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

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

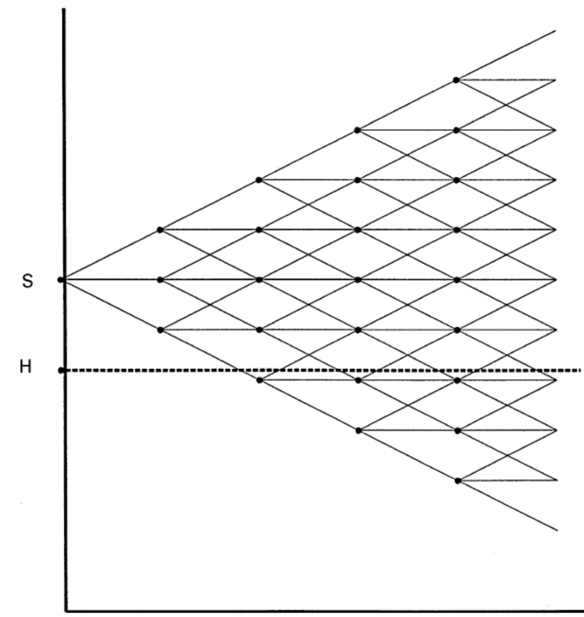


Fig. 1. A trinomial model for a barrier option. The barrier, H, lies just slightly less than two price steps below the current asset price, S. The option is knocked out if the price falls two steps below the initial price at any time prior to expiration.

B. Adaptive Mesh Model

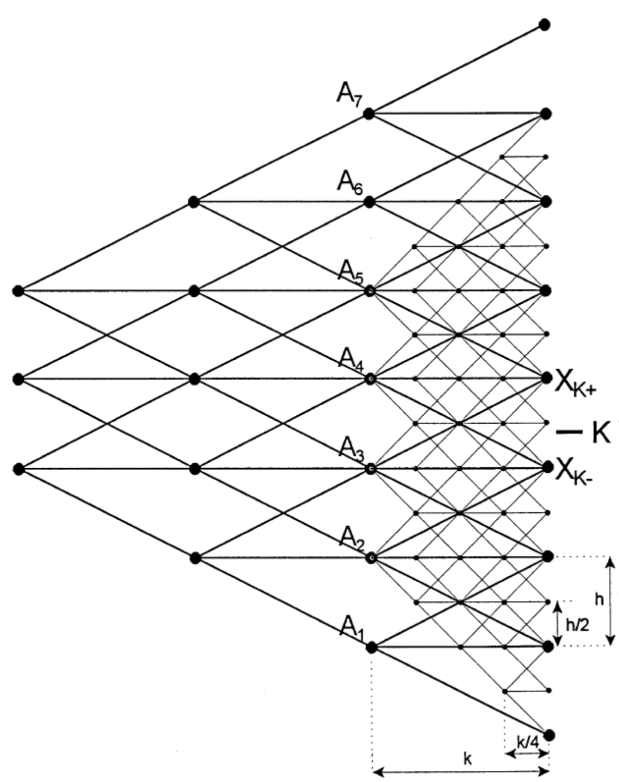


Fig. 2. An adaptive mesh model for the European put. This Trinomial tree shows the section of the pricing lattice in the immediate vicinity of the strike price in the last few periods before expiration. The coarse lattice, with price and time steps h and k, is represented by heavy lines. The "ne mesh, 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.

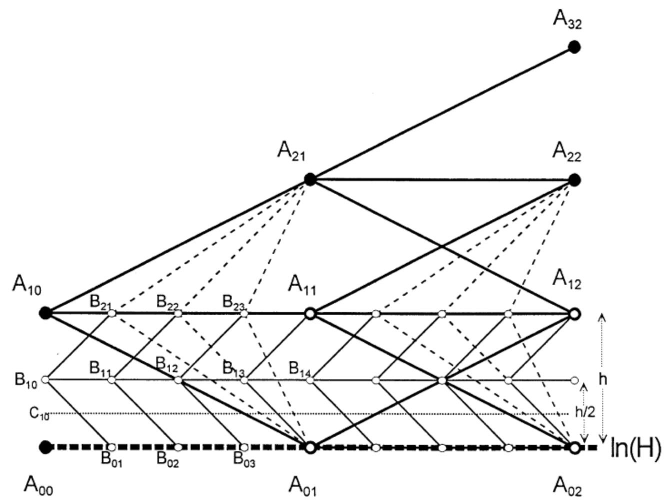


Fig. 3. Adaptive mesh model for a barrier option. The heavy lines indicate the coarse-mesh lattice, whose nodes are labelled Aij, with i indicating the number of coarse-mesh price steps above the barrier and j the number of the coarse time step. The "ne-mesh nodes are labelled Bij. The barrier price 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)

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

4. Difference Step between Stock Price

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

5. Next Price and Final Price

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

$$C(B_{23}) = \mathrm{e}^{-rk/4}(p_\mathrm{u}(h,k/4)C(A_{21}) + p_\mathrm{m}(h,k/4)C(A_{11}) + p_\mathrm{d}(h,k/4)C(A_{01})).$$

$$(B_{10}) = \mathrm{e}^{-rk/4}[p_\mathrm{u}(h/2,k/4)C(B_{21}) + p_\mathrm{m}(h/2,k/4)C(B_{11}) + p_\mathrm{d}(h/2,k/4)C(B_{01})].$$

