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, - σ is the volatility, - Δt is the time step size, - ϵ 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:

```
# Definiton of the parameters

import numpy as np

S0 = 100.0 # initial stock price
K = 100.0 # strike price
```

```
T = 1.0 # time to maturity
r = 0.06 # risk-free rate
q = 0.06 # dividend yield
sigma = 0.35 # volatility
n_times_steps = 100 # number of time steps
n_simulated_stock_paths = 4000 # number of simulated stock paths
option_type = "call" # "call" or "put"
np.random.seed(42) # make result reproducible

import numpy as np
def option_payoff(S, K, option_type):
    payoff = 0.0
    if option_type == "call":
        payoff = max(S - K, 0)
    elif option_type == "put":
        payoff = max(K - S, 0)
    return payoff

# Monte Carlo simulation

dt = T/n_times_steps
sqrt_dt = np.sqrt(dt)
payoff = np.zeros((n_simulated_stock_paths), dtype = float)
step = range(0, int(n_times_steps), 1)
for i in range(0, n_simulated_stock_paths):
    ST = S0
    for j in step:
        epsilon = np.random.normal()
        ST *= np.exp((r - q - 0.5*sigma*sigma)*dt + sigma*epsilon*sqrt_dt)
    payoff[i] = option_payoff(ST, K, option_type)

option_price = np.mean(payoff)*np.exp(-r*T)
# Calculate standard deviation of payoffs
payoff_std_dev = np.std(payoff) / np.sqrt(n_simulated_stock_paths) * np.exp(-r * T)

# Print the standard deviation
print(option_type + ' price =', round(option_price, 4))
print("Standard deviation of payoffs =", round(payoff_std_dev, 4))
```

Listing 1: Monte Carlo Simulation for European Call Option

Output:

```
call price = 13.31
Standard deviation of payoffs = 0.39
```

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^1_{t+1} and an antithetic path S^2_{t+1}:

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

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

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

Python Code:

```python
# Definiton of the parameters

import numpy as np

S0 = 100.0 # initial stock price
K = 100.0 # strike price
T = 1.0 # time to maturity
r = 0.06 # risk-free rate
q = 0.06 # dividend yield
sigma = 0.35 # volatility
n_times_steps = 100 # number of time steps
n_simulated_stock_paths = 4000 # number of simulated stock
    paths
option_type = "call" # "call" or "put"
np.random.seed(42) # make result reproducible

import numpy as np
def option_payoff(S, K, option_type):
    payoff = 0.0
    if option_type == "call":
        payoff = max(S - K, 0)
    elif option_type == "put":
        payoff = max(K - S, 0)
```

```
    return payoff

# Antithetic method

dt = T/n_times_steps
sqrt_dt = np.sqrt(dt)
payoff = np.zeros((n_simulated_stock_paths), dtype = float)
step = range(0, int(n_times_steps), 1)
for i in range(0, n_simulated_stock_paths):
    ST_1 = S0
    ST_2 = S0
    for j in step:
        epsilon = np.random.normal()
        ST_1 *= np.exp((r - q - 0.5*sigma*sigma)*dt + sigma*
           epsilon*sqrt_dt)
        ST_2 *= np.exp((r - q - 0.5*sigma*sigma)*dt - sigma*
           epsilon*sqrt_dt)
    payoff[i] = (option_payoff(ST_1, K, option_type) +
       option_payoff(ST_2, K, option_type)) / 2

option_price = np.mean(payoff)*np.exp(-r*T)
# Calculate standard deviation of payoffs
payoff_std_dev = np.std(payoff) / np.sqrt(
   n_simulated_stock_paths) * np.exp(-r * T)

# Print the standard deviation
print(option_type + ' price =', round(option_price, 4))
print("Standard deviation of payoffs =", round(payoff_std_dev
   , 4))
```

Listing 2: Antithetic Method for European Call Option

Output:

```
call price = 13.19
Standard deviation of payoffs = 0.23
```

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 β, 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:

```python
import numpy as np

# Definition of the parameters
S0 = 100.0  # initial stock price
K = 100.0  # strike price
T = 1.0  # time to maturity
r = 0.06  # risk-free rate
q = 0.06  # dividend yield
sigma = 0.35  # volatility
n_times_steps = 100  # number of time steps
n_simulated_stock_paths = 4000  # number of simulated stock
    paths
option_type = "call"  # "call" or "put"
np.random.seed(42)  # make result reproducible

# Expected terminal stock price under risk-neutral measure
expected_ST = S0 * np.exp((r - q) * T)

# Define the payoff function
def option_payoff(S, K, option_type):
    if option_type == "call":
        return max(S - K, 0)
    elif option_type == "put":
        return max(K - S, 0)

# First pass to estimate beta
dt = T / n_times_steps
sqrt_dt = np.sqrt(dt)
payoffs = []
ST_values = []

for i in range(n_simulated_stock_paths):
    ST = S0  # terminal stock price for this path
    for _ in range(n_times_steps):
```

```
        epsilon = np.random.normal()
        ST *= np.exp((r - q - 0.5 * sigma ** 2) * dt + sigma
            * epsilon * sqrt_dt)

    payoffs.append(option_payoff(ST, K, option_type))
    ST_values.append(ST)

payoffs = np.array(payoffs)
ST_values = np.array(ST_values)

# Estimate beta
cov_XY = np.cov(payoffs, ST_values)[0, 1]
var_Y = np.var(ST_values)
beta = cov_XY / var_Y

# Control variate method with beta adjustment
adjusted_payoffs = payoffs + beta * (expected_ST - ST_values)

# Discounted price with control variate adjustment
option_price = np.mean(adjusted_payoffs) * np.exp(-r * T)
# Calculate standard deviation of payoffs
payoff_std_dev = np.std(adjusted_payoffs) / np.sqrt(
    n_simulated_stock_paths) * np.exp(-r * T)

# Print the standard deviation
print(option_type + ' price =', round(option_price, 4))
print("Standard deviation of payoffs =", round(payoff_std_dev
    , 4))
```

Listing 3: Control Variate Method for European Call Option

Output:

```
call price = 13.21
Standard deviation of payoffs = 0.16
```

BSM Formula

Python Code:

```
import numpy as np
from scipy.stats import norm

# Parameters
S0 = 100.0        # initial stock price
```

```
K = 100.0          # strike price
T = 1.0            # time to maturity in years
r = 0.06           # risk-free interest rate
q = 0.06           # continuous dividend yield
sigma = 0.35       # volatility

# Black-Scholes-Merton formula
def bsm_call_price(S0, K, T, r, q, sigma):
    d1 = (np.log(S0 / K) + (r - q + 0.5 * sigma ** 2) * T) /
        (sigma * np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)
    call_price = S0 * np.exp(-q * T) * norm.cdf(d1) - K * np.
        exp(-r * T) * norm.cdf(d2)
    return call_price

# Compute the BSM price
bsm_price = bsm_call_price(S0, K, T, r, q, sigma)

# Display the result
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.