Options Pricing and Risk Analysis: A Monte Carlo-Based Evaluation of Variance

Archuthan Mohanathasan
King's College London

Abstract

This research study presents a comprehensive evaluation of Monte Carlo simulation methods for the pricing of European Options. It explores some variance reduction techniques – antithetic variates, delta control variates, gamma control variates and combinations of the previous - and the use of Option Greeks in pricing. To confirm and assess accuracy, the pricing, performance and Greeks of Monte Carlo methods are compared to that of the Black-Scholes Formula. The evaluation takes on the comparisons of standard error, computation time and payoff distribution to produce a conclusion. The Monte Carlo uncertainty across parameters: volatility, maturity and strike price, have been investigated using various means of quantitative analysis such as convergence studies and two-dimensional heatmaps. Results convey that the combination of antithetic, delta and gamma variance reduction techniques yields the highest efficiency, while remaining consistent with theoretical expectations. The study emphasises best practices in simulation design and provides a structured empirical evaluation of the trade-offs inherent in Monte Carlo-based option pricing.

1. Introduction

The Monte Carlo simulation is a fundamental numerical technique in quantitative finance for the pricing of derivatives, especially when no closed-form analytical solution exists. While the Black-Scholes model offers a closed-form pricing formula for European options, Monte Carlo remains necessary to validate risk measures, generalise to path-dependent derivatives, and estimate sensitivities (Greeks). However, Monte Carlo methods inherently face problems of high variance and slow convergence. Therefore, techniques for reducing variance become very important in attempts to improve the efficiency of simulation and arrive at reliable pricing results within reasonable computational budgets.

This paper formally evaluates various Monte Carlo enhancements applied to the pricing of European options. In particular, we analyse antithetic variates, delta control variates, gamma control variates, and combinations thereof. Each technique is compared to a vectorised baseline based on a harmonised simulation framework. Further supporting analysis-including convergence profiles, sensitivities to model parameters, and heatmap visualisations of standard error-shows how variance reduction affects the stability and accuracy of the simulation across different market conditions.

2. Literature Review

The first financial applications of Monte Carlo simulations date back to Boyle (1977), where the groundwork was laid for the use of simulation algorithms in the risk-neutral pricing approach. The theory of Monte Carlo methods in finance was fully elaborated in Glasserman (2004), where variance reduction techniques, discretisation algorithms, and statistical inference were exhaustively described. Antithetic variables and control variables, the other two methods surveyed in this paper, are proven variance reduction techniques using correlations and mathematical approximations. Broadie & Glasserman (1996) proposed the use of the pathwise derivative and the likelihood ratio techniques as rigorous methods for estimating Greeks in Monte Carlo simulation applications, which underpin the implementations surveyed in this study.

3. Methodology

This section provides an overview of the mathematical approach devised to calculate analytical option prices, model option payoffs, use variance reduction methods, and calculate Greeks.

3.1 Black-Scholes

Under the risk-neutral measure, the European option's price can be calculated using the following formula as given in the Black-Scholes model, when the asset grows based on geometric Brownian motion, the volatility is constant, and there is constant compounding. The Greeks are derived by differentiating the Black-Scholes formula.

3.2 Monte Carlo Simulation

The discounted expected payoffs in the Monte Carlo pricing are derived by simulating asset price paths based upon the discretised geometric Brownian motion process. The base simulation makes use of vectorised computing in the simulation of several paths to obtain the payoff distribution.

3.3 Variance Reduction Techniques

Various methods of variance reduction were applied:

- Antithetic variates: employs negatively correlated paths to decrease the variance of payoffs.
- Delta control variates. This model utilises the delta derived from the Black-Scholes model.
- Gamma control variates: utilises information about the curvature to reduce variance.

3.4 Monte Carlo Greeks

Greeks in Options were calculated using a combination of the pathwise derivative estimate and the central finite difference estimate. Delta utilises the pathwise estimator, whereas gamma, Vega, theta, and rho calculate the central difference values. The accuracy of the Greek estimates obtained by the Monte Carlo simulation can be verified by the analytical values.

4. Results

The analysis was conducted using the parameters specified in **Section 0.2: Simulation Parameters and Variable Definitions**.

4.1 Validation of Monte Carlo Greeks

The initial validation of the Monte Carlo framework was conducted by comparing the calculated Option Greeks against the analytical Black-Scholes (BS) values. The results, shown in the **Greeks comparison table (from Section 0.4.1)**, demonstrate a high degree of consistency between the two methods.

The Monte Carlo estimates for Delta, Gamma, Vega, Theta, and Rho are all within a narrow margin of the theoretical BS values. Critically, the deviations fall within the calculated Monte Carlo standard errors (MC SE), with minimal SE observed for Vega, Theta, and Rho. This confirms the accuracy of the simulation's sensitivity calculations and the overall stability of the implementation.

4.2 Comparison of Variance Reduction Techniques

The **"Bias vs Variance: MC Prices vs Analytical Price" plot (from Section 0.4.2)** illustrates the impact of each variance reduction technique on the option price estimate and its associated error. Before interpreting the plot, it is important to understand the two key sources of error:

1. **Variance (Standard Error):** This error arises from the stochastic nature of the simulation paths. The standard error (SE) bars on the plot represents this uncertainty. This error decreases as the number of simulation paths (M) increases, following the rate $SE \propto \frac{1}{\sqrt{M}}$. Variance reduction techniques are designed to shrink these SE bars without introducing bias.
2. **Bias:** This is a systematic deviation from the true option value, which can be introduced by factors like the discretisation of time steps. Unlike variance, bias does *not* decrease by increasing the number of paths (M).

The plot shows all methods produce estimates very close to the true analytical price (the dashed red line), indicating minimal bias. The key difference is in their variance:

- The 'Vectorised Baseline' and 'Gamma-based Control Variates' methods produced the largest standard error, with estimates fluctuating significantly around the true price.
- In contrast, the application of 'Antithetic Variates' and 'Delta Control' substantially reduced this error, yielding price estimates much closer to the BS value.
- The combined methods, 'Antithetic + Delta' and 'Antithetic + Delta + Gamma', yielded the tightest confidence intervals, visually confirming their superior accuracy.

4.3 Quantitative Efficiency Analysis

A detailed quantitative comparison of the methods is provided in the **quantitative benchmark table (from Section 0.4.3)**. While the 'Vectorised Baseline' was the fastest to compute (0.003359s), it also had the highest standard error (0.034317). 'Antithetic Variates' alone provided a significant 3.8x reduction in standard error.

Delta-based Control Variates offered a powerful 14.4x reduction in standard error, though at the cost of a 2.17x increase in computation time. The **'Antithetic, Delta AND Gamma Variates'** method was the clear winner in terms of accuracy, achieving a **remarkable 85.3x reduction in standard error** (from 0.034317 to 0.000402). While this method was the most computationally expensive (7.0x relative time), its gains in precision are an order of magnitude greater than its cost.

This trade-off is synthesised in the **"Monte Carlo Variants: Efficiency Comparison" bar chart (from Section 0.4.4)**, which plots the efficiency metric on a log scale. The metric is explicitly defined in the analysis notebook as:

$$Efficiency = \frac{1}{(SE)^2 \times Computation\ Time}$$

A higher bar indicates a method that achieves low variance in less time. The 'Antithetic, Delta AND Gamma Variates' method is shown to be exponentially more efficient than all other methods, followed by the 'Delta-based Control Variates' and 'Antithetic AND Delta Variates'. The 'Gamma-based Control Variates' performed similarly to the 'Vectorised Baseline', indicating its inefficiency as a standalone technique for these parameters.

4.4 Payoff Distribution Analysis

The underlying mechanism for this variance reduction is visible in the **payoff distribution histograms (from Section 0.4.5)**.

- The 'Vectorised Baseline' shows a wide, right-skewed distribution with a **sharp spike at the zero payoff**, which is expected as most paths expire out-of-the-money.
- The 'Gamma-based Control Variates' method also shows a wide, right-skewed distribution, explaining its high variance.
- 'Antithetic Variates' tightens this distribution significantly, pairing paths to reduce extreme outcomes and shifting the main spike closer to the expected payoff.
- The control variate methods ('Delta' and the combined methods) fundamentally alter the shape of the distribution, transforming it into a **tight, nearly symmetrical (Gaussian) distribution** centred on the true expected payoff.
- The 'Antithetic, Delta AND Gamma Variates' method produces the **narrowest distribution**, visually representing its extremely low standard error.

4.5 Convergence and Sensitivity Analysis

The study first confirmed the theoretical convergence rate of the Monte Carlo simulation using the **"SE vs M" convergence plots (from Section 0.4.6)**. The linear plot (left) shows the standard error decreasing as the number of paths (M) increases, following a decaying curve that flattens. The log-log plot (right) shows a clear linear relationship between the number of paths and the standard error. A linear fit of this log-log data yielded a slope of approximately **-0.5** (specifically, -0.487), confirming the expected $O\left(\frac{1}{\sqrt{M}}\right)$ convergence rate.

