$$\phi' = \frac{\phi - \bar{\phi}}{\sigma_\phi}, \tag{10}$$

where $\bar{\phi}$ is the sample mean, σ_ϕ is the sample standard deviation, and ϕ' is the adjusted sample satisfying $E[\phi'] = 0$ and $\mathrm{Var}[\phi'] = 1$. By centering the sample mean at zero, Moment Matching ensures that the terminal asset price distribution S_T in financial contracts is more accurately positioned. This adjustment reduces the random error inherent in Monte Carlo estimation, leading to faster convergence. The error of Moment Matching Monte Carlo (MM-MC) should be lower than that of standard Monte Carlo (Standard MC), particularly when the sample size N is small.

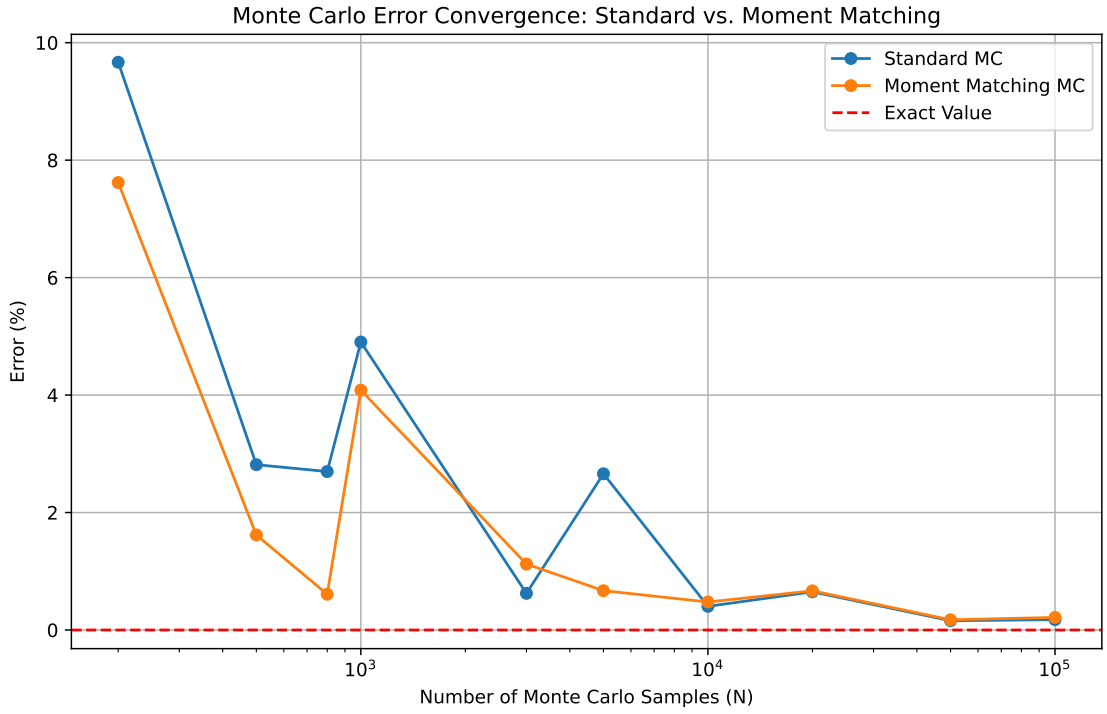


Figure 3: Monte Carlo Error Convergence: Standard and Moment Matching

Moment Matching is essentially a linear transformation applied to the standard normal samples

ϕ used in Monte Carlo simulations. This transformation strictly enforces the conditions that the sample mean equals zero and the sample variance equals one. The computational complexity of this process is $O(N)$, scaling linearly with the number of samples. Given that the overall complexity of Monte Carlo option pricing is also $O(N)$, the additional computational cost of Moment Matching is negligible, and the execution time remains nearly identical to that of standard Monte Carlo.

As illustrated in Figure 3, the empirical results align with theoretical expectations. When the sample size is small ($N < 10^3$), Moment Matching significantly reduces error. However, for large sample sizes ($N > 10^4$), its effect becomes marginal. This phenomenon occurs because, in large samples, the mean and variance of randomly generated normal variables naturally converge to their theoretical values due to the Central Limit Theorem. Since Moment Matching merely applies a linear transformation, its capacity for additional error reduction is inherently limited.

Halton Sequence The Halton sequence is an extension of the Van der Corput sequence, which distributes points within the unit interval by systematically jumping in a way that ensures balanced coverage. A Halton sequence in multiple dimensions is constructed by assigning a different Van der Corput sequence to each dimension, using distinct prime numbers as bases (Brandimarte, 2014). Compared to purely random sampling, Halton low-discrepancy sequences achieve a more uniform distribution over the interval, reducing stochastic fluctuations between samples and improving estimation accuracy.

Method	**N**	**Price**	**Error (%)**	**Time (s)**
Standard MC	1,000	31,840.839	3.879	0.002
Halton MC	1,000	30,546.927	0.342	0.006
Standard MC	5,000	29,974.323	2.210	0.004
Halton MC	5,000	30,666.662	0.049	0.006
Standard MC	10,000	30,966.483	1.027	0.004
Halton MC	10,000	30,648.316	0.011	0.007
Standard MC	50,000	30,383.257	0.876	0.018
Halton MC	50,000	30,652.661	0.003	0.036
Standard MC	100,000	30,472.624	0.584	0.039
Halton MC	100,000	30,651.406	0.001	0.089

Table 3: Comparison of Standard Monte Carlo and Halton Sequence Monte Carlo

Unlike Moment Matching, which primarily stabilizes small-sample performance, Halton sequences provide consistent stability across the entire sample range. As illustrated in Table 3, the observed error is significantly reduced throughout the simulation, confirming our expectations. One potential reason for this improvement is that if the sampled points cluster around X_1, the estimation error may be amplified due to small perturbations causing abrupt changes in the payoff function. Since Halton sequences are uniformly distributed, they mitigate the risk of local sample deficiencies, thereby enhancing both the stability and accuracy of the Monte Carlo estimate.

