Vrije Universiteit Amsterdam
Period 1, 2024-2025

Computer Assignment 1

Stochastic Processes : The Fundamentals

Instructors:
Eva Mynott and Mark-jan Boes

Yunji Eo (2735445)
y.eo@student.vu.nl
Archit Murkunde (2845576)
a.u.murkunde@student.vu.nl

Econometrics and Operations Research

September 23, 2024

1 Returns

1.1 (a)

	coefficient	σ	t	p-value	[0.275	0.975]
μ	1.000	0.417	2.399	0.018	0.172	1.828

Table 1: Result of the Estimation

$$R_{t:t+1} = \mu + \sigma\epsilon_{t:t+1} \tag{1}$$

$$\epsilon_{t:t+1} \sim N(0,1). \tag{2}$$

The table 1 exhibits the result of the estimation from the model 8. The coefficient for μ is 1.000, with a p-value of 0.018. Since the p-value is less then 0.05, we reject the null hypothesis. This means that μ is significantly different from zero at a 5% significance level.

1.2 (b)

$$N \geq \frac{z^2_{\alpha/2} \cdot \sigma^2}{12 \cdot \epsilon^2} \tag{3}$$

To calculate how many years of return data are needed to find a 96% interval [0.35%, 0.45%] for μ, the expression in Equation 3 is used. This expression is derived from the properties of standard error of the mean and the confidence interval. Notice that 12 is devided to convert the monthly data into years. It is assumed that the true parameters of model 8 are $\mu = 0.4\%$ and $\sigma = 6\%$. By plugging the values, we obtained :

$$N \geq \frac{(1.96)^2 \cdot (0.06)^2}{12 \cdot (0.0005)^2} = 4609.92 \approx 4610 \tag{4}$$

Hence, 4610 years of data are required to find a 96% interval [0.35%, 0.45%] for μ.

1.3 (c)

	coefficient	σ	t	p-value	[0.275	0.975]
μ	1.0542	0.004	275.180	0.000	1.047	1.062

Table 2: Result of the Estimation

We are asked to verify the number of years of return data, which is obtained from the previous question, required to find a 95% confidence interval for μ. First, we initialized the simulation with the parameters : $\mu = 0.04$, $\sigma = 0.06$, n = 55319. We also generated n standard normal random variables. Then, by using the Equation 8, we simulated returns based on the parameters. Finally, OLS regression is conducted with the simulated data to estimate μ.
The result of simulation is summarized in Table 2. The confidence interval is [1.047, 1.062], which is very close to the value of the coefficient (1.0542) and the interval is also narrow. Therefore, we can conclude that the simulation is conducted successfully.

1.4 (d)

From this question an alternative way is introduced,

$$S_{t+1} = S_t e^{\left(\tilde{\mu} - \frac{1}{2}\tilde{\sigma}^2\right) + \tilde{\sigma}\epsilon_{t:t+1}} \tag{5}$$

To estimate the parameters $\tilde{\mu}$ and $\tilde{\sigma}$, we first begin by calculating the log returns of the index :

$$\text{Log Ret}_t = \log\left(\frac{S_t}{S_{t-1}}\right) = \tilde{\mu} - \frac{1}{2}\tilde{\sigma}^2 + \tilde{\sigma}\epsilon_{t:t+1} \tag{6}$$

Then, the term $\tilde{\mu} - \frac{1}{2}\tilde{\sigma}^2$ is considered as the mean of log-return denoted as μ_{ret}. As a result, the parameter $\tilde{\mu}$ can be calculated as :

$$\tilde{\mu} = \mu_{ret} + \frac{1}{2}\tilde{\sigma}^2 \tag{7}$$

By plugging the values, we obtained the value of 1.1% for $\tilde{\mu}$ and 4.7% for $\tilde{\sigma}$ respectively.

1.5 (e)

Expected value of the index after n months is calculated as :

$$S_{t+n} = S_t \cdot e^{n\left(\tilde{\mu} - \frac{1}{2}\sigma^2\right) + \sigma\sum_{i=1}^{n}\epsilon_{t+i}}$$

$$E(S_{t+n}) = E\left(S_t \cdot e^{n\left(\tilde{\mu} - \frac{1}{2}\sigma^2\right) + \sigma\sum_{i=1}^{n}\epsilon_{t+i}}\right)$$