Finally, the sensitivity of the simulation's standard error (SE) to key option parameters was investigated.

- The **"Sensitivity of Price and SE to S" plot (from Section 0.4.7)** shows the option price (blue line) increasing as the underlying stock price (S) rises. This line clearly reflects the **option's convexity**: the price curve is "non-linear when the option is far out-of-the-money" and "become[s] approximately linear" as it moves past the strike. Critically, the standard error (red dashed line) is not constant; it **peaks when the option is at-the-money** (where, or in this case) and decreases as the option moves further in-the-money or out-of-the-money.
- The **"MC SE Heatmap: Volatility vs Maturity" (from Section 0.4.8)** shows that standard error increases with both **higher volatility and longer time to maturity**.
- The **"MC SE Heatmap: Strike vs Volatility" (from Section 0.4.9)** corroborates the finding from the sensitivity plot. The standard error is highest (brightest yellow) along the vertical band where the Strike Price is close to the Stock Price (around 101.15). This error is also clearly **amplified at higher volatilities**.

5. Conclusion

The work provides an exhaustive evaluation of the efficiency of various Monte Carlo pricing algorithms for European options. Variance reduction algorithms can significantly improve the accuracy of the estimates, the most efficient approach in this regard being the combined Antithetic, Delta, and Gamma approach. The rates of convergence also confirm the theoretical result of , where M denotes the number of simulated paths in the Monte Carlo algorithm. The numerical results show consistency between the Greeks in the algorithm and the analytical values predicted by the Black-Scholes model.

References

1. Boyle, P. (1977). Options: A Monte Carlo approach. *Journal of Financial Economics*, 4(3), 323-338.
2. Broadie, M., & Glasserman, P. (1996). Estimating security price derivatives using simulation. *Management Science*, 42(2), 269-285.
3. Glasserman, P. (2004). *Monte Carlo Methods in Financial Engineering*. New York: Springer-Verlag.

Appendix A Full Python Code and Results

November 29, 2025

0.1 Setup: Imports and Random Seed Initialization

This cell imports all necessary libraries for the notebook and sets the random seed to ensure **reproducibility** of Monte Carlo simulations.

[1]:
```
# Standard imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Reproducibility
np.random.seed(42)
sns.set_style("whitegrid")
```

0.2 Simulation Parameters and Variable Definitions

This cell defines all key variables used throughout the notebook, including option parameters (S, X, T, vol, r, type), Monte Carlo settings (N, M, Z), and other constants required for pricing and analysis.

[2]:
```
# Option parameters
S = 101.15       # Underlying asset price
X = 98.01        # Strike price
vol = 0.0991     # Annualized volatility
r = 0.015        # Risk-free rate
T = 60/365       # Time to maturity (years)
type = "C"       # Option Type

# Monte Carlo parameters
N = 20           # Time steps
M = 10000         # Simulation paths

# Standard normal random numbers
Z = np.random.normal(size=(N, M))
```

0.3 Implemntations of Options Pricing and Greeks Calculations

0.3.1 Black-Scholes

Black-Scholes Formula Implementation

[3]:
```
from scipy.stats import norm

def blackScholes(r: float, S: float, X: float, T: float, sigma: float,␣
 ↪option_type: str) -> float:
    """
    Calculate the Black-Scholes price of a European call or put option.

    Parameters:
        r (float): Risk-free interest rate.
        S (float): Current underlying stock price.
        X (float): Strike price.
        T (float): Time to expiration in years.
        sigma (float): Volatility of the underlying asset.
        option_type (str): 'C' for Call option, 'P' for Put option.

    Returns:
        float: Theoretical option price.
    """
    d1 = (np.log(S / X) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)

    if option_type == "C":
        price = S * norm.cdf(d1) - X * np.exp(-r * T) * norm.cdf(d2)
    elif option_type == "P":
        price = X * np.exp(-r * T) * norm.cdf(-d2) - S * norm.cdf(-d1)
    else:
        raise ValueError("option_type must be 'C' for Call or 'P' for Put")
    return price
```

Implementing Greeks from Black-Scholes Formula

[4]:
```
def bs_delta(r: float, S: float, X: float, T: float, sigma: float, option_type:␣
 ↪str) -> float:

    d1 = (np.log(S / X) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))

    if option_type == "C":
        delta = norm.cdf(d1)
    elif option_type == "P":
        delta = norm.cdf(d1) - 1
    else:
        raise ValueError("option_type must be 'C' for Call or 'P' for Put")
    return delta
```

```python
def bs_gamma(r: float, S: float, X: float, T: float, sigma: float) -> float:

    d1 = (np.log(S / X) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))
    gamma = norm.pdf(d1) / (S * sigma * np.sqrt(T))
    return gamma


def bs_vega(r: float, S: float, X: float, T: float, sigma: float) -> float:

    d1 = (np.log(S / X) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))
    vega = S * norm.pdf(d1) * np.sqrt(T)
    return vega / 100  # Scaled to 1% change in volatility


def bs_theta(r: float, S: float, X: float, T: float, sigma: float, option_type:␣
 ↪str) -> float:

    d1 = (np.log(S / X) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)

    if option_type == "C":
        theta = (-S * norm.pdf(d1) * sigma / (2 * np.sqrt(T))
                 - r * X * np.exp(-r * T) * norm.cdf(d2))
    elif option_type == "P":
        theta = (-S * norm.pdf(d1) * sigma / (2 * np.sqrt(T))
                 + r * X * np.exp(-r * T) * norm.cdf(-d2))
    else:
        raise ValueError("option_type must be 'C' for Call or 'P' for Put")

    return theta / 365  # Per day decay


def bs_rho(r: float, S: float, X: float, T: float, sigma: float, option_type:␣
 ↪str) -> float:

    d1 = (np.log(S / X) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)

    if option_type == "C":
        rho = X * T * np.exp(-r * T) * norm.cdf(d2)
    elif option_type == "P":
        rho = -X * T * np.exp(-r * T) * norm.cdf(-d2)
    else:
        raise ValueError("option_type must be 'C' for Call or 'P' for Put")

    return rho / 100  # Scaled to 1% change in interest rates
```

Helper Function: Calculating all Greeks from Black-Scholes Formumla

[5]:
```
def compute_greeks_bs(r: float, S: float, X: float, T: float, sigma: float,␣
 ↪option_type: str) -> tuple[float, float, float, float, float]:

    delta = bs_delta(r, S, X, T, sigma, option_type)
    gamma = bs_gamma(r, S, X, T, sigma)
    vega = bs_vega(r, S, X, T, sigma)
    theta = bs_theta(r, S, X, T, sigma, option_type)
    rho = bs_rho(r, S, X, T, sigma, option_type)
    return delta, gamma, vega, theta, rho
```

0.3.2 Monte Carlo

Baseline Implementation of Monte Carlo

[6]:
```
import time

def mc_baseline(
    S: float, X: float, vol: float, r: float, N: int, M: int, Z: np.ndarray, T:␣
 ↪float, type: str, return_payoffs: bool = False
) -> tuple[float, float, float] | np.ndarray:
    """
    Baseline vectorized Monte Carlo implementation for European option pricing.

    Args:
        S (float): Underlying asset price.
        X (float): Option strike price.
        vol (float): Annualized volatility.
        r (float): Risk-free interest rate.
        N (int): Number of time discretization steps.
        M (int): Number of Monte Carlo paths.
        Z (np.ndarray): Standard normal draws of shape (N, M).
        T (float): Time to maturity (years).
        type (str): Option type, "C" for Call or "P" for Put.

    Returns:
        tuple: (option price (float), standard error (float), computation time␣
 ↪(float, seconds))
    """
    start_time = time.time()
    # Baseline vectorized Monte Carlo implementation (efficient European option␣
 ↪pricing).
    # Precompute constants
    dt = T/N
    # Time step length (years)
    drift_dt = (r - vol**2/2)*dt
    # Drift component (risk-neutral measure)
    volsdt = vol*np.sqrt(dt)
```

```
    # Diffusion component
    lnS = np.log(S)
    # Initial log-price
    # Log-price increments
    delta_lnst = drift_dt + volsdt*Z
    # Cumulative log-price paths
    lnSt = lnS + np.cumsum(delta_lnst, axis=0)
    # Include initial log-price as row 0
    lnSt = np.concatenate((np.full(shape=(1, M), fill_value=lnS), lnSt))
    # Convert log-prices to asset prices
    ST = np.exp(lnSt)
    # Option payoff at maturity
    if type == "C":
        discounted_payoff = np.exp(-r*T) * np.maximum(0, ST[-1] - X)
    elif type == "P":
        discounted_payoff = np.exp(-r*T) * np.maximum(0, X - ST[-1])
    else:
        raise ValueError(f"Invalid option type '{type}'. Must be 'C' for Call␣
 ↪or 'P' for Put.")
    # Estimate option value and standard error
    # Monte Carlo price (mean discounted payoff)
    C0 = np.sum(discounted_payoff) / M
    # Standard deviation of discounted payoffs
    sigma = np.sqrt(np.sum((discounted_payoff - C0)**2) / (M - 1))
    # Standard error of estimate
    SE = sigma/np.sqrt(M)
    computation_time = time.time() - start_time
    if return_payoffs:
        return discounted_payoff
    return C0, SE, computation_time
```

