HW2

March 26, 2019

This file is the second homework of FE-621 class. The programming language is Python and this report is produced through Jupyter.

1 Problem 1. The Binomial Tree

1.1

In this quesiton, we need to construct a pattern that can be used as a general model for pricing the options through numerical ways. Then we will give an additive binomial tree construciton.

First of all, we need to define a class which represent the payoff of options, then we define the tree class to represent the tree method we may use in the following steps.

```
In [1]: import pandas as pd
        import math as m
        import numpy as np
        from scipy.stats import norm
        import matplotlib.pyplot as plt
        import os

        class Payoff(object):
            def __init__(self,Strike):
                self.Strike = Strike
            def getpayoff(self):
                pass
        """
        vanilla payoff funciton
        """
        class callpayoff(Payoff):
            def __init__(self,Strike):
                Payoff.__init__(self,Strike)
            def getpayoff(self,Price):
                return np.asarray([max(Price-self.Strike,0)])
            def getnodeprice(self,Price,Dis_price):
                return np.asarray(Dis_price)
            def getidentity(self):
                return "callpayoff"

        class putpayoff(Payoff):
```

```
    def __init__(self,Strike):
        Payoff.__init__(self,Strike)
    def getpayoff(self,Price):
        return np.asarray([max(self.Strike-Price,0)])
    def getnodeprice(self,Price,Dis_price):
        return np.asarray(Dis_price)
    def getidentity(self):
        return "putpayoff"
##########################################################
class tree():
    def __init__(self,T,S,r,sigma,N,payoff,D):
        self.T = T
        self.S = S
        self.r = r
        self.sigma = sigma
        self.N = N
        self.payoff = payoff
        self.D = D
    def build_tree(self):
        pass

class additive_binomial_tree(tree):
    def __init__(self,T,S,r,sigma,N,payoff,D):
        tree.__init__(self,T,S,r,sigma,N,payoff,D)
    def build_tree(self):
        self.delta_t = self.T/self.N
        self.nu = self.r-self.D-0.5*self.sigma**2
        self.x_u = m.sqrt(self.delta_t*self.sigma**2+self.nu**2*self.delta_t**2)
        self.x_d = -self.x_u
        self.p_u = 0.5+0.5*((self.nu*self.delta_t)/self.x_u)
        self.p_d = 1-self.p_u
        self.disc = np.exp(-self.r*self.delta_t)
        self.St = self.S*np.exp(np.asarray([i*self.x_d+(self.N-i)*self.x_u\
                                    for i in range(self.N+1)]))
        self.C = np.asarray([self.payoff.getpayoff(p) for p in self.St])
    def euro_discount(self):
        self.build_tree()
        while (len(self.C)>1):
            # compute discounted value of product
            self.dis_C = self.disc*(self.p_u*self.C[:-1]+self.p_d*self.C[1:])
            # compute stock price at that node
            self.St = np.exp(self.x_d)*self.St[:-1]
            # apply the condition on node
            self.C = np.asarray([self.payoff.getnodeprice(self.St[i],self.dis_C[i])\
                                for i in range(len(self.St))])
        return self.C[0]
    def amer_discount(self):
        self.build_tree()
```

```
            while (len(self.C)>1):
                self.dis_C = self.disc*(self.p_u*self.C[:-1]+self.p_d*self.C[1:])
                self.St = np.exp(self.x_d)*self.St[:-1]
                self.exc_C = np.asarray([self.payoff.getpayoff(p) for p in self.St])
                self.dis_C = np.asarray([self.payoff.getnodeprice(self.St[i],self.dis_C[i])
                                         for i in range(len(self.St))])
                self.C = np.where(self.dis_C < self.exc_C, self.exc_C, self.dis_C)
            return self.C[0]
```

The payoff class contains all the information about a specific option. The tree class contains three baisc methods which is build_tree, euro_discount and amer_discount which is used to define what kind of tree you want to construct. Here we construct an additive binomial tree.

1.2

Here we use the historical data and binomial tree to price the otions by applying the implied volatility we calculated before. We take the DATA2 which is a data set for the last assignment and we use the AMZN options which have 3 kinds of strike price and different type are seperated.

```
In [5]: os.chdir(r'D:\Grad 2\621\assignment\HW2\data')
        Data2 = pd.read_csv('data12.csv',index_col=0)
        Data2 = Data2.loc[:1493,:]
        Data2 = Data2[(Data2.loc[:,"Expiry"] == "2019-02-22") | \
                      (Data2.loc[:,"Expiry"] == "2019-03-22")\
                      | (Data2.loc[:,"Expiry"] == "2019-04-18")]
        Data2 = Data2[(Data2['Strike']/Data2['Underlying_Price_y']>0.95) & \
                      (Data2['Strike']/Data2['Underlying_Price_y']<1.05)]
        Data2 = Data2.sort_values(by = ["Expiry","Strike"],ascending=(True,True))
        Data2_call = Data2[Data2.Type == "call"]
        Data2_put = Data2[Data2.Type == "put"]
        r=0.024
        for ind1 in Data2_call.index:
            Data2_call.loc[ind1,"Tree_Euro_Price"] = \
            additive_binomial_tree(Data2_call.loc[ind1,"TtM_y"],
                          Data2_call.loc[ind1,"Underlying_Price_y"],r,
                          Data2_call.loc[ind1,"Implied_vol_bis"],
                          400,callpayoff(Data2_call.loc[ind1,"Strike"]),D=0).euro_discount()
            Data2_call.loc[ind1,"Tree_Amer_Price"] = \
            additive_binomial_tree(Data2_call.loc[ind1,"TtM_y"],
                          Data2_call.loc[ind1,"Underlying_Price_y"],r,
                          Data2_call.loc[ind1,"Implied_vol_bis"],
                          400,callpayoff(Data2_call.loc[ind1,"Strike"]),D=0).amer_discount()
        for ind2 in Data2_put.index:
            Data2_put.loc[ind2,"Tree_Euro_Price"] = \
            additive_binomial_tree(Data2_put.loc[ind2,"TtM_y"],
                          Data2_put.loc[ind2,"Underlying_Price_y"],r,
                          Data2_put.loc[ind2,"Implied_vol_bis"],
                          400,putpayoff(Data2_put.loc[ind2,"Strike"]),D=0).euro_discount()
```

```
            Data2_put.loc[ind2,"Tree_Amer_Price"] =\
            additive_binomial_tree(Data2_put.loc[ind2,"TtM_y"],
                                   Data2_put.loc[ind2,"Underlying_Price_y"],r,
                                   Data2_put.loc[ind2,"Implied_vol_bis"],
                                   400,putpayoff(Data2_put.loc[ind2,"Strike"]),D=0).amer_discount()
        Data2_call.index = range(len(Data2_call))
        Data2_put.index = range(len(Data2_put))
```

```
D:\Anaconda3\lib\site-packages\pandas\core\indexing.py:362: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html
  self.obj[key] = _infer_fill_value(value)
D:\Anaconda3\lib\site-packages\pandas\core\indexing.py:543: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html
  self.obj[item] = s
```

In [4]:
```
Data2_call.head()
```

Out[4]:

	Strike	Expiry	Type	Last_y	Bid_y	Ask_y	Vol_y	Underlying_Price_y
0	1550.0	2019-02-22	call	88.00	85.25	86.90	14.0	1629.09
1	1552.5	2019-02-22	call	82.25	82.65	83.80	1.0	1629.09
2	1555.0	2019-02-22	call	80.10	81.50	83.10	1.0	1629.09
3	1557.5	2019-02-22	call	77.95	79.35	80.05	1.0	1629.09
4	1560.0	2019-02-22	call	77.42	75.80	77.10	23.0	1629.09

