Homework 1

February 20, 2019

This is the first homework of Guanzhuo Qiao in FE-621 class. The data is analyzed by Pyhton and the report is produced by Jupyter.

1 Part 1

1.1

Download data from Yahoo finance using the "pandas_datareader" package in python. In order to download the data, I write a function to download every stock's equity data and options data in a given list and save them into a local direction.

```
In [10]: import pandas as pd
         from pandas_datareader import data as pdrd
         import pandas_datareader.yahoo.options as pdro
         import datetime as dt
         import math as m
         import numpy as np
         from scipy.stats import norm
         import matplotlib.pyplot as plt
         import time
         import os
```

```
In [11]: os.chdir('D:\\Grad 2\\621\\assignment\\data')
```

```
In [12]: def download(stocklist):
             starttime = dt.datetime(2018,1,1)
             endtime = dt.date.today()
             for stock in stocklist:
                 equity_data = pdrd.DataReader(stock, data_source='yahoo',
                                               start=starttime,end=endtime)
                 equity_data.to_csv('{}.{} {} equity.csv'.format(endtime.month,
                                                                  endtime.day,stock))
                 option_name = pdro.Options(stock)
                 option_data = option_name.get_forward_data(3,call=True,put=True)
                 option_data.to_csv('{}.{} {} option.csv'.format(endtime.month,endtime.day,
                                                                  stock))
```

This function can download the equity data from 2018-01-01 to the caurrent date and save them separately into a file.

Bonus

Create another function so that it can not only download but also combine all those data and save them.

```
In [26]: def download_combine_equity(stocklist,stock_startt, stock_endt):
             equitylist = []
             for stock in stocklist:
                 equitylist.append(pdrd.DataReader(stock, data_source='yahoo',
                                                   start=stock_startt,end=stock_endt))
             equitydata = pd.merge(equitylist[0],equitylist[1],how='inner',
                                   left_index=True,right_index = True,
                                   suffixes=('_'+stocklist[0],'_'+stocklist[1]))
             for i in range(2,len(equitylist)):
                 equitylist[i].columns = [name+'_'+stocklist[i] for name in equitylist[i]]
                 equitydata = pd.merge(equitydata,equitylist[i],how='inner',
                                       left_index=True,right_index = True)
             equitydata.to_csv('combined equity data.csv')

         def combine_options(options_file_list):
             optionslist=[]
             Download_Date = []
             for file_name in options_file_list:
                 Download_Date.append(file_name.split(' ')[0])
                 optionslist.append(pd.read_csv(file_name))
                 optionslist[-1]['Download_Date']=Download_Date[-1]
             optionsdata = pd.concat(optionslist,ignore_index=True,)
             optionsdata = optionsdata[['Download_Date']+\
                         list(optionsdata.columns[optionsdata.columns != 'Download_Date'])]
             optionsdata.to_csv('combined options data.csv')
```

The first function *download_combin_equity()* is created to download and combine the equity data for a given time period.

The second function *combine_options()* is for reading and combning the saved options *.csv* files. The combination is based on the date in which they are downloaded. Here I use the previously downloaded options data files to show the catual performance.

```
In [27]: download_combine_equity(['amzn','spy','^vix','aapl'],'2019-01-01', '2019-03-01')
         combine_options(['2.11 amzn option.csv','2.11 spy option.csv',
                          '2.11 ^vix option.csv','2.12 amzn option.csv',
                          '2.12 spy option.csv','2.12 ^vix option.csv'])
```

```
In [28]: pd.read_csv('combined equity data.csv').head()
```

```
Out[28]:          Date    High_amzn     Low_amzn    Open_amzn   Close_amzn  \
         0  2019-01-02  1553.359985  1460.930054  1465.199951  1539.130005
         1  2019-01-03  1538.000000  1497.109985  1520.010010  1500.280029
         2  2019-01-04  1594.000000  1518.310059  1530.000000  1575.390015
```

```
3  2019-01-07  1634.560059  1589.189941  1602.310059  1629.510010
4  2019-01-08  1676.609985  1616.609985  1664.689941  1656.579956

   Volume_amzn  Adj Close_amzn    High_spy     Low_spy    Open_spy  \
0      7983100     1539.130005  251.210007  245.949997  245.979996
1      6975600     1500.280029  248.570007  243.669998  248.229996
2      9182600     1575.390015  253.110001  247.169998  247.589996
3      7993200     1629.510010  255.949997  251.690002  252.690002
4      8881400     1656.579956  257.309998  254.000000  256.820007

        ...        Open_^vix  Close_^vix  Volume_^vix  Adj Close_^vix  \
0       ...        27.540001   23.219999            0       23.219999
1       ...        25.680000   25.450001            0       25.450001
2       ...        24.360001   21.379999            0       21.379999
3       ...        22.059999   21.400000            0       21.400000
4       ...        20.959999   20.469999            0       20.469999

    High_aapl    Low_aapl   Open_aapl  Close_aapl  Volume_aapl  Adj Close_aapl
0  158.850006  154.229996  154.889999  157.919998   37039700.0      157.245605
1  145.720001  142.000000  143.979996  142.190002   91312200.0      141.582779
2  148.550003  143.800003  144.529999  148.259995   58607100.0      147.626846
3  148.830002  145.899994  148.699997  147.929993   54777800.0      147.298264
4  151.820007  148.520004  149.559998  150.750000   41025300.0      150.106216

[5 rows x 25 columns]
```

In [29]:
```
pd.read_csv('combined options data.csv').head()
```

Out[29]:
```
   Unnamed: 0  Download_Date  Strike      Expiry  Type              Symbol  \
0           0           2.11   690.0  2019-02-15   put  AMZN190215P00690000
1           1           2.11   710.0  2019-02-15  call  AMZN190215C00710000
2           2           2.11   750.0  2019-04-18   put  AMZN190418P00750000
3           3           2.11   760.0  2019-03-15  call  AMZN190315C00760000
4           4           2.11   760.0  2019-03-15   put  AMZN190315P00760000

     Last     Bid     Ask  Chg  \
0    0.02    0.00    0.05  0.0
1  928.14  881.25  888.45  0.0
2    0.26    0.00    0.30  0.0
3  881.00  834.15  844.00  0.0
4    0.03    0.00    0.04  0.0

                      ...                            Vol  Open_Int  \
0                     ...                            8.0      64.0
1                     ...                           10.0      10.0
2                     ...                           10.0     254.0
3                     ...                           10.0       7.0
4                     ...                           20.0     369.0
```

```
          IV  Root IsNonstandard  Underlying Underlying_Price  \
0  2.117192  AMZN         False        AMZN          1591.13
1  3.467531  AMZN         False        AMZN          1591.13
2  0.604496  AMZN         False        AMZN          1591.13
3  1.423648  AMZN         False        AMZN          1591.13
4  0.718753  AMZN         False        AMZN          1591.13

            Quote_Time      Last_Trade_Date  \
0  2019-02-11 15:31:05  2019-01-25 18:35:43
1  2019-02-11 15:31:05  2019-01-19 04:49:48
2  2019-02-11 15:31:05  2019-02-08 16:00:49
3  2019-02-11 15:31:05  2019-01-30 14:49:01
4  2019-02-11 15:31:05  2019-02-07 15:39:23

                                                JSON
0  {'contractSymbol': 'AMZN190215P00690000', 'str...
1  {'contractSymbol': 'AMZN190215C00710000', 'str...
2  {'contractSymbol': 'AMZN190418P00750000', 'str...
3  {'contractSymbol': 'AMZN190315C00760000', 'str...
4  {'contractSymbol': 'AMZN190315P00760000', 'str...

[5 rows x 21 columns]
```

1.2

We should run the previous download function for two consecutive days. Then let's read the data into the program and do the combination and clean them. Here we apply the multi-index so that we can easily find the specific data that we need in the following questions.

```
In [17]: #download(['amzn','spy','^vix'])
```

```
In [22]: #read the data
         AMZN_option1 = pd.read_csv('2.11 amzn option.csv')
         SPY_option1 = pd.read_csv('2.11 spy option.csv')
         VIX_option1 = pd.read_csv('2.11 ^vix option.csv')
         AMZN_option2 = pd.read_csv('2.12 amzn option.csv')
         SPY_option2 = pd.read_csv('2.12 spy option.csv')
         VIX_option2 = pd.read_csv('2.12 ^vix option.csv')

         #create the data sets
         Data1 = AMZN_option1.append(SPY_option1).append(VIX_option1)
         Data1 = Data1.set_index(['Underlying',Data1.index]) # set the multi-index
         Data2 = AMZN_option2.append(SPY_option2).append(VIX_option2)
         Data2 = Data2.set_index(['Underlying',Data2.index])
         del AMZN_option1,SPY_option1 ,VIX_option1,AMZN_option2,SPY_option2 ,VIX_option2

         #clean the data
```

```
Data1 = Data1.drop_duplicates()
Data2 = Data2.drop_duplicates()
Data1 = Data1[Data1.notnull()]
Data2 = Data2[Data2.notnull()]
Data1 = Data1[['Strike','Expiry','Type','Last','Bid',
               'Ask','Vol','Underlying_Price']]
Data2 = Data2[['Strike','Expiry','Type','Last','Bid',
               'Ask','Vol','Underlying_Price']]
```

1.3

The symbols used in this program are shown here:

S: spot price, *K*: strike price, *P*: current option price, *T or TtM*: time to maturity, *vol*: volatility, *Avr_Price*: average price of the bid and ask, *opt_type*: options type.

