621 Final

May 17, 2019

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

1 Porblem A. Asian Option Pricing using Monte Carlo Control Variate

1.1

In this problem, we will construct the formula to price this geometric Asian call option.

```
In [2]: import numpy as np
        from scipy.stats import norm
        import time
        from sklearn import linear_model
        import os
        from pandas_datareader import data as pdrd
        import datetime as dt
        import pandas as pd
        from mpl_toolkits.mplot3d import Axes3D
        import matplotlib.pyplot as plt
        from scipy.interpolate import CubicSpline


        def analytical_asian(S0_,r_,sigma_,K_,T_):
            N = T_*252
            sigma_hat = sigma_*np.sqrt((2*N+1)/(6*(N+1)))
            rho = 0.5*(r_-0.5*sigma_**2+sigma_hat**2)
            d1 = (np.log(S0_/K_)+(rho+0.5*sigma_hat**2)*T_)/(np.sqrt(T_)*sigma_hat)
            d2 = (np.log(S0_/K_)+(rho-0.5*sigma_hat**2)*T_)/(np.sqrt(T_)*sigma_hat)
            Pg = np.exp(-r_*T_)*(S0_*np.exp(rho*T_)*norm.cdf(d1)-K_*norm.cdf(d2))
            return Pg

        pg = analytical_asian(S0_=100,r_=0.03,sigma_=0.3,K_=100,T_=5)
        print(pg)
```

```
15.171129680587903
```

The price of this asian option's price is about 15.1711.

1.2

Here we implement the Monte Carlo simulation to get the price.

```
In [7]: def Monte_Carlo_arith_asian(m_,T,S0,sig0,r,K,conf_level):
            start = time.time()
            dt = 1/252
            nudt = (r-sig0**2/2)*dt
            sigdt = sig0*np.sqrt(dt)
            dis = np.exp(-r*T)
            sum_C = 0
            sum_C2 = 0
            for i in range(int(m_)):
                lgXt = np.log(S0)
                sum_Xt = S0
                for j in range(int(T*252)):
                    z = np.random.randn()
                    lgXt += nudt+sigdt*z
                    sum_Xt += np.exp(lgXt)
                C = dis*max(sum_Xt/(T*252+1)-K,0)
                sum_C += C
                sum_C2 += C**2
            mean_C = sum_C/m_
            se = np.sqrt((sum_C2-m_*mean_C**2)/(m_-1)/m_)
            end = time.time()
            return mean_C,'[{},{}]'.format(mean_C-se*norm.ppf(conf_level),mean_C+se*norm.ppf(c
```

```
In [8]: pa_sim = Monte_Carlo_arith_asian(m_=1e4,T=5,S0=100,sig0=0.3,r=0.03,K=100,conf_level = (
```

```
In [9]: print(pa_sim)
```

```
(17.20143477832969, '[16.682849911408187,17.720019645251195]', 20.6727135181427)
```

The above is the price of an arithmetic asian option using the Monte Carlo simulation methods. The second one is the confidence interval when the confidence level is 0.95. It costs about 21 seconds to compute this result.

1.3

Now we implement a Monte Carlo scheme to price a geometric Asian Call option.

```
In [10]: def Monte_Carlo_geo_asian(m_,T,S0,sig0,r,K,conf_level):
             start = time.time()
             dt = 1/252
             nudt = (r-sig0**2/2)*dt
             sigdt = sig0*np.sqrt(dt)
             dis = np.exp(-r*T)
             sum_C = 0
```

```
        sum_C2 = 0
        for i in range(int(m_)):
            lgXt = np.log(S0)
            sum_Xt = lgXt
            for j in range(int(T*252)):
                z = np.random.randn()
                lgXt += nudt+sigdt*z
                sum_Xt += lgXt
            C = dis*max(np.exp(sum_Xt/(T*252+1))-K,0)
            sum_C += C
            sum_C2 += C**2
        mean_C = sum_C/m_
        se = np.sqrt((sum_C2-m_*mean_C**2)/(m_-1)/m_)
        end = time.time()
        return mean_C,'[{},{}]'.format(mean_C-se*norm.ppf(conf_level),mean_C+se*norm.ppf(c
```

```
In [12]: pg_sim = Monte_Carlo_geo_asian(m_=1e4,T=5,S0=100,sig0=0.3,r=0.03,K=100,conf_level = 0
         print(pg_sim)
```

```
(15.13154313784881, '[14.708489486233917,15.554596789463703]', 7.780169486999512)
```

The above is the price of an geometric asian option using the Monte Carlo simulation methods.