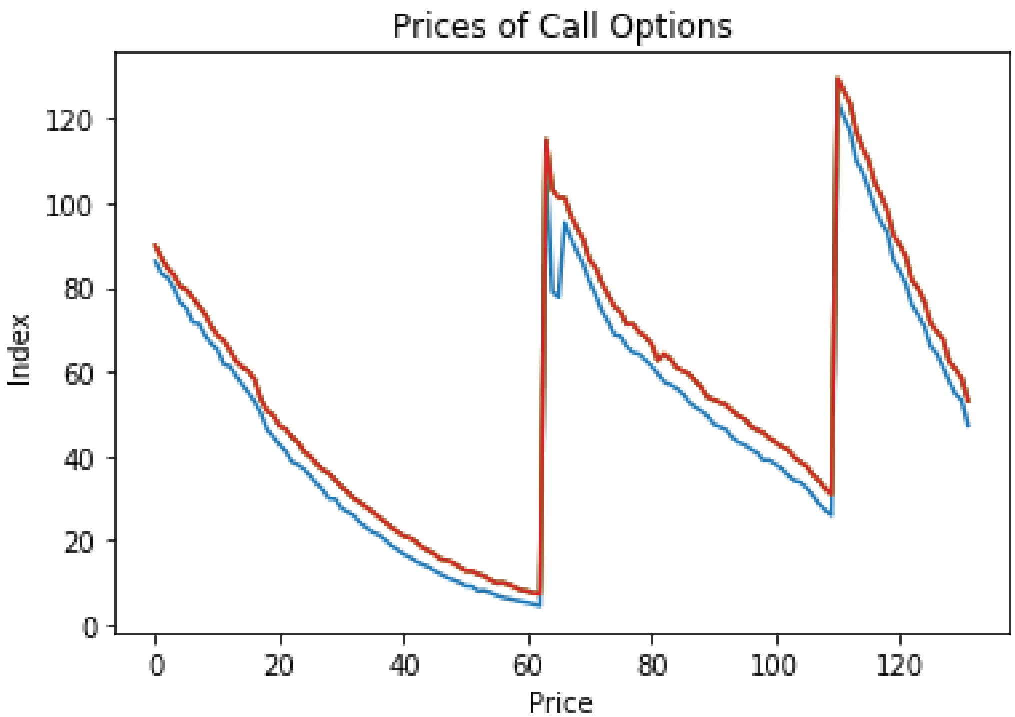
	TtM_y	Implied_vol_bis	BS_Price	Avr_Price	Tree_Euro_Price
0	0.027397	0.349519	89.826536	86.075	89.837318
1	0.027397	0.336308	86.879251	83.225	86.888144
2	0.027397	0.330599	84.469596	82.300	84.481063
3	0.027397	0.336125	82.898593	79.700	82.888293
4	0.027397	0.327110	80.258296	76.450	80.240095

	Tree_Amer_Price
0	89.837318
1	86.888144
2	84.481063
3	82.888293
4	80.240095

In [9]:
```
plt.plot(Data2_call["Avr_Price"])
plt.plot(Data2_call["BS_Price"])
plt.plot(Data2_call["Tree_Euro_Price"])
plt.plot(Data2_call["Tree_Amer_Price"])
```

```
plt.xlabel('Index')
plt.ylabel('Price')
plt.title('Prices of Call Options')
```

Out[9]: Text(0.5,1,'Prices of Call Options')



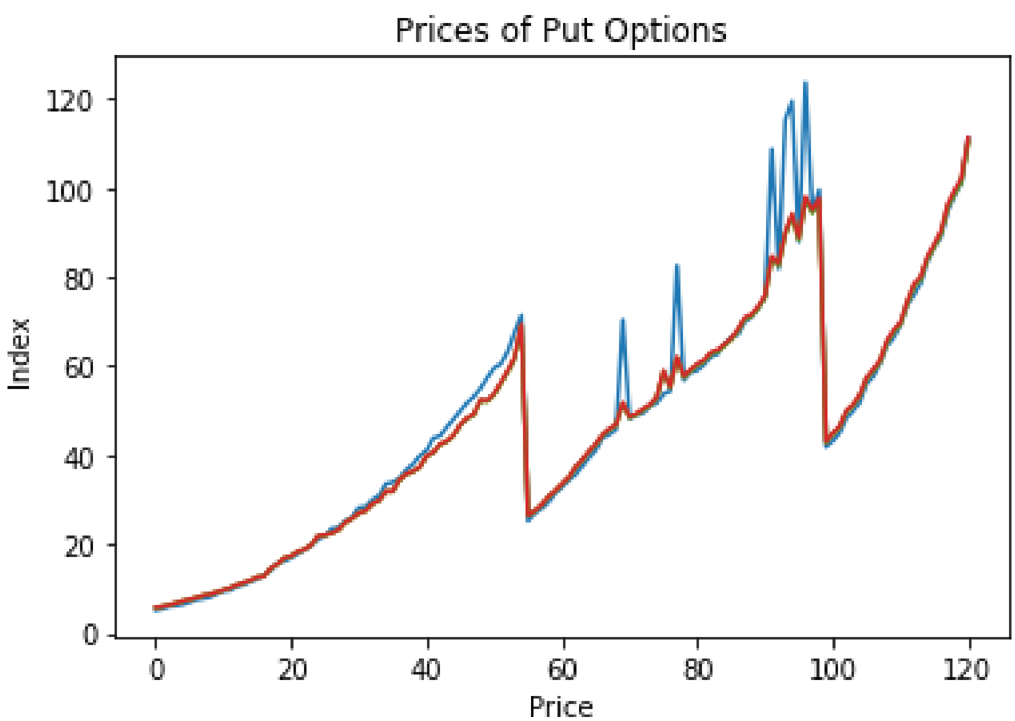
In [3]: Data2_put.head()

```
Out[3]:    Strike      Expiry Type  Last_y  Bid_y  Ask_y  Vol_y  Underlying_Price_y  \
        0  1550.0  2019-02-22  put    5.35   5.10    5.3  174.0             1629.09
        1  1552.5  2019-02-22  put    6.76   5.40    5.6    9.0             1629.09
        2  1555.0  2019-02-22  put    6.01   5.90    6.2   16.0             1629.09
        3  1557.5  2019-02-22  put    5.95   6.05    6.3    7.0             1629.09
        4  1560.0  2019-02-22  put    6.05   6.35    6.6   36.0             1629.09

              TtM_y  Implied_vol_bis  BS_Price  Avr_Price  Tree_Euro_Price  \
        0  0.027397         0.291025  5.826281      5.200         5.830749
        1  0.027397         0.287859  6.021406      5.500         6.014818
        2  0.027397         0.286814  6.360619      6.050         6.360295
        3  0.027397         0.286520  6.766204      6.175         6.774527
        4  0.027397         0.286938  7.241026      6.475         7.235889

           Tree_Amer_Price
        0         5.837920
```

```
1        6.022415
2        6.368661
3        6.783302
4        7.245364
```

```
In [10]: plt.plot(Data2_put["Avr_Price"])
         plt.plot(Data2_put["BS_Price"])
         plt.plot(Data2_put["Tree_Euro_Price"])
         plt.plot(Data2_put["Tree_Amer_Price"])
         plt.xlabel('Index')
         plt.ylabel('Price')
         plt.title('Prices of Put Options')
```

```
Out[10]: Text(0.5,1,'Prices of Put Options')
```



1.3

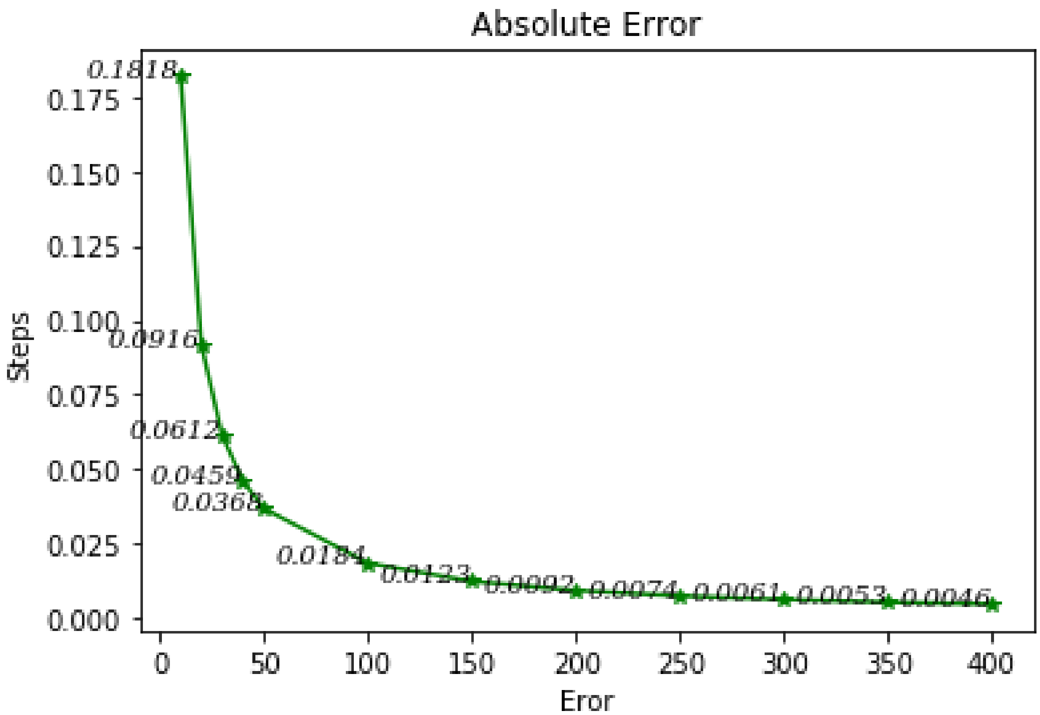
As we can see in these two tables, the price calculated by BS formula and treee with 400 steps are very close to each other.

