Chia-Wei Lai | clai2

Simulation Methods

46-773

Homework #4

In [ ]:
```
import numpy as np
import scipy.stats as stats
```

Problem #1 Completion of Problem 4, Homework #3

(a)

In [ ]:
```
# Parameters
S0 = 100
mu = 0.10
sigma = 0.20
r = 0.05
T = 1
n = 100000
strike_prices = [120, 140, 160]

# Generate standard normal random variables
Z = np.random.normal(0, 1, n)

# Simulate asset prices at maturity
ST = S0 * np.exp((r - 0.5 * sigma**2) * T + sigma * np.sqrt(T) * Z)

# Calculate call option prices for different strike prices
call_prices = {}
for K in strike_prices:
    payoffs = np.maximum(ST - K, 0)
    call_prices_pv = np.exp(-r * T) * payoffs
    stderr = np.std(call_prices_pv) / np.sqrt(n)

    call_prices[K] = np.mean(call_prices_pv), stderr


for K in strike_prices:
    print(f"K = {K}, call price = {call_prices[K][0]:.4f}, stdErr = {call_prices[K][1]:.4f}")
```

```
K = 120, call price = 3.2542, stdErr = 0.0275
K = 140, call price = 0.7913, stdErr = 0.0135
K = 160, call price = 0.1632, stdErr = 0.0060
```

(b)

In [ ]:
```
# Calculate put option prices for different strike prices using Monte Carlo
put_prices = {}
for K in strike_prices:
    payoffs = np.maximum(K - ST, 0)
    put_price_pv = np.exp(-r * T) * payoffs
    call_price_pv = put_price_pv + S0 - K * np.exp(-r * T)
    stderr = np.std(put_price_pv) / np.sqrt(n)

    put_prices[K] = np.mean(put_price_pv)
    call_prices[K] = np.mean(call_price_pv), stderr

for K in strike_prices:
    print(f"K = {K}, call price = {call_prices[K][0]:.4f}, stdErr = {call_prices[K][1]:.4f}")
```

```
K = 120, call price = 3.3287, stdErr = 0.0470
K = 140, call price = 0.8658, stdErr = 0.0581
K = 160, call price = 0.2377, stdErr = 0.0624
```

(c)

In [ ]:
```
# Control variate technique
expected_ST = S0 * np.exp(r * T)
b_values = {}
adjusted_call_prices = {}

for K in strike_prices:
    payoffs = np.maximum(K - ST, 0)
    cov = np.cov(payoffs, ST)[0, 1]
    var = np.var(ST)
    b = cov / var
    adjusted_payoffs = payoffs - b * (ST - expected_ST)
    adjusted_put_price_pv = np.exp(-r * T) * adjusted_payoffs
    stderr = np.std(adjusted_put_price_pv) / np.sqrt(n)

    adjusted_call_prices[K] = np.mean(adjusted_put_price_pv) + S0 - K * np.exp(-r * T), stderr


for K in strike_prices:
    print(f"K = {K}, call price = {adjusted_call_prices[K][0]:.4f}, stdErr = {adjusted_call_prices[K][1]:.4f}")
```

```
K = 120, call price = 3.2783, stdErr = 0.0181
K = 140, call price = 0.7995, stdErr = 0.0115
K = 160, call price = 0.1654, stdErr = 0.0057
```

(d)

In [ ]:
```
call_prices = {}

for K in strike_prices:
    L = (np.log(K / S0) - (r - 0.5 * sigma**2) * T) / (sigma * np.sqrt(T))
    u = np.random.uniform(size = n)
    x = stats.norm.ppf(u * (1 - stats.norm.cdf(L)) + stats.norm.cdf(L))

    ST = S0 * np.exp((r - 0.5 * sigma**2) * T + sigma * np.sqrt(T) * x)
    ci = np.exp(-r * T) * (ST - K) * (1 - stats.norm.cdf(L))

    call_prices[K] = ci.mean(), ci.std(ddof = 1) / np.sqrt(n)
for K in strike_prices:
    print(f"K = {K}, call price = {call_prices[K][0]:.4f}, stdErr = {call_prices[K][1]:.4f}")
```

```
K = 120, call price = 3.2533, stdErr = 0.0093
K = 140, call price = 0.7868, stdErr = 0.0023
K = 160, call price = 0.1592, stdErr = 0.0005
```

Problem #2 Practice on conditional Monte Carlo and importance sampling: barrier options

(a)