1.4

Now we implement a Monte Carlo scheme to find the relationship between the arithmetic Asian Option price and Geometric Asian Option price.

```
In [13]: def Monte_Carlo_get_b(m_,T,S0,sig0,r,K):
             dt = 1/252
             nudt = (r-sig0**2/2)*dt
             sigdt = sig0*np.sqrt(dt)
             dis = np.exp(-r*T)
             Xi = []
             Yi = []
             for i in range(int(m_)):
                 lgSt = np.log(S0)
                 sum_St1 = 0
                 sum_St2 = 0
                 for j in range(int(T*252)):
                     z = np.random.randn()
                     lgSt += nudt+sigdt*z
                     sum_St1 += np.exp(lgSt)
                     sum_St2 += lgSt
                 Xi.append(dis*max(sum_St1/(T*252+1)-K,0))
                 Yi.append(dis*max(np.exp(sum_St2/(T*252+1))-K,0))
             lr = linear_model.LinearRegression().fit(np.asarray(Xi).reshape(-1,1),np.asarray(Y
             b_star = lr.coef_
             return b_star[0,0]
```

```
In [14]: b_star = Monte_Carlo_get_b(m_=1e4,T=5,S0=100,sig0=0.3,r=0.03,K=100)
         print(b_star)
```

```
0.8550511911218861
```

We can get this b^* from doing the simulation. We can say that this two type of option have positive relationship.

1.5

Now we calculate the error of pricing the geometric Asian.

```
In [18]: Eg = pg-pg_sim[0]
         print(Eg)
```

```
0.03958654273909268
```

The absolute error is presented above.

1.6

Calculate the modified arithmetic option price using the reults above.

```
In [20]: pa_star = pa_sim[0]-b_star*Eg
         print(pa_star)
```

```
17.16758625780823
```

The modified value of the arithmetic asian price is 17.1675 and the result of the simulation is 17.2014. They are very similar to each other.

2 Problem B. A portfolio construction problem

2.1

Download the data from the yahoo finance. Then read those data from the local path. Here we only do the reading process, the download function is presented in the refernce. Here we use the XFL section and select the BAC, C, JPM and AFL as the stock.

```
In [4]: #stocklist = ['BAC','C','JPM','AFL','XLF']
        #def download(stocklist):
        #    starttime = dt.datetime(2012,1,1)
        #    endtime = dt.date.today()
        #    for stock in stocklist:
        #        equity_data = pdrd.DataReader(stock, data_source='yahoo',start=starttime,end=
        #        equity_data.to_csv('{} equity.csv'.format(stock))
        #download(stocklist)
```

```
os.chdir(r'D:\Grad 2\621\assignment\Final')
bac_p = pd.read_csv('BAC equity.csv',index_col=0,usecols = [0,6])
c_p = pd.read_csv('C equity.csv',index_col=0,usecols = [0,6])
jpm_p = pd.read_csv('JPM equity.csv',index_col=0,usecols = [0,6])
afl_p = pd.read_csv('AFL equity.csv',index_col=0,usecols = [0,6])
xlf_p = pd.read_csv('XLF equity.csv',index_col=0,usecols = [0,6])
```

2.2

Now we estimate the parameters values for each equity.

```
In [27]: bac = np.log(bac_p/bac_p.shift(1))[1:]
         c = np.log(c_p/c_p.shift(1))[1:]
         jpm = np.log(jpm_p/jpm_p.shift(1))[1:]
         afl = np.log(afl_p/afl_p.shift(1))[1:]
         xlf = np.log(xlf_p/xlf_p.shift(1))[1:]

         d_t = 1/255
         bac_theta2 = np.std(bac,ddof = 1)/np.sqrt(d_t)
         bac_theta1 = np.mean(bac)/d_t+0.5*bac_theta2**2

         c_theta2 = np.std(c,ddof = 1)/np.sqrt(d_t)
         c_theta1 = np.mean(c)/d_t+0.5*c_theta2**2

         jpm_theta2 = np.std(jpm,ddof = 1)/np.sqrt(d_t)
         jpm_theta1 = np.mean(jpm)/d_t+0.5*jpm_theta2**2

         afl_theta2 = np.std(afl,ddof = 1)/np.sqrt(d_t)
         afl_theta1 = np.mean(afl)/d_t+0.5*afl_theta2**2

         result_table1 = pd.DataFrame([[bac_theta1[0],bac_theta2[0]],
                                       [c_theta1[0],c_theta2[0]],
                                       [jpm_theta1[0],jpm_theta2[0]],
                                       [afl_theta1[0],afl_theta2[0]]],
                                      index = ['bac','c','jpm','afl'],
                                      columns = ['theta1','theta2'])
         print(result_table1)
```

```
       theta1    theta2
bac  0.267246  0.274968
c    0.156886  0.260348
jpm  0.209769  0.218083
afl  0.176451  0.180122
```