The same conclusion can be drawn by seeing these two plots. Each plot has 2 "jumps" whihc is caused by the different maturities. And when the maturities go up, the average price of the options are going up too, which is very intuitive. We can see the blue line in two plots is the bid and ask values and the red line is other prices we calculated. It seems that there are only two lines, however 4 lines are drawn but those three lines are too close to see the differences. These also illustrate that what we said before, that is BS method and tree method can present almost same results.

1.4

Let us compute the differences or in another word, absolute error of the tree method when the steps increase. And we plot them to give a conclusion.

```
In [11]: def BS_Formula(type_opt, r, vol, K, S, T, q=0):
             d_1 = float(m.log(S/K)+(r-q+vol**2/2)*T)/float(vol*m.sqrt(T))
             d_2 = d_1-vol*m.sqrt(T)
             if type_opt == 'call':
                 return norm.cdf(d_1)*S*m.exp(-q*T)-K*m.exp(-r*T)*norm.cdf(d_2)
             else:
                 return K*m.exp(-r*T)*norm.cdf(-d_2)-norm.cdf(-d_1)*S*m.exp(-q*T)

         N = [10,20,30,40,50,100,150,200,250,300,350,400]
         diff = []
         for n in N:
             P_bs = BS_Formula("put",0.06,0.2,100,100,1)
             P_bt = additive_binomial_tree(1,100,0.06,0.2,n,putpayoff(100),D=0).euro_discount(
             diff.append(abs(P_bs-P_bt))

         plt.plot(N,diff,'g*-')
         plt.xlabel('Eror')
         plt.ylabel('Steps')
         plt.title('Absolute Error')
         for i in range(0,len(N)):
             plt.text(N[i],diff[i],str(round(diff[i],4)),
                      family='serif', style='italic', ha='right', wrap=True)
```



As we can see, the absolute erroe decrease when the number of steps increase.

1.5 Bonus

In this problem we need to get the implied volatilities by using the bisection method with a tree kernal. This process is very similar to the

In [103]:
```
def Bisection(func,tolerance,up,down):
    if np.sign(func(down)) * np.sign(func(up)) > 0:
        return np.nan
    if abs(func(up))<tolerance:
        return up
    if abs(func(down))<tolerance:
        return down
    mid = (down + up)/2
    while ( abs(func(mid)) > tolerance ):
        if ( np.sign(func(down)) * np.sign(func(mid)) < 0 ):
            up = mid
        else:
            down = mid
        mid = (down + up)/2
    return mid

def get_avrprice(bid,ask):
    return 0.5*(bid+ask)
```

```
def get_iv_tree(type_opt, r, K, S, T, P, N, tolerance,up,down):
    if type_opt == "call":
        obj_func = lambda x: additive_binomial_tree(T,S,r,x,N,
                                                    callpayoff(K),D=0).amer_discount
    else:
        obj_func = lambda x: additive_binomial_tree(T,S,r,x,N,
                                                    putpayoff(K),D=0).amer_discount(
    return Bisection(obj_func,tolerance,up,down)

def get_iv_bs(type_opt, r, K, S, T, P,tolerance,up,down):
    obj_func= lambda x: BS_Formula(type_opt, r, x, K, S, T)-P
    return Bisection(obj_func,tolerance,up,down)

for ind3 in Data2.index[:50]:
    Data2.loc[ind3,"IV_tree"] = get_iv_tree(Data2.loc[ind3,"Type"], r,
             Data2.loc[ind3,"Strike"],
             Data2.loc[ind3,"Underlying_Price_y"], Data2.loc[ind3,"TtM_y"],
             get_avrprice(Data2.loc[ind3,"Bid_y"],\
                          Data2.loc[ind3,"Ask_y"]), 200, 10**-6,1,0.01)
    Data2.loc[ind3,"IV_bs"] = get_iv_bs(Data2.loc[ind3,"Type"], r,
             Data2.loc[ind3,"Strike"], Data2.loc[ind3,"Underlying_Price_y"],
                                       Data2.loc[ind3,"TtM_y"],
             get_avrprice(Data2.loc[ind3,"Bid_y"],Data2.loc[ind3,"Ask_y"]),
                                       10**-6,1,0.01)
Data2.head()
```

Out[103]:

	Strike	Expiry	Type	Last_y	Bid_y	Ask_y	Vol_y	\
450	1550.0	2019-02-22	call	88.00	85.25	86.9	14.0	
451	1550.0	2019-02-22	put	5.35	5.10	5.3	174.0	
461	1552.5	2019-02-22	call	82.25	82.65	83.8	1.0	
462	1552.5	2019-02-22	put	6.76	5.40	5.6	9.0	
463	1555.0	2019-02-22	call	80.10	81.50	83.1	1.0	

	Underlying_Price_y	TtM_y	Implied_vol_bis	BS_Price	Avr_Price	\
450	1629.09	0.027397	0.349519	89.826536	86.075	
451	1629.09	0.027397	0.291025	5.826281	5.200	
461	1629.09	0.027397	0.336308	86.879251	83.225	
462	1629.09	0.027397	0.287859	6.021406	5.500	
463	1629.09	0.027397	0.330599	84.469596	82.300	

	IV_tree	IV_bs
450	0.270785	0.270671
451	0.283680	0.284257
461	0.258467	0.258122
462	0.282724	0.283162
463	0.279665	0.279468

In [104]:
```
Data2.head()
```

```
Out[104]:       Strike      Expiry  Type  Last_y  Bid_y  Ask_y  Vol_y  \
          450  1550.0  2019-02-22  call   88.00  85.25   86.9   14.0
          451  1550.0  2019-02-22   put    5.35   5.10    5.3  174.0
          461  1552.5  2019-02-22  call   82.25  82.65   83.8    1.0
          462  1552.5  2019-02-22   put    6.76   5.40    5.6    9.0
          463  1555.0  2019-02-22  call   80.10  81.50   83.1    1.0

               Underlying_Price_y     TtM_y  Implied_vol_bis   BS_Price  Avr_Price  \
          450             1629.09  0.027397         0.349519  89.826536     86.075
          451             1629.09  0.027397         0.291025   5.826281      5.200
          461             1629.09  0.027397         0.336308  86.879251     83.225
          462             1629.09  0.027397         0.287859   6.021406      5.500
          463             1629.09  0.027397         0.330599  84.469596     82.300

                IV_tree     IV_bs
          450  0.270785  0.270671
          451  0.283680  0.284257
          461  0.258467  0.258122
          462  0.282724  0.283162
          463  0.279665  0.279468
```

The columns "IV_tree" and "IV_bs" are the impllied volatilities we computed. The results of implied volatilities calculated from two methods are very close to each other. The outcomes are not surprising to us because we can give the right price of a option by giving the same parameters. It is quite natural to think that the tree funciton can inversely give the right answer when applying the same numerical method as we do on BS function.

2 Problem 2. The Trinomial Tree

2.1

Construct a trinomial tree class which is very similar to the binomial tree, however we need input "dx" which refers to the little change on the log return of underlying asset. Note that the dx must satisify the condition of $dx \geq \sigma\sqrt{3\Delta t}$. The code is presented following.

```
In [14]: class additive_trinomial_tree(tree):
             def __init__(self,T,S,r,sigma,N,payoff,dx,D):
                 tree.__init__(self,T,S,r,sigma,N,payoff,D)
                 self.dx = dx
             def build_tree(self):
                 self.delta_t = self.T/self.N
                 self.nu = self.r-self.D-0.5*self.sigma**2
                 self.p_u = 0.5*((self.sigma**2*self.delta_t+self.nu**2*self.delta_t**2)/\
                                 (self.dx**2)+(self.nu*self.delta_t)/self.dx)
                 self.p_m = 1-(self.sigma**2*self.delta_t+self.nu**2*self.delta_t**2)/\
                 (self.dx**2)
                 self.p_d = 0.5*((self.sigma**2*self.delta_t+self.nu**2*self.delta_t**2)/\
                                 (self.dx**2)-(self.nu*self.delta_t)/self.dx)
```

```
            self.disc = np.exp(-self.r*self.delta_t)
            self.St = self.S*np.exp(np.asarray([self.N*self.dx-i*self.dx\
                                    for i in range(2*self.N+1)]))
            self.C = np.asarray([self.payoff.getpayoff(p) for p in self.St])
        def euro_discount(self):
            self.build_tree()
            while (len(self.C)>1):
                self.C = self.disc*(self.p_u*self.C[:-2]+\
                                    self.p_m*self.C[1:-1]+self.p_d*self.C[2:])
            return self.C[0]
        def amer_discount(self):
            self.build_tree()
            while (len(self.C)>1):
                self.C = self.disc*(self.p_u*self.C[:-2]+\
                                    self.p_m*self.C[1:-1]+self.p_d*self.C[2:])
                self.St = self.St[1:-1]
                self.C_exc = np.asarray([self.payoff.getpayoff(p) for p in self.St])
                self.C = np.where( self.C < self.C_exc, self.C_exc, self.C)
            return self.C[0]
```

2.2

In this question we compute the American and European options using the trinomial tree and given input which is $S_0 = 100$, $K = 100$, $T = 1$, $\sigma = 25\%$, $r = 6\%$, $\delta = 0.03$. What's more we also compute the price through BS method and check the absolute error again. Note that dx here must larger than 0.02165 by given the N = 400.