For the options symbol in the exchange, we have to explain the meaning here. Take this one for example:

"AMZN190215P00690000" stand for the options with underlying of AMZN and maturity date of 2019-02-15 and strike price of $ 00690.000

SPY is stand for the SPDR S&P 500 trust, an exchange-traded fund (ETF) which trades on the NYSE Arca. And an ETF is an investment fund traded on stock exchanges. It holds assets such as stocks, commodities, or bonds and generally operates with an arbitrage mechanism designed to keep it trading close to its net asset value. SPY is such a ETF that is designed to track the S&P 500 stock market index and it offers investors a way to diversify their portfolio without having to buy hole basket stocks.

The CBOE Volatility Index, known by its ticker symbol VIX, is a popular measure of the stock market's expectation of volatility implied by S&P 500 index options which have an average expiration of 30 days. It is commonly referred to as the fear index. The index takes as inputs the market prices of the call and put optionsand and risk-free U.S. treasury bill interest rates.

In the data we have downloaded, the nearist maturity is 1 day and the farest expiration date is at 2019-04-18 which is 66 days far from the download date. The average expiration of the at-the-money options' data we used in the following question is 27.23 days.

1.4

The current rate of Fedral funds (effective) is 2.4% annually. The underlying stocks' price is recorded in the data we downloaded previously, here we show the basic prices we'v got:

```
In [19]: # Set the basic asset prices
         rate=2.4/100
         asset_price = pd.DataFrame({'rate':[rate,rate],
                                  'AMZN_price':[Data1.loc[('AMZN',0),'Underlying_Price'],
                                               Data2.loc[('AMZN',0),'Underlying_Price']],
                                  'SPY_price':[Data1.loc[('SPY',0),'Underlying_Price'],
                                               Data2.loc[('SPY',0),'Underlying_Price']],
                                  'VIX_price':[Data1.loc[('^VIX',0),'Underlying_Price'],
                                               Data2.loc[('^VIX',0),'Underlying_Price']]},
                                  index = [dt.datetime(2019,2,11),dt.datetime(2019,2,12)])
         asset_price.index.names=['Date']
```

```
        asset_price.to_csv('asset price.csv')
        pd.read_csv('asset price.csv').head()
```

```
Out[19]:          Date   rate  AMZN_price  SPY_price  VIX_price
        0  2019-02-11  0.024     1591.13     270.49      16.27
        1  2019-02-12  0.024     1629.09     273.80      15.44
```

Now we need to get the time to maturity, we convert the form to the annual form.

```
In [23]: current_date = dt.datetime(2019,2,11)
         Data1['Expiry']=  [dt.datetime.strptime(i,'%Y-%m-%d') for i in Data1['Expiry']]
         Data1['TtM'] = [(i-current_date).days/365 for i in Data1['Expiry']]
         current_date = dt.datetime(2019,2,12)
         Data2['Expiry']=  [dt.datetime.strptime(i,'%Y-%m-%d') for i in Data2['Expiry']]
         Data2['TtM'] = [(i-current_date).days/365 for i in Data2['Expiry']]
         del current_date
```

2 Part 2

2.1

We define a function to calculate the option's price using Black-Scholes formulas. The funciton's name is *BS_Formula()*.

```
In [24]: def BS_Formula(type_opt, r, vol, K, S, T):
             d_1 = float(m.log(S/K)+(r+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-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
```

2.2

Implement the Bisection method to find the root of arbitrary functions. Apply this method to calculate the implied volatility on the first day ...

First of all, we need to define the biseciton method and define another function that gives us the average price of bid and ask.

```
In [25]: def Bisection(func,tolerance,up,down):
             #if the initail guesses are at the same side then return nan
             if np.sign(func(down)) * np.sign(func(up)) > 0:
                 return np.nan
             #if the initail guesses fit the condition then they are the roots
             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
    # do the average
    def get_avrprice(bid,ask):
        return 0.5*(bid+ask)
```

Notice that in the biseciton method, when the initial guess interval doesn't have the root, this method will give NaN as a return.

The second step is to select the data that we can use for this quesiton. we need such a data set that the option's volumn is not zero and neither of the ask or bid price should be zero. Finally, we can't have those options whose expiration is zero.

```
In [30]: # clean the data one more time
         data_6 = Data1['AMZN':'SPY']
         data_6 = data_6[data_6['Vol'] != 0]
         data_6 = data_6[(data_6['Ask'] != 0) & (data_6['Bid'] != 0)]
         data_6 = data_6[data_6['TtM'] != 0]
```

Then we define a function that implement the biseciton method to find the implied volatility.

```
In [31]: def get_iv(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)
         # time the loop
         start = time.time()
         # use the tolerance of 10e-6 and the left guess is 0.00001 and right guess is 7
         for ind in data_6.index:
             data_6.loc[ind,'Avr_Price'] = get_avrprice(data_6.loc[ind,'Bid'],
                                                         data_6.loc[ind,'Ask'])
             data_6.loc[ind,'Implied_vol_bis'] = get_iv(data_6.loc[ind,'Type'],
                                                         rate, data_6.loc[ind,'Strike'],
                                                         data_6.loc[ind,'Underlying_Price'],
                                                         data_6.loc[ind,'TtM'],
                                                         data_6.loc[ind,'Avr_Price'],
                                                         10**(-6), 7, 0.00001)


         end = time.time()
         timespent1 = end-start
         data_6 = data_6.dropna() # drop those don't have a root.
         amzn_at = data_6[(data_6['Strike']/data_6['Underlying_Price']>0.95) &
                          (data_6['Strike']/data_6['Underlying_Price']<1.05)]['AMZN':'AMZN']
         amzn_at.head()
```

```
Out[31]:                  Strike     Expiry  Type     Last     Bid     Ask  Vol  \
         Underlying
```

```
AMZN       477  1512.5 2019-02-15  call   93.30  86.45  87.90  1.0
           478  1512.5 2019-02-15   put    2.78   2.47   2.60  3.0
           479  1512.5 2019-02-22  call  151.25  92.35  93.65  1.0
           480  1512.5 2019-02-22   put    8.40   7.15   7.50  1.0
           481  1515.0 2019-02-22  call  152.93  90.40  91.70  5.0

                Underlying_Price       TtM  Avr_Price  Implied_vol_bis
Underlying
AMZN       477           1591.13  0.010959     87.175         0.525046
           478           1591.13  0.010959      2.535         0.365833
           479           1591.13  0.030137     93.000         0.386611
           480           1591.13  0.030137      7.325         0.306446
           481           1591.13  0.030137     91.050         0.386111
```

In [32]:
```python
spy_at = data_6[(data_6['Strike']/data_6['Underlying_Price']>0.95) &
                (data_6['Strike']/data_6['Underlying_Price']<1.05)]['SPY':'SPY']
spy_at.head()
```

Out[32]:
```
                Strike      Expiry  Type   Last    Bid    Ask    Vol  \
Underlying
SPY        903   257.0 2019-02-13  call  12.93  13.84  14.03   51.0
           904   257.0 2019-02-13   put   0.01   0.01   0.02  338.0
           905   257.0 2019-02-19  call  14.44  14.01  14.25    8.0
           906   257.0 2019-02-19   put   0.12   0.14   0.15    5.0
           907   257.0 2019-02-20  call   8.58  14.16  14.39  103.0

                Underlying_Price       TtM  Avr_Price  Implied_vol_bis
Underlying
SPY        903            270.45  0.005479     13.935         0.525145
           904            270.45  0.005479      0.015         0.287226
           905            270.45  0.021918     14.130         0.277817
           906            270.45  0.021918      0.145         0.204469
           907            270.47  0.024658     14.275         0.276260
```

Above is the at-the-money options's implied volatility.