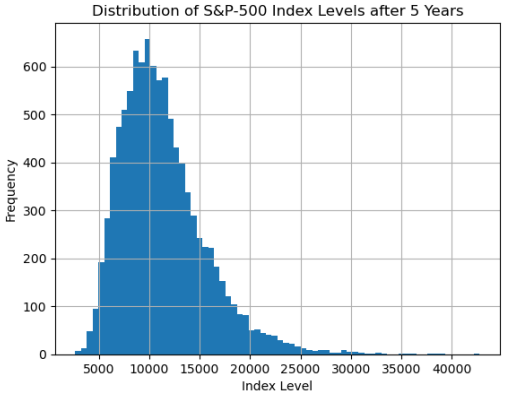
$$= E(S_t) \cdot E\left(e^{n\left(\tilde{\mu} - \frac{1}{2}\sigma^2\right) + \sigma\sum_{i=1}^{n}\epsilon_{t+i}}\right) \quad \text{where} \quad \epsilon_{t+i} \sim N(0,1)$$

$$= S_t \cdot \left(n\tilde{\mu} - \frac{1}{2}n\sigma^2 + \frac{1}{2}n\sigma^2\right)$$

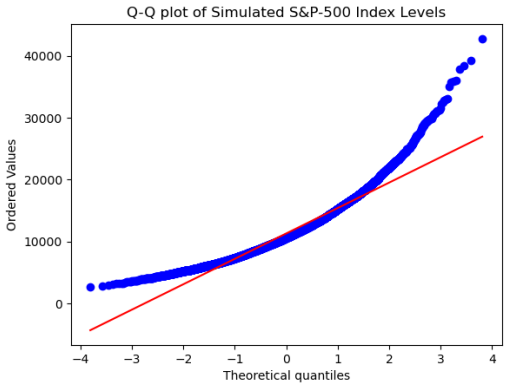
$$= S_t \cdot n\tilde{\mu}$$

Notice that $E(S_t)$ is S_t itself as the value is known at time t. $E(e^x) = \mu + \frac{\sigma^2}{2}$ where x is normally distributed. After plugging the values ($\tilde{\mu} = 4.7\%$, n = 60, S_0), we obtained the expected value of 3893.19.

1.6 (f)



(a) Plot of the Index Levels after 5 years



(b) Q-Q Plot

Figure 1: Plot of the Index Levels and Q-Q Plot after 5 years

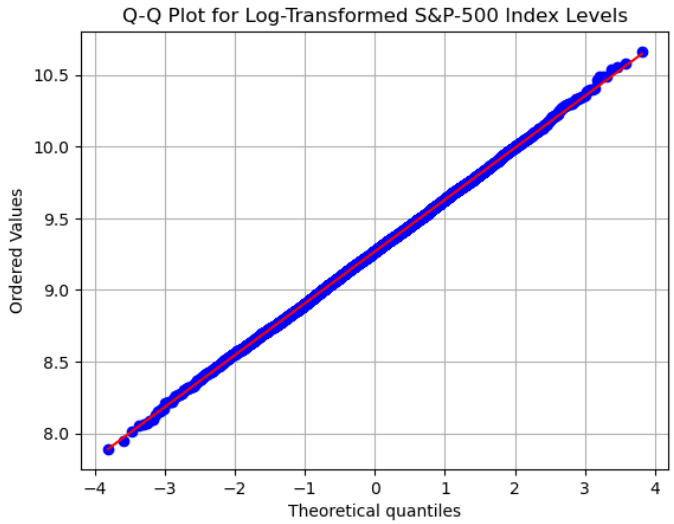


Figure 2: Q-Q Plot for Log-Transformed Index

The histogram 1a of the index levels shows a skewed distribution with a long right tail, which is characteristic of a log-normal distribution. In the Q-Q plot 1b, the

points deviate from the straight line noticeably, especially in the tails, indicating the simulated data is not normally distribution. To check whether the data is significantly log-normal distributed, we generate a Q-Q plot 2 for the log-transformed data. The points lie mostly on the red line, implying that the log-transformed data match the quantiles of a normal distribution. We also conducted the Shaprio-Wilk test and obtained the t-value of 0.99984 and the p-value of 0.7311, indicating that there is no significant evidence to suggest that the transformed data is not normally distributed. Hence, we can confirm that the simulated data is log-normal distributed. This result makes sense as the model 9 involves the exponent of a normally distribution variable. When this variable is taken, the result follows a log-normal distribution.

1.7 (g)

To find the no-arbitrage price of a 5-years digital put option, we used the risk-neutral valuation method with 10,000 simulations of the index, assuming a constant risk-free interest rate of 0%. After running the simulations based on the model 5, we calculated the risk-neutral probabilities where the index was above and below 5,600. The probabilities where the index exceeded and below 5,600 were 0.0385 and 0.9615 respectively. Since the digital put option pays 1 when the index is below 5,600, the price of the option is simply the probability. Therefore, the no-arbitrage price of the digital put option is 0.9615.

1.8 (h)

Using the 10,000 simulations from part (f), we calculated that the probability of the index being above 5,600 is 0.0004, which is the price of the digital call option. The sum of the digital put option price (0.9615) and the digital call option price (0.0385) equals 1. This makes sense because the index will either be above or below 5,600, meaning that one of the two options will always pay 1. As a result, the total value of the put and call options together is always 1, which follows the no-arbitrage rule.