First of all, we need to compute the prices.

```
In [16]: price1 = additive_trinomial_tree(1,100,0.06,0.25,400,
                                    callpayoff(100),0.022,0.03).euro_discount()[0]
         price2 = BS_Formula("call",0.06,0.25,100,100,1,0.03)
         price3 = additive_binomial_tree(1,100,0.06,0.25,400,
                                    callpayoff(100),D=0.03).euro_discount()[0]
         price4 = additive_trinomial_tree(1,100,0.06,0.25,400,
                                    putpayoff(100),0.022,0.03).euro_discount()[0]
         price5 = BS_Formula("put",0.06,0.25,100,100,1,0.03)
         price6 = additive_binomial_tree(1,100,0.06,0.25,400,
                                    putpayoff(100),D=0.03).euro_discount()[0]

         price_result1 =  pd.DataFrame([[price1,price2,price3],
                                    [price4,price5,price6]],
                                    index =["European Call","European Put"],
                                    columns = ["Trinomial","BS","Binomial"])
         print(price_result1)
```

```
               Trinomial         BS   Binomial
European Call  11.006925  11.013079  11.007052
European Put    8.138822   8.144979   8.139025
```

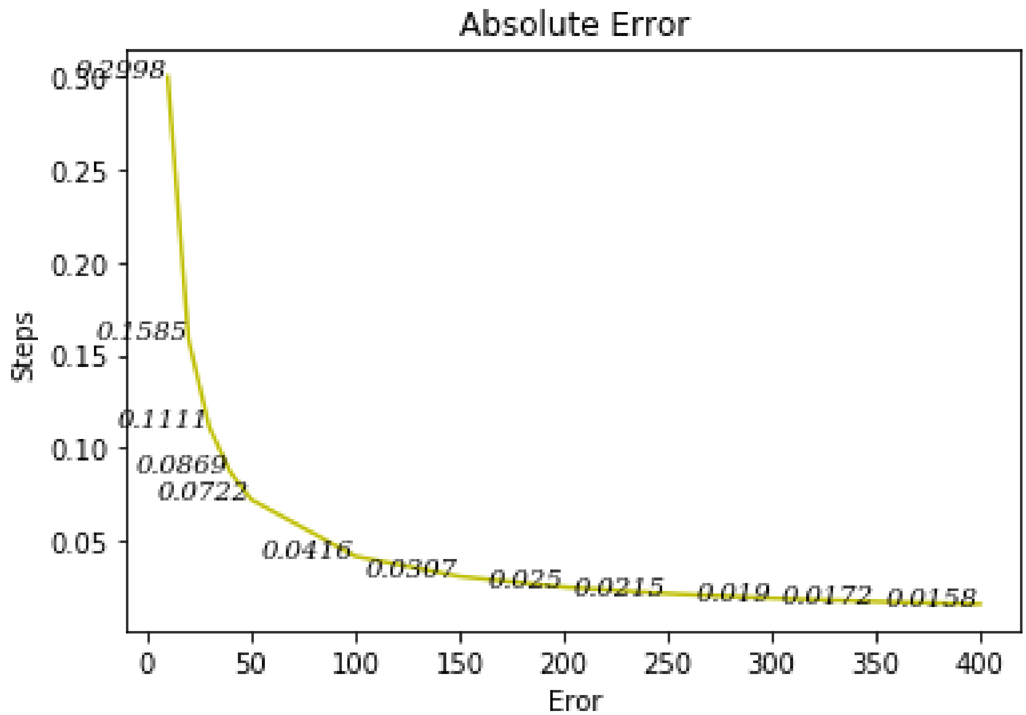
When computing the European options, binomial and trinomial tree methods' results are very close to each other and they are not far away fom the BS mothods. The little biase here is caused by steps, we can get more accurate price by increasing the steps of tree.

```
In [17]: price7 = additive_trinomial_tree(1,100,0.06,0.25,400,
                                         callpayoff(100),0.022,0.03).amer_discount()[0]
         price8 = additive_binomial_tree(1,100,0.06,0.25,400,
                                        callpayoff(100),D=0.03).amer_discount()[0]
         price9 = additive_trinomial_tree(1,100,0.06,0.25,400
                                         ,putpayoff(100),0.022,0.03).amer_discount()[0]
         price10 = additive_binomial_tree(1,100,0.06,0.25,400,
                                         putpayoff(100),D=0.03).amer_discount()[0]
         price_result2 =  pd.DataFrame([[price7,price8],
                                       [price9,price10]],index =["European Call","European Put
                                       columns = ["Trinomial","Binomial"])

         print(price_result2)

              Trinomial   Binomial
European Call  11.007075  11.007198
European Put    8.505841   8.508701
```

When computing the American options, the binomial tree and trinomial tree method give almost indentical results.

Secondly, we do the loop to find how the absolute error goes when the steps increase. We must note again that the dx term must satisfy the condition.

```
In [34]: N = [10,20,30,40,50,100,150,200,250,300,350,400]
         diff = []
         for n in [10,20,30,40,50,100,150,200,250,300,350,400]:
             P_bs = BS_Formula("put",0.06,0.25,100,100,1,0.03)
             P_bt = additive_trinomial_tree(1,100,0.06,0.25,n,putpayoff(100),0.25*\
                                             m.sqrt(3*1/n)+0.01,0.03).euro_discount()[0]
             diff.append(abs(P_bs-P_bt))
         plt.plot(N,diff,'y*-')
         plt.xlabel('Eror')
         plt.ylabel('Steps')
         plt.title('Absolute Error')
         for i in range(0,len(N)):
             plt.text(N[i],diff[i],str(round(diff[i],4)), family='serif',
                      style='italic', ha='right', wrap=True)
```



This plot is similar to the plot we draw previously, which also indicate that when the steps increase the absolute error decrease.

3 Problem 3. Pricing Exotic Options

3.1

In this question, we are asked to construct a tree to fit the path dependent options. However, due to the special construction of our class, the only thing we need to do is write a new paoff class which contains the information that add the constrains on each nodes in the tree.

Here we construct this barrier option's payoff class.

```
In [23]: class up_out_callpayoff(Payoff):
             def __init__(self,Strike,Barrier):
                 Payoff.__init__(self,Strike)
                 self.Barrier = Barrier
             def getpayoff(self,Price):
                 if Price >= self.Barrier:
                     return np.asarray([0])
                 else:
                     return np.asarray([max(Price-self.Strike,0)])
             def getnodeprice(self,Price,Dis_price):
                 if Price >=self.Barrier:
                     return np.asarray([0])
```

```
            else:
                return np.asarray(Dis_price)
        def getidentity(self):
            return "callpayoff"

    additive_binomial_tree(0.3,10,0.01,0.2,8000,
                           up_out_callpayoff(Strike = 10,Barrier = 11),D=0).euro_discount
```

```
Out[23]: 0.05344151138485562
```

So the price of this European Up-and-out call option is $0.05344.

It is necessary to talk about this payoff class. We need to input the strike price and barrier of this option and we define the getpayoff function, getnodeprice function and getidentity function. The getpayoff function is used to get the execute price by giving the underlying price and strikeprice. The getnodeprice function is about to add the constrain on the node price which is used to get the real discounted price. The last function is give the information about the option contract which can be used in else where.

