621_Homework_3_Guanzhuo_Qiao

April 21, 2019

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

1 Problem 1. Quadratic Volatility Model

1.1

In this quesiton, we need to construct the transiition probability function and then we plot the surface. The transition probability is presented below.

$$p(t,x|x_0) = \frac{1}{\sigma(x)\sqrt{2\pi t}}\frac{\sigma(x_0)}{\sigma x}exp(-\frac{1}{2t}(\int_{x_0}^{x} x^2\frac{\mathrm{d}u}{\sigma(u)}) + Qt)$$

Once we plug the x, x_0 into the formula and fix the value of t, we can get the probability. Note that we need to select the proper parameters of α, β, γ in order to let the volatility to locate on a normal level. Here we use $\alpha = 0.0001, \beta = 0.001$ and $\gamma = -0.0025$, with the $x_0 \in [40, 50]$ and $x_0 \in [30, 60]$ plot in the graph. We plot two times graph, one is for $t = 10$ and another is for $t = 40$.

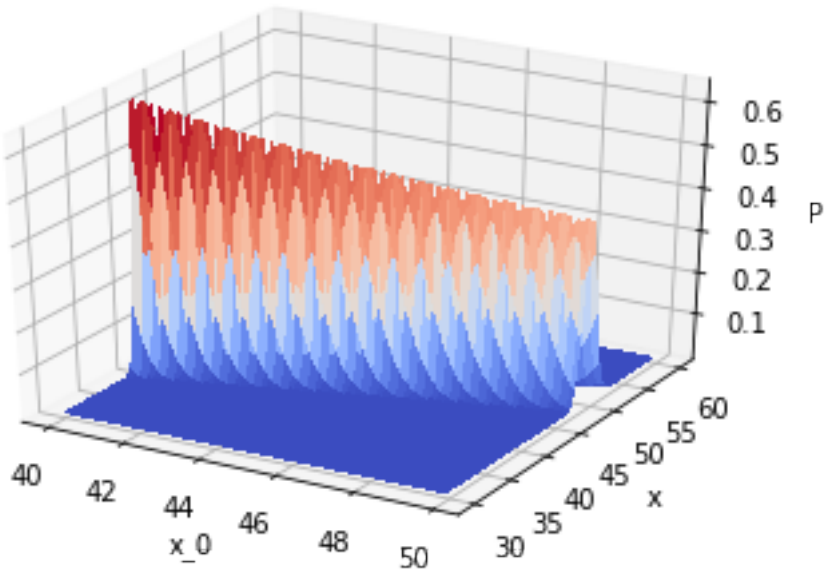
```
In [5]: import scipy.integrate as integrate
        import math
        from mpl_toolkits.mplot3d import Axes3D
        import matplotlib.pyplot as plt
        from matplotlib import cm
        import numpy as np
        from scipy.stats import norm

        # construc the transition density
        def transition_p(t,x,x0,alpha,gamma,beta):
            Q = alpha*gamma/2-beta**2/8
            def sigma(x):
                return (alpha*x**2+beta*x+gamma)
            s_x = integrate.quad(lambda xx: 1/sigma(xx),x0,x)[0]
            res = 1/(sigma(x)*math.sqrt(2*math.pi*t))*sigma(x0)/sigma(x)\
            *math.exp(-s_x**2/2/t+Q*t)
            return res
        # give the range of the point
        X = np.arange(40, 50, 0.1)
        Y = np.arange(30, 60, 0.1)
```

```
X, Y = np.meshgrid(X, Y)
Z = np.zeros(X.shape)
for i in range(X.shape[0]):
    for j in range(X.shape[1]):
        Z[i][j] = transition_p(t=10,x=Y[i][j],x0=X[i][j],
                               alpha=0.0001,gamma=-0.0025,beta=0.001)

# Plot the surface.
fig = plt.figure()
ax = fig.gca(projection='3d')
surf = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm,
                       linewidth=0, antialiased=False)
ax.set_xlabel("x_0")
ax.set_ylabel("x")
ax.set_zlabel("P")
```

