GEORGIA INSTITUTE OF TECHNOLOGY

Title: ISyE6785 Interim Project 1

Author(s): Weifeng Lyu
Instructor: Dr. Shijie Deng

TABLE OF CONTENTS

1. Introduction

1.1 Problem 3
1.2 Computer Configuration 3
1.3 Workflow Diagram 4

2. Technology Review

2.1 Algorithm Review 5
2.2 Algorithm Parameter 9

3. Project Architecture

3.1 Python Implementation 9
3.2 Result of Sample Data and Discussion. 25

4. Improvement 34

5. Conclusion 35

6. Reference 36

Appendices

7. Code and GitHub 36

1. Introduction

1.1 Problem

ISyE 6785 interim-project 1(Due on 6/11/2018)

Note: For all the computations, please report the configuration of your computer (e.g. Intel Core i-7 2.5GHz, 8GBRAM) and the computing CPU time in seconds. Barrier Options Pricing

1.(10 points) Implement a trinomial lattice to price a down-and-in call option with current S =100, strike K=100, r=10%, σ=0.3, time to maturity T = 0.6. Use barriers 95, 99.5 and 99.9. Record the accuracy and computational time.

2.(20 points) Implement the AMM for barrier options and replicate Table 3 on page 337 of the AMM paper.

3.(10 bonus points) Compute the delta and gamma of the barrier options using both the regular trinomial lattice and the AMM; report the errors with respect to the closed-form values; comment on the performance of the AMM for computing Greeks of the barrier options.

1.2 Computer Configuration

Manufacturer: Dell
Model: Inspiron 7559 Signature Edition
Processor: Intel® Core™ i7-6700HQ CPU @ 2.60GHz 2.60GHz
Installed Memory (RAM): 8.00GB (7.88 usable)
System Type: 64-bit Operating System, x64-based processor

1.3 Workflow Diagram

A. Trinomial Tree with down-and-in or down-and-out option

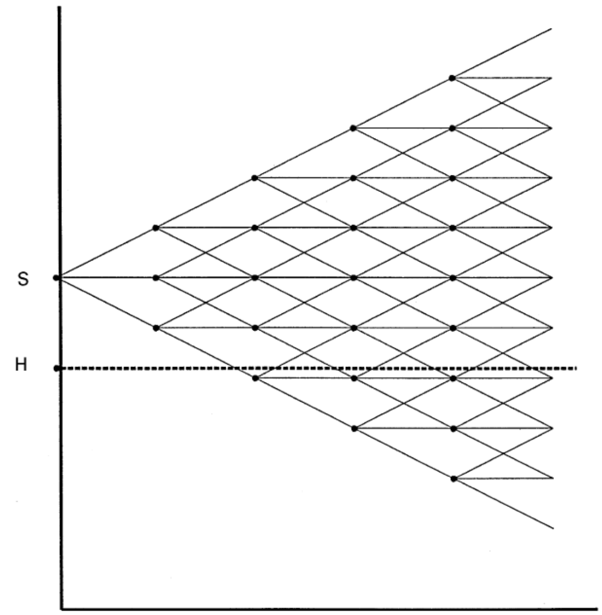


Fig. 1. A trinomial model for a barrier option. The barrier, H, lies just slightly less than two price steps below the current asset price, S. The option is knocked out if the price falls two steps below the initial price at any time prior to expiration.

B. Adaptive Mesh Model

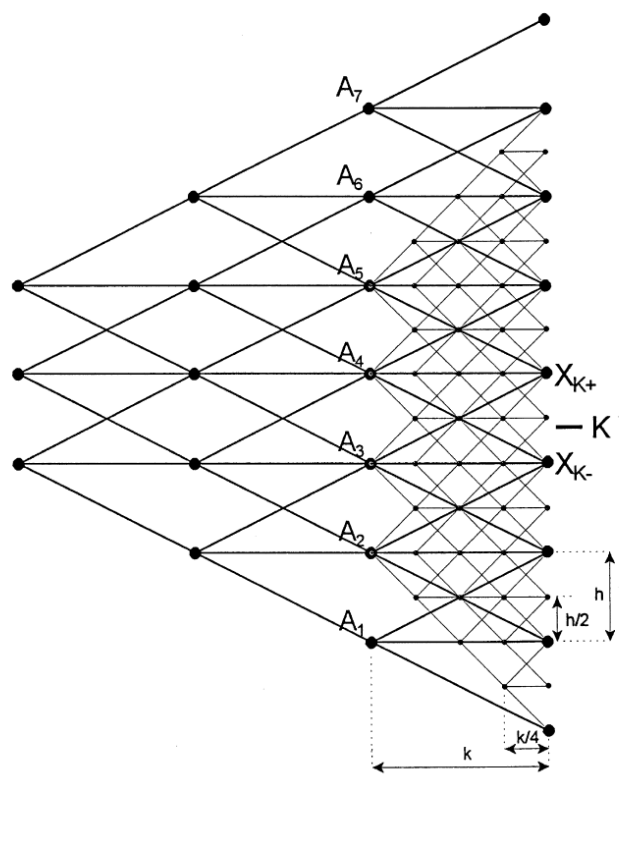


Fig. 2. An adaptive mesh model for the European put. This Trinomial tree shows the section of the pricing lattice in the immediate vicinity of the strike price in the last few periods before expiration. The coarse lattice, with price and time steps h and k, is represented by heavy lines. The "ne mesh, with price and time steps h/2 and k/4, is represented by light lines. The "ne mesh covers all ¹!k coarse nodes from which there are both "ne-mesh paths that end up in-the-money and "ne-mesh paths that end up out of-the-money. K is the strike price, and XK~ and XK` are the two date ¹ coarse-mesh asset prices that bracket the strike price.

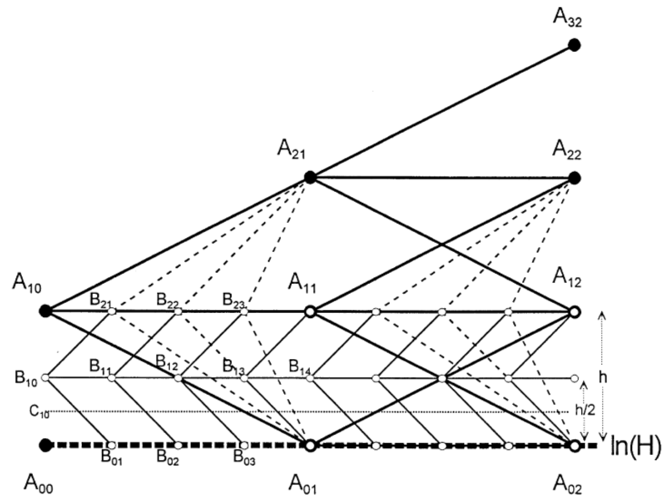


Fig. 3. Adaptive mesh model for a barrier option. The heavy lines indicate the coarse-mesh lattice, whose nodes are labelled Aij, with i indicating the number of coarse-mesh price steps above the barrier and j the number of the coarse time step. The "ne-mesh nodes are labelled Bij. The barrier price is ln(H). To compute the option value at the initial asset price of B10, "rst compute option values at all the A nodes. Second, use the coarse-mesh lattice to compute the option values at ln(H) and ln(H)#h for time intervals of k/4. Finally, calculate the remaining "ne-mesh nodes for the price ln(H)#h/2 at time intervals of k/4. The dotted lines indicate that nodes A01, A11, and A21, are used to calculate option values at B21, B22 and B23. Similarly, the light solid lines indicate, for example, that the option value at node B10 is based on nodes B01, B11, and B21. The "ne dotted line indicates where the middle nodes of the next level of mesh would be placed to compute the option value at the initial asset price C10.

2. Technology Review

2.1 Algorithm Review

Trinomial Tree Algorithm:

Essential Input:

1. self.S0 = S0 (Current Price)
2. self.K = K (Strike Price)
3. self.rf = rf (Risk-free Interest Rate)
4. self.divR = divR (Dividend Ratio)
5. self.sigma = sigma (Volatility)
6. self.tyears = tyears (Total Time Period Unit)

Optional Input:

1. M (Number of periods in Time Period Unit)
2. H (Barrier Option)

Derived Formula :

1. Alpha

$$\alpha = r - q - \sigma^2/2,$$

2. Rick-Neutral Probability

$$1 = p_{\mathrm{u}} + p_{\mathrm{m}} + p_{\mathrm{d}},$$

$$\mathrm{E}[X(t+k) - X(t)] = 0 = p_{\mathrm{u}}h + p_{\mathrm{m}}0 + p_{\mathrm{d}}(-h),$$

$$\mathrm{E}[(X(t+k) - X(t))^2] = \sigma^2 k = p_{\mathrm{u}}h^2 + p_{\mathrm{m}}0 + p_{\mathrm{d}}h^2,$$

$$\mathrm{E}[(X(t+k) - X(t))^4] = 3\sigma^4 k^2 = p_{\mathrm{u}}h^4 + p_{\mathrm{m}}0 + p_{\mathrm{d}}h^4.$$

$$p_{\mathrm{u}} = 1/6, \quad p_{\mathrm{m}} = 2/3, \quad p_{\mathrm{d}} = 1/6, \quad h = \sigma\sqrt{3k}.$$

3. Time Step (DeltaT)

$$k = T/N.$$

4. Difference Step between Stock Price

h=np.sqrt(3.0 * deltaT) * self.sigma

5. Next Price and Final Price

$$C(X,t) = \mathrm{e}^{-rk}(p_{\mathrm{u}}(h,k)C(X+h,t+k) + p_{\mathrm{m}}(h,k)C(X,t+k) + p_{\mathrm{d}}(h,k)C(X-h,t+k)).$$

6. Black-Scholes call formula

$$C_{\mathrm{DO}}(S, K, T, r, \sigma, H) = C_{\mathrm{BS}}(S, K, T, r, \sigma) - (H/S)^{2(r-(\sigma^2/2))}$$
$$\times C_{\mathrm{BS}}(H^2/S, K, T, r, \sigma),$$