2 Binomial Trees

2.1 (a)

We calculated the values of u and d by solving two equations:

$$0.56 \cdot r_u + 0.44 \cdot r_d = 0.006 \tag{8}$$

$$0.56 \cdot (r_u - 0.006)^2 + 0.44 \cdot (r_d - 0.006)^2 = V \tag{9}$$

Where $u = 1 + r_u$ and $d = 1 + r_d$.
Our values for u and d are 0.531826 and 0.468173 respectively.

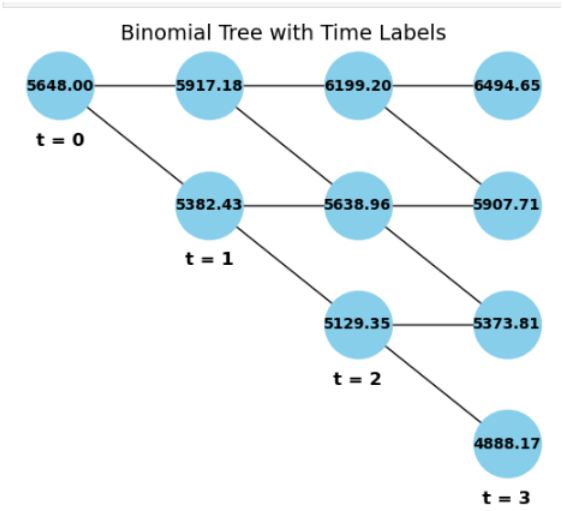


Figure 3: Binomial Tree with Time Labels

2.2 (b)

We calculated the risk-neutral probabilities using the following equations:

$$q_u = \frac{(1+R)-d}{u-d} \tag{10}$$

$$q_d = \frac{u-(1+R)}{u-d} \tag{11}$$

The 3-month european call option price we derived using these risk-neutral probabilities, K = 5600 and time to maturity as 3-months is **254.26**.

2.3 (c)

We calculated price of the same option using Black Scholes method, the price came out to **238.23**.

Price of European call option using Binomial Model: **254.263**
Price of European call option using BSM Model: **238.24**

Possible reason for the difference: The Binomial Model uses discrete steps for price changes, which can sometimes slightly overestimate the option price, especially with fewer steps. In contrast, the Black-Scholes Model assumes continuous price changes and constant volatility, making it more accurate for theoretical pricing with smooth price movements.

2.4 (d)

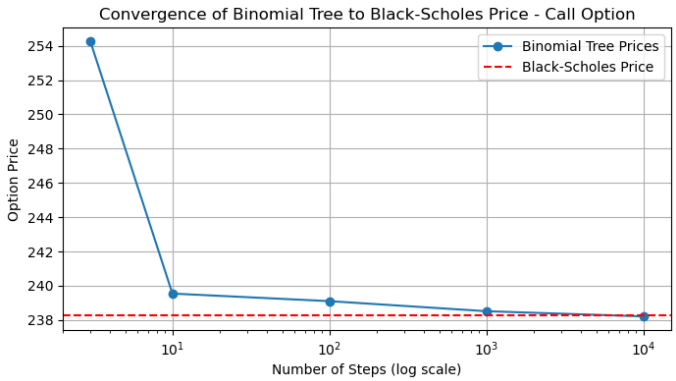


Figure 4: Convergence of Binomial Tree Price to Black-Scholes Price - Call Option

The graph in Figure 6 illustrates the convergence of option prices calculated using the binomial tree method to the Black-Scholes model for a call option. As the number of steps in the binomial tree increases, the prices calculated by the binomial tree (blue line) approach the price given by the Black-Scholes model (red dotted line).

2.5 (e)

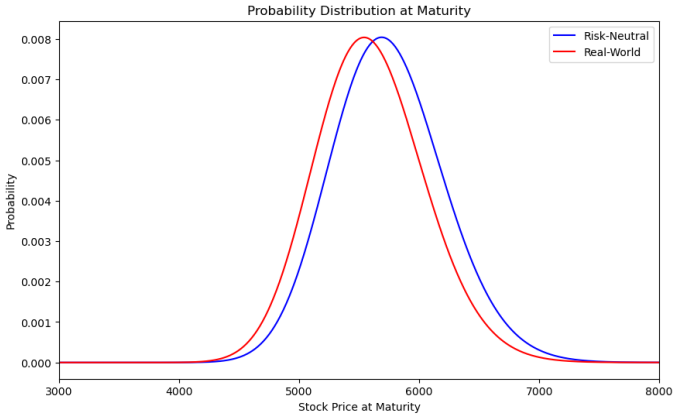


Figure 5: Risk-neutral vs. real-world probability distributions at maturity.

The slight shift between the two distributions indicates that, under the real-world measure, investors demand a higher expected return (compensation for risk), implying a positive excess return for holding the option relative to the risk-free rate.