In [35]:
```python
#pull out average implied volatilities of different options and assemble them
amzn_at = data_6[(data_6['Strike']/data_6['Underlying_Price']>0.95) &
                 (data_6['Strike']/data_6['Underlying_Price']<1.05)]['AMZN':'AMZN']
spy_at = data_6[(data_6['Strike']/data_6['Underlying_Price']>0.95) &
                (data_6['Strike']/data_6['Underlying_Price']<1.05)]['SPY':'SPY']
amzn_call_in_avriv =data_6['AMZN':'AMZN'].loc[(data_6['Strike']/
                                               data_6['Underlying_Price']<0.95) &
                   (data_6['Type']=='call')].Implied_vol_bis.mean()
amzn_call_out_avriv=data_6['AMZN':'AMZN'].loc[(data_6['Strike']/
                                               data_6['Underlying_Price']>1.05) &
                   (data_6['Type']=='call')].Implied_vol_bis.mean()
amzn_put_out_avriv =data_6['AMZN':'AMZN'].loc[(data_6['Strike']/
                                               data_6['Underlying_Price']<0.95) &
```

```
                   (data_6['Type']=='put')].Implied_vol_bis.mean()
amzn_put_in_avriv=data_6['AMZN':'AMZN'].loc[(data_6['Strike']/
                                          data_6['Underlying_Price']>1.05) &
                   (data_6['Type']=='put')].Implied_vol_bis.mean()
spy_call_in_avriv=data_6['SPY':'SPY'].loc[(data_6['Strike']/
                                        data_6['Underlying_Price']<0.95) &
                   (data_6['Type']=='call')].Implied_vol_bis.mean()
spy_call_out_avriv=data_6['SPY':'SPY'].loc[(data_6['Strike']/
                                         data_6['Underlying_Price']>1.05) &
                   (data_6['Type']=='call')].Implied_vol_bis.mean()
spy_put_out_avriv=data_6['SPY':'SPY'].loc[(data_6['Strike']/
                                        data_6['Underlying_Price']<0.95) &
                   (data_6['Type']=='put')].Implied_vol_bis.mean()
spy_put_in_aviv=data_6['SPY':'SPY'].loc[(data_6['Strike']/
                                      data_6['Underlying_Price']>1.05) &
                   (data_6['Type']=='put')].Implied_vol_bis.mean()
amzn_call_at_avriv = amzn_at[amzn_at['Type']=='call'].Implied_vol_bis.mean()
amzn_put_at_avriv = amzn_at[amzn_at['Type']=='put'].Implied_vol_bis.mean()
spy_call_at_avriv = spy_at[spy_at['Type']=='call'].Implied_vol_bis.mean()
spy_put_at_avriv = spy_at[spy_at['Type']=='put'].Implied_vol_bis.mean()

Avr_Iv = pd.DataFrame([[amzn_call_in_avriv,amzn_put_in_avriv],
                       [amzn_call_at_avriv,amzn_put_at_avriv],
                       [amzn_call_out_avriv,amzn_put_out_avriv],
                       [spy_call_in_avriv,spy_put_in_aviv],
                       [spy_call_at_avriv,spy_put_at_avriv],
                       [spy_call_out_avriv,spy_put_out_avriv]],
                        index = [['AMZN','AMZN','AMZN','SPY','SPY','SPY'],
                                 ['in-the money','at-the-money','out-the money',
                                  'in-the money','at-the-money','out-the money']],
                        columns=['Call','Put'])
Avr_Iv.index.names = ['Options','Moneyness']
Avr_Iv.columns.names = ['Type']
del amzn_call_in_avriv,amzn_call_out_avriv,amzn_put_out_avriv,\
amzn_put_in_avriv,spy_call_in_avriv,spy_call_out_avriv,\
spy_put_out_avriv,spy_put_in_aviv,amzn_at,spy_at,amzn_call_at_avriv,\
amzn_put_at_avriv,spy_call_at_avriv,spy_put_at_avriv
Avr_Iv
```

Out[35]:

```
Type                          Call       Put
Options Moneyness
AMZN    in-the money      0.594830  0.345811
        at-the-money      0.321056  0.277237
        out-the money     0.332643  0.424691
SPY     in-the money      0.495683  0.225012
        at-the-money      0.161982  0.145370
        out-the money     0.115556  0.298962
```

Above is the average implied volatility of different type of options.

2.3

Implement the Newton method/Secant method or Muller method to find the root of arbitrary functions...

Same to the last question, first we need to select the data and then we define the method, at last we calulate the implied volatility.

```
In [36]: data_7 = Data1['AMZN':'SPY']
         data_7 = data_7.loc[data_7['Vol'] != 0]
         data_7 = data_7[(data_7['Ask'] != 0) & (data_7['Bid'] != 0)]
         data_7 = data_7.loc[data_7['TtM'] != 0]

         def secant_method(func, guess1, guess2, tolerance):
             if abs(func(guess1))<tolerance:
                 return guess1
             elif abs(func(guess2))<tolerance:
                 return guess2
             else:
                 new_guess = guess2
             while(abs(func(new_guess))>tolerance):
                 k = float(func(guess2)-func(guess1))/float(guess2-guess1)
                 new_guess = guess2-func(guess2)/k
                 guess1 = guess2
                 guess2 = new_guess
             return new_guess
         def get_iv2(type_opt, r, K, S, T, P,guess1, guess2, tolerance):
             obj_func = lambda x: BS_Formula(type_opt, r, x, K, S, T) - P
             '''
             if even at 0.0001 it still can't satisfy the condition, then there
             is no root and return nan
             '''
             if obj_func(0.0001)>tolerance:
                 return np.nan
             return secant_method(obj_func, guess1, guess2, tolerance)
         start = time.time()
         for ind2 in data_7.index:
             data_7.loc[ind2,'Avr_Price'] = get_avrprice(data_7.loc[ind2,'Bid'],
                                                         data_7.loc[ind2,'Ask'])
             data_7.loc[ind2,'Implied_vol_secant'] = get_iv2(data_7.loc[ind2,'Type'], rate,
                                                             data_7.loc[ind2,'Strike'],
                                                             data_7.loc[ind2,
                                                                        'Underlying_Price'],
                                                             data_7.loc[ind2,'TtM'],
                                                             data_7.loc[ind2,'Avr_Price'],
                                                             2,1,10**(-6))
         end = time.time()
         timespent2 = end-start
         data_7 = data_7.dropna()
         data_7.head()
```

```
Out[36]:                 Strike      Expiry  Type    Last     Bid     Ask   Vol  \
         Underlying
         AMZN       1     710.0 2019-02-15  call  928.14  881.25  888.45  10.0
                    3     760.0 2019-03-15  call  881.00  834.15  844.00  10.0
                    8     765.0 2019-03-15  call  877.70  828.05  838.05   2.0
                    17    790.0 2019-04-18   put    1.15    0.02    0.62   1.0
                    18    800.0 2019-03-15  call  933.52  793.35  803.35   2.0

                         Underlying_Price       TtM  Avr_Price  Implied_vol_secant
         Underlying
         AMZN       1             1591.13  0.010959    884.850            3.846718
                    3             1591.13  0.087671    839.075            1.390122
                    8             1591.13  0.087671    833.050            1.336084
                    17            1591.13  0.180822      0.320            0.616173
                    18            1591.13  0.087671    798.350            1.272592
```

The above is the implied volatiltiy calulated by using the secant method.

```
In [37]: data_7 = Data1['AMZN':'SPY']
         data_7 = data_7.loc[data_7['Vol'] != 0]
         data_7 = data_7[(data_7['Ask'] != 0) & (data_7['Bid'] != 0)]
         data_7 = data_7.loc[data_7['TtM'] != 0]

         def muller_method(func, guess0, guess1, guess2, tolerance):
             if abs(func(guess0))<tolerance:
                 return guess0
             elif abs(func(guess1))<tolerance:
                 return guess1
             elif abs(func(guess2))<tolerance:
                 return guess2
             else:
                 new_guess = guess2
             while(abs(func(new_guess))>tolerance):
                 a = ((func(guess2)-func(guess1))/(guess2-guess1)-\
                 (func(guess1)-func(guess0))/(guess1-guess0))/(guess2-guess0)
                 b = (func(guess1)-func(guess0))/(guess1-guess0)+\
                 (func(guess2)-func(guess0))/(guess2-guess0)-\
                 (func(guess1)-func(guess2))/(guess1-guess2)
                 c = func(guess0)
                 delta = b**2-4*a*c
         # if it can't find a root when implelmenting a quadratic function, return nan
                 if delta<0:
                     return np.nan
                 new_guess1 = guess0-2*c/(b-m.sqrt(b**2-4*a*c))
                 new_guess2 = guess0-2*c/(b+m.sqrt(b**2-4*a*c))
                 if abs(func(new_guess1))<abs(func(new_guess2)):
                     new_guess = new_guess1
                 else:
```

```
                new_guess = new_guess2
            guess0 = guess1
            guess1 = guess2
            guess2 = new_guess
        return new_guess
    def get_iv3(type_opt, r, K, S, T, P, guess0, guess1, guess2, tolerance):
        obj_func = lambda x: BS_Formula(type_opt, r, x, K, S, T)-P
        if obj_func(0.0001)>tolerance:
            return np.nan
        return muller_method(obj_func, guess0, guess1, guess2, tolerance)

    start = time.time()
    for ind2 in data_7.index:
        data_7.loc[ind2,'Avr_Price'] = get_avrprice(data_7.loc[ind2,'Bid'],
                                                    data_7.loc[ind2,'Ask'])
        data_7.loc[ind2,'Implied_vol_muller'] = get_iv3(data_7.loc[ind2,'Type'],
                                                        rate,
                                                        data_7.loc[ind2,'Strike'],
                                                data_7.loc[ind2,'Underlying_Price'],
                                                        data_7.loc[ind2,'TtM'],
                                                        data_7.loc[ind2,'Avr_Price'],
                                                        2,1,0.5,0.000001)
    end = time.time()
    timespent3 = end-start
    data_7 = data_7.dropna()
    data_7.head()
```

Out[37]:

Underlying		Strike	Expiry	Type	Last	Bid	Ask	Vol
AMZN	1	710.0	2019-02-15	call	928.14	881.25	888.45	10.0
	3	760.0	2019-03-15	call	881.00	834.15	844.00	10.0
	8	765.0	2019-03-15	call	877.70	828.05	838.05	2.0
	17	790.0	2019-04-18	put	1.15	0.02	0.62	1.0
	18	800.0	2019-03-15	call	933.52	793.35	803.35	2.0

Underlying		Underlying_Price	TtM	Avr_Price	Implied_vol_muller
AMZN	1	1591.13	0.010959	884.850	3.846718
	3	1591.13	0.087671	839.075	1.390122
	8	1591.13	0.087671	833.050	1.336084
	17	1591.13	0.180822	0.320	0.616173
	18	1591.13	0.087671	798.350	1.272592

The above is the implied volatility calculated by using the muller method. As you can see in the table, the results of these two time's calulation is identical.

To be cleare, when using the secant method some implied volatility result in infinite. The reason may lies in the algorithm's defect which means the first derivative of the function at some points leads the next point goes far away, not near to, the real root and never comes back during the iteration.