7. Delta and Gamma Computation

$$\Delta = \frac{\partial C}{\partial S} = \frac{\partial C}{\partial \ln(S)} \frac{1}{S} \approx \frac{C(X_0 + \varepsilon) - C(X_0 - \varepsilon)}{2\varepsilon} \frac{1}{S},$$

$$\Gamma = \frac{\partial \Delta}{\partial S} = \frac{\partial^2 C}{\partial S^2} = \frac{\partial}{\partial S}\left(\frac{\partial C}{\partial \ln(S)} \frac{1}{S}\right) = \left(\frac{\partial^2 C}{\partial (\ln(S))^2} - \frac{\partial C}{\partial \ln(S)}\right)\frac{1}{S^2}$$
$$\approx \left(\frac{C(X_0 + \varepsilon) + C(X_0 - \varepsilon) - 2C(X_0)}{\varepsilon^2} - \frac{C(X_0 + \varepsilon) - C(X_0 - \varepsilon)}{2\varepsilon}\right)\frac{1}{S^2}.$$

Adaptive Mesh Model Algorithm:

Essential Input:

1. self.S0 = S0 (Current Price)
2. self.K = K (Strike Price)
3. self.rf = rf (Risk-free Interest Rate)
4. self.divR = divR (Dividend Ratio)
5. self.sigma = sigma (Volatility)
6. self.tyears = tyears (Total Time Period Unit)

Optional Input:

1. N (Layers of Adaptive Mesh)
2. H (Barrier Option)

Derived Formula

Derived Formula :

1. Alpha

$$\alpha = r - q - \sigma^2/2,$$

2. Rick-Neutral Probability

The probability of stock price upward:
qU = 1 / 2 * (self.sigma ** 2 * k / h ** 2 + alpha ** 2 * k ** 2 / h ** 2 + alpha * k / h)
The probability of stock price downward:
qD = 1 / 2 * (self.sigma ** 2 * k / h ** 2 + alpha ** 2 * k ** 2 / h ** 2 - alpha * k / h)
The probability of stock price unchanged:
qM = 1 - self.qU(alpha, k,h) - self.qD(alpha, k,h)

3. Time Step (DeltaT)

$$k = T/\mathrm{int}[(\lambda\sigma^2/h^2)T].$$

4. Difference Step between Stock Price

$$h = 2^M(\ln(S_0) - \ln(H)).$$

5. Next Price and Final Price

$$C(X, t) = e^{-rk}(p_u(h, k)C(X + h, t + k) + p_m(h, k)C(X, t + k) + p_d(h, k)C(X - h, t + k)).$$

$$C(B_{23}) = e^{-rk/4}(p_u(h, k/4)C(A_{21}) + p_m(h, k/4)C(A_{11}) + p_d(h, k/4)C(A_{01})).$$

$$(B_{10}) = e^{-rk/4}[p_u(h/2, k/4)C(B_{21}) + p_m(h/2, k/4)C(B_{11}) + p_d(h/2, k/4)C(B_{01})].$$

6. Black-Scholes call formula

$$C_{DO}(S, K, T, r, \sigma, H) = C_{BS}(S, K, T, r, \sigma) - (H/S)^{2(r-(\sigma^2/2))} \times C_{BS}(H^2/S, K, T, r, \sigma),$$

7. Delta and Gamma Computation

$$\Delta = \frac{\partial C}{\partial S} = \frac{\partial C}{\partial \ln(S)} \frac{1}{S} \approx \frac{C(X_0 + \varepsilon) - C(X_0 - \varepsilon)}{2\varepsilon} \frac{1}{S},$$

$$\Gamma = \frac{\partial \Delta}{\partial S} = \frac{\partial^2 C}{\partial S^2} = \frac{\partial}{\partial S}\left(\frac{\partial C}{\partial \ln(S)} \frac{1}{S}\right) = \left(\frac{\partial^2 C}{\partial (\ln(S))^2} - \frac{\partial C}{\partial \ln(S)}\right)\frac{1}{S^2}$$

$$\approx \left(\frac{C(X_0 + \varepsilon) + C(X_0 - \varepsilon) - 2C(X_0)}{\varepsilon^2} - \frac{C(X_0 + \varepsilon) - C(X_0 - \varepsilon)}{2\varepsilon}\right)\frac{1}{S^2}.$$

2.2 Algorithm Parameter

1. Trinomial Tree Algorithm (down-and-in):

S =100, strike K=100, r=10%, σ=0.3, time to maturity T = 0.6. Use barriers 95, 99.5 and 99.9, N = 5,15

2. Adaptive Mesh Model and Trinomial Tree Algorithm (down-and-out):

S =92, 91, 90.5, 90.25, 90.125, strike K=100, r=10%, σ=0.25, time to maturity T = 1. H (Barrier) = 90, Number of Steps = 388, 1535, 6108, 24367, 97335, M = 0,1,2,3,4

3. Delta and Gamma Computation:

Trinomial perturbation, e=0.001
Trinomial perturbation, e=0.01
N = 25, 100, 250, 1000