2.6 (f)

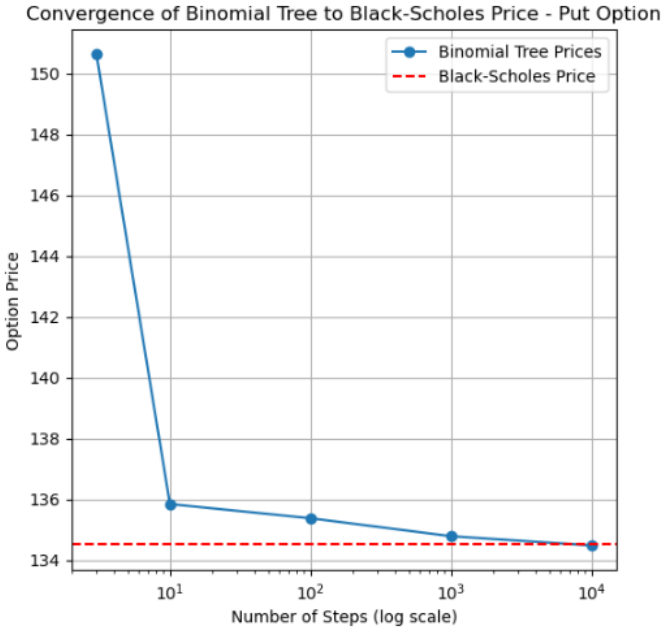


Figure 6: Convergence of Binomial Tree Price to Black-Scholes Price - Put Option

Price of 3-month European put option using Binomial Model: **150.63**
Price of 3-month European put option using BSM Model: **134.51**

Put-Call Parity

$$C - P = S_0 - K\left(1 + \frac{r}{n}\right)^{-nT}$$

For 3-month European option with Strike price of 5600 and risk-free rate of 4%

LHS = 103.63
RHS = 103.63
The Put-Call Parity holds in case of 3-month options.

2.7 (g)

American Call Price: 254.26
European Call Price: 254.26

American Put Price: 154.68
European Put Price: 150.63

The prices of American and European call options are identical, indicating no advantage in early exercise, which is expected as early exercise of call options is rarely beneficial, especially with low interest rates. However, for put options, the American option is priced slightly higher at 154.68 compared to the European put at 150.63, reflecting a small difference of 4.05. This premium represents the value of early exercise, which can be favorable in certain market conditions, particularly when the option is deep in-the-money. In this case, the modest difference suggests early exercise is not significantly advantageous.

3 Appendix

3.1 Question 1

```
Y = returns.dropna()
X = pd.Series(mean, index=range(len(Y)))
X.index = Y.index
model = sm.OLS(Y,X)
results = model.fit()
```

Listing 1: Question 1a

```
mu = 0.004
sigma = 0.06
z = 1.96 # 95% confidence interval
N = ((sigma*z)/(0.0045 - mu))**2
print("Years of Data required: ", N/12)
```

Listing 2: Question 1b

```
mu = 0.004
sigma = 0.06
n = N
e = np.random.normal(mu, sigma, int(np.round(n,0)))
d = sigma*e + mu
Y = d
X = pd.Series(mu, index=range(len(Y)))
model = sm.OLS(Y,X)
results = model.fit()
```

Listing 3: Question 1c

```
data_d['Log Ret'] = np.log(data_d/data_d.shift(-1))
sigma = data_d['Log Ret'].std()
mu = data_d['Log Ret'].mean() + 0.5 * sigma**2
```

Listing 4: Question 1d

```
#simulation
S_0 = data.iloc[0]['SP500']# initial value
simulations = 10000
```

```
months = 60

simulated_index = []

for _ in range(simulations) :
    eps = np.random.normal(0,1,months)
    index = S_0 * np.exp(np.cumsum((mu - 0.5 * sigma**2) + sigma * eps))
    simulated_index.append(index[-1])
```

Listing 5: Question 1f

```
simulated_index_array = np.array(simulated_index)
q_u = np.sum(simulated_index_array > strike) / len(simulated_index_array)
q_d = np.sum(simulated_index_array < strike) / len(simulated_index_array)
```

Listing 6: Question 1g&h

3.2 Question 2

```
def calculate_u_d(steps):

    r_u, r_d = sp.symbols('r_u r_d')

    # Monthly expected return and variance
    monthly_return = 0.006
    monthly_variance = 0.047**2

    # Adjust return and variance based on steps
    adjusted_return = monthly_return * (3 / steps)
    adjusted_variance = monthly_variance * (3 / steps)

    # Define the two equations
    expected_return_eq = 0.56 * r_u + 0.44 * r_d - adjusted_return  # Expected return equation
    variance_eq = 0.56 * (r_u - adjusted_return)**2 + 0.44 * (r_d - adjusted_return)**2 - adjusted_variance  # Variance equation

    # Solve the system of equations
    solution_discrete = sp.solve([expected_return_eq, variance_eq], (r_u, r_d))

    # Calculate u and d from the returns
    u_1 = np.round(float(1 + solution_discrete[0][0]), 5)  # First set (u and d)
    d_1 = np.round(float(1 + solution_discrete[0][1]), 5)

    return u_1, d_1
```