However, this improved precision comes at an increased computational cost, with the execution time approximately 2.5 times that of standard MC. This performance difference arises because Halton sequences rely on prime numbers to generate a well-distributed sample set. While Halton

sequences are naturally uniform over the interval $[0, 1]$, they must be transformed into a normal distribution using the inverse cumulative distribution function. This inverse transformation introduces additional computational overhead, as it requires numerical inversion, which is more expensive than directly generating standard normal random numbers with `np.random.randn()`. Despite this, the reduced variance and the ability to achieve the same error level with fewer samples N ultimately lead to faster overall computation, making Halton sequences a worthwhile alternative to standard MC.

Conclusion This study compares four variance reduction techniques in Monte Carlo option pricing: different random number generators, antithetic variates, moment matching and Halton sequences. The codes are in Figure 10. The results demonstrate that while standard random number generators provide a baseline for comparison, their effectiveness is limited by inherent stochastic fluctuations. Antithetic variates is ineffective when the payoff function is piecewise constant. Moment matching improves estimation accuracy, particularly for small sample sizes, by enforcing strict mean and variance constraints on the generated normal samples. However, its effect diminishes as the sample size increases due to the natural convergence of sample moments.

Among the methods tested, the Halton sequence outperforms the others by achieving significantly lower error rates across all sample sizes. Unlike antithetic variates and moment matching, which primarily stabilize variance under specific conditions, Halton sequences consistently enhance accuracy by ensuring a more uniform sample distribution. Although the computational cost of generating Halton sequences is higher due to the inverse cumulative distribution function transformation, this additional cost is offset by requiring fewer samples to achieve the same error level. As a result, Halton sequences offer the most efficient balance between accuracy and computational efficiency, making them the superior choice for Monte Carlo-based European biased option pricing.

2 Up-and-Out Barrier Call Option

We price a discrete up-and-out barrier call option V, assuming the underlying asset follows the risk-neutral process:

$$dS = (\alpha\theta - \beta S)dt + \sigma(|S|)^{\gamma} dW, \tag{11}$$

where S is the asset price, W is a standard Wiener process, and $\alpha, \beta, \gamma, \theta, \sigma$ are constant parameters.

The asset price is observed at $K + 1$ equally spaced times:

$$t_0, t_1, ..., t_k = k\Delta t, ..., t_K, \quad t_0 = 0, \quad t_K = T, \quad \Delta t = \frac{T}{K}.$$

Using an Euler discretization scheme, the asset price evolves as:

$$S^i(t_k) = S^i(t_{k-1}) + f(S^i(t_{k-1}), t_{k-1})\Delta t + v(S^i(t_{k-1}), t_{k-1})\sqrt{\Delta t}\,\phi_{i,k-1}, \tag{12}$$

where $\phi_{i,k} \sim N(0, 1)$ is a standard normal variable.

The option payoff depends on the full price path $\{S(t_0), ..., S(t_K)\}$, and its value is approximated via Monte Carlo:

$$V(S(t_0), t = 0) \approx e^{-rT} \frac{1}{n} \sum_{i=1}^{n} g(S^i(t_0), ..., S^i(t_K)). \tag{13}$$

2.1 Monte Carlo Approximation of the Option Value

To estimate the value of the up-and-out barrier call option, we use a standard Monte Carlo approximation with $N = 100,000$ simulated paths. The option value is computed as:

$$V \approx 0.2901948436956547.$$

The implementation of the Monte Carlo simulation is provided in Figure 11.

2.2 Convergence Analysis

To investigate the convergence of the Monte Carlo estimation, we performed simulations with varying numbers of paths N. Figure 4 presents the convergence behavior of the Monte Carlo approximation, illustrating how the option value stabilizes as N increases.

The final computed option price at $N = 1,000,000$ is:

$$V \approx 0.2850.$$

After multiple runs, this value remains consistent, making it our best estimate.