In [ ]:
```
# Parameters
S0 = 95
K_values = [96, 96, 96, 106]  # Example strike price
H_values = [94, 90, 85, 90]  # Example barrier
r = 0.05
```

```python
sigma = 0.15
T = 0.25
m = 50
n = 100000
dt = T / m

# Standard Monte Carlo
def simulate_gbm(S0, r, sigma, T, m, n):
    dt = T / m
    S = np.zeros((n, m+1))
    S[:, 0] = S0
    for t in range(1, m+1):
        Z = np.random.normal(0, 1, n)
        S[:, t] = S[:, t-1] * np.exp((r - 0.5 * sigma**2) * dt + sigma * np.sqrt(dt) * Z)
    return S

S_paths = simulate_gbm(S0, r, sigma, T, m, n)
M = np.min(S_paths[:, 1:], axis=1)
S_T = S_paths[:, -1]

std_mc = {}
for K, H in zip(K_values, H_values):
    payoffs = np.exp(-r * T) * np.where((S_T > K) & (M < H), 1, 0)
    option_price = np.mean(payoffs)
    standard_error = np.std(payoffs) / np.sqrt(n)
    std_mc[(K, H)] = standard_error
    print(f"H = {H}, K = {K}, Option Price = {option_price:.4f}, stdErr = {standard_error:.4f}")
```

```
H = 94, K = 96, Option Price = 0.3012, stdErr = 0.0014
H = 90, K = 96, Option Price = 0.0427, stdErr = 0.0006
H = 85, K = 96, Option Price = 0.0006, stdErr = 0.0001
H = 90, K = 106, Option Price = 0.0013, stdErr = 0.0001
```

(b)

- (a)

$$\begin{aligned}
\mathbb{E}^{\mathbb{Q}}[e^{-rT}1_{\{S_T>K\}}] &= e^{-rT}\mathbb{P}(S_T > K) \\
&= e^{-rT}\Phi(d_2) \\
\text{where } d_2 &= \frac{log(\frac{S_0}{K}) + (r - 0.5\sigma^2)T}{\sigma\sqrt(T)}
\end{aligned}$$

In [ ]:

```python
# Digital option price using Black-Scholes formula
def digital_call_price(S, K, r, sigma, T):
    if T == 0:
        return (S > K) * 1
    else:
        d2 = (np.log(S / K) + (r - 0.5 * sigma**2) * T) / (sigma * np.sqrt(T))
        return np.exp(-r * T) * stats.norm.cdf(d2)
```

- (b)
- (c)

In [ ]:

```python
# Conditional Monte Carlo
for K, H in zip(K_values, H_values):
    payoffs_cmc = np.zeros(n)
    for i in range(n):
        path = S_paths[i, :]
        if np.any(path < H):
            tau_idx = np.where(path < H)[0][0]
            tau = tau_idx * dt
            S_tau = path[tau_idx]
            digital_price = digital_call_price(S_tau, K, r, sigma, T - tau)
            payoffs_cmc[i] = np.exp(-r * tau) * digital_price

    option_price_cmc = np.mean(payoffs_cmc)
    standard_error_cmc = np.std(payoffs_cmc) / np.sqrt(n)

    print(f"T = {T}, m = {m}, H = {H}, K = {K}, Price = {option_price_cmc:.4f} Variance Ratio: {(std_mc[(K, H)] / standard_error_cmc)**2:.4f}")
```

```
T = 0.25, m = 50, H = 94, K = 96, Price = 0.3006 Variance Ratio: 8.0273
T = 0.25, m = 50, H = 90, K = 96, Price = 0.0425 Variance Ratio: 9.1370
T = 0.25, m = 50, H = 85, K = 96, Price = 0.0005 Variance Ratio: 58.9051
T = 0.25, m = 50, H = 90, K = 106, Price = 0.0013 Variance Ratio: 147.2727
```

Problem #3 Discrete versus continuous pricing

In [ ]:

```python
# Parameters
S0 = 100
K = 100
H = 95
sigma = 0.30
r = 0.10
T = 0.2
n_simulations = 100000
N_values = [25, 50]

def bs_call_price(S0, K, sigma, r, T):
    if T == 0:
        return max(S0 - K, 0)
    else:
        d1 = (np.log(S0/ K) + (r + 0.5 * sigma**2) * T) / (sigma * np.sqrt(T))
        d2 = d1 - sigma * np.sqrt(T)
        return S0 * stats.norm.cdf(d1) - K * np.exp(-r * T) * stats.norm.cdf(d2)

def down_and_in_call(S0, K, H, sigma, r, T):
    lamb = (r + sigma**2 / 2) / sigma**2
    y = np.log(H**2 / (S0 * K)) / (sigma * np.sqrt(T)) + lamb * sigma * np.sqrt(T)
    return S0 * (H / S0)**(2 * lamb) * stats.norm.cdf(y) - K * np.exp(-r * T) * (H / S0)**(2 * lamb - 2) * stats.norm.cdf(y - sigma * np.sqrt(T))
```