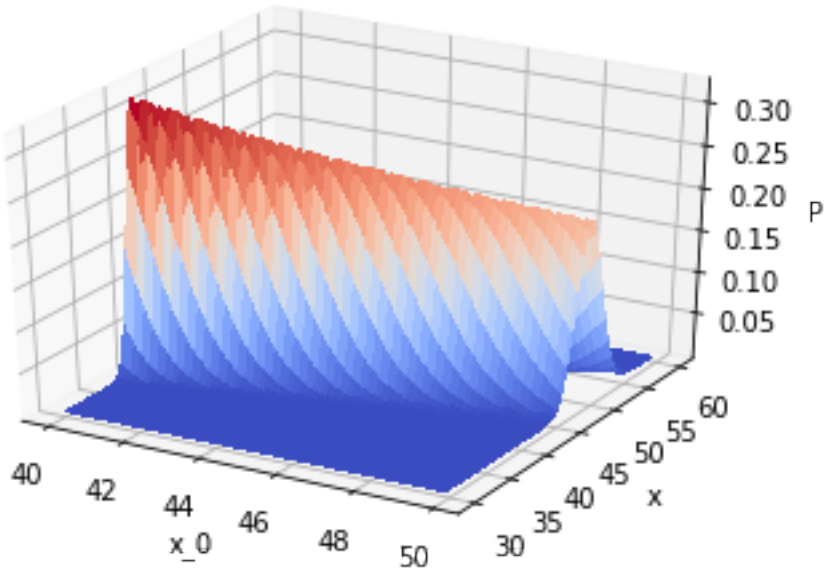
Out[5]: Text(0.5,0,'P')



In [4]:
```
X = np.arange(40, 50, 0.1)
Y = np.arange(30, 60, 0.1)
X, Y = np.meshgrid(X, Y)
Z = np.zeros(X.shape)
for i in range(X.shape[0]):
    for j in range(X.shape[1]):
        Z[i][j] = transition_p(t=40,x=Y[i][j],x0=X[i][j],
                               alpha=0.0001,gamma=-0.0025,beta=0.001)
```

```
        # Plot the surface.
        fig = plt.figure()
        ax = fig.gca(projection='3d')
        surf = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm,
                               linewidth=0, antialiased=False)
        ax.set_xlabel("x_0")
        ax.set_ylabel("x")
        ax.set_zlabel("P")
```

```
Out[4]: Text(0.5,0,'P')
```



From the two plot above, first of all we can tell that the transition probability is higher when x is near to x_0 which is very appropriate. The curve is "thinner" when x_0 being small. Becuase there is less volatility due to this quadratic function, so the probability to transit is small, the point is more likely to stay rather than transit.

When t becomes larger, the averall curve is "fatter" because when we increase maturities, there is more likely to allow the x_0 transit to other possible points therefore the curve is fatter.

1.2 Bonus

We use the finite difference method to calculate the transition probability and we compare the number with what we done in the first quesiton. To illustrate the error, we use the Mean Square error to indicate the difference between two methods.

In practice, we first generate the inital probability which is 1 when $x = x_0$ and 0 elsewhere, and using the PDE function to simulate the following level. in the final, we give the probability of the last layer.

```
In [21]: def explicit_fd_grid(T,N,dx,Nj,x0,alpha,beta,gamma):
             #initaite the grid
```

```
            delta_t = T/N
            def sigma(x):
                return (alpha*x**2+beta*x+gamma)
            p = np.asarray([0]*(2*Nj+1))
            p[Nj] = 1
            X_ = np.asarray([x0+Nj*dx-i*dx for i in range(2*Nj+1)])
            sig_X_ = np.vectorize(sigma)(X_[1:-1])
            p_u = sig_X_**2*delta_t/(2*dx**2)
            p_m = 1-sig_X_**2*delta_t/(1*dx**2)
            p_d = sig_X_**2*delta_t/(2*dx**2)
            # generate through the grid
            N_ = N
            while (N_>0):
                dis_p = (p_u*p[:-2]+p_m*p[1:-1]+p_d*p[2:])
                p_large = dis_p[0]
                p_small = dis_p[-1]
                p = np.concatenate(([p_large],dis_p,[p_small]))
                N_ -= 1
            return p
        # find the difference between two methods when x0:50 -> x:[10,90]
        nj = 40
        p_=[]
        for i in range(50+nj,50-nj-1,-1):
            p_.append(transition_p(t=2,x=i,x0=50,alpha=0.0001,gamma=0,beta=0.1))
        p_ = np.asarray(p_)
        p = explicit_fd_grid(T=2,N=1000,dx=1,Nj=nj,x0=50,alpha=0.0001,beta=0.1,gamma=0)
        mse = np.mean((p-p_)**2)
        print(mse)
```

```
3.611057583041696e-05
```