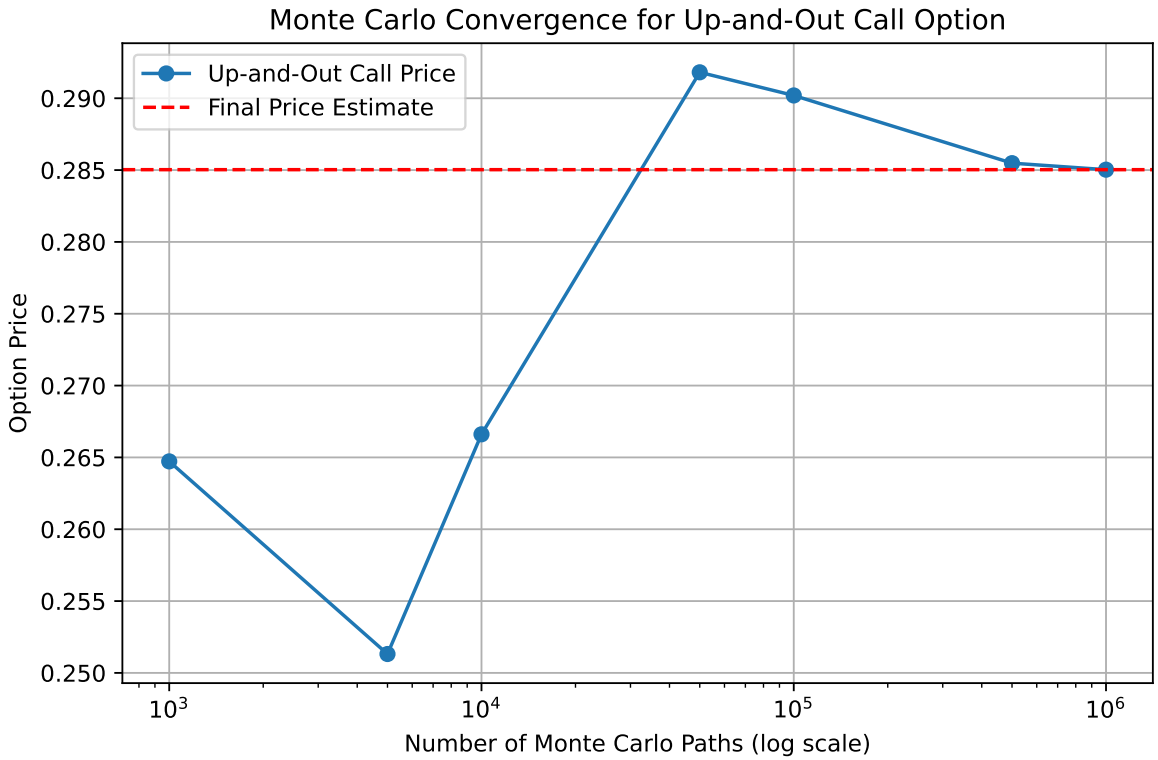


Figure 4: Convergence of Monte Carlo Simulation for Up-and-Out Barrier Call Option

In *The Mathematics of Financial Derivatives*, Wilmott, Howison, and Dewynne derive the pricing formula for knock-out options using the Method of Images. However, it is disappointing that the up-and-out call option requires solving within the constrained region $S < X$, which prevents the simple use of symmetric reflection to satisfy the boundary conditions. Therefore,

it cannot be directly solved using the Method of Images, and within the standard Black-Scholes framework, no closed-form solution exists. Consequently, we proceed with Monte Carlo-based methods for its valuation.

2.3 Sensitivity Analysis with Respect to β

Greeks play an essential role in understanding how model parameters affect the price of path-dependent options such as the up-and-out barrier call. In particular, the partial derivative $\frac{\partial V}{\partial \beta}$ measures how sensitive the option value V is to changes in the parameter β. In our asset as informed in Equation (11), the parameter β controls the mean-reverting speed of the process. A higher β leads to a faster convergence of S towards its long-term equilibrium $\frac{\alpha\theta}{\beta}$, while a lower β results in a slower adjustment and greater persistence of deviations. Accurate estimates of this sensitivity can inform hedging and risk management decisions.

To estimate $\frac{\partial V}{\partial \beta}$ at S_0 and $t = 0$ using Monte Carlo, we compute two option values at $\beta = 0.03$ and $\beta = 0.03 + \Delta\beta$, then apply a simple finite difference:

$$\frac{\partial V}{\partial \beta} \approx \frac{V(S_0,\, t=0;\, \beta = 0.03 + \Delta\beta) \;-\; V(S_0,\, t=0;\, \beta = 0.03)}{\Delta\beta}. \tag{14}$$

For example, when we set $\Delta\beta = 0.001$, we have $V(\beta = 0.03) = 0.2902, V(\beta = 0.03 + 0.001) = 0.2900$ hence,

$$\frac{\partial V}{\partial \beta} \approx -0.235543.$$

This shows a slight negative sensitivity of the option value to β, indicating that increasing β marginally reduces the price of the up-and-out barrier call under our model and simulation setup.

Different Types of Monte Carlo Simulation We compare four Monte Carlo simulation techniques: Halton Sequence (QMC), Importance Sampling, Antithetic Variates, and Control Variates. The implementation of the Control Variates method is detailed in Appendix A, while the corresponding code is provided in Figure 12, 13, 14 and 15.

Following Higham (2004), we conduct multiple trials and compute 95% confidence intervals to quantitatively compare the accuracy and variance of standard Monte Carlo (MC) and different MC techniques in estimating $\frac{\partial V}{\partial \beta}$. The results from these four approaches are summarized in Table 4.

Table 4: Comparison of Different Monte Carlo Methods for Estimating $\frac{dV}{d\beta}$

Method	**Mean $\frac{\mathbf{dV}}{\mathbf{d}\beta}$**	**CI Lower**	**CI Upper**	**CI Width**
Standard MC	8.252545	-4.813316	21.318406	26.131722
Quasi-MC (QMC)	10.963110	2.614557	19.311663	16.697106
Importance Sampling	5.256979	-3.697449	14.211408	17.908857
Antithetic Variates	8.249688	-0.513314	17.012689	17.526003
Control Variate	8.319935	-4.624250	21.264121	25.888371

From the comparative analysis, we find that under the same simulation scale, the Antithetic Variates method provides the most stable results with a relatively narrow confidence interval. This is likely due to the fact that the up-and-out barrier option is a highly path-dependent derivative,

and antithetic variates are often highly effective in high-dimensional, strong path-dependency settings. Unlike control variates, which require well-correlated auxiliary variables, or importance sampling, which depends on careful drift adjustment, the antithetic variates method achieves significant variance reduction as long as the payoff function maintains sufficient monotonicity with respect to the underlying randomness.

