Pricing European Options: A Comparison Between Black-Scholes and Monte Carlo Method

Francesco Distefano, 150964

Python for Finance, 2024-2025

1 Introduction - Definition of Options

Options are derivatives instruments that give the investor the right (but not the obligation) to buy or sell an underlying asset at a fixed price (”strike”) on a predetermined date (”maturity”). This project focuses on **European options**, that can only be exercised at maturity. On the other hand, **American options** can be exercised *until* the maturity date. A **call** option gives the investor the right to **buy** at maturity the underlying asset paying the strike price. A **put** option gives the investor the right to **sell** at maturity the underlying asset obtaining the strike price.

In particular, payoffs for european options at maturity are computed as follows:

$$\textbf{Call:}\max(S_T - K, 0), \quad \textbf{Put:}\max(K - S_T, 0)$$

This work aims to compare two option pricing methods:

- **Black-Scholes Model**
- **Monte Carlo Simulation**

2 Theoretical Framework

2.1 Black-Scholes Model

The Black-Scholes model assumes that the price of the underlying asset follows a Geometric Brownian Motion with constant volatility, acting in a frictionless and arbitrage free market. The following formulas are the closed form solution of the Black-Scholes model in the case (respectively) of European Call and Put options:

$$C = S_0 N(d_1) - K e^{-rT} N(d_2), \text{ and}$$

$$P = K e^{-rT} N(-d_2) - S_0 N(-d_1), \text{ where}$$

$$d_1 = \frac{\ln(S_0/K) + \left(r + \frac{1}{2}\sigma^2\right) T}{\sigma\sqrt{T}}, \quad d_2 = d_1 - \sigma\sqrt{T}$$

S_0: Current price of the underlying asset

K: Strike price of the option

T: Time to maturity (in years)

r: Constant risk-free interest rate

σ: Volatility of the underlying asset

N: Cumulative distribution function of the standard normal distribution

d_1, d_2: Intermediate terms used to compute the probability-weighted payoffs

2.2 Monte Carlo Simulation

The underlying asset price is modeled as a Geometric Brownian motion, and the option price is the discounted average of the simulated payoffs:

$$S_{t+\Delta t} = S_t \cdot \exp\left[\left(r - \frac{1}{2}\sigma^2\right)\Delta t + \sigma\sqrt{\Delta t}Z\right], \quad Z \sim \mathcal{N}(0,1)$$

$$\text{Option Price} = e^{-rT} \cdot \mathbb{E}[\text{Payoff}]$$

S_t: Asset price at time t

r: Constant risk-free interest rate

σ: Volatility of the underlying asset

Δt: Time increment (step size of the discretization)

Z: Standard normal random variable

3 Black-Scholes Pricing

The following Python code shows the creation of a formula to compute price of call and put options according to the Black-Scholes model. The function's inputs are the parameters of the Black-Scholes formula and the option type, in order to return the theoretical price of the option.

```
# BLACK-SCHOLES PRICING FUNCTION
def black_scholes_price(S, K, T, r, sigma, option_type='call'):

# Parameters:
# S: Current asset price
# K: Strike price
# T: Time to maturity (years)
# r: Risk-free annualized interest rate
# sigma: Volatility of the underlying asset

    d1 = (np.log(S / K) + (r + 0.5 * sigma**2) * T) / (sigma * np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T) # d1 & d2 formula

    if option_type == 'call': # B&S formula for call and put options
        price = S * norm.cdf(d1) - K * np.exp(-r * T) * norm.cdf(d2)
    elif option_type == 'put':
        price = K * np.exp(-r * T) * norm.cdf(-d2) - S * norm.cdf(-d1)
    else:
        raise ValueError("option_type must be 'call' or 'put'")
    return price

# Defining parameters
S = 100
K = 100
T = 1.0
r = 0.05
sigma = 0.2

# Calculating prices
call_price_bs = black_scholes_price(S, K, T, r, sigma, option_type='call')
put_price_bs = black_scholes_price(S, K, T, r, sigma, option_type='put')

# Displaying results
```

```python
display(Markdown("**Black-Scholes Pricing**")) # for printing in bold
print(f"Underlying Price (S): {S}") # printing with f-string
print(f"Strike Price (K): {K}")
print(f"Maturity (T): {T} year(s)")
print(f"Risk-Free Rate (r): {r*100:.2f}%") # .2f to choose the num. of decimals
print(f"Volatility (sigma): {sigma*100:.2f}%\n")

print(f"Call Option Price: {call_price_bs:.4f}")
print(f"Put Option Price: {put_price_bs:.4f}\n")
```