2.3

Estimate the correlation matrix Σ.

```
In [28]: stock_pool = pd.DataFrame([],index =  bac.index)
         stock_pool['bac'] = bac
         stock_pool['c'] = c
         stock_pool['jpm'] = jpm
         stock_pool['afl'] = afl
         corr_matrix = stock_pool.corr()
         print(corr_matrix)
```

```
          bac         c       jpm       afl
bac  1.000000  0.832754  0.811761  0.554781
c    0.832754  1.000000  0.835738  0.575161
jpm  0.811761  0.835738  1.000000  0.587724
afl  0.554781  0.575161  0.587724  1.000000
```

2.4

Then we contruct the geometric brownian motion for three stocks. The Euler-Milston method is introduced here to better produce the stock process.

```
In [29]: def Cholesky(A):
             n = A.shape[0]
             L = np.zeros(A.shape)
             L[0,0] = np.sqrt(A[0,0])
             for i in range(1,n):
                 L[i,0] = A[i,0]/L[0,0]
             for i in range(1,n):
                 for j in range(1,i+1):
                     if i == j:
                         L[j,j] = np.sqrt(A[j,j]-np.sum([(L[j,k])**2 for k in range(j)]))
                     else:
                         L[i,j] = (A[i,j]-np.sum([L[i,k]*L[j,k] for k in range(j)]))/L[j,j]
             return L

         def Monte_Carlo3(m_,T,S0,mu0,sig0,A):
             dt = 1/255
             result = []
             L = Cholesky(A)
             for i in range(int(m_)):
                 Xt = S0[0]
                 Yt = S0[1]
                 Zt = S0[2]
                 Qt = S0[3]
                 for j in range(int(T/dt)):
                     row_z = np.random.randn(4)
                     z = np.dot(L,row_z)
                     z1 = z[0]
                     z2 = z[1]
```

```
                z3 = z[2]
                z4 = z[3]
                Xt += mu0[0]*Xt*dt+sig0[0]*Xt*z1*np.sqrt(dt)+0.5*sig0[0]**2*(z1**2-1)*dt
                Yt += mu0[1]*Yt*dt+sig0[1]*Yt*z2*np.sqrt(dt)+0.5*sig0[1]**2*(z2**2-1)*dt
                Zt += mu0[2]*Zt*dt+sig0[2]*Zt*z3*np.sqrt(dt)+0.5*sig0[2]**2*(z3**2-1)*dt
                Qt += mu0[2]*Qt*dt+sig0[3]*Zt*z4*np.sqrt(dt)+0.5*sig0[3]**2*(z4**2-1)*dt
            result.append([Xt,Yt,Zt,Qt])
        return result
    A_ = np.matrix(corr_matrix)
    res = Monte_Carlo3(m_=1000,T=1,
                       S0=[bac_p.iloc[-1,0],c_p.iloc[-1,0],jpm_p.iloc[-1,0],afl_p.iloc[-1
                       mu0=[bac_theta1[0],c_theta1[0],jpm_theta1[0],afl_theta1[0]],
                       sig0=[bac_theta2[0],c_theta2[0],jpm_theta2[0],afl_theta2[0]],
                       A=A_)
    res = pd.DataFrame(res)
    print(res.head())
```

```
           0          1           2          3
0  39.718131  88.291557  136.018437  57.383277
1  29.439227  66.518875  120.224978  33.558189
2  27.986796  62.706287  172.965699  94.398685
3  34.778754  63.929147  115.828531  89.728165
4  29.844413  81.274294  127.644432  62.259990
```