In [ ]:

```python
print(f"The closed-form of down and in call price = {down_and_in_call(S0, K, H, sigma, r, T):.4f}")
```

```
The closed-form of down and in call price = 1.9466
```

(a)

In [ ]:

```python
# Function to simulate price paths
def simulate_paths(S0, r, sigma, T, N, n_simulations):
    dt = T / N
    paths = np.zeros((n_simulations, N + 1))
    paths[:, 0] = S0
    for t in range(1, N + 1):
        z = np.random.normal(0, 1, n_simulations)
        paths[:, t] = paths[:, t-1] * np.exp((r - 0.5 * sigma**2) * dt + sigma * np.sqrt(dt) * z)
    return paths

# Standard Monte Carlo
for N in N_values:
    paths = simulate_paths(S0, r, sigma, T, N, n_simulations)
    payoffs_pv = np.exp(-r * T)  * np.maximum(paths[:, -1] - K, 0) * (np.min(paths, axis=1) < H)
```

```
    price = np.mean(payoffs_pv)
    stdErr = np.std(payoffs_pv, ddof = 1) / np.sqrt(n)
    print(f"Standard Monte Carlo price for N = {N}: {price:.4f}, stdErr = {stdErr:.4f}")
```

```
Standard Monte Carlo price for N = 25: 1.2391, stdErr = 0.0124
Standard Monte Carlo price for N = 50: 1.4542, stdErr = 0.0136
```

(b)

In [ ]:

```
# Conditional Monte Carlo
for N in N_values:
    paths = simulate_paths(S0, r, sigma, T, N, n_simulations)
    conditional_payoffs = np.zeros(n_simulations)
    for i in range(n_simulations):
        if np.min(paths[i, :]) < H:

            first_hit = np.where(paths[i, :] < H)[0][0]
            conditional_payoffs[i] = bs_call_price(paths[i, first_hit], K, sigma, r, T * (N - first_hit) / N)
    payoffs_pv = np.exp(-r * T * (first_hit / N)) * conditional_payoffs
    price = np.mean(payoffs_pv)
    stdErr = np.std(payoffs_pv, ddof = 1) / np.sqrt(n)
    print(f"Conditional Monte Carlo price for N = {N}: {price:.4f}, stdErr = {stdErr:.4f}")
```

```
Conditional Monte Carlo price for N = 25: 1.2575, stdErr = 0.0040
Conditional Monte Carlo price for N = 50: 1.4317, stdErr = 0.0042
```

(c)

In [ ]:

```
# Conditional Monte Carlo with Importance Sampling
theta_values = [-0.45, -0.30]
for theta, N in zip(theta_values, N_values):
    dt = T / N
    paths = np.zeros((n_simulations, N + 1))
    paths[:, 0] = S0

    z = np.random.normal(theta, 1, (n_simulations, N))
    paths[:, 1:] = S0 * np.exp(np.cumsum((r - 0.5 * sigma**2) * dt + sigma * np.sqrt(dt) * z, axis = 1))

    conditional_payoffs = np.zeros(n_simulations)
    for i in range(n_simulations):
        if np.min(paths[i, :]) < H:
            first_hit = np.where(paths[i, :] < H)[0][0]
            conditional_payoffs[i] = bs_call_price(paths[i, first_hit], K, sigma, r, T * (N - first_hit) / N) * np.prod(np.exp(-theta * z[i, :first_hit] + 0.5 * theta**2
    payoffs_pv = np.exp(-r * T * (first_hit / N)) * conditional_payoffs
    price = np.mean(payoffs_pv)
    stdErr = np.std(payoffs_pv, ddof = 1) / np.sqrt(n)
    print(f"Conditional Monte Carlo with Importance Sampling price for N = {N}, theta = {theta}: {price:.4f}, stdErr = {stdErr:.4f}")
```

```
Conditional Monte Carlo with Importance Sampling price for N = 25, theta = -0.45: 1.2635, stdErr = 0.0017
Conditional Monte Carlo with Importance Sampling price for N = 50, theta = -0.3: 1.4183, stdErr = 0.0015
```

Problem #4 Practice on Conditional Monte Carlo

(a)