For the chosen parameters, we have the following outputs:

```
Black-Scholes Pricing

Underlying Price (S): 100
Strike Price (K): 100
Maturity (T): 1.0 year(s)
Risk-Free Rate (r): 5.00%
Volatility (sigma): 20.00%

Call Option Price: 10.4506
Put Option Price: 5.573
```

4 Monte Carlo Simulation Pricing

The following Python code shows the development of a formula to implement the Monte Carlo simulation method for pricing European options. The function's inputs are the same as the Black-Scholes model, with the addition of the number of simulations and steps (plus an optional random seed). This way we can price call and put options by simulating multiple paths of the underlying asset (which is assumed to evolve like a GBM).

```python
# MONTE CARLO PRICING FUNCTION
def monte_carlo_price(S, K, T, r, sigma, option_type='call', M=10000, N=252):

# Parameters:
# S: Current asset price
# K: Strike price
# T: Time to maturity (years)
# r: Risk-free annualized interest rate
# sigma: Volatility of the underlying asset
# M: Number of simulation
# N: Time steps per simulation

    # Preparing for the simulation:
    dt = T / N # length of every time step
    Z = np.random.standard_normal((M, N)) # mat of std nrm r.v. to simulate GBM
    S_paths = np.zeros((M, N + 1)) # matrix of 0s to store values
    S_paths[:, 0] = S # initial price = S in the first column

    for t in range(1, N + 1): # simulating the price evolution
        S_paths[:, t] = S_paths[:, t - 1] * np.exp(
            (r - 0.5 * sigma**2) * dt + sigma * np.sqrt(dt) * Z[:, t - 1])
            # computing and storing prices with the GBM formula

    S_T = S_paths[:, -1] # extracting the terminal value at T from every simul.
                         # using -1 to go to the last column

    # Computing payoff for call and put options
    if option_type == 'call':
        payoff = np.maximum(S_T - K, 0)
    elif option_type == 'put':
        payoff = np.maximum(K - S_T, 0)
    else:
        raise ValueError("option_type must be 'call' or 'put'")

    price = np.exp(-r * T) * np.mean(payoff) # discount of payoffs mean with r
    return price

# Defining parameters
S = 100
K = 100
T = 1.0
r = 0.05
sigma = 0.2
M = 10000
N = 252

# Calculating prices
call_price_mc = monte_carlo_price(S, K, T, r, sigma, option_type='call', M=M, N=N)
put_price_mc = monte_carlo_price(S, K, T, r, sigma, option_type='put', M=M, N=N)

# Displaying results
display(Markdown("**Monte Carlo Pricing**"))
print(f"Underlying Price (S): {S}")
print(f"Strike Price (K): {K}")
print(f"Maturity (T): {T} year(s)")
print(f"Risk-Free Rate (r): {r*100:.2f}%")
print(f"Volatility (sigma): {sigma*100:.2f}%")
print(f"Simulations (M): {M}")
print(f"Time Steps (N): {N}\n")

print(f"Call Option Price (MC): {call_price_mc:.4f}")
print(f"Put Option Price (MC): {put_price_mc:.4f}\n")
```

With the same parameters as before, using 10000 simulations and 252 steps, we get:

```
Monte Carlo Pricing

Underlying Price (S): 100
Strike Price (K): 100
Maturity (T): 1.0 year(s)
Risk-Free Rate (r): 5.00%
Volatility (sigma): 20.00%
Simulations (M): 10000
Time Steps (N): 252

Call Option Price (MC): 10.2720
Put Option Price (MC): 5.5576
```