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

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

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

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

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:

```python
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 (np.log(S0 / self.K) + (self.rf - self.sigma ** 2
/ 2.0) * self.tyears) / (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) *
self.BS_CallPrice(H**2/S0)

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

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

        # 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 = 2*N
        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)

        # Backward recursion for computing option prices in for
each time periods N-1, N-2, ... , 0
        for t in range(1, N + 1):
            fs = []
            for i in range(1, 2 * (N - t) + 2):  # number of
nodes at step j
                fs.append(0.0)
                if (t != N):
                    if X0 + (N - t - i + 1) * h > np.log(H):
                        fs[-1] = np.exp(-self.rf * deltaT) * (qU
* fs_pre[i - 1] + qM * fs_pre[i] + qD * fs_pre[i + 1])
                else:
                    fs[-1] = np.exp(-self.rf * deltaT) * (qU *
fs_pre[i - 1] + qM * fs_pre[i] + qD * fs_pre[i + 1])
            fs_pre = fs

        return fs[0]

    def TrinomialTreeEuroCallPriceDI(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.TrinomialTreeEuroCallPriceDO(S,N,H)

    def TrinomialTreeEuroCallPriceRTM(self, S0=100, H=100):

        startTime = time.time()

        # Constant Parameters
        X0 = np.log(S0)
        alpha = self.rf - (self.sigma ** 2) / 2
        h = X0 - np.log(H)
        k = self.tyears / math.floor((3.0 * self.sigma ** 2 / h
```

```
** 2) * self.tyears)
        N = int(self.tyears / k)

        # Risk-neutral probabilities:
        [pD, pM, pU] = self.computeProbas(alpha, h, k)

        # Initialize the stock prices and option values at
maturity with X0
        stk = [X0]

        for i in range(N):
            stk = [stk[0] + h] + stk + [stk[-1] - h]

        fs_pre = stk

        # Perform Down-And-Out Call Option Lattice
        for i in range(2 * N + 1):
            if stk[i] > np.log(H):
                fs_pre[i] = max(np.exp(stk[i]) - K, 0.0)

        # Backward recursion for computing option prices in for
each time periods N-1, N-2, ... , 0
        for j in range(1, N + 1):
            fs = []
            for i in range(1, 2 * (N - j) + 2):  # number of
nodes at step j
                fs.append(0.0)
                if (j != N):
                    if (np.log(S0) + (N - j - i + 1) * h >
np.log(H)):
                        fs[-1] = np.exp(-self.rf * k) * (pU *
fs_pre[i - 1] + pM * fs_pre[i] + pD * fs_pre[i + 1])
                else:
                    fs[-1] = np.exp(-self.rf * k) * (pU *
fs_pre[i - 1] + pM * fs_pre[i] + pD * fs_pre[i + 1])
            fs_pre = fs
        return [fs[0],time.time()-startTime]

    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 = 2 * N
        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, M, H):

        startTime = time.time()

        # Constant Parameters
```

```
        X0 = np.log(S0)
        alpha = self.rf - (self.sigma ** 2) / 2
        h = (2 ** M) * (X0 - np.log(H))
        k = self.tyears / math.floor((3.0 * self.sigma ** 2 / h
** 2) * self.tyears)
        N = int(self.tyears / k)

        # Initialize the stock prices and option values at
maturity with X0
        fs_pre = []
        for i in range(N + 1):
            stk = []
            for j in range(2 * i + 1):
                stk.append((np.log(H) + h - i * h) + j * h)
            fs_pre.append(stk)

        # Compute the payoff of the lattice A
        fs_A = []
        finalPayoffA = []
        for i in range(len(fs_pre[N])):
            finalPayoffA.append(max(np.exp(fs_pre[N][i])-
self.K,0))
            fs_A.append(finalPayoffA)
        # Calculate Risk-neutral probabilities:
        [pD, pM, pU] = self.computeProbas(alpha, h, k)
        for i in range(1, N + 1):
            opTree = []
            for j in range(2 * (N - i) + 1):
                if (np.exp(fs_pre[N - i][j]) > H):
                    C = np.exp(-self.rf * k) * (pD * fs_A[0][j]
+ pM * fs_A[0][j + 1] + pU * fs_A[0][j + 2])
                else:
                    C = 0.0
                opTree.append(C)
            fs_A.insert(0, opTree)
        finalValue = fs_A[0][0]
        # Construction of the lattice B
        delta = 0
        gamma = 0
        if M > 0:
            fs_B = []
            fs_B.append([0, 0, fs_pre[N][N]])
            j = 1
            B = [0, 0, 0]
            for i in range(1, N * 4 + 1):
                stepA = int(np.ceil(N - i / 4.0))
                [pD, pM, pU] = self.computeProbas(alpha, h / 2,
```

```
k / 4)
                B[1] = np.exp(-self.rf * k / 4) * (pD * 0 + pM *
fs_B[0][1] + pU * fs_B[0][2])
                if (j > 0):
                    [pD, pM, pU] = self.computeProbas(alpha, h,
j * k / 4)
                    B[2] = np.exp(-self.rf * j * k / 4) * (pD *
0 + pM * fs_A[stepA][stepA] + pU * fs_A[stepA][stepA + 1])
                else:
                    B[2] = fs_A[stepA][stepA]

                fs_B.insert(0, B)
                j += 1
                if (j > 3):
                    j = 0
            finalValue = fs_B[0][1]
            delta = (fs_B[0][2] - fs_B[0][0]) / (2.0 * h / 2 *
S0)
            gamma = (1.0 / S0 ** 2) * (((fs_B[0][2] + fs_B[0][0]
- 2 * fs_B[0][1]) / (h / 2 ** 2)) - (fs_B[0][2] - fs_B[0][0]) /
(2.0 * h / 2))
            # Construction of the lattice C
            if M > 1:
                fs_C = []
                fs_C.append([0, 0, fs_pre[N][N]])
                j = 1
                C = [0, 0, 0]
                for i in range(1, N * 16 + 1):
                    stepA = int(np.ceil(4 * N - i / 4.0))
                    [pD, pM, pU] = self.computeProbas(alpha, h /
4, k / 16)
                    C[1] = np.exp(-self.rf * k / 16) * (pD * 0 +
pM * fs_C[0][1] + pU * fs_C[0][2])
                    if (j > 0):
                        [pD, pM, pU] = self.computeProbas(alpha,
h / 2, j * k / 16)
                        C[2] = np.exp(-self.rf * j * k / 16) *
(pD * 0 + pM * fs_B[stepA][1] + pU * fs_B[stepA][2])
                    else:
                        C[2] = fs_B[stepA][1]

                    fs_C.insert(0, C)
                    j += 1
                    if (j > 3):
                        j = 0
                finalValue = fs_C[0][1]
                delta = (fs_C[0][2] - fs_C[0][0]) / (2.0 * h / 4
```

```
* S0)
                gamma = (1.0 / S0 ** 2) * (((fs_C[0][2] +
fs_C[0][0] - 2 * fs_C[0][1]) / (h / 4 ** 2)) - (fs_C[0][2] -
fs_C[0][0]) / (2.0 * h / 4))
                # Construction of the lattice D
                if M > 2:
                    fs_D = []
                    fs_D.append([0, 0, 0])
                    j = 1
                    D = [0, 0, 0]
                    for i in range(1, N * 64 + 1):
                        stepA = int(np.ceil(16 * N - i / 4.0))
                        [pD, pM, pU] = self.computeProbas(alpha,
h / 8, k / 64)
                        D[1] = np.exp(-self.rf * k / 64) * (pD *
0 + pM * fs_D[0][1] + pU * fs_D[0][2])
                        if (j > 0):
                            [pD, pM, pU] =
self.computeProbas(alpha, h / 4, j * k / 64)
                            D[2] = np.exp(-self.rf * j * k / 64)
* (pD * 0 + pM * fs_C[stepA][1] + pU * fs_C[stepA][2])
                        else:
                            D[2] = fs_C[stepA][1]

                        fs_D.insert(0, D)
                        j += 1
                        if (j > 3):
                            j = 0
                    finalValue = fs_D[0][1]
                    delta = (fs_D[0][2] - fs_D[0][0]) / (2.0 * h
/ 8 * S0)
                    gamma = (1.0 / S0 ** 2) * (((fs_D[0][2] +
fs_D[0][0] - 2 * fs_D[0][1]) / (h / 8 ** 2)) - (fs_D[0][2] -
fs_D[0][0]) / (2.0 * h / 8))
                    # Construction of the lattice E
                    if M > 3:
                        fs_E = []
                        fs_E.append([0, 0, 0])
                        j = 1
                        E = [0, 0, 0]
                        for i in range(1, N * 256 + 1):
                            stepA = int(np.ceil(16 * N - i /
4.0))
                            [pD, pM, pU] =
self.computeProbas(alpha, h / 16, k / 256)
                            E[1] = np.exp(-self.rf * k / 256) *
(pD * 0 + pM * fs_E[0][1] + pU * fs_E[0][2])
```

```
                                        if (j > 0):
                                            [pD, pM, pU] =
self.computeProbas(alpha, h / 16, j * k / 256)
                                            E[2] = np.exp(-self.rf * j * k /
256) * (pD * 0 + pM * fs_D[stepA][1] + pU * fs_D[stepA][2])
                                        else:
                                            E[2] = fs_D[stepA][1]

                                        fs_E.insert(0, E)
                                        j += 1
                                        if (j > 3):
                                            j = 0
                                    finalValue = fs_E[0][1]
                                    delta = (fs_E[0][2] - fs_E[0][0]) / (2.0
* h / 16 * S0)
                                    gamma = (1.0 / S0 ** 2) * (((fs_E[0][2]
+ fs_E[0][0] - 2 * fs_E[0][1]) / (h / 16 ** 2)) - (fs_E[0][2] -
fs_E[0][0]) / (2.0 * h / 17))

        return [finalValue, time.time() - startTime, delta,
gamma]

    def computeProbas(self, alpha, h, k):
        pU = 0.5 * ((self.sigma ** 2) * (k / (h ** 2)) + (alpha
** 2) * ((k ** 2) / (h ** 2)) + alpha * (k / h))
        pD = 0.5 * ((self.sigma ** 2) * (k / (h ** 2)) + (alpha
** 2) * ((k ** 2) / (h ** 2)) - alpha * (k / h))
        pM = 1 - pD - pU
        return [pD, pM, pU]

    def TTDI_timer(self, S, N, H):
        start = time.time()
        self.TrinomialTreeEuroCallPriceDI(S, N, H)
        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 = 200
    H = 99.5

    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(50, 1000, 50)
    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 50
- 1000 periods is: ',TTDI_vec)
    print('Trinomial Tree Call down-and-out option price from 50
- 1000 periods is: ',TTDO_vec)
    print('Black-Scholes Call down-and-in option price from 50 -
1000 periods is: ', BSDI_vec)
    print('Black-Scholes Call down-and-out option price from 50
- 1000 periods is: ', BSDO_vec)
    plt.plot(axis_n, TTDI_vec, 'r-', lw=2, label='TTDI')
    plt.plot(axis_n, BSDI_vec, 'b-', lw=2, label='BSDI')
    label = ['TTDI','BSDI']
    plt.xlabel("Number of Periods")
    plt.ylabel("Option Price")
    plt.title("European Call Down-And-In Option Price vs. Number
of Periods in a Lattice")
    plt.legend(label)
    plt.grid(True)
    plt.show()

    axis_h = np.array([95, 99.5, 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])
    TTDI_vec4 = np.array([call_test.TTDI_timer(S0, n_periods, H)
for n_periods in axis_n])
    plt.subplot(211)
    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.subplot(212)
    plt.plot(axis_n, TTDI_vec4, 'b-', lw=2)
    label = ['TTDI Computation Time']
    plt.xlabel("Time Steps")
    plt.ylabel("Computational Time")
    plt.title("Computational Time vs. Number of Time Steps")
    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
```