Monte Carlo with Antithetic Variates

[7]:
```
def mc_antithetic(
    S: float, X: float, vol: float, r: float, N: int, M: int, Z: np.ndarray, T:␣
 ↪float, type: str, return_payoffs: bool = False
) -> tuple[float, float, float] | np.ndarray:

    start_time = time.time()
    # Precompute constants
    dt = T/N
    drift_dt = (r - vol**2/2)*dt
    volsdt = vol*np.sqrt(dt)
    lnS = np.log(S)
    # Log-price increments (positive and negative shocks)
    delta_lnSt1 = drift_dt + volsdt*Z
    delta_lnSt2 = drift_dt - volsdt*Z
    lnSt1 = lnS + np.cumsum(delta_lnSt1, axis=0)
```

```
    lnSt2 = lnS + np.cumsum(delta_lnSt2, axis=0)
    # Asset price paths under positive/negative shocks
    ST1 = np.exp(lnSt1)
    ST2 = np.exp(lnSt2)
    # Option payoff at maturity
    if type == "C":
        discounted_payoff = np.exp(-r*T) * (np.maximum(0, ST1[-1] - X) + np.
 ↪maximum(0, ST2[-1] - X))/2
    elif type == "P":
        discounted_payoff = np.exp(-r*T) * (np.maximum(0, X - ST1[-1]) + np.
 ↪maximum(0, X - ST2[-1]))/2
    else:
        raise ValueError(f"Invalid option type '{type}'. Must be 'C' for Call␣
 ↪or 'P' for Put.")
    # Estimate option value and standard error
    C0 = np.sum(discounted_payoff) / M
    sigma = np.sqrt(np.sum((discounted_payoff - C0)**2) / (M - 1))
    SE = sigma/np.sqrt(M)
    computation_time = time.time() - start_time
    if return_payoffs:
        return discounted_payoff
    return C0, SE, computation_time
```

Monte Carlo with Delta Control Variates

[8]:
```
def mc_delta_control(
    S: float, X: float, vol: float, r: float, N: int, M: int, Z: np.ndarray, T:␣
 ↪float, type: str, return_payoffs: bool = False
) -> tuple[float, float, float] | np.ndarray:

    start_time = time.time()
    # Precompute constants
    dt = T / N
    # Time step length (years)
    drift_dt = (r - vol**2 / 2) * dt
    # Drift component (risk-neutral measure)
    volsdt = vol * np.sqrt(dt)
    # Diffusion component
    erdt = np.exp(r * dt)
    # Forward factor
    beta1 = -1  # Control variate coefficient

    # Log-price increments
    delta_St = drift_dt + volsdt * Z
    # Cumulative asset price paths
    ST = S * np.cumprod(np.exp(delta_St), axis=0)
    ST = np.concatenate((np.full(shape=(1, M), fill_value=S), ST))
    # Delta at each time step (excluding t=0)
```

```
    deltaST = bs_delta(r, ST[:-1].T, X, np.linspace(T, dt, N), vol, type).T
    # Cumulative Delta-based control variate adjustments
    cv = np.cumsum(deltaST * (ST[1:] - ST[:-1] * erdt), axis=0)
    # Option payoff adjusted with Delta control variate
    if type == "C":
        discounted_payoff = np.exp(-r * T) * (np.maximum(0, ST[-1] - X) + beta1␣
↪* cv[-1])
    elif type == "P":
        discounted_payoff = np.exp(-r * T) * (np.maximum(0, X - ST[-1]) + beta1␣
↪* cv[-1])
    else:
        raise ValueError(f"Invalid option type '{type}'. Must be 'C' for Call␣
↪or 'P' for Put.")
    # Estimate option value and standard error
    # Monte Carlo price (mean discounted payoff)
    C0 = np.sum(discounted_payoff) / M
    # Standard deviation of discounted payoffs
    sigma = np.sqrt(np.sum((discounted_payoff - C0) ** 2) / (M - 1))
    # Standard error of estimate
    SE = sigma / np.sqrt(M)
    computation_time = time.time() - start_time
    if return_payoffs:
        return discounted_payoff
    return C0, SE, computation_time
```

Monte Carlo with Gamma Control Variates

[9]:
```
def mc_gamma_control(
    S: float, X: float, vol: float, r: float, N: int, M: int, Z: np.ndarray, T:␣
↪float, type: str, return_payoffs: bool = False
) -> tuple[float, float, float] | np.ndarray:

    start_time = time.time()
    # Precompute constants
    dt = T / N
    # Time step length (years)
    drift_dt = (r - vol**2 / 2) * dt
    # Drift component (risk-neutral measure)
    volsdt = vol * np.sqrt(dt)
    # Diffusion component
    erdt = np.exp(r * dt)
    # Forward factor
    ergamma = np.exp((2 * r + vol**2) * dt) - 2 * erdt + 1
    beta2 = -0.5  # Control variate coefficient

    # Log-price increments
    delta_St = drift_dt + volsdt * Z
    # Cumulative asset price paths
```

```
    ST = S * np.cumprod(np.exp(delta_St), axis=0)
    ST = np.concatenate((np.full(shape=(1, M), fill_value=S), ST))
    # Gamma at each time step
    gammaST = bs_gamma(r, ST[:-1].T, X, np.linspace(T, dt, N), vol).T
    # Cumulative Gamma-based control variate adjustments
    cv2 = np.cumsum(gammaST * ((ST[1:] - ST[:-1]) ** 2 - ergamma * ST[:-1] **␣
↪2), axis=0)
    # Option payoff adjusted with Gamma control variate
    if type == "C":
        discounted_payoff = np.exp(-r * T) * (np.maximum(0, ST[-1] - X) + beta2␣
↪* cv2[-1])
    elif type == "P":
        discounted_payoff = np.exp(-r * T) * (np.maximum(0, X - ST[-1]) + beta2␣
↪* cv2[-1])
    else:
        raise ValueError(f"Invalid option type '{type}'. Must be 'C' for Call␣
↪or 'P' for Put.")
    # Estimate option value and standard error
    # Monte Carlo price (mean discounted payoff)
    C0 = np.sum(discounted_payoff) / M
    # Standard deviation of discounted payoffs
    sigma = np.sqrt(np.sum((discounted_payoff - C0) ** 2) / (M - 1))
    # Standard error of estimate
    SE = sigma / np.sqrt(M)
    computation_time = time.time() - start_time
    if return_payoffs:
        return discounted_payoff
    return C0, SE, computation_time
```

Monte Carlo with Delta Control and Antithetic Variates

[10]:
```
def mc_antithetic_delta(
    S: float, X: float, vol: float, r: float, N: int, M: int, Z: np.ndarray, T:␣
 ↪float, type: str, return_payoffs: bool = False
) -> tuple[float, float, float] | np.ndarray:

    start_time = time.time()
    # Precompute constants
    dt = T / N
    drift_dt = (r - vol**2 / 2) * dt
    volsdt = vol * np.sqrt(dt)
    erdt = np.exp(r * dt)
    beta1 = -1  # Control variate coefficient

    # Log-price increments for antithetic pairs
    delta_St1 = drift_dt + volsdt * Z
    delta_St2 = drift_dt - volsdt * Z
```

```
    # Cumulative asset price paths for each antithetic pair
    ST1 = S * np.cumprod(np.exp(delta_St1), axis=0)
    ST2 = S * np.cumprod(np.exp(delta_St2), axis=0)
    ST1 = np.concatenate((np.full(shape=(1, M), fill_value=S), ST1))
    ST2 = np.concatenate((np.full(shape=(1, M), fill_value=S), ST2))
    # Delta at each time step for both paths
    deltaST1 = bs_delta(r, ST1[:-1].T, X, np.linspace(T, dt, N), vol, type).T
    deltaST2 = bs_delta(r, ST2[:-1].T, X, np.linspace(T, dt, N), vol, type).T
    # Cumulative Delta-based control variate adjustments
    cv1 = np.cumsum(deltaST1 * (ST1[1:] - ST1[:-1] * erdt), axis=0)
    cv2 = np.cumsum(deltaST2 * (ST2[1:] - ST2[:-1] * erdt), axis=0)
    # Option payoff adjusted with Delta control variate, averaged over␣
 ↪antithetic pairs
    if type == "C":
        discounted_payoff = 0.5 * np.exp(-r * T) * (
            np.maximum(0, ST1[-1] - X) + beta1 * cv1[-1] +
            np.maximum(0, ST2[-1] - X) + beta1 * cv2[-1]
        )
    elif type == "P":
        discounted_payoff = 0.5 * np.exp(-r * T) * (
            np.maximum(0, X - ST1[-1]) + beta1 * cv1[-1] +
            np.maximum(0, X - ST2[-1]) + beta1 * cv2[-1]
        )
    else:
        raise ValueError(f"Invalid option type '{type}'. Must be 'C' for Call␣
 ↪or 'P' for Put.")
    # Estimate option value and standard error
    # Monte Carlo price (mean discounted payoff)
    C0 = np.sum(discounted_payoff) / M
    # Standard deviation of discounted payoffs
    sigma = np.sqrt(np.sum((discounted_payoff - C0) ** 2) / (M - 1))
    # Standard error of estimate
    SE = sigma / np.sqrt(M)
    computation_time = time.time() - start_time
    if return_payoffs:
        return discounted_payoff
    return C0, SE, computation_time
```