5 Comparison between the two methods

5.1 Price Comparison and Errors calculation

With the prices that we previously obtained, we can compute absolute and relative errors with the following code:

```python
# Calculating errors
call_abs_error = abs(call_price_mc - call_price_bs)
call_rel_error = call_abs_error / call_price_bs * 100

put_abs_error = abs(put_price_mc - put_price_bs)
put_rel_error = put_abs_error / put_price_bs * 100
avg_abs_error = (call_abs_error + put_abs_error) / 2

# Displaying results
display(Markdown("**Comparison between Black-Scholes and Monte Carlo**"))
print("Call Option:")
print(f"Black-Scholes Price: {call_price_bs:.4f}")
print(f"Monte Carlo Price: {call_price_mc:.4f}")
print(f"Absolute Error: {call_abs_error:.4f}")
print(f"Relative Error (%): {call_rel_error:.4f}")

print("\nPut Option:")
print(f"Black-Scholes Price: {put_price_bs:.4f}")
print(f"Monte Carlo Price: {put_price_mc:.4f}")
print(f"Absolute Error: {put_abs_error:.4f}")
print(f"Relative Error (%): {put_rel_error:.4f}")

print(f"\nAverage Absolute Error: {avg_abs_error:.4f}")
```

This way, we get the following output:

```
Comparison between Black-Scholes and Monte Carlo

Call Option:
Black-Scholes Price: 10.4506
Monte Carlo Price: 10.2720
Absolute Error: 0.1785
Relative Error (%): 1.7085

Put Option:
Black-Scholes Price: 5.5735
Monte Carlo Price: 5.5576
Absolute Error: 0.0159
Relative Error (%): 0.2857

Average Absolute Error: 0.0972
```

The obtained results show a close alignment between the two approaches. The Black-Scholes model produces a call option price of 10.4506 and a put option price of 5.5735. On the other hand, the Monte Carlo method produces a call option price of 10.2720 and a put option price of 5.5576, using 10,000 simulated paths and 252 steps.

The absolute error between the two methods is 0.1785 for the call option and 0.0159 for the put option. The corresponding relative errors are 1.71% and 0.29%, respectively. These results show good consistency between the two methods, considering the intrinsic variability of the Monte Carlo method.

Also, the average absolute error of the two option types is 0.0972, which suggests that the Monte Carlo method provides an accurate estimate of the theoretical prices of the Black-Scholes model under a sufficient number of simulations.

5.2 Monte Carlo Convergence to the Black-Scholes Model

To assess the reliability of the Monte Carlo method, we analyze how the estimated call option price gets closer to the theoretical Black-Scholes price as the number of simulations increases.

The plot below represents the convergence behavior for a European call option by comparing the Monte Carlo estimates for different numbers of simulations with the fixed reference price given by the Black-Scholes formula.

The plot shows that Monte Carlo prices fluctuate around the theoretical Black-Scholes price when M is low. However, as simulations increase, the estimated price stabilizes and converges the theoretical price.