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

    axis_s = [92,91,90.5,90.25]
    axis_sn = [92,91,90.5,90.25]
    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.TrinomialTreeEuroCallPriceRTM(s,H) for
s in axis_sn]
    print('Trinomial Tree Call down-and-out option price from
92, 91, 90, 90.5, 90.25, 90.125 barrier is: ',[vec[0] for vec in
TT_vec])
    print('Trinomial Tree Call computation time from 92, 91, 90,
90.5, 90.25, 90.125 barrier is: ',[vec[1] for vec in TT_vec])
    AMM_vec = [call_test2.AdaptiveMeshEuroCallPrice(s,M,H) for
(s,M) in axis_sm]
    print(AMM_vec)
    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: ',[vec[0] for vec in AMM_vec])
    print('Adaptive Mesh Call computation time from 92, 91, 90,
90.5, 90.25, 90.125 barrier with 0,1,2,3,4 mesh level is:
',[vec[1] for vec in AMM_vec])
    print('Adaptive Mesh Call Delta from 92, 91, 90, 90.5,
90.25, 90.125 barrier with 0,1,2,3,4 mesh level is: ',[vec[2]
for vec in AMM_vec])
    print('Adaptive Mesh Call Gamma from 92, 91, 90, 90.5,
90.25, 90.125 barrier with 0,1,2,3,4 mesh level is: ',[vec[3]
for vec in AMM_vec])

    plt.subplot(211)
    plt.plot(axis_s, BS_vec, 'r-', lw=2, label='BS')
    plt.plot(axis_s, [vec[0] for vec in 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, [vec[0] for vec in 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()

    plt.plot(axis_s, [vec[1] for vec in TT_vec], 'r-', lw=2,
label='TT')
    plt.plot(axis_s, [vec[1] for vec in AMM_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

    axis_time = [0,1,2,3]
    plt.plot(axis_time, [vec[2] for vec in AMM_vec], 'r-', lw=2,
label='Delta')
    plt.plot(axis_time, [vec[3] for vec in AMM_vec], 'b-', lw=2,
label='Gamma')
    label = ['Delta','Gamma']
    plt.xlabel("Level of Mesh")
    plt.ylabel("Delta and Gamma Value")
    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:

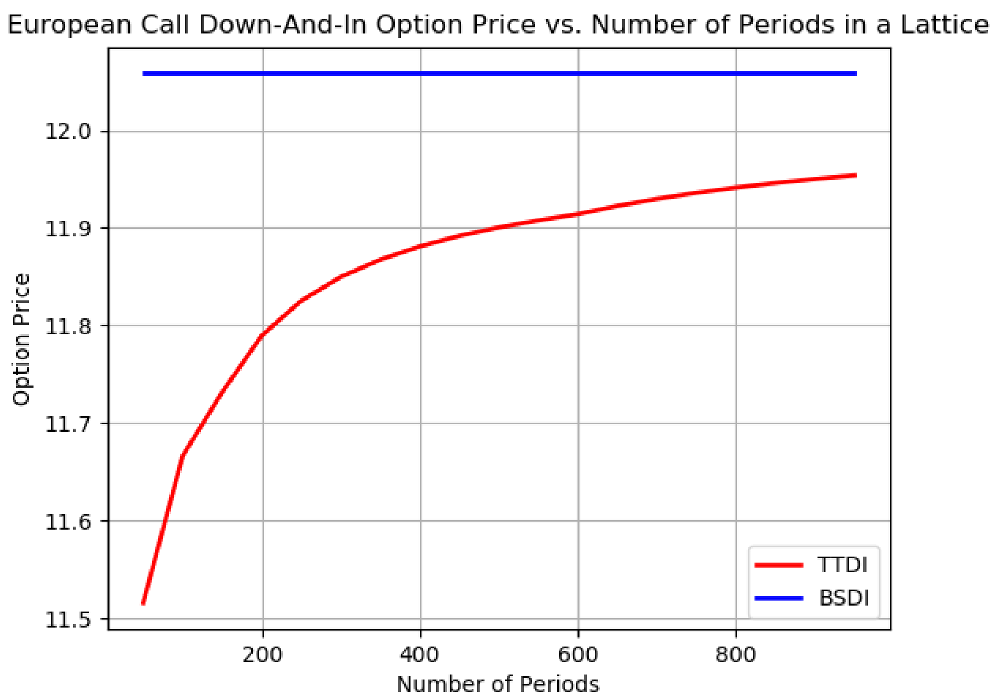
Number of Periods = 1000
Barrier = 99.9

Trinomial Tree Call down-and-in option price is: 11.956905538984914
Trinomial Tree Call down-and-out option price is: 0.23251897232274682
Black-Scholes Call down-and-in option price is: 12.058322170160004
Black-Scholes Call down-and-out option price is: 0.1301306693066948

Number of Periods = 50 - 1000
Barrier = 99.9

Trinomial Tree Call down-and-in option price from 50 - 1000 periods is:
[11.515528808555192, 11.666254930257423, 11.731566807992836, 11.789463170232116, 11.825186484674196, 11.849558521153913, 11.86731708157089, 11.88087259913231, 11.891584301045413, 11.900278529188608, 11.907487417083157, 11.913929032536073, 11.92238454676524, 11.92957316947411, 11.935752797586803, 11.941116325899682, 11.945810838619565, 11.94995039402108, 11.953624763053016]
Trinomial Tree Call down-and-out option price from 50 - 1000 periods is:
[0.6931063725157338, 0.5198864981630227, 0.4428114732405034, 0.3968047621839955, 0.3653938998063063, 0.34220544430477723, 0.3241854426599331, 0.3096632913669337, 0.29763806342509863, 0.2874687163898259, 0.2787231804019293, 0.2710982171675702, 0.2643739003993844, 0.25838632569229414, 0.25301050630426947, 0.2481492467055039, 0.24372567211288632, 0.23967807419177467, 0.23595626946896345]
Black-Scholes Call down-and-in option price from 50 - 1000 periods is:
[12.058322170160004, 12.058322170160004, 12.058322170160004, 12.058322170160004, 12.058322170160004, 12.058322170160004, 12.058322170160004, 12.058322170160004, 12.058322170160004, 12.058322170160004, 12.058322170160004, 12.058322170160004, 12.058322170160004, 12.058322170160004, 12.058322170160004, 12.058322170160004, 12.058322170160004, 12.058322170160004, 12.058322170160004]
Black-Scholes Call down-and-out option price from 50 - 1000 periods is:
[0.1301306693066948, 0.1301306693066948, 0.1301306693066948, 0.1301306693066948, 0.1301306693066948, 0.1301306693066948, 0.1301306693066948, 0.1301306693066948, 0.1301306693066948,

0.1301306693066948, 0.1301306693066948, 0.1301306693066948, 0.1301306693066948, 0.1301306693066948, 0.1301306693066948, 0.1301306693066948, 0.1301306693066948, 0.1301306693066948, 0.1301306693066948]



Number of Periods = 1000
Barrier = 99.5

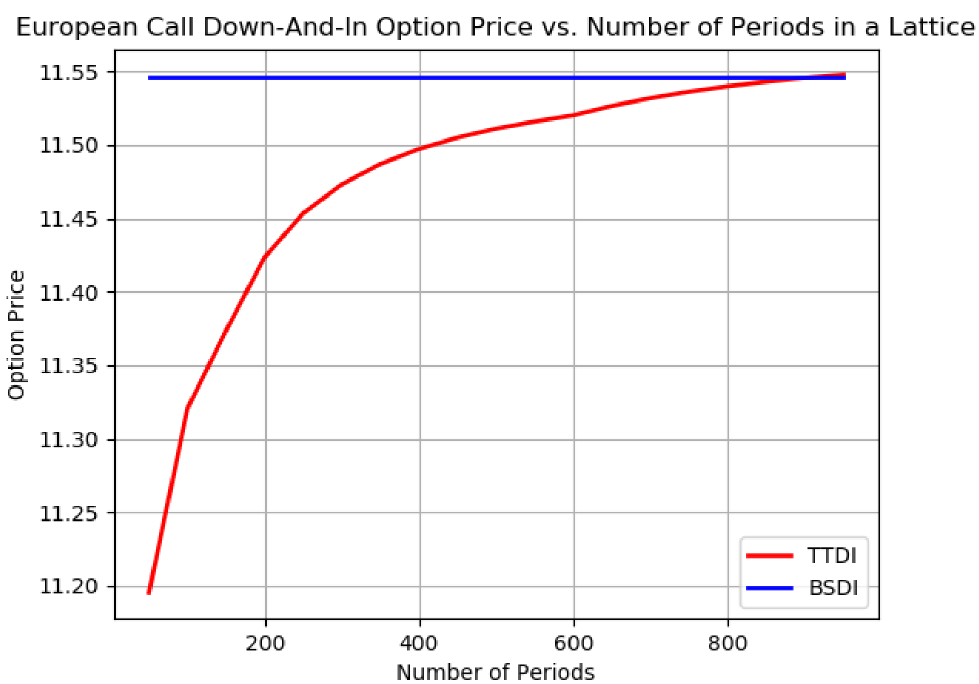
Trinomial Tree Call down-and-in option price is: 11.549572698024445
Trinomial Tree Call down-and-out option price is: 0.6398518132832163
Black-Scholes Call down-and-in option price is: 11.545788946722208
Black-Scholes Call down-and-out option price is: 0.6426638927444905

Number of Periods = 50 - 1000
Barrier = 99.5

Trinomial Tree Call down-and-in option price is: 5.486749636174782
Trinomial Tree Call down-and-out option price is: 6.6995182962413296
Black-Scholes Call down-and-in option price is: 6.670701053402194
Black-Scholes Call down-and-out option price is: 5.517751786064505