Listing 7: Question 2a

```python
def calculate_u_d(steps):

    r_u, r_d = sp.symbols('r_u r_d')

    # Monthly expected return and variance
    monthly_return = 0.006
    monthly_variance = 0.047**2

    # Adjust return and variance based on steps
    adjusted_return = monthly_return * (3 / steps)
    adjusted_variance = monthly_variance * (3 / steps)

    # Define the two equations
    expected_return_eq = 0.56 * r_u + 0.44 * r_d - adjusted_return
    # Expected return equation
    variance_eq = 0.56 * (r_u - adjusted_return)**2 + 0.44 * (r_d -
    adjusted_return)**2 - adjusted_variance  # Variance equation

    # Solve the system of equations
    solution_discrete = sp.solve([expected_return_eq, variance_eq],
    (r_u, r_d))

    # Calculate u and d from the returns
    u_1 = np.round(float(1 + solution_discrete[0][0]), 5)  # First
   set (u and d)
    d_1 = np.round(float(1 + solution_discrete[0][1]), 5)

    return u_1, d_1

    #Calculation Risk-neutral Probabilities
    r = 0.04
    u = u_1
    d = d_1

   # Using the formulas provided for risk-neutral probabilities
   R = r/12 # Risk-free rate per month

   # Calculate q_u and q_d
   q_u = ((1 + R) - d) / (u - d)
   q_d = (u - (1 + R)) / (u - d)
```

Listing 8: Question 2a

```python
# Visualization function
def visualize_binomial_tree(stock_tree, time_labels):
    G = nx.DiGraph()

    # Create edges with labels
    for i in range(stock_tree.shape[1]):
        for j in range(i + 1):
            G.add_node(f'{i},{j}', label=f'{stock_tree[j, i]:.2f}')
            if i < stock_tree.shape[1] - 1:
```

```
                G.add_edge(f'{i},{j}', f'{i+1},{j}', weight='U')  # Up move
                G.add_edge(f'{i},{j}', f'{i+1},{j+1}', weight='D')  # Down move

    # Create positions for the nodes
    pos = {}
    for i in range(stock_tree.shape[1]):
        for j in range(i + 1):
            pos[f'{i},{j}'] = (i, -j)  # Layout the nodes in a triangular pattern

    labels = nx.get_node_attributes(G, 'label')

    # Draw the graph
    plt.figure(figsize=(5, 4))
    nx.draw(G, pos, labels=labels, with_labels=True, node_size=2000, node_color='skyblue', font_size=10, font_weight='bold', arrows=False)

    # Add time labels
    for i in range(stock_tree.shape[1]):
        plt.text(i, -i - 0.5, f't = {time_labels[i]}', horizontalalignment='center', fontsize=12, fontweight='bold')

    plt.title("Binomial Tree with Time Labels", size=14)
    plt.show()

# Function to generate binomial tree using the calculated u and d values
def generate_binomial_tree(S0, u, d, N, output = "graph"):

    # Initialize the binomial tree
    stock_tree = np.zeros((N + 1, N + 1))

    # Fill the tree with stock prices
    for i in range(N + 1):
        for j in range(i + 1):
            stock_tree[j, i] = S0 * (u ** (i - j)) * (d ** j)

    # Output options
    if output == "graph" or output == "both":
        visualize_binomial_tree(stock_tree, [i for i in range(N + 1)])

    if output == "tree" or output == "both":
        print("Stock Price Binomial Tree:")
        print(stock_tree)

# Example input values
S0 = 5648  # Initial S&P-500 index level
u = u_1 # Calculated upward factor
```

```
d = d_1  # Calculated downward factor
N = 3  # Number of time steps (3 months)

# Generate and visualize the binomial tree
generate_binomial_tree(S0, u, d, N, output="both")
```

Listing 9: Question 2a

```
# Parameters
S0 = 5648
K = 5600
r = 0.04
N = 3  # Number of steps
u, d = calculate_u_d(3)

R = (r*0.25)/N # Monthly Rate

# Step 1: Initialize the stock price tree
stock_tree = np.zeros((N + 1, N + 1))

# Step 2: Fill the stock price tree
for i in range(N + 1):
    for j in range(i + 1):
        stock_tree[j, i] = S0 * (u ** (i - j)) * (d ** j)

# Step 3: Initialize the option price tree
option_tree = np.zeros((N + 1, N + 1))

# Step 4: Calculate the option payoff at maturity (t = N)
for j in range(N + 1):
    option_tree[j, N] = max(stock_tree[j, N] - K, 0)  # Payoff at
    maturity

# Step 5: Backward induction to calculate option price at earlier
    nodes
for i in range(N - 1, -1, -1):
    for j in range(i + 1):
        option_tree[j, i] = (q_u * option_tree[j, i + 1] + q_d *
    option_tree[j + 1, i + 1]) / (1 + R)

# The option price at the root of the tree (at t = 0)
option_price_at_t0 = option_tree[0, 0]
print("Price of 3-month European Call Option is", np.round(
    option_price_at_t0,4))
```