As for the muller method, there is also a critical issue. It is important for us to realize that the muller method is using the quadratic function to get us the root. However, when implementing the quadratic funciton to find the next point, it is possible that this quadratic function doesn't have roots itself, so it will stop iterating even though the original function may have the root over somewhere.

Although these little issues may lead us some warning when we running the program, nevertheless we can get most of the results. And we can still compare the time consumed of different methods.

```
In [38]: Time_Consumed = pd.DataFrame([[timespent1,timespent2,timespent3]],
                                      index=['Time consumed'],
                                      columns = ['Bisection(s)','Secant(s)','Muller(s)'])
         Time_Consumed
```

```
Out[38]:                Bisection(s)  Secant(s)  Muller(s)
         Time consumed     40.401037  21.118584  36.868597
```

As we can see, the secant method is the fastest method on this problem.

2.4

Select the previously calculated data which is data_6. Then we calculate the average implied volatility for every stock, options type and maturity.

```
In [40]: data_8 = data_6
         # Here is AMZN
         amzn_call_8 = data_8['AMZN':'AMZN'].loc[data_8.Type=='call']
         amzn_put_8 = data_8['AMZN':'AMZN'].loc[data_8.Type=='put']
         # create an index contain the expiry date both in call and put
         amzn_result_table = pd.DataFrame(list(set(amzn_call_8.Expiry)\
                                               .intersection(set(amzn_put_8.Expiry))),
                                               columns = ['Expiry'])
         # find those average implied volatility for each maturity and type.
         for ind8 in amzn_result_table.index:
             amzn_result_table.loc[ind8,'AMZN_Call_IV'] = amzn_call_8[amzn_call_8\
                                    .Expiry == amzn_result_table.loc[ind8,'Expiry']].\
                                    Implied_vol_bis.mean()
             amzn_result_table.loc[ind8,'AMZN_Put_IV'] = amzn_put_8[amzn_put_8\
                                    .Expiry == amzn_result_table.loc[ind8,'Expiry']].\
                                    Implied_vol_bis.mean()
         amzn_result_table = amzn_result_table.sort_values(by=['Expiry'])
         amzn_result_table = amzn_result_table.set_index(keys=['Expiry'])
         # Do the same thing to the SPY options
         spy_call_8 = data_8['SPY':'SPY'].loc[data_8.Type=='call']
         spy_put_8 = data_8['SPY':'SPY'].loc[data_8.Type=='put']
         spy_result_table = pd.DataFrame(list(set(spy_call_8.Expiry).\
                                 intersection(set(spy_put_8.Expiry))),columns = ['Expiry'])
         for ind8 in spy_result_table.index:
             spy_result_table.loc[ind8,'SPY_Call_IV'] = spy_call_8[spy_call_8.Expiry == \
```

```
                         spy_result_table.loc[ind8,'Expiry']].Implied_vol_bis.mean()
        spy_result_table.loc[ind8,'SPY_Put_IV'] = spy_put_8[spy_put_8.Expiry == \
                         spy_result_table.loc[ind8,'Expiry']].Implied_vol_bis.mean()
    spy_result_table = spy_result_table.sort_values(by=['Expiry'])
    spy_result_table = spy_result_table.set_index(keys=['Expiry'])
    del amzn_call_8,amzn_put_8,spy_call_8,spy_put_8
    amzn_result_table
```

Out[40]:

Expiry	AMZN_Call_IV	AMZN_Put_IV
2019-02-15	0.497942	0.448613
2019-02-22	0.407998	0.373496
2019-03-01	0.356439	0.297806
2019-03-08	0.313701	0.307496
2019-03-15	0.541567	0.457329
2019-03-22	0.300207	0.296794
2019-03-29	0.298183	0.290915
2019-04-18	0.335098	0.364129

In [41]:
```
spy_result_table
```

Out[41]:

Expiry	SPY_Call_IV	SPY_Put_IV
2019-02-13	0.432727	0.206788
2019-02-15	1.216522	0.256723
2019-02-19	0.231091	0.220734
2019-02-20	0.193170	0.177107
2019-02-22	0.254042	0.223280
2019-02-25	0.184780	0.182386
2019-02-27	0.158320	0.189325
2019-03-01	0.216543	0.208386
2019-03-04	0.159101	0.186544
2019-03-06	0.164503	0.182690
2019-03-08	0.172159	0.186817
2019-03-11	0.153474	0.167904
2019-03-13	0.153910	0.175712
2019-03-15	0.568965	0.320300
2019-03-18	0.128571	0.147787
2019-03-22	0.136494	0.190206
2019-03-29	0.157946	0.258679
2019-04-18	0.172012	0.265789

Comment on the observes difference in values obtained for AMZN and SPY.

As we can see in these two tables, the SPY options' implied volatility is, in general, smaller than those of AMZN options. The reason may be the underlying of SPY is index, in another words, whole bunch of stocks is much diversified than a single stock. This property of SPY leads to the lower volatility.

Compare with the current value of the VIX

The current value of VIX is 16.27 which refers to the implied volatility of SPY to be 0.1627. In the previous question we have already got the at-the-money call/put options' implied volatility of SPY, which is 0.161982 and 0.145370. These numbers is quite close to the price of SPY. The AMZN's options' volatility is greater than VIX price.

Comment on what happens when the maturity increases. Comment on what happen when the options become in the money respectively out of the money.

When the maturity increases, the volatility of the each underlying is decreasing. As we can see in the previous question, for the call options, in-the-money implied volatility is higher than those of out-of-the-money options. As for the put options, the conclusion is opposite.

2.5

For each option in your table calculate the price of the different type (Call/Put) using the Put-Call parity...

In order to get more available data for both call and put options, we use the at-the-money data. Then we use the put-call parity to givr their counterparties prices. The put-call parity is:

$$C - P = S - Ke^{-rT}$$

At here, we present the table of all the market prices and calculated prices from call-put parity.

```
In [42]: # use the at-the-money data to better present the results
         data_9 = data_6[(data_6['Strike']/data_6['Underlying_Price']>0.95) &\
                         (data_6['Strike']/data_6['Underlying_Price']<1.05)]
         # define the put-call parity method
         def put_call_parity(opt_type,  r, K, S,T, price):
             if opt_type == 'call':
                 return price-S+K*m.exp(-r*T)
             else:
                 return S-K*m.exp(-r*T)+price
         call_9 = data_9.loc[data_9['Type'] == 'call']
         put_9 = data_9.loc[data_9['Type'] == 'put']
         call_put_9 = pd.merge(call_9,put_9,on=['Expiry','Strike'],
                               how='outer',suffixes=('_call','_put'))
         for ind9 in call_put_9.index:
             call_put_9.loc[ind9,'Calculated_call'] =\
             put_call_parity(call_put_9.loc[ind9,'Type_put'],
                             rate,call_put_9.loc[ind9,'Strike'],
                             call_put_9.loc[ind9,'Underlying_Price_put'],
                             call_put_9.loc[ind9,'TtM_put'],
                             call_put_9.loc[ind9,'Avr_Price_put'])
             call_put_9.loc[ind9,'Calculated_put'] =\
             put_call_parity(call_put_9.loc[ind9,'Type_call'],
                             rate, call_put_9.loc[ind9,'Strike'],
                             call_put_9.loc[ind9,'Underlying_Price_call'],
                             call_put_9.loc[ind9,'TtM_call'],
                             call_put_9.loc[ind9,'Avr_Price_call'])
         call_put_9 = call_put_9[['Strike','Expiry','Underlying_Price_call',
                                  'Avr_Price_call','Calculated_call','Type_call',
```

```
                             'Type_put','Calculated_put','Avr_Price_put']]
call_put_9 = call_put_9.rename(columns={'Underlying_Price_call' : 'Underlying_Price'})
call_put_9.head()
```

```
Out[42]:    Strike     Expiry  Underlying_Price  Avr_Price_call  Calculated_call  \
         0  1512.5 2019-02-15           1591.13          87.175        81.562756
         1  1512.5 2019-02-22           1591.13          93.000        87.048577
         2  1515.0 2019-02-22           1591.13          91.050        84.900385
         3  1515.0 2019-03-01           1591.13          96.875        91.672035
         4  1515.0 2019-03-22           1591.13         114.675       107.960064

           Type_call Type_put  Calculated_put  Avr_Price_put
         0      call      put        8.147244          2.535
         1      call      put       13.276423          7.325
         2      call      put       13.824615          7.675
         3      call      put       18.952965         13.750
         4      call      put       34.664936         27.950
```

```
In [43]: (abs(call_put_9.Calculated_call-call_put_9.Avr_Price_call)+\
         abs(call_put_9.Calculated_put-call_put_9.Avr_Price_put)).mean()/2
```

```
Out[43]: 2.2850669975498796
```