The smallest confidence interval is observed for the QMC method. However, for high-dimensional, nonlinear integration problems, QMC can introduce systematic bias if appropriate randomization are not applied. This phenomenon "biased QMC" arises when low-discrepancy sequences fail to sufficiently explore the stochastic space.

Effect of N We first fix $\Delta\beta = 0.001$ and analyze the sensitivity of the option value estimation to the number of Monte Carlo paths, N.

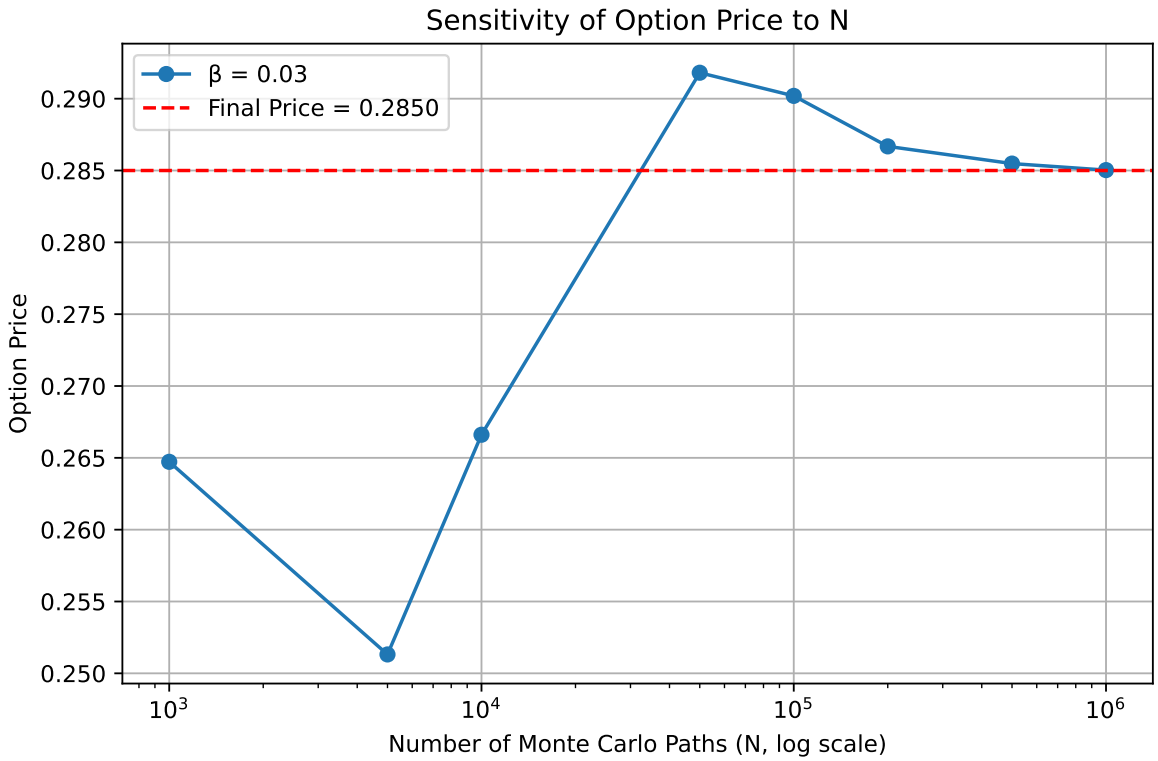


Figure 5: Sensitivity of Option Price to N

Using the implementation in Figure 13, we compute $V(S_0, t = 0; \beta = 0.03)$ for different values of N and plot its convergence behavior. This analysis is necessary because the derivative $\frac{\partial V}{\partial \beta}$ does not have a known exact value, whereas $V(S_0, t = 0; \beta = 0.03)$ has a final estimated value.

Figure 5 illustrates that the option price estimate stabilizes only when N exceeds approximately 200,000. This suggests that a sufficiently large number of Monte Carlo paths is required to obtain reliable estimates.

Effect of $\Delta\beta$ We perform a sensitivity test on $\Delta\beta$ using the finite difference formula (14) to assess how different values of $\Delta\beta$ affect the stability of the derivative estimate. By plotting the

convergence of $\frac{\partial V}{\partial \beta}$ for various choices of $\Delta\beta$, we determine the optimal range for minimizing numerical errors. The implementation is shown in Figure 13.