Listing 10: Question 2b

```
from scipy.stats import norm

def black_scholes(S0, K, r, T, sigma, option_type): # function for
    BSM model
    d1 = (np.log(S0 / K) + (r + 0.5 * sigma**2) * T) / (sigma * np.
    sqrt(T))
```

```
    d2 = d1 - sigma * np.sqrt(T)

    if option_type == "Call":
        option_price = S0 * norm.cdf(d1) - K * np.exp(-r * T) * norm.cdf(d2)
    elif option_type == "Put":
        option_price = K * np.exp(-r * T) * norm.cdf(-d2) - S0 * norm.cdf(-d1)
    else:
        raise ValueError("Invalid option type. Please input 'Call' or 'Put'.")

    return option_price

# Parameters
S0 = 5648
K = 5600
T = 3 / 12
r = 0.04
sigma_monthly = 0.047
sigma_annual = sigma_monthly * np.sqrt(12)

# Calculate the Black-Scholes price
bs_call_price = black_scholes(S0, K, r, T, sigma_annual,'Call')
```

Listing 11: Question 2c

```
def price_european_call_binomial(S0, K, r, u, d, N, q_u=None, q_d=None):

    #R = r * delta_t
    R = (r*0.25)/N
    if q_u is None:
        q_u = (1 + R - d) / (u - d)
    if q_d is None:
        q_d = 1 - q_u

    # Initialize stock price tree
    stock_tree = np.zeros((N + 1, N + 1))

    # Fill the stock price tree
    for i in range(N + 1):
        for j in range(i + 1):
            stock_tree[j, i] = S0 * (u ** (i - j)) * (d ** j)

    # Initialize option price tree
    option_tree = np.zeros((N + 1, N + 1))

    # Calculate option payoff at maturity (t = N)
    for j in range(N + 1):
        option_tree[j, N] = max(stock_tree[j, N] - K, 0)  # Payoff at maturity

```

```
    # Backward induction to calculate option price at earlier nodes
    for i in range(N - 1, -1, -1):
        for j in range(i + 1):
            option_tree[j, i] = (q_u * option_tree[j, i + 1] + q_d * option_tree[j + 1, i + 1]) / (1 + R)

    # Return the option price at the root of the tree (at t = 0), along with stock_tree and option_tree
    option_price_at_t0 = option_tree[0, 0]
    return stock_tree, option_tree, option_price_at_t0


# Given parameters
S0 = 5648  # Initial stock price
K = 5600   # Strike price
r = 0.04   # Annual risk-free rate
T = 3 / 12  # 3 months = 0.25 years
sigma = 0.047 * np.sqrt(12)  # Annual volatility

# Calculate Black-Scholes option price
bs_option_price = black_scholes(S0, K, r, T, sigma, 'Call')

# Step sizes and binomial tree calculations
steps = [3, 10, 100, 1000, 10000]  # Number of steps to try
binomial_prices_call = []
for N in steps:
    #delta_t = T / N  # Adjust delta_t for each N steps
    u, d = calculate_u_d(N)
    stock_tree, option_tree, price = price_european_call_binomial(S0, K, r, u, d, N)
    binomial_prices_call.append(price)

# Plot the results
plt.figure(figsize=(8, 4))
plt.plot(steps, binomial_prices_call, label="Binomial Tree Prices", marker='o')
plt.axhline(y=bs_option_price, color='r', linestyle='--', label="Black-Scholes Price")
plt.xscale('log')  # Log scale for better visualization of convergence
plt.xlabel('Number of Steps (log scale)')
plt.ylabel('Option Price')
plt.title('Convergence of Binomial Tree to Black-Scholes Price - Call Option')
plt.legend()
plt.grid(True)
plt.show()
```