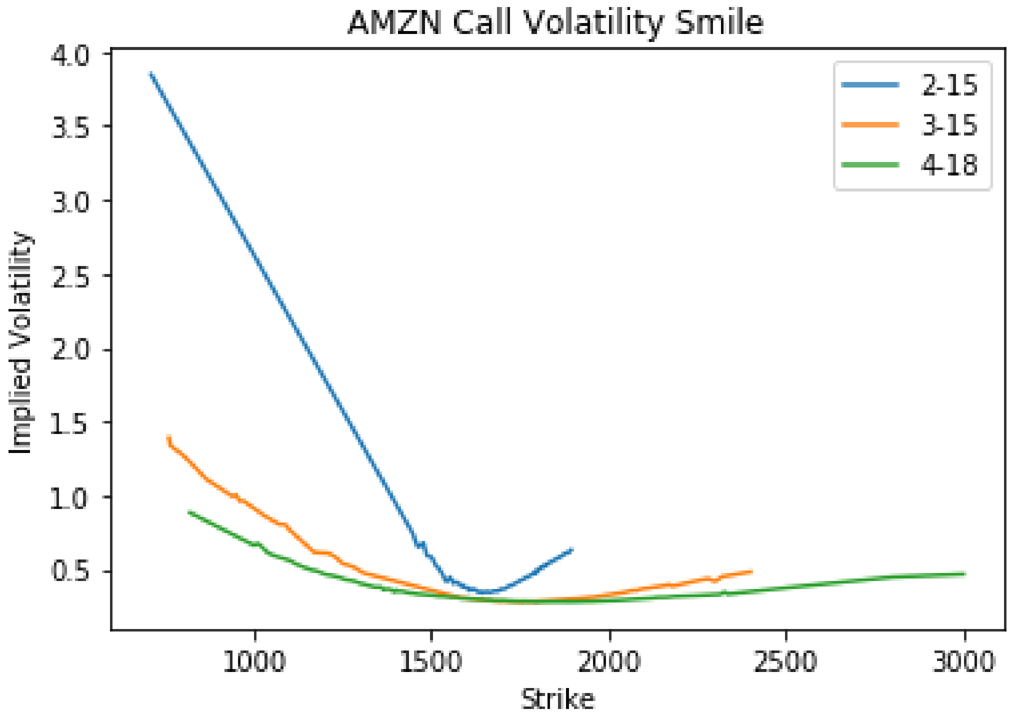
The average difference of the market price and calculated price is about 2.29.

2.6

Create a 2 dimensional plot of implied volatilities versus strike K for the closest to maturity options...

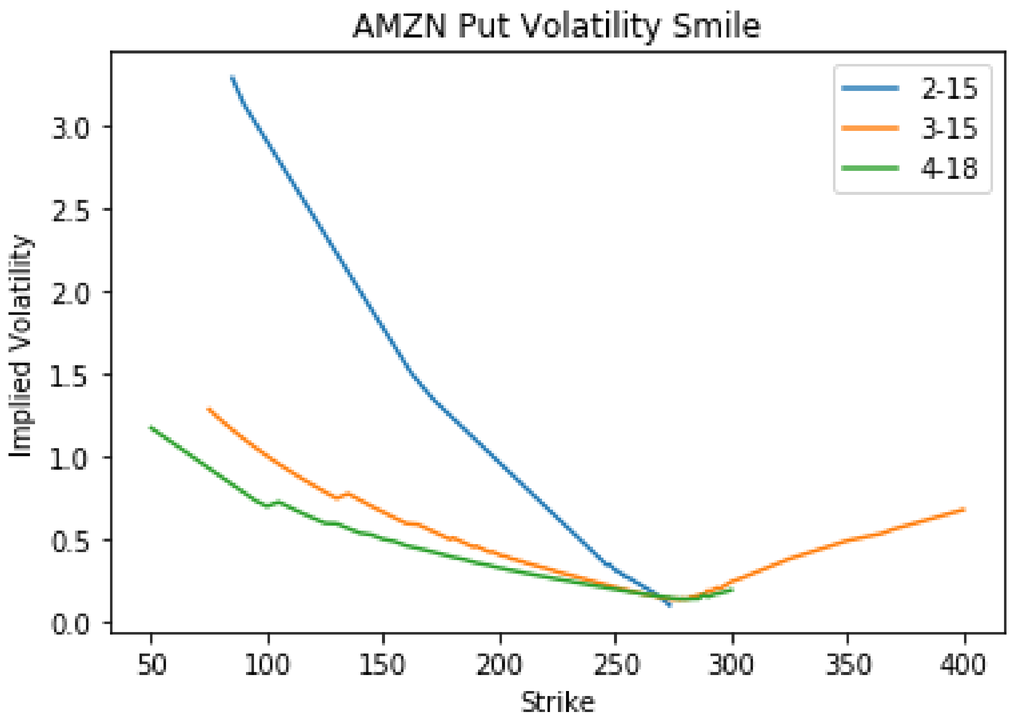
We take either the call or put options data and plot their implied volatility with strikes. The reason why we only get one type of stock option is to prevent the zigzag like graph which is caused by the different imlplied volatility of each type of options.

```
In [44]: # get the implied volatility data from bisection methods
         amzn_10 = data_6['AMZN':'AMZN']
         amzn_10 = amzn_10.loc[amzn_10.Type == 'call']
         # use 2019-02-15, 2019-03-15 and 2019-04-18 as the date points
         amzn_10_215 = amzn_10.loc[amzn_10['Expiry'] == '2019-02-15']
         amzn_10_315 = amzn_10.loc[amzn_10['Expiry'] == '2019-03-15']
         amzn_10_418 = amzn_10.loc[amzn_10['Expiry'] == '2019-04-18']
         # plot implied volatilities with different oprions' strike and expiry
         plt.plot(amzn_10_215['Strike'],amzn_10_215['Implied_vol_bis'],label = '2-15')
         plt.plot(amzn_10_315['Strike'],amzn_10_315['Implied_vol_bis'],label = '3-15')
         plt.plot(amzn_10_418['Strike'],amzn_10_418['Implied_vol_bis'],label = '4-18')
         plt.legend(loc = 0)
         plt.xlabel('Strike')
         plt.ylabel('Implied Volatility')
         plt.title('AMZN Call Volatility Smile')
         del amzn_10_215,amzn_10_315,amzn_10_418
```




```
In [45]: spy_10 = Data1['SPY':'SPY']
         spy_10 = spy_10.loc[spy_10['Vol'] != 0]
         spy_10 = spy_10.loc[spy_10['TtM'] != 0]
         spy_10 = spy_10.loc[spy_10.Type == 'put']
         # use the put options to better compare with the previous one
         for ind2 in spy_10.index:
             spy_10.loc[ind2,'Avr_Price'] = get_avrprice(spy_10.loc[ind2,'Bid'],
                                                         spy_10.loc[ind2,'Ask'])
             spy_10.loc[ind2,'Implied_vol_bis'] = get_iv(spy_10.loc[ind2,'Type'],
                                                         rate, spy_10.loc[ind2,'Strike'],
                                                         spy_10.loc[ind2,'Underlying_Price'],
                                                         spy_10.loc[ind2,'TtM'],
                                                         spy_10.loc[ind2,'Avr_Price'],
                                                         0.000001,7,0.001)
         spy_10 = spy_10.dropna()
         # use 2019-02-15, 2019-03-15 and 2019-04-18 as the date points
         spy_10_215 = spy_10.loc[spy_10['Expiry'] == '2019-02-15']
         spy_10_315 = spy_10.loc[spy_10['Expiry'] == '2019-03-15']
         spy_10_418 = spy_10.loc[spy_10['Expiry'] == '2019-04-18']

         plt.plot(spy_10_215['Strike'],spy_10_215['Implied_vol_bis'],label = '2-15')
         plt.plot(spy_10_315['Strike'],spy_10_315['Implied_vol_bis'],label = '3-15')
         plt.plot(spy_10_418['Strike'],spy_10_418['Implied_vol_bis'],label = '4-18')
         plt.legend(loc = 0)
```

```
plt.xlabel('Strike')
plt.ylabel('Implied Volatility')
plt.title('AMZN Put Volatility Smile')
del spy_10_215,spy_10_315,spy_10_418
```



The closest maturity monthly is 2019-02-15, and they are presented above. We can find that as the strike price increases, the implied volatility is about to decrease and goes up again when strike hits certain price.

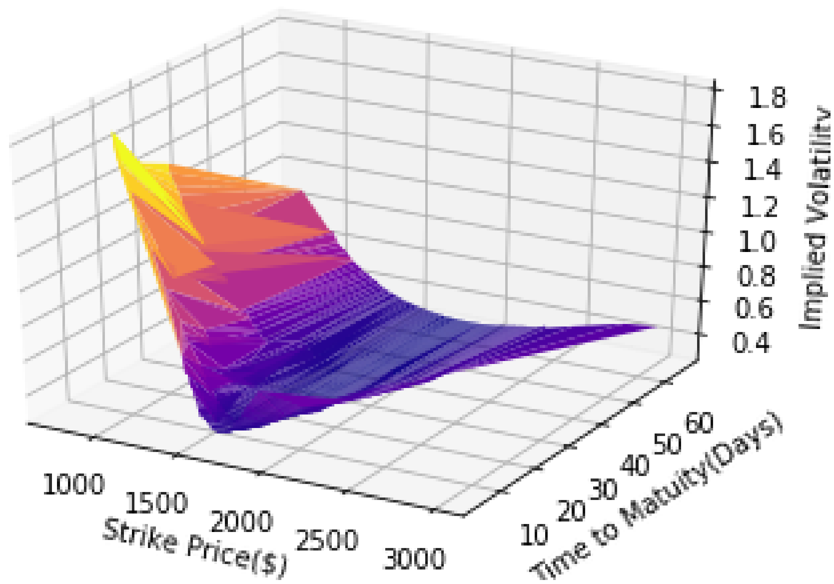
Bonus

Create a 3d plot of volatility surface.

```
In [46]: from mpl_toolkits.mplot3d import Axes3D
         # plot the volatility surface of AMZN call options
         amzn_10_3d = data_6['AMZN':'AMZN']
         amzn_10_3d = amzn_10_3d.loc[amzn_10_3d['Vol'] != 0]
         amzn_10_3d = amzn_10_3d.loc[amzn_10_3d['TtM'] != 0]
         amzn_10_3d = amzn_10_3d.loc[amzn_10_3d.Type == 'call']
         amzn_10_3d = amzn_10_3d.sort_values(by=['TtM'])

         x = np.array(amzn_10_3d.Strike[1:])
         y = np.array(amzn_10_3d.TtM[1:]*365)
         z = np.array(amzn_10_3d.Implied_vol_bis[1:])

         fig = plt.figure()
         ax = fig.gca(projection='3d')
```

```
ax.plot_trisurf(x,y,z,cmap='plasma')
ax.set_xlabel('Strike Price($)')
ax.set_ylabel('Time to Matuity(Days)')
ax.set_zlabel('Implied Volatility')
```

Out[46]: Text(0.5,0,'Implied Volatility')



2.7

Calculate the derivatives of the call option price with respect to S (Delta), and σ (Vega) and the second derivative with respect to S (Gamma).