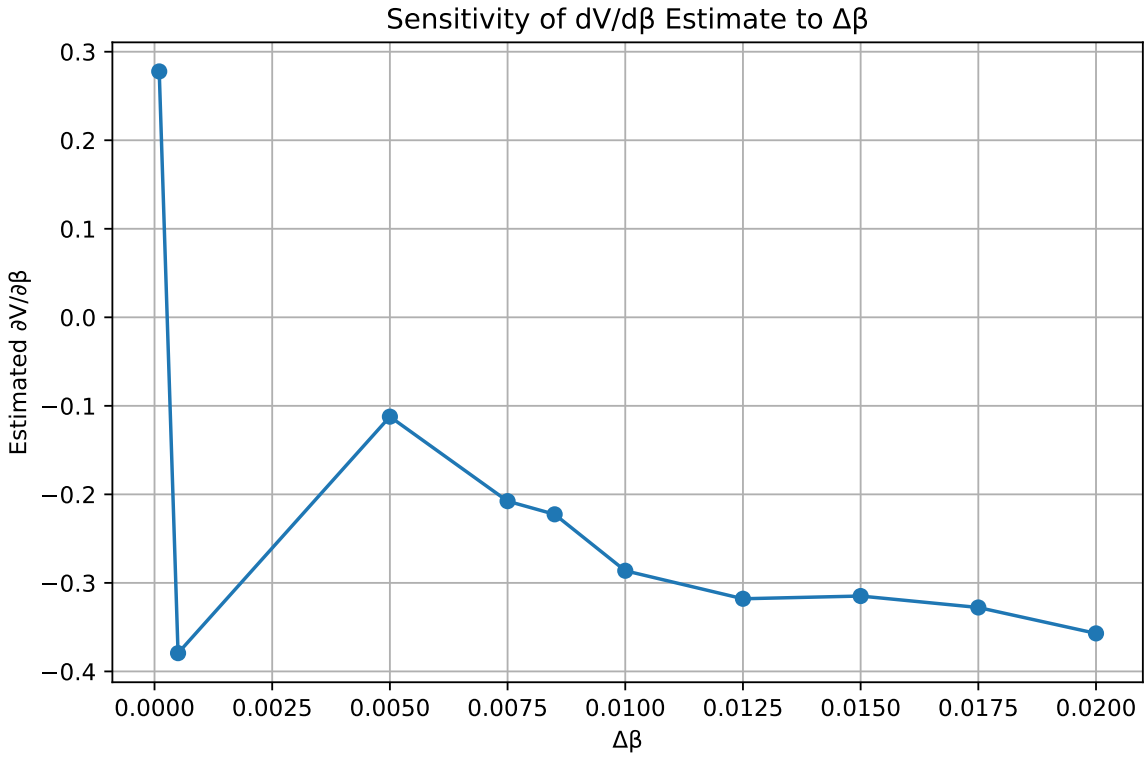


Figure 6: Sensitivity of $dV/d\beta$ Estimate to $\Delta\beta$

As depicted in Figure 6, the derivative estimate remains relatively stable for $\Delta\beta$ values around 0.001 (from 0.001-0.015). This suggests that excessively small $\Delta\beta$ introduces numerical instability, whereas excessively large values may lead to discretization errors.

Conclusion In this study, we implemented a Monte Carlo-based approach to price a discrete up-and-out barrier call option and analyzed the sensitivity of the option value with respect to the model parameter β. Through sensitivity analysis, we found that the Monte Carlo estimate of $V(S_0, t = 0; \beta = 0.03)$ stabilizes for $N > 200,000$, while an optimal choice of $\Delta\beta \approx 0.001$ provides a balance between numerical stability and discretization error.

Furthermore, we compared different Monte Carlo simulation techniques, including QMC, Importance Sampling, Antithetic Variates, and Control Variates. The Antithetic Variates method exhibited the most stable results with a relatively narrow confidence interval, making it the most effective technique for this path-dependent derivative. While QMC achieved the lowest variance, it also introduced systematic bias, highlighting the trade-offs between variance reduction and unbiasedness. Overall, our findings underscore the importance of selecting appropriate variance reduction methods when estimating sensitivities in highly path-dependent options.

References

[1] S. Asmussen and P. W. Glynn, *Stochastic Simulation: Algorithms and Analysis*. New York: Springer, 2007.

[2] J. S. Dagpunar, *Simulation and Monte Carlo: With Applications in Finance and MCMC*. School of Mathematics, University of Edinburgh, UK, 2007.

[3] D. Higham, *An Introduction to Financial Option Valuation: Mathematics, Stochastics and Computation*. Cambridge University Press, 2004.

[4] P. Brandimarte, *Handbook in Monte Carlo Simulation: Applications in Financial Engineering, Risk Management, and Economics*. John Wiley & Sons, 2014.

[5] P. Wilmott, S. Howison, and J. Dewynne, *The Mathematics of Financial Derivatives: A Student Introduction*. Cambridge University Press, 1995.

[6] J. C. Hull, *Options, Futures, and Other Derivatives*. Pearson, 2017.

[7] M. Capinski and T. Zastawniak, *Stochastic Calculus for Finance*. Cambridge University Press, 2004.

[8] Á. Cartea, S. Jaimungal, and J. Penalva, *Algorithmic and High-Frequency Trading*. Cambridge University Press, 2015.

[9] Author(s), "A model-free approximation for barrier options in a general stochastic volatility framework," *Journal of Futures Markets*, vol. 44, no. 6, pp. 877-1094, Jun. 2024.

Appendix A: Variance Reduction Method - Control Variate

To improve the Monte Carlo simulation of an up-and-out call option, we apply the control variates method, which helps reduce variance in the finite difference estimation of the derivative indicated in Equation (14). The approach proceeds as follows. For a given value of β, we compute the following within the same Monte Carlo simulation:

First, we determine the payoff of the barrier option, denoted as X. Specifically, if the asset path does not breach the barrier, the payoff is given by $X = \max(S_T - X, 0)$ whereas if the barrier is breached at any point, the payoff is zero.

Next, we compute the payoff of the corresponding European call option, denoted as Y, which is given by $Y = \max(S_T - X, 0)$ regardless of whether the path crosses the barrier.