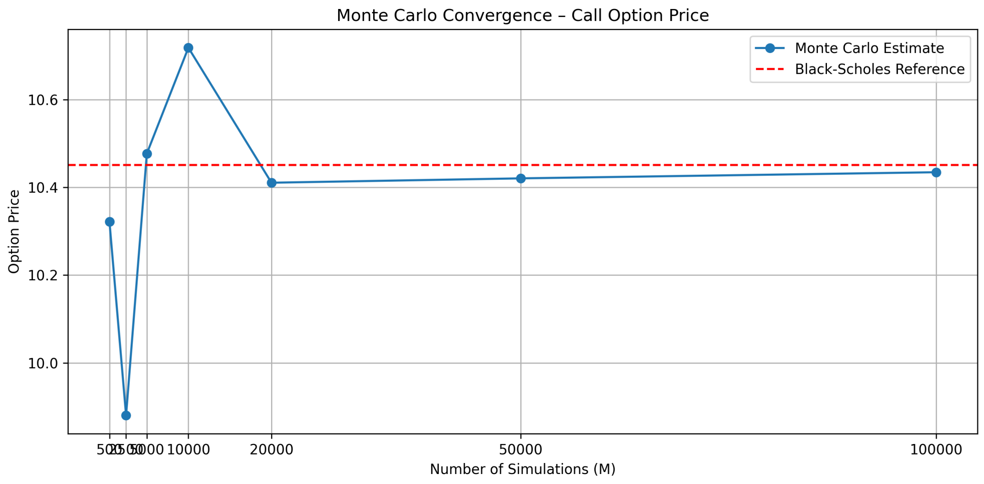

Figure 1: Price Convergence

5.3 Absolute and Relative Error Evolution

The following plot represents the evolution of the absolute error between the Monte Carlo call option price and the price obtained by the Black-Scholes formula as the number of simulations increases.

It is confirmed that the absolute error gets lower when the number of simulations increase. With a low number of simulations, the error is relatively high due to the stochastic variability belonging to the method. As M increases, the error rapidly decreases, showing the convergence of the Monte Carlo method to the Black-Scholes method.

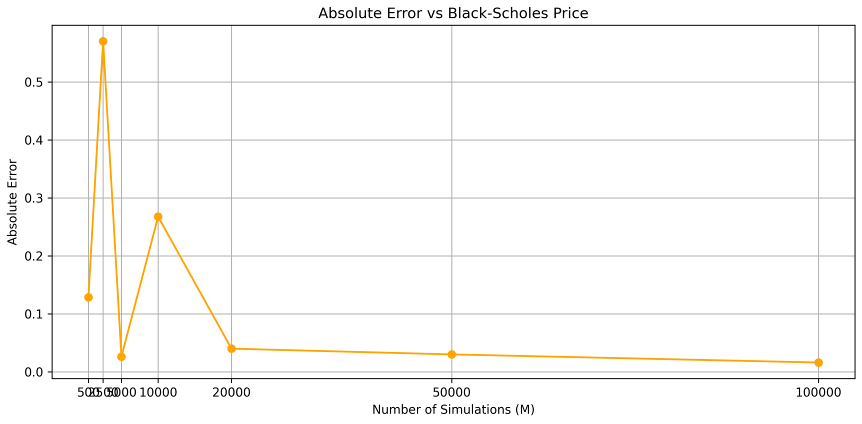

Figure 2: Error Evolution

5.4 Sensitivity Analysis - Volatility case

It is also interesting to see how the two models react when a parameter takes on different values (in this case volatility). The following plot shows a clear upward trend in call option prices as volatility increases, as theoretically expected. Although both models follow similar trajectories, the Monte Carlo estimates are slightly lower than the Black-Scholes values, especially when volatility takes on higher values. This is because of the simulation-based nature of the method, since an increase in the payoff spread introduces additional variance, which could affect the accuracy of the results.

