

# IEOR 221

## Homework 8

### Module 9 Options - Numerical Techniques

Louis Sallé-Tourne

November 1, 2024

## Problem 1

### 1) Monte Carlo Simulation

**Explanation:** In this problem, we use a plain Monte Carlo simulation to estimate the price of a European call option. The stock price at each step is simulated using the following equation for the evolution from  $S_t$  to  $S_{t+1}$ :

$$S_{t+1} = S_t \times \exp \left( (r - q - 0.5\sigma^2)\Delta t + \sigma\epsilon\sqrt{\Delta t} \right)$$

where: -  $r$  is the risk-free rate, -  $q$  is the dividend yield, -  $\sigma$  is the volatility, -  $\Delta t$  is the time step size, -  $\epsilon$  is a standard normal random variable.

We repeat this process over multiple steps up to maturity  $T$  for each path. Once we reach maturity, we calculate the payoff based on the terminal stock price  $S_T$  and average these payoffs over all paths, then discount to the present value.

#### Python Code:

---

```
1 # Definiton of the parameters
2
3 import numpy as np
4
5 S0 = 100.0 # initial stock price
6 K = 100.0 # strike price
```

```

7  T = 1.0 # time to maturity
8  r = 0.06 # risk-free rate
9  q = 0.06 # dividend yield
10 sigma = 0.35 # volatility
11 n_times_steps = 100 # number of time steps
12 n_simulated_stock_paths = 4000 # number of simulated stock
    paths
13 option_type = "call" # "call" or "put"
14 np.random.seed(42) # make result reproducible
15
16 import numpy as np
17 def option_payoff(S, K, option_type):
18     payoff = 0.0
19     if option_type == "call":
20         payoff = max(S - K, 0)
21     elif option_type == "put":
22         payoff = max(K - S, 0)
23     return payoff
24
25 # Monte Carlo simulation
26
27 dt = T/n_times_steps
28 sqrt_dt = np.sqrt(dt)
29 payoff = np.zeros((n_simulated_stock_paths), dtype = float)
30 step = range(0, int(n_times_steps), 1)
31 for i in range(0, n_simulated_stock_paths):
32     ST = S0
33     for j in step:
34         epsilon = np.random.normal()
35         ST *= np.exp((r - q - 0.5*sigma*sigma)*dt + sigma*
            epsilon*sqrt_dt)
36     payoff[i] = option_payoff(ST, K, option_type)
37
38 option_price = np.mean(payoff)*np.exp(-r*T)
39 # Calculate standard deviation of payoffs
40 payoff_std_dev = np.std(payoff) / np.sqrt(
    n_simulated_stock_paths)
41
42 # Print the standard deviation
43 print(option_type + ' price =', round(option_price, 4))
44 print("Standard deviation of payoffs =", round(payoff_std_dev
    , 4))

```

---

Listing 1: Monte Carlo Simulation for European Call Option

**Output:**

call price = 13.3137  
Standard deviation of payoffs = 0.4151

## 2) Antithetic method by sampling paths

**Explanation:** The antithetic variate method is a variance reduction technique in Monte Carlo simulation. In this method, we generate pairs of paths with opposite shocks, helping to reduce the variance in the option price estimate. For each simulation step, we use the following equations to simulate a forward path  $S_{t+1}^1$  and an antithetic path  $S_{t+1}^2$ :

$$S_{t+1}^1 = S_t \times \exp\left((r - q - 0.5\sigma^2)\Delta t + \sigma\epsilon\sqrt{\Delta t}\right)$$
$$S_{t+1}^2 = S_t \times \exp\left((r - q - 0.5\sigma^2)\Delta t - \sigma\epsilon\sqrt{\Delta t}\right)$$

where  $\epsilon$  is a standard normal random variable. The average payoff from the paired paths gives a more stable estimate of the option price.

### Python Code:

---

```
1 # Definiton of the parameters
2
3 import numpy as np
4
5 S0 = 100.0 # initial stock price
6 K = 100.0 # strike price
7 T = 1.0 # time to maturity
8 r = 0.06 # risk-free rate
9 q = 0.06 # dividend yield
10 sigma = 0.35 # volatility
11 n_times_steps = 100 # number of time steps
12 n_simulated_stock_paths = 4000 # number of simulated stock
    paths
13 option_type = "call" # "call" or "put"
14 np.random.seed(42) # make result reproducible
15
16 import numpy as np
17 def option_payoff(S, K, option_type):
18     payoff = 0.0
19     if option_type == "call":
20         payoff = max(S - K, 0)
21     elif option_type == "put":
22         payoff = max(K - S, 0)
```

```

23     return payoff
24
25 # Antithetic method
26
27 dt = T/n_times_steps
28 sqrt_dt = np.sqrt(dt)
29 payoff = np.zeros((n_simulated_stock_paths), dtype = float)
30 step = range(0, int(n_times_steps), 1)
31 for i in range(0, n_simulated_stock_paths):
32     ST_1 = S0
33     ST_2 = S0
34     for j in step:
35         epsilon = np.random.normal()
36         ST_1 *= np.exp((r - q - 0.5*sigma*sigma)*dt + sigma*
37                     epsilon*sqrt_dt)
38         ST_2 *= np.exp((r - q - 0.5*sigma*sigma)*dt - sigma*
39                     epsilon*sqrt_dt)
40     payoff[i] = (option_payoff(ST_1, K, option_type) +
41                 option_payoff(ST_2, K, option_type)) / 2
42
43 option_price = np.mean(payoff)*np.exp(-r*T)
44 # Calculate standard deviation of payoffs
45 payoff_std_dev = np.std(payoff) / np.sqrt(
46     n_simulated_stock_paths)
47
48 # Print the standard deviation
49 print(option_type + ' price =', round(option_price, 4))
50 print("Standard deviation of payoffs =", round(payoff_std_dev
51     , 4))