The above is the partial rseults of the simulation. Now we give the statistics for each stock.

```
In [34]: result_table2 = pd.DataFrame([np.mean(res),np.std(res),
                                       res.kurtosis(),res.skew()],
                                      index = ['Mean','Std','Kurtosis','Skewness'],
                                      )
         result_table2.columns = ['bac','c','jpm','afl']
         print(result_table2)
```

```
                bac          c         jpm        afl
Mean      36.635454  75.050130  134.180325  65.941750
Std       10.333512  20.196452   30.045192  24.842678
Kurtosis   1.632136   1.750072    1.420349   0.557402
Skewness   0.882619   0.915068    0.732289   0.345098
```

2.5

Now we do the same thing on the ETF.

```
In [38]: xlf_theta2 = np.std(xlf,ddof = 1)/np.sqrt(d_t)
         xlf_theta1 = np.mean(xlf)/d_t+0.5*xlf_theta2**2
         result_table3 = pd.DataFrame([xlf_theta1[0],xlf_theta2[0]],
                                      index = ['mu','sigma'],
```

```
                                       columns = ['xfl'])
         print(result_table3)
```

```
            xfl
mu     0.199171
sigma  0.192629
```

2.6

Run a multivariate regression using the historical data. Find the weights for the basket option.

```
In [39]: respond_v = np.asmatrix(xlf)
         predictor_v = np.asmatrix(stock_pool)
         fit_model = linear_model.LinearRegression().fit(predictor_v, respond_v)

         weights = fit_model.coef_
         print(weights)
```

```
[[0.14241245 0.15980227 0.26767723 0.20447015]]
```

2.7

Now we price the nonstandard contract, Note that we use $r = 6\%$.

```
In [40]: def Monte_Carlo4(m_,T,S0,mu0,sig0,A,weights_,r,opt_type):
             dt = 1/255
             dis = np.exp(-r*T)
             L = Cholesky(A)
             sum_C = 0
             for i in range(int(m_)):
                 Xt = S0[0]
                 Yt = S0[1]
                 Zt = S0[2]
                 Qt = S0[3]
                 etf = S0[4]
                 for j in range(int(T/dt)):
                     row_z = np.random.randn(4)
                     z = np.dot(L,row_z)
                     z1 = z[0]
                     z2 = z[1]
                     z3 = z[2]
                     z4 = z[3]
                     z5 = np.random.randn()
                     Xt += mu0[0]*Xt*dt+sig0[0]*Xt*z1*np.sqrt(dt)+0.5*sig0[0]**2*(z1**2-1)*dt
                     Yt += mu0[1]*Yt*dt+sig0[1]*Yt*z2*np.sqrt(dt)+0.5*sig0[1]**2*(z2**2-1)*dt
                     Zt += mu0[2]*Zt*dt+sig0[2]*Zt*z3*np.sqrt(dt)+0.5*sig0[2]**2*(z3**2-1)*dt
                     Qt += mu0[3]*Qt*dt+sig0[3]*Qt*z4*np.sqrt(dt)+0.5*sig0[3]**2*(z4**2-1)*dt
```

```
            etf += mu0[4]*etf*dt+sig0[4]*etf*z5*np.sqrt(dt)+0.5*sig0[4]**2*(z5**2-1)*c
        Ut = np.dot(weights_,np.array([Xt,Yt,Zt,Qt]))[0]
        if opt_type == 'call':
            C = max(Ut-etf,0)*dis
        else:
            C = max(etf-Ut,0)*dis
        sum_C += C
    mean_C = sum_C/m_
    return mean_C
```

```
In [41]: basket1 = Monte_Carlo4(m_=1000,T=1,
                      S0=[bac_p.iloc[-1,0],c_p.iloc[-1,0],jpm_p.iloc[-1,0],afl_p.iloc[-1,0],xl
                      mu0=[bac_theta1[0],c_theta1[0],jpm_theta1[0],afl_theta1[0],xlf_theta1[0]
                      sig0=[bac_theta2[0],c_theta2[0],jpm_theta2[0],afl_theta2[0],xlf_theta2[0
                      A=A_,
                      weights_=weights,
                      r = 0.06,
                      opt_type = 'call')
         basket2 = Monte_Carlo4(m_=1000,T=1,
                      S0=[bac_p.iloc[-1,0],c_p.iloc[-1,0],jpm_p.iloc[-1,0],afl_p.iloc[-1,0],xl
                      mu0=[bac_theta1[0],c_theta1[0],jpm_theta1[0],afl_theta1[0],xlf_theta1[0]
                      sig0=[bac_theta2[0],c_theta2[0],jpm_theta2[0],afl_theta2[0],xlf_theta2[0
                      A=A_,
                      weights_=weights,
                      r = 0.06,
                      opt_type = 'put')
```

```
In [42]: print(basket1,basket2)
```

```
31.349453088475773 0.027829355753707027
```

For the first option, the price is about 30 and for the second option, the price premium is very small.