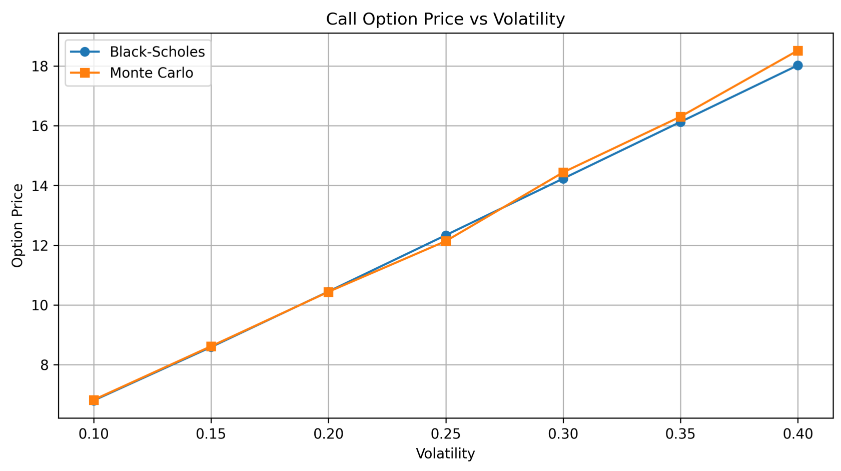

Figure 3: Sensitivity Analysis

6 Conclusion

This project has proved that both the Black-Scholes and Monte Carlo method provide reliable and accurate estimates for pricing European options. While Black-Scholes provides a theoretical closed form solution, Monte Carlo proves to be a flexible alternative that gets closer and closer to the theoretical value as the number of simulations increases, as proved in the errors and convergence analysis. The sensitivity analysis confirms that both models behave consistently for different levels of volatility, with Monte Carlo maintaining good accuracy.

7 Appendix

7.1 Libraries used for the project

```python
import numpy as np
from scipy.stats import norm
from IPython.display import Markdown, display
import matplotlib.pyplot as plt
from google.colab import files
```

7.2 Code for Convergence and Error plotting

```python
# M values list
# We use different numbers of simulation
M_values = [500, 2500, 5000, 10000, 20000, 50000, 100000]
```

```python
call_mc_prices = [] # initial empty list
errors = [] # initial empty list

# Black-Scholes reference price
call_bs_ref = black_scholes_price(S, K, T, r, sigma, option_type='call')

# Monte Carlo Simulation for the M values list
for M in M_values:
    price_mc = monte_carlo_price(S, K, T, r, sigma, option_type='call', M=M, N=N)
    call_mc_prices.append(price_mc) # attaching prices to the previous list
    errors.append(abs(price_mc - call_bs_ref)) # same for absolute error

# Plot of estimated price compared to B&S
plt.figure(figsize=(10, 5))
plt.plot(M_values, call_mc_prices, marker='o', label='Monte Carlo Estimate')
plt.axhline(call_bs_ref, color='red', linestyle='--', label='Black-Scholes
    Reference')
plt.title("Monte Carlo Convergence Call Option Price")
plt.xlabel("Number of Simulations (M)")
plt.ylabel("Option Price")
plt.legend()
plt.grid(True)
plt.xticks(M_values) # Set x-axis ticks
plt.savefig("mc_convergence_price.png", dpi=300)
plt.show()

# Absolute Error plot
plt.figure(figsize=(10, 5))
plt.plot(M_values, errors, marker='o', color='orange')
plt.title("Absolute Error vs Black-Scholes Price")
plt.xlabel("Number of Simulations (M)")
plt.ylabel("Absolute Error")
plt.grid(True)
plt.xticks(M_values)
plt.savefig("mc_absolute_error.png", dpi=300)
plt.show()


# Automatic download
# files.download("mc_convergence_price.png")
# files.download("mc_absolute_error.png")
```

7.3 Code for the Sensitivity Analysis

```python
# Let's test the sensitivity of the two models in the volatility case
# Sigma values' list
sigma_values = [0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4]
bs_prices = [] # initial empty list
mc_prices = [] # initial empty list

# Calculation for every sigma
for sigma_ in sigma_values:
    bs_sa = black_scholes_price(S, K, T, r, sigma_, option_type='call')
    mc_sa = monte_carlo_price(S, K, T, r, sigma_, option_type='call', M=10000, N
    =252)
    bs_prices.append(bs_sa) # attaching B&S prices to the previous list
    mc_prices.append(mc_sa) # same for MC prices
```

```

# Plot Black-Scholes vs Monte Carlo
plt.figure(figsize=(10, 5))
plt.plot(sigma_values, bs_prices, marker='o', label='Black-Scholes')
plt.plot(sigma_values, mc_prices, marker='s', label='Monte Carlo')
plt.title("Call Option Price vs Volatility")
plt.xlabel("Volatility")
plt.ylabel("Option Price")
plt.legend()
plt.grid(True)
plt.savefig("sensitivity_volatility.png", dpi=300)
plt.show()

# Automatic download
# files.download("sensitivity_volatility.png")
```