Monte Carlo with Delta Control, Gamma Control and Antithetic Variates

[11]:
```
def mc_antithetic_delta_gamma(
    S: float, X: float, vol: float, r: float, N: int, M: int, Z: np.ndarray, T:␣
 ↪float, type: str, return_payoffs: bool = False
) -> tuple[float, float, float] | np.ndarray:

    start_time = time.time()
    # Precompute constants
```

```
    dt = T / N
    drift_dt = (r - vol**2 / 2) * dt
    volsdt = vol * np.sqrt(dt)
    erdt = np.exp(r * dt)
    ergamma = np.exp((2 * r + vol**2) * dt) - 2 * erdt + 1
    beta1 = -1    # Delta control variate coefficient
    beta2 = -0.5  # Gamma control variate coefficient

    # Log-price increments for antithetic pairs
    delta_St1 = drift_dt + volsdt * Z
    delta_St2 = drift_dt - volsdt * Z
    # Cumulative asset price paths
    ST1 = S * np.cumprod(np.exp(delta_St1), axis=0)
    ST2 = S * np.cumprod(np.exp(delta_St2), axis=0)
    ST1 = np.concatenate((np.full(shape=(1, M), fill_value=S), ST1))
    ST2 = np.concatenate((np.full(shape=(1, M), fill_value=S), ST2))
    # Delta and Gamma at each time step for both paths
    deltaST1 = bs_delta(r, ST1[:-1].T, X, np.linspace(T, dt, N), vol, type).T
    deltaST2 = bs_delta(r, ST2[:-1].T, X, np.linspace(T, dt, N), vol, type).T
    gammaST1 = bs_gamma(r, ST1[:-1].T, X, np.linspace(T, dt, N), vol).T
    gammaST2 = bs_gamma(r, ST2[:-1].T, X, np.linspace(T, dt, N), vol).T
    # Cumulative Delta and Gamma-based control variate adjustments
    cv1d = np.cumsum(deltaST1 * (ST1[1:] - ST1[:-1] * erdt), axis=0)
    cv2d = np.cumsum(deltaST2 * (ST2[1:] - ST2[:-1] * erdt), axis=0)
    cv1g = np.cumsum(gammaST1 * ((ST1[1:] - ST1[:-1]) ** 2 - ergamma * ST1[:-1]␣
↪** 2), axis=0)
    cv2g = np.cumsum(gammaST2 * ((ST2[1:] - ST2[:-1]) ** 2 - ergamma * ST2[:-1]␣
↪** 2), axis=0)
    # Option payoff adjusted with Delta and Gamma control variates, averaged␣
↪over antithetic pairs
    if type == "C":
        discounted_payoff = 0.5 * np.exp(-r * T) * (
            np.maximum(0, ST1[-1] - X) + beta1 * cv1d[-1] + beta2 * cv1g[-1] +
            np.maximum(0, ST2[-1] - X) + beta1 * cv2d[-1] + beta2 * cv2g[-1]
        )
    elif type == "P":
        discounted_payoff = 0.5 * np.exp(-r * T) * (
            np.maximum(0, X - ST1[-1]) + beta1 * cv1d[-1] + beta2 * cv1g[-1] +
            np.maximum(0, X - ST2[-1]) + beta1 * cv2d[-1] + beta2 * cv2g[-1]
        )
    else:
        raise ValueError(f"Invalid option type '{type}'. Must be 'C' for Call␣
↪or 'P' for Put.")
    # Estimate option value and standard error
    # Monte Carlo price (mean discounted payoff)
    C0 = np.sum(discounted_payoff) / M
    # Standard deviation of discounted payoffs
```

```
    sigma = np.sqrt(np.sum((discounted_payoff - C0) ** 2) / (M - 1))
    # Standard error of estimate
    SE = sigma / np.sqrt(M)
    computation_time = time.time() - start_time
    if return_payoffs:
        return discounted_payoff
    return C0, SE, computation_time
```

Implementing Greeks Calculations from Monte Carlo Simulations

[12]:
```
def mc_delta(
    S: float, X: float, vol: float, r: float, N: int, M: int, Z: np.ndarray, T:␣
 ↪float, type: str) -> tuple[float, float]:

    # Precompute constants
    dt = T / N
    # Time step length (years)
    drift_dt = (r - vol**2 / 2) * dt
    # Drift component (risk-neutral measure)
    volsdt = vol * np.sqrt(dt)
    # Diffusion component
    erdt = np.exp(r * dt)
    ergamma = np.exp((2 * r + vol**2) * dt) - 2 * erdt + 1
    # Antithetic log-price increments
    delta_St1 = drift_dt + volsdt * Z
    delta_St2 = drift_dt - volsdt * Z
    # Asset price paths
    ST1 = S * np.cumprod(np.exp(delta_St1), axis=0)
    ST2 = S * np.cumprod(np.exp(delta_St2), axis=0)
    ST1 = np.concatenate((np.full((1, M), S), ST1))
    ST2 = np.concatenate((np.full((1, M), S), ST2))
    # Pathwise Delta estimate at t=0
    if type == "C":
        delta_paths = 0.5 * ((ST1[-1] > X).astype(float) * ST1[-1] / S +
                             (ST2[-1] > X).astype(float) * ST2[-1] / S)
    elif type == "P":
        delta_paths = 0.5 * (-(ST1[-1] < X).astype(float) * ST1[-1] / S -
                             (ST2[-1] < X).astype(float) * ST2[-1] / S)
    else:
        raise ValueError("type must be 'C' or 'P'")
    # Discounted back to present
    delta_paths *= np.exp(-r * T)
    delta_exp = np.mean(delta_paths)
    SE = np.std(delta_paths, ddof=1) / np.sqrt(M)
    return float(delta_exp), float(SE)

def mc_gamma(
```

```
    S: float, X: float, vol: float, r: float, N: int, M: int, Z: np.ndarray, T:␣
↪float, type: str, h: float = 0.5) -> tuple[float, float]:

    # Price with upward spot bump
    C1, SE1, comp1 = mc_antithetic_delta_gamma(S + h, X, vol, r, N, M, Z, T,␣
↪type, return_payoffs=False)
    # Price at baseline spot
    C2, SE2, comp2 = mc_antithetic_delta_gamma(S, X, vol, r, N, M, Z, T, type,␣
↪return_payoffs=False)
    # Price with downward spot bump
    C3, SE3, comp3 = mc_antithetic_delta_gamma(S - h, X, vol, r, N, M, Z, T,␣
↪type, return_payoffs=False)
    # Central difference approximation
    gamma_exp = (C1 - 2 * C2 + C3) / (h ** 2)
    # Standard error propagation
    SE = np.sqrt(SE1 ** 2 + 4 * SE2 ** 2 + SE3 ** 2) / (h ** 2)
    return float(gamma_exp), float(SE)

def mc_vega(
    S: float, X: float, vol: float, r: float, N: int, M: int, Z: np.ndarray, T:␣
↪float, type: str, h: float = 0.01) -> tuple[float, float]:

    # Price with upward volatility bump
    C1, SE1, comp1 = mc_antithetic_delta_gamma(S, X, vol + h, r, N, M, Z, T,␣
↪type, return_payoffs=False)
    # Price with downward volatility bump
    C2, SE2, comp2 = mc_antithetic_delta_gamma(S, X, vol - h, r, N, M, Z, T,␣
↪type, return_payoffs=False)
    # Central difference approximation
    vega_exp = (C1 - C2) / (2 * h)
    # Standard error propagation
    SE = np.sqrt(SE1 ** 2 + SE2 ** 2) / (2 * h)
    return float(vega_exp / 100), float(SE / 100)

def mc_theta(
    S: float, X: float, vol: float, r: float, N: int, M: int, Z: np.ndarray, T:␣
↪float, type: str, h: float = 1 / 365) -> tuple[float, float]:

    # Price with extended maturity
    C2, SE2, comp1 = mc_antithetic_delta_gamma(S, X, vol, r, N, M, Z, T + h,␣
↪type, return_payoffs=False)
    # Price with reduced maturity
    if T - h <= 0:
        raise ValueError("T - h must be positive for finite difference Theta␣
↪calculation.")
```

```
    C1, SE1, comp2 = mc_antithetic_delta_gamma(S, X, vol, r, N, M, Z, T - h,␣
↪type, return_payoffs=False)
    # Central difference approximation
    theta_exp = (C1 - C2) / (2 * h)
    # Standard error propagation
    SE = np.sqrt(SE1 ** 2 + SE2 ** 2) / (2 * h)
    return float(theta_exp / 365), float(SE / 365)

def mc_rho(
    S: float, X: float, vol: float, r: float, N: int, M: int, Z: np.ndarray, T:␣
↪float, type: str, h: float = 0.01 ) -> tuple[float, float]:

    # Price with upward rate bump
    C1, SE1, comp1 = mc_antithetic_delta_gamma(S, X, vol, r + h, N, M, Z, T,␣
↪type, return_payoffs=False)
    # Price with downward rate bump
    C2, SE2, comp2 = mc_antithetic_delta_gamma(S, X, vol, r - h, N, M, Z, T,␣
↪type, return_payoffs=False)
    # Central difference approximation
    rho_exp = (C1 - C2) / (2 * h)
    # Standard error propagation
    SE = np.sqrt(SE1 ** 2 + SE2 ** 2) / (2 * h)
    return float(rho_exp / 100), float(SE / 100)
```