3.2

In this question, we give the European barrier options' price by conducting the BS formula.

```
In [101]: def call_ui(r, vol, K, S, T, H, q=0):
              v = r-q-vol**2/2
              def C_bs(x1,x2):
                  return BS_Formula("call",r,vol,x2,x1,T,q)
              def P_bs(x1,x2):
                  return BS_Formula("put",r,vol,x2,x1,T,q)
              def d_bs(x1,x2):
                  return ((m.log(x1/x2)+v*T)/(vol*m.sqrt(T)))
              UI_bs = ((H/S)**(2*v/vol**2))*(P_bs(H**2/S,K)-P_bs(H**2/S,H)+\
                                            (H-K)*m.exp(-r*T)*norm.cdf(-d_bs(H,S)))+\
                                            C_bs(S,H)+(H-K)*m.exp(-r*T)*norm.cdf(d_bs(S,H))
              return UI_bs

          def call_uo(r, vol, K, S, T, H, q):
              v = r-q-vol**2/2
              def d_bs(x1,x2):
                  return ((m.log(x1/x2)+v*T)/(vol*m.sqrt(T)))
              def C_bs(x1,x2):
                  return BS_Formula("call",r,vol,x2,x1,T,q)
              UO_bs = C_bs(S,K)-C_bs(S,H)-(H-K)*m.exp(-r*T)*norm.cdf(d_bs(S,H))-\
                      ((H/S)**(2*v/vol**2))*(C_bs(H**2/S,K)-C_bs(H**2/S,H)-(H-K)*\
                      m.exp(-r*T)*norm.cdf(d_bs(H,S)))
              return UO_bs

          def call_di(r, vol, K, S, T, H, q):
              v = r-q-vol**2/2
              def C_bs(x1,x2):
```

```
        return BS_Formula("call",r,vol,x2,x1,T,q)
    DI_bs = (H/S)**(2*v/vol**2)*C_bs(H**2/S,K)
    return DI_bs

def call_do(r, vol, K, S, T, H, q):
    v = r-q-vol**2/2
    def C_bs(x1,x2):
        return BS_Formula("call",r,vol,x2,x1,T,q)
    DI_bs = C_bs(S,K)-(H/S)**(2*v/vol**2)*C_bs(H**2/S,K)
    return DI_bs
uo = call_uo(r=0.01, vol=0.2, K=10, S=10, T=0.3, H=11, q=0)
print(uo)
```

```
0.05309279660325303
```

Using the BS formula, we get the option with price of 0.05309. The two results are quite near to each other, however we have to give a very large number of step to get a comparable accurate price.

3.3

Two methods are applied in this question, the first one is using the BS function to directly calculate the price, and the second method is also use the BS function, while we use them to compute the out-and-put call option and vanilla call option then use the in-out parity to find the up-and-in call option price.

```
In [24]: # First approach
         ui = call_ui(r=0.01, vol=0.2, K=10, S=10, T=0.3, H=11, q=0)
         # Second approach
         c = BS_Formula("call",r=0.01, vol=0.2, K=10, S=10, T=0.3, q=0)
         print(ui,c-uo)
```

```
0.3981948482776454 0.3981948482776456
```

The final prices we give here are identical.

3.4

In this question we use a the tree method to directly price an American up-and-in put option.

All we need to do is again construct a payoff class that satisify the up and in condition. This time we need to give two prices so that we can better illustrate the characters of this barrier option. The reason we do this is that we don't know whether the option has been activated or not when we discounting from the top to root of the tree, so the very intuitive way to price is just give two price to the next nodes when we discounting. Once the stock price is above the barrier, we will automatically change the two option price into activated price. The result is like this.

```
In [29]: class up_in_putpayoff(Payoff):
             def __init__(self,Strike,Barrier):
                 Payoff.__init__(self,Strike)
                 self.Barrier = Barrier
             def getpayoff(self,Price):
                 if Price <= self.Barrier:
                     # we give two prices here which is a list
                     return np.array([max(self.Strike-Price,0),0])
                 else:
                     return np.array([max(self.Strike-Price,0),max(self.Strike-Price,0)])
             def getnodeprice(self,Price,Dis_price):
                 if Price <=self.Barrier:
                     return Dis_price
                 else:
                     # once the option is activated, we only discouted useing the activated va
                     return np.asarray([Dis_price[0],Dis_price[0]])
             def getidentity(self):
                 return "putpayoff"
         additive_binomial_tree(0.3,10,0.01,0.2,400,
                                up_in_putpayoff(Strike = 10,Barrier = 11),
                                D=0).amer_discount()[1]

Out[29]: 0.015215641498553261
```

Actually we can verify this answer by applying the in-out parity. We now calculate the American up-and-out put option using the binomial tree and American vanilla put option too, then we do the subtraction and check the number.

```
In [30]: class up_out_putpayoff(Payoff):
             def __init__(self,Strike,Barrier):
                 Payoff.__init__(self,Strike)
                 self.Barrier = Barrier
             def getpayoff(self,Price):
                 if Price >= self.Barrier:
                     return np.asarray([0])
                 else:
                     return np.asarray([max(self.Strike-Price,0)])
             def getnodeprice(self,Price,Dis_price):
                 if Price >=self.Barrier:
                     return np.asarray([0])
                 else:
                     return np.asarray(Dis_price)
             def getidentity(self):
                 return "putpayoff"
         auop = additive_binomial_tree(0.3,10,0.01,0.2,400,
                                       up_out_putpayoff(Strike = 10,Barrier = 11),
                                       D=0).amer_discount()
         avp = additive_binomial_tree(0.3,10,0.01,0.2,400,
```

```
                                        putpayoff(Strike = 10),D=0).amer_discount()
        print(avp-auop)
```

```
[0.01521563]
```

They are almost identical except for little biase due to the decimal computing process when running the program.

4 Problem 4. Finite Difference Methods

4.1

Construct the explicit finit difference grid class which can inheirt from the base class "tree". It is similar to the trinomial tree, however this time we generate a lattice.

```
In [39]: class explicit_fd_grid(tree):
             def __init__(self,T,S,r,sigma,N,payoff,dx,D,Nj):
                 tree.__init__(self,T,S,r,sigma,N,payoff,D)
                 self.Nj = Nj
                 self.dx = dx
             def build_tree(self):
                 self.delta_t = self.T/self.N
                 self.nu = self.r-self.D-0.5*self.sigma**2
                 self.p_u = 0.5*(self.sigma**2*self.delta_t/\
                                 self.dx**2+self.nu*self.delta_t/self.dx)
                 self.p_m = 1-self.sigma**2*self.delta_t/self.dx**2-self.r*self.delta_t
                 self.p_d = 0.5*(self.sigma**2*self.delta_t/\
                                 self.dx**2-self.nu*self.delta_t/self.dx)
                 # initialize the stock price and option prices at the end of the grid
                 self.St = self.S*np.exp(np.asarray([self.Nj*self.dx-i*self.dx\
                                                     for i in range(2*self.Nj+1)]))
                 self.C = np.asarray([self.payoff.getpayoff(p) for p in self.St])
             def euro_discount(self):
                 self.build_tree()
                 N = self.N
                 while (N>0):
                     # compute option prices on  Nj-1 nodes at the previous level
                     self.dis_C = (self.p_u*self.C[:-2]+self.p_m*self.C[1:-1]+self.p_d*self.C[2
                     # compute the first and last price according to boundary condition
                     if self.payoff.getidentity() == "callpayoff":
                         C_large = self.dis_C[0]+self.St[0]-self.St[1]
                         C_small = self.dis_C[-1]
                     else:
                         C_large = self.dis_C[0]
                         C_small = self.dis_C[-1]-(self.St[-1]-self.St[-2])
                     self.dis_C = np.concatenate(([C_large],self.dis_C,[C_small]))
                     self.C = np.asarray([self.payoff.getnodeprice(self.St[i],self.dis_C[i])\
```

```
                                    for i in range(len(self.St))])# apply the condition c
            N -= 1
        return self.C[self.Nj]
    def amer_discount(self):
        self.build_tree()
        N = self.N
        while (N>0):
            self.dis_C = (self.p_u*self.C[:-2]+self.p_m*self.C[1:-1]+\
                          self.p_d*self.C[2:])# compute discounted value of product
            if self.payoff.getidentity() == "callpayoff":
                C_large = self.dis_C[0]+self.St[0]-self.St[1]
                C_small = self.dis_C[-1]
            else:
                C_large = self.dis_C[0]
                C_small = self.dis_C[-1]-(self.St[-1]-self.St[-2])
            self.dis_C = self.dis_C = np.concatenate(([C_large],self.dis_C,[C_small]))
            self.dis_C = np.asarray([self.payoff.getnodeprice(self.St[i],self.dis_C[i]
                                    for i in range(len(self.St))])
            self.exc_C = np.asarray([self.payoff.getpayoff(p) for p in self.St])
            self.C = np.where(self.dis_C < self.exc_C, self.exc_C, self.dis_C)
            N -= 1
        return self.C[self.Nj]
```

4.2

Same to question one, we construct the implicit finit difference grid class. Note that we need also construct a function to solve the equation system in every step.