3 Problem C. Local Volatility

3.1

First of all, we need to read the data and find the implied volatility by bisection method.

```
In [5]: spx = pd.read_excel('SPX.xls', header = None)
        td_data = spx.iloc[0,:-1]
        spx = spx.iloc[1:,:]
        spx.columns = spx.iloc[0,:]
        spx = spx.reindex(spx.index[1:])


        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 BS_Formula(type_opt, r, vol, K, S, T):
        d_1 = float(np.log(S/K)+(r+vol**2/2)*T)/float(vol*np.sqrt(T))
        d_2 = d_1-vol*np.sqrt(T)
        if type_opt == 'call':
            return norm.cdf(d_1)*S-K*np.exp(-r*T)*norm.cdf(d_2)
        else:
            return K*np.exp(-r*T)*norm.cdf(-d_2)-norm.cdf(-d_1)*S

    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)

    for ind in spx.index:
        spx.loc[ind,'Implied_vol_bis'] = get_iv('call', td_data[2]/100, spx.loc[ind,'K']\
                                    , td_data[1], spx.loc[ind,'T'], spx.loc[ind,'Price'],
                                    10**(-6), 1, 0.00001)
    spx_result = spx.dropna()
```

```
In [6]: print(spx_result.head())
```

1	Date	T	K	Price	Implied_vol_bis
2	39892	0.0712329	850	0.45	0.192977
3	39892	0.0712329	875	0.6	0.250505
4	39892	0.0712329	900	0.15	0.243389
10	39892	0.0712329	400	370.25	0.735934
11	39892	0.0712329	425	345.35	0.802588

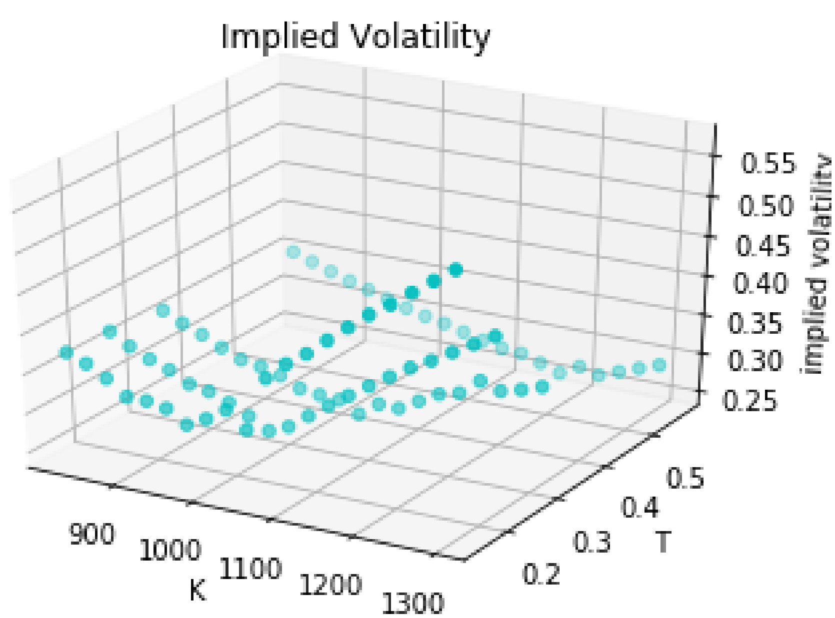
Now we plot the points with $T = 39920, 39948, 39983, 40074$ and $K = 825, 850, 875, 900, 825, 850, 975, 1000, 1125, 1150, 1175, 1200, 1225, 1250, 1275, 1300$.

```
In [7]: dd = spx_result[(spx_result['Date']==40074) \
                        | (spx_result['Date']== 39920) \
                        | (spx_result['Date']== 39948) \
                        | (spx_result['Date']== 39983)]
```

```
dd_ = dd.drop_duplicates(subset = 'Date',keep = 'first')
dd_date = dd_['Date']
K_list = dd['K']
for i in dd_date:
    B = dd.loc[dd['Date'] == i]
    B = B.loc[:,'K']
    B = B.drop_duplicates()
    K_list = np.intersect1d(K_list,B)
K_list = list(K_list[10:33])
data_c = dd.loc[dd['K'].isin(K_list)]
data_c = data_c.sort_values(by=['Date','K'])
data_c= data_c.drop_duplicates(subset = ['Date','K'],keep = 'first')
data_c.index = range(len(data_c))
```