Helper Function: Calculating all Greeks from Monte Carlo

[13]:
```
def compute_greeks_mc(
    S: float, X: float, vol: float, r: float, N: int, M: int, Z: np.ndarray, T:␣
↪float, type: str) -> tuple[float, float, float, float, float, float, float,␣
↪float, float, float]:

    MCdelta, SEdelta = mc_delta(S, X, vol, r, N, M, Z, T, type)
    MCgamma, SEgamma = mc_gamma(S, X, vol, r, N, M, Z, T, type)
    MCvega, SEvega = mc_vega(S, X, vol, r, N, M, Z, T, type)
    MCtheta, SEtheta = mc_theta(S, X, vol, r, N, M, Z, T, type)
    MCrho, SErho = mc_rho(S, X, vol, r, N, M, Z, T, type)
    return MCdelta, SEdelta, MCgamma, SEgamma, MCvega, SEvega, MCtheta,␣
↪SEtheta, MCrho, SErho
```

Helper Function: Comparing the veriants of Monte Carlo implementations

[14]:
```
def benchmark_mc_variants(
    S: float, X: float, vol: float, r: float, N: int, M: int, Z: np.ndarray, T:␣
↪float, type: str) -> pd.DataFrame:

    C02, SE2, comp2 = mc_baseline(S, X, vol, r, N, M, Z, T, type,␣
↪return_payoffs=False)
```

```
  C03, SE3, comp3 = mc_antithetic(S, X, vol, r, N, M, Z, T, type,␣
↪return_payoffs=False)
  C04, SE4, comp4 = mc_delta_control(S, X, vol, r, N, M, Z, T, type,␣
↪return_payoffs=False)
  C05, SE5, comp5 = mc_gamma_control(S, X, vol, r, N, M, Z, T, type,␣
↪return_payoffs=False)
  C06, SE6, comp6 = mc_antithetic_delta(S, X, vol, r, N, M, Z, T, type,␣
↪return_payoffs=False)
  C07, SE7, comp7 = mc_antithetic_delta_gamma(S, X, vol, r, N, M, Z, T, type,␣
↪return_payoffs=False)
  results = [
      {
          "Function": "Vectorized Baseline",
          "Standard Error": SE2,
          "Computation Time": comp2,
          "Standard Error Reduction Multiple": SE2/SE2,
          "Relative Computation Time": comp2/comp2,
      },
      {
          "Function": "Antithetic Variates",
          "Standard Error": SE3,
          "Computation Time": comp3,
          "Standard Error Reduction Multiple": SE2/SE3,
          "Relative Computation Time": comp3/comp2,
      },
      {
          "Function": "Delta-based Control Variates",
          "Standard Error": SE4,
          "Computation Time": comp4,
          "Standard Error Reduction Multiple": SE2/SE4,
          "Relative Computation Time": comp4/comp2,
      },
      {
          "Function": "Gamma-based Control Variates",
          "Standard Error": SE5,
          "Computation Time": comp5,
          "Standard Error Reduction Multiple": SE2/SE5,
          "Relative Computation Time": comp5/comp2,
      },
      {
          "Function": "Antithetic AND Delta Variates",
          "Standard Error": SE6,
          "Computation Time": comp6,
          "Standard Error Reduction Multiple": SE2/SE6,
          "Relative Computation Time": comp6/comp2,
      },
      {
```

```
            "Function": "Antithetic, Delta AND Gamma Variates",
            "Standard Error": SE7,
            "Computation Time": comp7,
            "Standard Error Reduction Multiple": SE2/SE7,
            "Relative Computation Time": comp7/comp2,
        }
    ]
    pd.set_option('display.max_colwidth', None)
    columns = [
        "Function",
        "Standard Error",
        "Computation Time",
        "Standard Error Reduction Multiple",
        "Relative Computation Time",
    ]
    df = pd.DataFrame(results, columns=columns)
    return df
```

0.4 Analysis

0.4.1 Comparing Greeks using Black-Scholes vs Monte Carlo

[15]:
```
def compare_greeks(S: float, X: float, vol: float, r: float, N: int, M: int, Z:␣
↪np.ndarray, T: float, type: str) -> pd.DataFrame:
    """
    Compare option Greeks computed using Black-Scholes formula and Monte Carlo␣
↪simulation.

    Parameters:
        S (float): Underlying asset price
        X (float): Strike price
        vol (float): Volatility
        r (float): Risk-free rate
        N (int): Number of time steps
        M (int): Number of simulations
        Z (np.ndarray): Matrix of random normal variates
        T (float): Time to expiry in years
        type (str): Option type ('C' for Call, 'P' for Put)

    Returns:
        pd.DataFrame: DataFrame summarizing Greeks from both methods and Monte␣
↪Carlo standard errors
    """
    # Calculate Greeks using Black-Scholes formula
    BSdelta, BSgamma, BSvega, BStheta, BSrho = compute_greeks_bs(r, S, X, T,␣
↪vol, type)
```

```
    # Calculate Greeks and standard errors using Monte Carlo simulation
    MCdelta, SEdelta, MCgamma, SEgamma, MCvega, SEvega, MCtheta, SEtheta,␣
↪MCrho, SErho = compute_greeks_mc(
        S, X, vol, r, N, M, Z, T, type
    )

    results = [
        {
            "Greek": "Delta",
            "Black-Scholes": BSdelta,
            "Monte Carlo": MCdelta,
            "MC SE": SEdelta,
        },
        {
            "Greek": "Gamma",
            "Black-Scholes": BSgamma,
            "Monte Carlo": MCgamma,
            "MC SE": SEgamma,
        },
        {
            "Greek": "Vega",
            "Black-Scholes": BSvega,
            "Monte Carlo": MCvega,
            "MC SE": SEvega,
        },
        {
            "Greek": "Theta",
            "Black-Scholes": BStheta,
            "Monte Carlo": MCtheta,
            "MC SE": SEtheta,
        },
        {
            "Greek": "Rho",
            "Black-Scholes": BSrho,
            "Monte Carlo": MCrho,
            "MC SE": SErho,
        },
    ]

    columns = ["Greek", "Black-Scholes", "Monte Carlo", "MC SE"]
    df = pd.DataFrame(results, columns=columns).round(3)
    pd.set_option('display.max_colwidth', None)
    return df
```

[16]:
```
df_greeks = compare_greeks(S, X, vol, r, N, M, Z, T, type)
df_greeks
```