Listing 12: Question 2d

```
def log_comb(n, k):
    """Compute the logarithm of the binomial coefficient."""
```

```
    return math.log(math.factorial(n)) - (math.log(math.factorial(k)) + math.log(math.factorial(n - k)))

def final_probabilities(S0, K, r, u, d, N, q_u=None, q_d=None):
    #R = r * delta_t
    R = (r*0.25)/N
    if q_u is None:
        q_u = (1 + R - d) / (u - d)
    if q_d is None:
        q_d = 1 - q_u

    # Initialize stock price tree
    stock_tree = np.zeros((N + 1, N + 1))

    # Fill the stock price tree
    for i in range(N + 1):
        for j in range(i + 1):
            stock_tree[j, i] = S0 * (u ** (i - j)) * (d ** j)

    # Binomial probabilities for each node at maturity using log-space to prevent overflow
    probabilities = np.zeros(N + 1)
    for j in range(N + 1):
        log_prob = log_comb(N, j) + (N - j) * math.log(q_u) + j * math.log(q_d)
        probabilities[j] = math.exp(log_prob)

    # Normalize the probabilities to ensure they sum to 1 (handling numerical precision)
    probabilities /= np.sum(probabilities)

    return stock_tree[:, -1], probabilities

def plot_probabilities(stock_prices, probabilities_risk_neutral, probabilities_real_world):
    plt.figure(figsize=(10,6))
    plt.plot(stock_prices, probabilities_risk_neutral, color='blue', label='Risk-Neutral')
    plt.plot(stock_prices, probabilities_real_world, color='red', label='Real-World')
    plt.xlabel("Stock Price at Maturity")
    plt.ylabel("Probability")
    plt.title("Probability Distribution at Maturity")
    plt.xlim(3000, 8000)
    plt.legend()
    plt.show()

# Parameters
S0 = 5648
K = 5600
r = 0.04
N = 10000  # Number of steps
```

```
u, d = calculate_u_d(N)


# Real-world probabilities can differ from risk-neutral, so assume a different q_u_real if needed
q_u_real = 0.56  # Real-world up probability
q_d_real = 1 - q_u_real

# Calculate for risk-neutral measure
stock_prices, probabilities_risk_neutral = final_probabilities(S0, K, r, u, d, N)

# Calculate for real-world measure
_, probabilities_real_world = final_probabilities(S0, K, r, u, d, N, q_u_real, q_d_real)

# Plot both probability distributions in one graph
plot_probabilities(stock_prices, probabilities_risk_neutral, probabilities_real_world)
```

Listing 13: Question 2e

```
def price_european_put_binomial(S0, K, r, u, d, N, q_u=None, q_d=None):
    # Adjust the rate R as in the call option function
    R = (r * 0.25) / N  # Discrete rate per step, adjusted for N steps
    if q_u is None:  # Risk-neutral probability for upward move
        q_u = (1 + R - d) / (u - d)
    if q_d is None:  # Risk-neutral probability for downward move
        q_d = 1 - q_u

    # Initialize stock price tree
    stock_tree = np.zeros((N + 1, N + 1))

    # Fill the stock price tree
    for i in range(N + 1):
        for j in range(i + 1):
            stock_tree[j, i] = S0 * (u ** (i - j)) * (d ** j)

    # Initialize option price tree
    option_tree = np.zeros((N + 1, N + 1))

    # Calculate put option payoff at maturity (t = N)
    for j in range(N + 1):
        option_tree[j, N] = max(K - stock_tree[j, N], 0)  # Put option payoff at maturity

    # Backward induction to calculate option price at earlier nodes
    for i in range(N - 1, -1, -1):
        for j in range(i + 1):
            option_tree[j, i] = (q_u * option_tree[j, i + 1] + q_d * option_tree[j + 1, i + 1]) / (1 + R)
```

```
28
29      # Return the option price at the root of the tree (at t = 0),
        along with stock_tree and option_tree
30      option_price_at_t0 = option_tree[0, 0]
31      return stock_tree, option_tree, option_price_at_t0
32
33
34  # Example usage
35  # Given parameters
36  S0 = 5648  # Initial stock price
37  K = 5600   # Strike price
38  r = 0.04   # Annual risk-free rate
39  T = 3 / 12  # 3 months = 0.25 years
40  sigma = 0.047 * np.sqrt(12)  # Annual volatility
41
42  steps =  [3, 10, 100, 1000, 10000]  # Number of steps to try
43
44  # Calculate Black-Scholes option price
45  bs_option_price = black_scholes(S0, K, r, T, sigma, "Put")
46
47  # Calculate binomial tree prices for different step counts
48  binomial_prices_put = []
49  for N in steps:
50      u, d = calculate_u_d(N)
51      stock_tree, option_tree, price = price_european_put_binomial(S0
        , K, r, u, d, N)
52      binomial_prices_put.append(price)
53
54  # Plot the results
55  plt.figure(figsize=(6, 6))
56  plt.plot(steps, binomial_prices_put, label="Binomial Tree Prices",
        marker='o')
57  plt.axhline(y=bs_option_price, color='r', linestyle='--', label="
        Black-Scholes Price")
58  plt.xscale('log')  # Log scale for better visualization of
        convergence
59  plt.xlabel('Number of Steps (log scale)')
60  plt.ylabel('Option Price')
61  plt.title('Convergence of Binomial Tree to Black-Scholes Price -
        Put Option')
62  plt.legend()
63  plt.grid(True)
64  plt.show()
65
66  # Given values
67  S0 = 5648  # Current stock price
68  K = 5600   # Strike price
69  r = 0.04   # Risk-free interest rate
70  T = 3 / 12 # Time to maturity (3 months = 0.25 years)
71  n = 12     # Compounding frequency (monthly compounding)
72
73  # Call and put prices (assumed to be American options, though put-
```

```
    call parity applies to European options)
C = np.round(binomial_prices_call[0],2) # Call option price
P = np.round(binomial_prices_put[0],2)# Put option price

# Put-Call Parity: Left-hand side (C - P)
put_call_parity_lhs = np.round(C - P,2)

# Right-hand side (S0 - K * (1 + r/n)^(-nT)) using discrete
    compounding
put_call_parity_rhs = np.round(S0 - K * (1 + r / n) ** (-n * T),2)

# Print the results
print(f"Left-hand side (C - P): {put_call_parity_lhs}")
print(f"Right-hand side (S0 - K * (1 + r/n)^(-nT)): {
    put_call_parity_rhs}")

# Check if they are approximately equal
if np.isclose(put_call_parity_lhs, put_call_parity_rhs):
    print("Put-Call Parity holds")
else:
    print("Put-Call Parity does not hold")
```