In [8]:
```
x_ = np.array(list(data_c.loc[:,'K']))
y = np.asarray(list(data_c.loc[:,'T']))
z = np.array(list(data_c.loc[:,'Implied_vol_bis']))

fig = plt.figure()
ax = fig.gca(projection='3d')
ax.scatter(x_,y,z,c='c')
ax.set_title('Implied Volatility')
ax.set_xlabel('K')
ax.set_ylabel('T')
ax.set_zlabel('implied volatility')
```

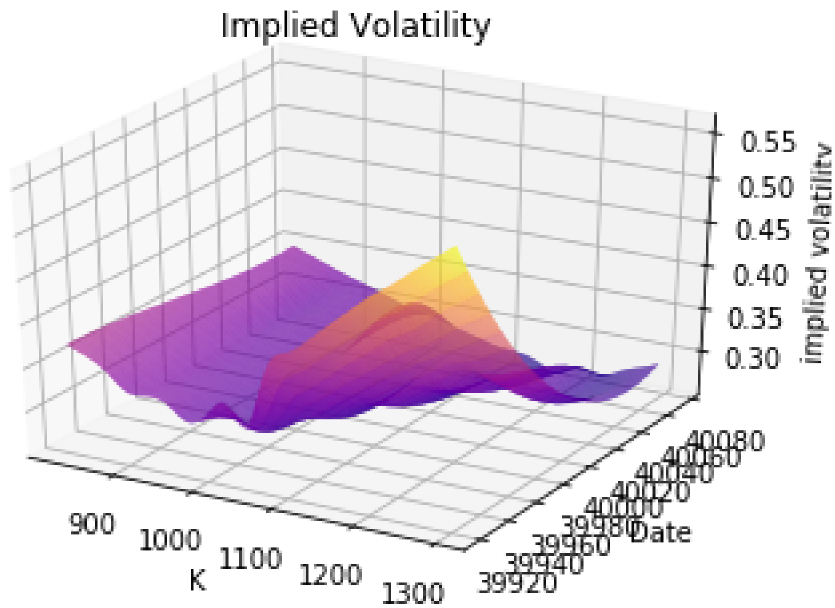
Out[8]:
```
Text(0.5,0,'implied volatility')
```



3.2

After cubic interpolation, we can plot the volatility surface.

```
In [9]: from scipy.interpolate import CubicSpline
        x = np.asarray(K_list[::])
        z1 = np.array(z[:20])
        z2 = np.array(z[20:40])
        z3 = np.array(z[40:60])
        z4 = np.array(z[60:80])
        cs1 = CubicSpline(x, z1,axis = 1)
        cs2 = CubicSpline(x, z2,axis = 1)
        cs3 = CubicSpline(x, z3,axis = 1)
        cs4 = CubicSpline(x, z4,axis = 1)
        cs = [cs1,cs2,cs3,cs4]
        xs = np.linspace(K_list[0],K_list[-1],500)
        xs2 = list(xs)*4
        ys2 = []
        zs2 = []
        for i in range(len(dd_date)):
            ys2+=[dd_date.iloc[i]]*500
            zs2+=list(cs[i](xs))
        z_1 = zs2[:500]
        z_2 = zs2[500:1000]
        z_3 = zs2[1000:1500]
        z_4 = zs2[1500:2000]
        cs_m = []*500
        for i in range(len(xs)):
            temp = CubicSpline(list(dd_date),[z_1[i],z_2[i],z_3[i],z_4[i]],axis = 1)
            cs_m.append(temp)
        ds = np.linspace(dd_date.min(),dd_date.max(),20)
        ys1 = list(ds)*500
        xs1 = []
        zs1 = []
        for i in range(len(xs)):
            xs1+=[xs[i]]*20
            zs1 += list(cs_m[i](ds))
        fig = plt.figure()
        ax = fig.gca(projection='3d')
        ax.plot_trisurf(xs1,ys1,zs1,cmap='plasma')
        ax.set_title('Implied Volatility')
        ax.set_xlabel('K')
        ax.set_ylabel('Date')
        ax.set_zlabel('implied volatility')

Out[9]: Text(0.5,0,'implied volatility')
```



3.3

For the points on the surface, I think there maybe some arbitrage. The calender arbitrage may hold when the volatility decrease along the time axis. As we can see in the plot we draw, the overall volatility goes down when the maturity goes up.