Barrier = 99.5
Number of Periods = 50 - 1000

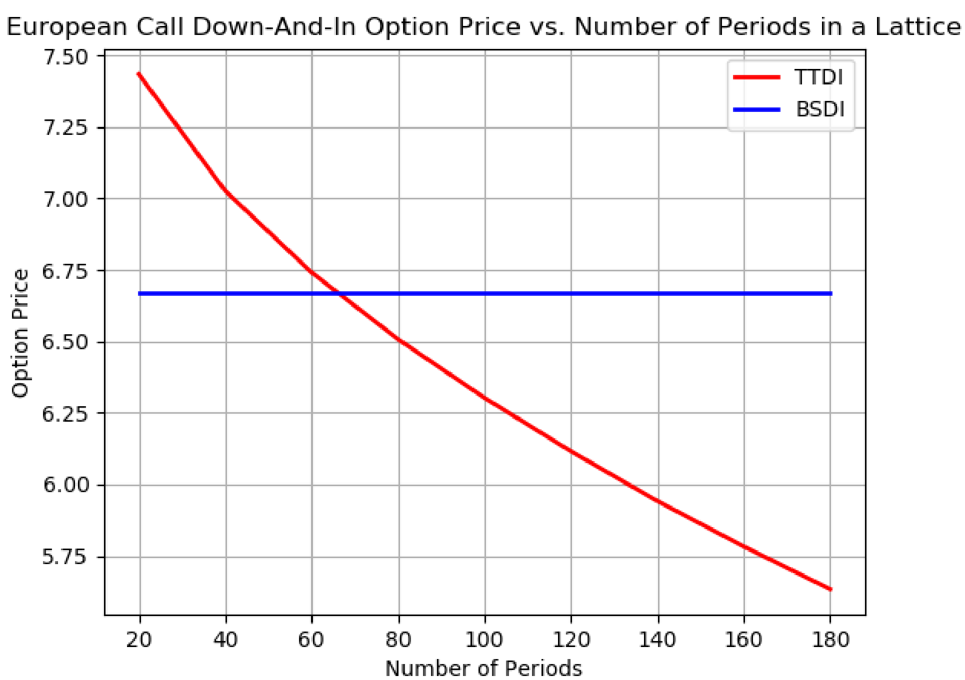
Trinomial Tree Call down-and-in option price from 4 - 12 periods is:
[11.787762981879787, 11.942074542866846, 12.002160426173491, 12.033167496361873, 12.052274667088195, 12.065126586808576, 12.074150589163658, 12.08063488294588]
Trinomial Tree Call down-and-out option price from 4 - 12 periods is:
[0.24614121031197378, 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]



Number of Periods = 200
Barrier = 95
Trinomial Tree Call down-and-in option price is: 5.486749636174782

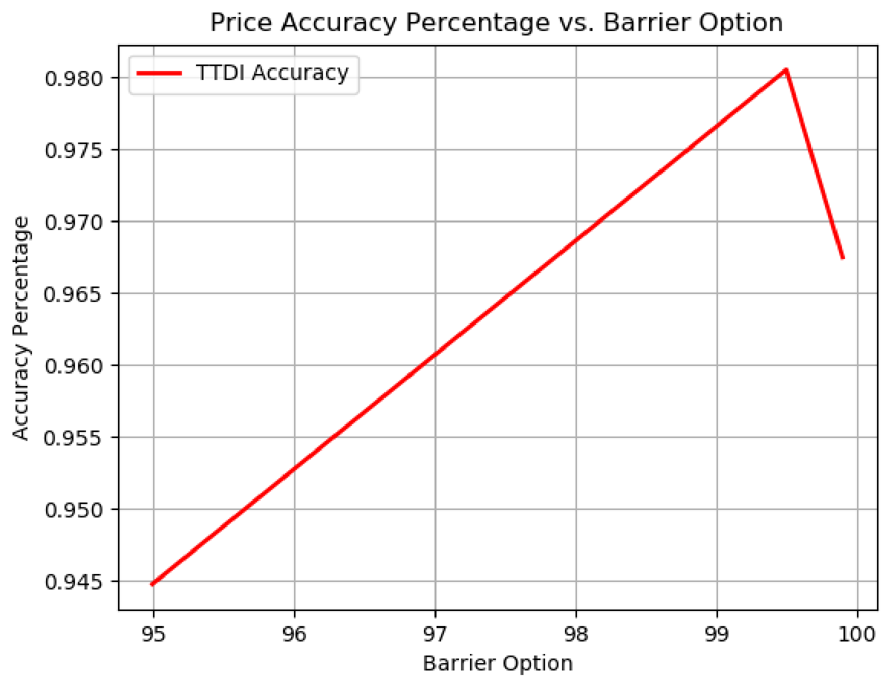
Trinomial Tree Call down-and-out option price is: 6.6995182962413296
Black-Scholes Call down-and-in option price is: 6.670701053402194
Black-Scholes Call down-and-out option price is: 5.517751786064505
Number of Periods = 20 – 200
Barrier = 95

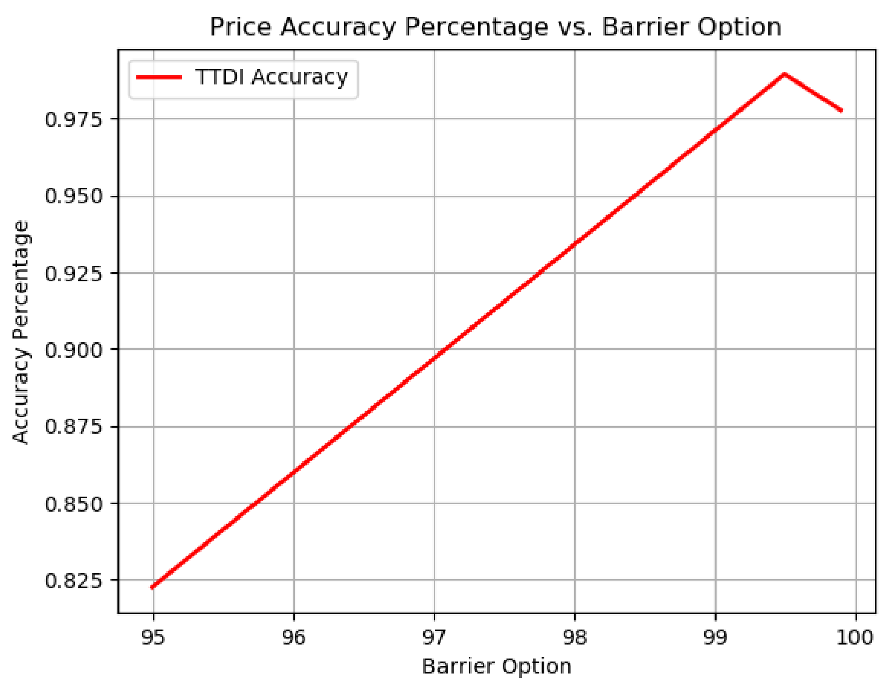
Trinomial Tree Call down-and-in option price from 20 - 200 periods is:
[7.434361100820392, 7.026238845484706, 6.740853454221703, 6.507243877204785, 6.302299349833451, 6.1155790775818994, 5.941448589771551, 5.782523128814383, 5.632976379930232]
Trinomial Tree Call down-and-out option price from 20 - 200 periods is:
[4.795135246830319, 5.189214938787796, 5.461917461780226, 5.686112134849451, 5.883842078586996, 6.064827489803288, 6.23426138384873, 6.395267419800518, 6.549883107319202]
Black-Scholes Call down-and-in option price from 20 - 200 periods is:
[6.670701053402194, 6.670701053402194, 6.670701053402194, 6.670701053402194, 6.670701053402194, 6.670701053402194, 6.670701053402194, 6.670701053402194, 6.670701053402194]
Black-Scholes Call down-and-out option price from 20 - 200 periods is:
[5.517751786064505, 5.517751786064505, 5.517751786064505, 5.517751786064505, 5.517751786064505, 5.517751786064505, 5.517751786064505, 5.517751786064505, 5.517751786064505]



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.