This is the geek letters of the call options.

$$\Delta = N(d_1)$$

$$d_1 = \frac{ln(S/K) + (r + \sigma^2/2)T}{\sigma\sqrt{T}}$$

$$\nu = S\sqrt{T}N(d_1)$$

$$\Gamma = \frac{N(d_1)}{S\sigma\sqrt{T}}$$

We will get different kinds of derivatives by two method, one way is the BS formula, the second way is using the numerical method. They will be very close to each others.

In [47]:
```
#define the BS methods

def Delta_call(r, vol, K, S, T):
    d_1 = float(m.log(S/K)+(r+vol**2/2)*T)/float(vol*m.sqrt(T))
    return norm.cdf(d_1)
def Gamma(r, vol, K, S, T):
    d_1 = float(m.log(S/K)+(r+vol**2/2)*T)/float(vol*m.sqrt(T))
    return norm.pdf(d_1)/(S*vol*m.sqrt(T))
def Vega(r, vol, K, S, T):
    d_1 = float(m.log(S/K)+(r+vol**2/2)*T)/float(vol*m.sqrt(T))
    return S*m.sqrt(T)*norm.pdf(d_1)

# define the numerical methods

def First_dir(func, small_gap, x):
    return (func(x+small_gap)-func(x))/(small_gap)
def Second_dir(func,small_gap,x):
    return (func(x+small_gap)-2*func(x)+func(x-small_gap))/(small_gap**2)
def get_num_delta_call( r, vol, K, S_with_respct, T,small_gap):
    BS_s = lambda s: BS_Formula('call',r,vol,K,s,T)
    return First_dir(BS_s, small_gap, S_with_respct)
def get_num_gamma( r, vol, K, S_with_respct, T,small_gap):
    BS_s = lambda s: BS_Formula('call',r,vol,K,s,T)
    return Second_dir(BS_s, small_gap, S_with_respct)
def get_num_vega( r, vol_with_respct, K, S, T,small_gap):
    BS_sigma = lambda sigma: BS_Formula('call',r,sigma,K,S,T)
    return First_dir(BS_sigma, small_gap, vol_with_respct)

# Get the call optons

data_11 = data_6.loc[data_6.Type == 'call']

# Do the calculation

for ind11 in data_11.index:
    data_11.loc[ind11,'Delta'] = Delta_call(rate,
                data_11.loc[ind11,'Implied_vol_bis'],
                data_11.loc[ind11,'Strike'],
                data_11.loc[ind11,'Underlying_Price'],
                data_11.loc[ind11,'TtM'])
    data_11.loc[ind11,'Gamma'] = Gamma(rate,
                data_11.loc[ind11,'Implied_vol_bis'],
                data_11.loc[ind11,'Strike'],
                data_11.loc[ind11,'Underlying_Price'],
                data_11.loc[ind11,'TtM'])
    data_11.loc[ind11,'Vega'] = Vega(rate,
                data_11.loc[ind11,'Implied_vol_bis'],
                data_11.loc[ind11,'Strike'],
```

```
                    data_11.loc[ind11,'Underlying_Price'],
                    data_11.loc[ind11,'TtM'])
        data_11.loc[ind11,'Numerical_Delta'] =get_num_delta_call(rate,
                    data_11.loc[ind11,'Implied_vol_bis'],
                    data_11.loc[ind11,'Strike'],
                    data_11.loc[ind11,'Underlying_Price'],
                    data_11.loc[ind11,'TtM'],10**(-4))
        data_11.loc[ind11,'Numerical_Gamma'] =get_num_gamma(rate,
                    data_11.loc[ind11,'Implied_vol_bis'],
                    data_11.loc[ind11,'Strike'],
                    data_11.loc[ind11,'Underlying_Price'],
                    data_11.loc[ind11,'TtM'],10**(-4))
        data_11.loc[ind11,'Numerical_Vega'] =get_num_vega(rate,
                    data_11.loc[ind11,'Implied_vol_bis'],
                    data_11.loc[ind11,'Strike'],
                    data_11.loc[ind11,'Underlying_Price'],
                    data_11.loc[ind11,'TtM'],10**(-4))

data_11 = data_11[['Strike','Expiry','Type','Delta','Gamma','Vega',
                   'Numerical_Delta','Numerical_Gamma','Numerical_Vega']]
data_11.head()
```

```
Out[47]:                Strike      Expiry  Type     Delta     Gamma       Vega  \
         Underlying
         AMZN       1    710.0 2019-02-15  call  0.986303  0.000055   5.833287
                    3    760.0 2019-03-15  call  0.977574  0.000081  25.130826
                    8    765.0 2019-03-15  call  0.980025  0.000077  22.786571
                    18   800.0 2019-03-15  call  0.978244  0.000087  24.495055
                    24   820.0 2019-03-15  call  0.977935  0.000091  24.789029

                       Numerical_Delta  Numerical_Gamma  Numerical_Vega
         Underlying
         AMZN       1         0.986303         0.000057        5.833589
                    3         0.977574         0.000045       25.133717
                    8         0.980025         0.000102       22.789477
                    18        0.978244         0.000102       24.498245
                    24        0.977935         0.000091       24.792385
```

As we can see in this table, the results of these two methods is indeed quite similar.

2.8

we will use the second dataset DATA2. For each strike price in the data use the Stock price for the same day, the implied volatility you...

Fist of all, we need also clean the Data2 and we need to find those options that share the same properties. Then we use the implied volatility obtained from Data1 to give the option price of Data2.

```
In [29]: # get all available data from Data2 and clean them
         data_12 = Data2['AMZN':'SPY']
         data_12 = data_12.loc[data_12['Vol'] != 0]
         data_12 = data_12[(data_12['Ask'] != 0) & (data_12['Bid'] != 0)]
         data_12 = data_12.loc[data_12['TtM'] != 0]
         """
         Merge the implied volatility data with the Data2 in order to find the options type
         that we can apply our previously calculated implied volatility onto the next day's
         data set.

         """
         data_12 = pd.merge(data_6,data_12,
                            on=['Expiry','Strike','Type'],
                            how='inner').loc[:,['Strike','Expiry','Type',
                            'Last_y','Bid_y','Ask_y','Vol_y','Underlying_Price_y',
                             'TtM_y','Implied_vol_bis']]
         # Do the calculation
         for ind12 in data_12.index:
             data_12.loc[ind12,'BS_Price'] = BS_Formula(data_12.loc[ind12,'Type'],
                         rate, data_12.loc[ind12,'Implied_vol_bis'],
                         data_12.loc[ind12,'Strike'], data_12.loc[ind12,'Underlying_Price_y'],
                         data_12.loc[ind12,'TtM_y'])
             data_12.loc[ind12,'Avr_Price'] = get_avrprice(data_12.loc[ind12,'Bid_y'],
                                                     data_12.loc[ind12,'Ask_y'])
         data_12 = data_12[['Strike','Expiry','Type','Avr_Price','BS_Price']]
         data_12.head()
```

Out[29]:

	Strike	Expiry	Type	Avr_Price	BS_Price
0	710.0	2019-02-15	call	920.05	920.293574
1	760.0	2019-03-15	call	870.45	875.704239
2	765.0	2019-03-15	call	866.85	869.835077
3	800.0	2019-03-15	call	832.10	835.063438
4	800.0	2019-03-15	put	0.04	0.039228

The options price (BS_Price) we calcualte is quite close to the real market price (Avr_Price) of the next day. So we can partly predict the options price by having the implied volatility from the first day.

3 Part 3

3.1

Implement the trapezoidal and the Simpson's quadrature rules to numerically approximate the inde

nite integral above.

```
In [3]: def func1(x):
            if x == 0:
```

```
            return 1
        else:
            return m.sin(x)/x

    def trapezoid_int(func,a,b,n):
        if n<1:
            print('wrong number')
        x = np.linspace(a,b,n+1)
        f_x = np.vectorize(func)(x)
        delta = (b-a)/n
        return delta/2*(f_x.sum()+f_x[1:-1].sum())



    def simpson_int(func,a,b,n):
        if n<1:
            print('wrong number')
        x = np.linspace(a,b,n+1)
        f_x = np.vectorize(func)(x)
        delta = (b-a)/n
        return delta/3*(f_x[0]+f_x[-1]+4*f_x[1:-1][::2].sum()+2*f_x[1:-1][1::2].sum())

    print(trapezoid_int(func1,-(10**6),(10**6),5000000))
    print(simpson_int(func1,-(10**6),(10**6),50000000))
```

```
3.141590805133173
3.1415907800862164
```

The results of using two methods are very close to the π.