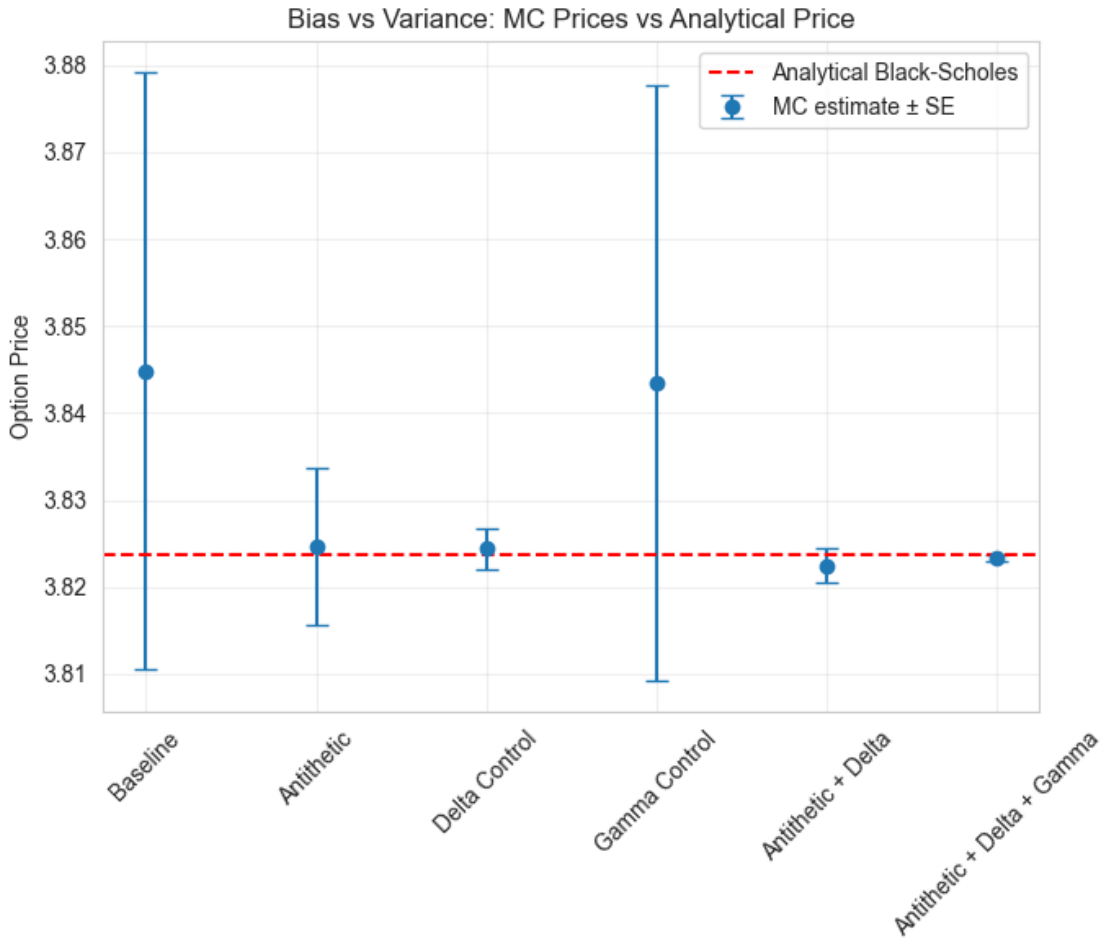
[16]:

	Greek	Black-Scholes	Monte Carlo	MC SE
0	Delta	0.807	0.804	0.002
1	Gamma	0.067	0.067	0.004
2	Vega	0.112	0.112	0.000
3	Theta	-0.012	-0.012	0.000
4	Rho	0.128	0.128	0.000

0.4.2 Bias vs Variance in Monte Carlo Simulation

[17]:
```
# Define MC variants and their corresponding functions
variants = ['Baseline', 'Antithetic', 'Delta Control', 'Gamma Control',␣
 ↪'Antithetic + Delta', 'Antithetic + Delta + Gamma']
mc_funcs = [
    mc_baseline,
    mc_antithetic,
    mc_delta_control,
    mc_gamma_control,
    mc_antithetic_delta,
    mc_antithetic_delta_gamma
]

# Run each MC variant and collect price and SE
mc_prices = []
SEs = []

for func in mc_funcs:
    price, SE, _ = func(S, X, vol, r, N, M, Z, T, type)
    mc_prices.append(price)
    SEs.append(SE)

# Compute analytical Black-Scholes price
bs_price = blackScholes(r, S, X, T, vol, type)

# Create DataFrame for reference
df_bias = pd.DataFrame({
    'Variant': variants,
    'MC Price': mc_prices,
    'SE': SEs,
    'BS Price': [bs_price]*len(variants)
})

df_bias

# Plot Bias vs Variance
plt.figure(figsize=(7,6))
plt.errorbar(
    x=df_bias['Variant'],
```

```
    y=df_bias['MC Price'],
    yerr=df_bias['SE'],
    fmt='o',
    capsize=5,
    label='MC estimate ± SE',
    color='tab:blue'
)
plt.axhline(y=bs_price, color='r', linestyle='--', label='Analytical␣
 ↪Black-Scholes')

plt.ylabel('Option Price')
plt.title('Bias vs Variance: MC Prices vs Analytical Price')
plt.xticks(rotation=45)
plt.grid(alpha=0.3)
plt.legend()
plt.tight_layout()
plt.show()
```



0.4.3 Comparing Monte Carlo Function variants (prices, SE, computation times)

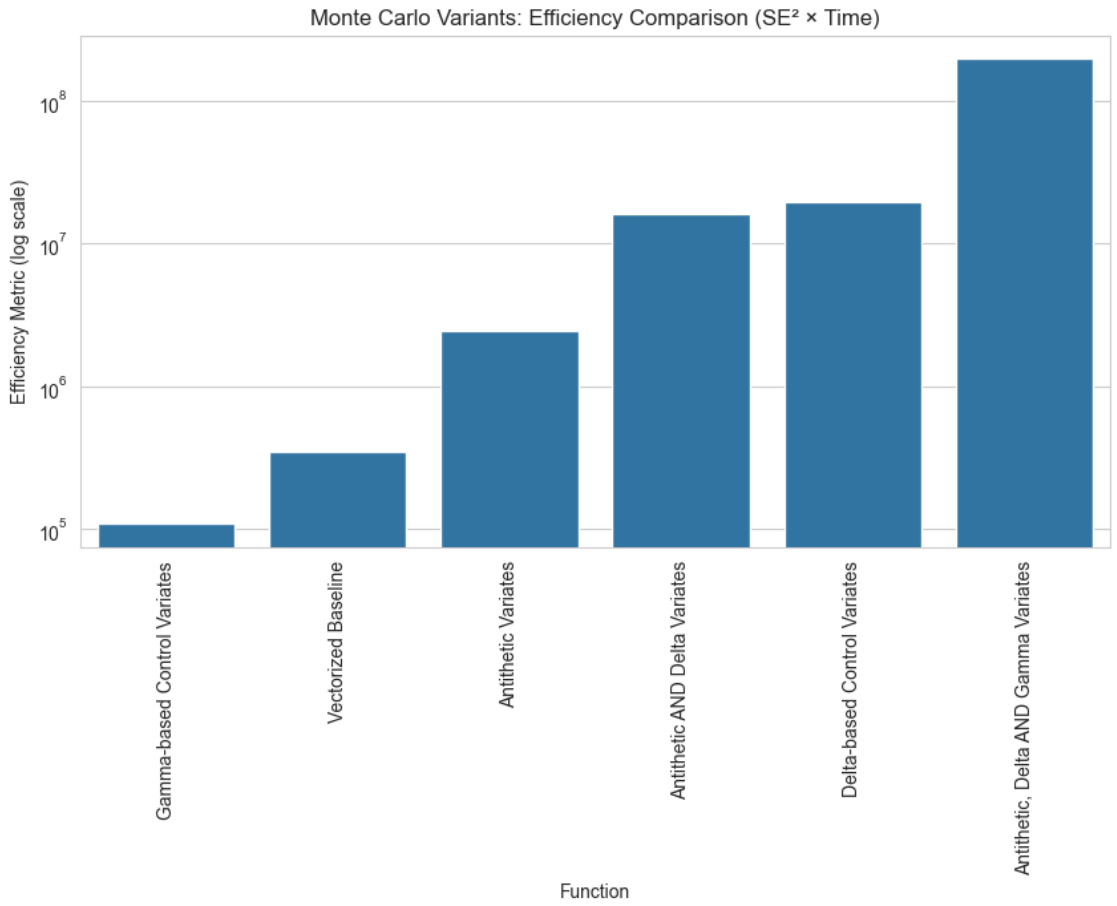
[18]:
```
df_benchmark = benchmark_mc_variants(S, X, vol, r, N, M, Z, T, type)
df_benchmark
```

[18]:

	Function	Standard Error	Computation Time	Standard Error Reduction Multiple	Relative Computation Time
0	Vectorized Baseline	0.034317	0.002454	1.000000	1.000000
1	Antithetic Variates	0.009036	0.005018	3.797611	2.044788
2	Delta-based Control Variates	0.002375	0.008985	14.447800	3.661226
3	Gamma-based Control Variates	0.034198	0.007858	1.003470	3.202176
4	Antithetic AND Delta Variates	0.001898	0.017045	18.081299	6.945691
5	Antithetic, Delta AND Gamma Variates	0.000402	0.031235	85.329272	12.727873