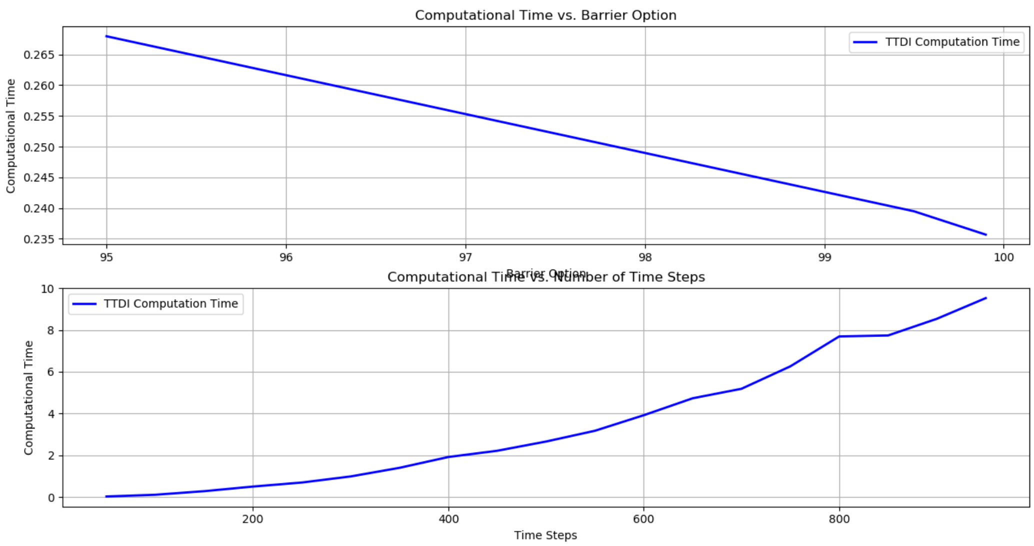
Pricing Accuracy vs. Barrier Option





These two figures show that the pricing accuracy is very close to 1, so it illustrates that Trinomial Tree is an effective model to price call option as well as Black-Scholes Method at time step = 100. We can also see that at barrier option of 95, the accuracy drops tremendously, at barrier option of 99.5, the accuracy reaches highest. However, if we increase the time step, the error at barrier option of 95 increases significantly, but the accuracy at barrier option of 99.5 and 99.9 also increases. It also concludes that the price error can occur at certain different barrier option in Trinomial Tree Model due to non-linearity error, quantization error, option specification error.

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 because once the option price hits the barrier, it can cancel the current iteration and start the next one, which saves some amount of computation time. However, the computation time is positively proportional to Time Step exponentially. The computation will be expensive when the time step reaches higher values though the option price can obtain greater accuracy.

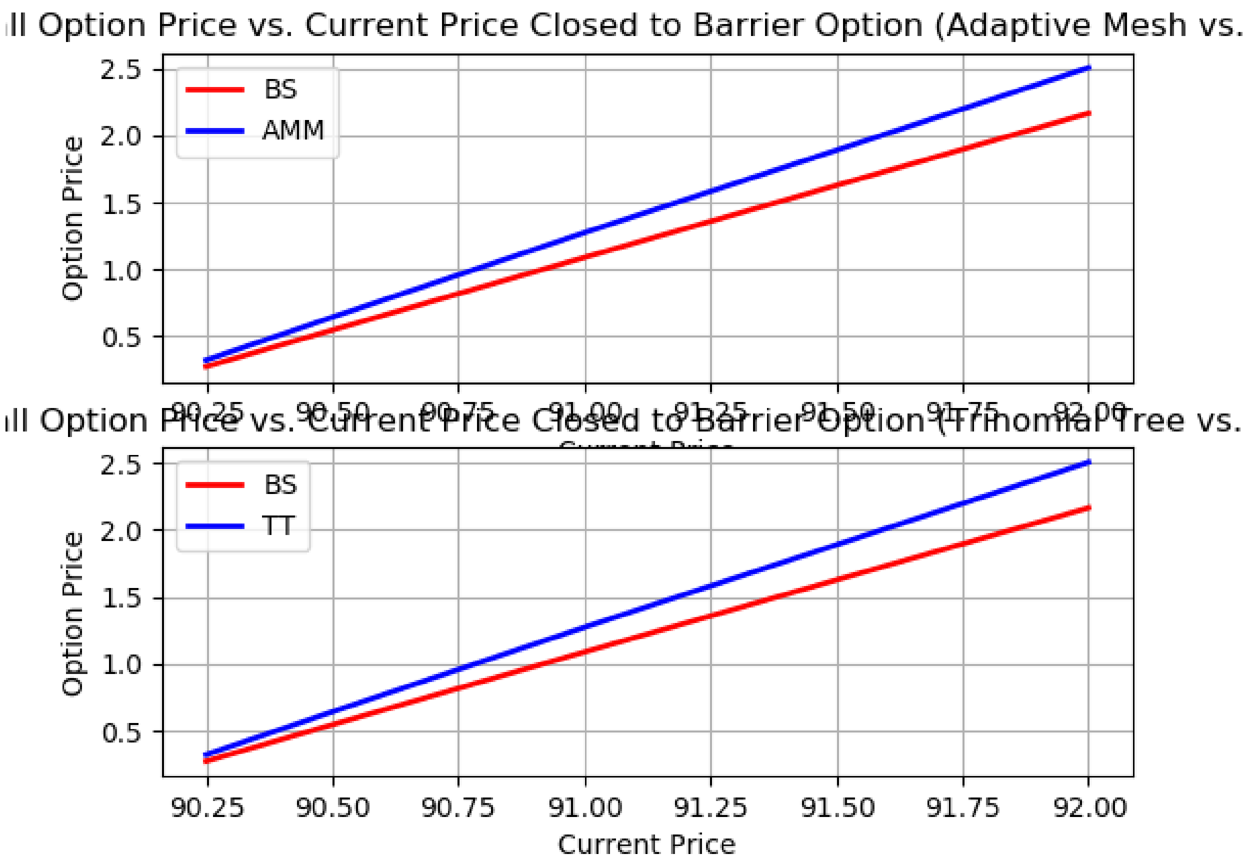
2. Implement the AMM for barrier options and replicate Table 3 on page 337 of the AMM paper.

Black-Scholes Call down-and-out option price from 92, 91, 90, 90.5, 90.25, 90.125 barrier is: [2.165500726801997, 1.089099169170618, 0.5462075328430895, 0.27352733609694724]

Trinomial Tree Call down-and-out option price from 92, 91, 90, 90.5, 90.25, barrier is: [2.506247701, 1.273847951, 0.642362906, 0.322585512]

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: [2.5062477009724997, 1.273829153886016, 0.6426128411960714, 0.32272016694809336, 0.16155856505222632]

The Black-Scholes Model generates some error.