3. Project Architecture

3.1 Python Implementation

Source Code:

```
import itertools
import numpy as np
import scipy.stats
import math
import time
import matplotlib.pyplot as plt


class CallOption(object):
    def __init__(self, S0, K, rf, divR, sigma, tyears):
        self.S0 = S0
        self.K = K
        self.rf = rf
        self.divR = divR
        self.sigma = sigma
        self.tyears = tyears

    def BinomialTreeEuroCallPrice(self, N=10):
        deltaT = self.tyears / float(N)
        # create the size of up-move and down-move
        u = np.exp(self.sigma * np.sqrt(deltaT))
        d = 1.0 / u

        # Let fs store the value of the option
        fs = [0.0 for j in range(N + 1)]
        fs_pre = [0.0 for j in range(N + 1)]

        # Compute the risk-neutral probability of moving up: q
        a = np.exp(self.rf * deltaT)
        q = (a - d) / (u - d)

        # Compute the value of the European Call option at
maturity time tyears:
        for j in range(N + 1):
            fs[j] = max(self.S0 * np.power(u, j) * np.power(d, N
- j) - self.K, 0)
        fs_pre = fs
        #print('Call option value at maturity is: ', fs)

        # Apply the recursive pricing equation to get the option
value in periods: N-1, N-2, ... , 0
        for t in range(N - 1, -1, -1):
            fs = [0.0 for j in range(t + 1)]  # initialize the
value of options at all nodes in period t to 0.0
            for j in range(t + 1):
                # The following line is the recursive option
pricing equation:
```

```
                fs[j] = np.exp(-self.rf * deltaT) * (q *
fs_pre[j + 1] + (1 - q) * fs_pre[j])
            fs_pre = fs
        return fs[0]

    def BS_d1(self, S0 = 100):
        return (np.log(S0 / self.K) + (self.rf + self.sigma ** 2
/ 2.0) * self.tyears) / (self.sigma * np.sqrt(self.tyears))

    def BS_d2(self, S0 = 100):
        return self.BS_d1(S0) - self.sigma *
np.sqrt(self.tyears)

    def BS_CallPrice(self, S0 = 100):
        return S0 * scipy.stats.norm.cdf(self.BS_d1(S0)) -
self.K * np.exp(-self.rf * self.tyears) *
scipy.stats.norm.cdf(self.BS_d2(S0))

    def BS_CallPriceDI(self, S0 = 100, H = 90):
        return np.power(H/S0,2*self.rf-self.sigma**2) * H**2/S0
* scipy.stats.norm.cdf((np.log(H**2/S0 / self.K) + (self.rf +
self.sigma ** 2 / 2.0) * self.tyears) / (self.sigma *
np.sqrt(self.tyears))) - self.K * np.exp(-self.rf * self.tyears)
* scipy.stats.norm.cdf((np.log(H**2/S0 / self.K) + (self.rf +
self.sigma ** 2 / 2.0) * self.tyears) / (self.sigma *
np.sqrt(self.tyears)) - self.sigma * np.sqrt(self.tyears))

    def BS_CallPriceDO(self, S0 = 100, H = 90):
        return self.BS_CallPrice(S0)- self.BS_CallPriceDI(S0, H)

    def TrinomialTreeEuroCallPriceDI(self, S0=100, N=10, H=100):
        deltaT = self.tyears / float(N)
        X0 = np.log(H**2/S0)
        alpha = self.rf - self.divR - np.power(self.sigma, 2.0)
/ 2.0
        h = np.sqrt(3.0 * deltaT) * self.sigma

        # Risk-neutral probabilities:
        qU = 1.0 / 6.0
        qM = 2.0 / 3.0
        qD = 1.0 / 6.0

        # Initialize the stock prices and option values at
maturity with 0.0
        stk = [0.0 for i in range(2 * N + 1)]
        fs = [0.0 for i in range(2 * N + 1)]
        fs_pre = [0.0 for i in range(2 * N + 1)]
```

```
        nd_idx = N
        pre_price = X0 - float(N + 1) * h

        # Initialize the stock price movement
        move = [1,0,-1]

        # Compute the stock prices and option values at maturity
        for indices in itertools.product(move, repeat=N):
            cur_price = X0
            time_counter = 0
            down_and_in=False
            for i in indices:
                cur_price = cur_price + move[i] * h
                time_counter = time_counter+1
                # Only Compute the stock price if the price hits
the barrier
                if cur_price < np.log(H):
                    down_and_in=True
                if time_counter == N and down_and_in and
(cur_price - pre_price) > h / 1000.0:
                    stk[nd_idx] = np.exp(cur_price + alpha *
self.tyears)
                    fs_pre[nd_idx] = max(stk[nd_idx] - self.K,
0)
                    pre_price = cur_price
                    nd_idx = nd_idx + 1

        return self.ComputeTrinomialTree(N, deltaT, qU, qM, qD,
fs_pre)

    def TrinomialTreeEuroCallPriceDO(self, S=100, N=10, H=100):
        # Use Regular call option value minus Down-and-in call
option value to get down-and-out call option value
        return self.TrinomialTreeEuroCallPrice(S,N)-
self.TrinomialTreeEuroCallPriceDI(S,N,H)

    def TrinomialTreeDelta(self, S0=100, N=10, e=0.01):
        deltaT = self.tyears / float(N)
        X0 = np.log(S0)
        alpha = self.rf - self.divR - np.power(self.sigma, 2.0)
/ 2.0
        h = np.sqrt(3.0 * deltaT) * self.sigma

        # Risk-neutral probabilities:
        qU = 1.0 / 6.0
        qM = 2.0 / 3.0
```

```
        qD = 1.0 / 6.0

        # Initialize the stock prices and option values at
maturity with 0.0
        stk = [0.0 for i in range(2 * N + 1)]
        fs_pre = [0.0 for i in range(2 * N + 1)]

        nd_idx = 0
        pre_price = X0 - float(N + 1) * h
        # Compute the stock prices and option values at maturity
        for i in range(N + 1):
            for j in range(N + 1):
                k = max(N - i - j, 0)
                cur_price = X0 + (i - k) * h
                if (cur_price - pre_price) > h / 1000.0:
                    stk[nd_idx] = np.exp(cur_price + alpha *
self.tyears)
                    # Compute the option value at the cur_price
level
                    fs_pre[nd_idx] = max(stk[nd_idx] - self.K,
0)
                    pre_price = cur_price
                    nd_idx = nd_idx + 1
        #print('Call option value at maturity is: ', fs_pre)

        fs_pre_a = [x + e for x in fs_pre]
        fs_pre_s = [x - e for x in fs_pre]

        return (self.ComputeTrinomialTree(N, deltaT, qU, qM, qD,
fs_pre_a) - self.ComputeTrinomialTree(N, deltaT, qU, qM, qD,
fs_pre_s))/(2*e)/self.S0

    def TrinomialTreeGamma(self, S0=100, N=10, e=0.01):
        deltaT = self.tyears / float(N)
        X0 = np.log(S0)
        alpha = self.rf - self.divR - np.power(self.sigma, 2.0)
/ 2.0
        h = np.sqrt(3.0 * deltaT) * self.sigma

        # Risk-neutral probabilities:
        qU = 1.0 / 6.0
        qM = 2.0 / 3.0
        qD = 1.0 / 6.0

        # Initialize the stock prices and option values at
maturity with 0.0
        stk = [0.0 for i in range(2 * N + 1)]
```

```
        fs_pre = [0.0 for i in range(2 * N + 1)]

        nd_idx = 0
        pre_price = X0 - float(N + 1) * h
        # Compute the stock prices and option values at maturity
        for i in range(N + 1):
            for j in range(N + 1):
                k = max(N - i - j, 0)
                cur_price = X0 + (i - k) * h
                if (cur_price - pre_price) > h / 1000.0:
                    stk[nd_idx] = np.exp(cur_price + alpha *
self.tyears)
                    # Compute the option value at the cur_price
level
                    fs_pre[nd_idx] = max(stk[nd_idx] - self.K,
0)

                    pre_price = cur_price
                    nd_idx = nd_idx + 1
        #print('Call option value at maturity is: ', fs_pre)

        # Compute Trinomial perturbation
        fs_pre_a = [x + e for x in fs_pre]
        fs_pre_s = [x - e for x in fs_pre]

        return ((self.ComputeTrinomialTree(N, deltaT, qU, qM,
qD, fs_pre_a) - self.ComputeTrinomialTree(N, deltaT, qU, qM, qD,
fs_pre_s) + 2 * self.ComputeTrinomialTree(N, deltaT, qU, qM, qD,
fs_pre)) / (e ** 2) - (self.ComputeTrinomialTree(N, deltaT, qU,
qM, qD, fs_pre_a) - self.ComputeTrinomialTree(N, deltaT, qU, qM,
qD, fs_pre_s)) / (2 * e)) / self.S0 ** 2

    def TrinomialTreeEuroCallPrice(self, S0=100, N=10):
        deltaT = self.tyears / float(N)
        X0 = np.log(S0)
        alpha = self.rf - self.divR - np.power(self.sigma, 2.0)
/ 2.0
        h = np.sqrt(3.0 * deltaT) * self.sigma

        # Risk-neutral probabilities:
        qU = 1.0 / 6.0
        qM = 2.0 / 3.0
        qD = 1.0 / 6.0

        # Initialize the stock prices and option values at
maturity with 0.0
        stk = [0.0 for i in range(2 * N + 1)]
        fs_pre = [0.0 for i in range(2 * N + 1)]
```

```
        nd_idx = 0
        pre_price = X0 - float(N + 1) * h
        # Compute the stock prices and option values at maturity
        for i in range(N + 1):
            for j in range(N + 1):
                k = max(N - i - j, 0)
                cur_price = X0 + (i - k) * h
                if (cur_price - pre_price) > h / 1000.0:
                    stk[nd_idx] = np.exp(cur_price + alpha *
self.tyears)
                    # Compute the option value at the cur_price
level
                    fs_pre[nd_idx] = max(stk[nd_idx] - self.K,
0)
                    pre_price = cur_price
                    nd_idx = nd_idx + 1
        #print('Call option value at maturity is: ', fs_pre)

        return self.ComputeTrinomialTree(N, deltaT, qU, qM, qD,
fs_pre)

    def ComputeTrinomialTree(self, N, deltaT, qU, qM, qD,
fs_pre):
        # Backward recursion for computing option prices in time
periods N-1, N-2, ... , 0
        for t in range(N - 1, -1, -1):
            fs = []
            for i in range(2 * t + 1):
                cur_optP = np.exp(-self.rf * deltaT) * (qU *
fs_pre[i + 2] + qM * fs_pre[i + 1] + qD * fs_pre[i])
                fs.append(cur_optP)
            fs_pre = fs
        return fs[0]

    def AdaptiveMeshEuroCallPrice(self, S0=100, H=100, M=1):
        X0 = np.log(S0)
        alpha = self.rf - self.divR - np.power(self.sigma, 2.0)
/ 2.0
        h = 2 ** M * (X0 - np.log(H))
        k = self.tyears / math.floor((3.0 * sigma ** 2 / h **
2)*self.tyears)
        N = int(self.tyears / k)

        # Initialize the stock prices and option values at
maturity with 0.0
        stk = [0.0 for i in range(2*N+1)]
```

```
        a_mesh = [0.0 for i in range(2*N+1)]

        nd_idx = 0
        pre_price = X0 - float(N + 1) * h

        # Compute the stock prices and option values at maturity
        for i in range(N + 1):
            for j in range(N + 1):
                l = max(N - i - j, 0)
                cur_price = X0 + (i - l) * h
                if (cur_price - pre_price) > h / 1000.0:
                    stk[nd_idx] = np.exp(cur_price + alpha *
self.tyears)
                    # Compute the option value at the cur_price
level
                    if stk[nd_idx] > H :
                        a_mesh[nd_idx] = max(stk[nd_idx] -
self.K, 0)
                        pre_price = cur_price
                        nd_idx = nd_idx + 1
        #print('Call option value at maturity is: ', a_mesh)

        return self.ComputeAMM(H, h, k, N, a_mesh, alpha, M)

    def AdaptiveMeshDelta(self, S0=100, M=1, e=0.01):
        X0 = np.log(S0)
        alpha = self.rf - self.divR - np.power(self.sigma, 2.0)
/ 2.0
        h = 2 ** M * (X0 - np.log(H))
        k = self.tyears / math.floor((3.0 * sigma ** 2 / h **
2)*self.tyears)
        N = int(self.tyears / k)

        # Initialize the stock prices and option values at
maturity with 0.0
        stk = [0.0 for i in range(2*N+1)]
        a_mesh = [0.0 for i in range(2*N+1)]

        nd_idx = 0
        pre_price = X0 - float(N + 1) * h

        # Compute the stock prices and option values at maturity
        for i in range(N + 1):
            for j in range(N + 1):
                l = max(N - i - j, 0)
                cur_price = X0 + (i - l) * h
                if (cur_price - pre_price) > h / 1000.0:
```

```
                        stk[nd_idx] = np.exp(cur_price + alpha *
self.tyears)
                        # Compute the option value at the cur_price
level
                        if stk[nd_idx] > H :
                            a_mesh[nd_idx] = max(stk[nd_idx] -
self.K, 0)
                            pre_price = cur_price
                            nd_idx = nd_idx + 1
        #print('Call option value at maturity is: ', a_mesh)

        a_mesh_a = [x+e for x in a_mesh]
        a_mesh_s = [x-e for x in a_mesh]

        return (self.ComputeAMM(H, h, k, N, a_mesh_a, alpha, M)-
self.ComputeAMM(H, h, k, N, a_mesh_s, alpha, M))/(2*e)/self.S0

    def AdaptiveMeshGamma(self, S0=100, M=1, e=0.01):
        X0 = np.log(S0)
        alpha = self.rf - self.divR - np.power(self.sigma, 2.0)
/ 2.0
        h = 2 ** M * (X0 - np.log(H))
        k = self.tyears / math.floor((3.0 * sigma ** 2 / h **
2)*self.tyears)
        N = int(self.tyears / k)

        # Initialize the stock prices and option values at
maturity with 0.0
        stk = [0.0 for i in range(2*N+1)]