As we can see, the error is relatively small, which refers that the transition probability density satisfies the initial PDE.

1.3

Using the equaiton presented, we can give an European call option's price by using this model. Note that we need to set α and γ equal to 0 to let this method and BS formula comparable. In such a situation, β will be the

```
In [9]: def C(t_, K, x0,alpha,gamma,beta):
            Q = alpha*gamma/2-beta**2/8
            def sigma(x):
                return (alpha*x**2+beta*x+gamma)
            s = abs(integrate.quad(lambda xx: 1/sigma(xx),x0,K)[0])
            dis1 = math.exp(s*math.sqrt(-Q))
            dis2 = math.exp(-s*math.sqrt(-Q))
            d1 = -s/math.sqrt(2*t_)-math.sqrt(-2*Q*t_)
```

```
            d2 = d1+2*math.sqrt(-2*Q*t_)
            res = max(x0-K,0)+sigma(K)*sigma(x0)/(2*math.sqrt(-2*Q))*\
                    (dis1*norm.cdf(d1)-dis2*norm.cdf(d2))
            return res
        def BS_Formula(type_opt, r, vol, K, S, T):
            d_1 = float(math.log(S/K)+(r+vol**2/2)*T)/float(vol*math.sqrt(T))
            d_2 = d_1-vol*math.sqrt(T)
            if type_opt == 'call':
                return norm.cdf(d_1)*S-K*math.exp(-r*T)*norm.cdf(d_2)
            else:
                return K*math.exp(-r*T)*norm.cdf(-d_2)-norm.cdf(-d_1)*S
```

```
In [10]: p1 = C(t_=1, K=90, x0=100,alpha=0,gamma=0,beta=0.03)
         p2 = BS_Formula(type_opt='call', r=0, vol=0.03, K=90, S=100, T=1)
         print(p1,p2)
```

```
9.98291177988507 10.000158644283331
```

The results are very close to each other.

To be clear, Here we find a optimal parameter to let these two prices be equal. The parameters must match the spot price, strike price and maturities. If we change the parameter we may get an answer far away from the theoretic value. The selection of the parameter may need real market data to fit this quadratic model. It's not simple to give the maximum likelyhood funciton of this model, so here we just find the best parameter to illustrate the conclusion.

Here we use another parameter set.

```
In [11]: p3 = C(t_=1, K=90, x0=100,alpha=0.0001,gamma=-3,beta=0.1)
         print(p3)
```

```
5.082938731633514
```

2 Problem 2. Fast Fourier Transform

2.1

In this question, we use fast fourier transform to give the price the a option.

First of all, we use the modified call price $c_T(k)$ to do the Fourier transform.

$$\psi_T(v) = \int_{-\infty}^{\infty} e^{ivk} c_T(k)\mathrm{dk} = \int_{-\infty}^{\infty} \mathrm{e}^{\mathrm{ivk}}\mathrm{e}^{\mathrm{ffk}}\mathrm{C_T(k)dk}$$

Then use the inverse transform.

$$C_T(k) = \frac{exp(-\alpha k)}{2\pi}\int_{-\infty}^{\infty} e^{-ivk}\psi_T(v)\mathrm{dv} = \frac{\exp(-\mathrm{ffk})}{\text{ß}}\int_0^{\infty} \mathrm{e}^{-\mathrm{ivk}}\blacksquare(\mathrm{v})\mathrm{dv}$$

Where $\psi(v)$ is the real part of $\psi_T(v)$. What's more, we have an explicit expression of $\psi_T(v)$.