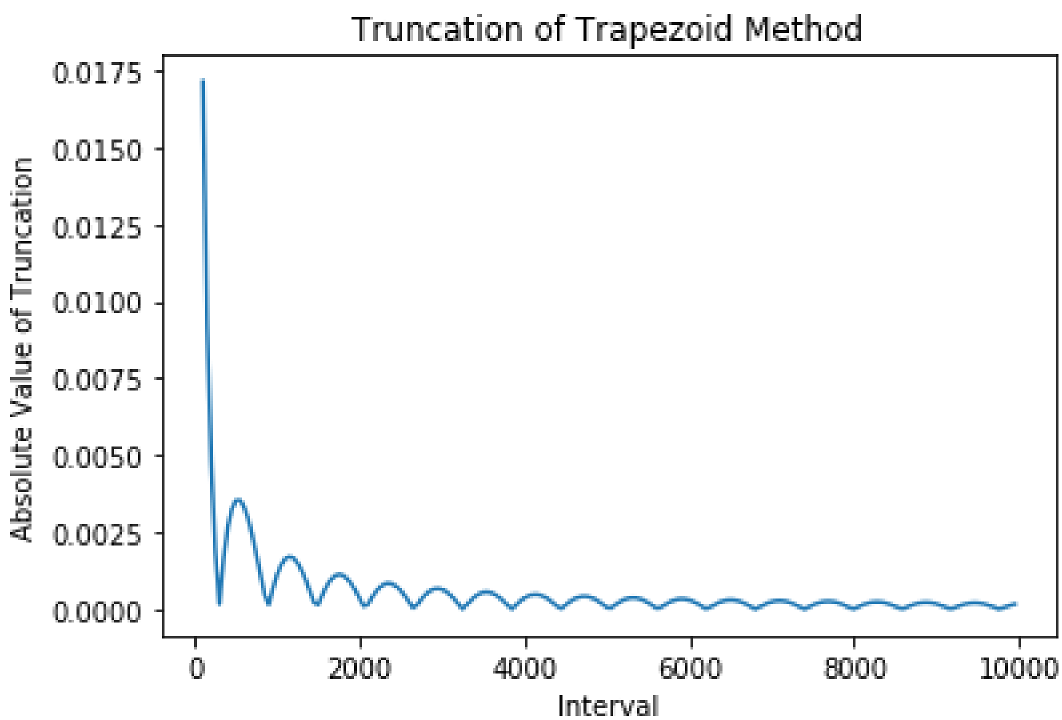
3.2

Compute the truncation error for the numerical algorithms implemented...
When fixing the number of sections and increase the length of interval:

```
In [34]: # Define the truncaiton function that do the subtraction
         def truncation_tra(a,N):
             return abs(trapezoid_int(func1,-a,a,N)-m.pi)
         def truncation_sim(a,N):
             return abs(simpson_int(func1,-a,a,N)-m.pi)
         def difference(a,N):
             return (trapezoid_int(func1,-a,a,N)-simpson_int(func1,-a,a,N))
         # fix the number of section N
         N=10**5
         a_list = np.arange(100,10**4,50)
         trunc1 = [truncation_tra(x,N) for x in a_list]
         trunc2 = [truncation_sim(x,N) for x in a_list]
         dif = [difference(x,N) for x in a_list]
```

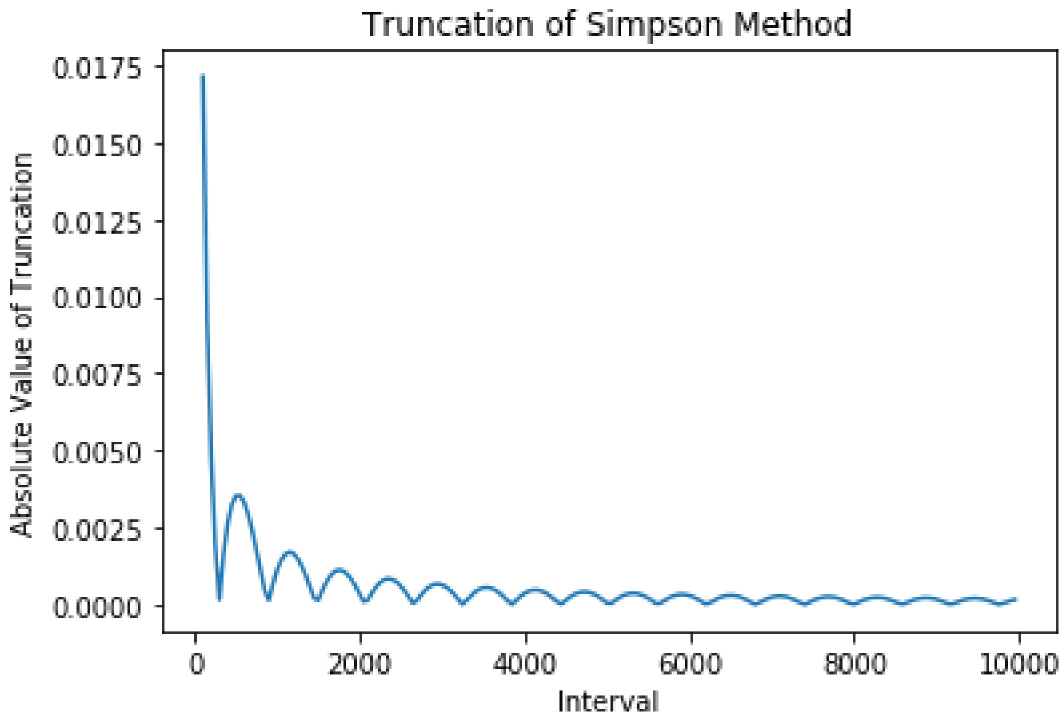
```
plt.plot(a_list,trunc1)
plt.xlabel('Interval')
plt.ylabel('Absolute Value of Truncation')
plt.title('Truncation of Trapezoid Method')
```

Out[34]: Text(0.5,1,'Truncation of Trapezoid Method')

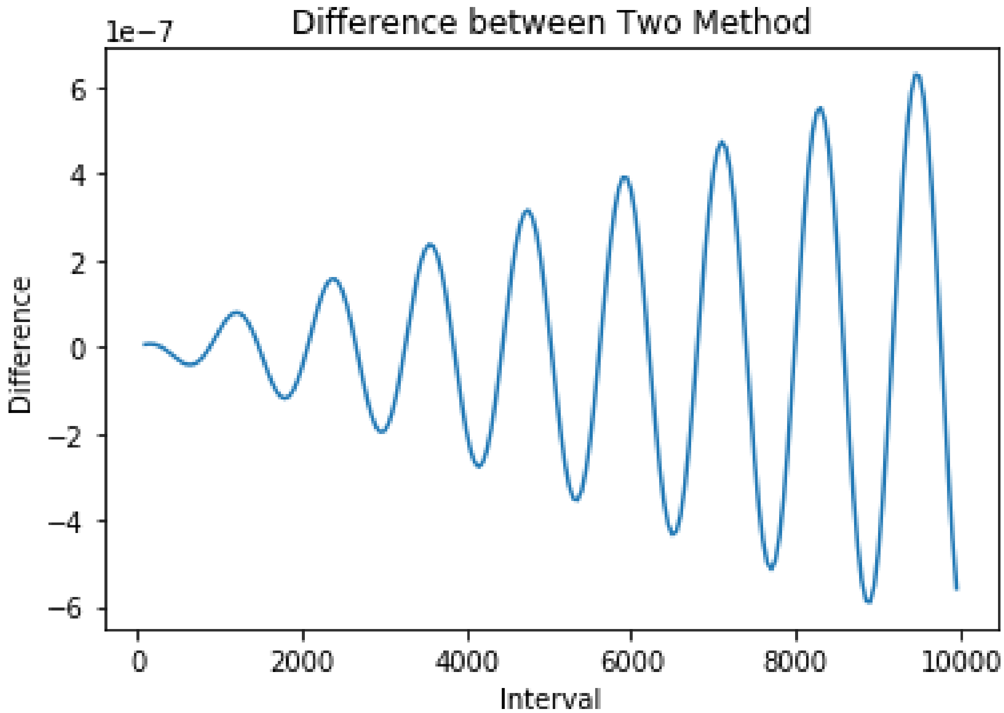


In [35]:
```
plt.plot(a_list,trunc2)
plt.xlabel('Interval')
plt.ylabel('Absolute Value of Truncation')
plt.title('Truncation of Simpson Method')
```

Out[35]: Text(0.5,1,'Truncation of Simpson Method')