In [40]:
```
# construct the function to solve the equations
def solve(p_u,p_m,p_d,C_ip1,Nj):
    pmp = np.zeros((2*Nj-1,1))
    pp = np.zeros((2*Nj-1,1))
    lambda_l = C_ip1[-1]
    lambda_u = C_ip1[0]
    pmp[-1] = p_m+p_d
    pp[-1] = C_ip1[-2]+p_d*lambda_l
    C_i = np.zeros((2*Nj+1,1))
    for i in range(2*Nj-2):
        pmp[2*Nj-3-i] = p_m-p_u/pmp[2*Nj-3-i+1]*p_d
        pp[2*Nj-3-i] = C_ip1[2*Nj-2-i]-pp[2*Nj-3-i+1]/pmp[2*Nj-3-i+1]*p_d
    C_i[0] = lambda_u+(pp[0]-lambda_u*p_u)/(pmp[0]+p_u)
    for i in range(2*Nj-1):
        C_i[i+1] = (pp[i]-p_u*C_i[i])/pmp[i]
    C_i[-1] = C_i[-2]-lambda_l
    return C_i, pmp, pp
```

In [41]:
```
class implicit_fd_grid(tree):
    def __init__(self,T,S,r,sigma,N,payoff,dx,D,Nj):
```

```python
        tree.__init__(self,T,S,r,sigma,N,payoff,D)
        self.Nj = Nj
        self.dx = dx
    def build_tree(self):
        self.delta_t = self.T/self.N
        self.nu = self.r-self.D-0.5*self.sigma**2
        self.p_u = -0.5*self.delta_t*((self.sigma/self.dx)**2+self.nu/self.dx)
        self.p_m = 1+self.delta_t*(self.sigma/self.dx)**2+self.r*self.delta_t
        self.p_d = -0.5*self.delta_t*((self.sigma/self.dx)**2-self.nu/self.dx)
        self.St = self.S*np.exp(np.asarray([self.Nj*self.dx-i*self.dx\
                                            for i in range(2*self.Nj+1)]))
        self.C = np.asarray([self.payoff.getpayoff(p) for p in self.St])
    def euro_discount(self):
        self.build_tree()
        N = self.N
        while (N>0):
            # generate the boundary conditions in every steps
            if self.payoff.getidentity() == "callpayoff":
                self.lambda_u = np.asarray([self.St[0] - self.St[1]])
                self.lambda_l = np.asarray([0])
            else:
                self.lambda_u = np.asarray([0])
                self.lambda_l = np.asarray([(self.St[-1] - self.St[-2])])
            self.C_ip1 = np.concatenate(([self.lambda_u],self.C[1:-1],[self.lambda_l]
            # get the "discounted" price by solving the equations
            self.dis_C, self.pmp, self.pp= solve(self.p_u,self.p_m,\
                                                 self.p_d,self.C_ip1,self.Nj)
            # add the conditions if needed
            self.C = np.asarray([self.payoff.getnodeprice(self.St[i],self.dis_C[i])\
                                 for i in range(len(self.St))])# apply the condition
            N -= 1
        return self.C[self.Nj]
    def amer_discount(self):
        self.build_tree()
        N = self.N
        while (N>0):

            if self.payoff.getidentity() == "callpayoff":
                self.lambda_u = np.asarray([self.St[0] - self.St[1]])
                self.lambda_l = np.asarray([0])
            else:
                self.lambda_u = np.asarray([0])
                self.lambda_l = np.asarray([(self.St[-1] - self.St[-2])])
            self.C = np.concatenate(([self.lambda_u],self.C[1:-1],[self.lambda_l]))
            self.C_ip1 = np.concatenate(([self.lambda_u],self.C[1:-1],[self.lambda_l]
            #solve equations
            self.dis_C, self.pmp, self.pp= solve(self.p_u,self.p_m,\
                                                 self.p_d,self.C_ip1,self.Nj)
```

```
                self.dis_C = np.asarray([self.payoff.getnodeprice(self.St[i],self.dis_C[i
                                         for i in range(len(self.St))])
                self.exc_C = np.asarray([self.payoff.getpayoff(p) for p in self.St])
                self.C = np.where(self.dis_C < self.exc_C, self.exc_C, self.dis_C)
                N -= 1
            return self.C[self.Nj]
```

4.3

Same to question two, we construct the Crank-Nicolson finit difference grid class. We should consider to upgrade the coefficients matrix and vector.

```
In [42]: class cn_fd_grid(tree):
             def __init__(self,T,S,r,sigma,N,payoff,dx,D,Nj):
                 tree.__init__(self,T,S,r,sigma,N,payoff,D)
                 self.Nj = Nj
                 self.dx = dx
             def build_tree(self):
                 self.delta_t = self.T/self.N
                 self.nu = self.r-self.D-0.5*self.sigma**2
                 self.p_u = -0.25*self.delta_t*((self.sigma/self.dx)**2+self.nu/self.dx)
                 self.p_m = 1+self.delta_t/2*(self.sigma/self.dx)**2+self.r*self.delta_t/2
                 self.p_d = -0.25*self.delta_t*((self.sigma/self.dx)**2-self.nu/self.dx)
                 self.St = self.S*np.exp(np.asarray([self.Nj*self.dx-i*self.dx\
                                                     for i in range(2*self.Nj+1)]))
                 self.C = np.asarray([self.payoff.getpayoff(p) for p in self.St])
             def euro_discount(self):
                 self.build_tree()
                 N = self.N
                 while (N>0):
                     # boundary conditions
                     if self.payoff.getidentity() == "callpayoff":
                         lambda_u = np.asarray([self.St[0] - self.St[1]])
                         lambda_l = np.asarray([0])
                     else:
                         lambda_u = np.asarray([0])
                         lambda_l = np.asarray([(self.St[-1] - self.St[-2])])
                     # upgrade cofficients vetor
                     self.C_ip1 = -self.p_u*self.C[:-2]-(self.p_m-2)*self.C[1:-1]\
                                  -self.p_d*self.C[2:]
                     self.C_ip1 = np.concatenate(([lambda_u],self.C_ip1,[lambda_l]))
                     # solve the equations
                     self.dis_C, self.pmp, self.pp= solve(self.p_u,self.p_m,\
                                                          self.p_d,self.C_ip1,self.Nj)
                     self.C = np.asarray([self.payoff.getnodeprice(self.St[i],self.dis_C[i])\
                                          for i in range(len(self.St))])# apply the condition
                     N -= 1
                 return self.C[self.Nj]
```

```
def amer_discount(self):
    self.build_tree()
    N = self.N
    while (N>0):

        if self.payoff.getidentity() == "callpayoff":
            lambda_u = np.asarray([self.St[0] - self.St[1]])
            lambda_l = np.asarray([0])
        else:
            lambda_u = np.asarray([0])
            lambda_l = np.asarray([(self.St[-1] - self.St[-2])])
        # build C
        self.C_ip1 = -self.p_u*self.C[:-2]-(self.p_m-2)*self.C[1:-1]\
        -self.p_d*self.C[2:]
        self.C_ip1 = np.concatenate(([lambda_u],self.C_ip1,[lambda_l]))
        #solve equation
        self.dis_C, self.pmp, self.pp= solve(self.p_u,self.p_m,self.p_d,\
                                             self.C_ip1,self.Nj)
        self.dis_C = np.asarray([self.payoff.getnodeprice(self.St[i],self.dis_C[i
                                 for i in range(len(self.St))])
        self.exc_C = np.asarray([self.payoff.getpayoff(p) for p in self.St])
        self.C = np.where(self.dis_C < self.exc_C, self.exc_C, self.dis_C) # do t
        N -= 1
    return self.C[self.Nj]
```

4.4

Now we can use the class to compute the prices.

If we want to obtain the desired error of $\epsilon \leq 0.001$, we need to compute the Δx, Δt and N_j. For both implicit and explicit finite defference, the order of the convergence is $O(\Delta x^2 + \Delta t)$.