The replicated table:

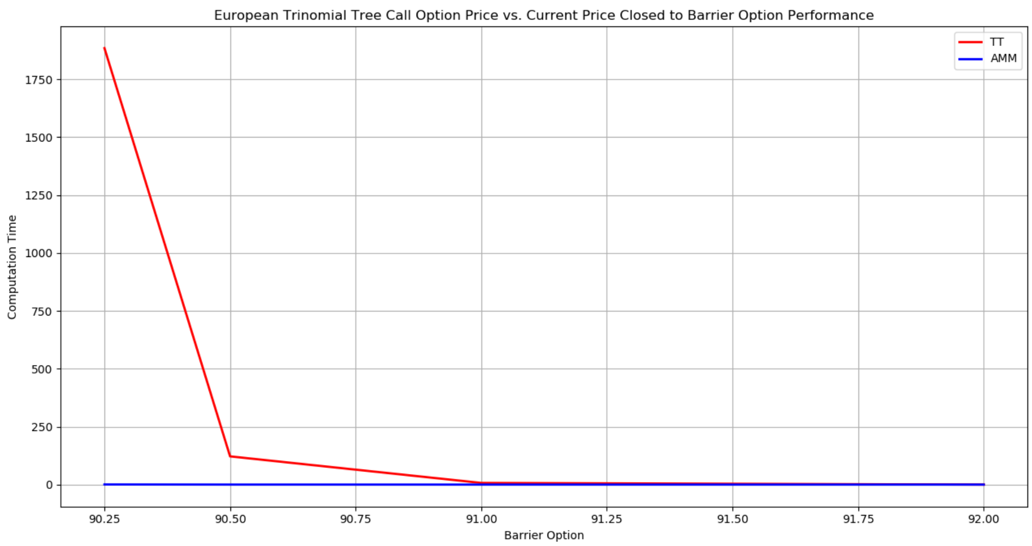
		RTM					AMM				
	Analytic Value	Value	Number of time step:	CPU time(s)			Value	AMM level	CPU time(s)	Delta	Gamma
92	2.506	2.50625	388	0.4947956			2.50625	0	0.584727287	N/A	N/A
91	1.274	1.27385	1535	7.6637075			1.27383	1	0.585361958	1.25295	-0.0144
90.5	0.642	0.64236	6108	122.32879			0.64261	2	0.668255091	1.27378	-0.0148
90.25	0.323	0.32259	24367	1884.5005			0.32272	3	1.183701277	1.28522	-0.0149
90.125	0.162	N/A	N/A	N/A			0.16156	4	5.763085127	1.29088	-0.0155

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 accurate option price very 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 exclude 90.125's computation to save large amount of time. In comparison, Adaptive Mesh performs much greater efficiency and higher accuracy than Trinomial Tree.

Trinomial Tree Call computation time from 92, 91, 90, 90.5, 90.25 barrier with 0,1,2,3,4 mesh level is: [0.494795561, 7.663707495, 122.3287914, 1884.500484]
Adaptive Mesh Call computation time from 92, 91, 90, 90.5, 90.25, 90.125 barrier with 0,1,2,3,4 mesh level is: [0.5913772583007812, 0.597357988357544, 0.6961963176727295, 1.1885857582092285, 5.763085126876831]
Adaptive Mesh Call Delta from 92, 91, 90, 90.5, 90.25, 90.125 barrier with 0,1,2,3,4 mesh level is: [0, 1.2529469455139473, 1.2737849749211472, 1.2852175209392578, 1.2908792763835915]
Adaptive Mesh Call Gamma from 92, 91, 90, 90.5, 90.25, 90.125 barrier with 0,1,2,3,4 mesh level is: [0, -0.014378329395382397, -

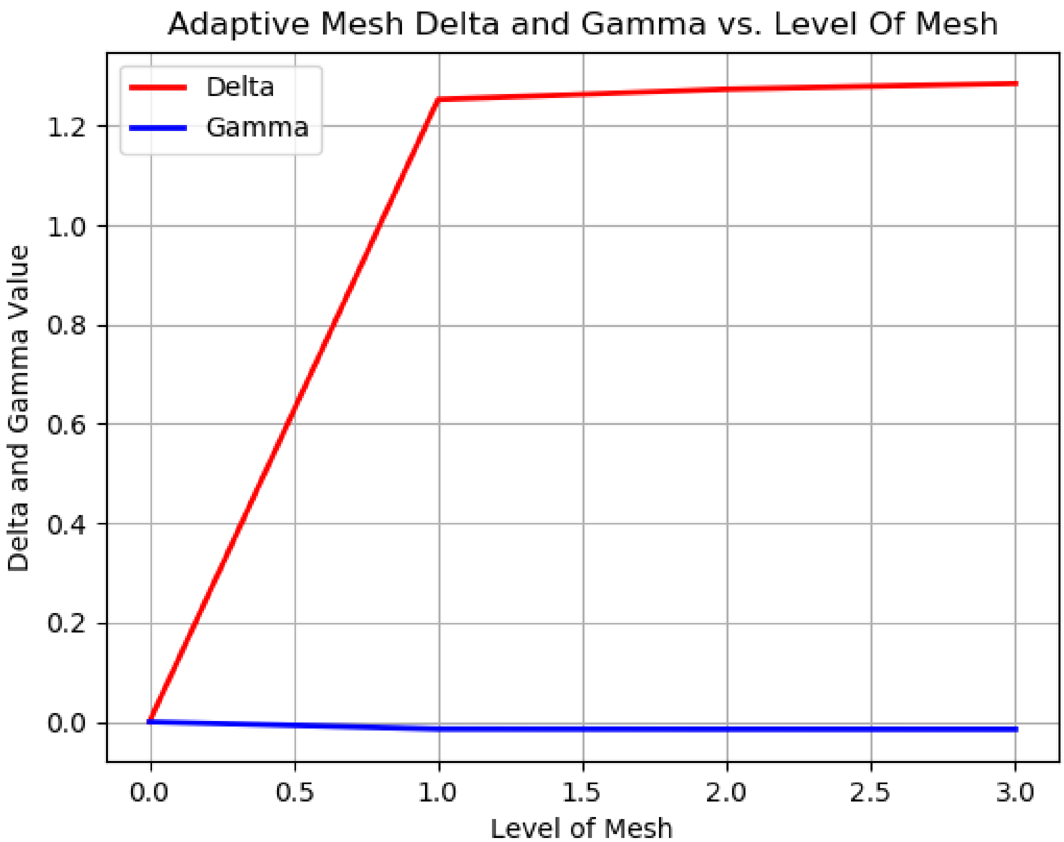
0.014772172719397535, -0.014927511039004271, -0.015464374131257489]