To reduce variance, we utilize the theoretical price of the European call option, denoted as μ, computed using the Black-Scholes formula. This serves as the control variate target value. We then adjust the barrier option's payoff for each simulated path using:

$$X_{\text{adj}} = X + c(\mu - Y), \tag{15}$$

where the optimal coefficient c^* is determined by minimizing variance and is given by:

$$c^* = \frac{\text{Cov}(X, Y)}{\text{Var}(Y)}. \tag{16}$$

Finally, the control variate-adjusted option price is obtained by averaging X_{adj} over all simulated paths and discounting accordingly.

To estimate the derivative $\frac{\partial V}{\partial \beta}$, we compute option prices using the control variates method for both $\beta = 0.03$ and $\beta = 0.03 + \Delta\beta$. The finite difference formula is then applied to obtain the sensitivity estimate. Corresponding code are in Figure 15.

Appendix B: Simulation Python Code

```python
# All library used in code
import numpy as np
import scipy.integrate as integrate
import scipy.stats as stats
import pandas as pd
import matplotlib.pyplot as plt
from scipy.stats import norm
# Parameters for European Contract Pricing
S0 = 74827.5
T = 1.75
r = 0.02
alpha = 0.06
beta = 0.04
theta = 74832
sigma = 0.17
gamma = 0.94
X1 = 75000
X2 = 85000
# Payoff function
def g_payoff(S):
    if S < X1:
        return X1 - X2 # -10000
    elif X1 <= S < X2:
        return X1      # 75000
    else:
        return X2 - X1 # 10000
```

Figure 7: Parameters of European Contract

```
# Analytic solution
f_value = S0 * (np.exp(alpha * T) - np.exp(beta * T)) + theta * (1 + np.tanh(alpha * T)
    - beta * T)
v_value = sigma * np.sqrt(1 + alpha * T) * (S0 ** gamma)
v_sqrtT = v_value * np.sqrt(T)
P1 = stats.norm.cdf((75000 - f_value) / v_sqrtT)
P2 = stats.norm.cdf((85000 - f_value) / v_sqrtT) - stats.norm.cdf((75000 - f_value) /
    v_sqrtT)
P3 = 1 - (P1 + P2)
expected_payoff = (-10000) * P1 + 75000 * P2 + 10000 * P3
C0_exact = np.exp(-r * T) * expected_payoff
```

Figure 8: Python code for Analytic solution

```
def mc_price(S0, T, r, alpha, beta, theta, sigma, gamma, N=100000):
    # Compute the mean and standard deviation of the normal distribution
    f_value = S0 * (np.exp(alpha * T) - np.exp(beta * T)) \
              + theta * (1 + np.tanh(alpha * T) - beta * T)
    v_value = sigma * np.sqrt(1 + alpha * T) * (S0 ** gamma)
    v_sqrtT = v_value * np.sqrt(T)
    # Generate N standard normal samples
    phi = np.random.randn(N)
    S_T = f_value + v_sqrtT * phi
    payoffs = np.array([g_payoff(s) for s in S_T])
    return np.exp(-r * T) * np.mean(payoffs)
# Simulation
N_values = [1000, 2000, 3000, 4000, 5000, 7500, 10000, 15000, 20000, 25000, 30000,
    35000, 40000, 45000, 50000, 75000, 100000]
mc_results = []
for N in N_values:
    price_est = mc_price(S0, T, r, alpha, beta, theta, sigma, gamma, N)
    mc_results.append(price_est)
# Monte Carlo Estimation (with multiple runs to smooth results)
mc_results = np.array([np.mean([mc_price(S0, T, r, alpha, beta, theta, sigma, gamma, N)
                                for _ in range(10)]) for N in N_values])
# Compute absolute error
errors = np.abs(np.array(mc_results) - C0_exact)
```

Figure 9: Python code for Monte-Carlo Simulation of European Option

```python
# Different RNG
rng_list = {
    "Default (PCG64)": np.random.default_rng(),
    "MT19937 (Mersenne Twister)": np.random.RandomState(),
    "PCG64": np.random.Generator(np.random.PCG64()),
    "Philox": np.random.Generator(np.random.Philox(10)),
    "SFC64": np.random.Generator(np.random.SFC64()),}
# Antithetic Monte Carlo
def mc_price_antithetic(S0, T, r, alpha, beta, theta, sigma, gamma, N=100000, rng=None):
    if rng is None:
        rng = np.random.default_rng()
    N_half = N // 2
    phi = rng.standard_normal(N_half)
    phi_antithetic = -phi
    S_T1 = f_value + v_sqrtT * phi
    S_T2 = f_value + v_sqrtT * phi_antithetic
    payoffs1 = np.array([g_payoff(s) for s in S_T1])
    payoffs2 = np.array([g_payoff(s) for s in S_T2])
    payoffs = 0.5 * (payoffs1 + payoffs2) # take average
    return np.exp(-r * T) * np.mean(payoffs)
# Moment Matching Monte Carlo
def mc_price_moment_matching(S0, T, r, alpha, beta, theta, sigma, gamma, N=100000, rng=
    None):
    if rng is None:
        rng = np.random.default_rng()
    phi = rng.standard_normal(N)
    phi_mean = np.mean(phi)
    phi_std = np.std(phi)
    phi_matched = (phi - phi_mean) / phi_std
    S_T = f_value + v_sqrtT * phi_matched
    payoffs = np.array([g_payoff(s) for s in S_T])
    return np.exp(-r * T) * np.mean(payoffs)
# Halton Sequence Monte Carlo
def mc_price_halton(S0, T, r, alpha, beta, theta, sigma, gamma, N=100000):
    sampler = Halton(d=1, scramble=True)
    phi = sampler.random(N).flatten() #generate Halton Sequence
    phi = norm.ppf(phi) # Transfer to Normal Distrbution
    S_T = f_value + v_sqrtT * phi
    payoffs = np.array([g_payoff(s) for s in S_T])
    return np.exp(-r * T) * np.mean(payoffs)
```