$$\psi_T(v) = \frac{e^{-rT}\phi_T(v-(\alpha+1)i)}{\alpha^2+\alpha-v^2+i(2\alpha+1)v}$$

By using the trapezoid rule to compute the integral, we can manipulate the form ro become a FFT transform and use the transform to get the final result.

$$C_T(k_u) \approx \frac{exp(-\alpha k_u)}{\pi} \sum_{j=1}^{N} e^{-i\lambda\eta(j-1)(u-1)} e^{ibv_j} \psi_T(v_j)\eta$$

In practice, we need to generate a set of $e^{ibv_j}\psi_T(v_j)\eta$ and use FFT get a set of summation after multipling the modifier we get the result which is $C(k_u)$. Then we pick the price we need to compare with BS results.

Note that $\alpha and \eta$ are chosen by us, we need let the price cover what we want. Here we choose $\alpha = 10$ and $\eta \approx 0.31415$ in order to let the interval small enough to approach the strike price.

```
In [13]: def phi(t,mu,sigma):
             return np.exp(np.complex(-sigma**2*t**2/2,t*mu))

         def PHI(v,r,T,alpha,s0,sigma_):
             value_phi = phi(np.complex(v,-(alpha+1)),np.log(s0)+\
                             (r-sigma_**2/2)*T,sigma_*np.sqrt(T))
             up = np.exp(-r*T)*value_phi
             down = np.complex(alpha**2+alpha-v**2,(2*alpha+1)*v)
             return up/down

         def input_generator(v,b,eta,r,T,alpha,s0,sigma_):
             A = np.exp(np.complex(0,b*v))
             B = PHI(v,r,T,alpha,s0,sigma_)
             #B = B.real
             C = eta
             return A*B*C

         N = 2000
         k = np.log(80)
         b_ = np.ceil(k)+5
         lambda_ = 2*b_/N
         eta_ = 2*np.pi/N/lambda_
         alpha_ = 10
         vv = np.array([eta_*i for i in range(0,N)])
         kk = np.array([np.round(-b_+lambda_*i,2) for i in range(0,N)])
         for ind in range(len(kk)):
             if kk[ind] == np.round(k,2):
                 break
         J = 1
         input_array = [0]*N
         # gnerate the input data
         while J<=N:
             input_array[J-1] = input_generator(v=vv[J-1],b=b_,eta=eta_,
                                 r=0.05,T=1,alpha=alpha_,s0=80,sigma_=0.1)
             J=J+1
         input_array = np.asarray(input_array)
```

```
res1 = np.fft.fft(input_array)[ind].real*np.exp(-alpha_*kk[ind])/np.pi
res2 = BS_Formula(type_opt='call', r=0.05, vol=0.1, K=80, S=80, T=1)
print(res1,res2)
```

```
5.654355836943183 5.4439661670577095
```

As we can see, the two results are very close to each other.

3 Bonus: SABR parameter estimation

3.1

When $\beta = 0.5$ we use the least square method to minimize the following target function and get the parameters.

$$(\hat{\alpha}, \hat{\rho}, \hat{\nu}) = \underset{\alpha,\rho,\nu}{argmin} \sum_i \{\sigma_i^{mkt} - \sigma B(f_i, K_i; \alpha, \rho, \nu)\}^2$$

Since we use the at-the-money contracts, the strike price is equal to spot price. Here we use the optimizer in python to conduct method. But first of all, we need to define the target function.

```
In [15]: import os
         import pandas as pd
         import scipy as sp
         os.chdir(r'D:\Grad 2\621\assignment\HW3')
         swap_dt = pd.read_excel('2017_2_15_mid.xlsx')
         swap_dt = swap_dt.loc[:,["Expiry","Unnamed: 1","3Yr","10Yr"]]
         swap_vol = swap_dt.loc[::2,"3Yr"]
         swap_strike = swap_dt.loc[1::2,"3Yr"]

         def sig_B(p,f,beta_,T_):
             alpha,rho,vega = p
             A = pow(1-beta_,2)*pow(alpha,2)/(24*pow(f,2-2*beta_))
             B = rho*beta_*vega*alpha/(4*pow(f,1-beta_))
             C = (2-3*pow(rho,2))*pow(vega,2)/24
             sig_B = alpha*(1+(A+B+C)*T_)/f**(1-beta_)
             return sig_B

         def sum_error(p,f,y,beta_,T_):
             error = np.sum((y-sig_B(p,f,beta_,T_))**2)
             return error
```