        a_mesh = [0.0 for i in range(2*N+1)]

        nd_idx = 0
        pre_price = X0 - float(N + 1) * h
        # Compute the stock prices and option values at maturity
        for i in range(N + 1):
            for j in range(N + 1):
                l = max(N - i - j, 0)
                cur_price = X0 + (i - l) * h
                if (cur_price - pre_price) > h / 1000.0:
                    stk[nd_idx] = np.exp(cur_price + alpha *
self.tyears)
                    # Compute the option value at the cur_price
level
                    if stk[nd_idx] > H :
                        a_mesh[nd_idx] = max(stk[nd_idx] -
self.K, 0)
                        pre_price = cur_price
```

```
                        nd_idx = nd_idx + 1
        #print('Call option value at maturity is: ', a_mesh)

        a_mesh_a = [x+e for x in a_mesh]
        a_mesh_s = [x-e for x in a_mesh]

        return ((self.ComputeAMM(H, h, k, N, a_mesh_a, alpha,
M)-self.ComputeAMM(H, h, k, N, a_mesh_s, alpha,
M)+2*self.ComputeAMM(H, h, k, N, a_mesh, alpha, M))/(e**2)-
(self.ComputeAMM(H, h, k, N, a_mesh_a, alpha, M)-
self.ComputeAMM(H, h, k, N, a_mesh_s, alpha,
M))/(2*e))/self.S0**2

    def ComputeAMM(self, H, h, k, N, a_mesh, alpha, M):

        # Initialize the mid mesh with log(H) + h / 2
        b_mesh_mid = np.log(H) + h / 2
        c_mesh_mid = np.log(H) + h / 4
        d_mesh_mid = np.log(H) + h / 8
        e_mesh_mid = np.log(H) + h / 16

        # Backward recursion for computing option prices in time
periods N-k, N-2k, ... , 0
        for t in range(N):
            pre_amesh = a_mesh
            for t_a in range(N):
                if t_a + 2 < N - t:
                    a_mesh[t_a + 1] = np.exp(-self.rf * k) * (
                                self.qU(alpha, k, h) *
pre_amesh[t_a + 2] + self.qM(alpha, k, h) * pre_amesh[
                            t_a + 1] + self.qD(alpha, k, h) *
pre_amesh[t_a])
            for t_a in range(4):
                h_a = h
                k_a = k * t_a / 4
                a_mesh_mid = (np.exp(-self.rf * k_a) * (
                            self.qU(alpha, k_a, h_a) *
pre_amesh[2] + self.qM(alpha, k_a, h_a) * pre_amesh[1] +
self.qD(
                        alpha, k_a, h_a) * pre_amesh[0]))
                if M > 0:
                    for t_b in range(4):
                        h_b = h / 2
                        k_b = k / 4
                        b_mesh_mid = (np.exp(-self.rf * k_b) * (
                                    self.qU(alpha, k_b, h_b) *
a_mesh_mid + self.qM(alpha, k_b,
```

```
h_b) * b_mesh_mid + self.qD(alpha,

k_b,

h_b) *
                                    pre_amesh[0]))
                        if M > 1:
                            for t_c in range(4):
                                h_c = h / 4
                                k_c = k / 16
                                c_mesh_mid = (np.exp(-self.rf *
k_c) * (
                                        self.qU(alpha, k_c,
h_c) * b_mesh_mid + self.qM(alpha, k_c,

h_c) * c_mesh_mid + self.qD(
                                        alpha, k_c, h_c) *
pre_amesh[0]))
                                if M > 2:
                                    for t_d in range(4):
                                        h_d = h / 8
                                        k_d = k / 32
                                        d_mesh_mid = (np.exp(-
self.rf * k_d) * (

self.qU(alpha, k_d, h_d) * c_mesh_mid + self.qM(alpha, k_d,

h_d) * d_mesh_mid + self.qD(
                                                alpha, k_d, h_d)
* pre_amesh[0]))
                                        if M > 3:
                                            for t_d in range(4):
                                                h_e = h / 16
                                                k_e = k / 64
                                                e_mesh_mid =
(np.exp(-self.rf * k_e) * (

self.qU(alpha, k_e, h_e) * d_mesh_mid + self.qM(alpha, k_e,

h_e) * e_mesh_mid + self.qD(
                                                        alpha,
k_e, h_e) * pre_amesh[0]))

        # Selectively return the final mesh value depending on
mesh level
        if M > 3:
```

```
            return e_mesh_mid
        elif M > 2:
            return d_mesh_mid
        elif M > 1:
            return c_mesh_mid
        elif M > 0:
            return b_mesh_mid
        else:
            return a_mesh_mid

    def qU(self, alpha, k, h):
        #Compute the risk-neutral probability upward
        return 1 / 2 * (self.sigma ** 2 * k / h ** 2 + alpha **
2 * k ** 2 / h ** 2 + alpha * k / h)

    def qD(self, alpha, k, h):
        # Compute the risk-neutral probability downward
        return 1 / 2 * (self.sigma ** 2 * k / h ** 2 + alpha **
2 * k ** 2 / h ** 2 - alpha * k / h)

    def qM(self, alpha, k, h):
        # Compute the risk-neutral probability unchanged
        return 1 - self.qU(alpha, k,h) - self.qD(alpha, k,h)

    def TTDI_timer(self, S, N, H):
        start = time.time()
        self.TrinomialTreeEuroCallPriceDI(S, N, H)
        end = time.time()
        return end-start

    def TTDO_timer(self, S, N, H):
        start = time.time()
        self.TrinomialTreeEuroCallPriceDO(S, N, H)
        end = time.time()
        return end-start

    def AMM_timer(self, S, H, M):
        start = time.time()
        self.AdaptiveMeshEuroCallPrice(S,H,M)
        end = time.time()
        return end-start

    def AMM_timer_Delta(self, H, M, e):
        start = time.time()
        self.AdaptiveMeshDelta(H, M, e)
        end = time.time()
        return end-start
```

```
    def AMM_timer_Gamma(self, H, M, e):
        start = time.time()
        self.AdaptiveMeshGamma(H, M, e)
        end = time.time()
        return end-start

    def TT_timer_Delta(self, S, N, e):
        start = time.time()
        self.TrinomialTreeDelta(S, N, e)
        end = time.time()
        return end - start

    def TT_timer_Gamma(self, S, N, e):
        start = time.time()
        self.TrinomialTreeGamma(S, N,e)
        end = time.time()
        return end - start

if __name__ == '__main__':

#1
    S0 = 100.0
    K = 100.0
    rf = 0.1
    divR = 0.0
    sigma = 0.3
    T = 0.6  # unit is in years

    n_periods = 10
    H = 99.9

    call_test = CallOption(S0, K, rf, divR, sigma, T)
    call_tri_di = call_test.TrinomialTreeEuroCallPriceDI(S0,
n_periods, H)
    call_tri_do = call_test.TrinomialTreeEuroCallPriceDO(S0,
n_periods, H)
    call_bs_di = call_test.BS_CallPriceDI(S0, H)
    call_bs_do = call_test.BS_CallPriceDO(S0, H)
    print('Trinomial Tree Call down-and-in option price is: ',
call_tri_di)
    print('Trinomial Tree Call down-and-out option price is: ',
call_tri_do)
    print('Black-Scholes Call down-and-in option price is: ',
call_bs_di)
    print('Black-Scholes Call down-and-out option price is: ',
call_bs_do)
```

```
    axis_n = np.arange(4, 12, 1)
    TTDI_vec = [call_test.TrinomialTreeEuroCallPriceDI(S0,n,H)
for n in axis_n]
    TTDO_vec = [call_test.TrinomialTreeEuroCallPriceDO(S0,n,H)
for n in axis_n]
    BSDI_vec = [call_test.BS_CallPriceDI(S0,H) for n in axis_n]
    BSDO_vec = [call_test.BS_CallPriceDO(S0,H) for n in axis_n]
    print('Trinomial Tree Call down-and-in option price from 4 -
12 periods is: ',TTDI_vec)
    print('Trinomial Tree Call down-and-out option price from 4
- 12 periods is: ',TTDO_vec)
    print('Black-Scholes Call down-and-in option price from 4 -
12 periods is: ', BSDI_vec)
    print('Black-Scholes Call down-and-out option price from 4 -
12 periods is: ', BSDO_vec)
    plt.plot(axis_n, TTDI_vec, 'r-', lw=2, label='TTDI')
    plt.plot(axis_n, TTDO_vec, 'c-', lw=2, label='TTDO')
    plt.plot(axis_n, BSDI_vec, 'g-', lw=2, label='BSDI')
    plt.plot(axis_n, BSDO_vec, 'b-', lw=2, label='BSDO')
    label = ['TTDI','TTDO','BSDI','BSDO']
    plt.xlabel("Number of Periods")
    plt.ylabel("Option Price")
    plt.title("European Call Option Price vs. Number of Periods
in a Lattice")
    plt.legend(label)
    plt.grid(True)
    plt.show()
    axis_h = np.array([95, 99, 99.9])
    TTDI_vec2 =
np.array([call_test.TrinomialTreeEuroCallPriceDI(S0, n_periods,
H) for H in axis_h])
    BSDI_vec2 = np.array([call_test.BS_CallPriceDI(S0,H) for H
in axis_h])
    plt.plot(axis_h, TTDI_vec2/BSDI_vec2, 'r-', lw=2)
    label = ['TTDI Accuracy']
    plt.xlabel("Barrier Option")
    plt.ylabel("Accuracy Percentage")
    plt.title("Price Accuracy Percentage vs. Barrier Option")
    plt.legend(label)
    plt.grid(True)
    plt.show()
    TTDI_vec3 = np.array([call_test.TTDI_timer(S0, n_periods, H)
for H in axis_h])
    plt.plot(axis_h, TTDI_vec3, 'b-', lw=2)
    label = ['TTDI Computation Time']
```

```
    plt.xlabel("Barrier Option")
    plt.ylabel("Computational Time")
    plt.title("Computational Time vs. Barrier Option")
    plt.legend(label)
    plt.grid(True)
    plt.show()

#2
    S0 = 92
    K = 100.0
    rf = 0.1
    divR = 0.0
    sigma = 0.25
    T = 1.0  # unit is in years

    n_periods = 10
    H = 90
    M = 1

    call_test2 = CallOption(S0, K, rf, divR, sigma, T)


    axis_s = [92,91,90.5,90.25,90.125]
    axis_sn = [(92,6), (91,8), (90.5,10), (90.25,12),
(90.125,14)]
    axis_sm = [(92,0), (91,1), (90.5,2), (90.25,3), (90.125,4)]
    BS_vec = [call_test2.BS_CallPriceDO(s,H) for s in axis_s]
    print('Black-Scholes Call down-and-out option price from 92,
91, 90, 90.5, 90.25, 90.125 barrier is: ',BS_vec)
    TT_vec = [call_test2.TrinomialTreeEuroCallPriceDO(s,n,H) for
(s,n) in axis_sn]
    print('Trinomial Tree Call down-and-out option price from
92, 91, 90, 90.5, 90.25, 90.125 barrier is: ',TT_vec)
    AMM_vec = [call_test2.AdaptiveMeshEuroCallPrice(s,H,M) for
(s,M) in axis_sm]
    print('Adaptive Mesh Call down-and-out option price from 92,
91, 90, 90.5, 90.25, 90.125 barrier with 0,1,2,3,4 mesh level
is: ',AMM_vec)

    plt.subplot(211)
    plt.plot(axis_s, BS_vec, 'r-', lw=2, label='BS')
    plt.plot(axis_s, AMM_vec, 'b-', lw=2, label='AMM')
    label = ['BS', 'AMM']
    plt.xlabel("Current Price")
    plt.ylabel("Option Price")
    plt.title("European Call Option Price vs. Current Price
Closed to Barrier Option (Adaptive Mesh vs. Black-Scholes)")
```

```
    plt.legend(label)
    plt.grid(True)

    plt.subplot(212)
    plt.plot(axis_s, BS_vec, 'r-', lw=2, label='BS')
    plt.plot(axis_s, TT_vec, 'b-', lw=2, label='TT')
    label = ['BS', 'TT']
    plt.xlabel("Current Price")
    plt.ylabel("Option Price")
    plt.title("European Call Option Price vs. Current Price
Closed to Barrier Option (Trinomial Tree vs. Black-Scholes)")
    plt.legend(label)
    plt.grid(True)
    plt.show()


    TT_time_vec=[call_test2.TTDO_timer(s,n,H) for (s,n) in
axis_sn]
    AMM_time_vec=[call_test2.AMM_timer(s,H,M) for (s,M) in
axis_sm]

    plt.plot(axis_s, TT_time_vec, 'r-', lw=2, label='TT')
    plt.plot(axis_s, AMM_time_vec, 'b-', lw=2, label='AMM')
    label = ['TT', 'AMM']
    plt.xlabel("Barrier Option")
    plt.ylabel("Computation Time")
    plt.title("European Trinomial Tree Call Option Price vs.
Current Price Closed to Barrier Option Performance")
    plt.legend(label)
    plt.grid(True)
    plt.show()

#3
    S0 = 90.5
    K = 100.0
    rf = 0.1
    divR = 0.0
    sigma = 0.25
    T = 1.0  # unit is in years

    n_periods = 10
    H = 90
    M = 2

    call_test3 = CallOption(S0, K, rf, divR, sigma, T)
    AMM_delta_vec = [call_test3.AMM_timer_Delta(S0,M,e) for
(S0,M,e) in
```

```
[(92,0,0.01),(91,1,0.01),(90.5,2,0.01),(90.25,3,0.01)]]
    print('Adaptive Mesh Delta for Mesh 92, 91, 90.5, 90.25 with
e=0.01 is: ', AMM_delta_vec)
    AMM_gamma_vec = [call_test3.AMM_timer_Gamma(S0,M,e) for
(S0,M,e) in
[(92,0,0.01),(91,1,0.01),(90.5,2,0.01),(90.25,3,0.01)]]
    print('Adaptive Mesh Gamma for Mesh 92, 91, 90.5, 90.25 with
e=0.01 is: ', AMM_gamma_vec)
    TT_delta_vec = [call_test3.TT_timer_Delta(S0,n,e) for n,e in
[(25,0.01),(50,0.01),(250,0.01),(1000,0.01)]]
    print('Trinomial Tree Delta for n=25 n=50 n=250 n=1000 with
e=0.01 is: ', TT_delta_vec)
    TT_gamma_vec = [call_test3.TT_timer_Gamma(S0,n,e) for n,e in
[(25,0.01),(50,0.01),(250,0.01),(1000,0.01)]]
    print('Trinomial Tree Gamma for n=25 n=50 n=250 n=1000 with
e=0.01 is: ', TT_gamma_vec)
    axis_time = [0,1,2,3]
    plt.plot(axis_time, AMM_delta_vec, 'r-', lw=2,
label='Delta')
    plt.plot(axis_time, AMM_gamma_vec, 'b-', lw=2,
label='Gamma')
    label = ['Delta','Gamma']
    plt.xlabel("Level of Mesh")
    plt.ylabel("Computation time for Delta and Gamma")
    plt.title("Adaptive Mesh Delta and Gamma vs. Level Of Mesh")
    plt.legend(label)
    plt.grid(True)
    plt.show()
```