Figure 10: Python code for 1.3, Variance Reduction Techniques

```python
# 1. Define Parameters
S0 = 65.15665 # Initial stock price
X  = 65       # Strike price
B  = 77.9     # Upper barrier level
T  = 0.5      # Maturity time (in years)
K  = 65       # Number of discrete sampling steps
r  = 0.03     # Risk-free interest rate
alpha = 0.02
beta  = 0.03
gamma = 1.07
sigma = 0.36
theta = 65.1404
N = 100000    # Number of Monte Carlo simulation paths
dt = T / K # Time step size
# 2. Monte Carlo Simulation
np.random.seed(42) # Set random seed for reproducibility
payoffs = np.zeros(N)
for i in range(N):
    S_t = S0
    barrier_hit = False
    # Generate K standard normal random variables for the path
    Z = np.random.randn(K)
    for k in range(K):
        dW = np.sqrt(dt) * Z[k]# Compute the Wiener process increment dW
        # Euler-Maruyama discretization:
        drift = (alpha * theta - beta * S_t) * dt
        diffusion = sigma * (abs(S_t)**gamma) * dW
        S_t = S_t + drift + diffusion
        if S_t > B:# Check if the barrier level is breached
            barrier_hit = True
            break
    if not barrier_hit:
        payoffs[i] = max(S_t - X, 0)
option_price = np.exp(-r * T) * np.mean(payoffs) # Compute Option Price
# Define different values of N (number of Monte Carlo simulation paths)
n_values = [500, 2000, 5000, 10000, 50000, 100000, 500000, 1000000]
option_prices = []
# Set a fixed random seed to ensure comparability across different values of N
np.random.seed(42)
for n in n_values:
    price = simulate_option_price(n)
    option_prices.append(price)
    print(f"Number of paths: {n}, Option Price: {price:.4f}")
```

Figure 11: Python code for Up-and Out Call Option Pricing

```python
def monte_carlo_up_and_out(n_paths, beta_value):
    payoffs = np.zeros(n_paths)
    np.random.seed(42) # Fixed seed for reproducibility
    for i in range(n_paths):
        S_t = np.full(K + 1, S0)
        Z = np.random.randn(K) # Generate standard normal variables
        for k in range(K):
            dW = np.sqrt(dt) * Z[k]
            drift = (alpha * theta - beta_value * S_t[k]) * dt
            diffusion = sigma * (abs(S_t[k]) ** gamma) * dW
            S_t[k + 1] = S_t[k] + drift + diffusion
        # Check for barrier crossing
        if np.max(S_t) > B:
            payoffs[i] = 0 # Knocked out, no value
        else:
            payoffs[i] = max(S_t[-1] - X, 0)
    option_price = np.exp(-r * T) * # Discounting the payoff
    return option_price
# Quasi-MC (QMC) - Based on Halton Sequence
def qmc_up_and_out(n_paths, beta_value, seed=42):
    U = sampler.random(n_paths)
    Z_mat = norm.ppf(U) # shape=(n_paths, K)
    payoffs = np.zeros(n_paths)
    for i in range(n_paths):
        S_t = S0
        barrier_hit = False
        Z = Z_mat[i, :]
        for k in range(K):
            dW = np.sqrt(dt) * Z[k]
            drift = (alpha * theta - beta_value * S_t) * dt
            diffusion = sigma * (abs(S_t)**gamma) * dW
            S_t += drift + diffusion
            if S_t > B:
                barrier_hit = True
                break
        if barrier_hit:
            payoffs[i] = 0.0
        else:
            payoffs[i] = max(S_t - X, 0)
    return np.exp(-r*T) * np.mean(payoffs)
def mean_confidence_interval(data, confidence=0.95):
    mean_val = np.mean(data)
    std_val = np.std(data, ddof=1)
    n = len(data)
    z = 1.96 # z distribution
    half_width = z * (std_val / np.sqrt(n))
    return mean_val, mean_val - half_width, mean_val + half_width
```

Figure 12: Python code for 2.3, Sensitivity Analysis - 1

```python
# Sensitivity Analysis for N
N_values = [1000, 5000, 10000, 50000, 100000, 200000, 500000, 1000000]
option_prices_baseline = []
errors = []
for n in N_values:
    price = monte_carlo_up_and_out(n, beta_base)
    option_prices_baseline.append(price)
    errors.append(abs(price - final_price)) # Compute absolute error
# Sensitivity Analysis for beta
V_beta = monte_carlo_up_and_out(N, beta_base) # Compute Base V(S0, beta=0.03)
# Define beta
delta_beta_values = [0.0001, 0.0005, 0.001, 0.005, 0.01, 0.0125, 0.015, 0.0175, 0.02]
derivative_estimates = []
for delta in delta_beta_values:
    V_beta_delta = monte_carlo_up_and_out(N, beta_base + delta)
    dV_dBeta = (V_beta_delta - V_beta) / delta
    derivative_estimates.append(dV_dBeta)
# Importance Sampling
def is_mc_up_and_out(n_paths, beta_value, seed=42, drift_adjust=0.1):
    np.random.seed(seed)
    payoffs = np.zeros(n_paths)
    for i in range(n_paths):
        S_t = S0
        weight = 1.0 # initialize weight
        barrier_hit = False
        Z = np.random.randn(K)
        for k in range(K):
            dW = np.sqrt(dt) * Z[k]
            drift = (alpha * theta - beta_value * S_t + drift_adjust) * dt
            diffusion = sigma * (abs(S_t)**gamma) * dW
            S_t = S_t + drift + diffusion
            weight *= np.exp(-drift_adjust * Z[k] - 0.5 * (drift_adjust**2) * dt) #Update
                Weight
            if S_t > B:
                barrier_hit = True
                break
        if barrier_hit:
            payoffs[i] = 0.0
        else:
            payoffs[i] = max(S_t - X, 0) * weight
    option_price = np.exp(-r*T) * np.mean(payoffs)
    return option_price
```