Then we do the optimize with the initial point of $P_0(0.1, 0.1, 0.5)$.

```
In [28]: p0 = [0.1,0.1,0.1]
         res_1 = sp.optimize.minimize(sum_error,p0,args=\
                 (np.asarray(swap_strike)/100,np.asarray(swap_vol)/10000,0.5,3))
         parameters_1 = res_1['x']
         sse_1 = res_1['fun']
         print(parameters_1,sse_1)
```

```
[0.00121395 0.09842321 0.09503974] 5.373834577569829e-05
```

3.2

Repeat the process.

```
In [29]: res_2 = sp.optimize.minimize(sum_error,p0,
                 args=(np.asarray(swap_strike)/100,np.asarray(swap_vol)/10000,0.7,3))
         res_3 = sp.optimize.minimize(sum_error,p0,
                 args=(np.asarray(swap_strike)/100,np.asarray(swap_vol)/10000,0.4,3))
         parameters_2 = res_2['x']
         parameters_3 = res_3['x']
         sse_2 = res_2['fun']
         sse_3 = res_3['fun']
         print(parameters_2,sse_2)
         print(parameters_3,sse_3)
```

```
[0.00255453 0.09774695 0.09017037] 4.0365581773592046e-05
[0.000837   0.08498348 0.05685423] 6.149337799019164e-05
```

3.3

Here we produce a table.

```
In [24]: res_all = pd.DataFrame([parameters_3,
                                 parameters_1,
                                 parameters_2],
                                index = ['beta=0.4','beta=0.5','beta=0.7'],
                                columns = ['alpha','rho','nu'])
         res_all.loc[:,'sse'] =[sse_3,sse_1,sse_2]
         print(res_all)
```

```
            alpha       rho        nu       sse
beta=0.4  0.000837  0.084983  0.056854  0.000061
beta=0.5  0.001214  0.098423  0.095040  0.000054
beta=0.7  0.002555  0.097747  0.090170  0.000040
```

As β increase, the $\sigma_B(f,f)$ decrease. To fit the same amount of market volatility, this will leads to larger α, ρ and ν. While at the same time, the error are going down.

3.4

As we can see from the last table, with $\beta = 0.7$ we can have the least error (or target function value) which is 4e-5. So when $\beta = 0.7, \alpha = 0.002555, \rho = 0.097747$ *and* $\nu = 0.09017$ will give as best estimation.

3.5

Now we use the best parameters from last question. and fit the option with 10 year data. Here is the results.

```
In [25]: swap_vol_test = swap_dt.loc[::2,"10Yr"]
         swap_strike_test = swap_dt.loc[1::2,"10Yr"]

         swap_vol_est = sig_B(parameters_2,np.asarray(swap_strike_test)/100,0.7,10)*10000

         res_bonus = pd.DataFrame([swap_vol_est,swap_vol_test])
         res_bonus = res_bonus.T
         res_bonus.columns = ['estimation','market data']
         print(res_bonus)
```

```
    estimation  market data
0    78.349608        72.92
1    77.968290        81.33
2    77.594885        84.40
3    77.229108        85.65
4    76.870686        86.28
5    75.837042        87.02
6    75.180324        87.87
7    74.782137        87.31
8    74.392930        90.62
9    74.239682        85.17
10   74.163569        83.76
11   74.087793        81.19
12   74.012351        79.22
13   74.087793        77.54
14   74.316135        75.96
15   74.703589        73.59
16   75.670529        68.23
17   76.260380        68.60
18   76.782200        58.15
```

```
In [27]: np.sqrt(np.mean((swap_vol_est-swap_vol_test)**2))
```

```
Out[27]: 9.511974867785364
```

The results are presented above and the at mean there will be almost error of 9. The result is not that bad. However what we estimated is not very sensitive to the strike price as the real data does.