3.2 Result of Sample Data and Discussion

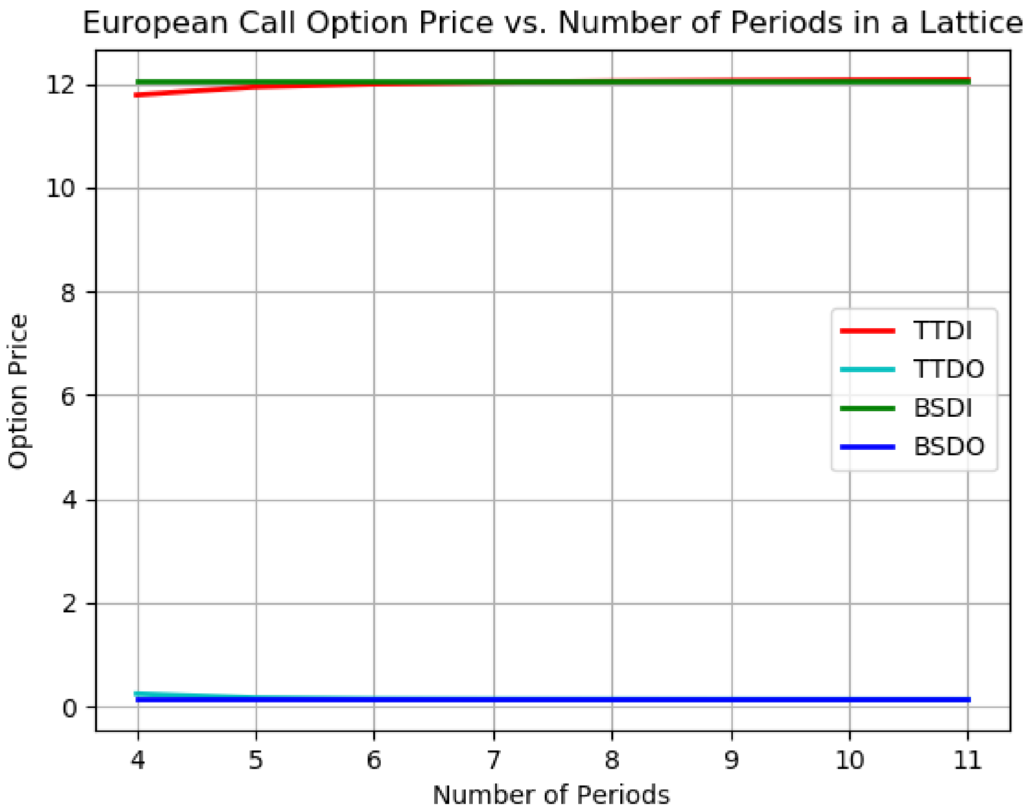
1. Implement a trinomial lattice to price a down-and-in call option with current S =100, strike K=100, r=10%, σ=0.3, time to maturity T = 0.6. Use barriers 95, 99.5 and 99.9. Record the accuracy and computational time.

Calculate call option using Trinomial Tree compared to Black-Scholes Call:

Number of Periods = 10
Barrier = 99.9

Trinomial Tree Call down-and-in option price is: 12.074150589163658
Trinomial Tree Call down-and-out option price is: 0.13914908430783157
Black-Scholes Call down-and-in option price is: 12.052590234785065

Black-Scholes Call down-and-out option price is: 0.13586260468163402

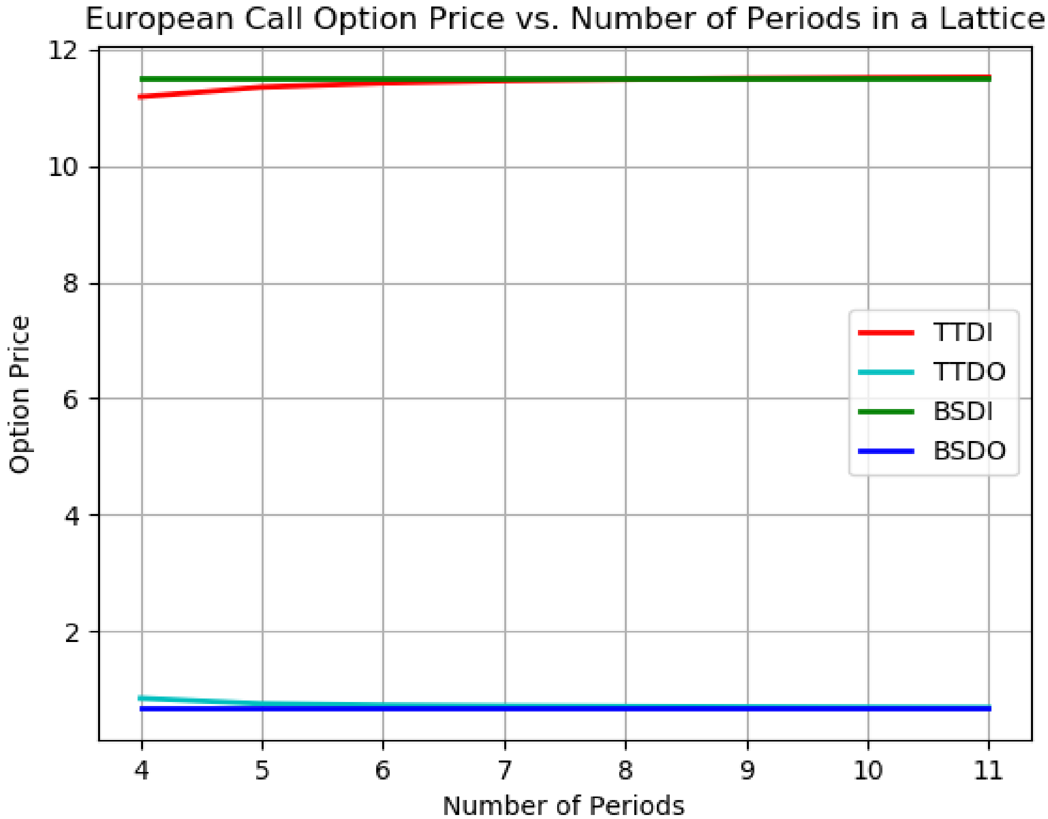


Number of Periods = 10
Barrier = 99.5

Trinomial Tree Call down-and-in option price is: 11.518963159520657
Trinomial Tree Call down-and-out option price is: 0.694336513950832
Black-Scholes Call down-and-in option price is: 11.517788368735062
Black-Scholes Call down-and-out option price is: 0.6706644707316372