Figure 13: Python code for 2.3, Sensitivity Analysis - 2

```python
# Antithetic Variates
N = 10000      # Needs to be even
delta_beta = 0.001
M = 10         # Repeat times
def av_mc_up_and_out(n_paths, beta_value, seed=42):
    if n_paths % 2 != 0:
        n_paths += 1 # Make sure n_paths is even
    np.random.seed(seed)
    n_pairs = n_paths // 2
    pair_payoffs = np.zeros(n_pairs)
    for i in range(n_pairs):
        Z = np.random.randn(K)
        S_t1 = S0 # Path 1 use Z
        S_t2 = S0 # Path 2 use -Z
        barrier_hit1 = False
        barrier_hit2 = False
        for k in range(K):
            dW1 = np.sqrt(dt) * Z[k]
            dW2 = np.sqrt(dt) * (-Z[k])
            drift1 = (alpha * theta - beta_value * S_t1) * dt
            diffusion1 = sigma * (abs(S_t1)**gamma) * dW1
            drift2 = (alpha * theta - beta_value * S_t2) * dt
            diffusion2 = sigma * (abs(S_t2)**gamma) * dW2
            S_t1 = S_t1 + drift1 + diffusion1
            S_t2 = S_t2 + drift2 + diffusion2
            if S_t1 > B:
                barrier_hit1 = True
            if S_t2 > B:
                barrier_hit2 = True
            if barrier_hit1 and barrier_hit2:
                break
        payoff1 = 0.0 if barrier_hit1 else max(S_t1 - X, 0)
        payoff2 = 0.0 if barrier_hit2 else max(S_t2 - X, 0)
        pair_payoffs[i] = 0.5 * (payoff1 + payoff2)
    option_price = np.exp(-r * T) * np.mean(pair_payoffs)
    return option_price
# Repeat the experiment M times
def estimate_derivative_av(n_paths, delta_beta, M):
    estimates = []
    for i in range(M):
        V1 = av_mc_up_and_out(n_paths, beta_base, seed=42+i)
        V2 = av_mc_up_and_out(n_paths, beta_base + delta_beta, seed=123+i)
        dV = (V2 - V1) / delta_beta
        estimates.append(dV)
    return np.array(estimates)
av_estimates = estimate_derivative_av(N, delta_beta, M)
```

Figure 14: Python code for 2.3, Sensitivity Analysis - 3

```python
# Control Variate
def european_call_bs(S0, X, T, r, sigma):
    d1 = (np.log(S0/X) + (r+0.5*sigma**2)*T) / (sigma*np.sqrt(T))
    d2 = d1 - sigma*np.sqrt(T)
    price = S0*norm.cdf(d1) - X*np.exp(-r*T)*norm.cdf(d2)
    return price
mu_euro = european_call_bs(S0, X, T, r, sigma)
def cv_mc_up_and_out(n_paths, beta_value, seed=42):
    np.random.seed(seed)
    X_payoffs = np.zeros(n_paths) # Barrier Option payoff
    Y_payoffs = np.zeros(n_paths) # European Call Option payoff
    for i in range(n_paths):
        S = S0
        barrier_hit = False
        Z = np.random.randn(K)
        for k in range(K):
            dW = np.sqrt(dt) * Z[k]
            drift = (alpha * theta - beta_value * S) * dt
            diffusion = sigma * (abs(S)**gamma) * dW
            S = S + drift + diffusion
            if S > B:
                barrier_hit = True
                break
        if barrier_hit:
            X_payoffs[i] = 0.0
        else:
            X_payoffs[i] = max(S - X, 0)
        Y_payoffs[i] = max(S - X, 0)# European Call Option Payoff
    covXY = np.cov(X_payoffs, Y_payoffs, ddof=1)[0,1] # Estimate best coefficient c
    varY = np.var(Y_payoffs, ddof=1)
    c_opt = covXY / varY if varY > 0 else 0.0
    X_adj = X_payoffs + c_opt * (mu_euro - Y_payoffs)
    option_price = np.exp(-r * T) * np.mean(X_adj)
    return option_price
# Repeat the experiment M times
def estimate_derivative_cv(n_paths, delta_beta, M):
    estimates = []
    for i in range(M):
        V1 = cv_mc_up_and_out(n_paths, beta_base, seed=42+i)
        V2 = cv_mc_up_and_out(n_paths, beta_base + delta_beta, seed=123+i)
        dV = (V2 - V1) / delta_beta
        estimates.append(dV)
    return np.array(estimates)
cv_estimates = estimate_derivative_cv(N, delta_beta, M)
```

Figure 15: Python code for 2.3, Sensitivity Analysis- 4