```

---

Listing 2: Antithetic Method for European Call Option

### Output:

```

call price = 13.1927
Standard deviation of payoffs = 0.2431

```

## 3) Control variate method

**Explanation:** The control variate method is another variance reduction technique where we use a related variable with a known expectation to adjust the estimate. Here, we use the terminal stock price  $S_T$  as the control variate, with its expected value  $E[S_T] = S_0 e^{(r-q)*(T-t)}$ . The adjustment term is scaled by a coefficient  $\beta$ , calculated as:

$$\beta = \frac{\text{Cov}(X, Y)}{\text{Var}(Y)}$$

where  $X$  is the option payoff, and  $Y$  is the control variate  $S_T$ . By adding  $\beta(E[Y] - Y)$  to the payoff, we reduce the variance in the estimate, producing a more accurate price. This adjustment leverages the correlation between  $X$  and  $Y$  to offset the variability in the payoff with the expected stock price.

### Python Code:

---

```

1 import numpy as np
2
3 # Definition of the parameters
4 S0 = 100.0 # initial stock price
5 K = 100.0 # strike price
6 T = 1.0 # time to maturity
7 r = 0.06 # risk-free rate
8 q = 0.06 # dividend yield
9 sigma = 0.35 # volatility
10 n_times_steps = 100 # number of time steps
11 n_simulated_stock_paths = 4000 # number of simulated stock
    paths
12 option_type = "call" # "call" or "put"
13 np.random.seed(42) # make result reproducible
14
15 # Expected terminal stock price under risk-neutral measure
16 expected_ST = S0 * np.exp((r - q) * T)
17
18 # Define the payoff function
19 def option_payoff(S, K, option_type):
20     if option_type == "call":
21         return max(S - K, 0)
22     elif option_type == "put":
23         return max(K - S, 0)
24
25 # First pass to estimate beta
26 dt = T / n_times_steps
27 sqrt_dt = np.sqrt(dt)
28 payoffs = []
29 ST_values = []
30
31 for i in range(n_simulated_stock_paths):
32     ST = S0 # terminal stock price for this path
33     for _ in range(n_times_steps):

```

```

34         epsilon = np.random.normal()
35         ST *= np.exp((r - q - 0.5 * sigma ** 2) * dt + sigma
36                     * epsilon * sqrt_dt)
37
38     payoffs.append(option_payoff(ST, K, option_type))
39     ST_values.append(ST)
40
41 payoffs = np.array(payoffs)
42 ST_values = np.array(ST_values)
43
44 # Estimate beta
45 cov_XY = np.cov(payoffs, ST_values)[0, 1]
46 var_Y = np.var(ST_values)
47 beta = cov_XY / var_Y
48
49 # Control variate method with beta adjustment
50 adjusted_payoffs = payoffs + beta * (expected_ST - ST_values)
51
52 # Discounted price with control variate adjustment
53 option_price = np.mean(adjusted_payoffs) * np.exp(-r * T)
54 # Calculate standard deviation of payoffs
55 payoff_std_dev = np.std(adjusted_payoffs) / np.sqrt(
56     n_simulated_stock_paths)
57
58 # Print the standard deviation
59 print(option_type + ' price =', round(option_price, 4))
60 print("Standard deviation of payoffs =", round(payoff_std_dev
61     , 4))

```

---

Listing 3: Control Variate Method for European Call Option

### Output:

```

call price = 13.213
Standard deviation of payoffs = 0.1669

```

## BSM Formula

### Python Code:

---

```

1 import numpy as np
2 from scipy.stats import norm
3
4 # Parameters
5 S0 = 100.0          # initial stock price

```

```

6 K = 100.0          # strike price
7 T = 1.0            # time to maturity in years
8 r = 0.06           # risk-free interest rate
9 q = 0.06           # continuous dividend yield
10 sigma = 0.35       # volatility
11
12 # Black-Scholes-Merton formula
13 def bsm_call_price(S0, K, T, r, q, sigma):
14     d1 = (np.log(S0 / K) + (r - q + 0.5 * sigma ** 2) * T) /
15         (sigma * np.sqrt(T))
16     d2 = d1 - sigma * np.sqrt(T)
17     call_price = S0 * np.exp(-q * T) * norm.cdf(d1) - K * np.
18         exp(-r * T) * norm.cdf(d2)
19     return call_price
20
21 # Compute the BSM price
22 bsm_price = bsm_call_price(S0, K, T, r, q, sigma)
23
24 # Display the result
25 print(f"BSM Call Option Price: {bsm_price:.2f}")

```

---

Listing 4: BSM Formula for European Call Option

### Output:

BSM Call Option Price: 13.08

## Conclusion

In this assignment, we implemented three methods for pricing a European call option using Monte Carlo simulation. The plain Monte Carlo method provides a basic estimate but can have high variance. The antithetic method reduces variance by simulating pairs of paths with opposite shocks, leading to a more stable price estimate. The control variate method further enhances accuracy by adjusting the payoff based on the known expectation of the terminal stock price, leveraging the correlation between the stock price and the option payoff.

Compared to the Black-Scholes-Merton (BSM) formula, which provides a theoretical benchmark, the Monte Carlo-based methods approximate the price, with the control variate method achieving the closest result. This highlights the effectiveness of variance reduction techniques in improving the

accuracy and stability of Monte Carlo estimates in option pricing. Additionally, calculating the standard deviation of the payoffs provides insight into the variability of the estimates, allowing for a more comprehensive assessment of the methods' performance.