Listing 14: Question 2f

```
def price_american_put_binomial(S0, K, r, u, d, N, q_u=None, q_d=
    None):
    # Risk-neutral probabilities using discrete compounding
    R = (r * 0.25) / N  # Discrete rate per step
    if q_u is None:  # Risk-neutral probability for upward move
        q_u = (1 + R - d) / (u - d)
    if q_d is None:  # Risk-neutral probability for downward move
        q_d = 1 - q_u

    # Initialize stock price tree
    stock_tree = np.zeros((N + 1, N + 1))

    # Fill the stock price tree
    for i in range(N + 1):
        for j in range(i + 1):
            stock_tree[j, i] = S0 * (u ** (i - j)) * (d ** j)

    # Initialize option price tree
    option_tree = np.zeros((N + 1, N + 1))

    # Calculate put option payoff at maturity (t = N)
    for j in range(N + 1):
        option_tree[j, N] = max(K - stock_tree[j, N], 0)  # Put
    option payoff at maturity

    # Backward induction to calculate option price at earlier nodes
    , allowing for early exercise
    for i in range(N - 1, -1, -1):
        for j in range(i + 1):
```

```
            # Expected value from holding the option (as in the European case)
            hold_value = (q_u * option_tree[j, i + 1] + q_d * option_tree[j + 1, i + 1]) / (1 + R)
            # Payoff from early exercise
            exercise_value = max(K - stock_tree[j, i], 0)
            # Take the maximum of holding the option vs early exercise
            option_tree[j, i] = max(hold_value, exercise_value)

    # Return the option price at the root of the tree (at t = 0), along with stock_tree and option_tree
    option_price_at_t0 = option_tree[0, 0]
    return stock_tree, option_tree, option_price_at_t0

def price_american_call_binomial(S0, K, r, u, d, N, q_u=None, q_d=None):
    # Risk-neutral probabilities using discrete compounding
    R = (r * 0.25) / N  # Discrete rate per step
    if q_u is None:  # Risk-neutral probability for upward move
        q_u = (1 + R - d) / (u - d)
    if q_d is None:  # Risk-neutral probability for downward move
        q_d = 1 - q_u

    # Initialize stock price tree
    stock_tree = np.zeros((N + 1, N + 1))

    # Fill the stock price tree
    for i in range(N + 1):
        for j in range(i + 1):
            stock_tree[j, i] = S0 * (u ** (i - j)) * (d ** j)

    # Initialize option price tree
    option_tree = np.zeros((N + 1, N + 1))

    # Calculate call option payoff at maturity (t = N)
    for j in range(N + 1):
        option_tree[j, N] = max(stock_tree[j, N] - K, 0)  # Call option payoff at maturity

    # Backward induction to calculate option price at earlier nodes, allowing for early exercise
    for i in range(N - 1, -1, -1):
        for j in range(i + 1):
            # Expected value from holding the option (as in the European case)
            hold_value = (q_u * option_tree[j, i + 1] + q_d * option_tree[j + 1, i + 1]) / (1 + R)
            # Payoff from early exercise
            exercise_value = max(stock_tree[j, i] - K, 0)
            # Take the maximum of holding the option vs early exercise
```

```
69                option_tree[j, i] = max(hold_value, exercise_value)
70
71        # Return the option price at the root of the tree (at t = 0),
          along with stock_tree and option_tree
72        option_price_at_t0 = option_tree[0, 0]
73        return stock_tree, option_tree, option_price_at_t0
```

Listing 15: Question 2g