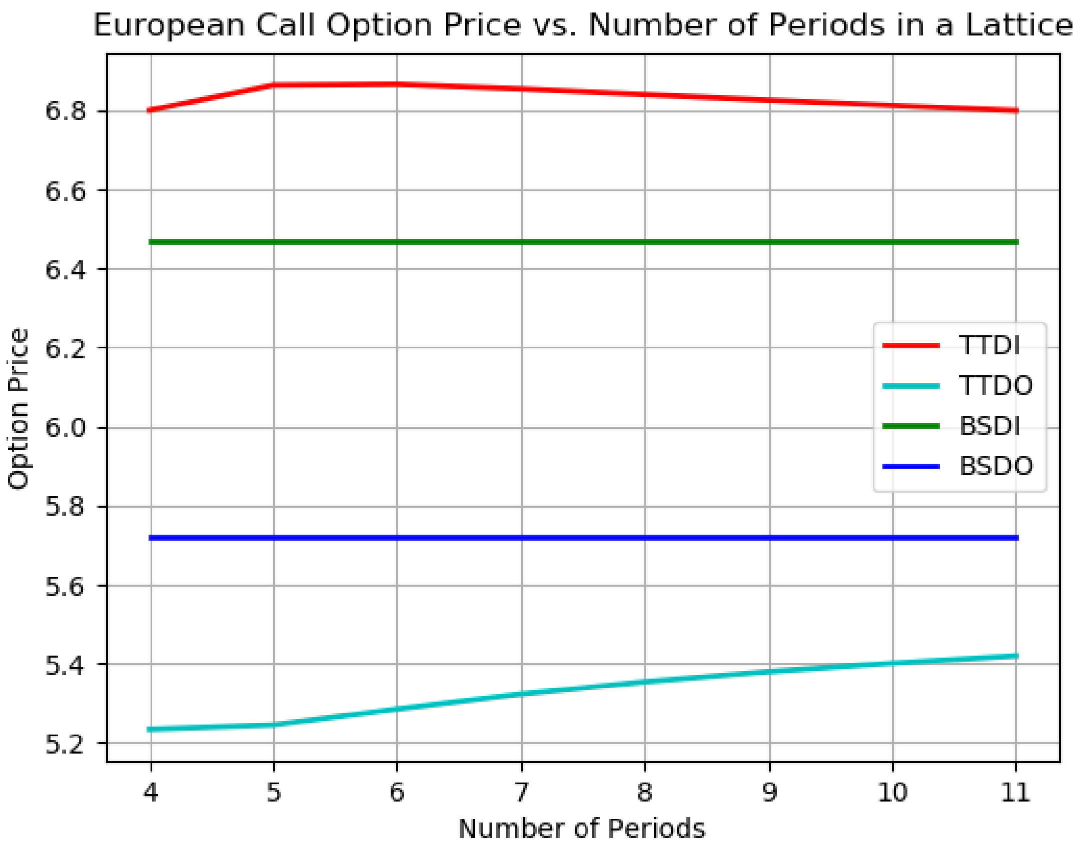
Trinomial Tree Call down-and-in option price from 4 - 12 periods is:
[11.787762981879787, 11.942074542866846, 12.002160426173491, 12.033167496361873, 12.052274667088195, 12.065126586808576, 12.074150589163658, 12.08063488294588]
Trinomial Tree Call down-and-out option price from 4 - 12 periods is:
[0.24614121031197378, 0.16629216719844564, 0.14868470090528518, 0.14381955666499024, 0.14168881391048416, 0.1402763304152277, 0.13914908430783157, 0.13819173323988387]
Black-Scholes Call down-and-in option price from 4 - 12 periods is:
[12.052590234785065, 12.052590234785065, 12.052590234785065,

12.052590234785065, 12.052590234785065, 12.052590234785065,
12.052590234785065, 12.052590234785065]
Black-Scholes Call down-and-out option price from 4 - 12 periods is:
[0.13586260468163402, 0.13586260468163402, 0.13586260468163402,
0.13586260468163402, 0.13586260468163402, 0.13586260468163402,
0.13586260468163402, 0.13586260468163402, 0.13586260468163402,
0.13586260468163402, 0.13586260468163402]



Number of Periods = 10
Trinomial Tree Call down-and-in option price from 4 - 12 periods is:
[11.185977298863635, 11.353090470910733, 11.423165836189835,
11.462009933885879, 11.487435136693138, 11.505516353235167,
11.518963159520657, 11.529253590398048]
Trinomial Tree Call down-and-out option price from 4 - 12 periods is:
[0.8479268933281254, 0.7552762391545595, 0.7276792908889416,
0.7149771191409844, 0.706528344305541, 0.6998865639886365,
0.694336513950832, 0.6895730257877162]
Black-Scholes Call down-and-in option price from 4 - 12 periods is:
[11.517788368735062, 11.517788368735062, 11.517788368735062,

11.517788368735062, 11.517788368735062, 11.517788368735062,
11.517788368735062, 11.517788368735062]
Black-Scholes Call down-and-out option price from 4 - 12 periods is:
[0.6706644707316372, 0.6706644707316372, 0.6706644707316372,
0.6706644707316372, 0.6706644707316372, 0.6706644707316372, 0.6706644707316372,
0.6706644707316372, 0.6706644707316372

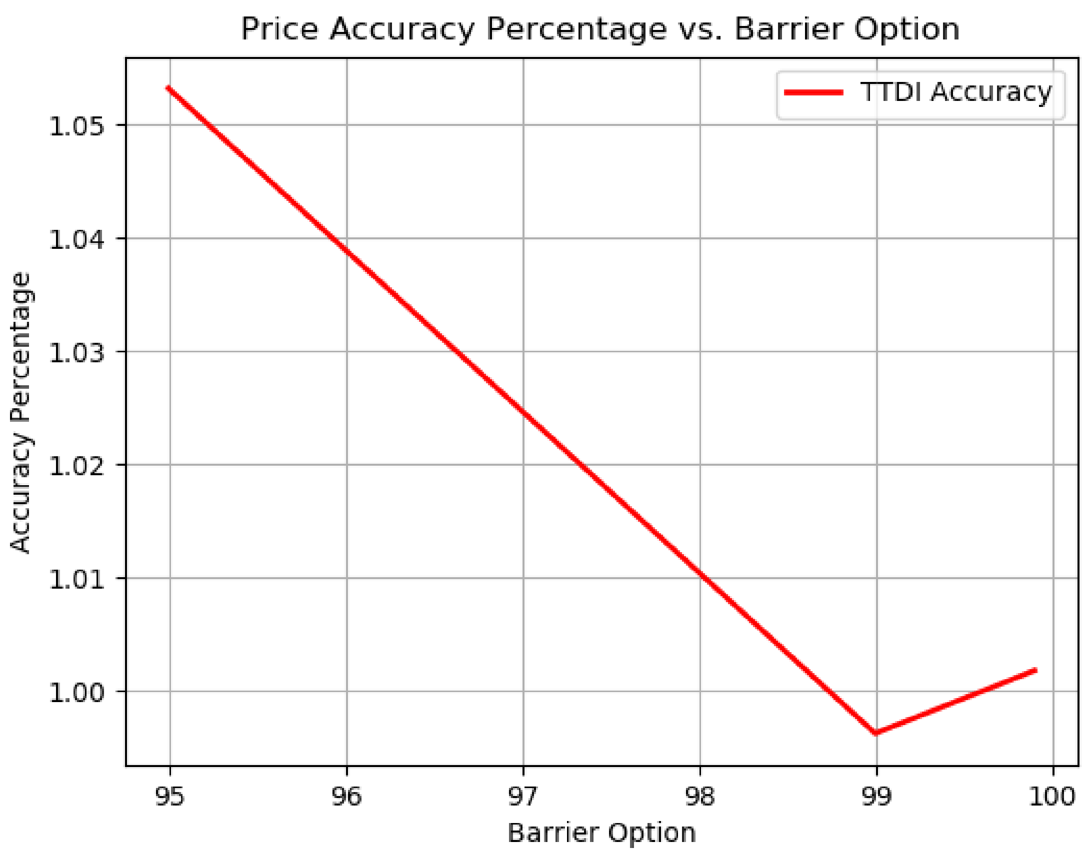


Trinomial Tree Call down-and-in option price from 4 - 12 periods is:
[6.799918990738577, 6.863275064294798, 6.865492188397239,
6.854106761589205, 6.839974555687965, 6.825738189921973,
6.812104041635913, 6.799262399755015]
Trinomial Tree Call down-and-out option price from 4 - 12 periods is:
[5.233985201453184, 5.245091645770494, 5.285352938681537,
5.3228802914376585, 5.3539889253107145, 5.379664727301831,
5.401195631835576, 5.419564216430749]
Black-Scholes Call down-and-in option price from 4 - 12 periods is:
[6.468132457782055, 6.468132457782055, 6.468132457782055,
6.468132457782055, 6.468132457782055, 6.468132457782055,
6.468132457782055, 6.468132457782055]