0.4.4 Monte Carlo Efficiency Analysis

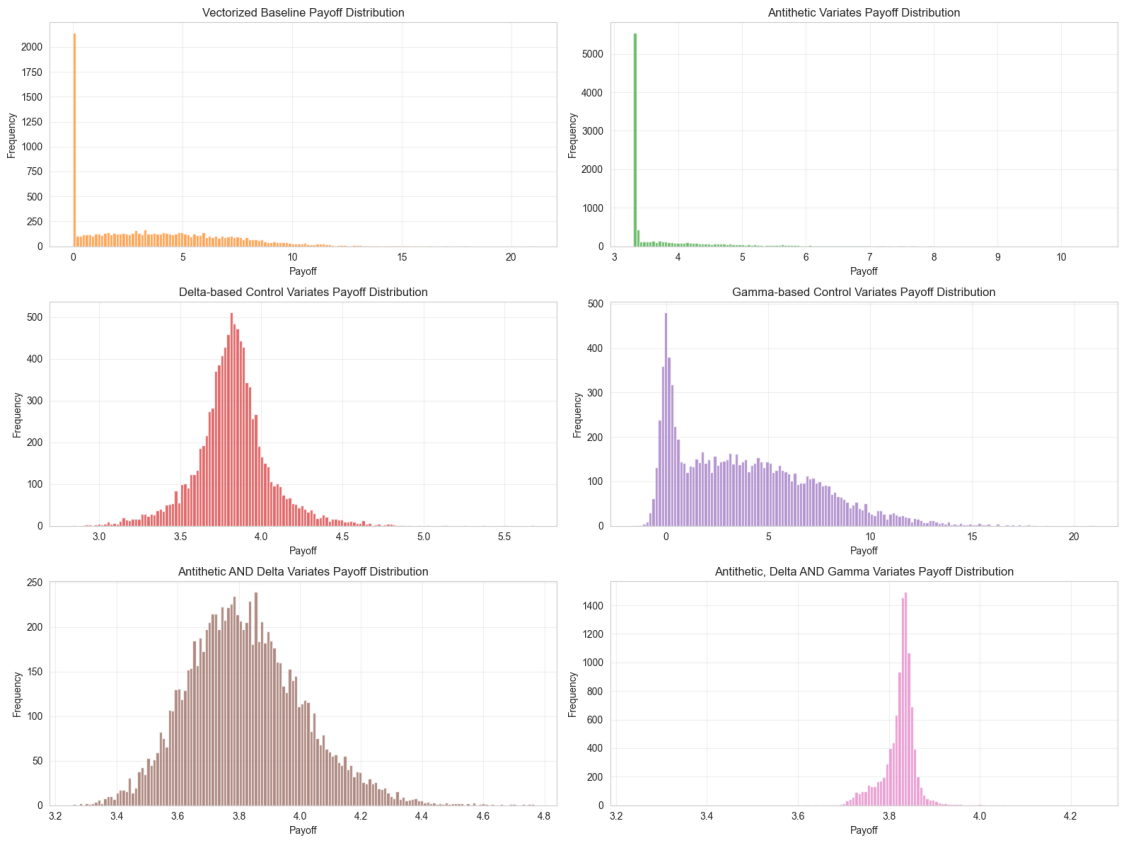
[19]:
```
# Compute efficiency metric: SE^2 * Computation Time
df_benchmark['Efficiency'] = 1 / (df_benchmark['Standard Error']**2 *␣
 ↪df_benchmark['Computation Time'])

# Sort by efficiency (lower is better)
df_benchmark_sorted = df_benchmark.sort_values('Efficiency')
df_benchmark_sorted

plt.figure(figsize=(10,5))
sns.barplot(x='Function', y='Efficiency', data=df_benchmark_sorted)
plt.title("Monte Carlo Variants: Efficiency Comparison (SE² × Time)")
plt.ylabel("Efficiency Metric (log scale)")
plt.yscale('log')  # <-- log scale
plt.xticks(rotation=90)
plt.show()

plt.show()
```



0.4.5 Monte Carlo Payoff Distribution Analysis

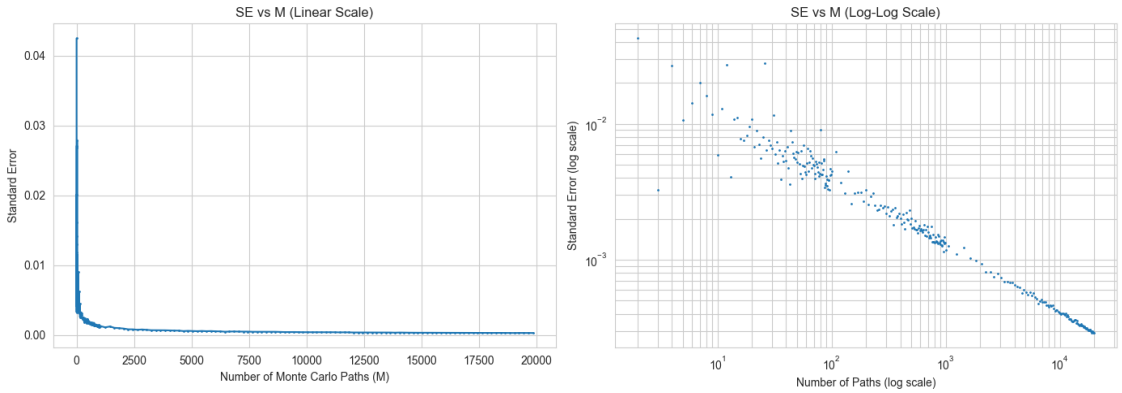
[20]:
```
# ------------------------
# Dictionary of MC variants
# Key = display name, Value = function object
# ------------------------
mc_variants = {
    'Vectorized Baseline': mc_baseline,
    'Antithetic Variates': mc_antithetic,
    'Delta-based Control Variates': mc_delta_control,
    'Gamma-based Control Variates': mc_gamma_control,
    'Antithetic AND Delta Variates': mc_antithetic_delta,
    'Antithetic, Delta AND Gamma Variates': mc_antithetic_delta_gamma
}

# ------------------------
# Plot histograms
# ------------------------
```

```
plt.figure(figsize=(16,12))

for i, (name, func) in enumerate(mc_variants.items(), 1):
    # Generate simulated payoffs
    payoffs = func(S, X, vol, r, N, M, Z, T, type, True)

    # Plot subplot
    plt.subplot(3, 2, i)
    plt.hist(payoffs, bins=150, alpha=0.7, color='C'+str(i))
    plt.title(f'{name} Payoff Distribution')
    plt.xlabel('Payoff')
    plt.ylabel('Frequency')
    plt.grid(alpha=0.3)

plt.tight_layout()
plt.show()
```



0.4.6 Convergence Analysis of Monte Carlo Standard Error

[21]:
```
# Convergence analysis parameters
M_values = list(range(2, 101, 1)) + list(range(110, 1001, 10)) +␣
 ↪list(range(1050, 20001, 200))
SEs = []

for m in M_values:
    Z_m = np.random.normal(size=(N, m))  # generate exactly m paths
    _, SE, _ = mc_antithetic_delta_gamma(S, X, vol, r, N, m, Z_m, T, type)
    SEs.append(SE)

# Create side-by-side plots
fig, axes = plt.subplots(1, 2, figsize=(14,5))

# Linear plot (SE vs M)
axes[0].plot(M_values, SEs, marker='.', markersize=2)
axes[0].set_xlabel("Number of Monte Carlo Paths (M)")
axes[0].set_ylabel("Standard Error")
axes[0].set_title("SE vs M (Linear Scale)")
axes[0].grid(True)

# Log-log plot
axes[1].loglog(M_values, SEs, marker='.', markersize=2, linestyle='None')
axes[1].set_xlabel("Number of Paths (log scale)")
axes[1].set_ylabel("Standard Error (log scale)")
axes[1].set_title("SE vs M (Log-Log Scale)")
axes[1].grid(True, which="both")

plt.tight_layout()
plt.show()

# Calculate slope on log-log scale
logM = np.log(M_values)
logSE = np.log(SEs)
slope, intercept = np.polyfit(logM, logSE, 1)
print(f"Estimated convergence rate (slope)  {slope:.3f}")
```