So,

$$\Delta x^2 + \Delta t = \epsilon where : \Delta x = \sigma\sqrt{3\Delta t}$$

Then we can solve the equation:

$$\Delta t = \frac{\epsilon}{1+3\sigma^2}\Delta x = \sigma\sqrt{\frac{3\epsilon}{1+3\sigma^2}}$$

Then we can get the time steps and space steps.

$$N = \lceil\frac{T}{\Delta t}\rceil Nj = \lceil\frac{n_{sd}\sqrt{N/3}-1}{2}\rceil$$

4.5

If we are given the parameters' specific number, we can calculate all the inputs as we mentioned above. Note that we give the error as 0.001 and the number of standard deviation is 6. That is:

$$\Delta t = 0.00084\Delta x = 0.01257N = 1188N_j = 60$$

```
In [44]: e = 0.001
         S0 = 100
         K = 100
         t = 1
         sig = 0.25
         r = 0.06
         div = 0.03
         d_t = e/(1+3*sig**2)
         d_x = sig*m.sqrt(3*d_t)
         n = m.ceil((3*0.25**2+1)/e)
         nj = m.ceil(3*m.sqrt(n/3)-0.5)

         excall = explicit_fd_grid(T=1,S=100,r=0.06,sigma=0.25,N=n,
                                  payoff = callpayoff(100),dx=d_x,
                                  D=0.03,Nj=nj).euro_discount()[0]
         exput = explicit_fd_grid(T=1,S=100,r=0.06,sigma=0.25,N=n,
                                 payoff = putpayoff(100),dx=d_x,
                                 D=0.03,Nj=nj).euro_discount()[0]
         imcall = implicit_fd_grid(T=1,S=100,r=0.06,sigma=0.25,N=n,
                                  payoff = callpayoff(100),dx=d_x,
                                  D=0.03,Nj=nj).euro_discount()[0]
         imput = implicit_fd_grid(T=1,S=100,r=0.06,sigma=0.25,N=n,
                                 payoff = putpayoff(100),dx=d_x,
                                 D=0.03,Nj=nj).euro_discount()[0]
         cncall = cn_fd_grid(T=1,S=100,r=0.06,sigma=0.25,N=n,
                            payoff = callpayoff(100),dx=d_x,
                            D=0.03,Nj=nj).euro_discount()[0]
         cnput = cn_fd_grid(T=1,S=100,r=0.06,sigma=0.25,N=n,
                           payoff = putpayoff(100),dx=d_x,
                           D=0.03,Nj=nj).euro_discount()[0]

         price_result3 = pd.DataFrame([[excall,imcall,cncall],
                                      [exput,imput,cnput]],index = ["Call","Put"],
                                     columns = ["Explicit FD","Implicit FD","Crank-Nicolson"])
         print(price_result3)
```

```
      Explicit FD  Implicit FD  Crank-Nicolson
Call    11.011566    11.009137       11.010352
Put      8.143203     8.140981        8.142092
```

The results we computed are very close to each others.

4.6

Here we use the iteration to find the actual steps the grid need to narrow the erro down to 0.001.

```
In [46]: grid1 = explicit_fd_grid(T=t,S=100,r=0.06,sigma=sig,N=n
                                 ,payoff = callpayoff(100),dx=d_x,
```

```
                            D=0.03,Nj=nj)
grid2 = explicit_fd_grid(T=t,S=100,r=0.06,sigma=sig,N=n,
                            payoff = putpayoff(100),dx=d_x,
                            D=0.03,Nj=nj)
grid3 = implicit_fd_grid(T=t,S=100,r=0.06,sigma=sig,N=n,
                            payoff = callpayoff(100),dx=d_x,
                            D=0.03,Nj=nj)
grid4 = implicit_fd_grid(T=t,S=100,r=0.06,sigma=sig,N=n,
                            payoff = putpayoff(100),dx=d_x,
                            D=0.03,Nj=nj)
grid5 = cn_fd_grid(T=t,S=100,r=0.06,sigma=sig,N=n,
                      payoff = callpayoff(100),dx=d_x,
                      D=0.03,Nj=nj)
grid6 = cn_fd_grid(T=t,S=100,r=0.06,sigma=sig,N=n,
                      payoff = putpayoff(100),dx=d_x,
                      D=0.03,Nj=nj)
def get_step(b_step,sig, t,nsd,error,grid,type_opt):
    if type_opt =="call":
        bs = BS_Formula(type_opt="call", r=0.06, vol=sig, K=100, S=100, T=t, q=0.03)
    else:
        bs = BS_Formula(type_opt="put", r=0.06, vol=sig, K=100, S=100, T=t, q=0.03)
    n = b_step
    nj = int(np.ceil((np.sqrt(n)*nsd/np.sqrt(3)-1)/2))
    d_x = nsd*sig*m.sqrt(t)/(2*nj+1)
    grid.N = n
    grid.Nj = nj
    grid.dx = d_x
    grid.T = t
    while (abs(grid.euro_discount()[0]-bs)>error):
        n = n+300
        nj = int((np.sqrt(n)*nsd/np.sqrt(3)-1)/2)
        d_x = nsd*sig*m.sqrt(t)/(2*nj+1)
        grid.N = n
        grid.Nj = nj
        grid.dx = d_x
    return grid.euro_discount(),n,nj,d_x,t/n
result1 = get_step(b_step=10,sig=0.25, t=1,nsd=6,error=0.001,grid = grid1, type_opt="
result2 = get_step(b_step=10,sig=0.25, t=1,nsd=6,error=0.001,grid = grid2, type_opt="
result3 = get_step(b_step=10,sig=0.25, t=1,nsd=6,error=0.001,grid = grid3, type_opt="
result4 = get_step(b_step=10,sig=0.25, t=1,nsd=6,error=0.001,grid = grid4, type_opt="
result5 = get_step(b_step=10,sig=0.25, t=1,nsd=6,error=0.001,grid = grid5, type_opt="
result6 = get_step(b_step=10,sig=0.25, t=1,nsd=6,error=0.001,grid = grid6, type_opt="
step_result = pd.DataFrame([[result1,result3,result5],
                               [result2,result4,result6]],index = ["Call","Put"],
                              columns = ["Explicit FD","Implicit FD","Crank-Nicolson"])
print(step_result)
```

```
   Explicit FD  Implicit FD  Crank-Nicolson
```

Call	1810	4510	3010
Put	2110	4810	3610

It takes over 4000 steps for Implicit and over 3000 steps Crank-Nicolson FD methods to get such a accuracy, while it takes almost 2000 or less steps for Explicit FD methods to get the same accuracy. It takes more steps to get to the put options than call options.

4.7 Bonus

We construct a new tree to implement the Rannacher modification where we replace the first time step of Crank-Nicolson method with four quarter steps of implicit method.

```
In [48]: class ran_fd_grid(tree):
             def __init__(self,T,S,r,sigma,N,payoff,dx,D,Nj):
                 tree.__init__(self,T,S,r,sigma,N,payoff,D)
                 self.Nj = Nj
                 self.dx = dx
             def build_tree(self):
                 self.delta_t = self.T/self.N
                 self.nu = self.r-self.D-0.5*self.sigma**2
                 self.p_u = -0.25*self.delta_t*((self.sigma/self.dx)**2+self.nu/self.dx)
                 self.p_m = 1+self.delta_t/2*(self.sigma/self.dx)**2+self.r*self.delta_t/2
                 self.p_d = -0.25*self.delta_t*((self.sigma/self.dx)**2-self.nu/self.dx)
                 self.St = self.S*np.exp(np.asarray([self.Nj*self.dx-i*self.dx \
                                                     for i in range(2*self.Nj+1)]))
                 self.C = np.asarray([self.payoff.getpayoff(p) for p in self.St])
             def euro_discount(self):
                 self.build_tree()
                 self.add_grid = implicit_fd_grid(T = self.delta_t ,S=self.S,r=self.r,
                                                 sigma=self.sigma,N=4,
                                                 payoff=self.payoff,dx=self.dx,
                                                 D=self.D,Nj=self.Nj)
                 self.add_grid.euro_discount()
                 self.C = self.add_grid.C
                 N = self.N-1
                 while (N>0):
                     # boundary conditions
                     if self.payoff.getidentity() == "callpayoff":
                         lambda_u = np.asarray([self.St[0] - self.St[1]])
                         lambda_l = np.asarray([0])
                     else:
                         lambda_u = np.asarray([0])
                         lambda_l = np.asarray([(self.St[-1] - self.St[-2])])
                     # build C
                     self.C_ip1 = -self.p_u*self.C[:-2]-(self.p_m-2)*self.C[1:-1]\
                                         -self.p_d*self.C[2:]
                     self.C_ip1 = np.concatenate(([lambda_u],self.C_ip1,[lambda_l]))
```

```
                self.dis_C, self.pmp, self.pp= solve(self.p_u,self.p_m,\
                                                    self.p_d,self.C_ip1,self.Nj)
                self.C = np.asarray([self.payoff.getnodeprice(self.St[i],self.dis_C[i])\
                                     for i in range(len(self.St))])# apply the condition
                N -= 1
            return self.C[self.Nj]
        def amer_discount(self):
            self.build_tree()
            self.add_grid = implicit_fd_grid(T = self.delta_t ,S=self.S,
                                             r=self.r,sigma=self.sigma,N=4,
                                             payoff=self.payoff,dx=self.dx,
                                             D=self.D,Nj=self.Nj)
            self.add_grid.amer_discount()
            self.C = self.add_grid.C
            N = self.N-1
            while (N>0):
                #build C
                if self.payoff.getidentity() == "callpayoff":
                    lambda_u = np.asarray([self.St[0] - self.St[1]])
                    lambda_l = np.asarray([0])
                else:
                    lambda_u = np.asarray([0])
                    lambda_l = np.asarray([(self.St[-1] - self.St[-2])])
                # build C
                self.C_ip1 = -self.p_u*self.C[:-2]-(self.p_m-2)*self.C[1:-1]\
                                -self.p_d*self.C[2:]
                self.C_ip1 = np.concatenate(([lambda_u],self.C_ip1,[lambda_l]))
                #solve equation
                self.dis_C, self.pmp, self.pp= solve(self.p_u,self.p_m,self.p_d,\
                                                    self.C_ip1,self.Nj)
                self.dis_C = np.asarray([self.payoff.getnodeprice(self.St[i],self.dis_C[i
                                         for i in range(len(self.St))])
                self.exc_C = np.asarray([self.payoff.getpayoff(p) for p in self.St])
                self.C = np.where(self.dis_C < self.exc_C, self.exc_C, self.dis_C) # do t
                N -= 1
            return self.C[self.Nj]
    grid1 = ran_fd_grid(T=1,S=100,r=0.06,sigma=0.2,N=3,payoff = \
                        putpayoff(100),dx=0.2,D=0.03,Nj=3)
    grid2 = cn_fd_grid(T=1,S=100,r=0.06,sigma=0.2,N=3,payoff = \
                       putpayoff(100),dx=0.2,D=0.03,Nj=3)
    print(grid1.amer_discount()[0])
    print(grid2.amer_discount()[0])
```

```
5.38958578051264
5.418377337366586
```