Black-Scholes Call down-and-out option price from 4 - 12 periods is: [5.720320381684644, 5.720320381684644, 5.720320381684644, 5.720320381684644, 5.720320381684644, 5.720320381684644, 5.720320381684644, 5.720320381684644, 5.720320381684644]

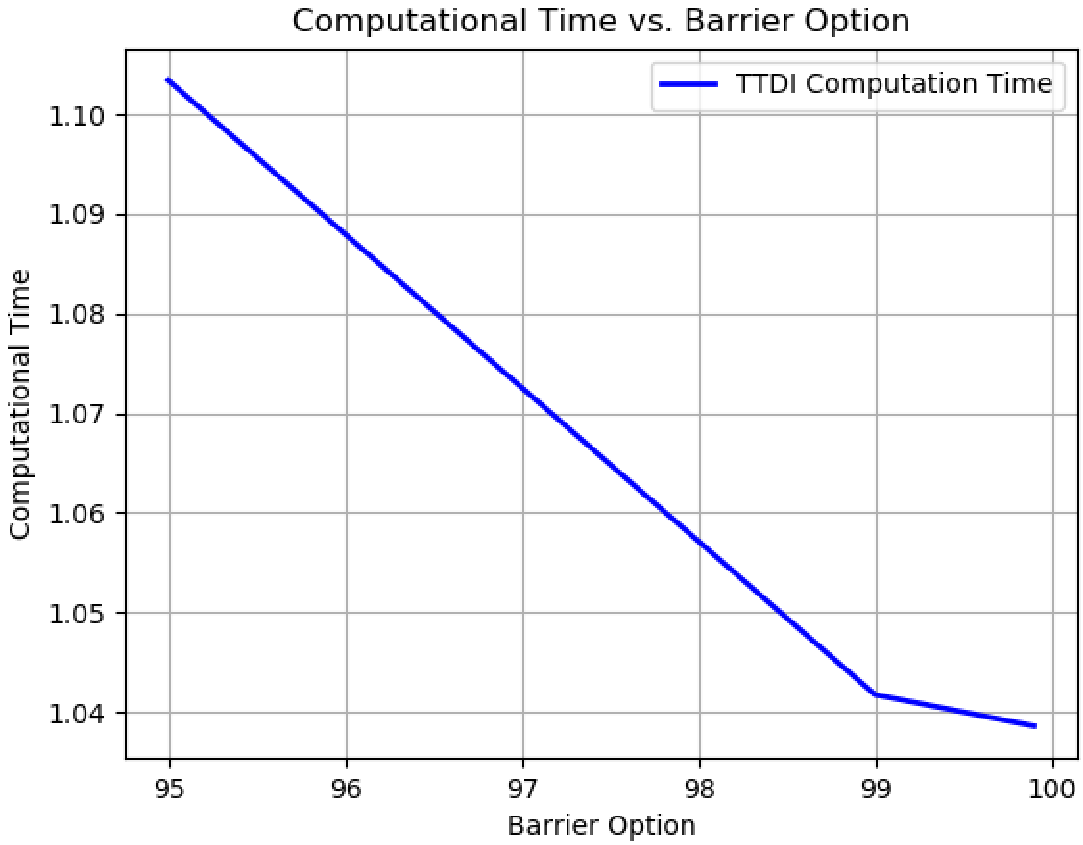
These values show that Trinomial Tree in this implementation works well to price call option as precisely as Black-Scholes Call option. As the number of periods increase, the values of call option are closer to analytic values.

Pricing Accuracy vs. Barrier Option



This figure shows that the pricing accuracy is very close to 1, so it means Trinomial Tree is an effective model to price call option as well as Black-Scholes Method. We can also see that as the barrier option moves away from current price and strike price, the accuracy starts decreasing, the difference gradually shows. It also concludes that the price error can occur in Trinomial Tree Model.

Computation Time vs. Barrier Option



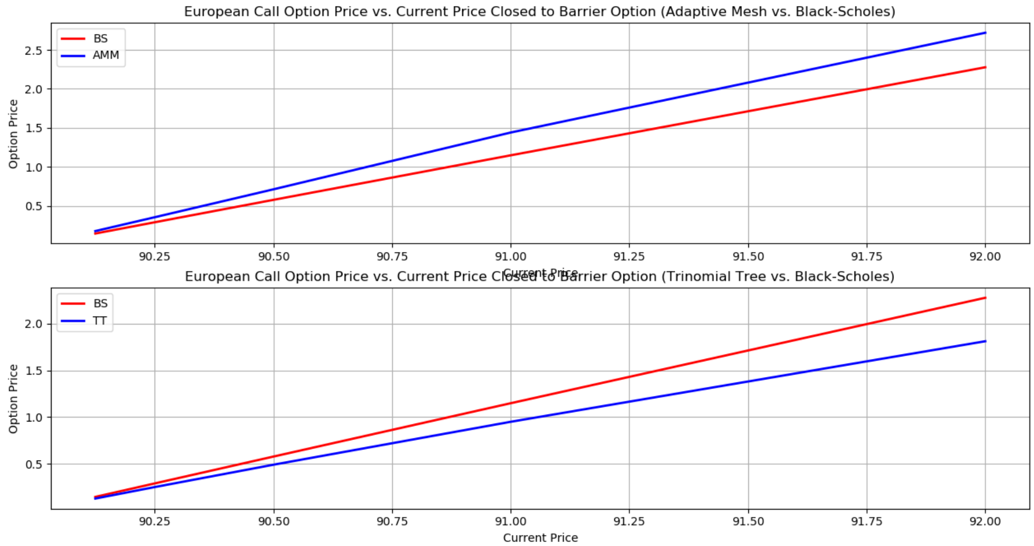
The computation time did not show much difference among different barrier option, the further barrier is away from current price, it takes more time to compute the call option.

2. Implement the AMM for barrier options and replicate Table 3 on page 337 of the AMM paper.

Black-Scholes Call down-and-out option price from 92, 91, 90, 90.5, 90.25, 90.125 barrier is: [2.276723795697471, 1.147400428140088, 0.5760458515913172, 0.28862030455074006, 0.14446088184287476]

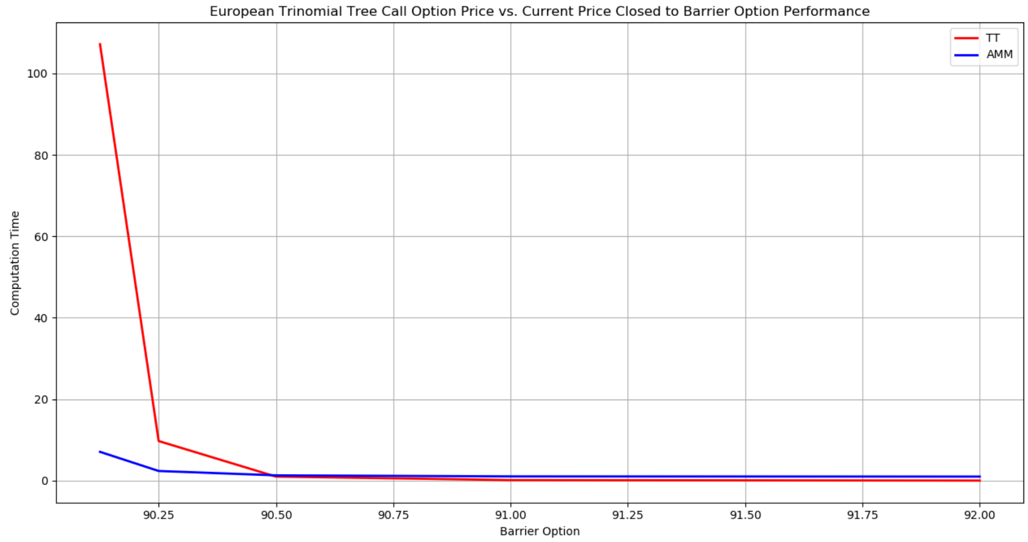
Trinomial Tree Call down-and-out option price from 92, 91, 90, 90.5, 90.25, 90.125 barrier is: [1.8113632958767276, 0.9485871810045623, 0.48880750913965976, 0.24959250523910903, 0.1267579106451766]

Adaptive Mesh Call down-and-out option price from 92, 91, 90, 90.5, 90.25, 90.125 barrier with 0,1,2,3,4 mesh level is: [2.7193529861344055, 1.4396986597498649, 0.7109766203222243, 0.35324494058798533, 0.17604081124088236]



S_0	Analytic value	RTM			AMM		
		Value	Number of time steps	CPU time (s)	Value	AMM level	CPU time (s)
92	2.506	2.507	388	0.033	2.507	0	0.033
91	1.274	1.274	1535	0.750	1.274	1	0.050
$90\frac{1}{2}$	0.642	0.642	6108	12.35	0.643	2	0.059
$90\frac{1}{4}$	0.323	0.323	24,367	364.3	0.323	3	0.117
$90\frac{1}{8}$	0.162	N/A	97,335	N/A	0.162	4	0.317

Both Trinomial Tree Call Option and Adaptive Mesh Method compute the option price closed to Black-Scholes Model (Analytic Value). Computing AMM using exact matching pair of (92,0), (91,1), (90.5, 2), (90.25, 3), (90.125, 4) generate the results closed to Analytic values. However, due to limited computer configuration, it takes significant amount of time to compute in the above number of time steps, so I determine to shrink the time step into 4-12. In comparison, Adaptive Mesh performs greater efficiency and higher accuracy than Trinomial Tree.