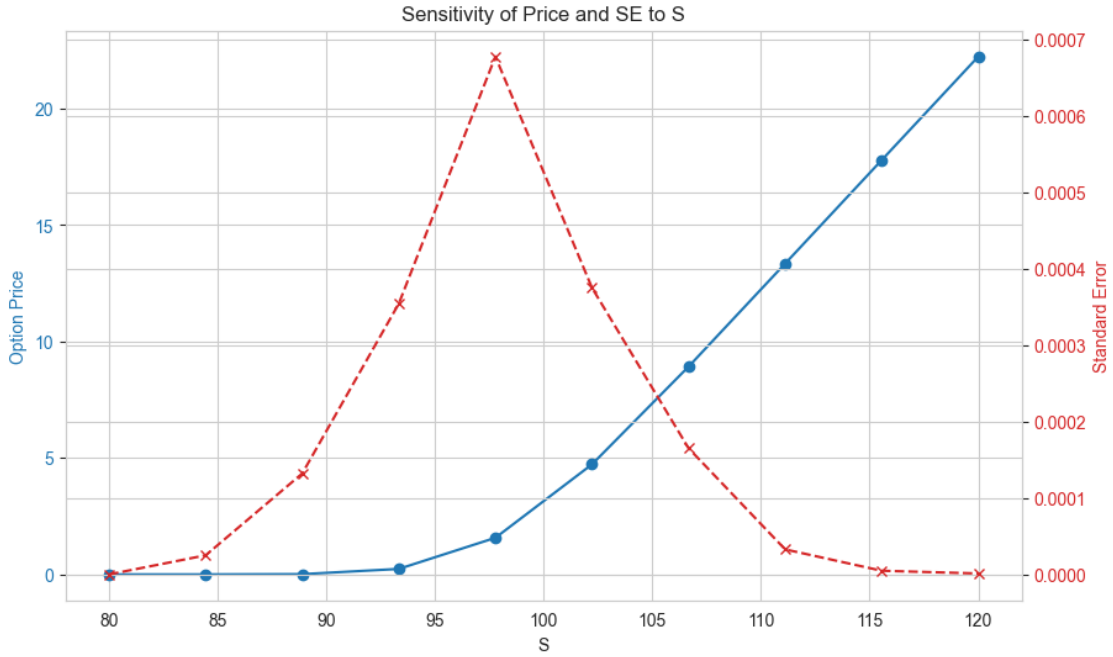



```
Estimated convergence rate (slope)  -0.487
```

0.4.7 Sensitivity Analysis of Option Price and Standard Error

[22]:
```
# %%
import matplotlib.pyplot as plt
import numpy as np

# Sensitivity analysis parameters
param_name = 'S'  # choose which parameter to vary: 'S', 'X', 'vol', 'T'
param_values = np.linspace(80, 120, 10)  # example range

prices = []
SEs = []

for val in param_values:
    # Set parameters for this iteration
    S_val = val if param_name == 'S' else S
    X_val = val if param_name == 'X' else X
    vol_val = val if param_name == 'vol' else vol
    T_val = val if param_name == 'T' else T

    # Generate Z
    Z_m = np.random.normal(size=(N, M))  # M = number of paths, N = timesteps

    # Run main MC function
    price, SE, _ = mc_antithetic_delta_gamma(S_val, X_val, vol_val, r, N, M,␣
 ↪Z_m, T_val, type)
    prices.append(price)
    SEs.append(SE)

# Plot results
fig, ax1 = plt.subplots(figsize=(10,6))
```

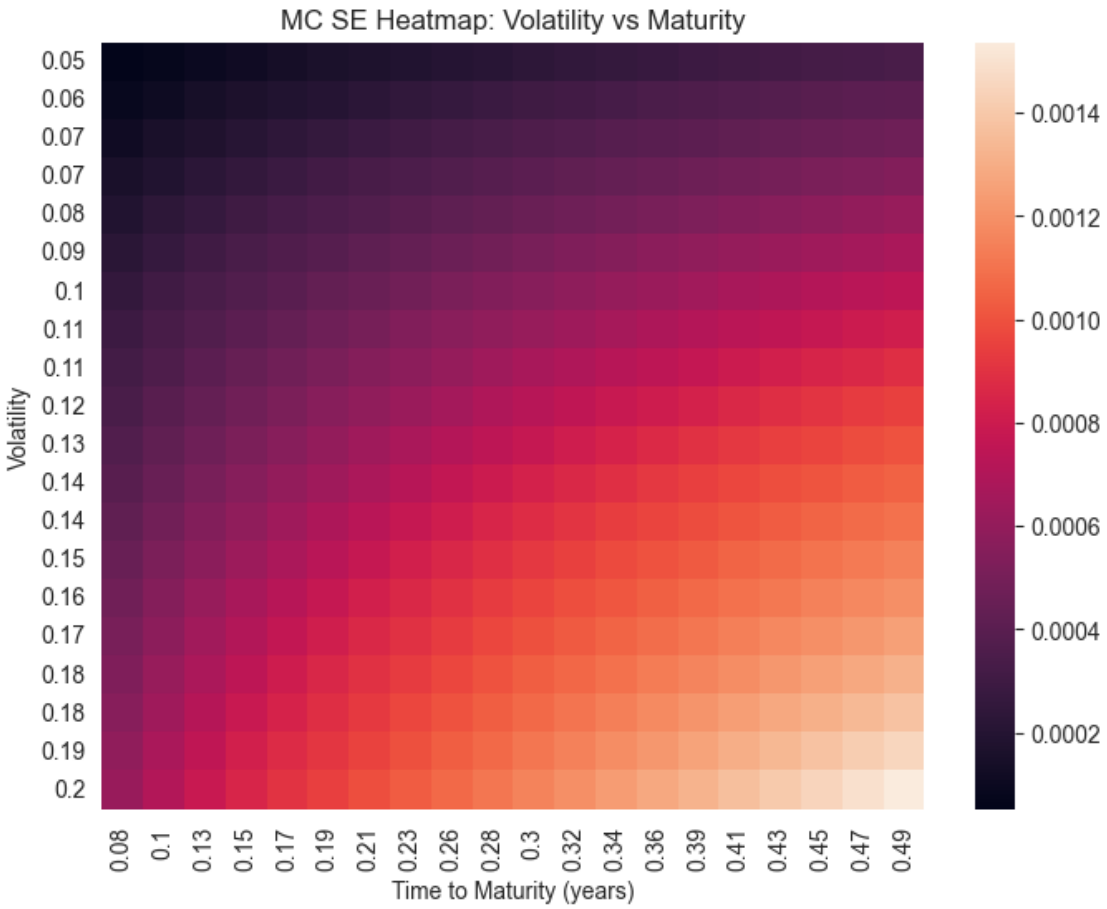
```
color = 'tab:blue'
ax1.set_xlabel(param_name)
ax1.set_ylabel('Option Price', color=color)
ax1.plot(param_values, prices, marker='o', color=color)
ax1.tick_params(axis='y', labelcolor=color)
ax1.grid(True)

ax2 = ax1.twinx()
color = 'tab:red'
ax2.set_ylabel('Standard Error', color=color)
ax2.plot(param_values, SEs, marker='x', linestyle='--', color=color)
ax2.tick_params(axis='y', labelcolor=color)

plt.title(f"Sensitivity of Price and SE to {param_name}")
plt.show()
```



0.4.8 Monte Carlo SE Heatmap: Volatility vs Time to Maturity

[23]:
```
vols = np.linspace(0.05, 0.2, 20)
Ts = np.linspace(30/365, 180/365, 20)
heatmap_data = np.zeros((len(vols), len(Ts)))

for i, vol_i in enumerate(vols):
    for j, T_j in enumerate(Ts):
```

```
        _, SE, _ = mc_antithetic_delta_gamma(S, X, vol_i, r, N, M, Z, T_j, type)
        heatmap_data[i, j] = SE

plt.figure(figsize=(8,6))
sns.heatmap(heatmap_data, xticklabels=np.round(Ts,2), yticklabels=np.
 ↪round(vols,2))
plt.xlabel("Time to Maturity (years)")
plt.ylabel("Volatility")
plt.title("MC SE Heatmap: Volatility vs Maturity")
plt.show()
```



0.4.9 Monte Carlo SE Heatmap: Strike vs Volatility

[24]:
```
strikes = np.linspace(80, 120, 20)  # example range around S
vols = np.linspace(0.05, 0.2, 20)

heatmap_data = np.zeros((len(vols), len(strikes)))
```

```
# Compute SE for each combination
for i, vol_i in enumerate(vols):
    for j, X_j in enumerate(strikes):
        _, SE, _ = mc_antithetic_delta_gamma(S, X_j, vol_i, r, N, M, Z, T, type)
        heatmap_data[i, j] = SE

# Plot heatmap
plt.figure(figsize=(8,6))
sns.heatmap(heatmap_data,
            xticklabels=np.round(strikes,1),
            yticklabels=np.round(vols,2),
            cmap="viridis")
plt.xlabel("Strike Price")
plt.ylabel("Volatility")
plt.title("MC SE Heatmap: Strike vs Volatility")
plt.show()
```