3.4

Now we compute the local volatility for such 80 points.

```
In [10]: def local_v(S_,K_,tau_,sigma_,r_,s2t,s2k,s2k2):
             d1 = (np.log(S_/K_)+(r_+0.5*sigma_**2)*tau_)/(sigma_*np.sqrt(tau_))
             up = (2*sigma_*s2t*tau_)+sigma_**2+2*sigma_*r_*tau_*K_*s2k
             down = ((1+K_*d1*s2k*np.sqrt(tau_))**2+K_**2*tau_*sigma_*(s2k2-d1*s2k**2*np.sqrt(
             result = np.sqrt(up/down)
             return result

         spx_result = spx_result.sort_values(by=['Date','K'])
         spx_result.index = range(len(spx_result))
         datt = spx.drop_duplicates(subset = 'Date',keep = 'first')['Date']
         new_dt = []
         for i in datt:
             frame = spx_result.loc[spx_result['Date'] == i]
             frame = frame.drop_duplicates(subset = 'K',keep = 'first')
             s2k_list = (np.asarray(frame['Implied_vol_bis'])[1:]-np.asarray(frame['Implied_vo
             (np.asarray(frame['K'])[1:]-np.asarray(frame['K'])[:-1])
             s2k2_list = (s2k_list[1:]-s2k_list[:-1])/(np.asarray(frame['K'])[1:-1]-np.asarray
```

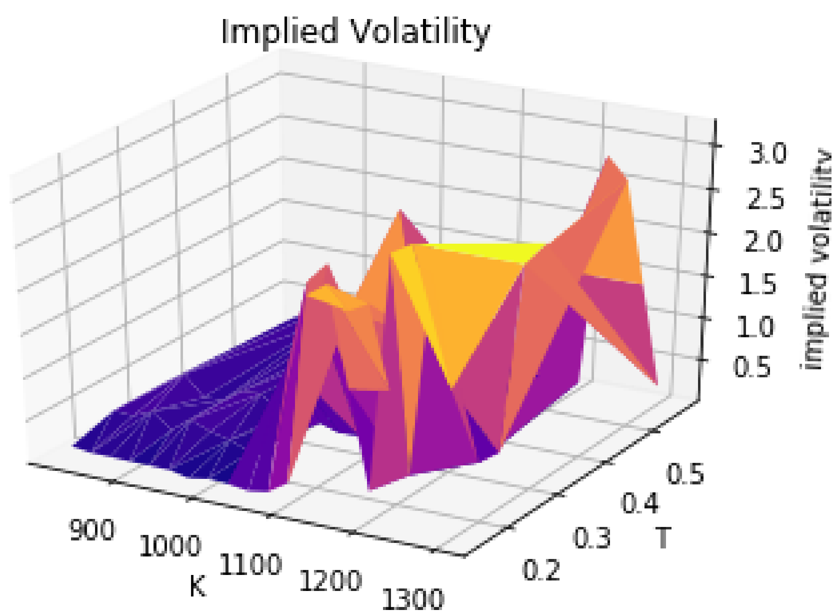
```
            new_frame = frame.iloc[:-2,:]
            new_frame['s2k'] = s2k_list[:-1]
            new_frame['s2k2'] = s2k2_list[::]
            new_dt.append(new_frame)
        new_dt = pd.concat(new_dt)
        new_dt = new_dt.sort_values(by=['K','Date'])
        new_dt.index = range(len(new_dt))
        katt = new_dt.drop_duplicates(subset = 'K',keep = 'first')['K']
        new_dt2 = []
        for i in katt:
            frame = new_dt.loc[new_dt['K'] == i]
            frame = frame.drop_duplicates(subset = 'Date',keep = 'first')
            if len(frame) == 1:
                continue
            else:
                s2t_list = (np.asarray(frame['Implied_vol_bis'])[1:]-np.asarray(frame['Implie
                (np.asarray(frame['T'])[1:]-np.asarray(frame['T'])[:-1])
                new_frame = frame.iloc[:-1,:]
                new_frame['s2t'] = s2t_list
                new_dt2.append(new_frame)
        new_dt2 = pd.concat(new_dt2)

        new_dt2 = new_dt2.sort_values(by=['Date','K'])
        new_dt2.index = range(len(new_dt2))
        for i in range(len(new_dt2)):
            new_dt2.loc[i,'lv'] = local_v(S_=td_data[1],K_ = new_dt2.loc[i,'K'],tau_ = new_dt
                        sigma_ = new_dt2.loc[i,'Implied_vol_bis'],r_ = td_data[2]/100,s2t = ne
                        s2k = new_dt2.loc[i,'s2k'],s2k2 = new_dt2.loc[i,'s2k2'])
        new_dt2['lv'] = new_dt2['lv'].interpolate()

        data_c2 = new_dt2[(new_dt2['Date'].isin(dd_date))&(new_dt2['K'].isin(K_list))]
        data_c2.index = range(len(data_c2))
```

```
D:\Anaconda3\lib\site-packages\ipykernel_launcher.py:18: 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.htm
D:\Anaconda3\lib\site-packages\ipykernel_launcher.py:19: 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.htm
D:\Anaconda3\lib\site-packages\ipykernel_launcher.py:34: 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.htm
```

```
D:\Anaconda3\lib\site-packages\ipykernel_launcher.py:5: RuntimeWarning: invalid value encounte
  """
```

```
In [11]: xs2d = list(data_c2['K'])
         ys2d = list(data_c2['T'])
         zs2d = list(data_c2['lv'])
         fig = plt.figure()
         ax = fig.gca(projection='3d')
         ax.plot_trisurf(np.asarray(xs2d),np.asarray(ys2d),
                         np.asarray(zs2d),cmap='plasma')
         ax.set_title('Implied Volatility')
         ax.set_xlabel('K')
         ax.set_ylabel('T')
         ax.set_zlabel('implied volatility')
```

```
Out[11]: Text(0.5,0,'implied volatility')
```