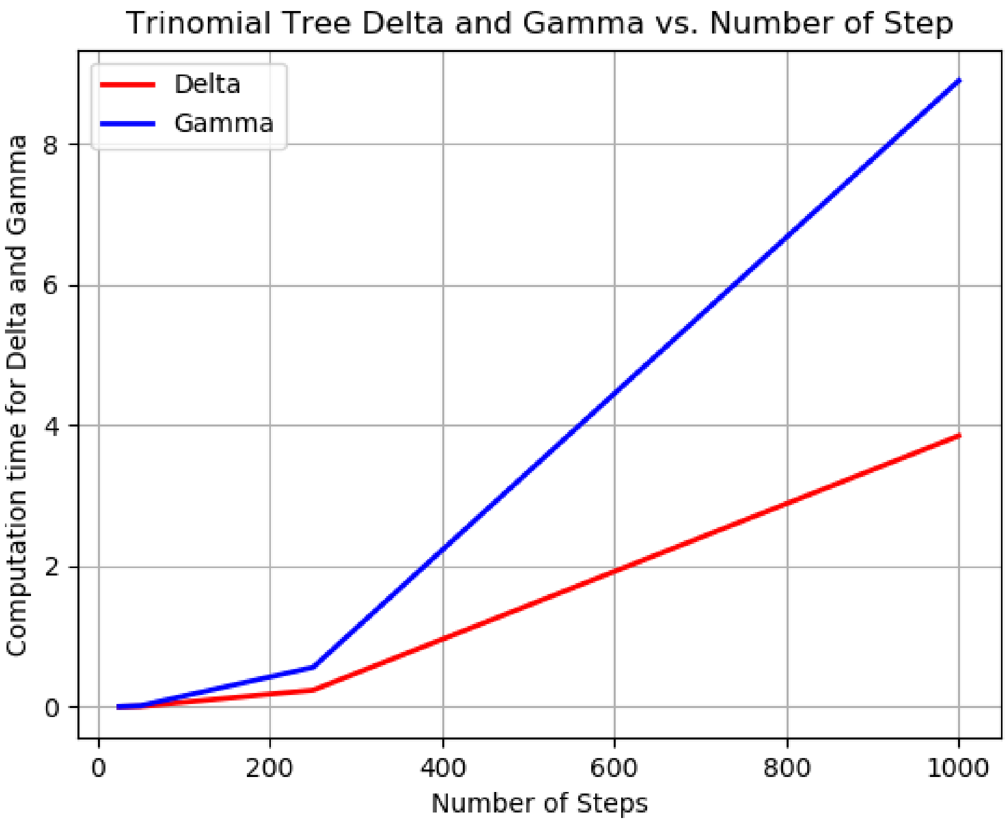


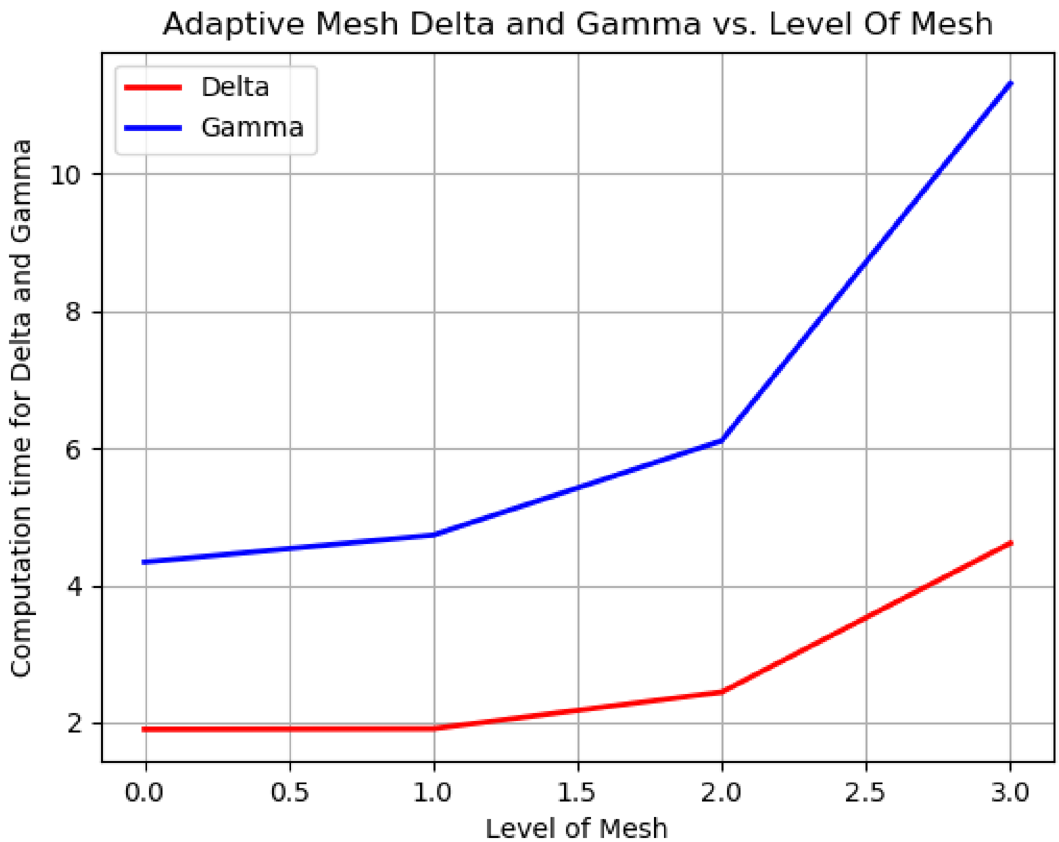
The above figure shows the significant difference of computation time between Trinomial Tree model and Adaptive Mesh Method to achieve similar accuracy, which reflects the table's content. The Trinomial Tree takes hundreds of seconds to compute call option. Instead, adaptive mesh with higher mesh level of 4, is able to complete the calculation within 10 seconds.

3. Compute the delta and gamma of the barrier options using both the regular trinomial lattice and the AMM; report the errors with respect to the closed-form values; comment on the performance of the AMM for computing Greeks of the barrier options.

Adaptive Mesh Delta for e=0.01 and e=0.001 is:
[2.439176082611084, 2.439232349395752]
Adaptive Mesh Gamma for e=0.01 and e=0.001 is:
[6.163419485092163, 6.074234962463379]
Adaptive Mesh Delta for Mesh 92, 91, 90.5, 90.25 with e=0.01 is:
[1.7746994495391846, 1.9749455451965332, 2.4314804077148438, 4.730239391326904]
Adaptive Mesh Gamma for Mesh 92, 91, 90.5, 90.25 with e=0.01 is:
[4.538653135299683, 4.629465579986572, 6.043651580810547, 11.027745485305786]
Trinomial Tree Delta for n=25 n=50 n=250 n=1000 with e=0.01 is:
[0.001993417739868164, 0.009234905242919922, 0.23792505264282227, 3.852433443069458]

Trinomial Tree Gamma for n=25 n=50 n=250 n=1000 with e=0.01 is:
[0.004268646240234375, 0.019402742385864258, 0.5622296333312988, 8.899118661880493]





In comparison of two diagrams, Adaptive Mesh shows its advantage of computing in less CPU time, especially, the trend of CPU time grows less significant as the level of mesh increments and compute more accurate delta and gamma than Trinomial Tree. Gamma requires more time and resource to compute because it utilizes extra computing algorithm to process the result.

4. Improvement

In this project, the algorithm used for calculating the call option in Trinomial Tree should be optimized. Currently, its takes too much resource and CPU time to compute in large number of time steps. Delta and Gamma results still need more improvement to obtain closer experiment data to the paper. Noticing the result of adaptive mesh and trinomial tree is slightly different than theoretical values, it needs more improvement to close the numeric gap and reach greater accuracy. However, the computer configuration, incomplete python library especially algorithm defects could contribute to those imperfect results.

5. Conclusions

In this project, I learned how to implement Binomial Tree Model, Trinomial Tree Model, Adaptive Mesh Method in computing European Call Option based on different parameters: number of time steps, barrier option, current price level, mesh level. All the implementations are based on solid understandings of European call option financial theorem.
Computing methods are derived from risk-neutral probability setup and parameters such as current pricing, strike pricing, alpha, sigma, time length, risk-free interest rate, dividend rate.
The Trinomial Tree down-and-in, down-and-out algorithms are expensive when it constructs large numbers of periods, requiring the complexity of 3^n to generate paths to reach maturity prices. This project implementation truncates time step from requirements and perform reasonable computational results as the paper states and Black-Scholes model. The computing methods share common characteristics between Binomial Tree and Trinomial Tree model.
When the computing method comes to adaptive mesh, it gets complicated because the algorithm needs to control mesh level and store many lists of data for further processing. The first mesh computes the values from top to down to barrier option price level, the second, third, fourth, fifth mesh computes deeper near the barrier option price level to obtain more precise call option value.
Delta and Gamma computation introduces much performance improve for adaptive mesh method not limited to save computation time, but also the accuracy.
Most importantly, this project involves significant amount of mathematics logic and formula to construct the model using Python, taking the implementation enhances my understandings of the algorithm. It takes me to learn many powerful python library such as not only numpy, matplotlib, but also iteratortools, which establishes pricing path movement across the current price to maturity. I believe this project experience is a valuable add-on to my programming skill, financial knowledge about European call option, and the implementation of mathematics model.

Reference:
[1] “Stephen Figlewski, Bin Gao", “The adaptive mesh model: a new approach to efficient option pricing”, Stern School of Business, New York University, New York, 44 West 4th Street, NY 10012, USA
"Graduate School of Business, University of North Carolina, Chapel Hill, NC 27599, USA
[2] “Niklas Westermark”, Barrier Option Pricing Degree Project in Mathematics, First Level
[3] “EVAN TURNER”, “THE BLACK-SCHOLES MODEL AND EXTENSIONS”

Appendices

Code:

The python program code file has been attached with the submission. With Pycharm and libaries

GitHub:
https://github.com/lvwf1/ISyE6785_Project1_TT_AMM/blob/master/calloption.py