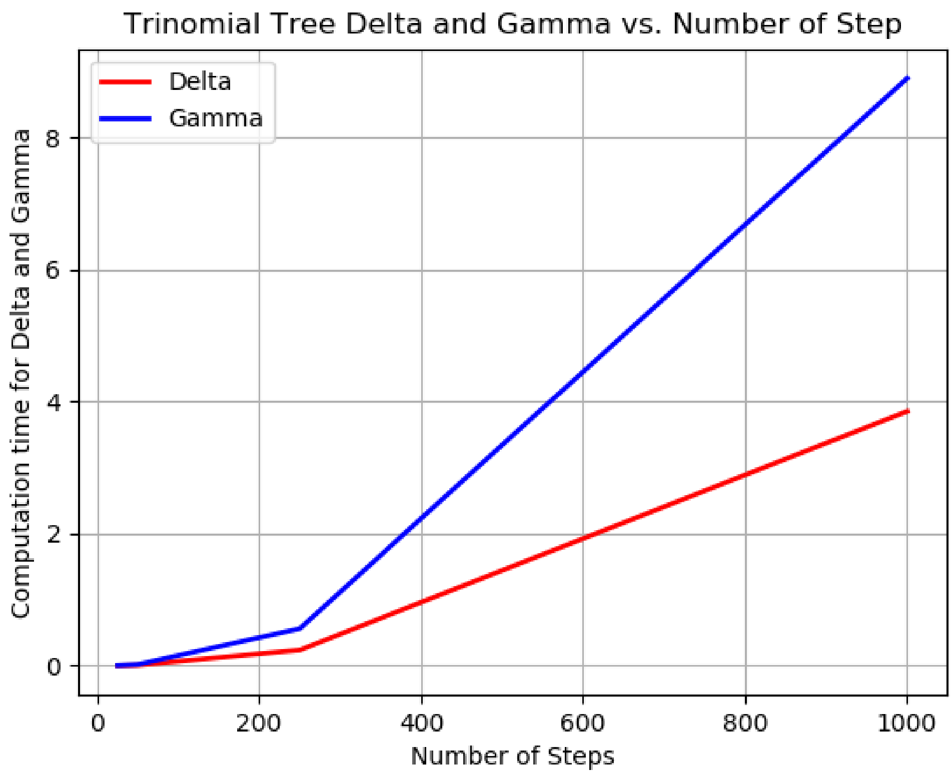
The above figure shows the significant difference of computation time between Trinomial Tree model and Adaptive Mesh Method to achieve similar accuracy, which reflects the table's content. As we can see from the raw data, the computation for 90.25 and 90.125 can be completed in 6s, Trinomial tree requires thousands of seconds to complete the calculation. This figure concludes that Adaptive Mesh Method is a better approach to generate accuracy result in reasonable amount of time.

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.

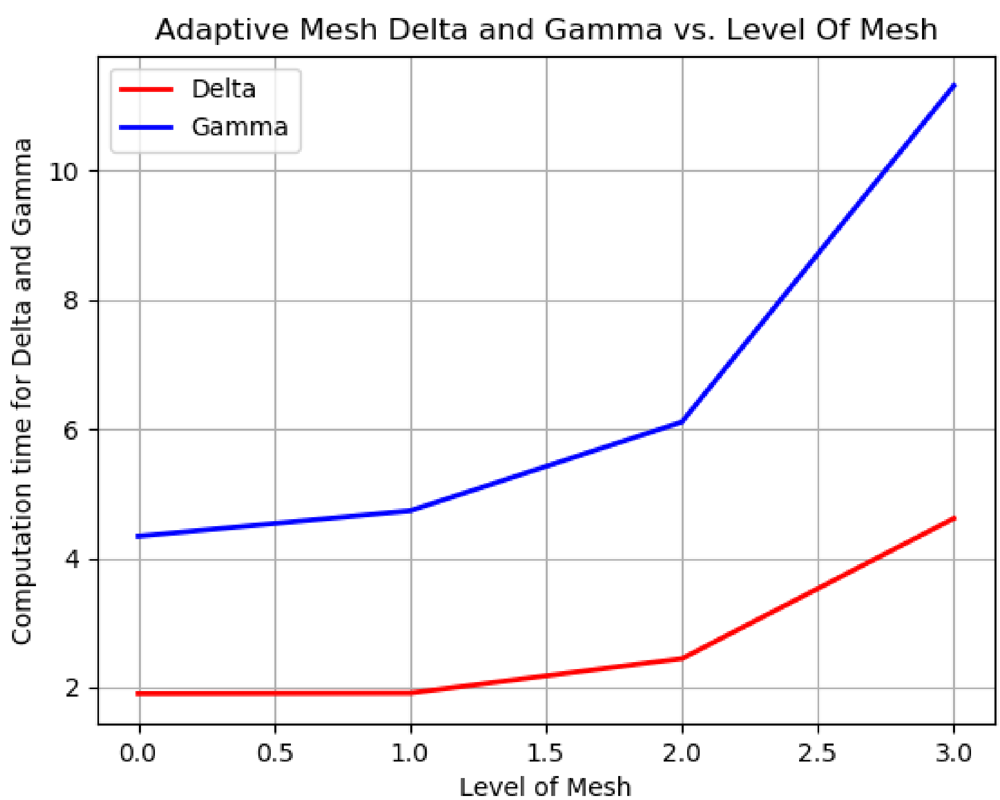
Adaptive Mesh Call Delta from 92, 91, 90, 90.5, 90.25, 90.125 barrier with 0,1,2,3,4 mesh level is: [0, 1.2529469455139473, 1.2737849749211472, 1.2852175209392578, 1.2908792763835915]
Adaptive Mesh Call Gamma from 92, 91, 90, 90.5, 90.25, 90.125 barrier with 0,1,2,3,4 mesh level is: [0, -0.014378329395382397, -0.014772172719397535, -0.014927511039004271, -0.015464374131257489]



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:
[0.0019934177398681640, 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]



Trinomial Tree Delta and Gamma vs. Number of Step
Delta
Gamma
Computation time for Delta and Gamma
Number of Steps
0
200
400
600
800
1000
0
2
4
6
8



Adaptive Mesh Delta and Gamma vs. Level Of Mesh
Delta
Gamma
Computation time for Delta and Gamma
Level of Mesh
0.0
0.5
1.0
1.5
2.0
2.5
3.0
2
4
6
8
10

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. Somehow, when the time step becomes larger, the accuracy does not increase. Instead, it drops. This error could be due to non-linearity error. 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 Black-Scholes Analytic Value is slightly different than theoretical values, it needs more improvement to close the numeric gap and reach greater accuracy. We rely on the paper’s formula [1] to compute Black-Scholes Analytic value, the computation algorithm might be incorrectly introducing or missing some important parameters. However, the computer configuration, 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 relatively expensive when it constructs large numbers of periods, requiring the complexity of n^2 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. The advantage of Restricted Trinomial Model is that this model can determine the number of step to perform and obtain optimal call option value. The advantage of Adaptive Mesh Method is to perform much higher efficiency than Restricted Trinomial Model.
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 libraries installed, the python code will be executable.