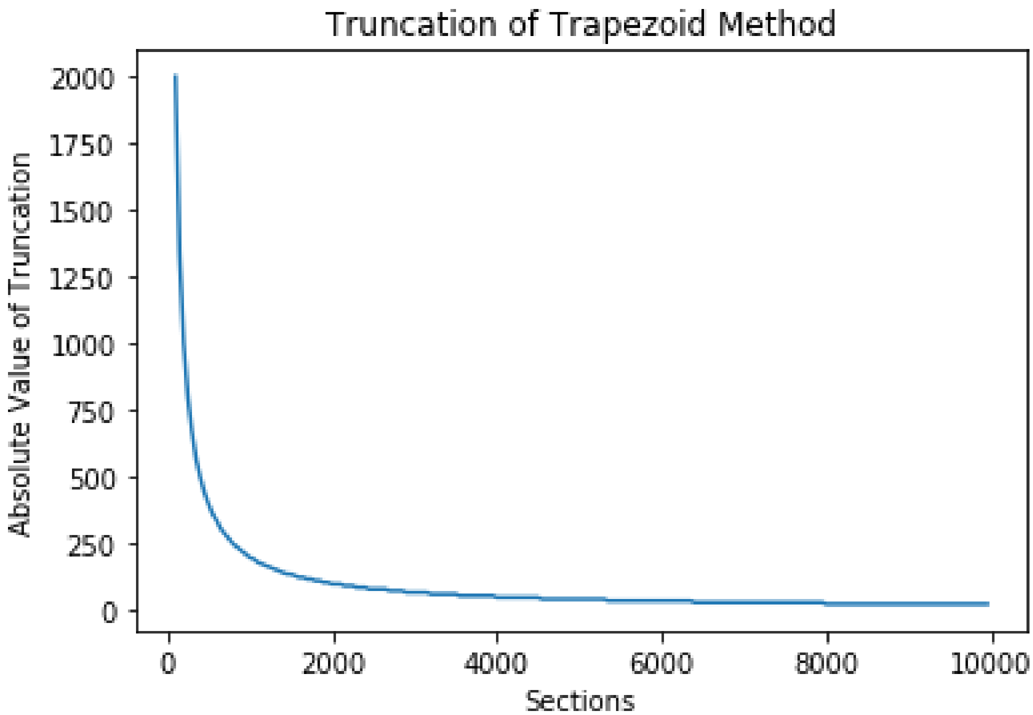

```
In [36]: plt.plot(a_list,dif)
         plt.xlabel('Interval')
         plt.ylabel('Difference')
         plt.title('Difference between Two Method')

Out[36]: Text(0.5,1,'Difference between Two Method')
```



These two method seems both converge to their limitation quickly, while from the third plot which is the results' difference between these two methods, we can tell that the gap of two integrals can flucatuate dramatically during the interval width goes large.

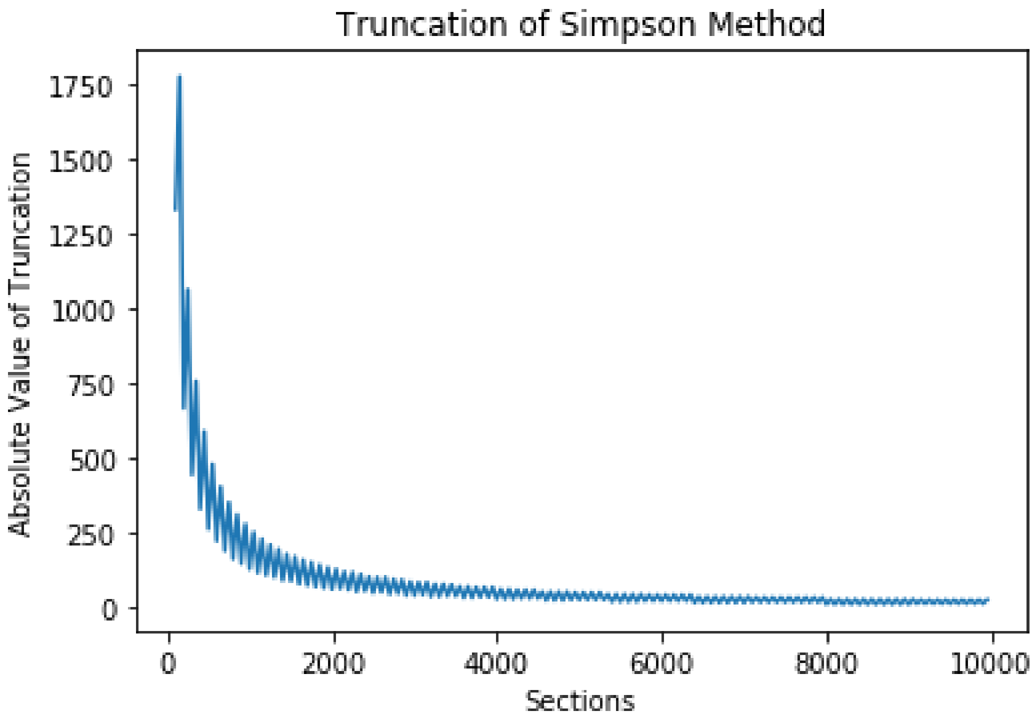
When fixing the interval and add the amount of sections:

```
In [37]: # fix the interval
         a = 10**5
         N_list=np.arange(100,10**4,50)
         trunc3 = [truncation_tra(a,n) for n in N_list]
         trunc4 = [truncation_sim(a,n) for n in N_list]
         dif2 = [difference(a,n) for n in N_list]

         plt.plot(N_list,trunc3)
         plt.xlabel('Sections')
         plt.ylabel('Absolute Value of Truncation')
         plt.title('Truncation of Trapezoid Method')

Out[37]: Text(0.5,1,'Truncation of Trapezoid Method')
```

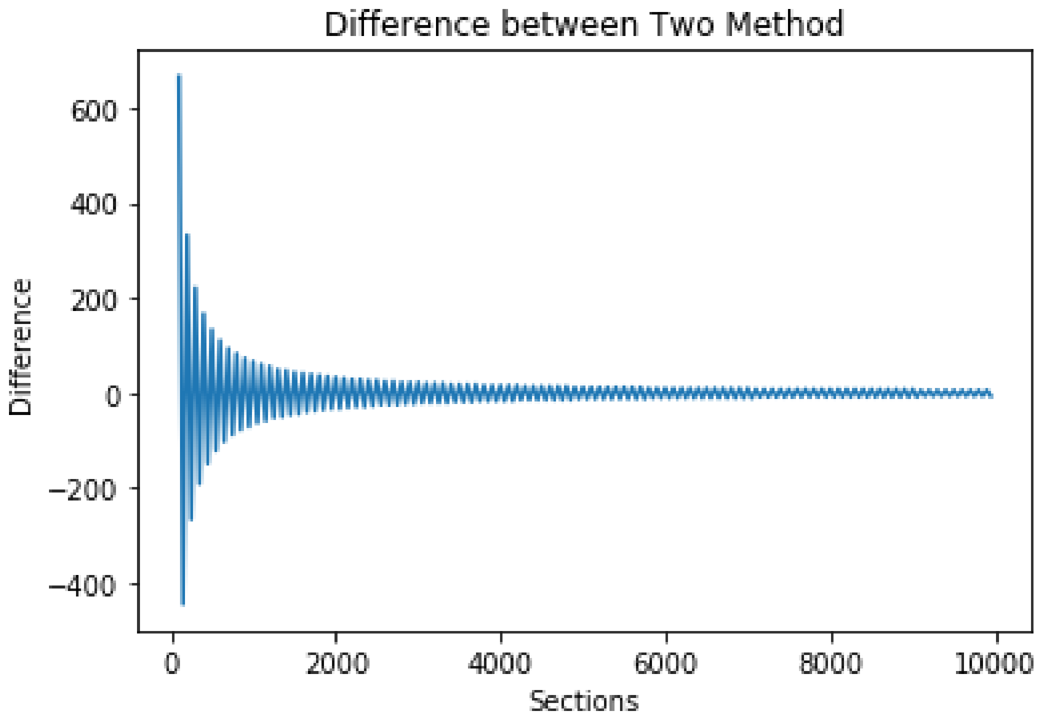


In [38]:
```
plt.plot(N_list,trunc4)
plt.xlabel('Sections')
plt.ylabel('Absolute Value of Truncation')
plt.title('Truncation of Simpson Method')
```

Out[38]:
```
Text(0.5,1,'Truncation of Simpson Method')
```




```
In [39]: plt.plot(N_list,dif2)
         plt.xlabel('Sections')
         plt.ylabel('Difference')
         plt.title('Difference between Two Method')

Out[39]: Text(0.5,1,'Difference between Two Method')
```



At this time, we fix the length of the interval that we integral and try to increase the number of sections. We can see that the simpson method, compared to the trapezoid method, is much more fluctruated when it approaching to the limitation. Howevr this time, the gap between this two methods are quickly going to 0 when we adding more sections into the integral.

3.3

Thus, to ensure the convergence of the numerical algorithms we pick a small tolerance value ...

Here we define a funciton to get the Intergral of a given funciton, and use the defination of the tolerence to be the guard of the loop.

```
In [50]: def Intergral(func, method, a, b, tolerance):
             n=1
             I_last = 0
             I_now = method(func,a,b,n)
             while (abs(I_now-I_last)>tolerance):
                 n=n+1
                 I_last = I_now
                 I_now = method(func,a,b,n)
             return I_now, '%d steps' % n
         result1,n1 = Intergral(func1,trapezoid_int,-(30000),(30000),10**-4)
         result2,n2 = Intergral(func1,simpson_int,-(30000),(30000),10**-4)
         print(result1,n1,result2,n2)
```

```
3.135482924547999 9611 steps 3.141626510960827 21103 steps
```

The trapezoid method can meet the condition in much less steps (9611<21103) however the result is not quite near to the real π. On the other hand, the simpson method although takes more steps, it can get to the final result in a more higher accuracy. The reason may be the defination of tolerence. The first method may not fluctruate during the converging process which leads to the fast speed to satisify this specific condition easily but still far away to its real limitation.

3.4

Intergral an arbitrary funciton

```
In [51]: def g_x(x):
             return 1+m.exp(-x**2)*m.cos(8*x**(2/3))

         Intergral(g_x,simpson_int,0,2,10**-4)
```

```
Out[51]: (1.9586979912115654, '6789 steps')
```

It takes 6789 steps to get the final answer which is 1.9587.