In [ ]:

```
# Parameters
S0 = K = 100
r = 0.05
T = 1
v0 = 0.04
alpha_values = [0.1, 0.2]
psi_values = [0.1, 1]
N = 50  # Time steps
M = 10000  # Number of paths
dt = T / N

for alpha, psi in zip(alpha_values, psi_values):
    # Initialize arrays for paths
    S_paths = np.zeros((M, N+1))
    v_paths = np.zeros((M, N+1))

    # Set initial values
    S_paths[:, 0] = S0
    v_paths[:, 0] = v0

    # Simulate paths
    for t in range(1, N+1):
        Z1 = np.random.normal(0, 1, M)
        Z2 = np.random.normal(0, 1, M)

        v_paths[:, t] = v_paths[:, t-1] * np.exp((alpha - 0.5 * psi**2) * dt + psi * np.sqrt(dt) * Z1)
        S_paths[:, t] = S_paths[:, t-1] + r * (S_paths[:, t-1]) * dt + np.sqrt(v_paths[:, t-1] * dt) * (S_paths[:, t-1]) * Z2

    # Compute option prices
    payoffs = np.maximum(S_paths[:, -1] - K, 0)
    discounted_payoffs = np.exp(-r * T) * payoffs
    price_mc = np.mean(discounted_payoffs)
    std_error_mc = np.std(discounted_payoffs) / np.sqrt(M)

    print(f"(alpha, psi) = ({alpha}, {psi}), call price = {price_mc:.4f}, stdErr = {std_error_mc:.4f}")
```

```
(alpha, psi) = (0.1, 0.1), call price = 10.4516, stdErr = 0.1488
(alpha, psi) = (0.2, 1), call price = 10.5457, stdErr = 0.1551
```

(b)

In [ ]:

```
for alpha, psi in zip(alpha_values, psi_values):
    # Initialize arrays for paths
    v_paths = np.zeros((M, N+1))

    # Set initial values
    v_paths[:, 0] = v0

    # Simulate paths
    for t in range(1, N+1):
        Z2 = np.random.normal(0, 1, M)

        v_paths[:, t] = v_paths[:, t-1] * np.exp((alpha - 0.5 * psi**2) * dt + psi * np.sqrt(dt) * Z2)

    sigma_avg = np.sqrt(np.sum(v_paths[:, 1:], axis = 1) / N)
    call_prices = bs_call_price(S0, K, sigma_avg, r, T)

    price_cmc = np.mean(call_prices)
    stdErr_cmc = np.std(call_prices, ddof = 1) / np.sqrt(M)
    print(f"(alpha, psi) = ({alpha}, {psi}), call price = {price_cmc:.4f}, stdErr = {stdErr_cmc:.4f}")
```

```
(alpha, psi) = (0.1, 0.1), call price = 10.6457, stdErr = 0.0023
(alpha, psi) = (0.2, 1), call price = 10.5321, stdErr = 0.0233
```

Problem #5 Practice on interest rate derivatives and CIR

(a)

In [ ]:

```
# CIR model parameters
alpha = 0.2
sigma = 0.1
b = 0.05
```

```
r0 = 0.04

# Simulation parameters
T = 1
n_paths = 1000
n_steps = 50
dt = T / n_steps

# Initialize the rate paths
rates = np.zeros((n_paths, n_steps + 1))
rates[:, 0] = r0

# Simulate the paths using the CIR model
for t in range(1, n_steps + 1):
    z = np.random.normal(size=n_paths)
    rates[:, t] = rates[:, t-1] + alpha * (b - rates[:, t-1]) * dt + sigma * np.sqrt(np.maximum(rates[:, t-1], 0)) * np.sqrt(dt) * z

# Calculate zero-coupon bond prices
zero_coupon_prices = np.exp(-np.sum(rates[:, :-1], axis=1) * dt)

# Average the zero-coupon bond prices
zero_coupon_price = np.mean(zero_coupon_prices)
zero_coupon_price_std = np.std(zero_coupon_prices) / np.sqrt(n_paths)

print(f"Zero-coupon bond price = {zero_coupon_price:.4f}, stdErr = {zero_coupon_price_std:.4f}")
```

```
Zero-coupon bond price = 0.9601, stdErr = 0.0003
```

(b)

In [ ]:

```
# Caplet parameters
L = 1
delta = 1 / 12
R = 0.05
t = 1
caplet_times = int(t / dt)

# Calculate the caplet payoff at t = 1
payoffs = L * delta * np.maximum(0, rates[:, caplet_times] - R)

# Discount the payoffs back to the present
discounted_payoffs = payoffs * np.exp(-np.sum(rates[:, :caplet_times], axis=1) * dt)

# Average the discounted payoffs to get the caplet price
caplet_price = np.mean(discounted_payoffs)
caplet_price_std = np.std(discounted_payoffs) / np.sqrt(n_paths)
print(f"Caplet price = {caplet_price:.6f}, stdErr = {caplet_price_std:.6f}")
```

```
Caplet price = 0.000312, stdErr = 0.000023
```

Appendix: timestamp

In [ ]:

```
from datetime import datetime

print(f"Generated on {datetime.now()}")
```

```
Generated on 2024-05-27 22:23:41.019279
```