Here we use the linear interpolation to plot this local volatility surface. Compared to the previous one, this surface also present little property of volatility smile, however the volatility is not intuitivly meaningful by the values. The local volatility can be very different from the implied volatility computed before.

3.5

Now we use the B-S fomular to calculate the price using the local volatility.

```
In [12]: for i in range(len(new_dt2)):
             new_dt2.loc[i,'p_bs'] = BS_Formula(type_opt='call', r = td_data[2]/100, vol = new_
```

```
                        K = new_dt2.loc[i,'K'], S = td_data[1], T = new_dt2.loc[i,'T'])
        new_dt2.loc[i,'p_lv'] = BS_Formula(type_opt='call', r = td_data[2]/100, vol = new_
                        K = new_dt2.loc[i,'K'], S = td_data[1], T = new_dt2.loc[i,'T'])
    print(new_dt2.head())
```

```
1   Date          T    K   Price  Implied_vol_bis          s2k         s2k2  \
0  39892  0.0712329  400  370.25         0.735934   0.00266614 -8.13911e-05
1  39892  0.0712329  450  320.55         0.818372  -0.00110604  1.54391e-05
2  39892  0.0712329  475  295.75         0.790721 -0.000720065 -1.62441e-05
3  39892  0.0712329  490  280.95         0.779920 -0.000963726 -1.92621e-05
4  39892  0.0712329  500   271.1         0.770282  -0.00115635 -8.27857e-06

1       s2t        lv        p_bs        p_lv
0 -0.405273  0.396819  370.250000  370.238011
1 -0.860125  0.969088  320.550001  321.298292
2  -1.39941  1.081460  295.750001  298.634172
3  -2.83178  1.096927  280.950000  284.987582
4  -2.62936  0.989658  271.100000  273.753074
```

Here we present the a part of the result we obtained, the column 'p_lv' is the price we calculated from the local volatility. They are accurate in some price level. However the price we compute fluctuate dramaticaly.

3.6

Present the table and write the data into a file.

```
In [13]: print(new_dt2.loc[:7,['T','K','Price','Implied_vol_bis','lv','p_lv']].head())
```

```
1          T    K   Price  Implied_vol_bis        lv        p_lv
0  0.0712329  400  370.25         0.735934  0.396819  370.238011
1  0.0712329  450  320.55         0.818372  0.969088  321.298292
2  0.0712329  475  295.75         0.790721  1.081460  298.634172
3  0.0712329  490  280.95         0.779920  1.096927  284.987582
4  0.0712329  500   271.1         0.770282  0.989658  273.753074
```

```
In [14]: new_dt2.to_csv('SPXvolatility.csv')
```