As we can see, Using the new pricing methods (top one) can get the similar results of using Crank-Nikolson method (bottom one).

5 Bonus

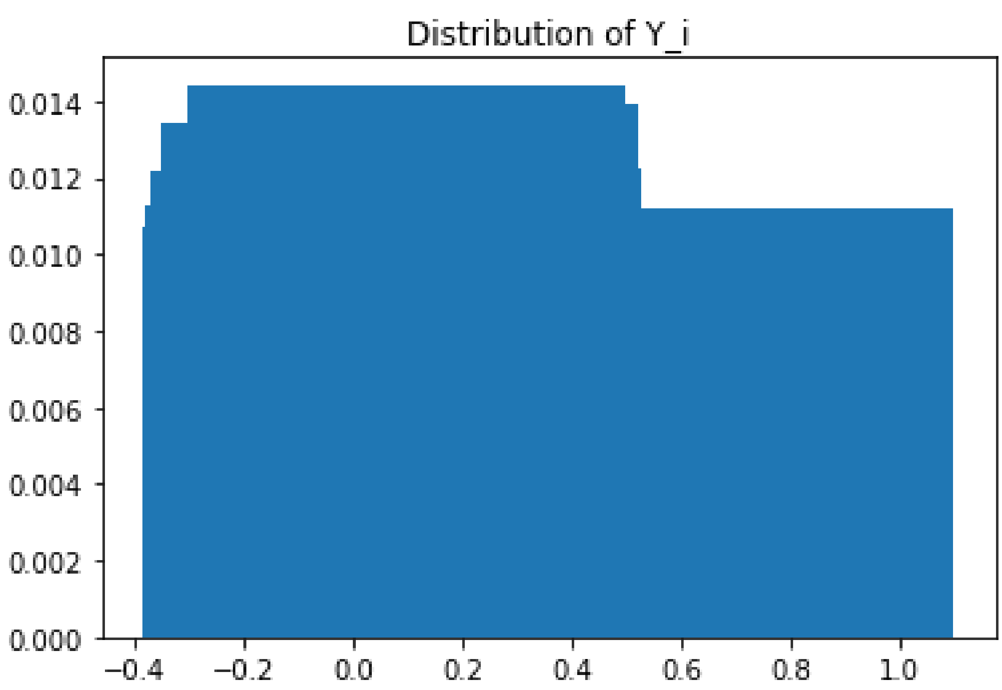
5.1

In this problem we need find the Y_i distrubution when attaining the historical stock prices. According to the essay, we need to first simulate the Y process and X process then use the results to give a discrete distrubution to best matching the real data. The result we give here is the final steps' Y distribution.

```
In [97]: def phi(x):
             if x<1 and x>-1:
                 return 1-abs(x)
             else:
                 return 0
         def phin(x,n):
             return n**(1/3)*phi(x*n**(1/3))
         def mutation(x,y,h,M,alpha,nu,mu,f,sig):
             dt = h/M
             count = M
             yp = y
             xp = x
             while (count>0):
                 xp = xp+dt*(mu-sig(yp)**2/2)+np.sqrt(dt)*sig(yp)*np.random.normal(0,1,1,)[0]
                 yp = yp+dt*alpha*(nu-yp)+np.sqrt(dt)*f(yp)*np.random.normal(0,1,1,)[0]
                 count = count-1
             return xp,yp

         def selection(Xp,Yp,func,xr,n):
             C = sum([func(xp-xr,n) for xp in Xp])
             prob = np.asarray([0]+[func(xp-xr,n)/C for xp in Xp])
             cumprob = prob.cumsum()
             rand = np.random.uniform(0,1,n)
             sample = []
             for num in rand:
                 for l in range(len(cumprob)-1):
                     if num> cumprob[l] and num<= cumprob[l+1]:
                         index = l
                 sample.append(Yp[index])
             return sample,prob[1:]

         def vol_dis(X,y0,h,M,alpha,nu,mu,f,sig,n):
             #step1
             count = n
             Xp =[]
             Yp = []
             while (count>0):
                 xp,yp = mutation(X[0],y0,h,M,alpha,nu,mu,f,sig)
                 Xp.append(xp)
                 Yp.append(yp)
```

```
            count = count-1
        sample,sample_prob = selection(Xp,Yp,phin,X[1],n)
        # step2
        for index in range(2,len(X)):
            Xp =[]
            Yp = []
            for y in sample:
                xp,yp = mutation(X[index-1],y,h,M,alpha,nu,mu,f,sig)
                Xp.append(xp)
                Yp.append(yp)
            sample,sample_prob = selection(Xp,Yp,phin,X[index],n)
        return Yp,sample_prob

    def bigphi(x):
        return x
    def sigfunc(x):
        return x
    os.chdir('D:\\Grad 2\\621\\assignment\\HM1\\data')
    data = pd.read_csv('combined equity data.csv')
    X = data["Close_amzn"]
    X = np.log(X/X.shift(1))[1:]
    X = list(X)
    Y,P = vol_dis(X,0.25,0.1,40,0.2,0.2,0.5,bigphi,sigfunc,100)
    plt.bar(Y, P)
    plt.title("Distribution of Y_i")
```

```
Out[97]: Text(0.5,1,'Distribution of Y_i')
```



Here we give the discrete distribution of Y (with amount of 100) by using the historical data of AMZN.

5.2

In this question, we give the successor of a given starting point and corresponding probability.

```
In [100]: def successor(x,L,Yi,dt,sig,p,r):
              vol = sig(Yi)*np.sqrt(dt)
              j = -L
              while (j*vol<x):
                  j = j+1
              x1 = (j+1)*vol+(r-sig(Yi)**2/2)*dt
              x2 = j*vol+(r-sig(Yi)**2/2)*dt
              x3 = (j-1)*vol+(r-sig(Yi)**2/2)*dt
              x4 = (j-2)*vol+(r-sig(Yi)**2/2)*dt
              if x2-x<=x-x3:
                  q = (x-x2)/vol
                  p1 = (1+q+q**2)/2-p
                  p2 = 3*p-q**2
                  p3 = (1-q+q**2)/2-3*p
                  p4 = p
              else:
                  q = (x-x3)/vol
                  p1 = p
                  p2 = (1-q+q**2)/2-3*p
                  p3 = 3*p-q**2
                  p4 = (1+q+q**2)/2-p
              return np.asarray([[x1,x2,x3,x4],[p1,p2,p3,p4]])
          successor(x=X[-1],L=100,Yi=Y[0],dt=0.01,sig=sigfunc,p=0.1,r=0.06)

Out[100]: array([[ 9.30193655e-03,  5.61805051e-04, -8.17832645e-03,
                  -1.69184580e-02],
                 [ 1.00000000e-01,  8.61884434e-02,  1.77212173e-01,
                   6.36599384e-01]])
```

The first 4 number is the seccessor of the last x in the historical data, and last 4 number is the corresponding probabilities.